BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

POLYMORPHIC CIRCUITS SYNTHESIS AND OPTIMIZATION

SYNTÉZA A OPTIMALIZACE POLYMORFNÍCH OBVODŮ

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. ADAM CRHA**
AUTOR PRÁCE

SUPERVISOR **Doc. Ing. RICHARD RŮŽIČKA, Ph.D. MBA.**
ŠKOLITEL

BRNO 2020

Abstract

This thesis deals with synthesis and optimization methods of polymorphic circuits. Ordinary and multi-functional synthesis and optimization methods are discussed. The main objective of this thesis is to introduce novel methodologies for scalable synthesis of multi-functional digital circuits. Despite the fact that several approaches have been proposed during recent years, those are applicable for small-scale circuits only or are based on various evolution-inspired techniques. Obviously, scalable synthesis methodology for complex multi-functional circuits does not exist yet. The proposed methodology is based on And-Inverter Graphs (AIGs) with built-in extension for multi-functional circuits where the employment of rewriting techniques reduces the area by sharing common resources of two different input circuits. Experiments performed on publicly available benchmark circuits demonstrate significant optimization achievements.

Abstrakt

Tato práce se zabývá metodami logické syntézy a optimalizací pro polymorfní obvody. V práci jsou jak diskutovány existující metody pro konvenční obvody, tak i představeny nové metody, aplikovatelné na polymorfní elektroniku. Hlavním přínosem práce je představení nových metod optimalizace a logické syntézy pro polymorfní obvody. Přesto, že v minulých letech byly představeny metody pro návrh polymorfních obvodů, jsou tyto metody založené na evolučních technikách nebo nejsou dobře škálovatelné. Z toho vyplývá, že stále neexistuje stabilní metodika pro návrh složitějších polymorfních obvodů. Tato práce představuje zejména reprezentaci polymorgních obvodů a metodiku pro jejich návrh založenou na And-Inverter grafech. Na polymorfní obvody reprezentované pomocí AIG je možné aplikovat známé techniky jako například přepisování [rewriting]. Nasazením techniky přepisování na polymorfní AIG získáme obvod, obsahující polymorfní prvky uvnitř obvodu, a je možné dosáhnout značných úspor prostředků, které mohou být sdíleny mezi dvěma funkcemi současně. Ověření návrhové metodiky pro polymorfní obvody bylo provedeno nad sadou veřejně dostupných obvodů, čímž je demonstrována efektivita metodiky.

Keywords

Polymorphic electronics, polymorphic circuit, logic synthesis, logic optimizations, AIG, PAIG.

Klíčová slova

Polymorfní elektronika, polymorfní obvod, syntéza číslicových obvodů, optimalizace číslicových obvodů, AIG, PAIG.

Reference

CRHA, Adam. *Polymorphic circuits synthesis and optimization.* Brno, 2020. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Richard Růžička, Ph.D. MBA.

Rozšířený abstrakt

Rozvoj číslicové techniky v šedesátých letech minulého století otevíral vědě nový, neprozkoumaný prostor. Nová technologie nabízela nové otázky, na které nebyly známé odpovědi a bylo nutné poznávat a pozorovat chování nových materiálů, ze kterých byla vyrobena logická hradla. S rostoucí integrací byly také kladeny požadavky na vhodné nástroje, pomocí kterých se z hradel navrhovaly složitější číslicové obvody. Nyní, po více než 60 letech existence číslicových obvodů, lze předpokládat existenci efektivních návrhových nástrojů, jejichž vývoj se stabilizoval a dnes už nedochází k tak bouřlivému rozvoji, jako v počátcích číslicové techniky.

Ano, toto tvrzení je zcela pravdivé, hovoříme-li o běžných číslicových obvodech. Avšak v roce 2001 představil A. Stoica moderní pojem "polymorfní elektronika", čímž otevřel další nepříliš prozkoumanou vědeckou oblast [106]. Jde o vícefunkční číslicové obvody, u kterých změna funkce není vyvolána přepínačem nebo rekonfigurací, jak je tomu známo u konvenční elektroniky. Namísto toho je změna funkce vyvolána uvnitř číslicového obvodu v závislosti na externím prostředí (teplota, světlo, ...) [105]. Objev polymorfní elektroniky s sebou přinesl nové technologie a otázky týkající se efektivního návrhu polymorfních obvodů.

Materiály, které dříve byly považovány za nestabilní a tudíž nepoužitelné, nacházejí uplatnění právě v polymorfní elektronice. Je možné sledovat značný pokrok ve vývoji grafenu, křemíkových nanotrubiček a organických materiálů [85] [73]. Jedná se tak o velmi mladou vědeckou disciplínu nabízející mnoho disertabilních témat.

Bohužel, konvenční návrhové metody a algoritmy nejsou dobře použitelné pro návrh polymorfních obvodů. Metody syntézy pro návrh polymorfních obvodů jsou mnohem složitější než metody syntézy konvenční elektroniky. Touto problematikou se již zabývalo několik výzkumníku, avšak dosud objevené syntézní metody nejsou natolik efektivní jako metody pro návrh konvenční elektroniky. Tato situace vyžaduje výzkum a vývoj nových, lepších a efektivnějších návrhových metod pro polymorfní obvody. Největší přínos polymorfní elektroniky je spatřován ve sdílení prostředků realizovaných funkcí v co největší možné míře. Je snahou objevovat metody, které budou generovat polymorfní obvody splňující tento předpoklad. Syntézní algoritmy pracují s obvody, nejčastěji reprezentovanými pravdivostní tabulkou, logickým výrazem, či binárním rozhodovacím diagramem. Výstupem by měla být co nejjednodušší reprezentace obvodu.

Cílem této práce je obecně představit polymorfní elektroniku a její otevřené problémy (kap. 4), návrhové techniky konvenčních obvodů (kap. 2) a současné návrhové techniky polymorfních obvodů (kap. 5). Práce představuje tři techniky sloužící k návrhu polymorfních obvodů (kap. 7 a kap. 6).

Jedním z hlavních přínosů práce je představení nové reprezentace polymorfních obvodů PAIG, díky které je možné reprezentovat polymorfní obvody v And-Inverter grafu. Na tuto novou reprezentaci je možné aplikovat již existující optimalizační metody, známé jako strukturální hashování či přepisování [rewriting], ale i další. Právě rewriting byl přizpůsoben tak, aby jej bylo možné spustit na reprezentaci PAIG za účelem optimalizace výsledného polymorfního obvodu a propagace polymorfních prvků do nitra obvodu. Práce prezentuje výsledky vykazující efektivitu metodiky a navrhuje další rozšíření.

Polymorphic circuits synthesis and optimization

Declaration

I declare that this thesis was prepared as an original author's work under the supervision of doc. Richard Růžička Ph.D., MBA. I declare that all relevant information sources used for this thesis are properly cited.

.....
Adam Crha
July 15, 2020

Acknowledgements

I would like to thank doc. Ing. Richard Růžička, Ph.D., MBA for his support and supervision of this thesis. Next, I would like to thank Ing. Václav Šimek for collaboration on publication activities. Big thanks belongs to my wife Tereza, who supported me, although I have been spending evenings with research instead of spending evenings with her. I'm thanking to my parents who supported me during whole Ph.D. study and also to my cousin, Luděk Bryan, who has been motivating me not only during Ph.D. study.

Contents

Chapter 1

Introduction

Reconfigurability as a phenomenon in the world of digital circuits brings more efficient ways to implement certain applications, opens new possibilities and also allows new applications of electronics. As a matter of fact, it makes hardware more flexible. Flexibility is one of the features that make software so popular as a way to implement various systems. But a wide range of applications still needs to be implemented in hardware. So the hardware reconfiguration is (and will be henceforward) very important for significant number of applications.

Typical implementation of the hardware reconfiguration consists of a field of reconfigurable elements, a controller, and memory that serves as a storage for different configurations [10]. The field of reconfigurable elements usually assumes various granularity levels - from coarse-grained elements like functional units or data processing units on RT level to transistor-level fine-grained field of elements. This allows not only the classic reconfiguration scheme (the hardware changes its structure and behavior according to the configurations prepared beforehand), but also effective implementation of so-called evolvable hardware (new configurations are being created as a direct response to actual circumstances) [62].

Another (and quite different) concept of hardware reconfiguration was proposed by Stoica et al. under the term „Polymorphic Electronics“ [106]. In this concept polymorphic circuits have a permanent structure (interconnections are fixed) and each element (or selected group of elements) of the circuit is sensitive to certain environmental factors (temperature, variation of supply voltage, etc.). Then, the function of a polymorphic circuit changes instantaneously in accordance with those specific factors. If these elements are efficiently implemented and the synthesis of the circuit is properly done, the resulting circuit will be highly efficient. Let me also note that due to the multi-functional nature of individual elements, synthesis of polymorphic circuits is much more complex than synthesis of an ordinary digital circuit.

1.1 Research motivation

It is possible to identify two main issues, which still significantly hinder more extensive adoption of polymorphic electronics as a technique for reconfigurable circuits. The first one results from a lack of suitable polymorphic components on all levels of synthesis. As the majority of polymorphic circuits have been designed on a gate level, the most-wanted polymorphic components are naturally polymorphic gates. Several useful polymorphic gates were proposed during the last decade [90] and some prospective sets of multi-functional

gates are emerging even today [77]. The second issue is dealing with multi-functional circuit synthesis using those polymorphic components. As the problem of polymorphic circuit synthesis is relatively hard to address in a conventional way, many of the previously devised polymorphic circuits have been synthesized using evolutionary principles (EA, CGP etc.). Time needed to evolve a result grows dramatically with complexity of a circuit and probability of obtaining reasonable and efficient implementation drops at the same time.

1.2 Thesis organization

Brief introduction to digital circuit design is reviewed in the following section 1.3. The section explains integrated circuits design flow from system specification to physical device. The design flow is demonstrated on a well known Y-chart, where logic synthesis phase takes place. Consequently, the logic synthesis phase is discussed.

Chapter 2 presents existing methods and principles of ordinary logic synthesis and optimization algorithms. Metrics and other terms, that are used further in this thesis, are established. Two-level and multi-level methods are discussed. Subsequently, And-Inverter Graphs, a key scheme of the thesis, is analyzed in detail in chapter 3.

A term polymorphic electronics is first mentioned in chapter 4. The chapter describes evolution of polymorphic electronics and current situation in the field since its introduction in 2001. Principles, ideas and manufacture of polymorphic electronics are discussed, together with application scenarios and open issues. Chapter 5 describes state of the art of currently known synthesis methods for polymorphic circuits.

Next chapter 6 describes proposed two-level synthesis and optimization methods for polymorphic circuits. The first mentioned method is based on boolean algebra optimizations in order to design polymorphic circuit having NAND/NOR polymorphic gates (section 6.1). The second method is suitable for detection of common parts in logic expression and thus finding differences in a desired polymorphic circuit (section 6.2).

The main contribution of this thesis is presented in chapter 7. In the beginning, a novel, multi-level representation for polymorphic circuits is introduced. The innovative representation is an extension of AIG, in order to add capability to handle polymorphic circuits. The chapter continues with a proposal of polymorphic-AIG (PAIG) rewriting of polymorphic circuits. Its aim is to optimize a PAIG network.

Major experiments related to PAIG rewriting are described in chapter 8 in detail. The chapter consists of four sections, where the first (section 8.1) optimizes one desired circuit with polymorphic behavior. The second section (section 8.2) focuses on optimization of two independent circuits in polymorphic mode. The second experiment The third experiment (section 8.3) compares PAIG rewriting that allows KL-cuts to PAIG rewriting which permits K-cuts only. The last one (section 8.4) compares the PAIG rewriting with the most famous synthesis method PolyBDD.

Conclusion, thesis contributions and suggestions for future research are discussed in chapter 9.

1.3 Digital circuits design background

In general, an electronic device is a composition of basic electronic components such as resistors, capacitors, inductors, diodes, and transistors interconnected with wires. Interconnection of mentioned components with wires creates an electronic circuit with an ability

to perform simple or complex operations such as computation, signal amplifying, data transfer, etc. Electronics can be divided into these groups: digital electronics, analog electronics and mixed electronics, which is mix of both previously mentioned [3].

Digital electronics is a subset of electronics, that operates on digital signals. A highlight of digital electronics in comparison to analog electronics is that digital signals can be transmitted without degradation caused by noise. For example, it is possible to reconstruct an audio signal transmitted as a sequence of ones and zeros without any damage, assuming that noise is not strong enough to prevent recognition of the zeros and ones in the sequence [46]

Electrical signals appearing in digital circuits are discrete and represent logic values. These values represent information that is usually further processed. In most of cases, binary logic is applied: One voltage level (typically positive value) represents logical '1', another voltage level (usually zero voltage) represents logical '0'. Digital circuits are built from logic gates (gates are built from transistors usually) and these gates offer functions of boolean algebra, such as AND, NAND, OR, NOR, XOR, XNOR etc. Combination and interconnection of these elementary gates can represent combinatorial digital circuit. [45]

Digital circuits can be divided in two groups: combinatorial and sequential circuits. Outputs of combinatorial digital circuits depend on and only on values attached to circuit inputs. Sequential digital circuits compute an output value based on values attached to circuit inputs and also on internal state of the sequential circuit. It suggests, the sequential circuits are enriched with memory, which can keep an internal state of a sequential circuit. For the purposes of this thesis, only combinatorial circuits will be discussed in the further text.

Nowadays, nearly all the computing machines are internally based on some variant of a digital circuit. From a formal point of view, its composition can be described in a straightforward way through the following definition [92] below, where its depicted as a variant of acyclic graph:

Definition 1. ***Digital circuit***
Let K be a set of functional blocks (e.g. logic gates), and let G is an acyclic graph $G = (V, E)$. Then, a digital circuit is $C = (V, E, \varphi)$, where

- *V is set of nodes (I/O ports of logic gates),*
- *$E = \{(a, b)|a, b \in V\}$ is a set of edges (interconnections),*
- *φ denotes a projection that assigns to each vertex from V a component from the set K, $\varphi : V \to K$.*

The definition 1 describes a structural description of a digital circuit on logical level in Y-chart. The term Y-chart is explained a few paragraphs below. The structural description is used for the purposes of this thesis.

In its most simple valid composition, a digital circuit may consist of a single logic gate or similar fundamental element. It's necessary to point out that in a real situation a circuit would be comprised of potentially high number of mutually interconnected logic blocks (or other substantial parts for its flawless operation).

It involves a term VLSI - Very Large Scale Integration, which means a process of creating an integrated circuit by combining millions of logic gates onto a single integrated circuit. VLSI began in the 1970s when integrated circuits were widely expanded, enabling

the development of complex semiconductor technologies. It is good to mention that microprocessors and memory chips are created by VLSI process. Before the introduction of VLSI technology, most integrated circuits had a limited set of functions they could perform. An electronic circuit might consist of a CPU, ROM, RAM and other glue logic. VLSI lets integrated circuits designers add all of these into one chip.

For development of integrated circuits, a Y-chart (also known as Gajski-Kuhn chart) is mostly used. Y-chart was developed in 1983 by Daniel Gajski and Robert Kuhn. The chart, visible in figure 1.1, represents the hardware development view as three domains that are depicted as three axises (behavioral, structural and physical) and looks like an Y. Along these axises, the abstraction levels describe the degree of abstraction. The outer shells are generalizations, the inner ones refinements of the same subject. Abstraction levels are illustrated in a figure 1.2 and also figure 1.1 presents the mentioned Y-chart. The system level describes the most abstract layer, such as processors or SoC's (System-On-Chip). Register Transfer Level describes digital circuits using registers (e.g. adder), logic level works with gates and circuit level operates with transistors.

Physical axis binds the structure to silicon. It specifies a Printed Circuit Board (PCB) layout or integrated circuit layout [75].

Behavioral axis reflects how a desired circuit should respond to a given input vector. Behavior may be specified by truth tables, Boolean equations, algorithms or any hardware description languages (HDLs) [75].

Structural axis describes how components are interconnected to perform a desired function. This representation uses a list of components and their interconnections [75].

The thesis content can be put to logic level, where gates are representatives of structural axis and Boolean expressions of behavioral axis.

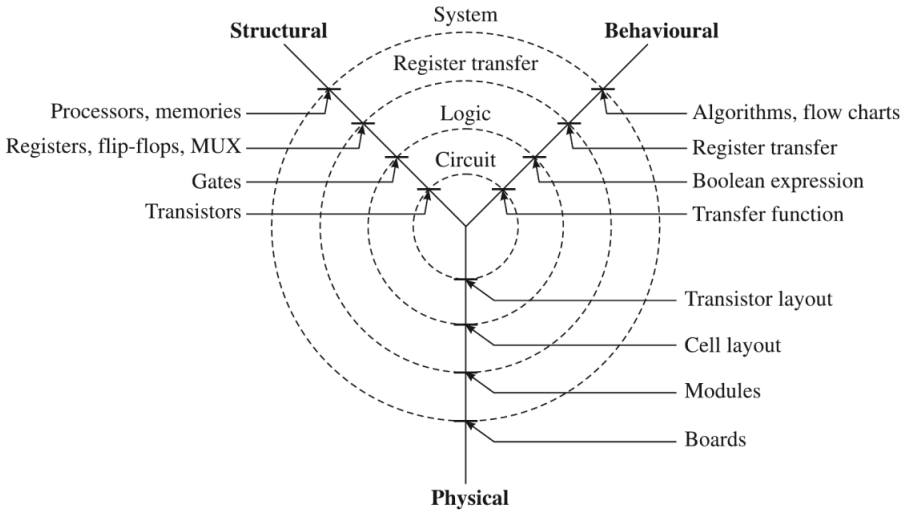

Figure 1.1: Y-chart [75].

1.3.1 Logic synthesis and optimization

On the basis given by Y-chart, a design of digital circuits is following the flow from outer shell inwards of Y-chart. To reach a systematic design of VLSI circuits, IC (Integrated circuit) design flow comes out from the Y-chart. IC design flow uses a limited set of

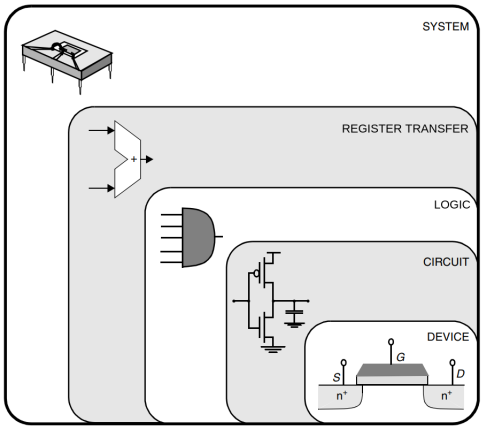

Figure 1.2: Abstraction levels in Y-chart [83].

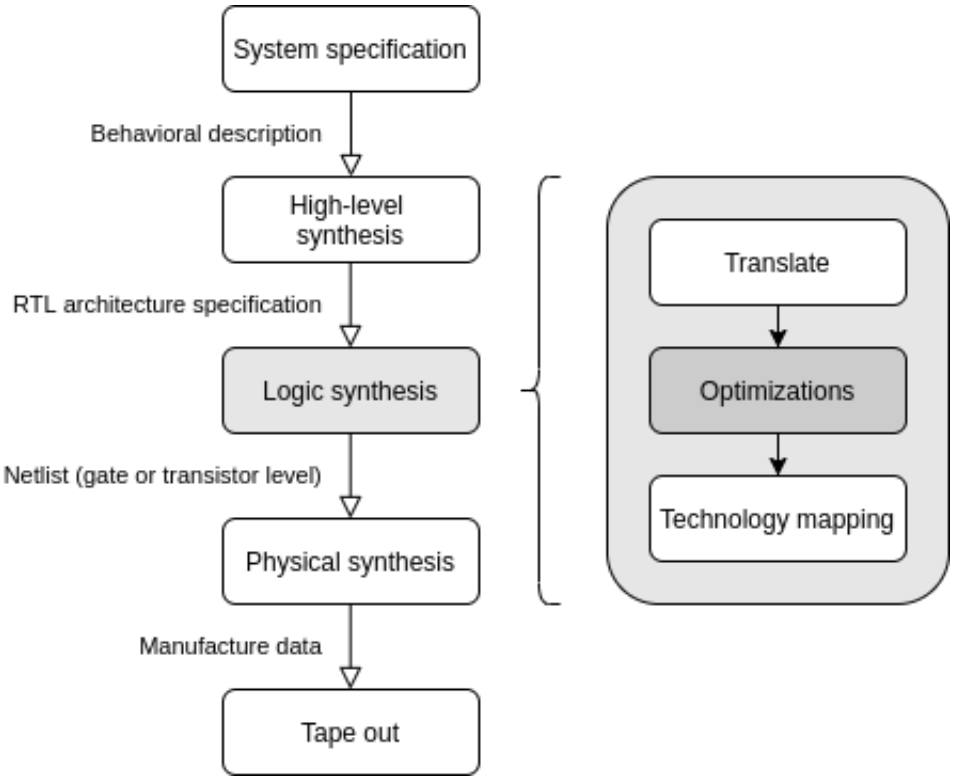

Figure 1.3: Design flow.

digital logic gates (a cell library), and the process can be divided into five parts: *System specification*, *High-Level synthesis*, *Logic synthesis*, *Physical synthesis* and *Tape out*.

- *System specification* simply describes the functional and non-functional requirements posed on a system element.
- *High-Level synthesis* makes a transformation at an architectural level, transforming an algorithmic description into an RTL.

- *Logic synthesis* performs a transformation from a behavioral circuit description into a netlist of logic gates of target technology [61].
- *Physical synthesis* (also known as Low-Level Synthesis) is responsible for transformation of a netlist, obtained from logic synthesis, into a set of geometric shapes and layers to be manufactured.
- *Tape out* is the final result of the design process for integrated circuits or printed circuit boards before they are sent for manufacturing.

A design flow is clearly illustrated in figure 1.3. The figure covers all mentioned abstraction layers with processes, that are applied in digital circuit design. The *logic synthesis* block is highlighted and expanded, because the block is essential for this thesis. Logic synthesis flow can be also divided into at least three parts: Translation, optimization and technology mapping:

- *Translation*
  A Register Transfer circuit description is transferred to input format of synthesis tool like PLA (Programmable Logic Array), BLIF (Berkeley Logic Interchange Format), Aiger, etc.
- *Optimization*
  An input is optimized by a synthesis tool. Synthesis tool produces an optimized result/circuit description.
- *Technology mapping*
  An optimized description is mapped onto target technology, where target technology elements are specified by a mapping library. The result is a *Netlist* on the gate level of target technology.

The term **logic synthesis** is a very important topic in EDA (Electronic Design Automation) area [53]. The logic synthesis is a transformation process of a circuit behavioral description into an optimized gate-level representation, i.e. a netlist of gates for a target technology. Main goals of logic synthesis are optimizations, typically area, delay and power optimizations, where these steps are common for two-level and multi-level representations and also for ordinary and multi-functional circuits. The logic synthesis methods attempt to minimize the number of required components, power consumption and delay of signal delivery.

Chapter 2

Overview of optimization methods

This chapter deals with design and optimization of ordinary combinatorial digital circuits, because the main thesis contribution is logic synthesis and optimization of polymorphic circuits. The most known design and optimization methods are discussed in a following text [119].

Optimization phase clings on optimization methods, which can be categorized into three groups, namely algebraic, graphic and algorithmic.

Algebraic methods take advantage of Boolean algebra laws, where a logic formula is optimized by sequential application of axioms and theorems of Boolean algebra.

Graphical methods are usually based on Boolean neighborhood. A logic function is projected graphically, where an engineer can see particular structures, which can be optimized. For example unit cube, Venn's diagrams, Karnaugh's and Svoboda's maps.

The last kind are algorithmic methods. Algorithmic methods are also based on Boolean neighborhood, but in comparison to graphical methods, it is possible to process optimization by computer. Typical algorithmic methods are Quine McCluskey and Espresso.

2.1 Optimization metrics

The categories of optimization methods has been introduced in the previous section. However, it is appropriate to deploy a quality evaluation of optimized expression. Quality of optimized expressions can be evaluated according to various criteria:

1. **Size of a circuit:** Size of a circuit can be measured as a number of logic gates AND, OR, NOT, ... used in a circuit implementation. The most common gates used in industry are two input NAND gates, which are the most widespread in gate fields.

2. **Delay of a circuit:** Propagation delay or gate delay is the length of time interval which starts when an input to a logic gate becomes stable and valid to change, to the moment that the output of that logic gate is stable and valid to change. Circuit delay is a sum of times of all gates one after another. Circuit delay can affect performance and data propagation through circuit.

3. **Number of wires:** Number of wires can have influence on an area requirements of circuit implementation. Each wire has physical dimensions, resistance and delay.

4. **Power:** The last most used criteria is power consumption. Each logic gate requires specific amount of power to work properly. Material and internal construction of logic gates have influences to power consumption of digital circuit.

Some of metrics mentioned above, are technology dependent and thus it is required to have known target technology during optimization. Because polymorphic electronics doesn't have a solid base in technology level, this thesis will use technology independent metrics such as number of gates, that is usable for wide spread of interested parties.

2.2 Description of logic circuits

A logic optimization process is an algorithm. Algorithm is a finite set of instructions to perform a computation on input data. In the case of logic optimizations, an algorithm optimizes a logic circuit given to an algorithm input. Thus, a logic circuit to be optimized (input data) must have an description format that is passed to optimization algorithm and optimization operations are applicable on description well. Such logic circuit can be described in many forms. The first five forms mentions Mr. Wakerly in his publication [119], where it is also possible to find detailed information. The rest of descriptions are mostly used for multi-level representation. The following text lists the most common description of logic functions [119, 42]:

- **Truth table:** For each primary input and primary output exists exactly one column. Input columns reflects state of primary inputs and output colums reflects all of the possible results of the logical operation that the table represents. Each row express one input combination that affects each primary output state.
- **Disjunctive Normal Form:** A logic formula composed of a disjunction of conjuctive clauses. In other words: DNF Boolean function written as a sum of minterms[1] (products). It is also known with these names: Sum of products (SOP) or „OR of AND's".
- **On-set:** List of line numbers of truth table, where function value is logical one. It is a list of truth table rows, where combination of input variables leads to true function value. On-set of f is $\{x|f(x) = 1 = f^{-1}(1) = f^1\}$. Denoted by character $\sum$.
- **Conjuctive Normal Form:** A logic formula composed of a conjunction of disjunctive clauses. In other words: CNF is Boolean function written as a product of maxterms[2] (sums). It is also known with these names: Product of Sums (POS) or „AND of OR's".
- **Off-set:** List of line numbers of truth table, where function value is logical zero. It is a list of truth table rows, where combination of input variables leads to false function value. Off-set of f is $\{x|f(x) = 0 = f^{-1}(0) = f^0\}$. Denoted by character Π.
- **Algerbraic expression or formula:** An algebraic expression is a mathematical notation that is made up of constants and variables [3], where meaningful relationships are created by using algebraic operations (eg, addition, multiplication) and parentheses.

[1] **Minterm**: A minterm is a product of literals. More specifically, if there are n variables, $x_1, x_2, ..., x_n$, a minterm is a product $y_1 y_2 ... y_n$, where y_i is x_i or $\overline{x_i}$.

[2] **Maxterm**: A maxterm is a sum of literals. More specifically, if there are n variables, $x_1, x_2, ..., x_n$, a maxterm is a sum $y_1 + y_2 + ... + y_n$, where y_i is x_i or $\overline{x_i}$. Maxterms are not usually used in practice, it is related to the fact that NAND gates are often used in practice.

[3] **Variable**: Input of Boolean function. Each $k - ary$ Boolean function has k variables $x_1, ..., x_k$

- **BDD - Binary decision diagrams**: Logic function is expressed by directed acyclic graph structures. Explained later in 2.4.2.
- **Boolean networks (AIG, MIG, ...)**: Another variant of directed acyclic graph to represent a logic function. Also explained later in 3.

This list contains the most known representations of logic functions applicable for two-level and also multi-level logic functions. Other representations can exist, just as exist derivatives of listed representations.

2.3 Two-level optimization

Optimization methods can be divided into two groups - two-level optimization and multi-level optimization methods. For two-level circuits are SOP or POS typical representations forms. Widely used implementation (format) for this kind of circuits is *PLA (Programmable Logic Array)*. On the basis of simplicity of the PLA format of a two-level circuits, the optimization tasks are mostly easy to understood. This chapter describes two-level optimization methods of ordinary circuits.

2.3.1 Boolean function representation

A two-level logic function can be described in many forms. The following list states the most used two-level forms (enumerated from listing in section 2.2): *Truth table, Disjunctive Normal Form, Conjunctive Normal Form, On-set, Off-set.*

Two-level methods generate formulas in disjunctive normal form or in conjunctive normal form. In contrast, multi-level optimization methods can handle and generate formulas with deeper immersion, which lead to optimized circuits with a path longer than two gates.

Input of two-level optimization methods is one of the listed representations, i.e. truth table, algebraic/logic formula or any other equal representation. Basic methods use principles of boolean algebra and applying boolean algebra rules as much as possible. For example, such rules are associativity rule, absorption rule, aggressiveness of zero and one, idempotent rule and also deMorgan's rules.

2.3.2 Karnaugh maps

Kaurnaugh map is a graphical method of minimization of Boolean functions. The principal basis is projection of n-dimensional tabular values into two-dimensional map. Then it is possible to extract, by human's pattern recognition capability, a minimal function from the two-dimensional map [49, 115]. This approach also allows identification and elimination of potential race conditions. Input variables, typically taken from a truth table, are ordered with respect to Gray code and inserted into two-dimensional map, where one cell represents exactly one combination of inputs. Cell value represents the corresponding output value. When the map is prepared on the basis of previous description, cells are collected into the largest possible groups containing 2^n cells, where n is a number of variables in a subexpression. Collected group is a cube[4], thus conjuction of variables in a group. Once a set of cubes is found, a logic sum of them produces a minimized boolean expression.

It is very simple method for minimization of Boolean expression up to 4 variables because of complexity of two-dimensional map. For more variables, a map becomes harder to read

[4] **Cube**: A cube is defined as the AND of a set of literal functions (conjunction of literals).

and orient in a map. Thus, Karnaugh maps are still used for optimization of Boolean expressions, but for resolution of less complex problems or subproblems only.

2.3.3 Quine-McCluskey

The Quine–McCluskey algorithm (or the method of prime implicants) is a method used for minimization of Boolean functions that was developed by Willard V. Quine and improved by Edward J. McCluskey [80, 81, 60]. The Quine–McCluskey algorithm uses the same idea as Karnaugh map, however it is based on a tabular representation in contrast to Karnaugh maps, which uses a graphical representation. The tabular representation allows resolution by computer algorithms and it also produce deterministic way to generate minimized Boolean function. The method can be divided into two steps: The first step is finding all prime implicants of the booelan function. The second step is creation of a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function.

Despite the fact that Quine-McCluskey method is more practical for handling with more than four input variables in comparsion to Karnaugh maps, the Quine-McCluskey algorithm has also a limited range of use since the resolution complexity is NP-complete [114]. The resolution time of Quine-McCluskey algorithm grows exponentialy with number of input variables.

2.3.4 Espresso

All previously mentioned methods can be considered as basic and exact methods. Methods produce minimal solutions, but most minimization tasks are too complex to by solved by these exact algorithms. Rather than a search of minimal function, heuristic methods are searching a near minimal solution in acceptable time. Espresso is a basic representative of heuristic methods. The Espresso algorithm follows a completely different approach to minimization than exact methods. It was developed by Brayton et al. at the University of California, Berkeley [13]. In a contrast to previous methods, which expand a logic function into minterms, the algorithm manipulates with „cubes“, representing the product terms in the On-set, DC-set[5] and Off-set covers iteratively. Despite the fact that the optimization result is not guaranteed to be the global minimum, in practice this is very closely approximated, while the solution is always free from redundancy. In comparison to other mentioned methods, this one is essentially more efficient, reducing memory usage and computation time by several orders of magnitude.

Espresso algorithm works in three steps:

- Reduce - Maximally reduce all cubes, so that a cover is retained.
- Expand - Maximally expand all cubes, so that a cover is retained.
- Irredundand cover - remove a reduntant cover. Then, repeat these steps to find alternative reduced implicants.

However, despite the success and good results, Espresso is applicable only to circuits up to 100 inputs or outputs. To this weakness aims BOOM - Boolean minimizer.

[5] **DC-set**: List of truth table rows, where combination of input variables does not matter output function.

2.3.5 BOOM - Boolean minimizer

Boolean minimizer (BOOM), introduced by Petr Fišer in 2001 [34], is a two-level minimization algorithm based on a new implicant generation paradigm. In comparison to all the mentioned minimization methods, where the implicants are generated in bottom-up order, the BOOM algorithm uses a top-down ordering. Thus, a dimension of a term is gradually decreased by adding new literals [6], instead of increasing the dimensionality of implicants by omitting literals from their terms. Boolean Minimizer has an ability to optimize logic functions with thousands variables in reasonable execution time.

2.4 Multi-level optimization

In the real world, computation tasks and requirements for digital circuits are too complex to be covered by two-level circuits. Based on the nature of two-level methods, it is evident that the two-level optimization and synthesis methods are not applicable to complex circuits in a full range. Thus, multi-level optimization and synthesis methods have an irreplaceable position here. Due to complexity of multi-level circuits, truth table or Sum-Of-Products/Products-Of-Sums are not robust enough to describe them, therefore graphical representation and graph operations are most commonly used. The following text and examples are based on these publications [31, 42, 44]. Figure 2.1 shows examples of two-level and multi-level logic circuit.

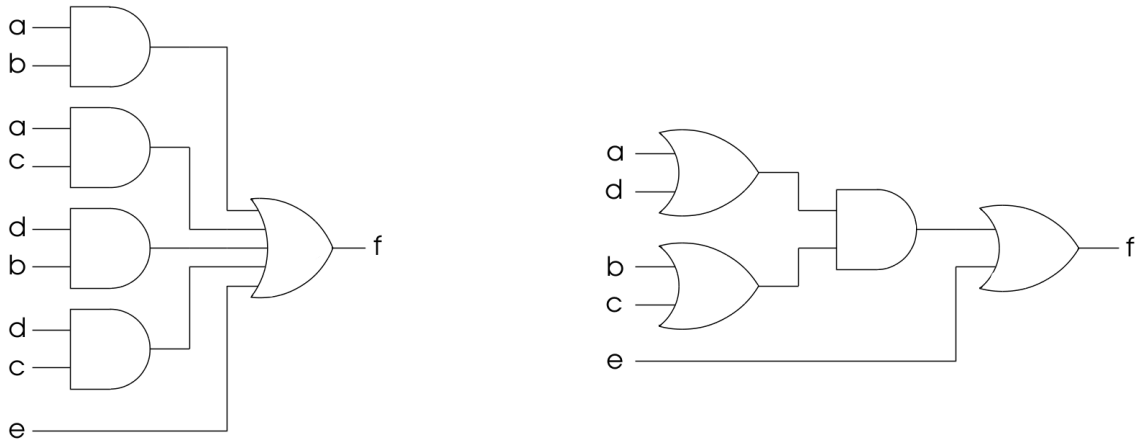

Figure 2.1: On the left is an example of two-level logic circuit. On the right is an example of multi-level logic circuit. Both performs the same logic function.

For multi-level optimization, five basic operations are widely know, such as:

- *Decomposition:* decomposing a complex Boolean function into elementary gates from a given library. i.e. express a single function by a set of functions. Usage of multiple subexpressions is expected and result need not be in factored form.

Example 1. $F = abc + abd + \overline{a}c\overline{d} + \overline{b}\overline{c}\overline{d} \quad \rightarrow \quad \begin{array}{l} F = XY + \overline{XY} \\ X = ab \\ Y = c + d \end{array}$

[6] **Literal**: A literal is a Boolean variable or the complement of a Boolean variable. For example $x_1, \overline{x_1}, x_2, \overline{x_2}$.

- *Extraction:* expressing a set of functions by a set of functions. Extraction extracts common subexpressions of more functions in comparison to decomposition.

Example 2.
$$\begin{array}{lcl} F = acd + bcd + e & & F = XY + e \\ G = a\overline{e} + b\overline{e} & \rightarrow & G = X\overline{e} \\ H = cde & & H = Ye \\ & & X = a + b \\ & & Y = cd \end{array}$$

- *Factorization (serial-parallel decomposition):* Searching factors, that is create a factored form from a SOP.

  Example 3.

  $F = abc + abd + \overline{a}c\overline{d} + \overline{b}\overline{c}\overline{d} \quad \rightarrow \quad F = ab(c + d) + \overline{c}\overline{d}(\overline{a} + \overline{b})$

- *Substitution*: Express a function using another function.

  Example 4.

$$\begin{array}{lcl} F = ac + ad + b\overline{c}\overline{d} & & F = Ga + \overline{G}b \\ G = c + d & \rightarrow & G = c + d \end{array}$$

- *Collapse (eliminate):* expressing a function without using another function (opposite of substitution). In other words, express a function using primary inputs only or express a network using a SOP form.

  Example 5.

$$\begin{array}{lcl} F = Ga + \overline{G}b & & F = ac + ad + b\overline{c}\overline{d} \\ G = c + d & \rightarrow & G = c + d \end{array}$$

2.4.1 Boolean division

Previously mentioned operations are a basis for algorithmic optimization. However, the question is how to do all the mentioned operations? Questions „How to find common factors?“, „What factor choose? Which factors are the best?“ or „How to find shared expressions?“ are in right place. As a reader can observe from examples, it is all about searching common subexpressions and it may seem like division by a common divisor. Unfortunately, *„Boolean division“* doesn't exists in Boolean algebra. Nevertheless, it is possible to factor a function into three parts:

$$f = p * q + r$$

where p is a Boolean divisor, q is a quotient and r is a remainder. Thanks to this equation, it is possible to perform boolean division of function f by a divisor p. Thus, the aim is to find functions (quotient) q and remainder r.

In the previous paragraph it has been mentioned that it is possible to perform boolean division of a given function f and a divisor p, but how to find the best divisor p? The answer is: *kerneling*.

Kerneling is a methodology of looking for common subexpressions (divisors) of a given function f. Kernels are divisors p for division of an expression f. Such kernel is a cube-free primary divisor. A cube-free expression is an expression without factors:

Example 6.
$ab + c$ is cube free.
$ab + ac$ is not cube free (a is a divisor).
abc is not cube free (a, b, c are a divisors).

Kerneling plays a key role in all mentioned operations (decomposition, factoring, extraction, substitution), where the best kernel intersections can be used as divisors for boolean division. It is also important to notice that the kerneling and Boolean division are not the only existing methods and other decomposition (Boolean) methods already exist, such as Ashenhurst [8], Curtis [30], Roth and Karp [88], Steinbach [70] or Karplus [50] and others [52, 84], but description of these boolean decomposition methods exceeds the content of this thesis.

For a representation of complex boolean functions, logic formulas are not robust enough and we need something to represent a complex logic network. The following text makes an introduction of existing, well known, multi-level representation and synthesis methods of conventional logic synthesis.

2.4.2 Binary Decision Diagrams

The main idea of Binary decision diagrams (BDDs) had grown from Shannon decomposition, that can be used to split a Boolean function into two simpler subfunctions until the constant functions 0 and 1 are obtained [32].

BDD is a data structure for representation of a Boolean functions introduced by Lee in 1959 [54, 32], where a function is represented by directed acyclic graph (DAG) composed of terminal and decision nodes. BDD can be formally defined as follows 2:

Definition 2. *Let $G = (V, E)$ is a directed, acyclic graph, where V is a set of nodes and E is set of oriented edges (see definition 1). Then, $G_{BDD} = (V, E, X)$ is Binary Decision Diagram over set of input variables $X = \{x_1, \ldots, x_n\}$, if:*

- *G_{BDD} has exactly one root $v \in V$,*
- *each terminal node in V is labeled with a value from $\{0, 1\}$,*
- *each non-terminal node is labeled with a variable $x_i \in X_n$ and has exactly two outgoing edges, whose ends are denoted by $lhs(v)$ resp. $rhs(v)$ $(lhs(v), rhs(v) \in V)$. lhs means „left hand side“ and rhs means „right hand side“.*

Thanks to graph representation of BDDs, many graph operations are applicable to them. The most of operations are feasible in polynomial time $\mathcal{O}(n^k)$ with respect to input variables, such as conjunction, disjunction and negation.

2.4.3 Ordered Binary Decision Diagram (OBDD)

In order to express functions by BDDs more effectively, a few compact variants exist. The first one is Ordered Binary Decision Diagram (OBDD). OBDD has ordered input variables, i.e input variables appear in the same order on all paths from the root node. It is possible to see complete ordered binary decision diagram in figure 2.2 on the left.

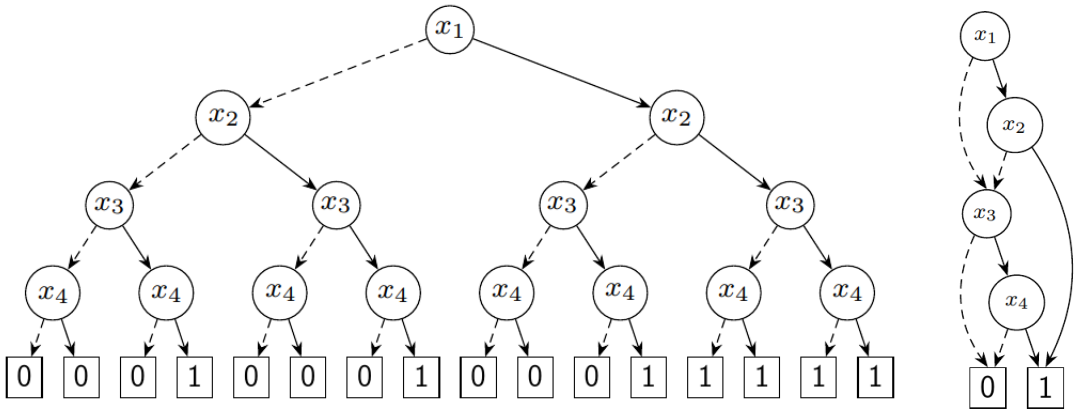

Figure 2.2: (OBDD) Complete and ordered binary decision diagram on the left. (ROBDD) Reduced ordered binary decision diagram on the right. Both graphs represents the same function.

2.4.4 Reduced Ordered Binary Decision Diagram (ROBDD)

Reduced Ordered Binary Decision Diagram (ROBDD) is another kind of BDD. Reduced BDD must fulfill two rules:

- All isomorphic subgraphs are merged.
- All nodes, whose children are isomorphic, are eliminated.

The main advantage of ROBDDs is canonicity[7]. This property enables usage in functional equivalence checking and functional technology mapping [14]. ROBDDs are the most commonly used BDDs and if BDD is mentioned, in the most cases ROBDD is meant. ROBDD is shown in figure 2.2 on the right.

2.4.5 Multi-terminal Binary Decision Diagram (MTBDD)

The last kind of BDD, that should be mentioned is Multi-terminal Binary Decision Diagram (MTBDD). Instead of logic zeros and logic ones, terminals can keep other integer values - multivalued logic.

2.4.6 And-Inverter graphs

And-inverter graphs (AIGs) are essential for this thesis and therefore AIGs are described in the next chapter 3 in detail.

2.4.7 Majority-Inverter graphs

Majority-Inverter Graphs (MIGs) are a relatively novel logic representation structure for efficient optimization of Boolean functions. MIGs were introduced by Luca Amarú in 2014 [5] and can be considered the most effective representation of logic functions today (see definition 4).

[7] **Canonical representation**: Representation is unique for a particular function and variable order.

MIG is a directed acyclic graph (DAG) composed of three-input majority nodes and regular/inverted edges.

Definition 3. *Majority operator is a function $M(a, b, c)$, where a, b, c are input Boolean variables. When a sum of positive logic values is higher than a sum of negative logic values, operator returns true, otherwise false.*

Definition 4. *MIG is a homogeneous logic network with indegree equal to 3 and with each node representing the majority function (operator). In MIG, edges are marked by a regular or complemented attribute [5].*

MIGs can be compared with And/Or Inverter Graphs (AOIGs), while MIGs offer a more compact representation for logic. Each AIOG network (optimized network included) can be expressed by MIG network, whereas MIG network can be further optimized. Conversion from AIOGs to MIGs follows from *Majority operator*.

Based on definition 3, if $c = 0$, majority operator performs $AND(a, b)$ operator, while $c = 1$ majority operator behaves as $OR(a, b)$ operator. It supports the claim that MIGs can express an arbitrary AIOG, OIG or AIG network: $MIGs \supset AIOGs \supset AIGs$, which implies that MIGs are the most universal representation form of all enumerated representations. See figure 2.3 for an example of AIOG conversion into MIG and further MIG optimization. For complete definitions and proofs, please see [5].

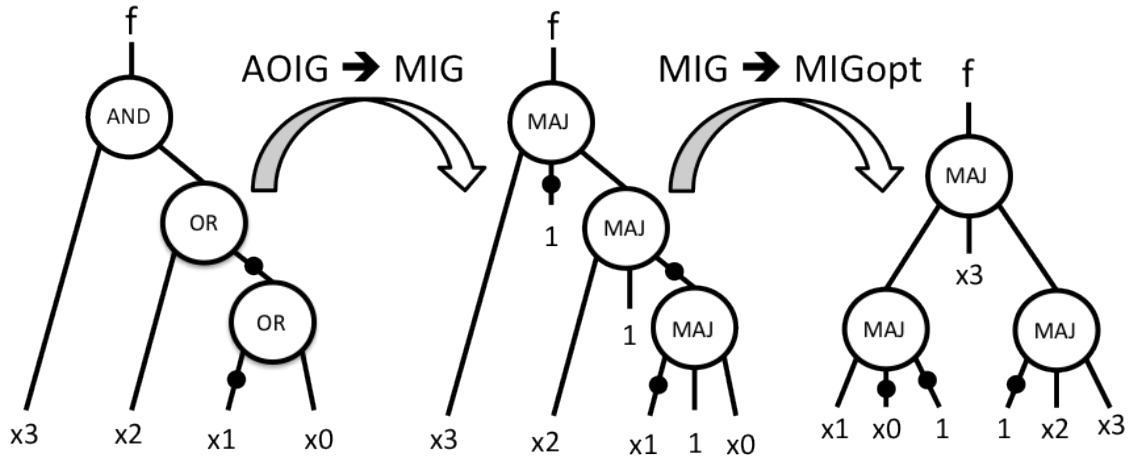

Figure 2.3: Example of conversion AOIG network into MIG and further MIG optimization [4].

2.4.8 MIG Boolean algebra

In order to support natural manipulation with MIGs, a new Boolean algebra has been introduced, based exclusively on majority and inverter operations, with a complete axiomatic system. Set of five transformation rules on Majority operator are enumerated [5]:

1. Commutativity - $\Omega.C$
   $M(x, y, z) = M(y, x, z) = M(z, y, x)$
2. Majority - $\Omega.M$
   if$(x = y)$: $M(x, y, z) = x = y$
   if$(x = y')$: $M(x, y, z) = z$

3. Associativity - $\Omega.A$
   $M(x, u, M(y, u, z)) = M(z, u, M(y, u, x))$

4. Distributivity - $\Omega.D$
   $M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$

5. Inverter propagation - $\Omega.I$ $M'(x, y, z) = M(x', y', z')$

These five primitive transformation are fully sufficient for transformation of any MIG α into any other logically equivalent MIG β, by sequential application. However, the length of transformation sequence may not be practical for modern computers. To avoid this problem, authors derived another three powerful transformations Ψ from Ω:

1. Relevance, replaces and simplifies reconvergent variables - $\Psi.R$
   $M(x, y, z) = M(x, y, z_{x/y'})$

2. Complementary Associativity, deals with variables appearing in both polarities - $\Psi.C$
   $M(x, u, M(y, u', z)) = M(x, u, M(y, x, z))$

3. Substitution, extends variable replacement also in the non-reconvergent case - $\Psi.S$
   $M(x, y, z) =$
   $M(v, M(v', M_{v/u}(x, y, z), u), M(v', M_{v/u'}(x, y, z), u'))$

2.4.9 MIG optimization

The optimization of an MIG, representing a logic function, is a transformation process into different, functionally equivalent MIG, having better quality terms such as size, delay or power consumption. By application of transformation rules from Ω, Ψ to an input MIG, it is possible to reduce size, optimize delay or decrease switching activity of an optimized MIG [5].

Another optimization method on MIGs, introduced in [101], consists in functional hashing of MIGs nodes, which is very similar to DAG-aware AIG rewriting (discussed in section 3.1.6) [68], but deployed on MIGs. This approach aims to size reduction of an initial MIG and experiments report remarkable results.

Chapter 3

And-Inverter Graphs

The most popular representation of digital circuits structural implementation in the last fifteen years are And-Inverter Graphs (AIGs) definitely. AIGs have found their application primarily in logic synthesis and optimizations. However, the first mention about AIGs we can find in an Alan Turing's paper [48] on neural networks, where he has been writing about randomized trainable network of NAND gates. Since the publication, AIGs had felt into oblivion, a few local transformations have been introduced. AIGs and their local transformations have began to appear in several logic synthesis and verification systems in 1980s, with the aim to reduce circuit area and delay during synthesis process or accelerate formal equivalence checking process [31, 100]. A big progress has been done in IBM by discovering important AIG properties, such as structural hashing. BDDs have been very popular for logic synthesis application in 1990s and thus AIGs have started rising up again around year 2000, when BDDs have reached their scalability boundaries in many of their applications. Interest in AIGs was resumed when AIGs have began to be used as a functional representation for a variety of tasks in synthesis and verification. It has been discovered, that when AIGs are used for circuit representation, a significant acceleration is observed in solving a wide variety of boolean problems.

An And-Inverter Graph is a directed acyclic graph for structural representation of Boolean logic circuit. AIG is composed only of two input AND gates and two kind of edges: wire and inverter. Terminals nodes are primary inputs and roots represent primary outputs. AIGs can represent an arbitrary logic function and offers very efficient manipulation of these functions. The following definition 5 supports a formal background of AIGs and figure 3.1 shows an example of AIG network before and after optimization.

Definition 5. *Let G is a directed acyclic graph $G = (V, E)$. AIG $A = (V, E, X, f)$ is an extension of graph G, where*

- *$V = X \cup N \cup \{0\}$ is a finite set of nodes, where $X = \{x_1, ..., x_n\}$ are primary inputs, $N = \{n_1, ..., n_k\} = V \setminus (X \cup \{0\})$ are non-terminal nodes representing the logic AND operator and 0 is the constant 0 input,*
- *$E = \{(a, b) | a \in N, b \in V\}$ is a set of edges (interconnections), so that every node $a \in N$ has exactly two outgoing edges.*
- *$f : E \to T$, where $T = \{wire, inverter\}$, is a function that specifies, whether the edge is a wire or an inverter.*

Figure 3.1 represents a convention how AIGs will be shown in this thesis. Each primary input is represented by a triangle and a square at the bottom of the picture, where squares

denote input latch registers. Rounded nodes represent a logic AND gates with 2-indegree numbered with even numbers. Even numbering reflects AIGER format and internal implementation described further in the text 7.1.2. Interconnection between nodes are solid or dotted arrows. Solid arrow represent regular wire and dotted arrows represent inverter in the connection[1]. Primary outputs are recognizable as triangles at the top of the network.

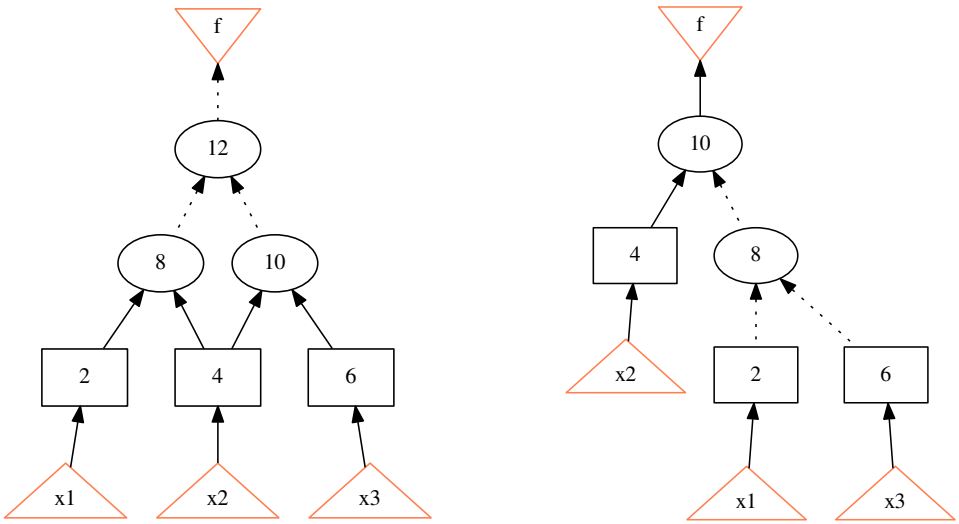

Figure 3.1: Example of AIG network. The network on the left represents function $f(x_1, x_2, x_3) = x_1x_2 + x_2x_3$. The network on the right represent a functionally equivalent, but structurally different function $f(x_1, x_2, x_3) = x_2(x_1 + x_3)$.

A transformation of an ordinary network, composed of logic gates, into AIG requires only a conversion table of each logic gate, expressed in terms of AND gates and inverters or application of DeMorgan's rules. Simply, to derivate an AIG, factor SOPs of the nodes in a logic network, then AND/OR factored forms convert to 2-indegree AND nodes and inverters by applying DeMorgan's rules. Thus, the transformation is scalable, fast and doesn't lead to unpredictable memory and runtime blowup. It makes the AIG an efficient representation compared to SOP (Sum-Of-Product) form or BDDs (Binary Decision Diagrams) representation (see section 2.4.2). SOP or BDD representations can also be handled as logic circuits, but they impose somewhat artificial constraints, which often deprive them of scalability. SOPs are circuits with at most two-levels and BDDs are canonical, it means, they require that input variables were evaluated in the same order on all paths [64].

AIGs can also represent sequential logic and sequential transformations, using D-flip-flops with an initial state. However, the thesis doesn't aim to the sequential circuits, but it can be considered as a future work.

AIG ordinary concept is already implemented in ABC tool [1], which is an academic synthesis tool, developed by Alan Mishchenko at the University of Berkeley. ABC is completely based on AIGs containing AIG-based synthesis and equivalence checking techniques.

[1]Graphical representation of edges differs from commonly used notation. Arrows lead from leafs to roots.

3.1 AIG Optimizations

The optimization of an AIG is a transformation of AIG α into different AIG β, that is functionally equivalent with α, having better parameter terms, such as area, depth or switching activity. For all mentioned items exist particular optimization techniques, and further research continues.

Logic optimization methods, except graphical methods, may be seen as two groups: Boolean methods and algebraic methods. Algebraic methods handle Boolean function as polynomial expression and thus these methods has limited set of allowed operations *(associativity, commutativity and identity)* in comparison to Boolean methods *(associativity, commutativity, identity, annihilation, distributivity, idempotence, absorption, complementation, De Morgan rule and double negation)*, which are based on Boolean algebra. It implies, that Boolean methods are more efficient (better optimization results) in comparison to algebraic methods, but they also require more computation time. In order to apply the Boolean methods onto large logic networks, a known practice is to make a Boolean transformation only in a particular subgraph of a large logic network at a time. This approach, namely local optimization, is essential for scalability of many optimization algorithms.

One of the subgraph selection methods is *Windowing*, introduced in [72, 65], which is applicable for local optimizations. The main aim of the windowing algorithm is to collect nodes around a node n.

3.1.1 AIG Cuts

Another methodology handling subgraphs, called cuts, is used in the most AIG optimization techniques and they are also necessary for this work. Because cuts have been already published many times, the following text is a survey of [68, 66, 79, 101] with adaptation to the thesis purposes.

A cut C of a node n is a set of nodes of the graph, called leaves of the cut, such that each path from a *primary input* to n passes through at least one leaf. Node n is called the root of cut C. The cut *size* is the number of its leaves [68, 66]. See formal definition 6.

Definition 6. ***Cut***
For a given AIG $A = (V, E, X)$, a pair (v, L) consisting of a root $v \in V$ and leafs $L \subseteq V \backslash \{0\}$ is called a cut (see [79, 66, 68]), if

1. *every path from v to a terminal visit at least one leaf $l \in L$,*
2. *each leaf is contained in at least one path from v,*
3. *paths to the constant node are exempt from (1) and (2) constraints,*

A set of all cuts of node v is denoted cuts(v). In other words, $cuts(v) = \{L | L \subseteq V$ $\{0\}$ so that (v, L) is a cut$\}$.

A trivial cut, let us say a „seed“ can be defined as follows, definition 7:

Definition 7. ***Trivial cut***
A trivial cut C of a node v is composed of the node itself only and constant only.
$\{\{\}\}$ *for constant,*
$\{\{v\}\}$, *for $v \in V$.*

In order to deduce a set of all possible cuts of node v, we can define a k-feasibility, definition 8:

Definition 8. ***k-Feasible cut***
A cut is called k-feasible if $|L| \leq k$*. A set of all k-feasible cuts is denote* $cuts_k(v)$*. In other words* $cuts_k(v) = \{L|(v, L)$ *is a cut and* $|L| \leq k\}$*.*

Example 1. Let $C = (12, \{1, 4, 7, 9\})$ be a cut with a root 12 and leaves $1, 4, 7$ and 9. As we can see, this cut is 4-feasible, i.e. $\{1, 4, 7, 9\} \in cuts_4(12)$.

All k-feasible cuts can be generated by a recursive algorithm. For cut enumeration purposes, an operation $\otimes$ is defined, see definition 9:

Definition 9. ***Cut construction operation***
Let A_1 *and* A_2 *be two sets of cuts of a common root* v *and* k *a maximum number of leafs of new cut. Then* $A_1 \otimes A_2 \equiv \{a_1 \cup a_2 | a_1 \in A_1, a_2 \in A_2, |a_1 \cup a_2| \leq k\}$*.*

Now, it is possible to set a set of k-feasible cuts of node v:

Lemma 1. ***Recursive cut enumeration***
$cuts_k(v) = cuts_k(v_{-1}) \otimes cuts_k(v_{-2})$*, where* $v \in V$ *and* v_{-1}, v_{-2} *are direct children of node* v*, for* $\forall v = \{0..n\}$*.*

Based on the previous definitions, it is possible to demonstrate an example of 3-feasible cuts enumeration in figure 3.2.

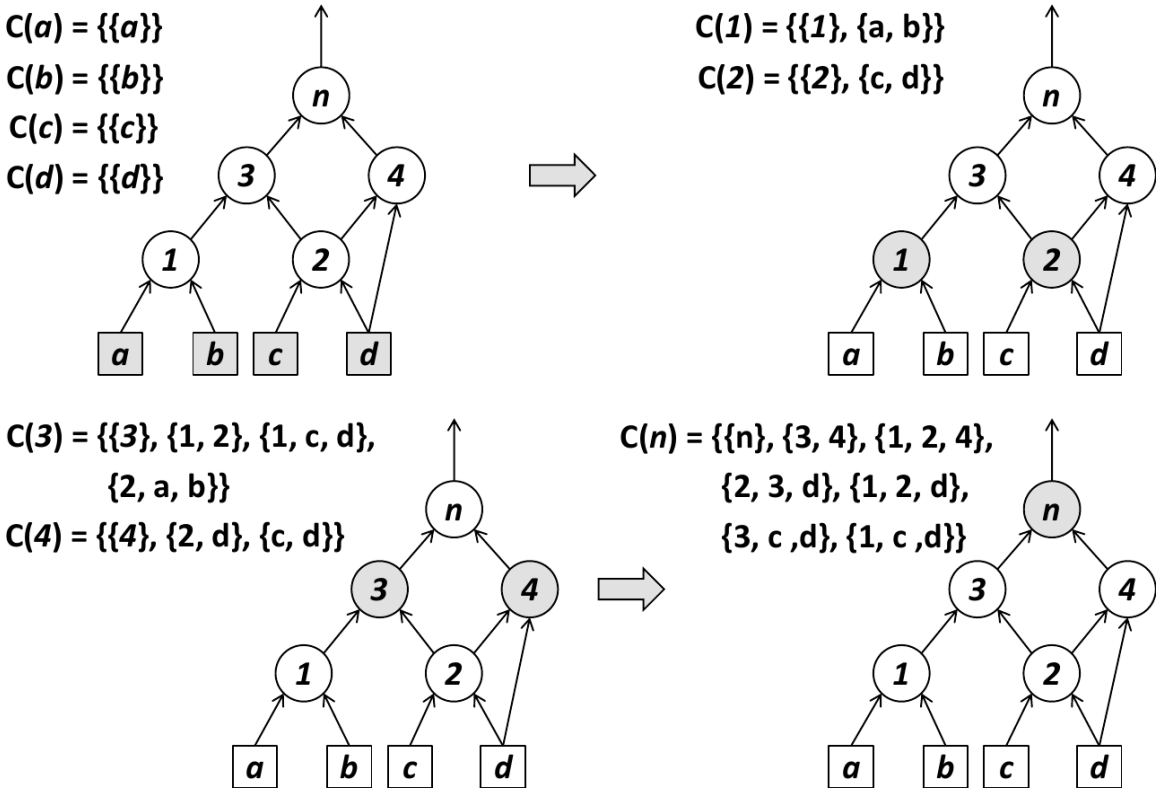

Figure 3.2: Example of 3-feasible cuts enumeration.

KL-Cuts

KL-cut can be defined as a subgraph G_{kl} of G, having $inputs \leq k$ and also $outputs \leq l$. Inputs and outputs are represented by two sets of nodes (G_k, G_l), where G_k is a set of

inputs and G_l is a set of outputs. If a node v belongs to a path between $n_k \in G_k$ and $n_l \in G_l$ and $v \notin G_k$, then v is in G_{kl}. All nodes in G_l are contained in G_{kl} and G_{kl} doesn't contain any node of G_k [59].

Differences between K and KL-cuts are following. Based on the definition in the previous paragraph, k-cut is a sub-circuit of a circuit which has exactly k primary inputs and *one* primary output. KL-cut is a sub-circuit of circuit which has k primary inputs and l primary outputs. See figure 3.3 for examples of K-cut and KL-cut.

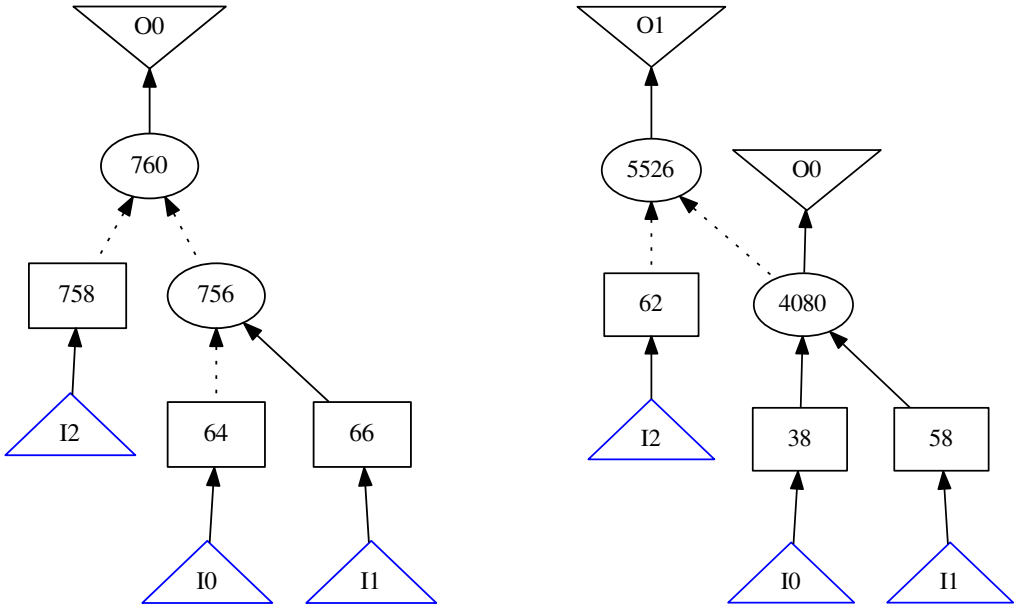

Figure 3.3: Example of k=3 cut is on the left and situation for k=3 in case of multi-output cut with two outputs is shown on the right.

Cut factorization

Enumeration of limited size cuts is a significant step in logic synthesis methods, such as technology mapping or rewriting. The cut enumeration algorithm isn't capable to enumerate all cuts beyond 7 inputs, because there are too many of them. Fortunately, researchers from University of Berkeley introduced *cut factorization* method [15], so that they can factor cuts of a network and use them to generate other cuts. With the factorization method, new terms had been presented (definitions 10 and 11):

Definition 10. ***DAG cut***
DAG cut/Factor cut is a cut of a node v in an AIG, which has two or more outputs from the node v.

Definition 11. ***Tree cut***
Tree cut is a cut of node v in an AIG, which has only one output of node v.

A reader can see an example of cut factorization on figure 3.4, that shows a decomposition of an AIG into factor trees. DAG/factor nodes are gray (see more than one output).

Using the cut factorization, it is possible to reduce number of cuts significantly and thus reduce time for enumeration. For more details and experiments, see literature [15].

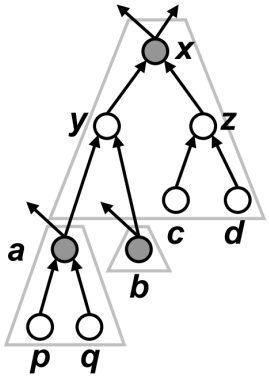

Figure 3.4: An AIG subgraph illustrating cut factorization. Nodes p, q, b, c and d are PIs [15].

NPN Classification

Cuts are a good technique for subgraph selection in DAGs, the number of all functions growths exponentially size as the number of inputs increase. The number of all functions can be calculated as $2^{2^{num\_inputs}}$. For example, if we are looking for all 4-input functions, the real number of all function is 65536. For 5-input cuts, the count of all combinations is 4294967296 [18].

In order to reduce the number of all combinations, NPN[2] classes were established. NPN classes allows to merge some circuits into equivalent one by permutation of inputs, negations or output negations.

- P-class: n-input functions equivalent under input permutations.
- NP-class: n-input functions equivalent under input permutations and negations.
- NPN-class: n-input functions equivalent under input permutations/negations and output negations.

The table 3.1 shows numbers of required functions merged into equivalent classes.

Table 3.1: NPN classes for numbers of inputs 1 - 5.

Inputs	Functions	P-classes	NPN-classes
1	4	4	2
2	16	12	4
3	256	80	14
4	65536	3984	222
5	4294967296	37333248	616126

[2] NPN - **N**egation of inputs, **P**ermutation of inputs, **N**egation of outputs.

3.1.2 Structural hashing

The first optimization technique is not a technique at all, but rather a property. AIGs are usually implemented as a hash table in software. Hash table key of a node is made of two previous nodes. If any new node should have the same predecessors as an existing node, the keys will match and new node is merged with existing one. Thus, structural hashing detects isomorphic subgraphs in a AIG. It leads to the fact, that the two nodes can't be redundant in an examined AIG.

3.1.3 Balancing

Reducing delay of a digital circuit is a significant topic in logic synthesis and therefore it has long history since early days of logic design.

Balancing [19, 66] is a technique widely known from graph theory and it is also applicable on AIGs. This technique is used for reducing depth of a network - reducing delay of a circuit, which is an important topic in logic synthesis.

Balancing is an algebraic tree-height reduction, performed by application of Boolean rules such as commutativity, distributivity and associativity, for example: $a(bc) = ab(c)$. Balancing operation has linear time complexity $\Omega(n)$ and it is frequently used to minimize logic depth. Figure 3.5 shows an example of balancing technique.

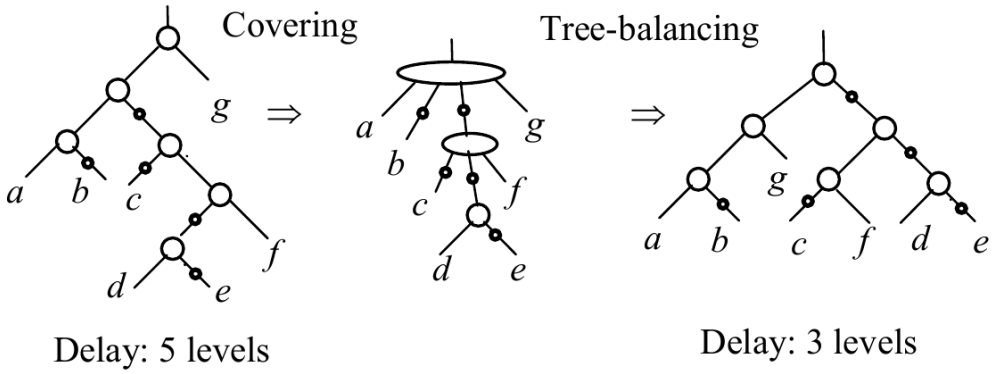

Figure 3.5: An example of AIG balancing transformation [66].

3.1.4 Refactoring

An optimization algorithm, mainly used for network area optimization, is refactoring. Refactoring uses a heuristic algorithm [67] to compute one large cut for each AIG node. Then, the refactoring attempts to replace cut of the current node by factored form. If there is an improvement, the modification is accepted [68].

3.1.5 Resubstitution

A resubstitution is another optimization method, also based on enumeration of one large cut for each node. It tries to re-express a Boolean function of a node by reusing other nodes present in a network, known as divisors (see section 2.4.1). Similarly to refactoring, if a smaller network is obtained, the transformation is approved [67, 51].

3.1.6 Rewriting

Rewriting is another important optimization method for reduction the network size. It is fast greedy algorithm for optimization the AIGs by iteratively selecting AIG subgraphs - cuts (see section 3.1.1) rooted at a node and replacing them with smaller, functionally equivalent, pre-computed subgraphs. Original algorithm uses typically 4-feasible cuts, that are enumerated using the cut enumeration algorithm (see definition 9) [79]. For each cut of each node n, the Boolean function is computed and its NPN-class (see section 3.1.1) is recognized by hash-table lookup. Truth tables, stored as 16-bit strings, are used for fast handling of 4-input optimum functions. If we recall an idea of NPN classes, it is required to have only 222 variants of optimized subgraphs. It has been experimentally found that approximately 100 of 222 optimum circuits are appearing in the rewriting process and only 40 of them are meaningful, in terms of network improvement [68].

```
Rewriting( network AIG, hash table PrecomputedStructures, bool UseZeroCost )
{
    for each node N in the AIG in the topological order {
        for each 4-input cut C of node N computed using cut enumeration {
            F = Boolean function of N in terms of the leaves of C
            PossibleStructures = HashTableLookup( PrecomputedStructures, F );
            // find the best logic structure for rewriting
            BestS = NULL; BestGain = -1;
            for each structure S in PossibleStructures {
                NodesSaved = DereferenceNode( AIG, N );
                NodesAdded = ReferenceNode( AIG, S );
                Gain = NodesSaved – NodesAdded;
                Dereference( AIG, S );  Reference( AIG, N );
                if ( Gain > 0 || (Gain = 0 && UseZeroCost) )
                    if (  BestS = NULL ||  BestGain < Gain )
                        BestS = S; BestGain = Gain;
            }
            if ( BestS == NULL ) continue;
            // use the best logic structure to update the netlist
            NodesSaved = DereferenceNode( AIG, N );
            NodesAdded = ReferenceNode( AIG, S );
            assert( BestGain = NodesSaved – NodesAdded );
        }
    }
}
```

Figure 3.6: Rewriting algorithm [68].

Figure 3.6 describes the AIG rewriting process, where the nodes are investigated in bottom-top topological order. Then, for each 4-feasible cut C of a node N, it obtains a boolean function F. Using hash-table lookup, it finds all possible structures. Subsequently, for each possible structure try: dereference old subgraph and the number of nodes, whose reference counts became 0, is returned. These nodes will be removed if the old subgraph is replaced. Then, a possible structure is added while counting the number of new nodes and the nodes whose reference count went from 0 to a positive value. These nodes will be added. The difference of the counters is the gain in the number of nodes if the replacement is done. The new node is de-referenced and the old node is referenced to return the AIG to its original state. While trying all possible structures, the maximum gain is remembered. The

structure with maximum gain is then applied. If the improvement is zero and „zero-cost“ is enabled, the possible structure is replaced [68].

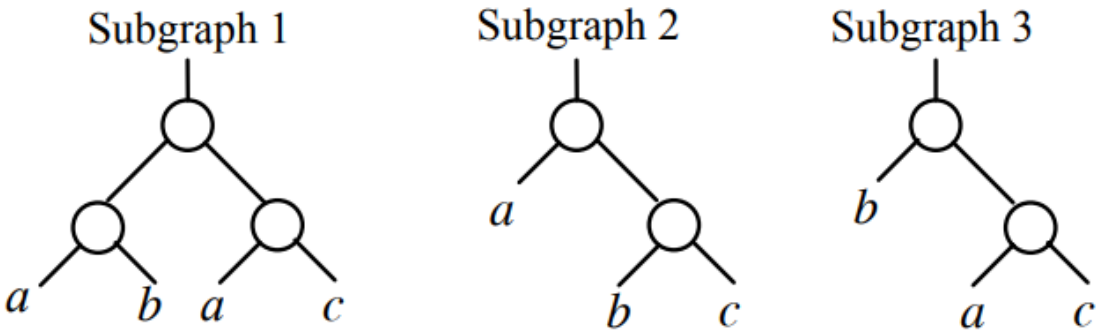

Figure 3.7: Different AIG structures for function $F = abc$ [68].

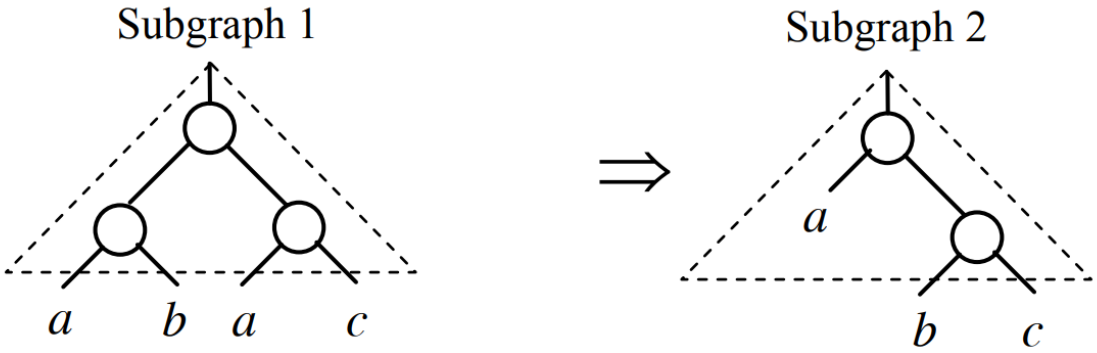

Figure 3.8: An example of rewrite procedure [68].

Figure 3.7 illustrates three possible structures of function $F = abc$. Figure 3.8 shows a replacement of subgraph 1 by subgraph 2, where it is possible to see one node reduction.

Rewriting algorithm leads to significant area improvements in logic synthesis and gave a lot of opportunities for further research.

Further work on AIG rewriting

The AIG rewriting has prepared a stable basis for local transformations of logic optimization. Rewriting is implemented in ABC tool [1] with opened sources, which allows source code modification by other researchers.

In 2007, authors of combinatorial AIG rewriting algorithm extended the algorithm for sequential circuits synthesis and introduced a new term: HAIG (History And-Inverter Graph) [12].

In 2011, Nan Li and Elena Dubrova proposed an extension of combinatorial rewriting. Researchers have tried to enable 5-input cuts in the original rewriting. However, as you can see in section 3.1.1, the number of functions, belonging to NPN equivalence classes, are 616162, that is still a high number. They experimentally checked which 5-input functions are appearing in all IWLS 2005 benchmarks [17] (2749 classes). After that, function (classes) with higher than twenty occurrences had been picked up (1185 classes). All these functions

(classes) had been precomputed and applied for 5-input cut rewriting. Researches achieved about 5.57% further reduction of area of heavily optimized large circuits on average [55].

A next interesting work on AIG rewriting has been taken by Ivo Háleček and Petr Fišer. They extended an ordinary AIG to XAIG (XOR-AND-Inverter-Graph) with adaptation of rewriting algorithm to XAIGs. This modification allows to work with XOR gates in a native way. Experimental results indicate that the proposed methodology is stronger in XOR identification than XOR-aware structural hashing already implemented in ABC [43, 47]. Figure 3.9 shows valid rewrite on XAIG.

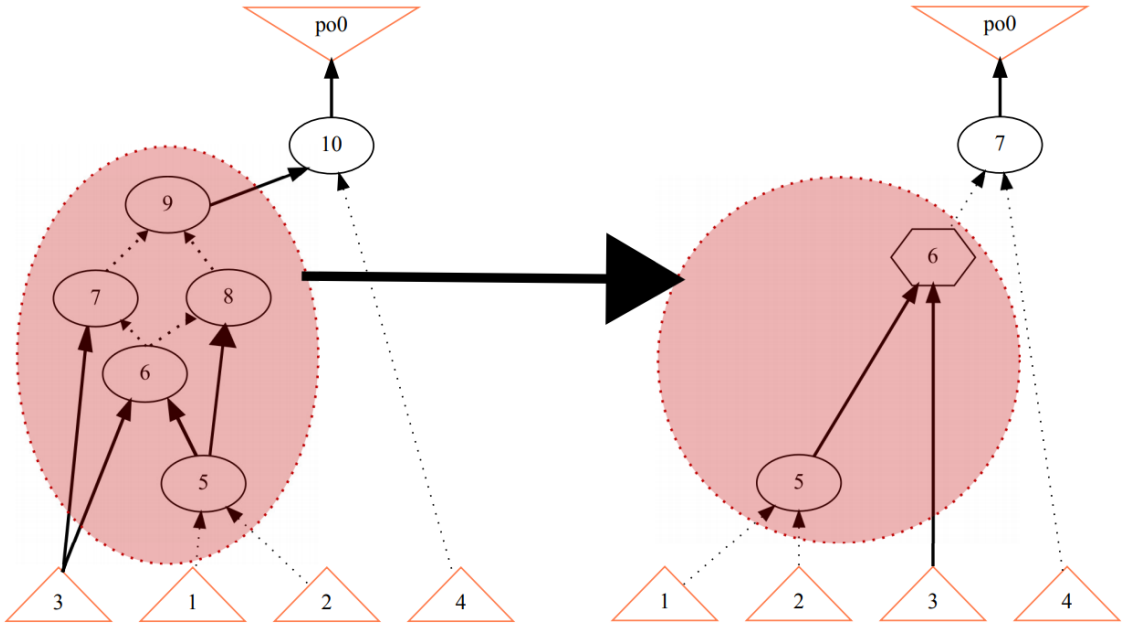

Figure 3.9: XAIG based rewrite example. Hexagon node represents a XOR gate [43].

A recent research also benefits from AIG rewriting. Exact synthesis[3] rises up and AIG rewriting offers a stable basis for further research activities. However, there are major problems of state of the art methods: enumerate Boolean functions to pre-compute optimum networks, thus limiting the approach to small subnetworks and non-satisfactory strategies to select subnetworks for rewriting. The following papers try to deal with mentioned problems [41, 6, 86, 102].

An interesting paper [87], published in 2019 by research group of Berkeley and EPFL introduces a novel methodology for multi-level synthesis, that is independent from a specific structure, but defines synthesis procedures using an abstract concept definition of a logic representation.

[3]Exact synthesis finds an optimal network that fulfills input parameters (e.g., depth or size).

Chapter 4

Introduction to polymorphic electronics

Closely before an inception of polymorphic electronics was an attention focused on the requirement to find appropriate electronic technology which could potentially yield substantial benefits for long-term autonomous unmanned space missions, where the fault-tolerant aspects become a major issue. It was discovered that polymorphic electronics can automatically compensate the fault states to a large extent and gradually adapt the system parameters to an actual operating environment with mostly harsh or extreme conditions. Adrian Stoica and his team at NASA JPL (Jet Propulsion Laboratory), who is regarded as the founder of polymorphic electronics field as such, has introduced the term of Polymorphic electronics in 2001 [106].

Polymorphic electronics is a relatively young discipline in the field of digital circuits and systems. It can by classified as a group of digital circuits having an ability to perform more than one intended function, while the interconnection of a given circuit remains the same layout in all the operating modes. A choice of an active function, which a circuit is going to perform, is strongly dependent on operating conditions (temperature, pressure, humidity, voltage polarity, etc.). A state of surrounding environment can be accurately described by means of a physical quantity with an impact on electrical properties. Then, it is possible to unambiguously determine what function will be executed with respect to a specific value of this variable [106], [20]. It is very important that a change of a polymorphic circuit function comes into effect without any eminent delay perceived and sensitivity to the environment is naturally embedded into the circuit itself [106].

It is important to mention that all the circuit functions are designed in a fully intentional manner rather than, for example, as a specific fault condition caused by exceeding certain operating parameters of a circuit. State of an environment, where a circuit is going to be deployed, can be expressed by a physical quantity having a direct influence on the electrical properties of circuit building elements. Subsequently, it is possible to determine an function to be performed by desired circuit according to a specific value of a physical quantity [105]. Such behavior is applicable for circuits that must adapt itself to unfriendly environment, e.g. by imposing power save mode [89].

An another significant attribute is stemming directly from the fundamental notion of polymorphic electronics itself. Construction of multi-functional circuits in an efficient way is expected. An advent of polymorphic-based circuit elements with multi-functional capa-

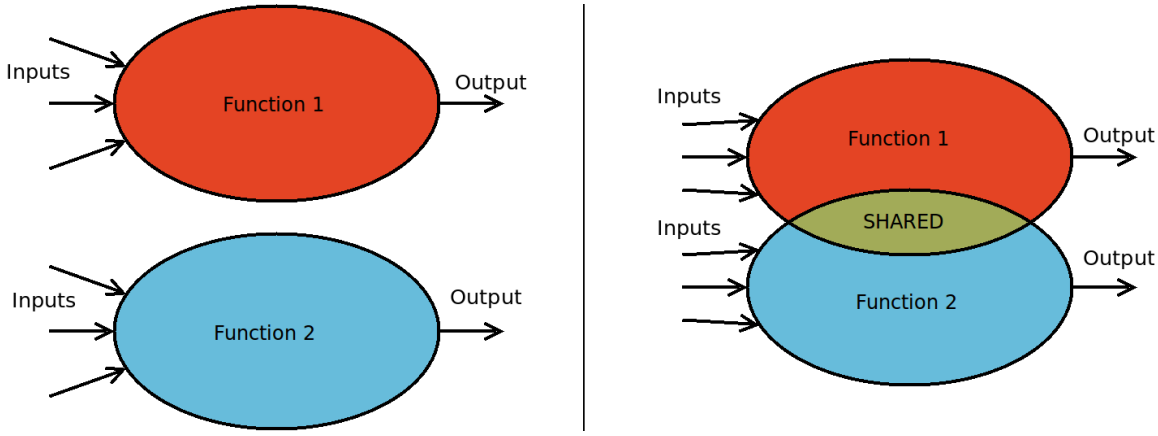

Figure 4.1: On the left are two circuits designed separately. On the right are two functions designed in polymorphic technology, where sharing of common resources is expected.

bilities may unlocks a way towards alternative solutions of the traditional reconfiguration of digital circuits [16], [76] or [93].

Typical characteristics with a substantial impact on a core functionality of polymorphic circuits [92] can be briefly specified by the following points:

1. Polymorphic circuits should place lower demands in terms of necessary chip area for their implementation. It is anticipated that two different functions can share a large portion of an existing circuit structure and thereby saving a considerable amount of resources. See figure 4.1.

2. Transition between different function modes, i.e. selection of an active function to be carried out, is substantially dependent on a current state of an environment. A switching is an immediate event without any pertinent overhead. It is purposely embodied into elementary circuit building blocks during the design phase. Polymorphic circuits respond to a current state of an environment in a global fashion. It technically means that all the circuit elements share the information about state of the environment at the same instant of time. Such type of circuit, thus, represents an example of a distributed system.

Polymorphic circuits can be built of polymorphic gates that exhibit polymorphic behavior. For example, polymorphic gate NAND/NOR has two modes, where the NAND operation is performed in mode one and the OR operation is performed in the second mode. These polymorphic gates are usually based and created from unconventional materials, such as silicon nano-wires, silicon nano-tubes etc., that exhibit unstable behavior.

4.1 Application scenarios

Adrian Stoica, the founder of polymorphic electronics, noted some utilization of polymorphic circuits in practice during introduction of polymorphic electronics term. The first advantage can be observed as „extra functions“ in addition to desired first function. These extra functions are triggered by external stimulus. As possible useful applications of polymorphic electronics can be considered usage in authentication signature / watermark, extra

protection from reverse engineering - where real operational function is appearing only in special conditions, protection of unauthorized usage by integration of biometric data into circuit, or providing another, hidden communication channel. Polymorphic electronics can replace some kinds of sensors/detectorsin case their usage is not allowed [109].

Another proposed application, also by A. Stoica, is called „smart fuses“, where polymorphic electronics can react for example to increase of temperature. This change of an environment triggers a new functionality of electronic device. Advantage of switching between functionalities within polymorphic circuits (as a default property of polymorphic electronics) is performed without complex chip reconfiguration [109].

The last possible application proposed by Stoica in [109] can be addressed to „security“ feature. Polymorphic circuits can perform hidden or a secret function during attacking a chip or reverse engineering in order to mask the original function. In other hand, polymorphic electronics can perform a primitive function (for example adder or clock generator) until the key conditions are not met. When key conditions (voltage level, voltage polarity, temperature, humidity, ...) are fulfilled, the secret function can be unlocked. This secret function may appear only when particular conditions are created. This feature can be deployed for tagging or some kind of verification of security check [109]. Sekanina et. al. had experimented on this topic and introduced how polymorphic electronics can be utilized to implement unique on-chip IDs [97]. Stoica also mentioned that polymorphic circuits sensitive to biometric patterns are able to unlock secret function only if circuit receive specific biometric signal.

Other applications of polymorphic electronics are presented by Sekanina et. al. This research group have applied polymorphic electronic in the area of test and diagnostic. The literature [103] introduces reduction of test vectors volume based on polymorphic reconfiguration of some gates (replacement of ordinary gates by polymorphic gates) of circuit under the test. This approach significantly helps to reduce test length to approximately 70% of runtime. Further useful applications can be observed in the field of self-checking circuits. The paper [90] describes a polymorphic self-checking adder with capability to recognize failures. In the first mode, the adder behaves as usual, while in the second mode, the circuit provides an error code to the primary outputs.

The same research group had invented another interesting application using polymorphic components, namely polymorphic FIR Filter [99]. The filter can operate in two modes, where the first mode is considered as a standard mode (filter performs normal operation), while the second mode operates with reconfigured filter coefficients and disconnected some parts of filter. It can lead to power consumption savings with preservation of reasonable quality of filtering [95]. In the sequel on the filters topic, the same research group also implemented a polymorphic bi-functional image filter [98], where authors observed, that solutions exhibit a significant reduction in utilized operations and interconnects with respect to multiplexing of conventional solutions.

Other work, presented by Růžička, demonstrates utilization of polymorphic electronics to design digital circuit controllers, to design digital circuit controllers, that elegantly behaves in the case of inconvenient situations, e.g. when a battery goes low or a chip temperature cross some safe level. The paper [89] describes an algorithm for designing gracefully degrading circuit controllers using polymorphic electronics. Mentioned research implies that polymorphic electronics is also applicable in the field of sequential circuits that has been confirmed by Adrian Stoica who designed JK flip-flop gates composed of polymorphic gates [82]. Růžička also tried to use polymorphic electronics for construction of reliable circuits with aim to work under extreme conditions, such as high temperature, low

voltage, etc. For example, circuits designed this way can fall into save mode and reduce features in order to withstand difficult conditions [91].

Application of polymorphic electronics has been observed also in research area of cellular automatons. An interesting work is an implementation of transition functions by polymorphic circuits with globally switchable rules [118].

Last but not least is utilization of polymorphic circuits to build of mono-function circuits with expectation of efficient implementation by polymorphic gates in comparison to conventional solution. Gajda presents a 2-bit full adder and 5-input majority/parity functions composed of polymorphic gates, where area reduction have been achieved [38].

The most of listed applications has been built and verified on REPOMO32 platform [96], which contains an array of polymorphic and ordinary digital circuit elements based on CMOS technology. REPOMO32 was utilized in order to achieve the desired multi-functional behaviour in a target environment for these applications:

1. Polymorphic FIR signal filter [99].
2. Transition function of cellular automaton [118].
3. Safety and fault tolerant systems [92], [91].
4. Security measures physically unclonable functions [97].

4.2 Polymorphic circuits

From a formal point of view, a polymorphic circuit is an electronic digital circuit which can be described as follows in definition 12 [20, 92]:

Definition 12. ***Polymorphic circuit***
Let K is a set (library) of logic gates. Let G is a graph $G = (V, E)$. Then, a polymorphic circuit is $G = (V, E, \phi)$, where:

1. *V is a set of vertices (ports of circuit components),*
2. *$E = \{(a, b)|a, b \in V\}$ is a set of edges (interconnections),*
3. *$\phi = \{\varphi_1, ..., \varphi_n\}$ is a set of projections, where $|\phi| > 1$. Each projection $\varphi_i \in \phi$, assigns a logic gate from set K to each node from V, $\varphi_i : V \to K$ for $\forall i = 0..n$.*

Then, graph G explicitly determines interconnection of the individual gates from set K and, therefore, particular structure of a given circuit, which is able to realize one of the meaningful intended functions from a set $\Phi = \{F_1, ..., F_n\}$, where $|\Phi| > 1$.

Once the polymorphic circuit is defined, it is possible to define an environment sensitivity of polymorphic circuit to physical quantity (definition 13):

Definition 13. ***Environment sensitivity***
Let X be a physical quantity, assuming values of the real numbers domain R and describing an operating environment of the circuit. A projection $\pi : Y \to \Phi$, where $Y = \{I_i|I_i \subset R\}$ is a disjunct set of intervals of values of the quantity X. If the quantity X has a value $X(t_1) \in I_k$ at a time t_1, where $I_k \subset R$ is an interval from R, then the circuit represented by the graph G performs a function $F_k \in \Phi$ at the time t_1, briefly $\pi(I_k) = F_k$. If the quantity X has a value $X(t_2) \in I_m$ at a time t_2, where $I_m \subset R \wedge I_m \cap I_k = \emptyset$, then the circuit

represented by the graph G executes a function $F_m \in \Phi$ at the time t_2, briefly $\pi(I_m) = F_m$. Note that even such intervals of X may exist, on which the function of the circuit is not defined [92].

Operating environment affects a polymorphic circuit, within the meaning of function selection from $|\Phi|$, which may be, for example, level of power supply voltage, temperature, humidity, pressure or similar physical quantity. Note that a function may be undefined for some intervals of particular physical quantity. However, one exception is made by „external signal", that may control function selection from $|\Phi|$. Then, the polymorphic circuit is not sensitive to environment, but is sensitive to external signal.

Definition of a polymorphic circuit introduced above (definition 12) reveals that the structure of a circuit – a graph G, i.e. specific rendition of circuit components interconnection, always keeps its layout. On the other hand the function of the circuit is, of course, allowed to transition from one mode to another, and the function of individual components must therefore vary for different modes (functions to be performed). So the key to the circuit polymorphism lies in a set of fundamental building components. These are exactly the devices that change their function in accordance to a value of physical quantity describing an actual state of the environment. This observation is regarded as a key pillar of the approach. It also makes the whole concept more universal and independent on specific technology used for implementation of the logic.

4.2.1 Polymorphic logic gates

Polymorphic circuits are constructed of polymorphic components, namely polymorphic gates. A polymorphic gate represents a simple circuit building component which implements a set of elementary logic (boolean) functions. An actual function of the gate is selected due to an influence of the operating environment. If the gate exhibit e.g. NAND function for some range of the power supply voltage (Vdd) and e.g. NOR function for another range of the Vdd, the gate could be specified as a NAND/NOR gate controlled by Vdd. It is assumed that a polymorphic gate may perform no more than one function at a given moment in time. It is also important to emphasize, that rules discussed in previous section 4.2 are valid for polymorphic gates also. Each polymorphic gate can be considered as a small polymorphic circuit with difference, that gates are located in Y-chart one level lower.

Today's gates are often based on unipolar semiconductor transistors, but the concept of polymorphic electronics has more general nature and allows to conveniently employ new emerging devices like graphene [110] or nanowire structures [120], ambipolar devices utilizing suitable organic polymers with semiconductor-like properties [111] etc., which make it possible to obtain a new generation of advanced multi-functional logic gates.

Polymorphic gates may be classified into particular classes according to sensitivity to physical quantity. Three the most common/known classes have been already introduced by Stoica in the literature [109] presenting the *polymorphic electronics* term. The first class is a group of gates sensitive to power supply. These gates can morph with a change of voltage level or voltage polarity [90, 103, 109, 108]. There were also a few success attempts to fabricate polymorphic gate (see figure 4.2) with described behavior [96].

The second class marks polymorphic gates sensitive to environment temperature. However, there are still not many temperature sensitive gates fabricated yet in comparison to the first class. It is probably caused by a skeptical view of „controlling electronics by temperature". However, transistors based on anorganic semiconductors exhibit significant

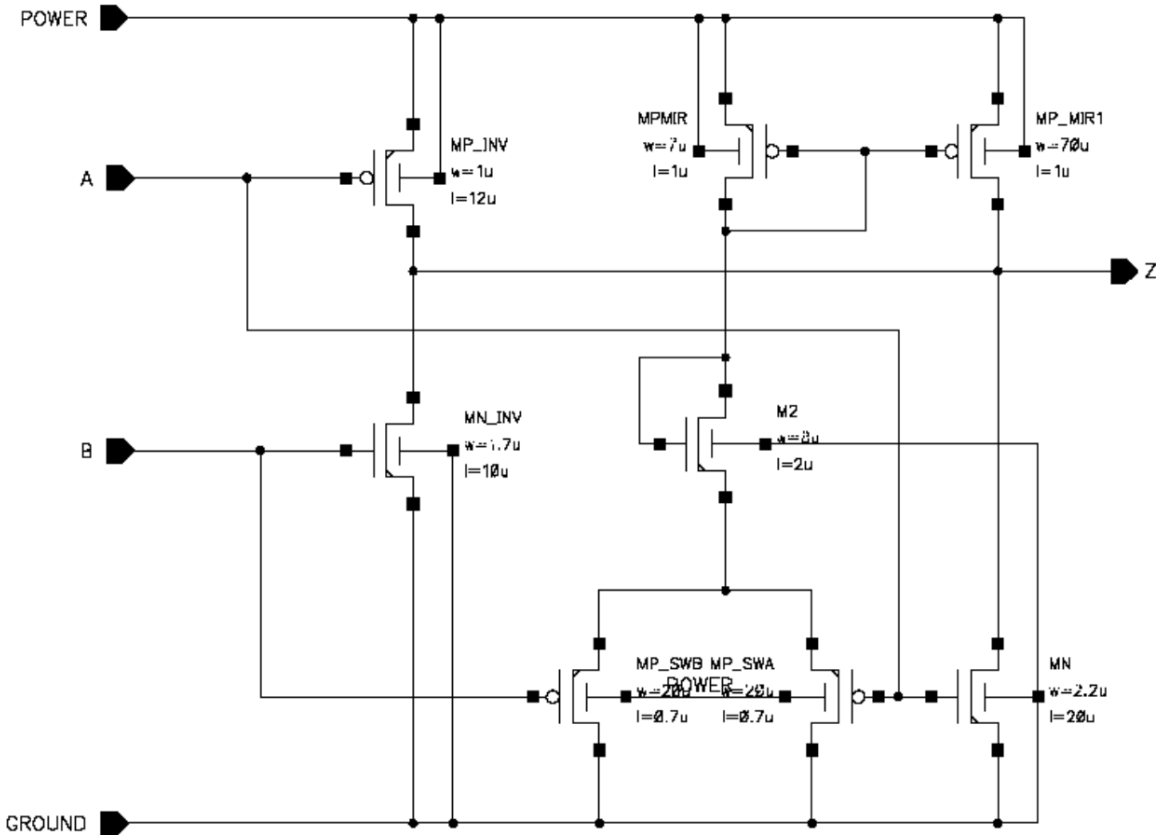

Figure 4.2: Polymorphic NAND/NOR gate: the NOR when Vdd = 3.3V and the NAND when Vdd = 5V [96].

temperature dependency. This property is meaningful for polymorphic components, despite the conventional trend to develop linear transistors as much as possible. Polymorphic gate array, REPOMO32, developed at Brno University of Technology, exhibits polymorphic behavior dependent on environment temperature [96].

Polymorphic gates controlled by external signal forms the third class of the most common types of polymorphic gates. Gate polymorphism is controlled by wired signal, spread over the entire circuit, carrying *environment* information. This kind of polymorphic gates may look like multiplexers. However, in the case polymorphic gates, it is possible to imagine multiplexers integrated inside one polymorphic gate. Polymorphic gates controlled by external signal can be built of so-called *ambipolar transistor*. Ambipolar transistors are described in section 4.2.1. Anyway, polymorphic gates might not be built of ambipolar transistors necessarily.

Logic gates are usually available as a library with known gate properties, such as resistance, delay, power consumption, power supply voltage, input signal tolerance, output signal levels and dimensions. These information help to coordinate synthesis process and low level layers (below or equal to transistor level) are hidden for digital designer. A composition of polymorphic gates on transistor level has been researched by Mr. Nevoral, who designed polymorphic gate library, namely *PoLibSi* [78], which contains eight sets of efficient bi-functional two-input polymorphic gates, whose functions are selected by mutual polarity of dedicated power rails. Sets differ in utilized transistor type (MOSFET or multi-

gate ambipolar transistors) and they are optimized to delay, switching power, transistor count or input impedance. The library has been designed using evolutionary algorithms and further validated in a SPICE simulator. Nevoral highlights that the sets are complete. It means that any pair of two-input Boolean functions is covered by a set in the library.

Utilization of the polymorphic electronics concept, and therefore construction of more complex circuit arrangements, is limited by the availability of suitable polymorphic gates. Two kinds of the polymorphic gates have been physically fabricated already. Polymorphic gates reported in literature were either only simulated or tested in a FPTA [122] are belonging to *gates of old generation*. For instance, the 6-transistor NAND/NOR gate controlled by Vdd was fabricated in a 0.5-micron HP technology [107]. Another NAND/NOR gate controlled by Vdd was introduced in [90]. The gate was designed with the aim to achieve properties and criteria outlined in this section. Further research activities of J. Nevoral brought a set of polymorphic gates belonging to a new generation - PoLibSi. Gates in a set PoLibSi exhibit stable and digital behavior. Another aspect, that limits an expansion of polymorphic circuits is missing univerzal and scalable synthesis method. This thesis brings a solution for this limitation caused by missing synthesis methods.

Ambipolar transistors

The previous section mentioned a term *Ambipolar transistor*. Transistors having ambipolar properties may exhibit different behavior with respect to another physical quantity. Ambipolar transistors are mostly fabricated with four electrodes. The first three electrodes *gate, source, drain*, are identical with conventional transistors of type N and P. The fourth electrode, namely *„Polarity gate“*, is commonly used for selection of required behavior, and so P or N. Figure 4.3 shows a four-electrode ambipolar FET Silicon Nanowire transistor [112].

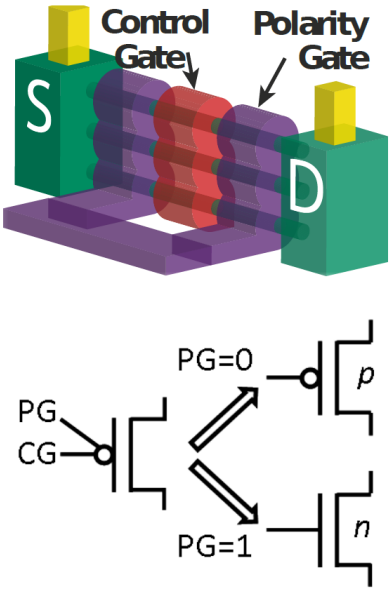

Figure 4.3: Ambipolar FET SiNW structure and polarity control scheme [112].

This kind of transistors already exists and the most of labs are able to fabricate them. A disadvantage of this property is the fourth electrode in addition, that increases number of

connected wires to the transistor. While the number of transistor is increased, the number of wires is increased linearly. The fourth electrode is a step back in the case of polymorphic electronics, based on the opinion of an author of this thesis.

A global effort is to find ambipolar transistors with three electrodes only. In contrast to conventional technology, disadvantages related to the fourth electrode will not appear. Transistor mode selection would be possible to perform by power supply polarity attached to electrodes *source* or *drain*.

Based on the fact, that a three-electrode ambipolar transistor doesn't exists in real world, I tried to build a three-electrode ambipolar transistor using P-type and N-type conventional transistors. Experiments had been performed in SPICE simulator and by using real transistors also. With a power supply polarity change, the transistor has changed a function as expected, but output voltage level has been highly dependent on attached load. This negative property had been resolved using semiconductor diodes *D1, D2* connected to drain electrodes of both transistors used in replacement scheme (see figure 4.4). Figure 4.4 shows a replacement scheme of three electrode ambipolar transistor using conventional technology sensitive to power supply polarity.

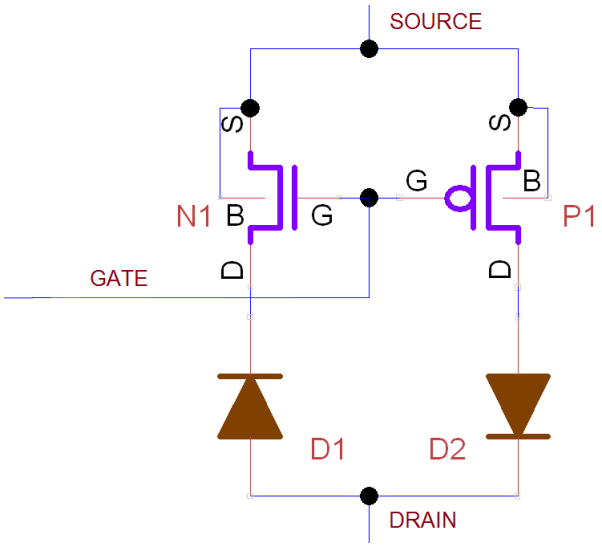

Figure 4.4: Model of three-electrode ambipolar transistor for simulation purposes.

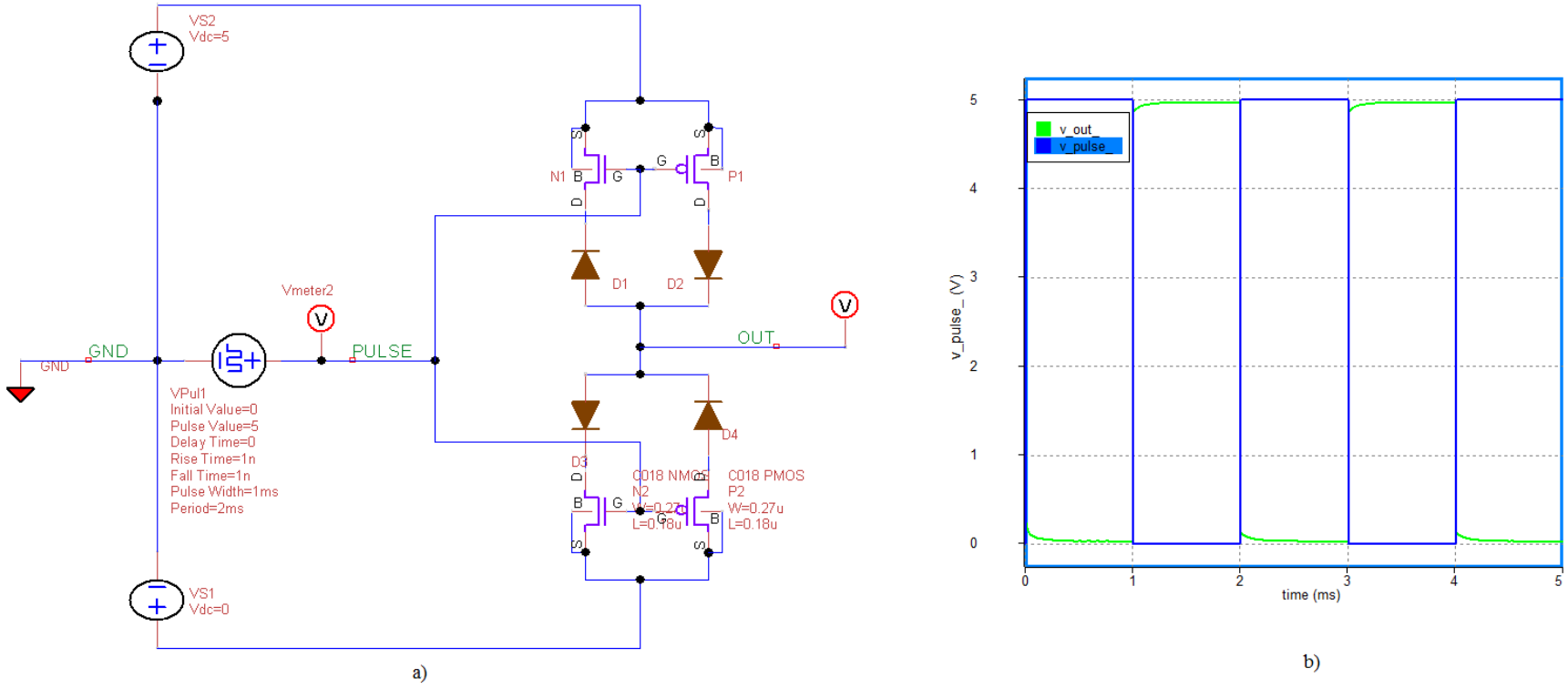

Figure 4.5: A scheme of a polymorphic inverter composed of designed ambipolar transistors and SPICE simulation.

Figure 4.5 illustrates a scheme of inverter, composed of ambipolar transistors discussed previously, at the bottom (on the left). Behavior of the inverter is valid regardless to attached voltage polarity. There is a waveform simulation in SPICE tool on the top (on the right) in figure 4.5.

It can be opposed that the introduced inverter is composed of four conventional transistors. Please note that this composition is only an attempt to build a polymorphic gate using conventional technology. It is expected that unconventional technology may decrease a number of building components to two transistors.

4.2.2 Open issues of polymorphic electronics

Despite the availability of formal apparatus and previously achieved research outcomes, there can be still revealed a number of open problems within the domain of polymorphic electronics that has to be addressed carefully.

Some of the most important problems of polymorphic electronics, which deserve further attention in order to be resolved or further improved from the current level of advancement, are especially the following ones.

The 1st issue is obviously connected with a effort to provide suitable polymorphic components (gates). As it was demonstrated above, their availability is rather limited, as only few types have been physically fabricated. Development of fully qualified logic gates (on material as well as on transistor level), that would be able to precisely and predictably react to the phenomena residing in surrounding operational environment is still in progress. Recent decade has seen an appearance of highly promising materials including graphene, organic materials with semiconductor properties, silicon nanowires and many other nanoscale semiconducting structures, which are expected to push forward facilities for polymorphic components design and fabrication dramatically [85][73]. In addition, a relatively new work exists that has expanded the number of available polymorphic gates based on conventional MOS transistors, researched by J. Nevoral: PoLibSi [78].

The 2nd aspect is a problem of an appropriate design and synthesis methods for polymorphic circuits. One of the most common approaches to polymorphic circuits design is based on using evolutionary methods. Thus, a research of efficient design methods and circuit synthesis techniques suitable for the domain of polymorphic circuits are still required. An obvious requirement rises especially in connection with construction procedure of graph G as it is a common practice to take into account presence of polymorphic gates alongside the conventional ones. Such method must be able to construct a graph G of polymorphic gates, which represents a given circuit. Furthermore, desired circuit must perform all the required functions once exposed to an influence of target operating environment. The circuit must be able to perform intended functions as prescribed in accordance to the present state of the target operating environments.

Finally, the 3rd problem is basically concentrated around an appropriate method of how to efficiently describe a structure of a polymorphic circuit and alleviate its potential optimization in a reasonable time frame.

This thesis tries to cover the 2nd and 3rd open problems, with aim to create a logic synthesis basis for polymorphic circuits. In view of this thesis and thesis of J. Nevoral, it is very probable, that polymorphic electronics will be very close to practical deployment, because both works exhibit good results of above mentioned problems.

Chapter 5

Polymorphic circuit synthesis and optimization

Synthesis methods of ordinary digital circuits have to solve a problem an interconnection and nodes placement of graph G, searching just for one particular function F. If a suitable canonical form[1] of F is found, a structure of G can be easily inferred from it. For polymorphic circuits, this approach tends to exhibit higher complexity. The reason is that just one graph has to cover several functions from an existing set $\phi = \{F_1, \ldots, F_n\}$, that are requested from a given circuit, and the demand of multi-functional operation has to be fulfilled in the same time. The task to find the same form for all the functions F_1 to F_n (with different elementary functions on the same position) is, therefore, not trivial at all.

Contemporary the polymorphic circuit design takes place mostly at a gate level. During the course of numerous experiments carried out within the field of polymorphic circuits it became obvious that circuits designed solely with polymorphic gates are less useful than in situation, which involves both polymorphic and conventional elements. It should be noted that the number of conventional gates typically exceeds the number of polymorphic gates of a target circuit currently. In many cases it is also sufficient to use a single polymorphic gate type, if such gate executes logically complete functions (e.g. NAND / NOR). If a wider selection of polymorphic gates is available, it could ultimately lead to better solution. However, the overall complexity of the problem could increase (in state space) [92] significantly.

5.1 Existing polymorphic design and optimization methods

Recent advances in the domain of multi-functional circuits have brought into being several approaches how to handle the synthesis process of polymorphic circuits performed at a gate level. However, all of the methods presented up to now fail to comply in some measure with the general requirements shared by common applications (e.g., resulting size, propagation delay, operating frequency, runtime duration, etc.), which is particularly apparent for demanding synthesis tasks.

[1] **Canonical representation**: Representation is unique for a particular function and variable order.

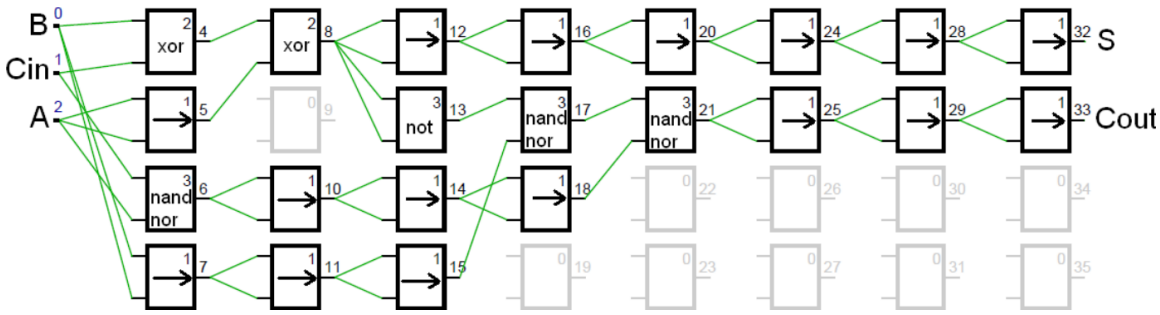

Figure 5.1: Self-checking polymorphic adder in REPOMO32 designed by CGP [96].

5.1.1 Ad - hoc: common sense design

Ad hoc approach is regarded as a circuit design method without using any explicit formal design techniques or supplementary tools. Only fundamental knowledge and an experienced designer is expected. This method allows to construct only relatively small circuits. The method is therefore not the right choice for bigger circuits [92].

5.1.2 Evolutionary design

Regardless of the existing drawbacks visited in parent section, utilization of evolutionary algorithms and techniques still plays an important role in connection with the optimization of multi-functional circuits. In fact, such methods have been a natural choice ever since the invention of polymorphic electronics. Especially in situations when certain awareness of the expected result exist, it becomes less clear how a particular objective was achieved. However, it is not an exception to obtain decent solutions.

CGP (Cartesian Genetic Programming) [63, 62] is considered to be one of several field-proven methods generating satisfactory results. The design of polymorphic circuits using CGP is almost the same compared to the conventional CGP design of circuits except the fact that it involves extended fitness function. The difference lies only in the fitness function. It is necessary to ensure that correctness of a circuit is evaluated for all functions / modes that the circuit has to perform. However, scalability becomes the major issue for really complex circuits due to possible explosion of state space, which needs to be searched [92]. Number of evolutionary-based techniques is capable of providing very efficient polymorphic circuit solutions even from scratch [94, 57]. Nevertheless, it has been found that usage of such methods makes sense for small problems only (up to 15 inputs [39]). Figure 5.1 shows an polymorphic adder designed by CGP.

Unfortunately, a process and result of various evolutionary techniques and optimization schemes derived from them is hard to predict in advance and get firmly under control. Another important problem is scalability aspect. Despite that, the evolutionary design is the most effective approach today. The proposed algorithms are capable to find many solutions, which may not be satisfying at the beginning, but the algorithm can generate successively better solutions. New solutions are derived as long as the previous ones do not achieve a perfect match with the requested functionality specified by, for example, the truth table. This approach may be conceived in a such way the fulfillment of even several parameters may be demanded.

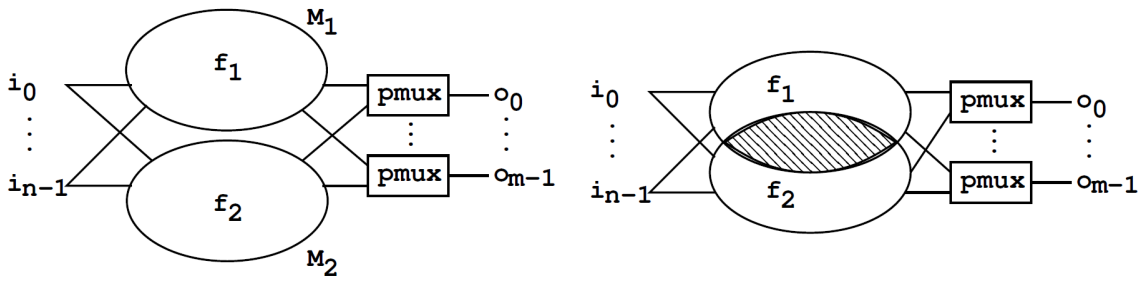

Figure 5.2: Multiplexing of conventional circuits by means of using polymorphic multiplexers: a) independent modules, and b) sharing gates between modules [37].

5.1.3 Polymorphic multiplexing

A simple and straightforward design method of multi-functional circuits is called polymorphic multiplexing - switching functions with respect to an environment state [37]. This technique was designed by Gajda and Sekanina [39]. This is a rather simple method that strives to adhere to the principles of conventional circuit design. In short, the principle is this: Every function that the polymorphic circuit will perform is designed in a conventional way using standard CMOS-based blocks. The output of each proposed circuit is connected to a so-called polymorphic multiplexer, that finally performs selection of a given input, depending on environmental conditions.

Polymorphic multiplexing can be explained formally as follows: Let's have two conventional digital circuits M_1 and M_2 implementing two different logic functions f_1 and f_2 (see a) on figure 5.2). Both circuits may be optimized using ABC tool [1] and have the same number of inputs PI and outputs PO. Outputs are connected using polymorphic multiplexers. In the first mode, primary output 0 of circuit M_1 is propagated to a common output o_0, primary output 0 of circuit M_2 is propagated to a common output o_0 in the second mode. In fact, the intended sharing of common parts is not achieved without additional steps of the synthesis process.

This initial approach is not very efficient in terms of occupied area (no sharing of similar circuit parts), which is in a direct contrast to expected benefits of using polymorphic electronics [92] [37]. A closer analysis of conventional circuits, which are supposed to be merged together using polymorphic principles, reveals the fact that it is possible to share common parts of the participating circuits. In fact, the original version of polymorphic multiplexing method can be slightly optimized with this assumption in mind. Figure 5.2 shows closer view at the principals of polymorphic multiplexing method.

5.1.4 PolyBDD

Gajda [37] suggested a method for synthesis of polymorphic circuits using binary decision diagrams (BDD). This method is called PolyBDD. It uses a concept of so-called multi-terminal BDD, which is an extension of binary decision diagrams with the terminal nodes of the diagram containing integer values. The PolyBDD method is using these values to represent a possible relation between input variables and a relevant output. In case of polymorphic circuit expected to implement two different functions working in two allowable

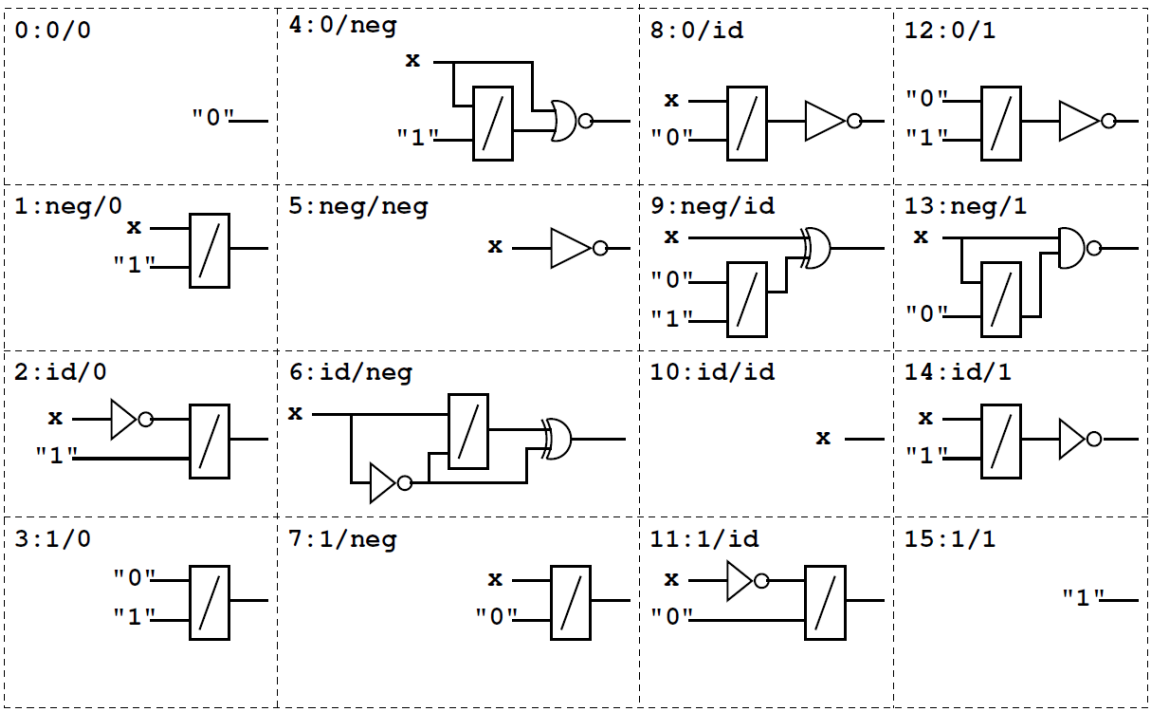

Figure 5.3: Conversion table for the transformation procedure of PolyBDD into a polymorphic circuit [37].

modes while using polymorphic gates, it yields 16 possible combinations. Values in terminal nodes of MTBDD tree will be therefore integers from an interval <0-15> (figure 5.3). Detailed explanation of internal principles of PolyBDD method can be found in [37] and [39]. Principal drawbacks of this method lie in relatively sparse exploitation of polymorphic gates, i.e. these are practically used only in a role of input/output switches. Further optimizations relies on evolutionary optimization of polymorphic circuit.

5.1.5 Recent work on logic synthesis of polymorphic circuits

Some further work on polymorphic circuit logic synthesis has been done in recent years. This section summaries interesting published papers related to polymorphic logic synthesis topic.

Evolutionary design of polymorphic circuits with the improved evolutionary repair [124]

In 2013, a Chinese research team tried to improve evolutionary synthesis of polymorphic circuits by including *Repair algorithm* into evolutionary synthesis process [124]. Evolutionary synthesis algorithms are the most usable techniques for design of polymorphic circuits, but they still face scalability problems. The most complex polymorphic circuit designed evolutionary is a sorter/multiplier with 6 primary inputs and 6 primary outputs. Thus, designing more complex polymorphic circuits is still the biggest obstacle in the case of use of evolutionary algorithms.

The team published the *Repair algorithm* for evolutionary synthesis algorithms in order to accelerate the evolution process and overcome *Stall effect* in 2012 [123]. *Stall effect* is

a state when the best fitness of the evolutionary population is not increasing or increases slowly. It may take several generations to find a population that will have progressive fitness again. The repair algorithm is deployed when stalling state is detected. It repairs incomplete, best individual to a target circuit directly. This repair technique removes time wasted in stall effect.

Authors applied the repair algorithm onto evolutionary design of polymorphic circuits in order to make evolutionary techniques more scalable.

Evolutionary Design of Polymorphic Circuits with Weighted Sum [57]

Sekanina et. al. experimentally applied a CGP design of logic circuits. The fitness is computed for the whole polymorphic circuit without respect to complexity of each desired function [94]. A Chinese research team extended CGP evolutionary approach applying weighted sum, that helps to increase the success ratio and decrease the evolutionary generations in 2007 [57]. Without loss of generality, authors expect that a polymorphic circuit can perform two independent logic functions in two independent modes. Both circuit (performing function 1 or function 2) can be regarded as a traditional digital circuit. Circuits in mode 1 or mode 2 may have different complexity. Thus, a characteristics of polymorphic circuits (to be evolved) should be taken into account for generating the expected polymorphic circuit. Authors involve weighted sums W_1 and W_2 in computation of fitness function of desired polymorphic circuit in the following form: $F = W_1 * F_1 + W_2 * F_2$, where W_1 W_2 are weight coefficients, and F_1 F_2 are common fitness for circuits in desired modes. For detailed description, see experiments presented in [57]. Unfortunately, experiments are presented on quite small circuits only.

In 2015, the team extended the weighted sum approach with periodical weight adjustment, changing weight factor are changed periodically according to the sinusoid function, expressed as: $W_i = sin(2\pi t_i) + 1$, where $1 \leq i \leq n$. Authors present improvement in comparison to initial weighted sums work, thus, experiments are presented on small circuits only. For more details, visit [58].

Design Methods for Polymorphic Combinational Logic Circuits based on the Bi-Decomposition Approach [56]

The newest paper related to design and optimization of polymorphic circuits is dealing with Bi-Decomposition approach [56]. Authors apply Bi-Decomposition, known from traditional design of logic circuits [104, 71]. The work promises an algorithm for design of relatively large circuits with gate-efficient implementation and utilization of polymorphic gates in contrast to PolyBDD, where the lesser utilization of polymorphic gates is criticized. Experiments report number of used gates and utilization of polymorphic gates. This paper looks very promising, however, it is currently available on arXiv[2] only without admitting reviews.

[2] arXiv is a free distribution service and an open archive for scholarly articles.

Chapter 6

Proposed two-level design and optimization methods

An initial research of synthesis and optimization methods of polymorphic circuits, related to this thesis, has started with two-level design methods due to their simplicity. During this period, two two-level approaches had been introduced and had served as a relatively stable basis for further research of synthesis methods of more complex polymorphic circuits.

6.1 Design of polymorphic circuits using NAND/NOR Gates

The initial research had been triggered by physical availability of polymorphic gate array REPOMO32 [96], having 32 NAND/NOR polymorphic gates. REPOMO32 was developed by Lukas Sekanina et. al. at the Faculty of Information Technology, Brno University Of Technology for experimental purposes and for validation of evolutionary approaches. REPOMO32 polymorphic gates are special structures developed on transistor level encapsulated in ceramic DIL package having 28 pins.

Synthesis of polymorphic logic, suitable for REPOMO32, has been done using evolutionary approaches. At the beginning of the research, the main aim was to find easy design methods for designing polymorphic circuits. Thus, experimental work has started with study of two-level representations and looking for ways to design NAND/NOR polymorphic circuits. Results lead to following design method using NAND/NOR gates and it is well applicable to REPOMO32 gate array.

Following text describes details of the proposed method, while the description contains illustrative examples. Method can be divided into 7 rigid steps:

1. Input: truth tables of both functions. See table 6.1 for example.

2. Minimization of both functions. An output of this step must be boolean formula in SOP (Sum of Products) describing the first function and boolean formula in POS (Product of Sums) describing the second function. In this step we obtain two formulas. The first one comprises group of terms connected by operator OR and second formula consists of groups of terms connected by operator AND.

3. Transform the (level 1) ORs into NANDs at the formula 1. Transform the (level 1) ANDs into NORs at the formula 2 (deMorgan's laws).

Table 6.1: Truth table specification of initial functions.

	A	B	C	F_1	F_2	note
1.	0	0	0	0	1	
2.	0	0	1	1	1	
3.	0	1	0	1	1	
4.	0	1	1	0	1	
5.	1	0	0	0	1	
6.	1	0	1	0	0	inverted 3^{rd} F_1 inputs
7.	1	1	0	0	0	inverted 2^{nd} F_1 inputs
8.	1	1	1	0	1	

4. The obtained expressions now require an adjustment using Boolean algebra in order to make them look as much identical as possible.

5. The parts of the expressions, which differ only in the operator (NAND/NOR) will be implemented by means of deploying the polymorphic principles.

6. Isolation of dissimilar parts. Because some parts of the circuit can not be joined (parts of expressions are different), it is necessary to isolate them from each other. It is recommended to use some polymorphic gates, which will serve as polymorphic multiplexer, identity / negation and negation / identity.

7. Join the two formulas together and create the final formula describing polymorphic circuit.

Let us assume that the first function is true for some input combinations, otherwise is false. The second function is true for the most of the combinations except for the **inverted** inputs of the first function, where it was previously true, see table 6.1.

The proposed method brings certain benefits:

- It generates solutions with the polymorphic gates naturally embedded into the circuit during the application of individual steps of the method.
- The method seems to deliver satisfactory results for reasonable sized designs of polymorphic circuits.

However, certain constraints of the proposed design flow have been identified:

- The 4^{th} step is not deterministic. It requires an intervention of an operator or a heuristic algorithm.
- XOR gates are not supported as it isn't a logical complete function (it is possible to express by combination of simpler gates). Therefore its utilization potential would be rather limited. The next reason of absence XOR gates is connected with the fact that XORs are not currently available inside REPOMO32 chip as a polymorphic gate [96].

6.1.1 Examples

This section shows two examples demonstrating various aspects of the proposed method application. The first example simply demonstrates the best case, having an ideal combination of the two functions. The second example shows an application of the proposed method with a random functions combination.

Example: The best case

1. Let us have two functions specified by On-set and Off-set form:

$$\Pi_{F_1} = (0, 4, 10, 14)$$

$$\sum_{F_2} = (1, 5, 11, 15)$$

2. The functions are minimized by means of Karnaugh maps. These resulting formulas were obtained (to understand notation, see section 2.2):

$$F_1 = AC\bar{D} + \bar{A}\bar{C}\bar{D}$$

$$F_2 = (A + C + \bar{D})(\bar{A} + \bar{C} + \bar{D})$$

3. Modify functions into NAND/NOR form at level 1:

$$F_1 = \overline{\overline{AC\bar{D}} * \overline{\bar{A}\bar{C}\bar{D}}}$$

$$F_2 = \overline{\overline{(A + C + \bar{D})} + \overline{(\bar{A} + \bar{C} + \bar{D})}}$$

4. At this moment we have very similar formulas and nothing needs to be isolated. Now we can create the final formula ($\frac{*}{+}$ is polymorphic gate NAND/NOR):

$$F = \overline{\overline{(A \underset{+}{\overset{*}{-}} C \underset{+}{\overset{*}{-}} \bar{D})} \underset{+}{\overset{*}{-}} \overline{(\bar{A} \underset{+}{\overset{*}{-}} \bar{C} \underset{+}{\overset{*}{-}} \bar{D})}}$$

This example shows the best case. This solution saves more than a half of gates in comparison to conventional solution. The polymorphic solution requires 7 polymorphic gates in total. As it can be seen, conventional solution needs two 3-input AND gates, one 2-input OR and two 3-input OR with 2-input AND.

The polymorphic solution consumes two 3-input NAND/NOR gates and one 2-input NAND/NOR gate. Finally, it brings the overall savings of 57%.

Example: The usual case

1. Let us have two functions specified by On-set:

$$\Pi_{F_1} = (2, 3, 5)$$
$$\sum_{F_2} = (2, 4, 5, 6)$$

2. The functions are minimized by means of using Karnaugh maps [49][116]. These resulting formulas were obtained:

$$F_1 = A\bar{B}C + \bar{A}B$$
$$F_2 = (\bar{B} + C)(\bar{A}B)$$

3. Modification of functions into NAND / NOR form at level 1:

$$F_1 = \overline{\overline{A\bar{B}C} * \overline{\bar{A}B}}$$
$$F_2 = \overline{\overline{(\bar{B} + C)} + \overline{(\bar{A}B)}}$$

4. As it can be obviously seen, input A is left over in function 1. Therefore, Input A must be isolated. Polymorphic multiplexer (labeled as "|" in the expression [1]) will be used for this purpose. Now we can create the final formula:

$$F = \overline{\overline{(A|0\underset{+}{*}\bar{B}\underset{+}{*}C)}\underset{+}{*}\overline{(\bar{A}\underset{+}{*}B)}}$$

This example shows a case in which is necessary to isolate some parts of a circuit. This solution have used polymorphic multiplexer for the isolation of input A in mode 2. The resulting solution saves nearly a half amount of gates in comparison with conventional solution, which is comprising 7 gates in total. In this case, the adoption of polymorphic principles helps to achieve 43% savings. However, overall savings may be affected by a combination of input functions.

6.1.2 Related summary

The proposed synthesis method is based on a formal Boolean representation of corresponding input functions. Its main advantage can be recognized in rigid notation with an employment of minimization techniques, which is in a direct contrast to the competitive solutions, predominantly based on heuristic approaches.

Despite some existing constraints of the proposed approach that were identified during the theoretical analysis and subsequent experiments, it was successfully applied in connection with real functions specified by the truth table. The obtained results clearly suggest its benefits in a comparison to the conventional techniques. It is possible to estimate that further improvements can be achieved, especially when new types of polymorphic circuit component based on emerging materials will be prepared.

[1] $(left)|(right)$ - sign for 2-input polymorphic multiplexer. Left side is active in mode 1, the right side is active in mode 2.

6.2 Optimization of polymorphic circuits by searching common parts

Searching for the corresponding interconnection of graph G (see closer explanation in section 5) may not be an easy task in case of polymorphic circuitry. Directly related to this observation is a goal to propose a synthesis method that would address the weak spots of previous the attempt.

The main idea behind the approach is based on an identification of common parts across the source circuits, which are virtually shared between them as a so called common divisors, by means of exploiting techniques of function kerneling and Boolean division [42]. Kerneling was introduced in [11] to provide means for finding subexpressions common to two or more expressions. All operations used to find kernels are algebraic. Also see also section 2.4.1 for detailed background.

Typical execution flow behind the proposed method consists of the following sequence of individual steps, which are further outlined below:

1. Minimized expressions in DNF notation depict the input functions – F and G. Both functions are initially provided in two-level PLA format as a truth table.

   Example:

   $$F = ab\bar{d} + b\bar{c}\bar{d} + \bar{b}\bar{c}d + a\bar{c} + \bar{a}\bar{b}c\bar{d}$$

   $$G = ab\bar{c} + a\bar{c}d + \bar{a}\bar{b} + \bar{b}c\bar{d} + \bar{a}c$$

2. Creation of an intersection table at a dimensions given by $m * n$, where m denotes the number of term groups of F and n has the same meaning for G. This table is laid out in such way that the first column contains term groups belonging to F and the first row holds the number of terms of G. Individual cells within the table are filled up in the following way: *group of terms intersection (remaining terms of F / remaining terms of G).*

   Example:

F \ G	$ab\bar{c}$	$a\bar{c}d$	$\bar{a}\bar{b}$	$\bar{b}c\bar{d}$	$\bar{a}c$
$ab\bar{d}$	$ab(\bar{d}\|\bar{c})$	$a(b\bar{d}\|\bar{c}d)$	$\emptyset$	$\bar{d}(ab\|\bar{b}c)$	$\emptyset$
$b\bar{c}\bar{d}$	$b\bar{c}(\bar{d}\|a)$	$\bar{c}(b\bar{d}\|ad)$	$\emptyset$	$\bar{d}(b\bar{c}\|\bar{b}c)$	$\emptyset$
$\bar{b}\bar{c}d$	$\bar{c}(\bar{b}d\|ab)$	$\bar{c}d(\bar{b}\|a)$	$\bar{b}(\bar{c}d\|\bar{a})$	$\bar{b}(\bar{c}d\|c\bar{d})$	$\emptyset$
$a\bar{c}$	$a\bar{c}(1\|b)$	$a\bar{c}(1\|d)$	$\emptyset$	$\emptyset$	$\emptyset$
$\bar{a}\bar{b}c\bar{d}$	$\emptyset$	$\emptyset$	$\bar{a}\bar{b}(c\bar{d}\|1)$	$\bar{b}c\bar{d}(\bar{a}\|1)$	$\bar{a}c(\bar{b}\bar{d}\|1)$

3. The main task here is to identify those entries that exhibit a mutual intersection of a maximum size. These so called minterms[2] are shared for both input functions. Once a minterm is registered in the final expression, corresponding row and column is eliminated from the table.

[2] **Minterm**: A minterm is a product of literals. More specifically, if there are n variables, $x_1, x_2, ..., x_n$, a minterm is a product $y_1 y_2 ... y_n$, where y_i is x_i or $\overline{x_i}$.

Example:

F \ G	$ab\bar{c}$	$a\bar{c}d$	$\bar{a}\bar{b}$	$\bar{b}c\bar{d}$	$\bar{a}c$
abd	$ab(\bar{d}\|\bar{c})$	$a(b\bar{d}\|\bar{c}d)$	$\emptyset$	$\bar{d}(ab\|\bar{b}c)$	$\emptyset$
$b\bar{c}\bar{d}$	$b\bar{c}(\bar{d}\|a)$	$\bar{c}(b\bar{d}\|ad)$	$\emptyset$	$\bar{d}(b\bar{c}\|\bar{b}c)$	$\emptyset$
$\bar{b}\bar{c}d$	$\bar{c}(\bar{b}d\|ab)$	$\bar{c}d(\bar{b}\|a)$	$\bar{b}(\bar{c}d\|\bar{a})$	$\bar{b}(\bar{c}d\|c\bar{d})$	$\emptyset$
$a\bar{c}$	$a\bar{c}(1\|b)$	$a\bar{c}(1\|d)$	$\emptyset$	$\emptyset$	$\emptyset$
$\bar{a}\bar{b}cd$	$\emptyset$	$\emptyset$	$\bar{a}\bar{b}(cd\|1)$	$\bar{b}c\bar{d}(\bar{a}\|1)$	$\bar{a}c(\bar{b}d\|1)$

$$H = \bar{b}c\bar{d}(\bar{a}|1)+$$

4. Second pass through the table constructed in step 2) is commenced. This time, the task is to find a next largest intersection. The cell fulfilling this requirement is then rewritten into the final expression and the whole row and column with this particular cells are eliminated from the table.

5. Previous step 4) is continuously repeated while the table contains uncovered cells with at least some intersection. Once all the intersections are covered, it's possible to proceed with a next step.

   Example: The final expression obtained at this step is following:

$$H = \bar{b}c\bar{d}(\bar{a}|1) + b\bar{c}(\bar{d}|a) + \bar{c}d(\bar{b}|a)+$$

6. Now, it's necessary to apply a special functional block called polymorphic multiplexer (labeled as "|" in the expression), which isolates contradictory parts of functions F and G.

   Example: Now, the table contains just the remaining groups of terms which do not have any common divisor.

F \ G	$\bar{a}\bar{b}$	$\bar{a}c$
abd	$\emptyset$	$\emptyset$
$a\bar{c}$	$\emptyset$	$\emptyset$

$$H = \bar{b}c\bar{d}(\bar{a}|1) + b\bar{c}(\bar{d}|a) + \bar{c}d(\bar{b}|a) + ((abd + a\bar{c})|(\bar{a}\bar{b} + \bar{a}c))$$

   Decomposition of the obtained expression will be done as a measure towards the best possible mapping onto a set of available circuit components. The resulting expression ready for technology mapping phase may have the following composition:

   $A_2 = (a|\bar{a})$ - positive polymorphic inverter
   $B_2 = (b|\bar{b})$ - positive polymorphic inverter
   $C_1 = (\bar{c}|c)$ - negative polymorphic inverter
   $Z = (\bar{d}|1)$ - polymorphic multiplexer

   The final expression describing the example circuit:

$$H = \bar{b}c\bar{d}(\bar{a}|1) + b\bar{c}(\bar{d}|a) + \bar{c}d(\bar{b}|a) + A_2B_2Z + A_2C_1$$

7. If the both functions F and G have different number of term groups, there will remain a certain number of uncovered terms belonging to a function with higher number of term groups. Those uncovered terms are put into the resulting expression by means of polymorphic operator "|" and neutral element for addition denoted as "0".

6.2.1 Experiments with generated circuits

The proposed synthesis technique for polymorphic circuits has been tested on several circuits defined by a truth table in two-level PLA format. Detailed specification of these circuits can be found in figure 6.1. These circuits were randomly generated using PLA Generator [74]. The first column identifies a test batch. A test batch is a set of test circuits that differ only in one property (number of input variables, number of product terms or size of On-set). It means, the test batch #1 is composed of three circuits having 4 inputs, 16 product terms and 25, 50 and 75% on-set size. Similarly for other test batches. All circuits have one primary output.

Test batch#	**Example circuits properties**		
	Input variables	***Product terms [count]***	***Onset [%]***
1	4	16	25, 50, 75
2	5	16, 32	25, 50, 75
3	6	16, 32, 64	25, 50, 75
4	8	16, 32, 64, 128, 256	25, 50, 75
5	10	16, 32, 64, 128, 256, 512, 1024	25, 50, 75
6	12	16, 32, 64, 128, 256, 512, 1024, 2048, 4096	25, 50, 75
7	16	16, 32, 64, 128, 256	25, 50, 75
8	4, 5, 6, 8, 10, 12, 16	16	25, 50, 75
9	5, 6, 8, 10, 12, 16	32	25, 50, 75
10	6, 8, 10, 12, 16	64	25, 50, 75
11	8, 10, 12, 16	128	25, 50, 75
12	8, 10, 12, 16	256	25, 50, 75
13	10, 12, 16	512	25, 50, 75
14	10, 12	1024	25, 50, 75
15	12	2048	25, 50, 75
16	12	4096	25, 50, 75
17	4, 5, 6, 8, 10, 12, 16	16, 32, 64, 128, 256, 512, 1024, 2048, 4096	25
18	4, 5, 6, 8, 10, 12, 16	16, 32, 64, 128, 256, 512, 1024, 2048, 4096	50
19	4, 5, 6, 8, 10, 12, 16	16, 32, 64, 128, 256, 512, 1024, 2048, 4096	75

Figure 6.1: Specification of circuits properties.

In order to verify the proposed method, circuits from figure 6.1 has been passed to proposed algorithm described in the previous section. Achieved results were compared to Espresso algorithm implemented in SIS synthesis tool. Each test batch has a direct relation to a batch in 6.1 and numbers are averaged over particular a changing property. Espresso synthesized circuit have two outputs in order to share common logic and all circuits may use only two input gates. Figure 6.2 reports experimental results of proposed methodology and comparison to Espresso synthesis. All columns, except *test batch* and *improvement*, report number of required 2-input gates. The column *improvement* denotes improvement of proposed method against Espresso in percentage.

Results from the table 6.2 are graphically illustrated in figures 6.3 and 6.4 depending on variable property. Number of 2-input gates was chosen as a metric in these experiments.

Test batch#	Polymorphic synthesis tool (averaged values)						Espresso – two outputs mode (averaged values)				Improvement [%]
	2-AND	*2-OR*	*INV*	*P_INV*	*P_MUX*	*sum*	*INV*	*2-AND*	*2-OR*	*sum*	
1	9	3	4	3	2	21	5	12	8	25	13,97
2	19	6	5	2	4	35	7	17	20	44	21,21
3	44	10	6	4	7	70	8	28	49	85	17,89
4	164	26	8	9	18	226	10	66	214	290	23,03
5	645	81	10	15	53	803	12	151	844	1008	22,76
6	2443	243	12	29	175	2903	14	409	3200	3623	24,38
7	**767**	**50**	**16**	**25**	**2**	**859**	**18**	**172**	**1016**	**1206**	**27,68**
8	60	6	8	7	2	83	10	35	58	103	19,67
9	124	13	10	9	3	159	11	41	155	208	22,77
10	275	27	10	15	6	333	12	71	367	450	24,84
11	599	55	12	20	12	698	14	112	838	963	27,30
12	903	88	11	21	36	1059	13	259	1243	1516	28,27
13	1747	183	11	22	102	2064	13	327	2477	2817	26,26
14	2616	287	11	58	197	3169	13	473	3583	4069	20,54
15	6067	610	12	24	499	7212	14	974	7787	8775	19,25
16	**7376**	**782**	**13**	**24**	**732**	**8927**	**14**	**1119**	**8902**	**10035**	**10,40**
17	678	62	10	15	39	804	12	145	1033	1190	28,69
18	1069	106	10	16	73	1276	12	216	1405	1633	25,08
19	1190	130	10	21	82	1434	12	207	1411	1630	15,82

Figure 6.2: Optimization results of method based on searching common parts applied on synthetic circuits.

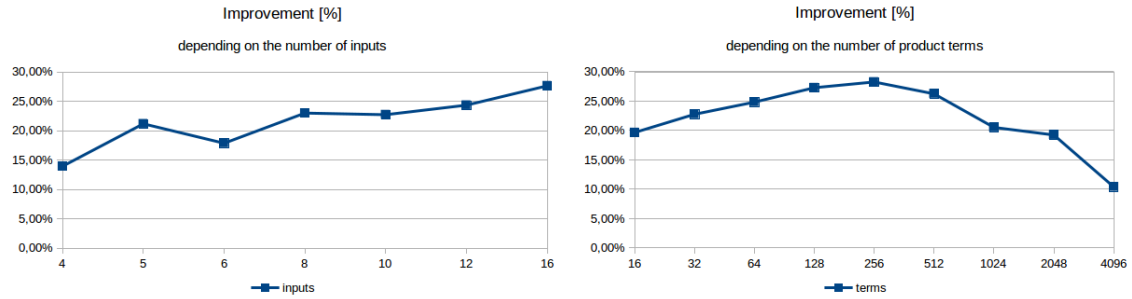

Figure 6.3: Graph on the left shows improvement depending on number of primary inputs. Graph on the right shows improvement depending on on-set size.

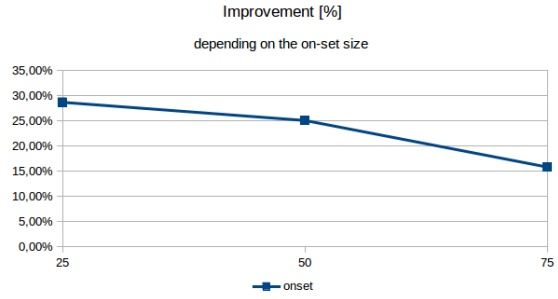

Figure 6.4: Improvement depending on number of product terms.

6.2.2 Experiments with MCNC benchmark circuits

The proposed synthesis method for a design of polymorphic circuits based on searching common parts has been tested on synthetic circuits summarized in figure 6.1. In addition, another experiments on real circuits defined by truth tables in two-level PLA format taken from MCNC benchmark set [121]. Circuits have been chosen with respect to the same number of input for one synthesis run.

Basic specification of these circuits can be found in table 6.2. Original names of circuits are in brackets in the table. Letters in brackets show which outputs are synthesized (all noted letters) and capital letters tell us which output is active while circuit works in mode one, or in mode two respectively. When there are no brackets, two different circuits are synthesized and polymorphism is responsible for switching between circuit function one and circuit function two. For example *rd84.pla (WxYz) / rd84.pla (wXyZ* means that WXYZ are output functions. In the first mode, the polymorphic circuit performs functions tainted to W and Y, while in the second mode, it performs functions tainted to X and Z. Basically, output functions W and X or W and Y are polymorphically switched depending on environment state.

Finally, results provided by polymorphic synthesis tool are shown and compared with results from conventional synthesis tool SIS [33]. With the aim of straightforward comparison, all circuits were built of 2-input gates only. The only exception in this context is an inverter. We have chosen a number of actually deployed 2-input gates as the main parameter for comparison. A percentage improvement over the conventional solution is noted in the last column of the table 6.2 as the number of used gates in polymorphic solution versus convectional solution. Number of used gates in circuits synthesized by polymorphic synthesis tool versus conventional synthesis by SIS are compared in a figure 6.5. Percentage improvement is plotted also in figure 6.5.

6.2.3 Related summary

In this section, a simple synthesis methodology using polymorphic multiplexers and polymorphic inverters was proposed. Obvious benefits obtained through a proper exploitation of polymorphic electronics offer significant improvement over conventional solution. A method based on a formal basis has been formulated. The obtained results, performed on randomly generated circuits, indicate that it's possible to achieve around 27% improvement especially in comparison to the standard synthesis tool called Espresso. A set of real experiments with complex circuits was performed in order to evaluate the proposed synthesis method. In one case it was possible to achieve almost 40% gates saving. Then, an average improvement on real benchmark MCNC circuits is about 20%. The method was validated on two-level circuits. However, an integration of the method into multi-level apparatus may be promising.

In order to further increase the synthesis efficiency of polymorphic circuits, next steps will explore, for example, the applicability of AIG graphs and rewriting technique for better identification of circuit parts that can be shared between two (or even more) functions subjected to the synthesis process.

no.			Polymoprhic synthesis tool						SIS 1.3.6 tool				Improvement
#	Circuit 1	Circuit 2	INV	2-AND	2-OR	P-MUX	P-INV	SUM	INV	2-AND	2-OR	SUM	[%]
1	rd84.pla (W)	rd84.pla (X)	8	896	92	93	8	1097	10	218	1232	1460	24.86
2	rd84.pla (X)	rd84.pla (Y)	8	896	1	1	2	908	9	518	512	1039	12.61
3	rd84.pla (W)	rd84.pla (Z)	8	600	91	72	0	771	10	364	574	948	18.67
4	rd84.pla (WxYz)	rd84.pla (wXyZ)	8	1448	92	75	10	1633	11	509	1302	1822	10.37
5	newtpla1.pla-X	newtpla1-Y.pla	4	15	1	1	1	22	5	26	5	36	38.89
6	9sym.pla	Z9sym.pla	9	447	85	77	11	629	9	860	85	954	34.07
7	t481.pla	ryy6.pla	16	4293	112	113	9	4543	17	4382	992	5391	15.73
8	ryy6.pla	newtag.pla	5	34	7	7	5	58	10	39	19	68	14.71
9	max46.pla	9sym.pla	9	549	46	47	8	659	18	779	130	927	28.91
10	rd73.pla	sqn.pla	7	749	65	49	9	879	19	293	669	981	10.40
11	sao2.pla (xWYz)	sao2.pla (XwyZ)	10	333	49	33	8	433	14	258	178	450	3.78

Table 6.2: Comparison of the results from the synthesis tool and the SIS 1.3.6 tool.

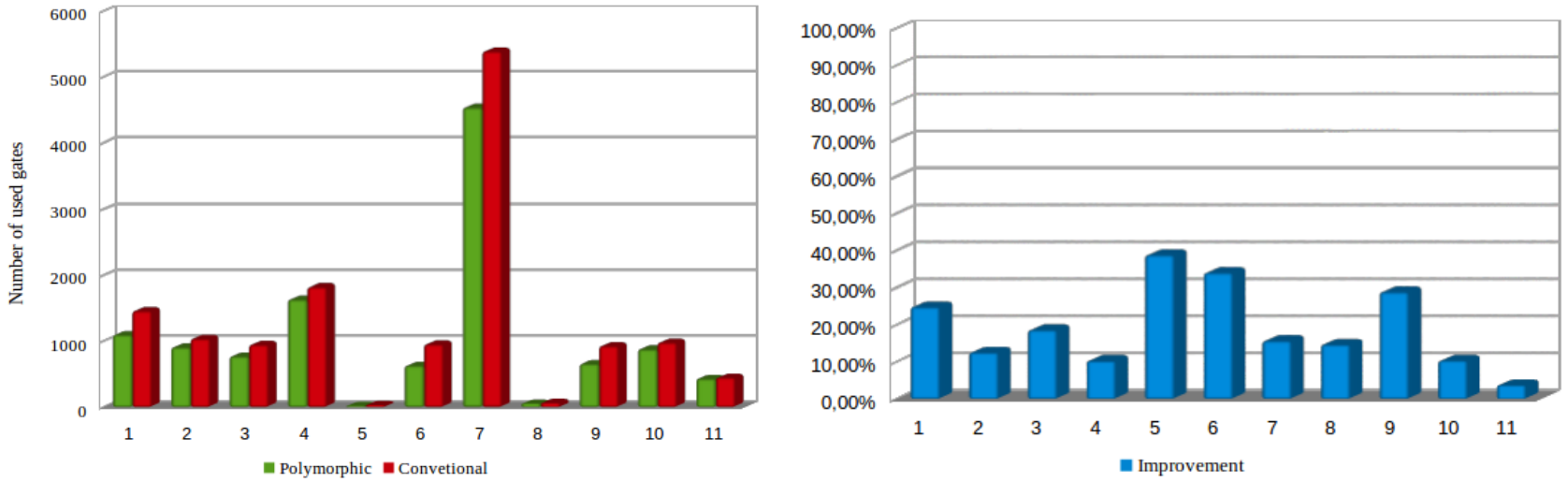

Figure 6.5: The graph on the left shows number of used gates synthesized by polymorphic synthesis tool versus conventional synthesis by SIS 1.3.6 tool. Numbers correspond to the table rows. The graph on the right shows percentage improvement of polymorphic synthesis in comparison to conventional synthesis by SIS 1.3.6 tool. Numbers correspond to the table rows.

Chapter 7

Proposed multi-level design and optimization method

In fact, a specification of logic function itself can appear in several, mutually different forms. An elaborate discussion on five of the most common description using two-level arrangement can be found in [119] and also in chapter 2.2. These cases are mostly focused on various representations of truth table forms together with disjunctive/conjunctive notation. For the sake of completeness it is important to point out that synthesis and minimization techniques in digital circuit domain are based extensively on multi-level representations as well [31, 42, 44], especially due to reasonable compromise between compact representation and efficient manipulation. Probably one of the most illustrative examples here is tied with AIG as a widely adopted scheme in logic optimizations.

Those minimization and synthesis techniques could be, as a matter of fact, roughly classified as two-level or multi-level oriented. In case of two-level methods the final circuit composition is delivered as a logic expressions in conjunctive or disjunctive notation. This approach then leads to the situation when input signals will only pass through two logic gates at most. On the other hand, multi-level techniques are generating so called nested expressions with the resulting data path (or interconnection of the individual gates) spanning even far more than two circuit elements within the final circuit arrangement.

The previous chapter discussed two-level optimizations of polymorphic circuits. Presented optimization techniques are applicable to two-level circuit representations. Literals are inputs and it is assumed that a final circuit is represented in the same way as a represented function. It leads to significant number of multi-input AND gates and one big OR gate. This fact may be a motivation to focus multi-level optimization methods. Multi-level representations are more realistic in the most cases.

In order to develop a method for multi-level logic optimization, a valid multi-level representation of logic circuit for an optimization algorithm is necessary. Multi-level circuit representations for ordinary logic circuit already exist, such as BDDs, AIGs or factored forms. It is assumed that a node can perform an arbitrary function and a number of literals can be significantly reduced. Unfortunately, similar descriptions for polymorphic circuits are missing.

It is a motivation to propose a multi-level representation of polymorphic circuits. This representation shall prepare a basis for optimization processes. Last years, AIG's are very popular representation useful for optimizations of ordinary circuits. Popularity of AIG's has

led to a focus the AIG representation and hence PAIG (Polymorphic And-Inverter Graph), which is described in following section, was introduced.

7.1 PAIG - An extension of AIG for polymorphic circuits

The problem outlined above, which is being pursued by this thesis, represents a challenge that stands out particularly in a situation when the circuit complexity is reaching beyond the boundary of more than just a few tens of gates. A key obstacle here is given by the fact that standard methods for representation of a typical digital circuit fail to adequately capture the specifics within the polymorphic electronics domain, which in turn renders their performance quite unsatisfactory. However, a solution leading ultimately to an inception of a novel format for representation of polymorphic circuits. The novel format could be identified in integration of corresponding extension into the foundation of conventional schemes like e.g. And-Inverter Graphs [68] or Binary Decision Diagrams [2].

Hence, the need to design a novel format for transparent representation of polymorphic circuits, which satisfies the requirements of easy manipulation in terms of synthesis and optimization tasks, clearly emanates from those aspects.

7.1.1 Elements of And-Inverter Graph scheme

One of the most ubiquitous schemes used for representation of a conventional digital circuit is known as And-Inverter graph (AIG) (see section 3). In fact, its core principle is based on an acyclic network of nodes and edges, where a node is two-input AND gate and an edge behaves like a negation of the logic signal passing through it between nodes. A significant advantage of using AIG concept for representation of a digital circuits is undoubtedly given by the fact that its foundation is relying on well-established graph theory. Hence this observation suggests a possibility to apply various operations that are generally known from graph theory also in case of AIGs. The continuous exploration and refinement of AIG with regard to its origins of theoretical background, which has been done already for more than a decade, brought a couple of advanced operations, which turn out to be very useful for digital circuits optimization.

Most of these operations have originated from research activities of Alan Mischenko [68]. For the sake of clarity, let's mention just some of them:

- Balancing: reduction of the depth.
- Structural hashing: detection of an isomorphic subgraphs.
- Functional reduction: detection of an isomorphic subfunctions in a graph.
- Rewriting: identification of ineffective parts and their replacement.
- and a rich set of additional operations.

And-Inverter graph offers modern, effective and transparent representation for necessary minimization operations requested by logic synthesis. For this reason, I decided to use AIG 's as a base for a new unique representation format of polymorphic circuits.

7.1.2 Toolset for operations with AIGs

The term AIGER denotes a format and, in the same time, also set of utilities for And-Inverter Graphs processing, which was developed at Johannes Kepler University in Linz. AIGER has been presented to the general audience at the Alpine Verification Meeting 2006 in Ascona [7]. The main idea was to provide a simple, compact file format for a model checking purposes. In fact, a specification of AIGER format [9] is available in two, slightly different versions: an ASCII and a binary. Each version is conceived with the aim to accommodate somewhat different purpose.

The ASCII version is the format of choice when it comes to AIG circuit representation, which needs to be saved for further reading and editing by a human circuit designer. It is simple to generate and it less constrained in comparison to the binary format. The binary version is, in fact, a compressed version of ASCII variant. Binary format saves data and is unreadable without corresponding AIGER reader. ASCII format requires more disk space, but it is readable by human.

Further within the context of this contribution the ASCII format will be considered only, especially because it gives better means for explaining the fundamentals of novel approach to the representation of polymorphic circuits.

Every circuit description file compliant with AIGER format should begin with one-line header, where the exact version of AIGER is given:

- „aig“ - binary identifier
- „aag“ - ascii identifier

This identifier is followed by 5 unsigned integers M I L O A which denote the following items respectively:

- M - maximum variable index
- I - number of inputs
- L - number of latches
- O - number of outputs
- A - number of ands

After this sequence of integer identifiers, the file format simply continues with an AIG description of a circuit structure. Now, each object (input, output, and, latch) has a unique numeric identifier. In order to keep a tidy arrangement, inputs are described first - each input (unique numeric identifier) is specified on a dedicated line:

```
<input identifier>
```

After the specification of all relevant input, latches can be placed. However, latches are not supported at the moment for polymorphic circuits, because this work focuses to combinatorial circuits only:

```
<latch input> <latch output>
```

Outputs must be specified in the same way as inputs:

```
<output identifier>
```

Finally, there still remains the need to specify the inner portion of AIG circuit representation consisting of AND-based nodes and their mutual links based on inverters:

```
<node identifier> <left leaf> <right leaf>
```

The following box contains an example of AIGER format in ASCII encoding [9]. The example is based on the previous instructions:

```
aag 6 2 0 2 2
2 #input 0
4 #input 1
8  #output 0
7  #output 1
6 2 4  #and node
8 3 5  #and node
```

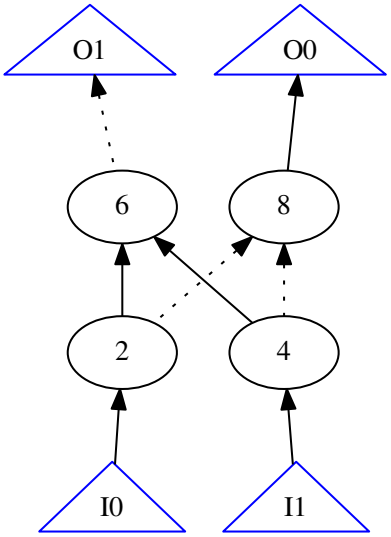

Figure 7.1: Representation of circuitry that contains two combinatorial functions, NAND - output O1 and NOR - output O2, using AIG paradigm.

It is possible to notice on figure 7.1 and example of AIG description given above, all object identifiers are **even**. It is required by internal implementation of AIGER interconnections, where even number is a wire. An inverting edge (dotted) is specified by **odd** object identifier (+1). For example an inverting edge from node 8 to node 2 will be noted as follows[1]:

```
8 3
```

7.1.3 Newly proposed AIG format for polymorphic circuits: PAIG

As it was concisely demonstrated in the previous section, AIGER and other widespread conventional tools and techniques do not offer, if any at all, an immediate support for the representation of polymorphic circuits. Hence, the need to design a novel format for

[1] Graphical representation of edges differs from commonly used notation. Arrows lead from leafs to roots.

transparent representation of polymorphic circuits, which satisfies the requirements of easy manipulation in terms of synthesis and optimization tasks.

Technical details of PAIG

The intention to develop new format for representation of polymorphic circuits was primarily motivated by the constraints and severe limitations of the available conventional methods. In addition, there was also an objective to get a compact format which may facilitate the tasks of synthesis and further optimization of circuit structure. A key decision was to preserve backward compatibility to AIGER format due to its widespread acceptance and relative simplicity.

With the aim to keep the complex nature of polymorphic circuit synthesis at a reasonable level, a number of permissible modes for each node within AIG representation was limited to the number of two. It means that the final polymorphic circuit can ultimately work in just two different operating modes, where an actual mode is switched by a state of the environment. In general, a resulting behaviour of the circuit built in AIG can be affected only by two aspects:

1. Interconnection.
2. Edges (wire or inverter).

One of the possible ways how to enhance the capabilities of AIGs involves definition of new edge types. Thanks to the polymorphism it is possible to change behaviour of gates and also inverters. AIG contains only AND gates, however, any other function can be built from AND gates and their appropriate interconnection. This idea has resulted into the extended variety of edge types - from the initial two types to total of four types now:

1. Normal edge - wire.
2. Inverted edge - inverter.
3. Polymorphic edge 1:
    - In mode 1 - wire,
    - in mode 2 - inverter.
4. Polymorphic edge 2:
    - In mode 1 - inverter,
    - in mode 2 - wire.

Black solid line represents a normal wire. Black dotted line represents an inverter. New polymorphic edges are denoted as a double solid line in the case of polymorphic edge 1 and as dotted double line in the case of polymorphic edge 2. See figure 7.2 for examples illustrating those types of edges. Finally, those four types of edges enable design arbitrary polymorphic circuit with two operating modes.

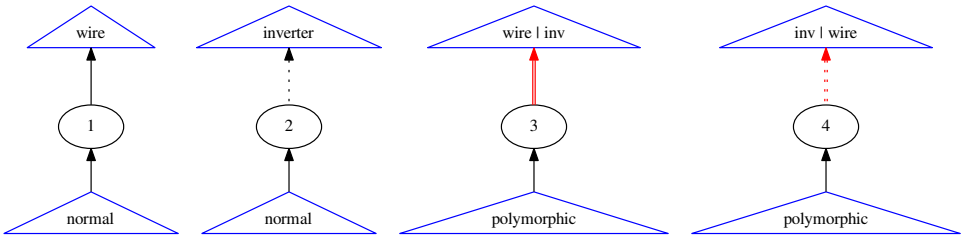

Figure 7.2: Graphical representation of edge types. Edge types from the left: wire, inverter, polymorphic wire, polymorphic inverter. Polymorphic wire is working as a normal wire in mode 1, while in mode 2 it assumes the function of inverter. Polymorphic inverter of is shown on the right side. In mode 1 it provides the functionality of inverter. Then, in mode 2, its behavior resembles wire.

At the beginning it is necessary to inform an AIGER parser about an intention to use the extended format. Format identifier in header must contain string „paag“. Then, the extension for polymorphic circuits will be correctly recognized:

- „paig“ - binary identifier for polymorphic AIGER (not supported yet),
- „paag“ - ascii identifier for polymorphic AIGER.

The ordinary AIGER format works with unsigned integers only. Even reference indexes are treated as „wires“ and odd are being seen as „inverters“. Extending AIGER to work with **signed integers** is necessary for the support of new edge types - polymorphic edges. Ordinary edges are staying unchanged, while the polymorphic edges have negative prefix before their object index. Following example highlights the proposed extension:

- Polymorphic edge 1 (mode 1 = wire, mode 2 = inverter) will be noted as **negative even** integer.
- Polymorphic edge 2 (mode 1 = inverter, mode 2 = wire) will be noted as **negative odd** integer.

```
paag 4 4 0 4 0
2 #input 0
4 #input 1
6 #input 2
8 #input 3
2 #output 0
5 #output 1
-6 #output 2
-9 #output 3
```

7.1.4 New constructions offered by PAIG extension

AIG extension of polymorphic edges has brought new constructions that may appear in a network. These constructions may bring new, more effective interconnections, but in other hand they require an attention during AIG manipulation.

One of the new construction that does not make sense in AIG, but in PAIG has significant importance, is connection of both edges from node a A to a node B. This construction has no meaning in AIG, because it propagates constant value, but in PAIG, it may change output dependently on polymorphic state. It is possible to insert a polymorphism into a circuit easily and this construction is also applicable inside a network. See figures 7.3 for better understanding.

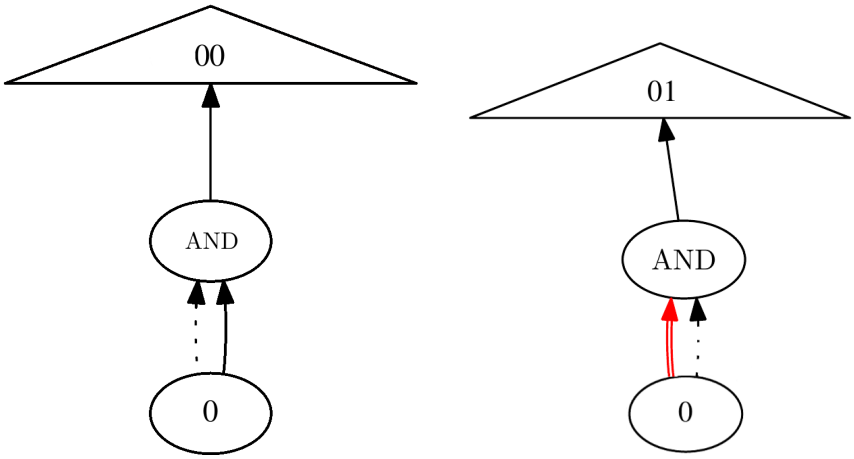

Figure 7.3: The left network shows an interconnection useless in AIG. However, the right network shows the same interconnection with polymorphic edge, that has significant usability in proposed PAIG networks.

This construction is further used for transformation of random primary input to virtual polymorphic input in performed experiments and evaluation. These constructions can appear inside a PAIG network also.

7.1.5 Experiments and demonstration

no.	Description		
#	Description	Circuit 1	Circuit 2
1	2-bit ALU	Logic ALU	Arithmetic ALU
2	2-bit Adder	SUM	Carry
3	Cellular tr.function	Rule 30	Rule 100
4	GRAY/BCD Coder	Gray	BCD
5	Self-checking adder	Carry	Carry

Table 7.1: Description of experimental circuits for PAIG extensions.

For the purpose of demonstrating properties of the newly proposed format for polymorphic circuits representation five different experiments were prepared in total (table 7.1). These experiments clearly show the efficiency of polymorphic circuits handling using new PAIG/PAAG format in comparison to the conventional solution based on standard AIGER format without additional modifications. Selection of benchmark circuits was random in order to clearly demonstrate the concept of PAIG representation.

no.	Conventional solution			Polymorphic solution		Improvement	
#	Circuit 1	Circuit 2	SUM	AIG	PAIG	Conv. vs PAIG.	AIG vs PAIG
	[ANDs]	[ANDs]	[ANDs]	[ANDs]	[ANDs]	[%]	[%]
1	9	7	16	18	10	44.44	37.50
2	4	6	10	13	7	46.15	30.00
3	4	4	8	13	6	53.85	25.00
4	16	7	23	26	16	38.46	30.43
5	4	4	8	8	4	50.00	50.00

Table 7.2: Comparison results of conventional AIG representations and PAIG representations.

Experiment no.1: Logic/Arithmetic ALU

The first experiment is dealing with a combination of two different ALU structures. The first ALU should be working in a logic mode and the second ALU should be working in an arithmetic mode. Conventional AIG implementation of these two ALUs is shown in figure 7.4. In this case, the logic operation mode of the ALU requires 9 AND gates and the arithmetic ALU consumes only 7 AND gates. See table 7.2 stating the overall number of used gates.

If an addition of polymorphism feature (switching of operating modes) is conceived using conventional AIG, then a mode switching is achieved thanks to an additional, dedicated virtual input labelled Mode. Let's call this solution „virtual polymorphism". Virtual polymorphism of ALU requires 18 AND gates. It is possible to see a graphical representation of this circuit variant in figure 7.5 on the left.

If the PAIG/PAAG format is used, it is possible to reach significant savings of resources. See figure 7.5 on the right. It shows a polymorphic ALU working in both logic and arithmetic mode, in which an operating mode is depends on a state of a target operating environment. This implementation is built upon the exploitation of new polymorphic edge types.

Experiment no.2: 2-bit adder

The second experiment is comprising a polymorphic adder, where Carry and Sum bits are switched in polymorphic way. In the first mode, adder calculates SUM, in the second mode, adder calculates carry. An adder circuit representation using conventional AIG scheme requires 10 AND gates. If the PAIG novel format is used, common resources are shared and whole the adder implementation requires only 7 AND gates. See table 7.2 for details.

Experiment no.3: Cellular transition functions

Another application of polymorphic circuits is demonstrated as a transition function of cellular automaton. This experiment executes two transition functions: Rule 30 and Rule 110. Rules are switched with regard to the operating conditions. Two conventional functions build from AIGs require 8 gates in total. Polymorphic AIG solution saves 2 gates while still preserving the original functionality. See table 7.2 for details.

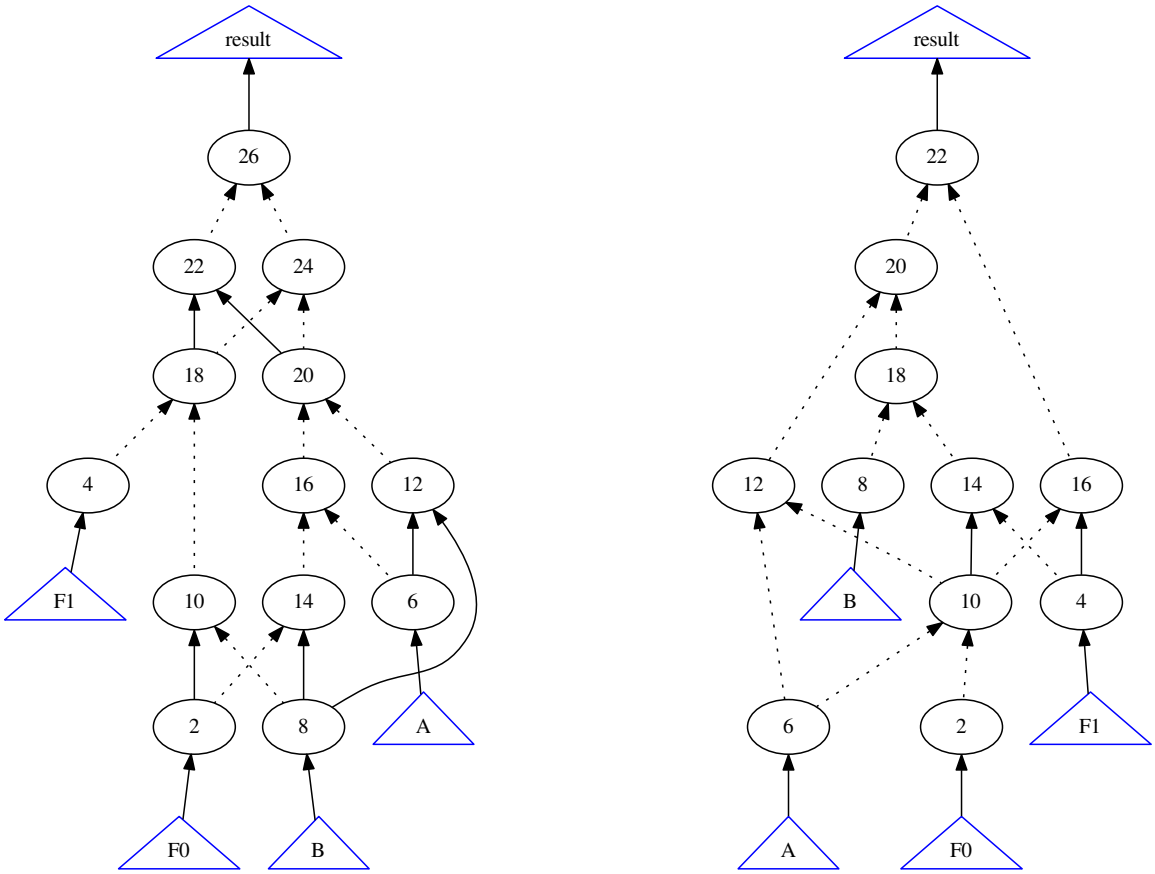

Figure 7.4: There is a conventional AIG of logic ALU on the left. There is a conventional AIG of arithmetic ALU on the right.

Experiment no.4: Gray/BCD decoder

This experiment combines two 4-input and 4-output circuits - Gray coder and BCD coder. Polymorphism switches between gray coding and BCD coding. A conventional solution of both circuits requires 23 AND gates in total. Virtual polymorphism needs 26 AND gates and a PAIG solution consumes only 16 AND gates. An improvement of the polymorphic solution reaches 38.46% in comparison to the conventional solution.

Experiment no.5: 2-Bit self-checking adder

The last experiment takes aim on a special kind of 2-bit adder with self-checking ability. This idea originated at Faculty of Information Technology, Brno University of Technology [90] as an illustrative example of polymorphic circuits application. Adder works equally in both polymorphic modes. An additional feature is hidden in fault detection mechanism. The adder calculates carry in the first mode. In the second mode, the adder performs the same - carry. If the carry is equal in both modes, no fault is present. But, if the carry is different in operational modes with respect the same inputs, an fault is signalized.

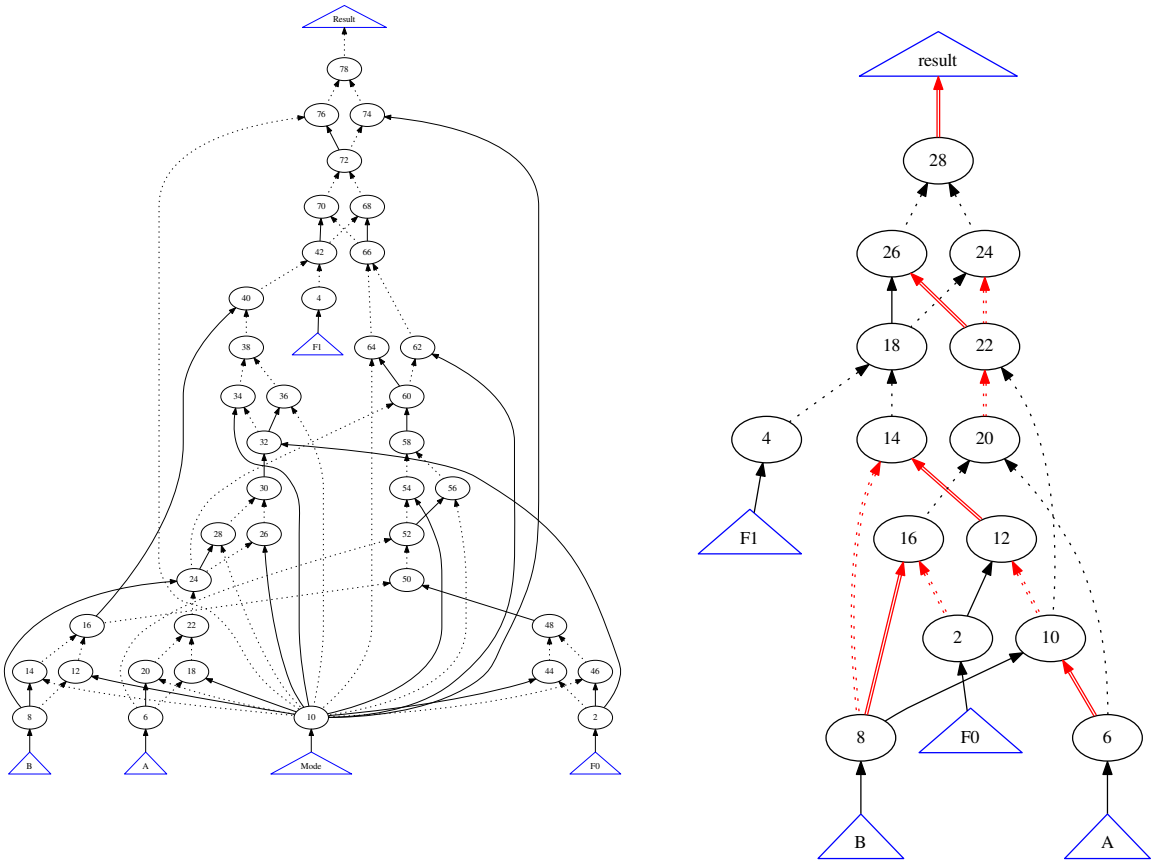

Figure 7.5: There is a conventional AIG of virtual polymorphic logic/arithmetic ALU on the left, in which the mode is controlled by virtual input labeled as MODE. There is a polymorphic AIG (PAIG) version of logic/arithmetic ALU on the right. Is possible to note significant savings of AND gates.

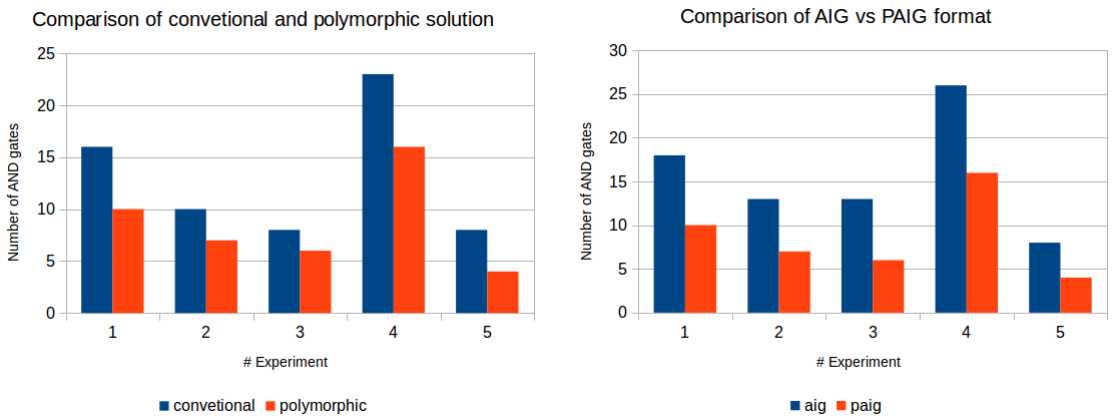

Figure 7.6: Left graph shows a comparison between two circuits synthesized as two separate circuits and polymorphic solution using PAIG. Right graph shows a comparison between AIG and PAIG, where both are representing a polymorphic circuit.

PAIG extension evaluation

The table 7.2 summarizes results of all conducted experiments outlined in the previous section. The table is separated into two main columns: Conventional solution and polymorphic solution. The conventional solution contains number of AND gates used with conventional technology and AIG representation. The polymorphic solution contains number of ANDs required by polymorphic technology. An AIG column contains number of ANDs required in case of using virtual polymorphism (basic AIG representation). A column PAIG contains number of used ANDs in PAIG/PAAG representation. The third column shows an improvement between conventional solution vs polymorphic solution in a percentage ratio and also the comparison of virtualized polymorphism to PAIG/PAAG representation. The results demonstrate, that the proposed PAIG/PAAG representation scheme can provide average saving of 34.59% AND gates (see figure 7.6 for details).

This evaluation presents the novel PAIG/PAAG representation on relatively small circuits. A main intention of the evaluation is clear explanation of PAIG/PAAG representation principles. More complex circuits are evaluated later, in chapter 8.

7.1.6 PAIG extension summary

A novel format for representation of polymorphic circuits using AIG was proposed. It is an extension of AIGER format, which is fully supported in well known tools like ABC and others. A few experiments show that the novel format seems to be very effective approach how to represent polymorphic circuits, including very complex variants (complex variants discussed in chapter 8).

7.2 Polymorphic AIG Rewriting

In comparison to conventional circuit design, designing of polymorphic circuit is much more complex. It is mainly caused by an ability to change behavior of building elements, while interconnection remains the same. Many designing methods have been proposed (see section 5.1), but most of them are two-level, or they have a scalability problem. Two-level polymorphic design methods were already published [25, 24], but a scalable, multi-level and straight methodology is still missing. Considering the AIG as a very popular concept for conventional circuit design, I decided to create a polymorphic circuit design methodology based on AIG.

In order to synthesize and optimize multi-functional circuits by means of using a scalable methodology based upon a formal foundation, a polymorphic-AIG (PAIG) rewriting is proposed as a modification of original AIG Rewriting [68] and PAIG extension. Original AIG Rewriting is described in [68] in detail or in this thesis, section 3.1.6. However, it is not applicable for synthesis and optimization of polymorphic circuits without further modifications of the former algorithm.

Basic idea of the original rewriting technique is based on replacement of AIG sub-graphs by optimal, smaller sub-graphs in order to reduce total number of gates. The algorithm proceeds iteratively from leaves to roots of an AIG and if a better solution is found, replacement is applied. Sub-graphs which are investigated have typically 4-inputs (K-feasibility K=4) and L outputs (L=1).

PAIG-based rewriting approach uses rewriting technique with a slightly different idea in mind. As it was mentioned in previous subsections, main intention of multi-functional

circuits is to take an advantage of sharing common logic of two completely different functions. At first, it is necessary to insert polymorphism (multi-functionality) into a circuit that is going to be synthesized. Two options open up here. The first option, an usual procedure implements two different functions in a conventional manner, so the polymorphic multiplexing is used as a seed of polymorphism. Polymorphic multiplexers are connected to primary outputs with respect to functions that are being switched. In PAIG, I designed two variants of polymorphic multiplexer, which are depicted in figure 7.7. Let's note that both variants consist of 3 AND gates. Adding polymorphic multiplexers creates full-fledged

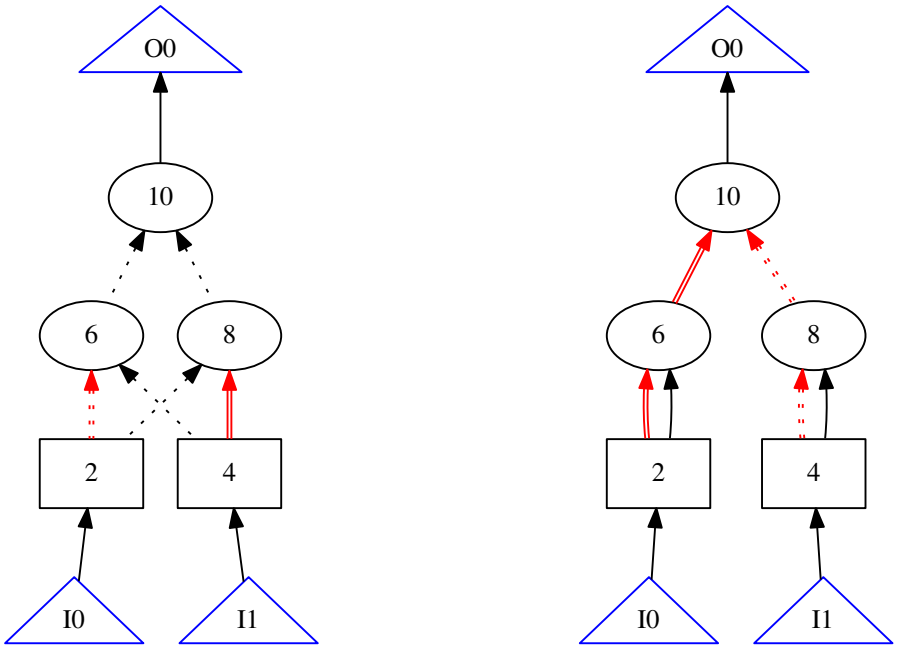

Figure 7.7: Two variants of polymorphic multiplexer represented in PAIG network.

polymorphic circuit with two functions. The second option is conversion of conventional circuit into polymorphic circuit by removing one primary input and making the primary input polymorphically driven.

However, sharing of common resources is not reached in either options, in contrary to the main objective here. So here comes the right moment for unleashing PAIG rewriting algorithm for optimization of polymorphic circuit in order to share common logic resources. In comparison to the original AIG rewriting, a few modification have been applied, such as support of PAIG representation, modified cut enumeration and generating optimal subcircuits. All these modifications are described in detail in the following sections.

7.2.1 PAIG rewriting algorithm

For optimization and synthesis of polymorphic circuits, a rewriting algorithm has been chosen for his popularity, quality and efficiency in conventional synthesis. Rewriting algorithm

is well described in [68], where it was introduced. Brief of conventional rewriting was already mentioned in section 3.1.6. In the recap, rewriting technique goes through all nodes in a network and all cuts of each node (see cut definition 6). Then, each cut is analyzed and replaced by its optimal implementation (note that all optimum 222 NPN-classes are pre-computed). The cut having the best gain of nodes is replaced.

Algorithm 1 PAIG rewriting algorithm

```
1:  procedure REWRITE(AIG, use_zero_gain)
2:      for each node N in AIG in the reverse topological order do
3:          C_best = NULL;
4:          gain_max = 0;
5:          for each 4-input poly cut C of node N do
6:              F = simulate_cut(C);
7:              C_optimal = Generate_optimal_subgraph(F);
8:              AIG_sand = copy(AIG);
9:              replace_cut(AIG_sand, C_optimal);
10:             AIG_sand = structural_hashing(AIG_sand);
11:             gain = AIG_num_gates - AIG_sand_num_gates;
12:             if ((gain > gain_max) ||
13:           ( (gain_max == 0) && (use_zero_gain) )
14:           ) then
15:                 C_best = C_optimal;
16:                 gain_max = gain;
17:                 polyedges_max = polyedges(C);
18:         if (C_best != NULL) then
19:             replace cut(AIG, C_best);
20:     return AIG;
```

Pseudo-algorithm of PAIG rewriting procedure is denoted in Algorithm 1. The Algorithm 1 describes a single rewrite iteration through a circuit C. For maximum efficiency of a synthesis process, it is recommended to run more than one iteration, until zero gain is achieved - this principle is valid for original rewriting also.

In comparison to the original AIG rewriting is iteration through AND nodes in the *reverse topological* order. The reverse iteration from roots to leaves will ensure propagation of polymorphism deeper into a network. To achieve maximum expansion of polymorphism into a network, in contrast to ordinary rewriting, PAIG rewriting is processing each node unless the gain of cut or whole network is negative.

PAIG rewriting algorithm is a key element behind the first synthesis and optimization method targeting multi-functional circuits which does not involve usage of any heuristic aspects at its core. All steps in the algorithm are strictly defined and the optimization process is fully controlled in comparison to evolutionary optimizations. The algorithm ensures polymorphism propagation deeper into the circuit structure and enables sharing of common sources of both desired functions.

7.2.2 Cut enumeration in PAIG

Cut enumeration completely follows the definition 6 and process of construction and recursive enumeration from section 1. DAG cuts and tree cuts are considered with respect to DAG-aware rewriting.

It is quite typical for a conventional rewriting procedure that 4-input (k = 4) cuts are commonly used. If k = 3, the chance to find a replacement with reasonable potential to improve the circuit representation is reduced significantly. If k = 5, a number of subgraphs grows rapidly instead. A 4-input polymorphic cut capable to switch between two functions require an additional 5th input. The additional input ensures the function switching and for purposes of this thesis the input is named *virtual polymorphic input*. The virtual polymorphic input in conjunction with 4-input cuts for the purpose of optimizing polymorphic circuits makes up altogether 5-input cuts. It is possible to observe a direct influence on the expansion of state-space comprising all permissible cuts.

Cut enumeration produces a set of all k-feasible cuts assigned to each node. Cut enumeration starts at leaves and continues in topological order to the root of AIG. Each node contains at least trivial cut. Each set of cuts of node n is computed as Cartesian product of two previous cut sets of nodes a and b. Formal notation of cut sets computation of node n is following [69]:

$$\phi(n) = \{\{n\}\} \cup \{u \cup v | u \in \phi(a), u \in \phi(b), |u \cup v| \leq k\}$$

Cartesian product of two sets creates a new cut set of node n, while keeping only K-feasible cuts.

7.2.3 Optimal circuit generator - MinCirc

Based on AIG rewriting principles discussed in section 3.1.6, the rewriting replaces non-optimal subgraphs by optimal. These optimal subgraphs must be somehow available. There are two options here: to have a precomputed library of optimal subgraphs or compute optimal subgraphs during runtime. For 4-input subgraphs and usage of NPN-class, we must have available 222 optimal structures, that are used for the ordinary rewriting algorithm (4-input cuts are used). Unfortunately, polymorphism adds additional virtual input - a polymorphic control wire, hence the polymorphic cut enumeration searches for 5-input cuts instead. Due to very high number of 5-input cuts (4-real input, 1-virtual input), a number of possible solutions is growing up at an extremely fast pace and, thus, it is rendering the option to keep all the pre-computed optimal cuts structures unrealistic. It simply becomes necessary to compute optimum cuts on-line.

Nan Li and Elena Dubrova proposed a technique for AIG rewriting using 5 input cuts. 5-input NPN equivalence classes circuits counts 616126, this number of graphs is too big to precompute and store. Authors experimentally discovered that only 2749 classes appear in all IWLS 2005 benchmarks [17]. Further they picked 1185 classes of 2749 with more than 20 occurrences and they generated best circuits for representative functions [55].

To generate an optimum circuit is $\sum_2 -complete$ problem [113] and this problem worries researchers since 1970's. Some methods have been introduced, for example methods based on ILP (Integer Linear Programming) or SAT based approaches (listed in [35]).

Although the generation process of optimum circuit implementation is well-mastered process in case of conventional circuits, no such approach hadn't existed for polymorphic circuits until introduction a MinCirc tool [35]. The MinCirc tool is a tool for generation of optimum circuits including polymorphic circuits using proposed PAIG format. MinCirc is

mainly used for an on-line computation of *optimum sub-graphs* in PAIG rewriting. Once the MinCirc produces optimum subgraphs (structures) for particular function, the optimum structure is chosen and deployed on the basis of achieved gain of overall network.

Because the generation process of optimum functions is essential for the rewriting algorithm, let us look under-hood the MinCirc. The initial version of MinCirc was mainly designed for generating optimum circuits with XOR gates [36], while later was extended for polymorphic circuits. MinCirc extension for polymorphic circuits exploits a property that polymorphism can be viewed as an additional primary input. Thus, a polymorphic stimulus P is introduced and enables (switches) between polymorphic modes. Basically, polymorphic stimulus P is a virtual polymorphic input mentioned in previous section 7.2.2.

An example of formula describing a polymorphic gate implementing a function AND/OR, is following:

$$F = \bar{P}(a * b) + P(a + b)$$

where P is the polymorphic stimulus (virtual polymorphic input) and a, b are primary inputs of gate.

Algorithm

To understand problematics of generation of optimum circuits, this short section briefs MinCirc algorithm, based on [35].

The problem of optimum circuit generation is solved by its reduction to a *decision CNF-SAT problem* [40]. These and similar problems belong to disparate complexity classes of polynomial hierarchy, the reduction is not polynomial. The optimization problem is elegantly limited by a simple trick applied to a decision problem : „Does there exists an n-node implementation of a given k-input function?“. Initial value of $n = 1$. If the answer is „no“, procedure is repeated with incremented n. The algorithm repeats until answer „yes“ is obtained. Then, it is a solution of the original problem.

The algorithm outlined by a pseudo-code in the Algorithm 2. The algorithm input is a truth table of the intended function and the output is an optimal structure. The key procedure in algorithm is *Generate_CNF()*. Detailed description of CNF generation can be found in [35] as well as experimental results.

Algorithm 2 MinCirc algorithm [35]

```
1: procedure GENERATE_OPTIMUM_STRUCTURE(truth_table f, int k)
2:     n = 1;
3:     do
4:         CNF = Generate_CNF(f, k, n);
5:         Sol = SAT_Solve(CNF);
6:         if (Sol.unsat) n++;
7:     while (Sol.unsat);
```

The MinCirc tool also offer to configure some parameters, such as required delay (depth), gate cost, etc. The MinCirc is deployed in PAIG rewriting for generating optimum subcircuits and completely covers *Genereate_optimal_subgraph(F)* in PAIG rewriting Algorithm 1.

7.2.4 Cut replacing

The original AIG algorithm performs *dereferencing* and *referencing* nodes during replacement and structural hashing is immediately applied per each node change. PAIG rewriting, implemented in the PAIG tool, removes nodes matching an inspected cut from a network and adds new nodes into a network corresponding to optimal sub-graph (generated by MinCirc). So modified network is structurally re-hashed at once. It may affect the performance of PAIG algorithm. Nevertheless, the complexity of original AIG rewriting algorithm is well-known. Cut replacement implementation can be improved later by referring the original AIG rewriting.

Chapter 8

Evaluation of multi-level polymorphic design and optimization method

In order to prove and evaluate the proposed PAIG rewriting algorithm, a set of experiments has been prepared for thorough evaluation. Experiments are divided into two groups, where deployment of polymorphic circuits make a sense. The first group of experiments rests in conversion of conventional circuits to polymorphic, and so converting one primary input to a virtual polymorphic input. The experiments may prove that circuits composed of polymorphic components may be solved more effective.

The second group of experiments consist in a joining of two different circuits and switching their output function using polymorphism. The main idea is to share common resources of two completely different circuit using polymorphism and demonstrate that polymorphic circuits saves resources.

Both kinds of experiments are working with the same batch of 21 combinatorial circuits from a publicly available benchmark set *LGSynth91 [121]*. Table 8.1 summaries the properties of combinatorial circuits used for experiments. Circuits in a set are chosen with a various number of AND nodes, high number of primary inputs and outputs in order to thoroughly evaluate proposed algorithm.

All the experiments presented in this section are successively performed in accordance to the synthesis flow previously discussed in Section 7.2.

In order to put the proposed solution and obtained results presented in this contribution into a proper context, an important aspect behind the experimental work takes aim at providing an illustrative comparison in terms of synthesis efficiency between polymorphic-based rewriting approaches against other convenient optimization methods exhibiting scalable properties. However, it is necessary to take into account an important fact that no easily scalable methodology for synthesis and optimization of polymorphic circuits has been reported up to the date.

8.1 Conversion of primary input to virtual polymorphic input

Experiment that demonstrates applicability of PAIG rewriting on conventional circuits is presented in this section. Conventional circuit is modified into a polymorphic circuit in the

Table 8.1: List of combinatorial circuits used for both kinds of experiments.

Index	Circuit name	Primary Inputs	Primary Outputs	AND nodes
1	cht.aig	47	36	185
2	apex1.aig	45	45	2604
3	apex6.aig	135	99	659
4	apex7.aig	49	37	221
5	lal.aig	26	19	109
6	c8.aig	28	18	169
7	misex2.aig	25	18	119
8	misex3.aig	14	14	1549
9	misex3c.aig	14	14	721
10	pcler8.aig	27	17	71
11	my_adder.aig	33	17	176
12	ttt2.aig	24	21	218
13	C499.aig	41	32	400
14	C1355.aig	41	32	504
15	seq.aig	41	35	2411
16	count.aig	35	16	127
17	unreq.aig	36	16	112
18	pdc.aig	16	40	1621
19	vda.aig	17	39	924
20	k2.aig	45	45	1998
21	rot.aig	135	107	550

following way: a random primary input is converted to a polymorphic driven one on the basis of environmental state.

The main objective of the proposed experiment is to produce an optimized structures of polymorphic circuits.

8.1.1 Specification of benchmark set

For demonstration purposes, all 21 polymorphic circuits has been selected from the table 8.1 for thorough evaluation, where circuit properties are also summarized. Each test case C consists of one combinatorial circuit from a publicly available benchmark set *LGSynth91* (listed in table 8.1). The set has been selected with various number of inputs PI and outputs PO. Selection of the individual test circuits used during the consecutive experimental evaluation did not reflect any further consideration or properties (e.g. signal propagation delay or interconnection complexity of a given circuit structure) than the aspect explicitly mentioned above. However, it could be interesting to assess the proposed approach behavior also from this standpoint during some of the future research activities.

Each test circuit C has number of primary inputs PI^C and number of primary outputs PO^C. One of PI^C is removed and substituted by virtual polymorphic input P in initial circuit, that cause the PI^C is controlled by the basis of environmental state. See figure 8.1 for more details of the conversion.

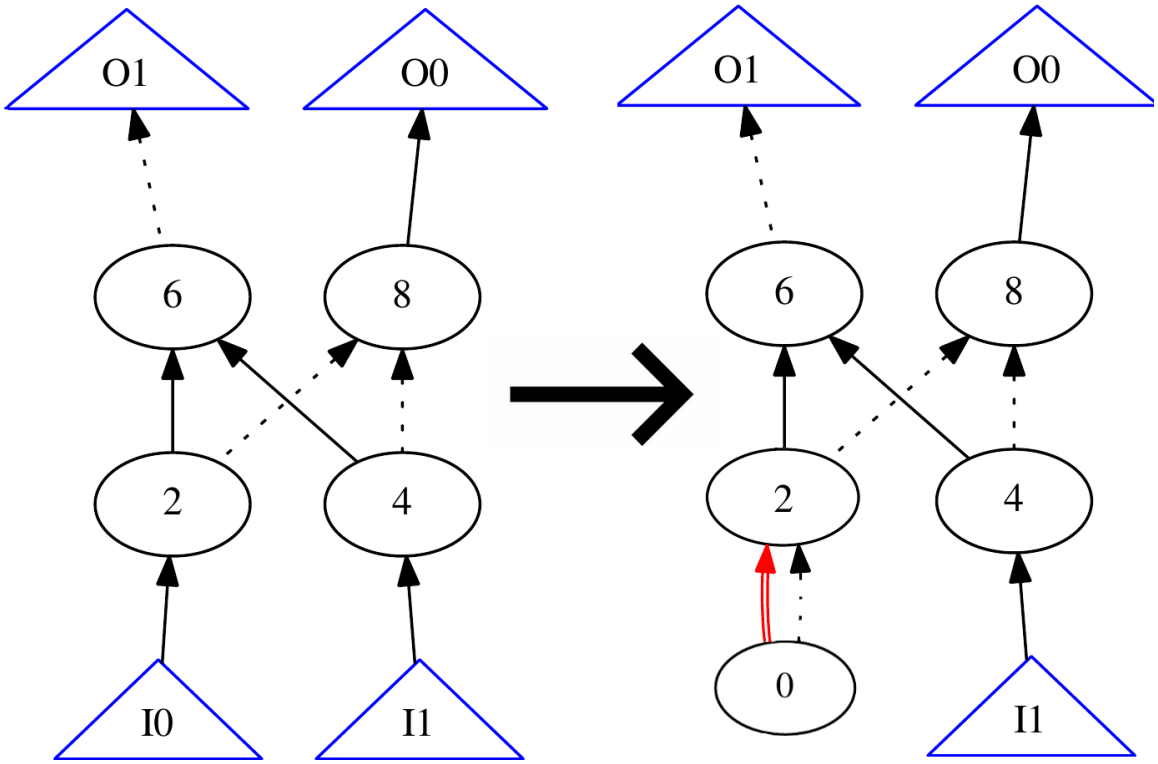

Figure 8.1: Example of primary input *I0* conversion to virtual polymorphic input.

8.1.2 Results analysis

Details on results obtained thanks to the polymorphic-oriented modification of conventional rewriting process are shown in table 8.2. The table itself is further divided into four sections, where the first one identifies the individual circuits from a given benchmark set. Then, the second one outlines the average results after optimization by PAIG rewriting that led to an observable circuit improvement. The third section reports an optimization results obtained with ABC tool and finally, the fourth one depicts the improvement PAIG rewriting comparison against ABC tool results. All results in the table 8.2 are averaged values of permutation over all primary inputs.

A column *num of rewrites* in second section reports average number of applied sub-circuit replacements per one iteration (one iteration = one rewrite command). A column *AND nodes before* represents a number of AND nodes of initial circuit. A column *AND nodes after* denotes the resulting number of AND gates required by a given target circuit once the synthesis process is finished. Analogically, a column *gain* reports a number of saved AND nodes and it is computed as subtraction of number ands after from number ands before. Both columns P_{edges} denotes the overall number of polymorphic edges used in that circuit. A column *avg cuts per it* reports average number of all found cuts per one iteration (one rewrite command). Next column *rewrite iterations* denotes a number of called „rewrite“ command to reach the best optimization. *Improvement PAIG* shows the details on a percentage improvement against the initial circuit. *Time* column contains an assessment of the elapsed time of the whole synthesis process including time for generation of optimal sub-circuit.

For the purpose of drawing a relevant comparison a contribution of standard ABC tool was used for an optimization of combinatorial circuits from every circuit under optimization given in table 8.1. With the aim to make optimization results more relevant, the optimization is performed especially for each primary input PI^C of each initial circuit and results are averaged. Thus, results report average values of optimization for all primary inputs.

The third section reports briefly maximal optimization reached by ABC tool, whereas several iterations has been applied too. The best optimization using ABC tool was achieved no later than in the in fourth iteration of the ABC „rewrite“ command. In summary, the optimization flow exploiting the conventional ABC-based rewriting methodology has resulted into the average reported improvement of 17.50% against the initial benchmark circuits arrangement. The last section contains only one column, reporting the difference of improvement of PAIG optimization against to ABC tool optimization.

An improvement or gain expressed in terms of AND nodes saving achieved by PAIG throughout all the experiments is reaching 21.02% in average. Further details mentioned here within the second section of table 8.2 give an overview of the proposed approach characteristics in situation when the objective was to achieve the best possible refinement of polymorphic circuits structure from the specified benchmark set. Efficiency of the PAIG-based rewriting algorithm is enumerated in column *Improvement PAIG*. The best derived solution (averaged per all primary inputs) has 34.76% improvement, which confirms the ability to design competitive multi-functional circuits.

The table 8.2 provides also a closer insight into the performance and runtime behavior (which is indeed a significant aspect when it comes to optimization of complex circuits) of the proposed approach in case of the chosen benchmark set, when the on-line computation of optimum sub-circuits (replacement cuts) is taken into account.

On-line optimum cut computation is a difficult task, which is managed by MinCirc tool within the proposed synthesis flow. In order to minimize the impact of that particular property exhibited by MinCirc, already generated PAIG graphs are reused without even launching the tool. The MinCirc produces optimum sub-graphs in a quite fast manner for the majority of functions (combinatorial circuits handed in to the synthesis process), however, some of functions are too difficult to be resolved in an acceptable time.

Therefore, a 5 seconds timeout period for MinCirc tool has been chosen. It is necessary to specify the timeout period in a cautious manner because its overly constrained value prevents the generation of good sub-graphs having an importance for circuit improvement. Tens of optimal subgraphs are usually generated in this period. On the contrary, too generous timeout significantly extends the overall duration of synthesis process. If the MinCirc runtime period expires for a particular function, the function is noted as difficult and MinCirc skips the function in the future with the aim to reduce the necessary synthesis time. Unfortunately, such timeout could potentially lead to non-deterministic behavior on a different computing platforms than the one actually used in that case. To get rid of this drawback, the future research activities could explore the possibility of creating the database of precomputed optimum circuits. The algorithm is expected to run just a few seconds with precomputed optimum cuts and effective PAIG code implementation. The experiments were performed using a workstation equipped with Intel(R) Core(TM) i7 CPU 920 processor.

Table 8.2: Optimization results for polymorphic rewriting.

average per inputs	num of rewrites	AND nodes before	AND nodes after	AND nodes gain	P_{edges} before	P_{edges} after	avg cuts per it	rewrite iterations	improvement PAIG [%]	time [s]	improvement ABC [%]	PAIG against to ABC [%]
cht	115,33	185	143,50	41,50	4,35	109,81	11 094,63	2,06	22,43	14,45	20,00	2,43
apex1	3 558,91	2604	2 105,56	498,44	41,53	394,02	1 308 036,86	5,40	19,14	6 861,38	19,12	0,02
apex6	392,48	659	621,80	37,20	3,71	55,20	61 165,19	4,98	5,64	306,56	2,88	2,76
apex7	168,54	221	192,33	28,67	3,50	37,00	11 582,29	3,42	12,97	305,05	9,50	3,47
lal	93,11	109	72,11	36,89	3,81	16,48	2 391,04	4,30	33,84	251,93	31,19	2,65
c8	327,31	169	110,21	58,79	6,52	59,62	7 904,00	6,28	34,79	221,14	28,99	5,79
misex2	86,58	119	99,08	19,92	4,81	22,31	2 946,73	3,54	16,74	187,57	19,33	-2,59
misex3	4 387,07	1549	1 240,40	308,60	81,80	440,13	677 783,33	8,13	19,92	12 319,53	18,40	1,52
misex3c	1 409,27	721	597,80	123,20	42,20	225,20	155 692,93	5,47	17,09	3 584,50	15,26	1,83
pcler8	325,89	169	110,25	58,75	6,14	59,57	7 910,18	6,25	34,76	3,65	9,86	24,90
my_adder	127,50	176	123,00	53,00	3,53	62,29	8 190,06	2,62	30,11	32,42	25,57	4,55
ttt2	207,32	218	164,60	53,40	7,56	45,76	9 597,48	4,12	24,50	633,52	23,85	0,64
C499	1 319,93	400	381,69	18,31	5,14	296,17	135 858,07	4,14	4,58	1 330,76	2,00	2,58
C1355	1 397,21	504	381,50	122,50	5,14	290,64	138 408,69	4,40	24,31	2 399,39	15,08	9,23
seq	3 145,46	2411	1 912,44	498,56	43,76	297,61	917 020,68	6,00	20,68	7 056,94	20,32	0,36
count	131,92	127	108,67	18,33	3,22	51,03	7 882,17	2,44	14,44	61,37	11,81	2,62
unreg	120,76	112	108,00	4,00	3,51	78,59	8 772,11	2,00	3,57	9,99	0,00	3,57
pdc	4 158,53	1621	1 076,18	544,82	75,82	365,59	549 684,82	8,71	33,61	13 569,84	29,61	4,00
vda	2 694,53	924	694,47	229,53	29,00	220,76	222 492,59	7,65	24,84	10 845,27	25,97	-1,13
k2	1 337,81	1998	1 384,84	613,16	6,19	67,62	284 169,27	7,32	30,69	3 128,60	29,93	0,76
rot	332,07	550	480,32	69,68	3,21	66,61	63 168,34	3,30	12,67	327,31	8,91	3,76
Average	1 230,36	740,2857143	576,61	163,68	18,31	155,34	218 654,83	4,88	21,02	3 021,48	17,50	3,51

8.2 Switching between two different functions

This section describes experimental results that demonstrate usability of PAIG rewriting for joining two different circuits into one polymorphic circuit, where their outputs are multiplexed on the basis of environmental state.

A main objective of the proposed approach is to produce an optimized structure of polymorphic circuits while mutual, non-conflict sharing of common resources between two initial circuits (input of the synthesis toolkit based on the proposed PAIG rewriting technique) is naturally ensured.

8.2.1 Specification of benchmark set

For experimental purposes, 15 polymorphic circuits has been selected from the the table 8.1 for thorough evaluation. Circuit properties are summarized in the table 8.3. Each test case C consists of two circuits A and B (listed in the table 8.1), where both of them have a similar number of inputs PI and outputs PO. Choice of individual test circuits or their mutual combination into a target circuit pair C used during the consecutive experimental evaluation did not reflect any further consideration or properties (e.g. signal propagation delay or interconnection complexity of a given circuit structure) than the aspect explicitly mentioned above.

Each test circuit C has a number of primary inputs $PI^C = max(PI^A, PI^B)$ and a number of primary outputs $PO^C = max(PO^A, PO^B)$. Since the primary inputs of circuit C are shared, the primary outputs $min(PO^A, PO^B)$ are connected using polymorphic multiplexers. See figure 7.7 for more details on implementation of polymorphic multiplexers. Remaining outputs are assumed to have permanent/constant function. Column $AND^{A,B,C}$ denotes the number of two-input AND nodes used in a given circuit. In case of circuit C, the initial number of AND nodes is following: $AND^C = AND^A + AND^B + AND^{pmux} * min(PO^A, PO^B)$, where $AND^{pmux} = 3$.

8.2.2 Results analysis

This subsection is divided into two parts: Results collected during the benchmark circuits processing by ABC tool and results collected during the benchmark circuits processing by PAIG tool.

ABC results

Similarly as previous experiment, for the purpose of drawing a relevant comparison, was the contribution of standard ABC tool used for optimizing A and B combinatorial circuits from every test case C given in table 8.3. Then, the optimized variants of both A and B combinatorial circuits are switched accordingly through the polymorphic multiplexer.

Results collected during the benchmark circuits processing by ABC tool are shown in a table 8.4). The table itself is further divided into three large sections, where the first one identifies the individual circuits from a given benchmark set. Then, the second one outlines the results after the first iteration that led to an observable circuit improvement and finally, the third one depicts the results when the synthesis process reached the best optimization level.

Each section of the table also provides the details on total number of AND nodes required by a given benchmark circuit C in three different situations (no optimization took place,

Table 8.3: Combination of combinatorial circuits used for evaluation PAIG rewriting by joining two different circuits. Polymorphic multiplexers are connected to primary outputs of both circuits. Thus P_{edges} represents number of polymorphic edges in polymorphic circuit to be optimized.

polymorphic circuit C	circuit A	circuit B	AND nodes	P_{edges}
C1	cht.aig	apex7.aig	514	144
C2	lal.aig	c8.aig	332	72
C3	misex2.aig	c8.aig	342	72
C4	pcler8.aig	c8.aig	291	68
C5	my_adder.aig	count.aig	351	64
C6	misex2.aig	lal.aig	282	72
C7	ttt2.aig	lal.aig	384	76
C8	ttt2.aig	misex2.aig	391	72
C9	lal.aig	pcler8.aig	231	68
C10	C499.aig	C1355.aig	1000	128
C11	count.aig	unreq.aig	287	64
C12	my_adder.aig	unreg.aig	336	64
C13	pdc.aig	vda.aig	2662	156
C14	apex1.aig	k2.aig	4737	180
C15	misex3.aig	misex3c.aig	2312	56

after the first iteration of ABC processing with some improvement, the best optimization result accomplished). Results shown in table 8.4 were obtained in the following way:

- optimized circuits were joined by polymorphic multiplexers connected to primary outputs in order to create polymorphic circuit,
- a rewrite command was issued on the input circuits A and B until any improvement at all,
- a final number of AND nodes has been counted (*column A+B+pmux*) and compared to the situation with initial circuits (*column Impr.*).

In summary, the optimization flow exploiting the conventional ABC-based rewriting methodology has resulted into the average reported improvement of 17.83% against the initial benchmark circuits arrangement.

PAIG results

Results achieved by the polymorphic rewriting are shown in table 8.5. The table itself is further divided into three large sections. The first one identifies the individual circuits from a given benchmark set. Then, the second one outlines the results after the first iteration that led to an observable circuit improvement and finally, the third one depicts the results when the synthesis process reached the best optimization level. A column AND_C in second and third section simply denotes the resulting number of AND gates required by a given target circuit once the synthesis process is finished. P_{edges} denotes the overall number of polymorphic edges used in that circuit. *Rwrts* shows the number of sub-circuit replacements. *Gain* reports number of saved AND gates and *Impr.* shows the details on a

percentage improvement against the initial circuit. *Runtime* column contains an assessment of the elapsed time in case of the first iteration that brought an observable improvement (second section of the table) and also of the whole synthesis process (third section of the table).

It is important to explicitly mention the fact that so called KL-cuts were enabled during the first iteration in order to get rid of the polymorphic multiplexers occurrence at the primary outputs. In general, KL-cuts offer an easier way how to find a proper cut (a sub-circuit eligible to be replaced by its optimized version) with polymorphic edges.

The table 8.5 provides a closer insight into the performance of the PAIG rewriting in case of the chosen benchmark set, when precomputed optimum cuts are taken into account. An improvement or gain expressed in terms of AND nodes saving achieved throughout all the experiments is reaching 23.00% in average after the first iteration with multi-output cuts option enabled. Further details mentioned here within the third section of table 8.5 give an overview of the proposed approach characteristics in situation when the objective was to achieve the best possible refinement of polymorphic circuits structure from the specified benchmark set. Efficiency of the PAIG-based rewriting algorithm is enumerated in column *Impr*. The best derived solution has 48.11% improvement, which confirms the ability of the proposed synthesis method to design multi-functional circuits while simultaneously trying to employ the principle of common resources sharing. Finally, it is possible to notice an average improvement of 25.95% across the whole benchmark set in comparison to the initial circuit C_n.

Table 8.4: Optimization results for conventional ABC rewriting.

Initial circuit			First rewrite iteration [AND nodes]					Rewrite iterating to the best [AND nodes]				
circuit C	A+B	A+B+pmux	A	B	A+B	A+B+mux	Impr [%]	A	B	A+B	A+B+pmux	Impr [%]
C1	406	514	148	201	349	457	11.09	148	200	348	456	11.28
C2	278	332	86	128	214	268	19.28	75	120	195	249	25.00
C3	288	342	100	128	228	282	17.54	96	120	216	270	21.05
C4	240	291	64	128	192	243	16.49	64	120	184	235	19.24
C5	303	351	131	112	243	291	17.09	131	112	243	291	17.09
C6	228	282	100	86	186	240	14.89	96	75	171	225	20.21
C7	327	384	179	86	265	322	16.15	166	75	241	298	22.40
C8	337	391	179	100	279	333	14.83	166	96	262	316	19.18
C9	180	231	86	64	150	201	12.99	75	64	139	190	17.75
C10	904	1000	394	444	838	934	6.60	392	428	820	916	8.40
C11	239	287	112	112	224	272	5.23	112	112	224	272	5.23
C12	288	336	131	112	243	291	13.39	131	112	243	291	13.39
C13	2545	2662	1172	712	1884	2001	24.83	1141	684	1825	1942	27.05
C14	4602	4737	2123	1474	3597	3732	21.22	2106	1400	3506	3641	23.14
C15	2270	2312	1278	616	1894	1936	16.26	1264	611	1875	1917	17.08
Avg:							15.19					17.83

Table 8.5: Optimization results for polymorphic rewriting.

Initial circuits	First rewrite iteration						Rewrite iterating to the best					
circuit C	AND_C	P_{edges}	Rwrts	Gain	Runtime [h:m:s]	Impr [%]	AND_C	P_{edges}	Iters	Gain	Runtime [h:m:s]	Impr [%]
C1	399	310	240	115	00:05:53	22.37	395	318	4	119	00:06:45	23.15
C2	239	166	145	93	00:02:14	28.01	235	178	4	97	00:03:10	29.22
C3	260	193	150	82	00:02:10	23.98	256	198	3	86	00:03:15	25.15
C4	223	158	128	68	00:02:02	23.37	151	151	3	140	00:02:42	48.11
C5	274	112	141	77	00:00:06	21.94	257	146	4	94	00:00:07	26.78
C6	228	137	130	54	00:02:30	19.15	224	144	4	58	00:03:35	20.57
C7	314	158	160	70	00:06:14	18.23	304	177	4	80	00:08:10	20.83
C8	308	123	156	83	00:06:34	21.23	300	133	4	91	00:08:19	23.27
C9	193	104	104	38	00:00:36	16.45	188	110	3	43	00:00:47	18.61
C10	711	580	759	289	00:02:04	28.90	709	507	3	291	00:05:35	29.10
C11	225	147	158	62	00:01:15	21.60	225	147	2	62	00:01:31	21.60
C12	241	160	207	95	00:01:21	28.27	241	160	2	95	00:01:27	28.27
C13	1979	258	773	683	01:49:11	25.66	1938	288	5	724	01:54:46	27.20
C14	3573	399	1329	1164	02:47:13	24.57	3536	423	6	1201	03:03:47	25.35
C15	1819	170	410	493	00:33:10	21.32	1803	163	5	509	00:36:29	22.02
Avg:						23.00						25.95

Experimental measurements have revealed that the largest chunk of computational time is consumed by PAIG code execution caused by ineffective implementation. But, from logic point of view, rewriting technique is very fast, while the code is written well. For example, ABC code is pretty well optimized and optimum cuts are stored in effective way. The experiments were performed using a workstation equipped with Intel(R) Core(TM) i7 CPU 920 processor.

For evaluation and comparison of PAIG rewriting algorithm efficiency with ABC conventional rewriting, it is possible to analyse in detail tables 8.4 and 8.5 respectively. In this way it is possible to recognize a significant advantage behind PAIG rewriting algorithm when it comes to the optimization of polymorphic circuits. A closer look will reveal the fact that the algorithm is able to achieve about 8.12% better optimization of polymorphic circuits than the conventional approach based on the utilization of ABC rewriting.

8.3 KL-cuts influence on PAIG rewriting

Referring to section 3.1.1, K-cuts are an efficient representatives of a region of an AIG, where the region has one output. Lets imagine a multiple output region. Such region would have to be covered by count of K-cuts. KL-cuts are novelty introduced in [59], that covers a multiple output region. It is supposed, that KL-cuts may help to find a more cuts in an AIG and thus offer to PAIG rewriting more paths for optimization of polymorphic circuits.

For this kind of experiment, a hypothesis was set: *Forced deployment of cuts with zero contribution to the optimization of a circuit structure during replacement stage allows to propagate polymorphism deeper into the circuit. Then, the utilization of KL-cuts can help to generate more comprehensive pool of cuts with the possibility to achieve improvement* > 0 *and, thus, perform synthesis of polymorphic circuits in terms of better area results.*

In this approach, KL-cuts are generated in the same way as K-cuts. However, KL-cuts are not dropped during the generation process. Generation process of KL-cuts does not conceal any other difficulty instead of expansive growth of generated solutions.

Despite the fact that the generation process of KL-cuts is quite straightforward, some complications still do emerge during the consecutive replacement process.

For the purpose of examining the influence of KL-cuts on the optimization efficiency of polymorphic circuits, the implementation of PAIG optimization tool was used. This section provides detailed overview of the experimental results obtained with this tool on set of benchmark circuits.

8.3.1 Specification of benchmark set

Benchmark set *(LGSynth91 [121])* has been chosen as the starting point for subsequent evaluation of the proposed PAIG-based rewriting scheme using KL-cuts at its core. The experiments were performed using 15 pairs of similar conventional circuits from that benchmark set. Detailed overview of selected circuits properties is provided in table 8.3. See section 8.2 for more details, because this experiment use the same benchmark set and initial circuit setup as experiments called „Switching between two different functions“.

The experimental part of my contribution presented in this chapter is closely related to the objective to confirm or deny the hypothesis formulated in section 8.3 that the utilization of PAIG-based rewriting with KL-cuts for the purpose of polymorphic circuit synthesis tasks is expected to deliver better results in terms of resulting area optimization. Thus all the

experiments are first conceived as a comparison of polymorphic rewriting algorithm with K-cuts to the variant which, on the contrary, involves KL-cuts. Second, the comparison is shown as an area improvement for both variants expressed in percentage value, where the attention is also given to the possible state space expansion in case of KL-cuts.

The main objective of polymorphic-aware rewriting is to produce an optimized structure of polymorphic circuits while mutual, non-conflict sharing of common logic resources between two initial circuits (the desired functions to be performed by the resulting circuitry) is accomplished.

Experiments are performed in two stages, with the assistance of PAIG tool implemented in C language. First of all, synthesis of circuits from the benchmark set is performed with the K-cuts. During the second round of experiments, the utilization of KL-cuts is allowed. Details of the initial circuits configuration are given in table 8.3.

8.3.2 Results analysis

Obtained results are depicted in table 8.6. The organization of the table summarizing the results is conceived with two main sections. Each of them is dedicated to a separate round of experiments in case of K-cuts and KL-cuts.

Each section in table 8.6 has the following sub-columns: Column *Iters* denotes how many iterations of polymorphic rewrite were used for a particular circuit *Cn* until it becomes resilient to further attempts of its optimization. Column *Tot.rewrites* counts the number of performed rewrites (replacements) as a sum of all iterations. Column *Ands* denotes a number of all gates (nodes) within the optimized circuit. *Gain* column contains information about number of saved gates in comparison to the initial circuit pair. *Impr.* column denotes how much area has been saved (number of nodes) using polymorphic rewrite algorithm in percentage value. This description is applicable for both experimental stages (first one with K-cuts and second one with KL-cuts enabled).

In addition, it is possible to notice also a third main section entitled *Influence* within table 8.6. Its sole purpose is to provide a comparison between using K-cuts and KL-cuts. A column *Impr.* in this section gives an account of the area improvement while KL-cuts usage is enabled in contrast to the situation with KL-cuts considered as prohibited. In other words, *Impr.* column illustrates the effect of KL-cuts onto the quality of resulting solution produced by the polymorphic rewriting optimization algorithm. At last, column *Growth* shows space explosion of investigated cuts in the case of enabled KL-cuts.

As it becomes apparent from a closer inspection of table 8.6, KL-cuts are undoubtedly helping to get more optimized polymorphic circuit structure in most of the situations, when the average improvement is reaching the level of 4.39%. The maximum value of the obtained area improvement of polymorphic circuit optimization using KL-cuts is 31.10 % against K-cuts rewriting for the circuit variant identified as *C10*. Whereas in the case of circuits *C11* and *C12* the utilization of KL-cuts did not have any impact on optimization at all. Please, refer figure 8.2, where both variants are graphically compared.

However, although the exploitation of KL-cuts brings only positive area optimization result, another important aspect deserves a further attention - explosion of cut set range (set of cuts to be investigated). State space explosion comprising different variants of cuts can be observed in the last column of table 8.6 and it is also depicted in figure 8.3. As it can be clearly seen, number of investigated cuts grows to 172.07% in average, where the maximum observed value of the state space growth is 459.64%.

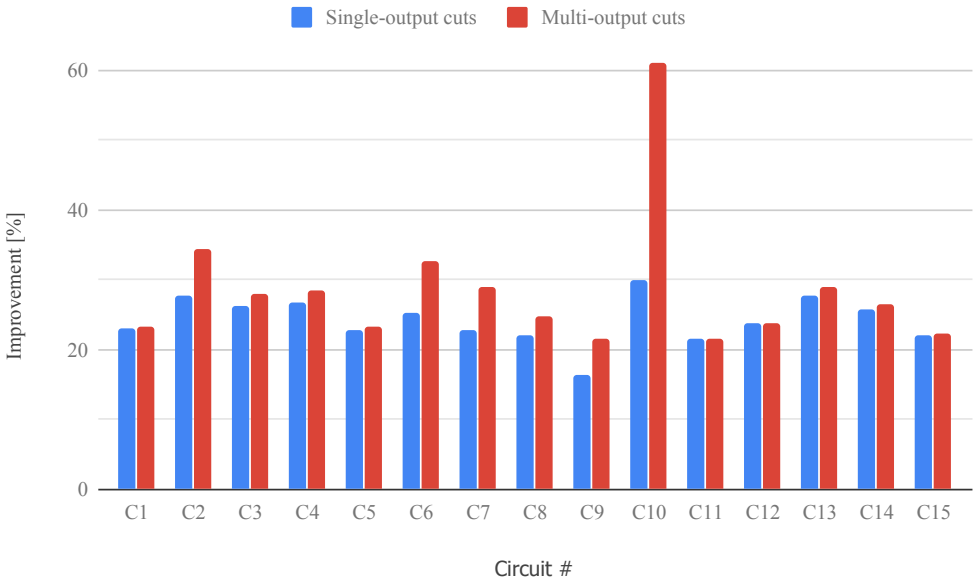

Figure 8.2: Graph shows the improvement in case of optimized polymorphic circuits: blue bars denote percentage improvement of circuits with K-cuts only and red bars denote percentage improvement of circuits with KL-cuts allowed.

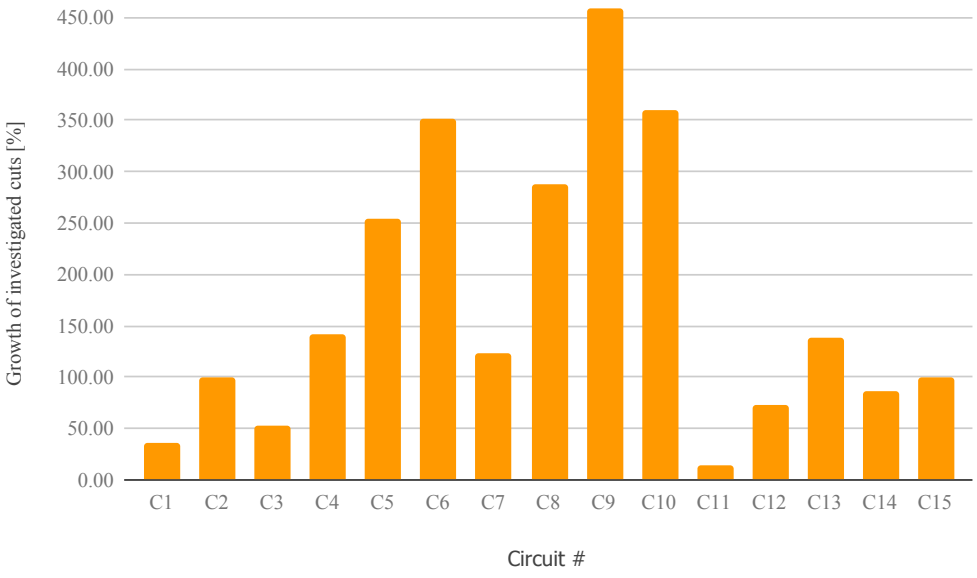

Figure 8.3: Graph reflects percentage growth of the number of investigated cuts for all chosen test circuit pairs when KL-cuts were allowed.

Table 8.6: Results of polymorphic circuit synthesis.

	K-cuts						KL-cuts						Influence	
circuit C	Iters	Tot.cuts	Tot.rewrites	Ands	Gain	Impr. [%]	Iters	Tot.cuts	Tot.rewrites	Ands	Gain	Impr. [%]	Impr. [%]	Growth [%]
C1	5	368410	180	395	119	23.15	5	503860	216	394	120	23.35	0.19	36.77
C2	8	193296	151	240	92	27.71	13	386958	280	218	114	34.34	6.63	100.19
C3	7	205296	162	252	90	26.32	9	315270	219	246	96	28.07	1.75	53.57
C4	5	127190	100	213	78	26.80	11	308011	236	208	83	28.52	1.72	142.17
C5	4	227904	128	271	80	22.79	8	806760	398	269	82	23.36	0.57	253.99
C6	11	161073	168	211	71	25.18	19	728137	622	190	92	32.62	7.45	352.05
C7	13	423280	342	296	88	22.92	16	941728	550	273	111	28.91	5.99	122.48
C8	10	293330	215	305	86	21.99	19	1139221	654	294	97	24.81	2.81	288.38
C9	4	67604	70	193	38	16.45	14	378336	334	181	50	21.65	5.19	459.64
C10	5	2438025	242	700	300	30.00	16	11207584	1489	389	611	61.10	31.10	359.70
C11	4	158692	130	225	62	21.60	4	180348	132	225	62	21.60	0.00	13.65
C12	4	211668	128	256	80	23.81	4	367064	191	256	80	23.81	0.00	73.41
C13	18	14216436	1359	1923	739	27.76	24	33897768	2389	1889	773	29.04	1.28	138.44
C14	19	41769448	2249	3516	1221	25.78	20	78111460	3209	3481	1256	26.51	0.74	87.01
C15	8	4593200	480	1803	509	22.02	14	9168838	838	1794	518	22.40	0.39	99.62
Average:						**24.29**						**28.67**	**4.39**	**172.07**

8.3.3 Related summary

This subsection was dealing with the investigation of a hypothesis that the usage of KL-cuts may lead to an improvement of polymorphic circuit optimization. The proposed approach is employed in a close conjunction with the PAIG-based rewriting algorithm. Especially due to the number of modifications that were introduced in case of polymorphic rewriting itself (e.g. forced replacement due to propagation of polymorphic edges deeper into the circuits structure) in comparison to the conventional rewriting variant, the hypothesis (see section 3.1.1) has been successfully confirmed. The obtained results clearly demonstrate a positive impact of the KL-cuts scheme, yet the improvement reach beyond the level of just a few percent.

8.4 Comparison of PAIG rewriting to PolyBDD

In order to compare PAIG rewriting results to the most famous methodology for polymorphic circuit design PolyBDD [37], the PAIG rewriting was applied to the same circuits that Gajda reports in his thesis.

Mr. Gajda has selected a number of test circuits for evaluation of PolyBDD method. Each polymorphic circuit is composed of two circuits performing independent functions. Gajda has used only polymorphic gates of type NAND/NOR and AND, OR, XOR, NAND, NOR, inverter and multiplexer gates. Unfortunately, his results report just overall numbers of required gates after synthesis by PolyBDD method, regardless of the price of used gates (especially XOR and multiplexer). An optimum implementation of XOR gate and 2-way multiplexer is consisting of 3 two-input AND gates. In table 8.7 in a column *PolyBDD*, numbers of required gates are reported, includes expensive XOR and MUX gates after PolyBDD synthesis. Figure 8.4 shows a PolyBDD structure (a) and corresponding polymorphic circuit (b) (note that circuit is composed of multiplexers mainly). The number of multiplexers in non-reduced BDD grows with power of 2 of primary inputs.

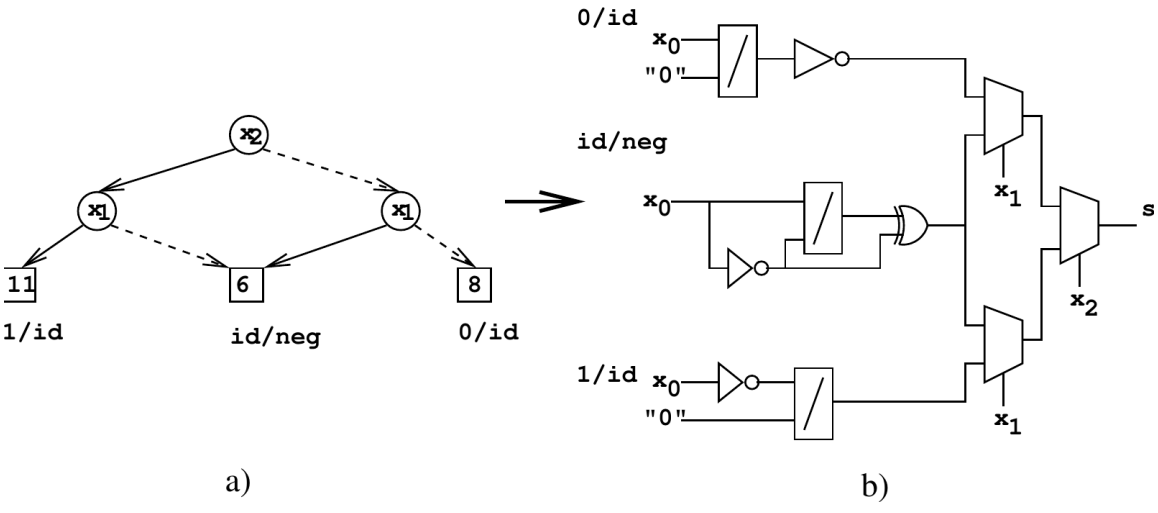

Figure 8.4: PolyBDD circuit (a) and corresponding polymorphic circuit (b) [37].

The proposed PAIG rewriting approach can handle only AND gates, stemming from the nature of AIG. Thus AIG cannot represent complex gates such as *XOR gate* and *multiplexer* natively. Despite that, one node type is not a disadvantage. It allows effective simple

optimizations. Simple AIG structures are mappable to complex target technology during technology mapping process. However, different metrics are appearing and comparison of PAIG to PolyBDD may be inaccurate.

Table 8.7 reports a comparison of PAIG to PolyBDD method. As it is outlined above, metrics of methods are different. The column *Circuit* denotes a name of a desired polymorphic circuit. M/S is Majority/Sorter, M/P is Majority/Parity and xA/xB is multiplier by A and B. The column *Inputs* contains number of primary inputs, analogically the column *Outputs* contain number of primary outputs. The column *PolyBDD* contains numbers of used gates after PolyBDD synthesis considering these types of gates: *{NAND/NOR polymorphic gate, AND, OR, XOR, NAND, NOR, 2-way multiplexer, inverter}*.

The next column *PolyBDD Evo.* contains a number of gates required after evolutionary optimization of PolyBDD synthesized circuits. Evolutionary optimization of PolyBDD circuit is performed by CGP, where are two-input elements only. Thus evolutionary results do not reflect complex multiplexers, that are often placed in PolyBDD circuits (see figure 8.4). Unfortunately, complex XOR gates are included, which may be a disadvantage for PAIG comparison. It is especially visible in the case of Majority/Parity circuits, that are mainly composed of XOR gates. However, the PAIG is still competitive, although XOR gate costs three 2-input AND gates.

The fifth column reports results of PAIG rewriting synthesis and values denotes a number of used 2-input AND gates. Analogically, the sixth column reports a number of required 2-input gates using conventional AIG (results from ABC tool).

A reader can compare efficiency of PAIG with PolyBDD by focusing columns *PolyBDD Evo.* and *PAIG*. *PolyBDD Evo.* includes complex XOR gates, that are not possible to express in PAIG structure natively.

Circuit	Inputs	Outputs	PolyBDD	PolyBDD Evo.	PAIG	AIG (abc)
M/S4	4	4	31	45	19	27
M/S5	5	5	50	71	43	49
M/S6	6	6	94	131	88	97
M/S7	7	7	150	212	162	170
M/S8	8	8	269	375	311	321
M/S9	9	9	428	697	602	614
M/P7	7	1	31	41	42	43
M/P9	9	1	41	60	60	61
M/P11	11	1	59	81	86	91
M/P13	13	1	73	114	114	115
x67/x127	7	14	228	274	187	215
x131/x251	8	16	430	547	441	468
x257/x509	9	18	348	410	279	310
x521/x1021	10	20	905	1028	865	894

Table 8.7: Comparison of PolyBDD and PAIG rewriting method.

Chapter 9

Conclusion

Logic synthesis and optimizations techniques are still popular topics despite the fact that they've been researched for at least 50 years. The need for further investigation of these topics is mainly related to growing complexity of digital circuits. Therefore, more effective and scalable synthesis methods are required. New research areas may be opened by emerging technologies or applications, such as multi-functional or polymorphic electronics. The concept of polymorphic electronics was introduced in 2001 [109], almost 20 years ago, and a non-evolutionary, scalable optimization technique still does not exist. Mentioned situation gave me an opportunity to start research of scalable synthesis and optimization methods for polymorphic circuits.

The main goal of this thesis was to propose an effective, scalable method for synthesis and optimization of multi-functional circuits. Initial research started with two-level design methods. The first proposed method, using only NAND/NOR gates, is well applicable to small circuits deployable to REPOMO32. The second method is based on boolean division and kerneling, which is suitable for detection of common parts of two desired circuits. Ongoing research has followed up on multi-level methods and brought new „PAIG" representation for polymorphic circuits in And-Inverter Graphs, which was very useful for further optimization. Since the polymorphic representation was designed, the AIG rewriting technique was adapted to work with the new PAIG representation. The whole synthesis and optimization process of polymorphic circuits clearly gives promising results.

9.1 Thesis contribution

The thesis contribution was partially mentioned in the previous paragraph. The thesis presents a proposal of synthesis and optimization methods for polymorphic circuits.

The first approach, using NAND/NOR gates (section 6.1) and dealing with the issues of multi-functional logic circuits synthesis, was introduced. The proposed synthesis method was based on a formal Boolean representation of corresponding input functions. Its main advantage can be recognized in its simplicity and an employment of boolean minimization techniques, which is in a direct contrast to existing solutions, predominantly based on heuristic approaches. Despite some constraints of the proposed approach, that were identified during the theoretical analysis and subsequent experiments, the method was successfully applied to real functions specified by the truth table. The obtained results clearly suggest benefits of the proposed approach in comparison the the conventional techniques. It

is safe to say that further improvements can be achieved, especially when new types of polymorphic circuit components based on the emerging materials will be prepared [22, 21, 78].

The second milestone was an introduction of kerneling based synthesis method (section 6.2). The method searches common parts of two desired circuits that were randomly generated. The obtained results indicate that it's possible to achieve around 27% improvement, especially in comparison to the synthesis tool called Espresso. Next experiment was performed on real-life complex circuits. Especially, in one case it was possible to achieve almost 40% gates saving. An average improvement on benchmark MCNC circuits is about 20% [117, 26]. Results were also published in Journal of Electrical Engineering [23]

In order to increase the synthesis efficiency of polymorphic circuits, a research was focused to an applicability of AIG and structural hashing for better identification of circuit parts that can be shared between two functions subjected to the synthesis process.

As a result, the novel polymorphic AIG representation format for polymorphic circuits was introduced (section 7.1). It is an extension of AIGER format, which is fully supported in well known tools like ABC. A few experiments showed that the novel format can be very effective representation of polymorphic circuits for future, more complex synthesis processes [27].

An innovative scalable methodology (section 7.2), called PAIG rewriting scheme, capable of synthesis of multi-functional circuits was introduced. The methodology is inspired by an existing rewriting algorithm [67, 68] that was used mainly for optimization tasks of conventional digital logic circuits. The PAIG rewriting offers strictly rigid, algorithm-based and scalable methodology, which is capable of producing valid results in a finite, predictable amount of time. More precisely, a defined background of the proposed approach to predict accurately the amount of time needed to obtain an acceptable solution. The PAIG rewriting method also contrasts to the state-space exploration (searching for the valid solution) involving, for example, various evolution-inspired techniques [28, 29].

The obtained experimental results indicate significant contribution in the field of synthesis of multi-functional circuits, that could potentially help to increase adoption of the polymorphic circuits for various application scenarios within the domain of multi-functional digital circuits. Research activities behind this contribution, including AIG extension for polymorphic circuits, open a new path for the synthesis of polymorphic circuits and create a stable basis for further research. This contribution may move the areas of synthesis and optimization polymorphic circuits forward significantly, mentioned in opened problems (section 4.2.2).

9.2 Future work

This thesis presents innovative design methods for multi-functional circuits. The thesis prepares a basis for a new direction in logic synthesis of multi-functional circuits in research area. Possible ways to continue this research are outlined in the following paragraphs:

A focus was given to verification of proposed principles mainly instead of development of optimized tools. Thus, a C code implementation of PAIG tool in order to speed up handling a graph may be the first task on the agenda.

Then, future work should be focused on exploration of the most frequent cuts, and preparation of an on-line available cut library in order to reduce the burden of time-consuming need to generate all the sub-graphs in an on-line manner, as proposed in [55].

Further focus on development of the proposed scheme should be aimed to technology mapping issues, i.e. translation from PAIG network structure to building components of a

target technology. PAIG network can represent conventional AND gates, wires, inverters and polymorphic wires only, but real circuits are not expressed entirely in this way. Real circuits are composed of more complex gates, such as XOR or even polymorphic complex gates and thus it is supposed that a mapping to a target polymorphic technology may further shrink an area of desired polymorphic circuits.

List of Figures

List of Tables

Bibliography

[1] Berkeley Logic Syntehsis and Verification Group: ABC: A System for sequential synthesis and verification.
Retrieved from: https://people.eecs.berkeley.edu/~alanmi/abc/

[2] Akers, S. B.: Binary Decision Diagrams. *IEEE Transactions on Computers*. vol. C-27, no. 6. June 1978: pp. 509–516. ISSN 0018-9340.

[3] Alexander, C.; Sadiku, M.: *Fundamentals of Electric Circuits*. McGraw Hill Higher Education. fourth edition. 2008. ISBN 9780071284417.

[4] Amaru, L.; Gaillardon, P.-E.; Micheli, G.: Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. vol. 35. 01 2015: pp. 1–1.

[5] Amarú, L.; Gaillardon, P.-E.; Micheli, G.: Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 06 2014. ISBN 978-1-4503-2730-5. pp. 1–6.

[6] Amarú, L.; Soeken, M.; Vuillod, P.; et al.: Enabling exact delay synthesis. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017. pp. 352–359.

[7] Armin, B.: Aiger format. In *Proceedings on Alpine Verification Meeting*. 2006. pp. 186–197.

[8] Ashenhurst, R.: The decomposition of switching functions. *Ann. Comp. Lab., Harward University*. vol. 29. 1959: pp. 74–116.

[9] Biere, A.; Heljanko, K.; Wieringa, S.: AIGER 1.9 And Beyond. Technical report. FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria. 2011.

[10] Bobda, C.: *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. The address: Springer. first edition. 2007. ISBN 978-1-4020-6088-5.

[11] Brayton, R.; McMullen, C.: The Decomposition and Factorization of Boolean Expressions. In *Proceedings of the ISCAS*. 1982.

[12] Brayton, R. K.; Mishchenko, A.: Sequential Rewriting. In *Proceedings of the IWLS 2007*. 2007. pp. 1–8.

[13] Brayton, R. K.; Sangiovanni-Vincentelli, A. L.; McMullen, C. T.; et al.: *Logic Minimization Algorithms for VLSI Synthesis*. USA: Kluwer Academic Publishers. 1984. ISBN 0898381649.

[14] Bryant: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*. vol. C-35, no. 8. Aug 1986: pp. 677–691.

[15] Chatterjee, S.; Mishchenko, A.; Brayton, R.: Factor Cuts. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*. ICCAD '06. New York, NY, USA: ACM. 2006. ISBN 1-59593-389-1. pp. 143–150.

[16] Cherepacha, D.; Lewis, D.: DP-FPGA: an FPGA architecture optimized for datapaths. *VLSI Design*. vol. 4. 01 1996.

[17] Christoph, A.: IWLS 2005 Benchmarks.
Retrieved from: http://iwls.org/iwls2005/benchmarks.html

[18] Correia, V. P.; Reis, A. I.; Porto, C.; et al.: Classifying n-Input Boolean Functions. In *Proceedings on the IWS*. 2001. page 58.

[19] Cortadella, J.: Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. vol. 22, no. 6. June 2003: pp. 675–685.

[20] Crha, A.: Polymorfní elektronika a metody syntézy. *Sborník příspěvků PAD2014*. vol. 2014, no. 1. 2014: pp. 57–62.

[21] Crha, A.; Růžička, R.; Šimek, V.: On the Synthesis of Multifunctional Logic Circuits. In *Abstracts Proceedings of International FLASH Conference*. Faculty of Electrical Engineering and Communication BUT. 2015. ISBN 978-80-214-5270-1. pp. 52–53.

[22] Crha, A.; Růžička, R.; Šimek, V.: Synthesis Methodology of Polymorphic Circuits Using Polymorphic NAND/NOR Gates. In *Proceedings on UKSim-AMSS 17th International Conference on Computer Modelling ans Simulation*. IEEE Computer Society. 2015. ISBN 978-1-4799-8713-9. pp. 612–617.

[23] Crha, A.; Růžička, R.; Šimek, V.: Novel Approach to Synthesis of Logic Circuits Based on Multifunctional Components. *Journal of Electrical Engineering*. vol. 67, no. 1. 2016: pp. 29–35. ISSN 1339-309X.

[24] Crha, A.; Růžička, R.; Šimek, V.: Toward Efficient Synthesis Method of Multifunctional Logic Circuits. In *Proceedings of the 27th International Conference on Microelectronics (ICM 2015)*. IEEE Computer Society. 2015. ISBN 978-1-4673-8759-0. pp. 21–24.

[25] Crha, A.; Růžička, R.; Šimek, V.: Synthesis Methodology of Polymorphic Circuits Using Polymorphic NAND/NOR Gates. In *Proceedings on UKSim-AMSS 17th International Conference on Computer Modelling ans Simulation*. IEEE Computer Society. 2015. ISBN 978-1-4799-8713-9. pp. 612–617.

[26] Crha, A.; Šimek, V.; Růžička, R.: Synthesis tool for design of complex polymorphic circuits. In *2017 12th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE Circuits and Systems Society. 2017. ISBN 978-1-5090-6376-5. pp. 149–154.

[27] Crha, A.; Šimek, V.; Růžička, R.: Towards novel format for representation of polymorphic circuits. In *13th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE Circuits and Systems Society. 2018. ISBN 978-1-5386-5290-9. pp. 1–2.

[28] Crha, A.; Šimek, V.; Růžička, R.: PAIG Rewriting: The Way to Scalable Multifunctional Digital Circuits Synthesis. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. Institute of Electrical and Electronics Engineers. 2019. ISBN 978-1-7281-2861-0. pp. 335–342.

[29] Crha, A.; Šimek, V.; Růžička, R.: KL-cuts influence on optimization of polymorphic circuits based on PAIG rewriting. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. Institute of Electrical and Electronics Engineers. 2020. ISBN 978-1-7281-9938-2. pp. 1–6.

[30] Curtis, H.: Generalized Tree Circuit—The Basic Building Block of an Extended Decomposition Theory. *Journal ACM*. vol. 10. 10 1963: pp. 562–581.

[31] Darringer, J.; Jr, W.; Berman, C.; et al.: Logic Synthesis Through Local Transformations. *IBM Journal of Research and Development*. vol. 25. 07 1981: pp. 272–280.

[32] Drechsler, R.; Becker, B.: *Binary Decision Diagrams: Theory and Implementation*. Springer US. 2013. ISBN 9781475728927.

[33] Ellen, S.; Luciano, L.; Alexander, S.; et al.: SIS: A System for Sequential Circuit Synthesis . *Electronics Research Laboratory Memorandum No. UCBERL M92/41*. 1992.

[34] Fiser, P.; Hlavička, J.: BOOM, A Heuristic Boolean Minimizer. *Computing and Informatics*. vol. 22. 01 2003: pp. 19–51.

[35] Fiser, P.; Simek, V.: Optimum polymorphic circuits synthesis method. In *2018 13th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*. 04 2018. pp. 1–6.

[36] Fišer, P.; Háleček, I.; Schmidt, J.: SAT-Based Generation of Optimum Function Implementations with XOR Gates. In *2017 Euromicro Conference on Digital System Design (DSD)*. 2017. pp. 163–170.

[37] Gajda, Z.: Evolutionary Approach to Synthesis and Optimization of Ordinary and Polymorphic Circuits . In *PhD thesis*. UPSY FIT VUT Brno. 2011.

[38] Gajda, Z.; Sekanina, L.: Reducing the Number of Transistors in Digital Circuits Using Gate-Level Evolutionary Design. In *2007 Genetic and Evolutionary Computation Conference*. Association for Computing Machinery. 2007. ISBN 9781595936974. pp. 245–252.

[39] Gajda, Z.; Sekanina, L.: On Evolutionary Synthesis of Compact Polymorphic Combinational Circuits. *Journal of Multiple-Valued Logic and Soft Computing*. vol. 17, no. 6. 2011: pp. 607–631. ISSN 1542-3980.
Retrieved from: https://www.fit.vut.cz/research/publication/9621

[40] Garey, M. R.; Johnson, D. S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman and Co.. 1990. ISBN 0716710455.

[41] Haaswijk, W.; Soeken, M.; Amarù, L.; et al.: A novel basis for logic rewriting. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017. pp. 151–156.

[42] Hachtel, G.; Somenzi, F.: *Logic Synthesis and Verification Algorithms*. 01 2006. ISBN 978-0-387-31004-6.

[43] Halecek, I.; Fiser, P.; Schmidt, J.: On XAIG rewriting. In *Proceedings of the IWLS 2017*. 2017.

[44] Hassoun, S.; Sasao, T.: *Logic Synthesis and Verification*. 01 2002. ISBN 978-1-4613-5253-2.

[45] Hayes, J. P.: *Introduction to Digital Logic Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.. first edition. 1993. ISBN 0201154617.

[46] Horowitz, P.; Hill, W.: *The Art of Electronics*. New York, NY, USA: Cambridge University Press. 1989. ISBN 0-521-37095-7.

[47] Háleček, I.; Fišer, P.; Schmidt, J.: Are XORs in logic synthesis really necessary? In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. April 2017. pp. 134–139.

[48] Ince, D. C. (editor): *Mechanical Intelligence (Collected Works of A. M. Turing)*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co.. 1992. ISBN 0-444-88058-5.

[49] Karnaugh, M.: The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*. vol. 72, no. 5. Nov 1953: pp. 593–599. ISSN 0097-2452.

[50] Karplus, K.: Using if-then-else DAGs for Multi-Level Logic Minimization. *Proceedings of Advanced Research in VLSI*. 09 1991.

[51] Kravets, V. N.; Kudva, P.: Implicit Enumeration of Structural Changes in Circuit Optimization. In *Proceedings of the 41st Annual Design Automation Conference*. DAC '04. New York, NY, USA: ACM. 2004. ISBN 1-58113-828-8. pp. 438–441.

[52] Lai, Y.-T.; Pedram, M.; Vrudhula, S.: BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis. 01 1993. pp. 642–647.

[53] Lavagno, L.; Martin, G.; Scheffer, L.: *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. USA: CRC Press, Inc.. 2006. ISBN 0849330963.

[54] Lee, C.: Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*. vol. 38. 07 1959.

[55] Li, N.; Dubrova, E.: AIG rewriting using 5-input cuts. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 2011. pp. 429–430.

[56] Li, Z.; Luo, W.; Yue, L.; et al.: Design Methods for Polymorphic Combinational Logic Circuits based on the Bi_Decomposition Approach. *CoRR*. 2017.

[57] Liang, H.; Luo, W.; Wang, X.: Designing Polymorphic Circuits with Evolutionary Algorithm Based on Weighted Sum Method. 2007: pp. 331–342.

[58] Liang, H.; Xie, R.; Chen, L.: Designing Polymorphic Circuits with Periodical Weight Adjustment. In *2015 IEEE Symposium Series on Computational Intelligence*. Dec 2015. ISSN null. pp. 1499–1505.

[59] Martinello, O.; Marques, F. S.; Ribas, R. P.; et al.: KL-Cuts: A new approach for logic synthesis targeting multiple output blocks. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. 2010. pp. 777–782.

[60] McCluskey Jr., E. J.: Minimization of Boolean Functions*. *Bell System Technical Journal*. vol. 35, no. 6. 1956: pp. 1417–1444.

[61] Micheli, G. D.: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education. first edition. 1994. ISBN 0070163332.

[62] Miller, J.; Job, D.; K. Vassilev, V.: Principles in the Evolutionary Design of Digital Circuits—Part I. *Genetic Programming and Evolvable Machines*. vol. 1. 04 2000: pp. 7–35.

[63] Miller, J. F.: *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2011. ISBN 978-3-642-17310-3. pp. 17–34.

[64] Mishchenko, A.: What is an And-Inverter Graph? In *The SIGDA Electronic Newsletter*. USA. February 2006.

[65] Mishchenko, A.; Brayton, R.: SAT-based complete don't-care computation for network optimization. 04 2005. ISBN 0-7695-2288-2. pp. 412– 417 Vol. 1.

[66] Mishchenko, A.; Brayton, R.; Jang, S.; et al.: Delay Optimization Using SOP Balancing. In *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '11. Piscataway, NJ, USA: IEEE Press. 2011. ISBN 978-1-4577-1398-9. pp. 375–382.

[67] Mishchenko, A.; Brayton, R. K.: Scalable Logic Synthesis using a Simple Circuit Structure. In *Proceedings on the IWLS 2006*. 2006.

[68] Mishchenko, A.; Chatterjee, S.; Brayton, R.: DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*. July 2006. ISSN 0738-100X. pp. 532–535.

[69] 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*. vol. 26, no. 2. Feb 2007: pp. 240–253. ISSN 0278-0070.

[70] Mishchenko, A.; Steinbach, B.; Perkowski, M.: An algorithm for bi-decomposition of logic functions. 02 2001. ISBN 1-58113-297-2. pp. 103– 108.

[71] Mishchenko, A.; Steinbach, B.; Perkowski, M.: An algorithm for bi-decomposition of logic functions. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. June 2001. ISSN 0738-100X. pp. 103–108.

[72] Mishchenko, A.; Wang, X.; Kam, T.: A new-enhanced constructive decomposition and mapping algorithm. 07 2003. ISBN 1-58113-688-9. pp. 143– 148.

[73] Morris, J.; Iniewski, K.: *Nanoelectronic Device Applications Handbook*. Devices, Circuits, and Systems. Taylor & Francis. 2013. ISBN 9781466565234.

[74] Měchura, T.: Random Circuits Generators.
Retrieved from: https://ddd.fit.cvut.cz/prj/Circ_Gen/index.php?page=pla

[75] N, C.; Manjunath, k.: *VLSI CAD*. PHI Learning Pvt. Ltd.. ISBN 9788120342866.

[76] Nageldinger, U.: *Coarse-grained reconfigurable architecture design space exploration.* 01 2001. ISBN 978-3-925178-65-8.

[77] Nevoral, J.; Růžička, R.; Mrázek, V.: Evolutionary Design of Polymorphic Gates Using Ambipolar Transistors. In *2016 IEEE Symposium Series on Computational Intelligence*. Institute of Electrical and Electronics Engineers. 2016. ISBN 978-1-5090-4239-5. pp. 1–8.

[78] Nevoral, J.; Šimek, V.; Růžička, R.: PoLibSi: Path Towards Intrinsically Reconfigurable Components. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. Institute of Electrical and Electronics Engineers. 2019. ISBN 978-1-72812-861-0. pp. 328–334.

[79] Pan, P.; Lin, C.-C.: A New Retiming-based Technology Mapping Algorithm for LUT-based FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. FPGA '98. New York, NY, USA: ACM. 1998. ISBN 0-89791-978-5. pp. 35–42.

[80] Quine, W. V.: The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*. vol. 59, no. 8. 1952: pp. 521–531.

[81] Quine, W. V.: A Way to Simplify Truth Functions. *The American Mathematical Monthly*. vol. 62, no. 9. 1955: pp. 627–631.

[82] R., Z.; Adrian, S.: Ripple Counters Controlled by Analog Voltage. In *NASA Tech briefs*. 2006. page 30(3):2.

[83] Rabaey, J. M.: *Digital Integrated Circuits: A Design Perspective*. USA: Prentice-Hall, Inc.. 1996. ISBN 0131786091.

[84] Rawski, M.; Jozwiak, L.; Łuba, T.: Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures. *Journal of Systems Architecture*. vol. 47. 02 2001: pp. 137–155.

[85] Raza, H.: Graphene Nanoelectronics: Metrology, Synthesis, Properties and Applications. . 2012.

[86] Riener, H.; Haaswijk, W.; Mishchenko, A.; et al.: On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019. pp. 1649–1654.

[87] Riener, H.; Testa, E.; Haaswijk, W.; et al.: Scalable Generic Logic Synthesis: One Approach to Rule Them All. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019. pp. 1–6.

[88] Roth, J.; Karp, R.: Minimization Over Boolean Graphs. *IBM Journal of Research and Development*. vol. 6. 04 1962: pp. 227–238.

[89] Růžička, R.: Gracefully Degrading Circuit Controllers Based on Polytronics. In *Proc. of 13th Euromicro Conference on Digital System Design*. IEEE Computer Society. 2010. ISBN 978-0-7695-4171-6. pp. 809–812.

[90] Růžička, R.; Sekanina, L.; Prokop, R.: Physical Demonstration of Polymorphic Self-checking Circuits. In *Proc. of the 14th IEEE Int. On-Line Testing Symposium*. IEEE Computer Society. 2008. ISBN 978-0-7695-3264-6. pp. 31–36.

[91] Růžička, R.; Šimek, V.: Chip Temperature Selfregulation for Digital Circuits Using Polymorphic Electronics Principles. In *Proceedings of 14th Euromicro Conference on Digital System Design*. IEEE Computer Society Press. 2011. ISBN 978-0-7695-4494-6. pp. 205–212.

[92] Růžička, R.: Polymorphic electronics. In *Habilitation thesis*. DCSY FIT Brno University of Technology. 2011. page 118.

[93] Sekanina, L.: Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware. *Lecture Notes in Computer Science*. vol. 2003, no. 2606. 2003: pp. 186–197. ISSN 0302-9743.
Retrieved from: https://www.fit.vut.cz/research/publication/7150

[94] Sekanina, L.: Evolutionary Design of Gate-Level Polymorphic Digital Circuits. vol. 3449. 2005: pp. 185–194.

[95] Sekanina, L.; Růžička, R.; Gajda, Z.: Polymorphic FIR Filters with Backup Mode Enabling Power Savings. In *Proc. of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE Computer Society. 2009. ISBN 978-0-7695-3714-6. pp. 43–50.

[96] Sekanina, L.; Růžička, R.; Vašíček, Z.; et al.: REPOMO32 - New Reconfigurable Polymorphic Integrated Circuit for Adaptive Hardware. In *Proc. of the 2009 IEEE Symposium Series on Computational Intelligence - Workshop on Evolvable and Adaptive Hardware*. IEEE Computational Intelligence Society. 2009. ISBN 978-1-4244-2755-0. pp. 39–46.

[97] Sekanina, L.; Růžička, R.; Vasicek, Z.; et al.: Implementing A Unique Chip Id On A Reconfigurable Polymorphic Circuit. *Information technology and control*. vol. 42. 03 2013: pp. 7–14.

[98] Sekanina, L.; Salajka, V.: Towards New Applications of Multi-Function Logic: Image Multi-Filtering. In *Proc. of the 2012 Design, Automation and Test in Europe*.

European Design and Automation Association. 2012. ISBN 978-1-4577-2145-8. pp. 824–827.

[99] Šimek, V.; Růžička, R.: More Complex Polymorphic Circuits and Their Physical Implementation. In *Proceedings of the 20th Electronic Devices and Systems IMAPS CS International Conference*. Brno University of Technology. 2013. ISBN 978-80-214-4754-7. pp. 189–194.

[100] Smith, G. L.; Bahnsen, R. J.; Halliwell, H.: Boolean Comparison of Hardware and Flowcharts. *IBM Journal of Research and Development*. vol. 26, no. 1. Jan 1982: pp. 106–116.

[101] Soeken, M.; Amarù, L. G.; Gaillardon, P.-E.; et al.: Optimizing Majority-inverter Graphs with Functional Hashing. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. DATE '16. San Jose, CA, USA: EDA Consortium. 2016. ISBN 978-3-9815370-6-2. pp. 1030–1035.

[102] Soeken, M.; Amarù, L. G.; Gaillardon, P.; et al.: Exact Synthesis of Majority-Inverter Graphs and Its Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. vol. 36, no. 11. 2017: pp. 1842–1855.

[103] Stareček, L.; Sekanina, L.; Kotásek, Z.: Reduction of Test Vectors Volume by Means of Gate-Level Reconfiguration. In *Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*. IEEE Computer Society. 2008. ISBN 978-1-4244-2276-0. pp. 255–258.

[104] Steinbach, B.; Lang, C.: Exploiting Functional Properties of Boolean Functions for Optimal Multi-Level Design by Bi-Decomposition. *Artificial Intelligence Review*. vol. 20, no. 3. Dec 2003: pp. 319–360. ISSN 1573-7462.

[105] Stoica, A.; Zebulum, R.: Multifunctional logic gate controlled by temperature. In *NASA Tech Briefs*. California Institute of Technology. 2005. page 18. NPO-30795.

[106] Stoica, A.; Zebulum, R.; Keymeulen, D.: Polymorphic Electronics. In *Evolvable Systems: From Biology to Hardware*. Springer Berlin Heidelberg. 2001. ISBN 978-3-540-45443-4. pp. 291–302.

[107] Stoica, A.; Zebulum, R.; Keymeulen, D.; et al.: On polymorphic circuits and their design using evolutionary algorithms. In *Proc. of IASTED International Conference on Applied Informatics*. 2002.

[108] Stoica, A.; Zebulum, R. S.; Guo, X.; et al.: Taking evolutionary circuit design from experimentation to implementation: some useful techniques and a silicon demonstration. *IEE Proceedings - Computers and Digital Techniques*. vol. 151, no. 4. July 2004: pp. 295–300. ISSN 1350-2387.

[109] Stoica, A.; Zebulum, R. S.; Keymeulen, D.: Polymorphic Electronics. In *Proceedings of the 4th International Conference on Evolvable Systems: From Biology to Hardware*. ICES '01. Berlin, Heidelberg: Springer-Verlag. 2001. ISBN 3-540-42671-X. pp. 291–302.

[110] Tanachutiwat, S.; Lee, J. U.; Wang, W.; et al.: Reconfigurable Multi-function Logic Based on Graphene P-N Junctions. In *Proceedings of the 47th Design Automation Conference*. DAC '10. New York, NY, USA: ACM. 2010. ISBN 978-1-4503-0002-5. pp. 883–888.

[111] Tesař, R.; Šimek, V.; Růžička, R.; et al.: Polymorphic Electronics Based on Ambipolar OFETs. In *EDS 2014 IMAPS CS International Conference Proceedings*. Brno University of Technology. 2014. ISBN 978-80-214-4985-5. pp. 106–111.

[112] Turkyilmaz, O.; Clermidy, F.; Amaru, L.; et al.: Self-checking ripple-carry adder with Ambipolar Silicon NanoWire FET. 05 2013. ISBN 978-1-4673-5760-9. pp. 2127–2130.

[113] Umans, C.: The Minimum Equivalent DNF Problem and Shortest Implicants. *Journal of Computer and System Sciences*. vol. 63, no. 4. 2001: pp. 597 – 611. ISSN 0022-0000.

[114] Umans, C.; Villa, T.; Sangiovanni-Vincentelli, A. L.: Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. vol. 25, no. 7. July 2006: pp. 1230–1246. ISSN 0278-0070.

[115] Vingron, S.: *Switching Theory: Insight through Predicate Logic*. Springer Berlin Heidelberg. 2013. ISBN 9783662101742. 57-76 pp.

[116] Vingron, S.: *Switching Theory: Insight through Predicate Logic*. Springer Berlin Heidelberg. 2013. ISBN 9783662101742. 57-76 pp.

[117] Šimek, V.; Růžička, R.; Crha, A.: Toward Efficient Synthesis Method of Multifunctional Logic Circuits. In *Proceedings of the 27th International Conference on Microelectronics (ICM 2015)*. IEEE Computer Society. 2015. ISBN 978-1-4673-8759-0. pp. 21–24.

[118] Šimek, V.; Růžička, R.; Crha, A.; et al.: Implementation of a Cellular Automaton with Globally Switchable Rules. In *11th International Conference on Cellular Automata for Research and Industry, ACRI 2014*, *Lecture Notes in Computer Science*, vol. 8751. Springer Science+Business Media B.V.. 2014. ISBN 978-3-319-11519-1. pp. 378–387.

[119] Wakerly, J.: *Digital Design: Principles and Practices*. Prentice Hall Series in Computer Engineering. Prentice Hall. 1994. ISBN 9780132114592.

[120] Weber, W.; Heinzig, A.; Trommer, J.; et al.: Reconfigurable Nanowire Electronics - Enabling a Single CMOS Circuit Technology. *IEEE Transactions on Nanotechnology*. vol. 13. 11 2014: page 1020.

[121] Yang, S.: Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0. Microelectronics Center of North Carolina (MCNC). jan 1991.

[122] Zebulum, R.; Stoica, A.; Keymeulen, D.: A Flexible Model of a CMOS Field Programmable Transistor Array Targeted for Hardware Evolution. vol. 1801 of LNCS. 2000: pp. 274–283.

[123] Zhang, X.; Luo, W.: Evolutionary repair for evolutionary design of combinational logic circuits. *2012 IEEE Congress on Evolutionary Computation*. 2012: pp. 1–8.

[124] Zhang, X.; Luo, W.: Evolutionary design of polymorphic circuits with the improved evolutionary repair. In *2013 IEEE Congress on Evolutionary Computation*. June 2013. ISSN 1941-0026. pp. 2192–2200.