A bottom-up approach to multiple-level logic synthesis

A bottom-up approach to multiple-level logic synthesis for look-up table based FPGAs

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus, prof.dr. M. Rem,
voor een commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
maandag 29 september 1997

door
Franciscus Adrianus Maria Volf
geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
en
prof. Dr.-Ing. J.A.G. Jess

Copromotor: dr.ir. L. Jóźwiak

Druk: Universiteitsdrukkerij Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Volf, Franciscus A.M.

A bottom-up approach to multiple-level logic synthesis for look-up table based FPGAs / by Franciscus A.M. Volf. - Eindhoven : Technische Universiteit Eindhoven, 1997.
Proefschrift. - ISBN 90-386-0380-0
NUGI 832
Trefw.: logische schakelingen ; algoritmen /programmeerbare logische schakelingen / combinatorische logica / digitale geintegreerde schakelingen.
Subject headings: logic CAD / field programmable gate arrays / combinational circuits / minimisation of switching nets.

Abstract

Despite the fact that the automatic hardware synthesis of complex digital circuits and systems has been an important topic of research during the last two decades, many synthesis problems are still not solved in a satisfactory way. This thesis deals with one of these problems: the multiple-level synthesis of Boolean functions for implementations with look-up table based Field-Programmable Gate Arrays (FPGAs).

The topic addressed in this thesis is very important for many reasons. Perhaps, the two most important reasons are the following. Firstly, FPGAs have a number of important advantages compared to the other implementation techniques, and therefore the use of look-up table based FPGAs for the implementation of digital systems has increased dramatically over the last few years. Secondly, due to the internal structure of FPGAs, traditional logic synthesis methods cannot be applied to them without extensive modifications and additions, and even modified they have some fundamental shortcomings.

In this thesis, an original approach to the general (functional) decomposition of Boolean functions into multiple-level networks of look-up tables is proposed. The approach has four main characteristic features. The first of them is the use of set systems that provides a compact maximal functionally complete representation of logic blocks and Boolean functions, whereas traditional logic synthesis methods use minimal functionally complete representations. Secondly, a compositional bottom-up approach is used for the synthesis; this is different from all other known functional decomposition methods that use a top-down reduction approach. Thirdly, the proposed method searches explicitly for common subfunctions, whereas other methods find subfunctions only implicitly as the by-product of the synthesis process. Finally, the method does not use two-level minimization prior to the actual multiple-level synthesis what enables the exploitation of the whole design freedom during the actual multiple-level synthesis.

Because of the computational complexity of the considered logic synthesis problem, it was necessary to develop some heuristic algorithms to make the synthesis method effective and efficient.

For the research of general decompositions a software environment called LUTSYN (*L*ook-*U*p *T*able *SYN*thesis) has been created. LUTSYN is specially designed to provide a lot of internal information, so the program can be experimentally used for research purposes, and especially for analysis and fine-tuning of all heuristic parts of the algorithms.

The primary goal of the synthesis process is to minimize the number of look-up tables that are used to implement a given Boolean function. Additional goals consist of minimizing the number of levels in the lookup table network (which is the main contribution to speed of the resulting circuit) and to minimize the number of long wires (what im-

proves both the routability and speed of the FPGA). The implemented algorithm allows the trade-off between the quality of the synthesis results and the computer resources used to find an implementation.
Although LUTSYN is not yet a complete automated synthesis tool, it can be used as a prototype automatic logic synthesis tool. Preliminary results show that the algorithm performs well compared to other known methods. A more complete analysis of the results will be available in the nearest feature.

Acknowledgement

I would like to thank prof.ir. M.P.J. Stevens and dr.ir. L. Jóźwiak for providing me the opportunity to perform this Ph.D. research and for their valuable advice and support while carrying out this work.

Furthermore I'm grateful to a number of people that have been involved in this research or provided assistance in a different way. In alphabetical order: Max Bonsel, Franck de Bruyn, Ad Chamboné, Artur Chojnacki, John van Houtert, Paweł Konieczny, Jan Kolsteren, Marcel Kolsteren, Wido Kruijtzer, Andy Ong, Rinus van Weert and Alex Wijffels.

A few people deserve extra credits: Wil Brant who single-handed created all the figures in this thesis, Jeroen Voeten for helping me to get all the mathematical details right and Rian van Gaalen for her continuous support and assistance.

Finally, I want to express my gratitude to my parents. Without their support writing this thesis would not have been possible.

Contents

List of Definitions

List of Theorems

List of Figures

Chapter 1

Introduction

The topic of this thesis is the logic synthesis for look-up table field programmable gate arrays. In this chapter the introduction to the subject and an overview of the thesis is presented.
The first part of this chapter (Section 1.1) will place the topic into context and will explain what field programmable gate arrays are.
In the second part of this chapter, a closer look will be taken at the topic of the thesis (Section 1.2). In this section the importance of logic synthesis in the design process is stressed and it is explained why a new method for logic synthesis, dedicated to look-up table field programmable gate arrays, is necessary. This section will also provide an overview of the work that has already been performed in this area.
In Section 1.3 the work performed in the scope of the reported Ph.D. project will be discussed and a number of characteristic features of the work will be presented.
The final section of this chapter explains the organization of this thesis.

1.1 Design of digital integrated circuits

Since the introduction of the first semiconductor devices in the early 1960s, enormous progress in integrated circuit technology has been made. Today it is possible to integrate several millions transistors on a single integrated circuit ("chip"). More than that, it has been predicted that the number of transistors on a single chip doubles every 2 or 3 years. This prediction has been proved to be valid for the past and is still valid in the 1990s (Figure 1.1)[Mic94]. This means that transistor densities of tens of millions transistors become available to mankind in the very near feature.
These numbers show immense possibilities, but also cause major problems. Advances in design theory and design technology have been unable to keep up with the possibilities offered by integrated circuit technology. Designers have to deal with the lack of general purpose specification methods that can be used to fully automate the synthesis of the circuit. CAD tool builders face the computational complexity that is inherent to most synthesis problems and are forced to develop efficient and effective heuristic algorithms.
Not only from the technical point of view research into new synthesis algorithms is important; also economical factors (e.g. bringing a product to the market on time is vital for a company to survive the competition with other companies) and the care for the

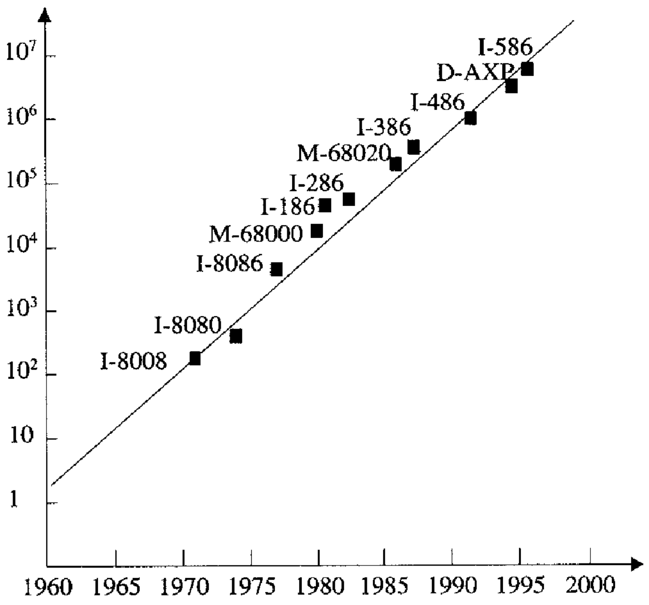


Figure 1.1: The number of devices placed on a single integrated circuit

environment (using as few as possible of the natural resources left on earth and limiting the environmental pollution) are important reasons to keep this research going.

Throughout the years, many different ways to fabricate integrated circuits have been developed. One of them, known as *prewired programmable integrated circuits* (or: programmable logic devices (PLD) has become very popular in the last decade. The feature of these prewired programmable integrated circuits is that they obtain their functionality by a programming mechanism. Programming can be performed by the end user without involvement of the manufacturer of the devices. This is a major advantage: fabrication times are extremely short (programming of a programmable logic devices often takes only a few minutes) and it has the additional advantage that proprietary information about the functionality of the device is kept in house. Simple programmable devices that can implement Boolean two-level sum-of-product or two-level product-of-sum expressions exist since the mid 1970s (PAL, PLA) [Phi90]. Unfortunately, these devices have a rather primitive architecture and are therefore unable to implement large and complex functions. Broad interest in programmable logic devices came with the recent introduction of new series of complex programmable logic devices; for examples device from Xilinx [Xil93], Actel [Act95], Altera [Alt95], [Atm95], Cypress [Cyp94] and AMD [AMD95].

The term *Field Programmable Gate Array (FPGA)* is used for a subset of the family of programmable logic devices. A field programmable gate array is a programmable logic device that consists of a matrix (array) of fine grain programmable blocks (gates) and of a programmable mechanism to interconnect these blocks. The Xilinx XC3000 and XC4000 families [Xil93] are typical examples of FPGAs.
If each programmable block of the FPGA consists of a small memory cell, then such a block is able to implement any Boolean function that does not exceed the number of inputs and outputs of the memory cell. FPGAs that consist of this kind of programmable blocks are called *look-up table based FPGAs*. For brevity, throughout this thesis, the abbreviation *LUT* will be used instead of look-up table. The blocks of the Xilinx XC4000 family [Xil93] can be programmed to be 4-input, 2-output LUTs or to be 5-input, 1-output LUTs [Tri93]. The blocks in the multiplexer architecture of Actel [Act95] can also be considered 2-input, 1-output LUTs [GHB93]. Altera FLEX FPGAs [Alt95] have basic cells that can be programmed as 4-input, 1-output LUTs.
The reader interested in the architectures of different FPGAs is referred to [RGS93] and [JP95] for good tutorials on this subject.

1.2 Logic synthesis

The design process starts with the recognition of a design problem and continues until the design problem is solved or it turns out that it is impossible to solve it given the available resources. It involves problem analysis and requirement specification of the required solution [Jóź95c][Jóź96a][Jóź96b].
Because of the complexity of the synthesis process, it is a common practice to decompose the design process into a number of consecutive synthesis steps. One of the synthesis

steps is called the *logic synthesis*. The term logic synthesis refers to all transformations in the design of digital hardware which are related to the implementation of (binary or symbolic) switching functions and sequential machines. Logic synthesis aims at transforming a (symbolic) description of a digital system modules into a network of (binary) logic building blocks. Such a network has to realize the behaviour of the module, satisfy specified constraints and optimize stated objectives [Jóź95b][JP95]. The logic synthesis considered in this thesis transforms a Boolean function (represented by a function table) into a network of LUTs. The goal of this section is to provide a sketch of the different logic synthesis techniques that have been used in logic synthesis in the past three decades and to show how the work reported in this thesis fits into this picture.

In this thesis, the term *primitive logic block* refers to any logic level function representation that can be mapped one-to-one onto a primitive hardware building block of a certain technology. A *primitive hardware building block* is the smallest logic level hardware unit that is used to implement binary functions in a certain technology. Examples of primitive hardware building blocks are the logic gates in the technology library for cell based integrated circuits or the LUTs for LUT based FPGA implementations.
Algorithms for logic synthesis of combinational Boolean functions can be divided in two groups: *two-level logic synthesis algorithms* and *multiple-level logic synthesis algorithms*. The word two-level refers to the fact that logic networks are constructed for which any path from input to output passes no more than two gates. Multiple-level networks do not have this restriction and inputs can pass an arbitrary number of gates before reaching an output. Typical two-level networks are AND-OR-NOT (or sum-of-product) networks [Kar53] [JB62] [BHMS84] [DAR86] [MSBS93], NAND-NAND or NOR-NOR networks, OR-AND-NOT (product-of-sum) networks and AND-EXOR networks [BS93].
The AND-OR-NOT representation is popular because it was used extensively to implement TTL-logic networks and it can be mapped efficiently on PLA structures. NAND-NAND and NOR-NOR implementations are equivalent to AND-OR-NOT implementations, NAND-NAND and NOR-NOT implementations were often used in TTL implementation styles, because they most naturally represented the basic elements used in this technology and can be mapped easily on physical hardware blocks. AND-EXOR representations cause interest because they often result in smaller expressions compared to AND-OR-NOT networks and they are extensively used in error correcting devices [BS93]. Two-level Boolean expressions and two-level logic synthesis techniques have been studied extensively. Although the problem of optimal two-level synthesis is NP-complete, reasonable exact and efficient heuristic methods exist for solving the two-level minimization problem [BHMS84][MSBS93][OCF93][Cou94].
Since 1980, interest in multiple-level logic synthesis has rapidly increased, because the multiple-level networks often allow a more compact implementation of combinational logic in comparison to the two-level networks. They enable the implementation of common subexpressions that are shared among multiple functions or subfunctions. The introduction of the new generation of FPGAs has recently created a new and very strong stimulus for research in multiple-level logic synthesis. The internal structure of the FPGAs is in fact a programmable multiple-level network and therefore these devices require the use of multiple-level logic synthesis techniques in order to effectively exploit their abilities. Unfortunately, the synthesis of general multiple-level networks is much

more complicated than the synthesis of two-level logic. One of the main reasons for this is the difficulty in defining the nature of the "optimal solution" in the multiple-level synthesis problem. For example, in the case of two-level AND-OR-NOT logic, the commonly considered "optimal solution" was the solution with the minimal number of product terms, which was a relatively good measure for the complexity of the PLA implementation (at least if the number of inputs could not be minimized). In multiple-level logic, the structure of the logic is less uniform and can be considerably more complex than two-level logic. Also, the design decisions have a much more substantial impact on the many factors that decide the total quality of a multiple-level logic network: area, speed, power dissipation, testability etc. Furthermore, these factors are no longer simple functions of the implementation structure as it was the case for two-level logic. Moreover, there is a trade-off between these factors. Finally, the design space is much larger, and in fact, includes properly the two-level logic networks as a special case.
During the last decade many different approaches have been proposed to solve special cases of the multiple-level logic synthesis problem. A good, but currently somewhat outdated overview of multiple-level logic synthesis can be found in [BHS90]; an overview of popular logic synthesis methods for a number of well known field programmable gate architectures can be found in [SGR93]. The most popular multiple-level logic synthesis approaches are the following:

- algebraic and Boolean division techniques [BRSW87] [RV92] [SFA93],
- multiple-level (binary decision diagram) BDD and other decision graph approaches [Bry86] [Bry92] [Kar90] [MF89] [PCSS95] [LPP96],
- algorithms based on the minimization of the communication complexity between blocks [HOI92],
- multiple-valued logic based approaches [MLBS92] [CYP89] [YC91],
- methods based on spectral analysis techniques (see [FSP92] for an overview),
- methods based on the iteration of gate transforms and gate reductions, the so called transduction methods [MKLC89].

In recent years a number of papers have been published which implicitly or explicitly use a concept of functional decomposition introduced by Ashenhurst [Ash59], Curtis [Cur61], Roth and Karp [RK62] as a synthesis paradigm (e.g. [RS87], [CY92] [JV92] [WP92], [ŁSK93], [HOIW94], [ŁS95], [VJ95], [LPP96]). The distinctive feature of these methods is that they are all based on special cases of the general decomposition theory formulated by Jóźwiak [Jóź95a] and recalled in Chapter 2.

In the general decomposition approach, a multiple-output Boolean function is decomposed into a network of communicating subfunctions (logic blocks) in such a way that this network realizes the specified behaviour, satisfies specified constraints and optimizes given objectives. Decomposition decisions are based on the analysis of the structure of the information streams in the function and on the relations between this structure and the specified constraints and objectives. The constraints imposed by hardware building blocks and their possible interconnections are innately taken into account.

The general decomposition approach has a number of interesting properties. For example synthesis methods based on general decomposition integrate the technology mapping phase into the logic synthesis: they construct a network of logic blocks that can be mapped one-to-one onto a network of primitive hardware building blocks. Furthermore, the internal structure of Xilinx FPGAs and similar fine granularity FPGAs is in fact a programmable multiple-level logic block structure which can be innately modeled using the general decomposition theory presented in Chapter 2. Therefore methods for the synthesis of these types of fine granularity FPGAs can be relatively easily constructed using the general decomposition approach.

The logic synthesis approach presented in this thesis is based on the general decomposition theory.

1.3 General decomposition based logic synthesis

In the previous sections logic synthesis has been introduced and the state of the art in logic synthesis has been sketched. In this section, the problem tackled in this Ph.D. thesis will be preliminary introduced. Its more precise definition can be found in Chapter 4. The solution to this problem represents a new method for the synthesis of multiple-level combinational logic circuits based on the general decomposition theory. This method has been named *LUTSYN*; an acronym for *L*ook-*U*p *T*able *SYN*thesis.

The aim of the research work reported in this thesis was the construction of an effective and efficient synthesis method for the multiple-level implementation of multiple-output combinational Boolean functions using LUT based FPGAs and the development of a software environment in which decomposition experiments can be performed. Although the method is general in the sense that any sort function can be processed with it, the software environment focuses on controller type functions, i.e. Boolean functions resulting from specifications of combinational and sequential controllers. The original function tables of such functions can be characterized as having relatively few product terms compared to the number of all possible product terms and having relatively many don't cares. The input data structure of the method is thus a function table of an incompletely specified multiple output Boolean function. The term "incompletely specified" refers to the fact that one or more of the outputs can have don't care values for input combinations. This table has to be transformed into a (near) optimal network of LUTs. Each LUT is able to implement any Boolean function that has no more than NI inputs and no more than NO outputs, where NI and NO are small positive integers (typically $NI \leq 5$ and $NO \leq 2$). All LUTs are assumed to have the same size. The aim of the transformation is a trade-off between the number of LUTs necessary for the implementation, the number of long wires of the implementation and minimizing the number of levels in the network.

Since the problem of optimal logic synthesis is computationally complex, strictly optimal circuit implementations cannot always be obtained using a realistic amount of resources. Therefore heuristic algorithms will be used that strive to find a solution being as close as possible to the optimal solution. To evaluate the overall method and the effectiveness of the heuristics used, the software environment contains a framework synthesis tool. This framework tool is in fact an engine that is able to generate all possible decomposition structures. It can be used for the exhaustive search of the solution space; but even for small circuits this is not very practical. The actual use of the framework follows from the fact that it is that it is easy to add various heuristic search algorithms to it. This way, it allows to implement, modify and test new heuristic algorithms in a flexible and fast way. Although the LUTSYN environment is not yet a complete automated synthesis tool it can be used as a prototype automatic logic synthesis tool. Some benchmark circuits from well known benchmark sets and some specially constructed benchmarks have been used to perform decomposition experiments and to test LUTSYN.

LUTSYN has a number of characteristic features that differentiate it from other known logic synthesis methods:

1. LUTSYN uses the canonical set system representation which is a compact maximal functionally complete representation of logic blocks and Boolean functions, whereas traditional methods typically use minimal functionally complete representations. Set systems model the LUT functions and information flows between LUTs very naturally (Section 4.2.2).

2. LUTSYN is based on the general decomposition approach. This means among others that it implicitly considers all possible loop-free decomposition structures. The solution space is not limited in any way a priori, but the most appropriate solution structures are constructed by the method for each particular problem instance (Section 4.3.6).

3. LUTSYN searches explicitly for common subfunctions, whereas other methods find subfunctions only implicitly as the by-product of the synthesis process. Furthermore, only these explicitly found subfunctions will be actually implemented. This is done to prevent the creation of many very small subfunctions whose values would need to be distributed all over the FPGA and therefore would cause many long wires without substantially decreasing the active area (Section 4.3.5).

4. The construction algorithm uses a bottom-up approach. Starting from the inputs, it constructs a network of LUTs until all output functions have been implemented. Also, the algorithm constructs a number of the most potential solutions in parallel. This is different from all other synthesis algorithms that use a top-down reduction technique for the network construction and that construct solutions serially or do not create more than one solution at all (Section 4.3.2).

5. All heuristic decisions of the search algorithms are based on a single heuristic decision framework that is applied consequently everywhere, thus ensuring that all decisions work consistently in the same direction (Section 4.3.3).

6. The method does not use two-level minimization prior to the actual multiple-level synthesis what enables the use of the whole design freedom during the actual multiple-level synthesis (Section 4.3.4).

We believe that LUTSYN has many attractive properties and we hope that it will form an interesting and significant contribution to the logic synthesis field.

1.4 Organization of the thesis

The next chapter will introduce the necessary theory. First some basic definitions as well as partition and partition pair theory will be presented. Then, the general decomposition theory is introduced. Based on the general decomposition theory it is possible to make a classification system of all possible types of decompositions; Chapter 2 will also introduce a number of popular decomposition classes.
In Chapter 3, the state of the art in logic synthesis for different types of FPGAs will be presented. It will also contain methods based on the division approach introduced by Brayton, because this multiple-level approach is most commonly known and many of these methods have been extended for synthesis on FPGAs.
Chapter 4 and Chapter 5 deal with LUTSYN itself. Chapter 4 discusses the properties and concepts that good logic synthesis methods for LUT based FPGAs should have. In Chapter 5 these properties and concepts are transformed into the experimental software environment. Since the logic synthesis problem is NP complete, this chapter also deals with the construction of heuristics.
The algorithms described in Chapter 5 have been implemented in the form of a C++ program. Some experimental results obtained with this program can be found in Chapter 6.
The final chapter (Chapter 7) summarizes and evaluates the performed research work and provides suggestions for improvements of LUTSYN and for further research.

Chapter 2

Decomposition theory

In this chapter a general mathematical framework for the decomposition of logic level representations is presented. The framework is called a general decomposition. It will be shown that every possible decomposition structure can be derived from this general decomposition. This framework will be used throughout this thesis to analyze and classify synthesis approaches for combinational functions.
The first section of the chapter (Section 2.1) recalls briefly some mathematical concepts that are used throughout this thesis. The reader is assumed to be quite familiar with these concepts.
In the general decomposition, a building block is represented by an algebraic model called a combinational machine. Combinational machines are the subject of Section 2.2.
The subject of Section 2.3 are partitions and partition theory. The partition theory will provide a general purpose technique for the description of the behaviour of symbolic blocks. These blocks can be seen as the hardware blocks of a decomposition. Translations of symbolic to binary blocks and rules for calculating the combined and the decomposed behaviour are also presented. The partition theory is very well suited for the description of LUT based decompositions of logic function and provides these descriptions in a very natural way. Partition theory gives the mathematical modeling background for the general decomposition. The decomposition method presented in this thesis will not use the classical partition theory, but its generalization called set system theory (Section 4.2.2).
Based on the partition theory, the general decomposition theory provides the necessary formalization to prove that a network of blocks implements the behaviour of some other network, i.e. the general decomposition theory allows correctness by construction (Section 2.4).
Unfortunately, no methods have been published till now that use the actual general decomposition paradigm to find near-optimal decompositions. All researches reported consider only some special cases of the general decomposition. A number of known special decomposition models can be found in Section 2.5.
In Chapter 3, the general decomposition concept will be used to classify and analyze a number of known decomposition approaches.

2.1 Basic definitions

In this section, a number of basic mathematical concepts used in this thesis will be briefly recalled. This section is not intended for learning these concepts, it has only recollection in mind. Readers really not familiar with them are referred to text books on algebra, for example [LP81]. Familiarity with sets and operations on sets is assumed in the following definitions.

Definition 2.1 *Power set*
The power set of a set V (denoted as 2^V) is defined as the set containing all the subsets of V:

$$2^V = \{W \mid W \subseteq V\} \tag{2.1}$$

□

Definition 2.2 *Cartesian product*
The Cartesian product S of n non-empty sets S_i $(1 \leq i \leq n)$ is the set with all n-tuples $(s_1, s_2, ..., s_n)$ such that

$$S = S_1 \otimes S_2 \otimes ...S_n = \bigotimes_{1 \leq i \leq n} S_i = \{(s_1, s_2, ..., s_n) \mid s_i \in S_i\} \tag{2.2}$$

□

Definition 2.3 *A binary relation*
A binary relation R on set X and Y is a subset of the Cartesian product $X \otimes Y$. The notation xRy is used to denote that $(x, y) \in R$. □

Definition 2.4 *Single input/output complete function*
Let X and Y be non-empty sets. Let $\exists!$ be the unique existential qualifier. Binary relation F on set X and Y is called a (single input/output) complete function of X into Y, if and only if

$$\forall_{x \in X} \exists!_{y \in Y} (x, y) \in F \tag{2.3}$$

i.e. F assigns to each element $x \in X$ an element $y \in Y$. The function will be denoted as:

$$F : X \rightarrow Y \tag{2.4}$$

and the assignment will be written as:

$$F(x) = y \tag{2.5}$$

□

Definition 2.5 *Single input/output partial function*
Let X and Y be non-empty sets. Binary relation F on set X and Y is called a (single input/output) partial function of X into Y, if and only if

$$\forall_{(x,y),(x',y') \in F} (x = x') \Rightarrow (y = y') \tag{2.6}$$

i.e. F assigns to each element $x \in X$ *at most one* element $y \in Y$. The partial function will be denoted the same way as complete functions:

$$F : X \rightarrow Y \tag{2.7}$$

and the assignment will be written as:

$$F(x) = y \tag{2.8}$$

□

Usually, the word complete is skipped and the term function is used to refer to complete function. The word partial is never left out.
Sometimes it is useful to have explicitly a (complete or partial) function with more than one input and/or with more than one output, for example, if the inputs and outputs have well defined practical meaning. In such a case the following definition can be used.

Definition 2.6 *Multiple input/output (complete or partial) function*
Let ni be the number of inputs of the function, let no be the number of outputs of the function. Let $X_i (1 \leq i \leq ni)$ and $Y_j (1 \leq j \leq no)$ be non-empty sets. Let

$$X = \bigotimes_{1 \leq i \leq ni} X_i$$

and let

$$Y = \bigotimes_{1 \leq j \leq no} Y_j$$

Binary relation F on sets X and Y is an ni-input, no-output (complete or partial) function, written as:

$$F : X \rightarrow Y \tag{2.9}$$

i.e., F assigns to each ni-tuple from X a no-tuple from Y. The assignment will be written as:

$$F(x_1, x_2, ..., x_{ni}) = (y_1, y_2, ..., y_{no}) \tag{2.10}$$

□

From this definition it can be seen that a multiple output function is in fact a collection of single output functions. These single output functions are referred to as *component functions*. It is evident that a multiple input/output function is equivalent to a single output function defined on the Cartesian product of the inputs and the Cartesian product of the outputs and that further distinguishing between these two types of functions is not necessary from the theoretical view point.

Definition 2.7 *An Onto (or surjective) function*
A complete function $F : X \rightarrow Y$ is called onto (or surjective) if and only if

$$\forall_{y \in Y} \exists_{x \in X} F(x) = y \tag{2.11}$$

□

Definition 2.8 *Equivalence relation*
$\equiv$ is called an equivalence relation on $S \otimes S$ if and only if for all $x, y, z \in S$

$$x \equiv x \text{ reflexive property} \tag{2.12}$$

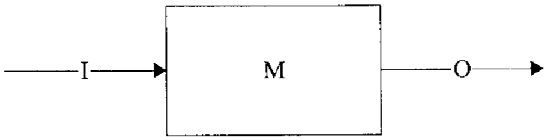


Figure 2.1: Graphical representation of a combinational machine

$$x \equiv y \Rightarrow y \equiv x \text{ symmetric property} \tag{2.13}$$

$$x \equiv y \wedge y \equiv z \Rightarrow x \equiv z \text{ transitive property} \tag{2.14}$$

□

Definition 2.9 *Equivalence class*
If $\equiv$ is an equivalence relation on S then all elements in S are divided into a number of equivalence classes $B_{\equiv}(s)$ as follows: $B_{\equiv}(s) = \{t \mid s \equiv t\}$ □

Equivalence relations and equivalence classes play an important role in the partition theory presented in Section 2.3.

2.2 Combinational machines

In the general decomposition theory combinational logic functions are represented by a mathematical model called a combinational machine [Jóź95a][VJS95].

Definition 2.10 *Completely specified combinational machine*
A completely specified combinational machine M is an algebraic system defined by:

$$M = (I, O, \lambda) \tag{2.15}$$

where:

I - a finite non-empty set of inputs

O - a finite non-empty set of outputs

λ - the (single) output function $\lambda : I \rightarrow O$

□

The graphical representation of a combinational machine can be found in Figure 2.1; a box with an input, an output and a functional specification of its behaviour, but no information on the internals of the box.
In this definition the combinational machine is seen from a purely symbolic view point. It takes a symbolic input and has a symbolic output. In specification of hardware it can

be useful to allow more inputs and outputs, if each of the inputs and outputs has a well defined "real world" meaning. In such a case, the following (mathematical equivalent) model can be used.

Definition 2.11 *Completely specified combinational machine with multiple inputs and outputs*
A completely specified combinational machine M with ni inputs and no outputs is an algebraic system defined by:

$$M^* = (I^*, O^*, \lambda^*) \tag{2.16}$$

where:

$I^* = \bigotimes_{1 \leq j \leq ni} I_j$

I_j - a non empty set of inputs symbols for input j

$O^* = \bigotimes_{1 \leq k \leq no} O_k$

O_k - a non empty set of output symbols for input k

λ^* - the multiple input/output function $\lambda^* : I^* \rightarrow O^*$

$\square$

The design requirements do not always completely specify a machine. It is possible that for certain input combination one or more of the outputs of the machine are not defined because these input combinations will never occur in practice. From the behavioural viewpoint, the designer does not care what will be the output value for such an input value. In all such situations one talks about so called "don't care" conditions. "Don't cares" are commonly denoted by "-". In order to account for them, the combinational machine definition should be slightly modified by changing the definition of the machine function.

Definition 2.12 *Incompletely specified multiple input/output combinational machine*
An incompletely specified combinational machine M with ni inputs and no outputs is an algebraic system defined by:

$$M^{dc} = (I^{dc}, O^{dc}, \lambda^{dc}) \tag{2.17}$$

where:

$I^{dc} = \bigotimes_{1 \leq j \leq ni} I_j$

I_j - a non empty set of inputs symbols for input j

$O^{dc} = \bigotimes_{1 \leq k \leq no} O_k \cup \{-\}$

O_k - a non empty set of output symbols for input k

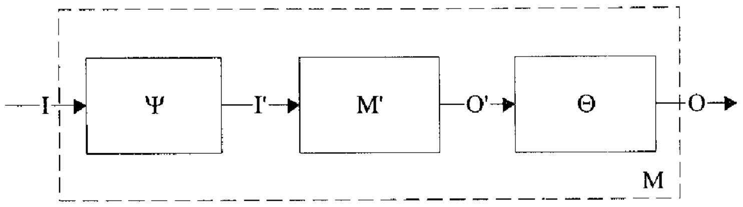


Figure 2.2: Realization of combinational machine M by combinational machine M'

λ^{dc} - the multiple input/output function $\lambda^{dc} : I^{dc} \rightarrow O^{dc}$

□

A combinational machine without "don't cares" will be referred to as completely specified and a combinational machine with "don't cares" will be said to be incompletely specified. In many cases it is useful to express that two machines have the same behaviour, except for the don't care output values. This is called a realization. Before a realization can be introduced it is first necessary to define a covering.

Definition 2.13 *Symbol covering*
Given a non empty symbol set A and the set $A^{dc} = A \cup \{-\}$. Symbol $b \in A^{dc}$ is said to cover symbol $a \in A^{dc}$ (denoted as $a \leq b$) if and only if $(a = b) \bigvee (b = -)$ □

Informally, b covers a means that a is more specific than b is. Similarly, a vector defined on a Cartesian product is said to cover another vector defined on that Cartesian product if and only if each element of the first vector covers the corresponding element of the second vector.

Definition 2.14 *Realization of a machine*
Incompletely specified machine $M' = (I', O', \lambda')$ is a realization of incompletely specified machine $M = (I, O, \lambda)$ (Figure 2.2) if and only if the relations $\Psi : I \rightarrow I'$ (a function) and $\Theta : O' \rightarrow O$ (a surjective partial function) exist, so that

$$\forall_{x \in I} \; \Theta(\lambda'(\Psi(x))) \leq \lambda(x) \tag{2.18}$$

□

The machine composed as a structure consisting of Ψ, M' and Θ and being the realization structure for M defined by M' will be denoted by $str(M')$. From this definition it follows that that if M' is a realization of M then for all possible inputs of M (i.e. $x \in I$), the outputs $\lambda'(\Psi(x)) \in O'$ produced by the realization machine M' are covered (after decoding by $\Theta(\lambda'(\Psi(x)))$) by the corresponding output $\lambda(x) \in O$ of the specification machine M.

I	$x_1x_2x_3$	y_1y_2
a	0 0 0	- -
b	0 0 1	0 0
c	0 1 0	1 1
d	0 1 1	1 1
e	1 0 0	1 0
f	1 0 1	0 0
g	1 1 0	1 -
h	1 1 1	1 1

Table 2.1: Boolean function F for the realization example

Example 2.1 *Realization of a combinational machine*
In Table 2.1, the function table of an incompletely specified Boolean function F is given. The function has 3 input variables (x_1, x_2, x_3) and two output variables (y_1, y_2). In this example the inputs will be modeled as a single symbolic variable, while the separate variable values will be used for the outputs. Each input combination has been labeled with a unique symbolic name (a, b, c, d, e, f, g and h). The output value is a tuple consisting of two components being the values of y_1 and y_2, respectively. This functions can be represented as the following incompletely specified combinational machine with 1-input and 2-outputs:

$$M = (I, O, \lambda)$$

such that

$$I = \{a, b, c, d, e, f, g, h\}$$

$$O = O_1 \otimes O_2 = \{(-,-), (0,0), (1,1), (1,0), (1,-)\}$$

with $O_1 = \{0, 1, -\}$ and $O_2 = \{0, 1, -\}$ and

$$\lambda : I \to O$$

as specified in Table 2.1.
Machine

$$M' = (I', O', \lambda')$$

such that

$$I' = \{\alpha, \beta, \gamma, \delta\}$$

$$O' = \{I, II, III\}$$

and

$$\lambda' = \{(\alpha, I);\ (\beta, II);\ (\gamma, III);\ (\delta, III)\}$$

is a realization of machine M, since the mappings

$$\Psi : I \rightarrow I'$$

$$\Psi = \{(a,\alpha);\ (b,\alpha);\ (c,\beta);\ (d,\beta);\ (e,\gamma);\ (f,\alpha);\ (g,\delta);\ (h,\beta)\}$$

and

$$\Theta : O' \rightarrow O$$

$$\Theta = \{(I,(0,0));\ (II,(1,1));\ (III,(1,0))\}$$

satisfy the relation

$$\forall x \in I : \Theta(\lambda'(\Psi(x)) \leq \lambda(x)$$

For example, take $x = e$: $\lambda(e) = (1,0)$ (see Table 2.1) and $\Theta(\lambda'(\Psi(e)) = \Theta(\lambda'(\gamma)) = \Theta(III) = (1,0)$ and $(1,0) \leq (1,0)$. Similarly, $\lambda(g) = (1,-)$ and $\Theta(\lambda'(\Psi(g)) = \Theta(\lambda'(\delta)) = \Theta(III) = (1,0)$ and $(1,0) \leq (1,-)$. In this realization the don't care in the second output variable has been filled in with a 0. Verification of this relation for other combinations can be performed in a strictly analogous way. □

The definition of realization of a combinational machine is used to guarantee that the combinational digital circuit that supposedly is an implementation of the combinational machine that was the starting point of the synthesis process, actually preserves the behaviour of the specified combinational machine.

2.3 Partitions

Partitions and partition pairs, originally introduced by Hartmanis to model information in sequential machines [HS66] are useful for modeling information and information flows inside and between combinational machines. Together with the concept of combinational machines, partition theory forms the mathematical base of the general decomposition theory described in the next section.

Definition 2.15 *Partition*
Let S be a non empty set of elements. Partition π on S is defined as a collection of disjoint non-empty subsets of S whose set union is S. More formally, if

$$\pi_S = \{B_i\} \tag{2.19}$$

then π_S is called a partition if and only if the following conditions are satisfied:

$$\forall_{i \neq j}\ B_i \cap B_j = \varnothing \tag{2.20}$$

$$\underset{i}{\forall} B_i \neq \varnothing \tag{2.21}$$

$$\bigcup_i B_i = S \tag{2.22}$$

□

If it is unambiguously determined on which set a partition is defined, the notation π without the set name index is simply used. Any partition π on S can be interpreted as an equivalence relation defined on S with the equivalence classes being the blocks of π. Using this interpretation, the partition π gives information about the elements of S with precision to the equivalence class. With this information, it is possible to distinguish elements from different classes although it is impossible to distinguish elements that are in the same class.
For example, if $S = \{1,2,3,4\}$ then $\pi = \{\overline{1,2};\ \overline{3,4}\}$ is a partition defined on set S. This is the way that partitions are denoted: the blocks of the set are overlined instead of using an extra set of brackets. If symbols are used for which an ordering exists in the real world then the blocks of the partition are sorted with respect to the lowest ordered element in that block. It is important to realize that this is only a notational convention and it has nothing to do with the ordering relations defined on partitions.

For a given s∈S, the *block of a partition π containing s* is denoted as $[s]\pi$ while $[s]\pi = [t]\pi$ denotes that *s and t are in the same block of π*. Similarly, the block of a partition π containing S', where $S' \subseteq S$, is denoted by $[S']\pi$.

The partition containing each element of S in a separate block is called a *zero partition* and denoted by $\pi_S(0)$. The partition containing all the elements of S in one block is called an *identity partition* and is denoted by $\pi_S(I)$.

Definition 2.16 *Partition product*
Let π_1 and π_2 be two partitions on S. The partition product $\pi_1 \cdot \pi_2$ is the partition on S such that $[s]\pi_1 \cdot \pi_2 = [t]\pi_1 \cdot \pi_2$ if and only if $[s]\pi_1 = [t]\pi_1$ and $[s]\pi_2 = [t]\pi_2$. □

Definition 2.17 *Partition sum*
Let π_1 and π_2 be two partitions on S. The partition sum $\pi_1 + \pi_2$ is the partition on S such that $[s]\pi_1 + \pi_2 = [t]\pi_1 + \pi_2$ if and only if a sequence: $s = s_0, s_1, \ldots, s_n = t$, $s_i \in S$ for i=1..n, exists for which either $[s_i]\pi_1 = [s_{i+1}]\pi_1$ either $[s_i]\pi_2 = [s_{i+1}]\pi_2$ for $0 \leq i \leq n-1$. □

From the above definitions, it follows that the blocks of $\pi_1 \cdot \pi_2$ are obtained by intersecting the blocks of π_1 and π_2, while the blocks of $\pi_1 + \pi_2$ are obtained by uniting all the blocks of π_1 and π_2 which contain common elements.

Definition 2.18 *Partial ordering $\leq$ of partitions*
π_2 is greater than or equal to π_1 : $\pi_1 \leq \pi_2$ if and only if each block of π_1 is included in a block of π_2. Thus $\pi_1 \leq \pi_2$ if and only if $\pi_1 \cdot \pi_2 = \pi_1$ if and only if $\pi_1 + \pi_2 = \pi_2$. □

The partial ordering relation $\leq$ denotes the fact that if $\pi_1 \leq \pi_2$ then π_1 (and so the associated equivalence relation) provides information about elements of S, that is at least as precise as information given by π_2 (and its associated equivalence relation). After all, π_1 is able to make all the distinctions between the elements of S that π_2 can make.
A zero partition provides complete information about the elements of S and an identity partition gives no information at all. The partition product can be interpreted as a product of the appropriate equivalence relations introduced by these partitions; it represents the combined information about the elements of S provided by both relations together. The partition sum can be interpreted as a sum of the appropriate equivalence relations introduced by these partitions and it represents the combined abstraction of both relations; where the abstraction of two elements from S is defined as they are in the same block.

Remark: In spoken language the definition of $\leq$ is often a source of confusion: the *smaller partition* is the partition that provides *more information* about elements of S. Similarly, the larger partition expressed with $\leq$ possesses less information about the elements of S.

Example 2.2 *Partitions and operations on partitions*
In this example the set $S = \{a, b, c, d, e, f, g, h\}$ is used. $\pi_S(0) = \{\overline{a};\ \overline{b};\ \overline{c};\ \overline{d};\ \overline{e};\ \overline{f};\ \overline{g};\ \overline{h}\}$ is the zero input partition and $\pi_S(I) = \{\overline{a, b, c, d, e, f, g, h}\}$ is the input identity partition. The zero partition provides complete information (each symbol of S is in a separated class (block) and all symbols can be distinguished from each other). The identity partition does not express useful information: no symbol can be distinguished from another symbol.
Let $\pi_1 = \{\overline{a, b};\ \overline{c, d};\ \overline{e, f, g, h}\}$ and $\pi_2 = \{\overline{a, b};\ \overline{c, e};\ \overline{d, f, g, h}\}$ be partitions on set S. The product $\pi_1 \cdot \pi_2 = \{\overline{a, b};\ \overline{c};\ \overline{d};\ \overline{e};\ \overline{f, g, h}\}$ denotes the combined information of π_1 and π_2, for example in π_1 the symbols c and d are equivalent and hence partition π_1 cannot distinguish between these two symbols. π_2 can make the distinction between c and d (because they are in different blocks of π_2). The product $\pi_1 \cdot \pi_2$ represents the partition that makes the union of the distinctions of π_1 and π_2 and combines in one block only those elements which are equivalent (not distinguished) in both partitions. Similarly, $\pi_1 + \pi_2 = \{\overline{a, b};\ \overline{c, d, e, f, g, h}\}$ represents the combined abstraction of the two partitions. For example, π_1 cannot distinguish between symbol c and d, π_2 does not express the difference between c and e, so the combined abstraction does not distinguish between these 3 symbols (and hence they are placed in one block).
Finally, for the partitions $\pi_3 = \{\overline{a, b};\ \overline{c, d};\ \overline{e, f, g, h}\}$ and $\pi_4 = \{\overline{a, b};\ \overline{c, d, e, f, g, h}\}$, $\pi_3 \leq \pi_4$, because π_3 makes all distinctions that π_4 makes. This can also be checked by the definition of $\leq$: $\pi_3 \leq \pi_4 \Leftrightarrow \pi_3 \cdot \pi_4 = \pi_3$ and $\pi_3 \cdot \pi_4 = \{\overline{a, b};\ \overline{c, d};\ \overline{e, f, g, h}\} = \pi_3$.
$\leq$ is a partial ordering relation: therefore it needs not be defined for all pairs of partitions: $\pi_5 = \{\overline{a, b};\ \overline{c, d};\ \overline{e, f, g, h}\}$ and $\pi_6 = \{\overline{a, c, e};\ \overline{b, f};\ \overline{d, g, h}\}$ are not related by $\leq$, since $\pi_5 \cdot \pi_6 = \{\overline{a, b};\ \overline{c, d};\ \overline{e, f, g, h}\} \cdot \{\overline{a, c, e};\ \overline{b, f};\ \overline{d, g, h}\} = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, \overline{f}, \overline{g}, \overline{h}\} = \pi(0)$. This can also seen from the following reasoning: π_5 makes the distinction of a and c (and π_6 does not) and π_6 makes the distinction of a and b (but π_5 cannot make this distinction). Hence π_5 does not implement a subset of the behaviour of π_6 nor does π_6 implement all the partial equivalences of π_5. So, π_5 and π_6 are not related by $\leq$. □

Definition 2.19 *I-O partition pair*
Given $M = (I, O, \lambda)$, let π_I be a partition on I and let π_O be a partition on O. (π_I, π_O) is an I-O partition pair if and only if

$$\forall_{A \in \pi_I} \; \exists_{C \in \pi_O} \; \overline{\lambda(A)} \subseteq C \tag{2.23}$$

where:

$$\overline{\lambda(A)} = \{\lambda(x) \mid x \in A\} \tag{2.24}$$

□

Informally, (π_I, π_O) is an I-O partition pair if and only if each block of π_I unambiguously determines the block of π_O in which the output is contained. If (π_I, π_O) is a partition pair then π_I is called the first partition of the pair and π_O is called the second partition of that pair.

Definition 2.20 *M and m partitions of a partition pair*
Given $M = (I, O, \lambda)$. Let π_I be a partition on I and let π_O be a partition on O. The *minimal second partition* which forms an I-O partition pair with π_I as a first partition will be denoted $m_{I-O}(\pi_I)$. The *maximal first partition* which forms an I-O partition pair with π_O as a second partition will be denoted $M_{I-O}(\pi_O)$. It can be proved [HS66] that:

$$m_{I-O}(\pi_I) = \prod \{\pi_j \mid (\pi_I, \pi_j) \text{ is a I-O partition pair}\} \tag{2.25}$$

$$M_{I-O}(\pi_O) = \sum \{\pi_j \mid (\pi_j, \pi_O) \text{ is a I-O partition pair}\} \tag{2.26}$$

□

For a given π_I, $m(\pi_I)$ describes the largest amount of information which can be computed about the output of M knowing the block of π_I that contains the input. $M(\pi_O)$ describes the least amount of information which must be known about the input of M, in order to be able to compute the information about the output with precision to π_O.

Definition 2.21 *Mm partition pair*
Let $M = (I, O, \lambda)$ be a combinational machine. Let π_I be a partition on I and let π_O be a partition on O. (π_I, π_O) is a Mm I-O partition pair if and only if $\pi_I = M(\pi_O)$ and $\pi_O = m(\pi_I)$ □

Mm partition pairs play an important role in the characterization of a combinational machine M. They form in fact the skeleton of all partition pairs of the machine, because all other partition pairs can be derived from the Mm partition pairs of a machine M by increasing the information in the first partition of a Mm pair and/or by decreasing the amount of information in the second partition of that pair.

I	$x_1x_2x_3$	y_1y_2	O
a	000	01	x
b	001	11	z
c	010	00	w
d	011	10	y
e	100	10	y
f	101	11	z
g	110	00	w
h	111	00	w

Table 2.2: Function table for partition pair example

Definition 2.22 *Input partition induced by an output partition*
Let $M = (I, O, \lambda)$ be a combinational machine. Let π_I be a partition on I and let π_O be a partition on O. π_I is an input partition induced by an output partition π_O (notation: $\pi_I \in ind(\pi_O)$) if and only if:

$$\underset{x,y \in I}{\forall} : [\lambda(x)]\pi_O = [\lambda(y)]\pi_O \Rightarrow [x]\pi_I = [y]\pi_I \tag{2.27}$$

□

In other words, if π_I is an input partition induced by an output partition π_O and, if it is known that the present output y of M is contained in a block $C \in \pi_O$, then it is known that current input I of M is contained in a block $B \in \pi_I$, where block B is unambiguously indicated by block C. It is possible to prove that π_I is an input partition induced by an output partition π_O if and only if $\pi_I \geq M_{I-O}(\pi_O)$, i.e. the smallest input partition induced by a certain π_O is $M_{I-O}(\pi_O)$.

Example 2.3 *Partition pairs and operations on partition pairs*
In this example the function from Table 2.2 is used. The table can be transformed to the following symbolic combinational machine $M(I, O, \lambda)$:

$$I = \{a, b, c, d, e, f, g, h\}$$

$$O = \{w, x, y, z\}$$

and

$$\lambda = \{(a, x);\ (b, z);\ (c, w);\ (d, y);\ (e, y);\ (f, z);\ (g, w);\ (h, w)\}$$

is used. Given the partition $\pi_I = \{\overline{a, c};\ \overline{b, f};\ \overline{d, e};\ \overline{g, h}\}$ on set I and partition $\pi_O = \{\overline{w, x};\ \overline{y, z}\}$ on set O. (π_I, π_O) is a I-O partition pair and this can be easily checked by checking all blocks of π_I following definition (2.23) (e.g. $\lambda(\{a, c\}) = \{x, w\}$

which is a subset of a block of π_O etc). $m_{I-O}(\pi_I) = \{\overline{w,x};\ \overline{y};\ \overline{z}\}$ represents the maximal information about the output of M that can be calculated using π_I. Similar $M_{I-O}(\pi_O) = \left\{\overline{a,c,g,h};\ \overline{b,d,e,f}\right\}$ represents the minimal input information that is necessary as an input to calculate π_O. From these calculations it also follows that (π_I, π_O) is not a Mm partition pair.
The partitions $\pi'_I = \{\overline{a,c,g,h};\ \overline{b,d,e,f}\}$ and $\pi'_O = \{\overline{w,x};\ \overline{y};\ \overline{z}\}$ do form an Mm I-O partition pair. This can be easily verified by calculating the M and m partitions of π'_I and π'_O.
Let $\pi''_O = \{\overline{w,x};\ \overline{y};\ \overline{z}\}$ then $\left\{\overline{a,c,g,h};\ \overline{b,d,e,f}\right\}$ and $\left\{\overline{a,c,g,h};\ \overline{b,f};\ \overline{d,e}\right\}$ are both induced input partitions of π''_O. □

For the purpose of a bit decomposition (in which the input/output bits are distributed instead of the input/output symbols), the concept of bit partitions has been introduced [Jóź89].
Let $B = \{b_1, b_2, .., b_{|B|}\}$ be a set of binary (input or output) variables (bits). Let $T = \{t_1, t_2, ..t_{|T|}\}$ be a set of (input or output) symbols (bit value patterns). Each input/output bit $b_k \in B$, introduces a two-block partition $\pi_T(b_k)$ on the set of symbols T: one block of $\pi_T(b_k)$ contains the symbols for which bit b_k has the value 0 and the other block contains the symbols for which b_k has the value 1. The product of the partitions $\pi_T(b_k)$ for all the bits $b_k : b_k \in B$ will unambiguously define the set of all input/output symbols, i.e. it will be a zero partition on T.
For incompletely specified machines, the partition $\pi_T(b_k)$ is defined on the subsets of T for which a value for this bit is specified. Therefore these partitions cannot be simply multiplied. In [HS66] set system theory has been introduced to overcome this problem. Set systems are an attempt to extend partition theory with don't care symbols. In Section 4.2.2 set systems will be introduced.

Definition 2.23 *Bit partition*
A partition π_B on the set of bits B $\pi_B = \{\overline{b_1};\ \overline{b_2};\ ...;\ \overline{b_k}, (\overline{b_{k+1}, .., b_{|B|}})\}$ is called a bit-partition. In a bit-partition the important bits (for distinguishing between certain symbols) $b_1, .., b_k$ are kept in separate blocks and the don't care bits $b_{k+1}, .., b_{|B|}$ are kept in a single block called a don't care block (denoted by $dcb(\pi_B)$). □

The product $(\cdot)$ and sum $(+)$ operations as well as the ordering relations $(\leq)$ for bit partitions are defined in the same way as for "normal" partitions with the following supplement: the product of a block (important or don't care) with important blocks is an important block in the product partition; whereas the sum of a block (important or don't care) with a don't care block is a don't care block in the sum partition. The *zero bit-partition* is defined as a bit partition with an empty don't care block. The *identity bit-partition* is defined as a bit-partition with all elements in the don't care block.

Definition 2.24 *Symbol partition induced by a bit partition*
π_T is a symbol partition induced by a bit partition π_B ($\pi_T \in ind(\pi_B)$) if and only if

$$\pi_T \geq \prod_{b_k \in (B - dcb(\pi_B))} \pi_T(b_k) \tag{2.28}$$

□

Definition 2.25 *The smallest symbol partition induced by a bit partition*
π_T is the smallest symbol partition induced by a bit partition π_B ($\pi_T = ind^*(\pi_B)$) if and only if

$$\pi_T = \prod_{b_k \in (B - dcb(\pi_B))} \pi_T(b_k) \tag{2.29}$$

□

From the above definitions it follows that any partition $\pi_T \geq ind^*(\pi_B)$ is an induced partition of π_B.

Definition 2.26 *Bit partition induced by a symbol partition*
π_B is a bit partition induced by a symbol partition π_T ($\pi_B \in ind(\pi_T)$) if and only if

$$\forall_{b_k \in (B - dcb(\pi_B))} : \pi_T(b_k) \geq \pi_T \tag{2.30}$$

□

If $\pi_T \in ind(\pi_B)$ then, having π_B the blocks of π_T can be computed. If $\pi_B \in ind(\pi_T)$ then, having the block of π_T the values of all the important bits from π_B can be computed.

Example 2.4 *Bit partitions*
In this example the function from Table 2.2 will be considered as a binary instead of a symbolic combinational machine. Bit-partitions on the input bits can be made. The set with of input bits is denoted with X, i.e. $X = \{x_1, x_2, x_3\}$. In this case the input symbols are labeled with lowercase letters from the alphabet, i.e. the set I of the input symbols is defined as $I = \{a, b, c, d, e, f, g, h\}$. Often, no new symbols for the inputs will be introduced, but simply the bit value vectors will be used as the input symbols. In that case, the set of input symbols can be defined as $I' = \{000, 001, 010, 011, 100, 101, 110, 111\}$.
An input bit represents a symbolic partition which contains in the first block all symbols for which this bit is 0 and in the second block all symbols for which this bit is 1: $\pi_I(x_1) = \left\{\overline{a, b, c, d};\ \overline{e, f, g, h}\right\}$, $\pi_I(x_2) = \left\{\overline{a, b, e, f};\ \overline{c, d, g, h}\right\}$ etc.
A bit partition on X is for example the partition $\pi_X = \{\overline{x_2};\ \overline{x_3};\ (\overline{x_1})\}$. Given any partition τ, τ is a symbol partition induced by π_X if and only if:

$$\tau \geq \prod_{b_x \in (X - dcb(\pi_X))} \pi_I(b_x)$$

which can be calculated as follows:

$$\tau \geq \pi_I(x_2) \cdot \pi_I(x_3) \Rightarrow$$

$$\tau \geq \left\{\overline{a,b,e,f};\ \overline{c,d,g,h}\right\} \cdot \left\{\overline{a,c,e,g};\ \overline{b,d,f,h}\right\} \Rightarrow$$

$$\tau \geq \left\{\overline{a,e};\ \overline{b,f};\ \overline{c,g};\ \overline{d,h}\right\}$$

Similarly, an example of a bit partition induced by the symbol partition $\left\{\overline{a,c,e};\ \overline{b,d,f,h};\ \overline{g}\right\}$ is the partition $\tau_X = \{\overline{x_3};\ (\overline{x_1,x_2})\}$. In this case τ_X is the only possible bit partition. □

2.4 General decomposition

The topic of this section is the general decomposition theory. This theory will be an aid in the analyzes and the classification of decomposition structures. A number of special decomposition types can be derived from the general decomposition model. Furthermore, a theorem is formulated that provides the necessary and sufficient conditions in order to create a general decomposition or one of its special cases.
The theory of general decomposition of sequential machines has been formulated by Jóźwiak [Jóź95a]. This thesis is concerned with the synthesis of combinational logic. However, since a combinational machine is merely a special case of a sequential machine with one state and a trivial next-state function, the general decomposition theory can also be applied to combinational machines. A special case of the general decomposition theory [Jóź95a] related to combinational machines is presented below.

In a general decomposition of a combinational machine $M = (I, O, \lambda)$ a composition of n cooperating partial machines $M_i = (I_i, O_i, \lambda_i)$ as well as the mappings

$$\Psi : I \rightarrow \bigotimes_i I_i$$

and

$$\Theta : \bigotimes_i O_i \rightarrow O$$

have to be found in such a way that the composition of M_i together with the mappings Ψ and Θ realize machine M.
The implementation of such a general decomposition model requires three components: the input coder (preprocessor) Ψ, the simultaneously working communicating component machines (main processors) M_i and the output decoder (postprocessor) Θ. The component machines, input coder and the output decoder are implemented as combinational circuits. The model is general and it contains all elements necessary for the construction of circuit networks which implement combinational circuits: parallel processing elements with possibilities for information exchange between them; divergent preprocessing elements for abstracting and splitting information and representing it in the appropriate form; and convergent postprocessing elements for joining and combining information from parallel processors and representing it in the appropriate form.

The decomposition can be characterized by the type of connections between the component machines and by the type of input/output encoding/decoding. In a general composition, each partial machine can use (partial) output information from another in order to compute its own output. Two well-known special cases of a general composition are the parallel and the serial composition. In a parallel composition, no connections exist between the partial machines. Each partial machine is able to compute its own output independently. In a serial composition, machines are ordered and only the component machines M_i, $i \geq j$, can use information from the machines M_j in order to compute their own output. The formal definition for a general composition is given below.

Definition 2.27 *General composition*
A general composition of n combinational machines $M_i : GC(\{M_i\}, \{Con_i\})$, consists of the following objects:

1. $\{M_i = (I_i^*, O_i, \lambda_i), I_i^* = I_i \otimes I_i', 1 \leq i \leq n\}$, a set of machines referred to as component machines, and
2. $\left\{ Con_i : \bigotimes_{1 \leq j \leq n} O_j \rightarrow I_i', 1 \leq i \leq n \right\}$ a set of surjective functions referred to as *connection rules* of the component machines.

□

Theoretically it is possible to use the output of a combinational machine as the input of this machine. This is called a *local connection*. The use of local connections in a decomposition can be dangerous, utmost care must be taken to prevent loops in the implementation that would transform the time independent combinational circuit to a time dependent sequential circuit or worse to an unstable circuit. To overcome this problem in practical situations local connections and loops are usually prohibited. This is done by creating an ordering on the component machines in the general composition: machine M_i can only use the output of component machines $M_j (1 \leq j < i)$. A (small) price to pay, is the fact that disallowing loops the composition looses its true general character.

Definition 2.28 *"General" composition without loops*
A general composition of n combinational machines without loops $M_i : GC(\{M_i\}, \{Con_i\})$, consists of the following objects:

1. $\{M_i = (I_i^*, O_i, \lambda_i), I_i^* = I_i \otimes I_i', 1 \leq i \leq n\}$, a set of machines referred to as component machines, and
2. $\left\{ Con_i : \bigotimes_{1 \leq j < i} O_j \rightarrow I_i', 1 \leq i \leq n \right\}$ a set of surjective functions referred to as *connection rules* of the component machines.

□

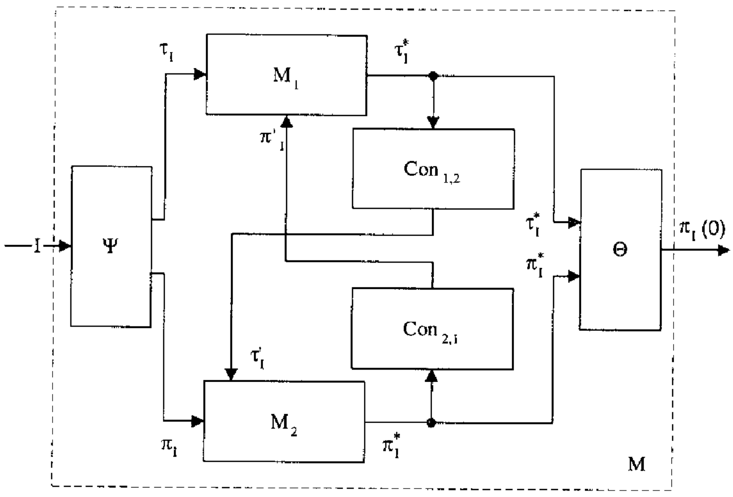


Figure 2.3: General decomposition using two component machines

Definition 2.29 *General composition machine*
A general composition GC of n combinational machines defines the general composition machine $M_{GC}(GC) = (I_{GC}, O_{GC}, \lambda_{GC}) = M_{GC}(\{M_i\}, \{Con_i\})$ with

- $I_{GC} = \bigotimes_i I_i$,
- $O_{GC} = \bigotimes_i O_i$,
- $\lambda_{GC} : I_{GC} \rightarrow O_{GC}$,
- $\lambda_{GC} = \bigotimes_i \lambda_i(x_i, Con_i(y_1, ..., y_n))$, where y_i represents the output of component machine i.

□

No distinction between the general composition and the composition machine it defines will be made, unless this can lead to misunderstanding. In the original theory [Jóź95a] the output can be looped back to the input of a machine. In such a case, λ_{GC} is recursively defined. The physical implementation of the machine can be problematic if it contains loops; therefore in the scope of this thesis a general composition without loops will be used in practice.

Definition 2.30 *General decomposition*
The combinational machine $str(M_{GC})$ is a *general decomposition* of machine M if and only if the general composition machine M_{GC} (with or without loops) realizes M (see Figure 2.3 for the case of two partial machines). □

In [Jóź95a] the following theorem has been proved.

Theorem 2.1 *Existence of general decomposition*
A combinational machine $M(I, O, \lambda)$ has a general decomposition into n component machines if and only if n partition doubles (π_I^i, π_I^{*i}) exist and they satisfy the following conditions:

1.
$$\pi_I^i \cdot \pi_I'^i \leq \pi_I^{*i} \tag{2.31}$$
where:
$$\pi_I'^i \geq \prod_{i=1..n} \pi_I^{*i} \tag{2.32}$$

2.
$$\prod_{i=1..n} \pi_I^i \leq \pi_I^{*i} \tag{2.33}$$

3.
$$\left(\prod_{i=1..n} \pi_I^{*i} \, , \, \pi_O(0) \right) \text{ is an I-O partition pair} \tag{2.34}$$

□

The above theorem does not exclude trivial decompositions (like decompositions consisting of empty or duplicated component machines). A general decomposition is said to be non trivial if each of the component machines is necessary for obtaining the output of the machine.

The theorem will be discussed using a special case of the above theorem which has only two combinational machines. The theorem for this case is presented below (see also Figure 2.3).

Theorem 2.2 *Existence of general decomposition using two component machines*
A combinational machine $M(I, O, \lambda)$ has a general decomposition with two partial machines and without local connections if and only if two partition doubles (π_I, π_I^*) and (τ_I, τ_I^*) exist that satisfy the following conditions:

1.
$$\tau_I \cdot \pi_I' \leq \tau_I^* \text{ and } \pi_I \cdot \tau_I' \leq \pi_I^* \tag{2.35}$$
where:
$$\pi_I' \geq \pi_I^* \text{ and } \tau_I' \geq \tau_I^* \tag{2.36}$$

2.

$$\pi_I \cdot \tau_I \leq \pi_I^* \text{ and } \pi_I \cdot \tau_I \leq \tau_I^* \tag{2.37}$$

3.

$$(\pi_I^* \cdot \tau_I^*, \pi_O(0)) \text{ is an I-O partition pair} \tag{2.38}$$

□

In Figure 2.3 the theorem is visualized. The combinational machine M_1 has as its inputs τ_I and π'_I. τ_I represents the information obtained from the primary inputs of the circuit. π'_I represents information obtained from the other component machine M_2. Since M_1 is a combinational machine it has no memory and the output τ_I^* of M_1 has to be calculated from only the inputs τ_I and π'_I of M_1. This condition is expressed by the first half of Equation (2.35) that states that the partition product of τ_I and π'_I (that represents the combined information of these two inputs) should be sufficient to calculate τ_I^*.The second part of (2.35) expresses the same condition for machine M_2.
Equation (2.36) expresses the information processing capacity of the connection rules $Con_{1,2}$ and $Con_{2,1}$: the connection rules can transmit a part of the output information of a certain component machine to the other machine.
To make sure that it is possible to construct the general composition of M_1 and M_2 as a legal decomposition Equation (2.37) is used. It states that the exchanged information can be computed (directly or indirectly) from the primary input information of the partial machines.
The last condition (Equation (2.38)) assures that the information computed by the component machines is sufficient to calculate the output of machine M. This is expressed by the I-O partition pair: the combined output information of the component machines $(\pi_I^* \cdot \tau_I^*)$ should form a partition pair with the zero output partition, i.e. it should be able to distinguish between all output symbols.
In this way, the general decomposition model and theorem define the solution space of general decompositions, by providing a generator of this space.

By repeated use of the general decomposition model or its special cases, all functionally correct combinational circuit structures can be constructed. The general decomposition theorem defines the systems of partitions that result in correct general decompositions; however it does not provide details how to find the systems of partitions that will result in the optimal decomposition.
The remaining of this section provides a larger example that once again illustrates the concepts introduced.

Example 2.5 *General decomposition*
In Table 2.3, the specification of a completely specified function F is given. In this example it will be shown how this function can be decomposed into two-input, one-output cells. It will be assumed that cells for all possible two-input, one-output functions will be available in the cell library or that a two-input, one-output LUT is available. A general decomposition theory based synthesis tool might find an implementation F' with 4 LUTs as presented in Figure 2.4. This implementation would most likely be obtained in

I	$x_1x_2x_3x_4$	F
0	0 0 0 0	1
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	0
4	0 1 0 0	0
5	0 1 0 1	1
6	0 1 1 0	1
7	0 1 1 1	1
8	1 0 0 0	1
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	0
12	1 1 0 0	0
13	1 1 0 1	0
14	1 1 1 0	1
15	1 1 1 1	0

Table 2.3: Boolean function F for general decomposition example

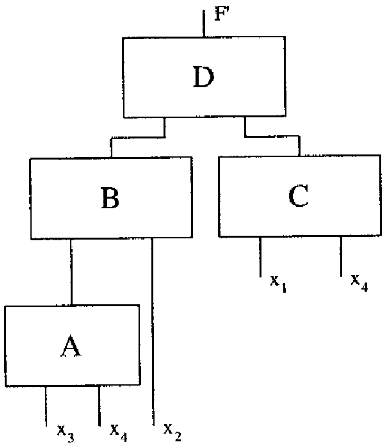


Figure 2.4: Implementation of Boolean function F obtained using general decomposition

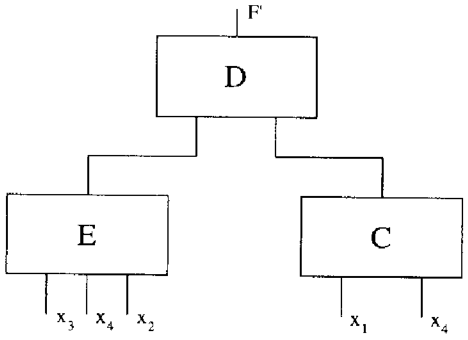


Figure 2.5: Step 1 of a general decomposition algorithm

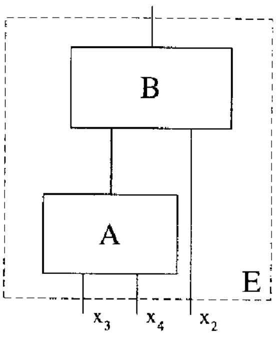


Figure 2.6: Step 2 of a general decomposition algorithm

the following 2 steps.

In a first step a parallel decomposition into two blocks C and E is performed (see Figure 2.5). Since the input symbols are already binary encoded, an input coder is not required and a subset of the input bits can be directly distributed to each machine C and E. Because one-output gates are used, the output partitions π_C^* and π_E^* can only have two blocks. The following two partition doubles satisfy the general decomposition theorem for two component machines: (π_C, π_C^*) and (π_E, π_E^*) where:

$$\pi_C^* = \left\{\overline{0,1,2,3,4,5,6,7,8,10,12,14};\ \overline{9,11,13,15}\right\}$$

$$\pi_E^* = \left\{\overline{0,5,6,7,8,13,14,15};\ \overline{1,2,3,4,9,10,11,12}\right\}$$

π_C is the smallest input symbol partition induced by the inputs x_1 and x_4, i.e.

$$\pi_C = ind^*\left(\{\overline{x_1};\ \overline{x_4};\ (\overline{x_2,x_3})\}\right) =$$

$$\left\{\overline{0,2,4,6};\ \overline{1,3,5,7};\ \overline{8,10,12,14};\ \overline{9,11,13,15}\right\}$$

Similarly:

$$\pi_E = ind^*\left(\{\overline{x_2};\ \overline{x_3};\ \overline{x_4};\ (\overline{x_1})\}\right) =$$

$$\left\{\overline{0,8};\ \overline{1,9};\ \overline{2,10};\ \overline{3,11};\ \overline{4,12};\ \overline{5,13};\ \overline{6,14};\ \overline{7,15}\right\}$$

Verifying that (π_C, π_C^*) and (π_E, π_E^*) satisfy the conditions of the general decomposition

theorem is easy. Since the decomposition in Figure 2.5 is a parallel decomposition, there is no information flow from one component machine to the other and hence π'_I and τ'_I are identity partitions. Equation (2.35) therefore reduces to $\pi_C \geq \pi^*_C$ and $\pi_E \geq \pi^*_E$ which can be easily verified.
To verify condition (2.37) the partition product $\pi_C \cdot \pi_E$ has to be calculated. This turns out to be the zero partition $\pi_I(0)$ so Equation (2.37) is trivially satisfied.
Finally it has to be shown that the partition doubles satisfy Equation (2.38). This is relatively easy:

$$\pi^*_C \cdot \pi^*_E = \left\{\overline{0,5,6,7,8,14};\ \overline{1,2,3,4,10,12};\ \overline{13,15};\ \overline{9,11}\right\}$$

Inspection of Table 2.3 for the blocks of this product partition, shows that the product indeed forms an I-O partition pair with $\pi_O(0)$.

In a second iteration of the algorithm, block E is then further serially decomposed into two blocks (Figure 2.6). The partition doubles (π_A, π^*_A) and (π_B, π^*_B) where,

$$\pi_A = ind^*\left(\overline{x_3};\ \overline{x_4};\ (\overline{x_1, x_2})\right) =$$

$$\left\{\overline{0,4,8,12};\ \overline{1,5,9,13};\ \overline{2,6,10,14};\ \overline{3,7,11,15}\right\}$$

$$\pi^*_A = \left\{\overline{0,4,8,12};\ \overline{1,2,3,5,6,7,9,10,11,13,14,15}\right\}$$

$$\pi_B = ind^*\left(\overline{x_2};\ (\overline{x_1};\ \overline{x_3, x_4})\right) =$$

$$\left\{\overline{0,1,2,3,8,9,10,11,14};\ \overline{4,5,6,7,12,13,14,15}\right\}$$

$$\pi^*_B = \pi^*_E = \left\{\overline{0,5,6,7,8,13,14,15};\ \overline{1,2,3,4,9,10,11,12}\right\}$$

realize the behaviour of block E.
Since machine B uses output information from machine A, but machine A does not use any output information from machine B, the general decomposition theorem can be slightly simplified:

1. $\pi_A \leq \pi^*_A$ and $\pi_B \cdot \pi'_A \leq \pi^*_B$, where $\pi'_A \geq \pi^*_A$
2. $\pi_A \cdot \pi_B \leq \pi^*_A$ and $\pi_A \cdot \pi_B \leq \pi^*_B$
3. $\pi^*_A \cdot \pi^*_B \leq \pi^*_E$

It should be noted that condition (3) is slightly modified, this is due to the fact that the partitions π^*_A, π^*_B and π^*_E are all partitions on I (there is no output decoder) and the partition pair property is reduced to the $\leq$ property.
It must be shown that the partition doubles satisfy all these conditions. As a result of the fact that each block of π_A is included in a block of π^*_A it follows $\pi_A \leq \pi^*_A$. Since all

x_3x_4	A
0 0	0
0 1	1
1 0	1
1 1	1

Table 2.4: Boolean function for block A

Ax_2	B (E)
0 0	0
0 1	1
1 0	1
1 1	0

Table 2.5: Boolean function for block B

x_1x_4	C
0 0	1
0 1	1
1 0	1
1 1	0

Table 2.6: Boolean function for block C

BC	D (F')
0 0	0
0 1	1
1 0	0
1 1	0

Table 2.7: Boolean function for block D

output information of machine A is used as input information for machine B, it can be assumed that $\pi'_A = \pi^*_A$ and condition (1) is then satisfied whenever condition (2) is also satisfied. For condition (2) the product

$$\pi_A \cdot \pi_B = \left\{\overline{0,8};\ \overline{1,9};\ \overline{2,10};\ \overline{3,11};\ \overline{4,12};\ \overline{5,13};\ \overline{6,14};\ \overline{7,15}\right\}$$

is calculated. Using this product, it is easy to see that condition(2) is satisfied. Finally, for condition (3) it is necessary to calculate

$$\pi^*_A \cdot \pi^*_B = \left\{\overline{0,8};\ \overline{4,12};\ \overline{1,2,3,9,10,11};\ \overline{5,6,7,13,14,15}\right\}$$

With this product partition, it is obvious that condition (3) is satisfied and hence the decomposition is correct. The functions for block A, B $(=E)$, C and D $(=F')$ can be found in Tables 2.4, 2.5, 2.6 and 2.7 respectively. It is interesting to note that this realization uses different gates: OR, NAND, EXOR and the gate $\overline{a} \cdot b$. All these gates are innately and directly obtained from decomposition without the use of technology mapping. □

2.5 Special cases of general decomposition

To the best of the author's knowledge, none of the published methods have been able to produce near-optimal solutions for a true general decomposition of combinational machines. All published results are based on some special cases of the general decomposition model. These special cases can be classified in many ways. In this thesis a classification based on the type of the connections of the components machines, the type of the input encoder and the type of the output decoder will be used. This classification was originally introduced in [Jóź95a].

Three types of connections between the component machines are being distinguished: general, parallel and serial. In parallel connections (see Figure 2.7 for a two component parallel decomposition), the component machines do not use information on the output of other component machines. In serial connections (see Figure 2.8 for a serial decomposition with two component machines), component machines use the output of other machines, but in such a way, that the information is used in only one direction, i.e. an ordering of the component machines exists and component machine i uses only information about the outputs of the combinational machines $j : j \leq i$. A decomposition with general, parallel or serial connections will be called general decomposition, parallel decomposition and serial decomposition, respectively.

For the input encoder only two types are recognized: the presence of the encoder and the absence of the encoder. In the latter case, the input bits are directly distributed to the component machines. Of course, only those bits are distributed that are necessary for the computations of the component machines. If a decomposition uses direct distribution of the input bits then it is called an input-bit decomposition (Figure 2.9), if the encoder is present, then no special prefix is added to the name.

For the output decoder the naming convention is similar to that of the input encoder: a decomposition is called an output-bit decomposition (Figure 2.10) when the decoder is absent (i.e. the output bits of all component machines together form the outputs of the original combinational machine). If a decomposition is both an input-bit decomposition and an output-bit decomposition then it is simply called a bit decomposition.

In Section 3.3 a number of special cases of the general decomposition reported in literature will be discussed. These decompositions will be classified using the terminology presented in this section.

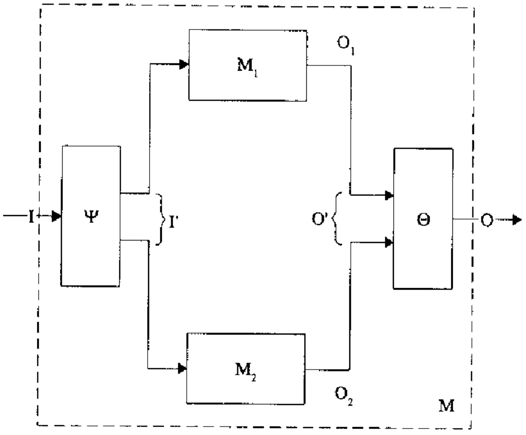


Figure 2.7: Parallel decomposition into two component machines

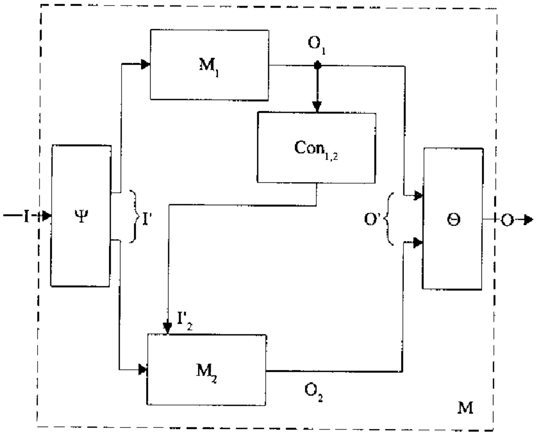


Figure 2.8: Serial decomposition into two component machines

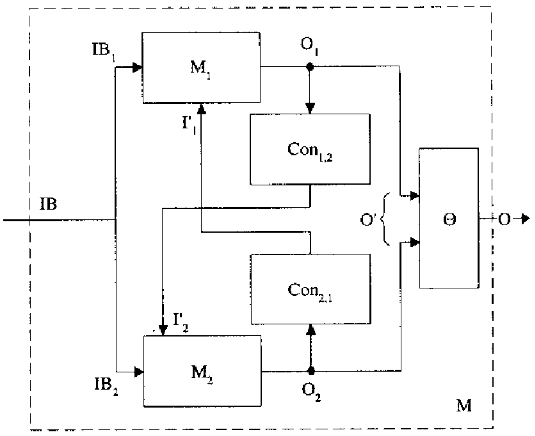


Figure 2.9: Input-bit general decomposition into two component machines

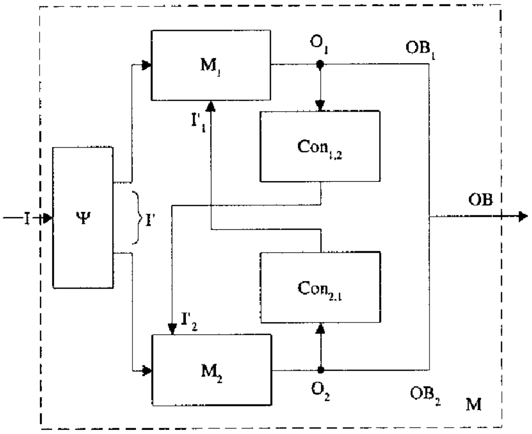


Figure 2.10: Output-bit general decomposition into two component machines

Chapter 3

State of the art in logic synthesis

This chapter overviews the state of the art in logic synthesis. For each of the considered methods its concepts and algorithms are briefly explained and discussed. In Chapters 4 and 5, conclusions of these discussions will be used for development of a new method for logic synthesis. If it is relevant, the partition theory will be used to show the difference between traditional methods and methods based on the concepts of general decomposition and partition theory. To keep related methods together, this chapter has been divided in four parts.
The first part of the chapter (Section 3.1) will contain methods originally developed for random logic synthesis based on the division approach introduced by Brayton. The division approach is the first and by far the most popular multiple-level logic synthesis paradigm. Although it is not primary intended for decomposition based logic synthesis, many of these methods have been extended for synthesis using FPGAs and some requirements for LUTSYN have been derived from a careful analysis of this approach, therefore division based synthesis methods have been presented in this thesis.
In the second part of this chapter methods for logic synthesis that use graph based representations of Boolean functions are briefly reviewed (Section 3.2).
In Section 3.3, the state of the art in logic synthesis primarily targeted on fine granularity FPGAs is presented. The methods considered in this part are the closest related to the method tackled in this thesis and therefore their careful analysis was necessary. LUTSYN's experimental results will be compared to the results obtained from these methods.
The last part of this chapter (Section 3.4) a few interesting synthesis methods that use multiple-valued encoding techniques will be discussed.

3.1 Division based logic synthesis

3.1.1 Factorization algorithm

The fundamentals for division based multiple-level logic synthesis were established by Robert K. Brayton in 1982 [BM82] [BCH$^+$82] [BRSW87]. In this section, Brayton's original method and a number of extensions to that method will be reviewed. The algorithm of Brayton will be called *standard factorization algorithm,* because, all other factorization algorithms are based on it. This section is a summary of a more elaborate discussion published in [VJS95].

The reader is assumed to be familiar with the following basic definitions used commonly in logic synthesis: a Boolean variable, a literal, a cube and a Boolean expression. The support $sup(f)$ of a Boolean expression f is the set of Boolean variables which appear either complemented or uncomplemented in the cubes of expression f.

In division based logic synthesis parenthesized Boolean expressions are used to define the structures of Boolean function implementations. Such a parenthesized expression is called a *factored form* and the process of obtaining a factored form of a Boolean expression is called *factorization.*
An *incompletely specified function* F is represented by a triple of completely specified Boolean functions: $F(f, d, r)$, where f represents the on-cover, i.e. all input space points (values) for which F evaluates to 1. Similarly, d and r are representations of the don't care-cover and the off-cover, respectively. Every possible input vector belongs to precisely one of these covers. If $d = 0$ then the function F is called *completely specified.*

It is possible to calculate the product of two Boolean expressions f and g by simply applying the AND operator. If f and g have disjoint input supports, then $f \cdot g$ is called the *algebraic product*, otherwise $f \cdot g$ is called the *Boolean product.*

Definition 3.1 *Boolean divisor*
A completely specified function g is a Boolean divisor of an incompletely specified function $F = (f, d, r)$, if two completely specified functions h and i exist so that

$$g \cdot h \not\subseteq d \tag{3.1}$$

and

$$f \subseteq g \cdot h + i \subseteq f + d \tag{3.2}$$

where $g \cdot h$ represents the Boolean product. □

h is called the quotient and i is called the remainder of the division, similar to the names used in number calculus. However, in a Boolean algebra h and i are not necessarily unique.

Example 3.1 *Boolean division*
The Boolean function

$$F(a, b, c, d, e) = a \cdot c + \overline{a} \cdot b + b \cdot c + d + \underline{e}$$

where the underlined cube is a cube from the don't care set, has a Boolean divisor $(a + b)$, with a quotient $(\overline{a} + c)$ and a rest expression d, because $(a + b) \cdot (\overline{a} + c) + d = a \cdot c + \overline{a} \cdot +b \cdot c + d \subseteq F$. Because $(a + b)$ and $(a + \overline{c})$ have intersecting input supports, the product is a Boolean product. □

The goal of division based synthesis is to find Boolean divisors to factorize the Boolean function. The disadvantage of this approach is that the set of Boolean divisors is usually very large and therefore good divisors cannot be found easily. Consequently, an alternative approach has been proposed which considers only a special case of division. It is

based on the fact that in the sum-of-product term representation of a Boolean function efficient common algebraic factors can be identified as being common factors of the product terms. The idea is motivated by the fact that manipulations of sum-of-product terms are in most cases quickly performed (many algebraic operations have linear time complexity). The disadvantage of this idea is that it does not guarantee optimal solutions, because it limits the considered solution space a priori to solutions that can be obtained with only algebraic division.
Brayton introduced an alternative approach based on this idea [BRSW87][BHS90]. In this approach, the incompletely specified function F is minimized to obtain a two-level minimal representation of the on-set f of F (using the two-level minimizer Espresso [BHMS84]) and algebraic division is used to manipulate f.

Definition 3.2 *Algebraic divisor*
A completely specified function p is an algebraic divisor of a completely specified function f if q and r exist, such that $p \neq 0$, $q \neq 0$ and

$$f = p \cdot q + r \tag{3.3}$$

where $p \cdot q$ is the algebraic product. □

q is once again called the quotient; the quotient will be denoted as $q = f/p$; r is called the *remainder* of the division; q and r need not be unique. However, if q is defined as the largest set for which $f = p \cdot q + r$ then q and r are unique. This special type of algebraic division is called *weak division*. Note that in $f = p \cdot q + r$, the names of the divisor and the quotient are arbitrary, if p is a divisor and q is the associated quotient, then q can also be called the divisor and p can be called the quotient.

Example 3.2 *Algebraic division*
$(a + b)$ is an algebraic divisor of the function $f = (a + b) \cdot (c + d + e) + g$. Both $q = c + d$, $r = a \cdot e + b \cdot e + g$ and $q = c + d + e$, $r = g$ are valid quotient/remainder pairs for f divided by $(a + b)$. If the calculations are restricted to weak division then the second pair is the unique quotient/remainder pair. □

An expression is said to be *cube free* if there exists no cube that is part of all the cubes in the expression (e.g. $a \cdot b + c$ is cube free, but $a \cdot b + a \cdot c$ is not cube free, since cube a is part of both $a \cdot b$ and $a \cdot c$.

Definition 3.3 *Primary divisors*
The primary divisors of a Boolean function f are the elements of the set $D(f)$ defined as:

$$D(f) = \{f/c \mid c \text{ is a cube}\} \tag{3.4}$$

□

Definition 3.4 *Kernels*
The set of kernels of a Boolean function f is the set:

$$K(f) = \{g \mid g \in D(f) \wedge g \text{ is cube free}\} \tag{3.5}$$

□

In other words, the kernels of an expression f are the cube-free, primary divisors of f. The cube c used to obtain kernel $k(k = f/c)$ is called the *co-kernel* of k, and $C(f)$ is used to denote the set of co-kernels of f.

Example 3.3 *Kernels*
For the function $f = a \cdot (b + c) + d$ the set of divisors is

$$D(f) = \{a;\ b + c;\ a \cdot (b + c) + d;\ 1\}$$

The set kernels for this function is:

$$K(f) = \{b + c;\ a \cdot (b + c) + d\}$$

□

Theorem 3.1 *Existence of multiple cube divisor*
Functions f and g have a common multiple cube divisor d if and only if

$$\underset{k_f \in K(f)}{\exists}\ \underset{k_g \in K(g)}{\exists}\ d = k_f \cap k_g$$

□

This theorem (proved in [BM82]) states that two functions only have a multiple cube divisor if an intersection of a kernel of f and a kernel of g has more than one cube. It is the fundamental theorem used in the standard factorization algorithm of Brayton.

The optimization goal of the standard factorization algorithm is to minimize the number of literals in the factorized function. This is a quite accurate measure for the implementation complexity of small AND-OR-NOT based Boolean expressions in CMOS, because in CMOS technology, each literal corresponds to 2 transistors (one PMOS and one NMOS) [BRSW87]. Although this measure is reasonably accurate for the active area of a physical implementation, it is not adequate enough for the routing area of the circuit. Since the routing area of complex multiple-level logic circuit can be much larger than the active area and heavy parenthesized expressions increase the delay, the number of literals can be a weak selection criterion for large logic circuits. If more complex gates are possible (like the AND-NOR-21 gate: $f = \overline{a \cdot b + c}$) then a non-trivial technology mapping is required as this will transform a set of Boolean expressions with minimum literal count into a gate network using the minimum amount of area. This problem is known to be NP-hard. The same problem can occur when the Boolean expression uses operators with too many inputs: for example, the expression $f = a \cdot b \cdot c \cdot d \cdot e \cdot h \cdot i \cdot j$ probably requires technology mapping, because not so many technology libraries support 8 inputs AND gates.

The standard factorization algorithm can be summarized as follows: each kernel/co-kernel pair obtained using the multiple cube divisor theorem is applied to all functions and the gain in the number of literals obtained by this pair is calculated. The kernel/co-kernel pair with the highest literal gain is selected and applied to all functions. The algorithm continues to select the kernel/co-kernel with the highest gain in the number of literals, until the gain is smaller than a threshold value X. Then, all single-cube divisors which have a literal gain larger than X are extracted. If no more multiple-cube or single-cube divisors can be found then the threshold value X is decreased and the extraction process is continued.

The standard factorization algorithm has a few drawbacks:

1. The input to the algorithm is the two-level, minimized on-set f' of an incompletely specified function $F(f, d, r)$ obtained using the two-level minimizer Espresso [BHMS84]. This makes the minimization result much easier to handle than the original incompletely specified multiple output functions, but the loss of design freedom expressed by the don't cares of the original specification before the division algorithm actually starts will almost certainly lead to a less satisfactory implementation.

2. The method is based on finding kernel/co-kernel pairs. Kernels form however only a very small subset of all algebraic divisors of a function. Algebraic divisors are a special subset of another group of divisors: the Boolean divisors. Therefore, the set of kernels of a Boolean expression is a very small subset of all possible divisors and can be too restrictive to find near-optimal solutions.

3. The kernel/co-kernels of Brayton are not *algebraically compatible*. This means that after choosing and applying a certain kernel, other kernel/co-kernel pairs may no longer be valid or their quality may have been changed. Therefore, after each kernel selection, the kernel/co-kernel set needs to be recalculated. This is a rather expensive operation and therefore the standard factorization algorithm selects a few kernels/co-kernels before rebuilding the kernel/co-kernel set, which means that the algorithm works with somewhat inaccurate quality estimations and furthermore some extra checks are required to assess whether a kernel/co-kernel pair is still feasible.

4. The algorithm for selection of kernels is a greedy algorithm and it is based on the momentary (local) gain of the number of literals of a kernel. It does not predict the total gain in the number of literals that can be expected by choosing this kernel. This is unfortunate because the choice of a kernel can block some other possible valuable kernel/co-kernel pairs and, after choosing the currently best kernel, no good kernel/co-kernel pairs may remain, whereas after selecting, for example, the second good kernel some other good pairs could remain.

To overcome some of these drawbacks a lot of extensions have been proposed by other researchers. Perhaps, the two most useful extensions are the following:

1. **Lexicographical factorization**
   The goal of lexicographical factorization [ASSP90] [ABS+91] [ASP91] [SFA93] is to improve the routing factor of the multiple-level implementation. In the lexicographical factorization, the variables of each output function are factored out in order of their appearance in a certain variable ordering. For example, for the function $f = a \cdot b \cdot c + a \cdot b \cdot d + a \cdot e + b \cdot \overline{c} + b \cdot e + \overline{c} \cdot d$ and an input order $\{a, b, c, d, e\}$, the factorized form of f with respect to this order is $f = a \cdot (b \cdot (c + d) + e) + b \cdot (\overline{c} + e) + \overline{c} \cdot d$. The result is that inputs no longer need to be distributed globally over the circuit tree; variables in the beginning of the variable order are needed at the base of the function tree, whereas the last variables are used near the roots.
   Part of the input order can be imposed externally to account for external factors (e.g. late arrival times of some input). If this enforced input ordering is not complete then the kernel/co-kernel pair with the largest gain in the number of literals that does not violate the input order is selected and the input order is updated. Selecting kernel/co-kernel pairs is then repeated until the input order is completed. Since the variable ordering is known, factorization is very simple. Negated variables are factored out immediately after the non-negated variables. Because of the input order, the search for common subexpressions is very efficient.
   One of the main advantages of lexicographical factorization over standard factorization is the fact that lexicographical compatible kernel/co-kernel pairs are also algebraically compatible. Therefore, lexicographical factorization does not require the recalculation of the set of kernel/co-kernel pairs after selection of each pair from the set. The result is that lexicographical factorization is much faster compared to standard factorization.
   Experimental results ([SFA93]) show that the lexicographical factorization results in implementations which have much smaller routing areas compared to the standard factorization algorithm of Brayton, however they use larger active area because respecting the variable ordering prevents some good kernels from being selected.

2. **Concurrent decomposition algorithm**
   The characteristic feature of the concurrent decomposition [RV90][VR90][RV92] method is that it limits itself to the use of double cube divisors (i.e. kernels with only two cubes), single cube divisors with only two literals and the complements of these single and double cube divisors. It has been found that these divisors (in spite of their simplicity) can be used to synthesize circuits with small area and short delay times. Furthermore, by restricting the calculations to double cube divisors and single cube divisors with two literals, the calculations of these divisors are now polynomial time operations (whereas the calculation of the set of kernels can require an exponential amount of time).
   The divisor selection algorithm is a greedy algorithm selecting the (single or double cube) divisor (and possibly its complement divisor) with the largest gain in the number of literals. The set of double cube divisors is not algebraically compatible. However, because the complexity of the double cube generation algorithm is quadratic, this algorithm is much faster when compared to the standard factorization (which occasionally can have exponential time complexity).

Originally, concurrent decomposition has been developed with design for testability in mind. Concurrent decomposition is based on testability preserving transformations and the factorized multiple-level network is fully testable by a complete test set derived from the original two-level circuit. In order to preserve testability the algorithm has to be used on single output functions and not on multiple output functions. This means that good subexpressions cannot be shared among different output functions. Furthermore, the transformation of a multiple output function to a set of single output functions can increase the number of product terms by at most a factor equal to the number of outputs in the multiple output function. It seems reasonable to expect further gain in the number of literals if the testability condition will be dropped and common subfunctions and common double cube divisors will be allowed.

3.1.2 Example of division based synthesis

In this section the discussed division based algorithms are illustrated with an example. The function $w(a,b,c,d,e,f,g)$ defined as

$$w(a,b,c,d,e,f,g) = a \cdot b \cdot c \cdot e + a \cdot b \cdot d + \overline{a} \cdot e \cdot f + \overline{b} \cdot e \cdot f + e \cdot f \cdot g$$

will be used in this example. w is a single output completely specified function and is minimal with respect to the number and the size of the terms.
The standard factorization algorithm presented in Section 3.1.1 applied to w results in the function $x(a,b,c,d,e,f,g)$:

$$x(a,b,c,d,e,f,g) = a \cdot b \cdot (c \cdot e + d) + e \cdot f \cdot (\overline{a} + \overline{b} + g)$$

(see also Figure 3.1); it is a solution that uses 10 literals.
The lexicographical factorization algorithm cannot find this solution because the used kernel/co-kernel pairs are incompatible: the pair $(c \cdot e + d, a \cdot b)$ requires (among others) the precedence relation a precedes e, whereas the second kernel/co-kernel pair of x $\left(\overline{a} + \overline{b} + g, e \cdot f\right)$ requires the precedence relation e precedes a, which is incompatible with the first relation. The lexicographical factorization finds the following factorization and respects the variable ordering $\{d, e, c, f, a, b, g\}$:

$$y(a,b,c,d,e,f,g) = d \cdot y_1 + e \cdot (c \cdot y_1 + f \cdot (\overline{y_1} + g))$$

with

$$y_1 = a \cdot b$$

(see Figure 3.2). The function also requires 10 literals. In this case, a subfunction y_1 has been introduced. Because the lexicographical factorization algorithm uses NAND functions as an internal representation, it was able to identify $a \cdot b$ and $\overline{a} + \overline{b}$ as common subexpressions.
If the realization obtained by lexicographical factorization is compared with standard factorization then it shows the properties of lexicographical factorization: each input is

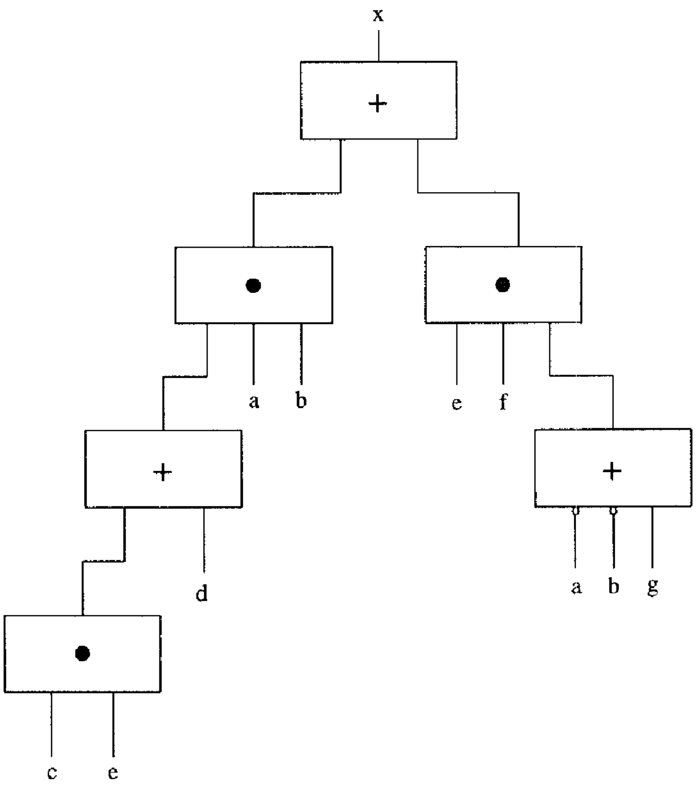


Figure 3.1: Function x found by standard factorization

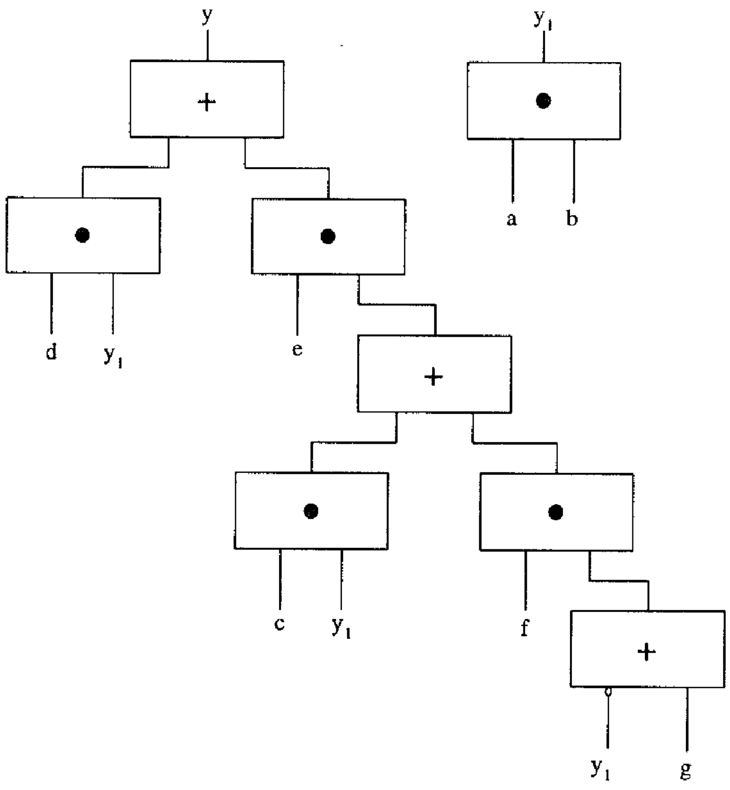


Figure 3.2: Function y found by lexicographical and concurrent factorization

only once connected to a gate and therefore the routing complexity for the inputs is reduced. In most cases, the lexicographical factorization algorithm needs more gates than standard factorization (increase of active area). The real power of the lexicographical factorization algorithm can only be shown on large examples, where the gain in active area and the extra routing area for subfunctions is compensated by a large reduction in the routing area for the inputs. The benchmark results in [SFA93] illustrate the effectiveness of the lexicographical factorization for large circuits.

The concurrent factorization algorithm searches explicitly for the complement of the kernel $\overline{a} + \overline{b}$, whereas in lexicographical factorization this equivalence was implicitly found. The function z found by the concurrent factorization is the same factorization as lexicographical factorization (function $z = y$) (see Figure 3.2). As it can be seen only double cube divisors and cubes with two literals have been used.

3.1.3 Technology mapping

The previous sections have dealt with techniques for the minimization of Boolean expressions. The Boolean expressions used are a technology independent description and hence must be translated into a network of blocks that are available in the library of the implementation technology. The blocks from the implementation library are called the basic building blocks. In case of FPGAs, the implementation library consists of all possible configurations of the programmable block used in a given FPGA.
The translation process from a Boolean expression (technology independent logic network) to a network of basic building blocks of a certain technology is called *technology mapping*. In this section a number of techniques for technology mapping will be discussed.

The technology independent logic synthesis is only useful if the work done by it is preserved by the technology mapping. This means that the structure obtained from the logic synthesis process should not be destroyed by the technology mapping; i.e. the implementation library should offer at least those basic building blocks that can implement the logic operators used in the synthesized structure. E.g. for the logic synthesis of random multiple-level logic based on the factorization of Boolean expressions the implementation library should at least include AND gates, OR gates (with all input sizes used in the Boolean expressions) and an INVERTER. Technology mapping is generally harder and less effective if the distance between the available blocks and the used operators is larger.

A number of approaches for technology mapping exist. Essentially these methods are all derivations from a technique called *pattern matching*. In this technique the Boolean expression is represented by a graph. The vertices of the graph represent the Boolean operators and the edges are the interconnections between these operators. Likewise, the blocks of the technology library are represented by a graph of their Boolean functions. The pattern matching technique then consists in covering (i.e. "matching") all vertices and edges of the graph of the Boolean expression with the graphs from the technology library. If a node cannot be matched in any way the node is called *infeasible* and logic

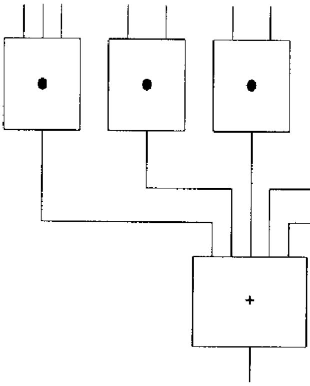


Figure 3.3: Chortle initial network : Two-level sum-of-product expression

synthesis must be used to find a network of feasible nodes that provides the same functionality.
The most typical goal of technology mapping is to find a mapping that uses as few as possible basic building blocks and minimizes the number of edges (interconnections) between these blocks. Sometimes, other optimization criteria should be fulfilled as well, for example the minimization of the sum of the active area size used by the blocks. The technology mapping problem is known to be NP-complete.
In [CSS91] pattern matching based on perfect and semi-perfect matching has been introduced. A perfect matching is a pattern that matches exactly to a subtree of the Boolean expression tree. A semi-perfect matching, is a perfect matching that is obtained after splitting of a node into a tree of nodes. The algorithm is applied to lexicographical factorized expressions. In [MBS92] a similar method is used for the mapping on Xilinx configurable logic blocks for sequential machines. The internal structure of a Xilinx logic block can be programmed (configured) in a large number of ways. In this paper 18 different configurations for the Xilinx configurable logic block have been distinguished. This approach shows the complexity of using pattern matching (and technology mapping in general) for complex FPGAs.

Chortle-crf [FRV91a] is the successor of the Chortle algorithm [FRC90]. Both approaches represent a Boolean expression as a tree and try to determine an optimal mapping of this tree. The results of Chortle-crf are much better than those of Chortle and it is much faster. Therefore, Chortle is not discussed here any further. The starting point of the Chortle-crf algorithm are the two-level Boolean expressions (Figure 3.3). The first step of this algorithm is to create a two-level decomposition consisting of a set of LUTs as the first level and single OR that connects the outputs of the LUT as the second level (Figure 3.4). *A constructive bin packing* algorithm is used to find two-level decompositions, where the bins are the LUTs and the AND terms from the sum-of-product table

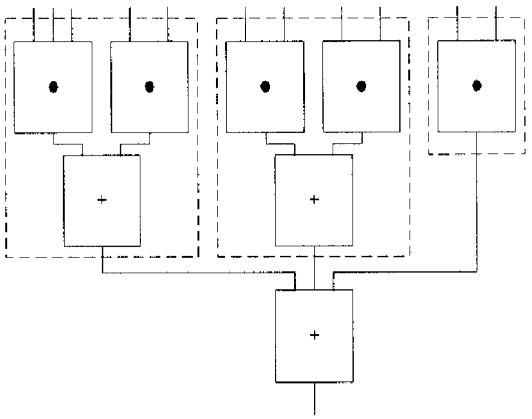


Figure 3.4: Chortle step 1 : Two-level decomposition

are packed in them. The second step of the algorithm, a multiple-level decomposition is created by using the unused inputs of the LUTs to implement part of the second level logic (Figure 3.5). The algorithm is also able to use reconvergent paths and replication of gates to find better solutions; however these last two techniques are local instead of global optimizations. In [FRV91b] the authors of Chortle present a version of Chortle called Chortle-d that reduces the depth in the final circuit. It produces fast circuits, but uses more LUTs than Chortle-crf.

In [MSBS91a] the successor of an algorithm called *mis_pga* [MNS$^+$90] is presented. This successor is simply called *new mis_pga*. Both algorithms perform technology mapping on LUTs. Mis_pga is based on Boolean expressions obtained using the standard factorization algorithm of Brayton. If an infeasible node has been created, Roth-Karp decomposition (see Section 3.3.1) or partitioning based on kernel extraction is used to make the node feasible. The algorithm then tries to collapse nodes into a single new feasible node. The results of mis-pga have been improved by Chortle-crf. A derivation of mis-pga targeted for minimizing the delay has been describe in [MSBS91b]. The delay is calculated using a k by k grid (representing the topological structure of a Xilinx FPGA) and a simulated annealing algorithm that tries to minimize the delay by swapping nodes in the grid. The critical path delay of this algorithm is better than those produced by Chortle-d.
New mis_pga uses a technique very similar to Chortle-crf: *cube packing*. In this approach the cubes of the functions are considered as items and the Xilinx configurable logic blocks are the bins. New mis_pga can also use Shannon decomposition, AND-OR decomposition (each node is either a 2-input AND, 2-input OR or an INVERTER node and the disjoint decomposition techniques used by mis_pga. It is however unclear in

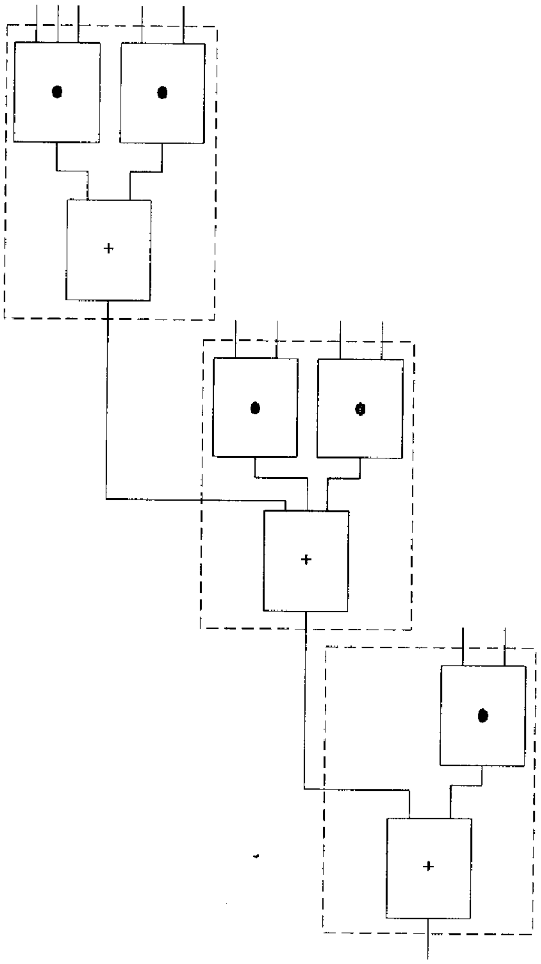


Figure 3.5: Chortle step 2 : Multiple-level decomposition

which order and in which situations these algorithms are used.

Technology mapping is necessary if logic synthesis is performed without taking into account the hardware implementation style that is going to be used. Technology mapping is a difficult problem, nevertheless a number of good algorithms have been constructed for it. Unfortunately, a number of fundamental drawbacks still exist that argue for the integration of logic synthesis and technology mapping into a single phase synthesis process. These drawbacks are listed below:

1. Separated logic synthesis and technology mapping are only useful if the operators used in the synthesized Boolean function representation correspond to those offered by the technology library. Optimal mapping results can only be obtained if the logic synthesis process uses the full set of hardware building blocks provided by the implementation technology.

2. By not taking into account the basic building blocks of the technology library, the logic minimization may produce infeasible nodes. Technology mapping can only remove these infeasible nodes by (locally) redoing the work of the logic synthesis algorithm.

3. Merging of nodes and bin packing is generally bad for the routing complexity of FPGAs because these techniques do not take into account the positions of the nodes to merge in the layout. These algorithms should take into account positions of the nodes to merge, to prevent long and expensive wires in the final layout.

4. Library based technology mapping is infeasible for modern LUT based FPGAs [JP95].

3.2 Graph based logic synthesis

3.2.1 Introduction

Besides division based multiple-level logic synthesis another important trend in multiple-level logic synthesis exists. It consists of algorithms based on graphical representations of the logic synthesis problem. Most work in this area is based on the concept of *binary decision diagrams (BDDs)* introduced by Bryant [Bry86]. To this basic concept many extensions have been published in the past ([Bry92]). BDDs and the most important extensions on them are the subject of this section.

3.2.2 Classical binary decision diagrams

Binary decision diagrams are not new, they have been introduced in the past by Lee [Lee59] and Akers [Ake78]. However, their current popularity can be attributed to Randal Bryant. In his fundamental paper [Bry86] the use of the mathematical concept of a binary decision diagram for the representation and manipulation of Boolean functions has been introduced. The following definitions of binary decision diagrams have been introduced in [Bry86].

Definition 3.5 *Binary Decision Diagram (BDD)*
A binary decision diagram (BDD) is a directed acyclic graph (DAG). The graph has two sink nodes labeled 0 and 1 representing the Boolean functions 0 and 1. Each non sink node is labeled with a Boolean variable v and has two outgoing edges labeled 1 (also called *then* or right) and 0 (or *else* or left). □

The Boolean function corresponding to a non sink node labeled with variable v is defined as the Boolean function corresponding to the node's "then" child in case the Boolean variable $v = 1$. If the variable v equals to 0, then the Boolean function corresponding to the node, is the Boolean function corresponding to the node's "else" child.
An example of the graphical representation of a BDD can be found in Figure 3.6. A node is represented by a circle labeled with the Boolean variable that is tested in this node. Sinks are represented by squares. Each node has two outgoing edges, the edge on the left is the else edge, the edge on the right is the then edge.
A special, but very important type of BDDs are the ordered binary decision diagrams (OBDDs).

Definition 3.6 *Ordered Binary Decision Diagram (OBDD)*
An ordered binary decision diagram is an ordinary BDD extended with an ordering on all input variables. In every path from source to sink in the OBDD the inputs of the visited nodes respect this ordering. □

For example, the BDD in Figure 3.6 is an OBDD that respects the input ordering $(x_1, x_2, x_3, x_4, x_5, x_6)$.

Definition 3.7 *Reduced Ordered Binary Decision Diagram (ROBDD)*
A reduced ordered binary decision diagram (ROBDD) is an OBDD with the minimal number of nodes. It can be obtained from a OBDD by applying the following steps to the OBDD:

1. Merge all 0 sink nodes to a single 0 sink node.
2. Merge all 1 sink nodes to a single 1 sink node.
3. Merge all nodes that represent the same Boolean subfunction.

□

ROBDDs are the fundamental concept in BDD based Boolean function representation and logic verification, because it has been proved in [Bry86] that all ROBDD representations of a certain Boolean function f (with the same input ordering) are isomorphic, and hence a ROBDD representation for a certain input ordering $(x_1, x_2, ..., x_n)$ is canonical. The canonicity of ROBDDs has an important consequence: it is trivial to check for equivalence of two Boolean functions, because this test consists in determining whether the ROBDDs of the two functions (with respect to the same input ordering) are isomorphic.

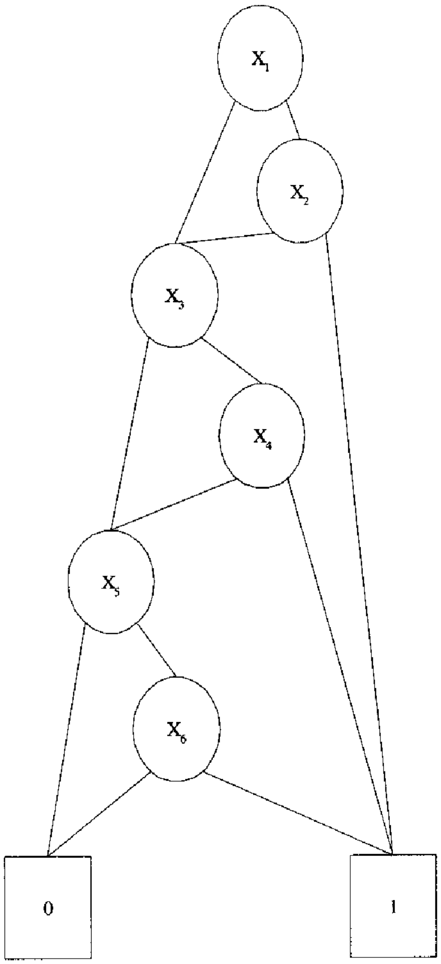


Figure 3.6: BDD representation of function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ using input ordering $(x_1, x_2, x_3, x_4, x_5, x_6)$

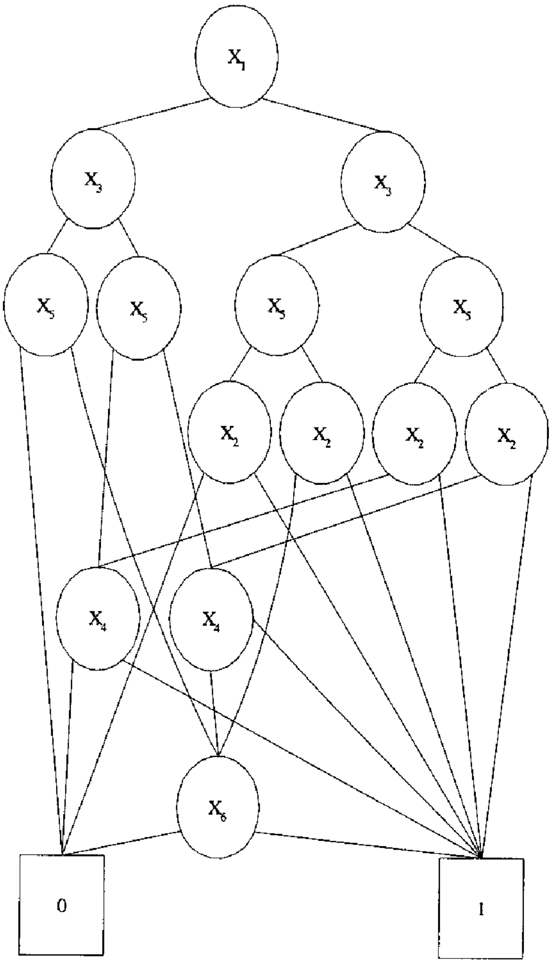


Figure 3.7: BDD representation of function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ using input ordering $(x_1, x_3, x_5, x_2, x_4, x_6)$

The main advantage of ROBDD representations is the efficiency of the operations that can be performed on them. An important operation is the *reduce operation* which transforms an OBDD to a canonical ROBDD. The complexity of reduce is $O(\mid G \mid \cdot log(\mid G \mid))$ where $\mid G \mid$ is the number of nodes in the graph. Another important operator is the *apply operation*. It can be used to perform a Boolean operation on two ROBDDs. The complexity of apply is comparable to the product of the sizes of the two ROBDDs. Unfortunately after the use of the procedure apply a reduction step is required. Details of these and other operations can be found in [Bry86]. The operations have an important property: they preserve the input ordering.

The main difficulty in ROBDDs synthesis is the determination of a good input ordering for a certain (set of) functions. A classical example [Bry86] can be found in Figure 3.6 and Figure 3.7. Both graphs in this figure represent the function

$$f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$$

but the ROBDD in Figure 3.6 uses input ordering $(x_1, x_2, x_3, x_4, x_5, x_6)$ whereas, the ROBDD presented in Figure 3.7 uses the ordering $(x_1, x_3, x_5, x_2, x_4, x_6)$. It can be seen very easily that the representation in Figure 3.6 is much more compact (and therefore much easier to work with) compared to the BDD in Figure 3.7.
There is no general rule for finding a good (suboptimal) input ordering. A number of heuristic solutions exist (see Section 3.2.3). Moreover, some classes of Boolean functions (like multipliers) cannot be represented efficiently with ROBDDs regardless of the chosen input ordering. Therefore ROBDDs are not suitable for manipulating this type of Boolean functions and alternative representations are required for them.

In general, ROBDDs are a useful way to represent Boolean functions, apart from the following drawbacks:

1. The size of a ROBDD and the time to construct the ROBDD can be large for certain functions and it is difficult to predict which functions will require a large ROBDD. Even worse, it has been proved that some classes of Boolean functions can never be represented using the ROBDD representation regardless the input ordering that is used. For example in [Bry86, Appendix], it has been proved that functions describing the outputs of an integer multiplier have graphs that grow exponentially in the word size regardless of the input ordering.

2. The influence of the variable ordering on the size of the ROBDD is large, which makes the choice of a good variable ordering essential. Unfortunately, there exists no method that would always find a good variable ordering.

3. Operations on two ROBDDs require that these two ROBDDs use the same input ordering; it has been proved that performing an operation on two BDDs with different input orderings is NP hard [GM93]. This means that an input ordering has to been found that is good for both ROBDDs, even then, the input ordering may not be a good choice for the resulting ROBDD.

4. Don't cares cannot be handled using ROBDDs (there is no don't care sink). This is a very important shortcoming since, especially in multiple-level logic synthesis, the smart use of don't cares is of vital importance to construct optimal multiple-level Boolean networks. In the section a number of extensions to ROBDDs that will be able to model don't cares will be discussed.

3.2.3 Extensions to BDDs

Despite the disadvantages mentioned in the previous section, ROBDDs are useful representations. In this section a number of extensions of the classical BDDs will be presented. The extensions attempt to overcome the drawbacks mentioned above. An elaborate discussion of many of these extensions can be found in [Bry92].

- In [MWBS88] an ordering heuristic for the input variables is described. It is based on an observation that the network representation of a Boolean function is correlated with the BDD of this Boolean function. The correlation suggests that the network topology can be used to guide the variable ordering, and inputs should to be ordered by their level in the multiple output network. This agrees with results obtained by Saucier e.a. [BBCS92] where the variable ordering for lexicographical factorization is also a good ordering for BDDs.

- In [BRB90] a new implementation of a BDD package is introduced. By using a number of programming tricks (hash tables, caches) a faster package compared to Bryant's implementation has been obtained. Also a number of ROBDD extensions are considered, the most important among them are the *complemented edges*: a complement edge is an ordinary edge with a extra bit set to indicate that the corresponding formula is to be interpreted as the complement of the ordinary formula. If the rule "the then edge of every node must be a regular edge" is observed then the canonical form is preserved. Experiments show reduction in the size of the ROBDD and a factor 2 reduction in the construction time of the ROBDD.

- In [CJZT91] MBDs (modified BDDs) are introduced. The major new aspect of this BDD type is the addition of an additional sink for don't cares. This sink makes it possible to construct BDDs for incompletely specified output functions. As a result of this addition the equivalence of two BDDs is so much easier checked, because this test requires the operation $f_{on} + f_{dc}$. In traditional BDDs this is the OR of two BDDs, with MBDs this is much easier accomplished by making the don't care sink a 1-sink.
Another interesting new feature of MBDs is an algorithm which allows to swap two adjacent inputs in the input ordering. A feature of this algorithm is that only two levels of the MBD need to be reconstructed not the entire MBD. In a follow up article [CZJ+92] a simulated annealing like algorithm (called stochastic evaluation) is used to find pairs of inputs in the ordering to swap to obtain a good ordering. This algorithm finds very good input orderings but is very slow.

- In [Jóź97b] term trees (or don't care BDDs) have been introduced. A term tree is a BDD in which each decision node has an extra outgoing edge, located between

the "then" and the "else" edge, that is called the "don't care" edge. A term tree can be used to compactly represent a set of product terms. In the article a heuristic algorithm is presented that is able to find term trees with the minimal number of nodes. Experimental results show that this algorithm is very effective and efficient: (near)-optimal results have been obtained for a set of 20 small, medium and large benchmarks.
Term trees have been used for the automatic test pattern generation (ATPG). In the same article [Jóź97b] it is shown how two types of faults (stuck-at-zero and stuck-at-one) can be identified in the term tree. Based on this identification a heuristic algorithm is proposed that it is able to find a minimal test pattern set that guarantees 100% coverage of all non-redundant faults. Experimental results show that the ATPG algorithm is very effective and efficient.

- It has been known for a long time that for some types of Boolean functions (notably arithmetic functions) EXOR based representations can be considerable smaller compared to purely AND-OR representations. This is the reason for the interest in Reed-Muller and AND-EXOR representations of Boolean functions. Recently, a number of authors have recognized that this fact can also be applied to BDDs.
In [SDG95] another attempt to minimize the size of ROBDDs is presented. In this paper Free Boolean Diagrams (FBDs) are introduced. The Boolean function of a node of a FBD can be: an input variable (like in ordinary BDDs), the product of the left and the right branch, or the exclusive-or of both branches. This approach allows the efficient representation of exclusive-or based Boolean functions, a class of Boolean functions that cannot be efficiently implemented using ordinary BDDs. The results show that the FBDs can reduce the graph size by about 20% on average. However, a few large benchmarks had a factor 6 reduction in their graph size. The FBD package requires on average 1.4 times more CPU time compared to the standard ROBDD package.

- The Free Boolean Diagram mentioned above show one way how these EXORs can be applied. Another interesting approach has been taken in [DST+94][PCSS95] by the introduction of Kronecker Functional Decision Diagrams (KFDD). A KFDD is a decision diagram where the nodes can represent three different types of functions:

  1. Shannon expansion : $f = \overline{x_i} \cdot f_i^0 + x_i \cdot f_i^1$
  2. Positive Davio expansion : $f = f_i^0 \otimes \left(x_i \cdot (f_i^0 \otimes f_i^1)\right)$
  3. Negative Davio expansion : $f = f_i^1 \otimes \left(\overline{x_i} \cdot (f_i^0 \otimes f_i^1)\right)$

  where $f_i^j = f(x_1, ..., x_{i-1}, j, x_{i+1}, ..., x_n)$ and where n represents the number of input variables. This way KFDDs can mix AND-OR nodes (the Shannon expansion) with EXOR based Davio expanded nodes [KSR92]. The construction process for KFDDs can use a number of different heuristic algorithms to determine the best expansion type for each nodes. Many derivations and generalizations of KFDDs have been proposed. An overview is presented in [PCSS95].

- Karplus [Kar92] [Kar88b] has generalized the BDD concept, to that of if-then-else direct acyclic graphs (ITE DAGs). The function of a node in a ITE DAG is represented by the three tuple (a, b, c) that expresses the function: $a \cdot b + \overline{a} \cdot c$ i.e. "if a then b else c"). The only difference between a BDD and a ITE DAG is that a BDD uses a single variable in the if-part whereas the if-part of an ITE DAG may be formed by any Boolean expression.
  Karplus has determined the necessary conditions on the sub-DAGs allowed as the if, then and else part of a ITE node to make canonical representations using DAGs [Kar92] and he has studied the ways Boolean functions can be best represented using ITE DAGs [Kar88a]. These two factors show the potential usefulness of these DAGs. ITE DAGs are also used for the logic synthesis of multiple-level Boolean functions; see the next section for more details.

3.2.4 Logic synthesis using BDDs

In the previous sections BDDs have been considered as a representation for the values of Boolean functions. Some attempts have also been made to use ROBDDs for representing the structure of a Boolean function, i.e. to use ROBDDs for the logic synthesis of combinational circuits.
Since the function corresponding to a node of a ROBDD is in fact a multiplexer, ROBDDs are feasible structures for modeling multiple-level multiplexer networks. ROBDDs have the restriction that the selection function of a multiplexer has to be a primary input variable, hence ROBDDs do not provide a general model for multiplexer networks. ITE DAGs allow any Boolean function to be the selection function, and hence they form a general representation for multiple output multiplexer networks.
In this section a number of logic synthesis algorithms based on ROBDD variants will be discussed.

Most graph based logic synthesis methods, use ACTEL FPGAs [Act95] as the implementation technology for the function. The basic combinational cell used in the ACT-1 series FPGAs can be found in Figure 3.8, the basic cell for the ACT-2 and ACT-3 series is represented in Figure 3.9. The difference between these two types of cells is that the SA and SB lines from ACT-1 cells are and-ed together in the ACT-2 and ACT-3 families. Both types of cells can be seen as a 4-input multiplexers; BDD networks can be mapped quite efficiently on the multiplexer networks.

Karplus as one of the first used direct acyclic graphs for the logic synthesis of Boolean functions. In [Kar90] [Kar91a] an algorithm for mapping on ACT-1 cells is presented. The algorithm takes a ITE-DAG as its input and transforms it to an ACT-1 cell network. The approach consists of a recursive algorithm that starts at the roots of the tree, creates an ACT cell for this node and then invokes itself for its unfinished children. The algorithm attempts to use as much as possible of the ACT-1 cell, and uses a number different heuristics to find the best use of a cell.
In [Kar90] [Kar91b] a similar method targeted to Xilinx FPGAs is presented. The algorithm consists of a three phase approach: in the first phase nodes will be marked that will be the output of a Xilinx logic block, then the polarity of each of these nodes

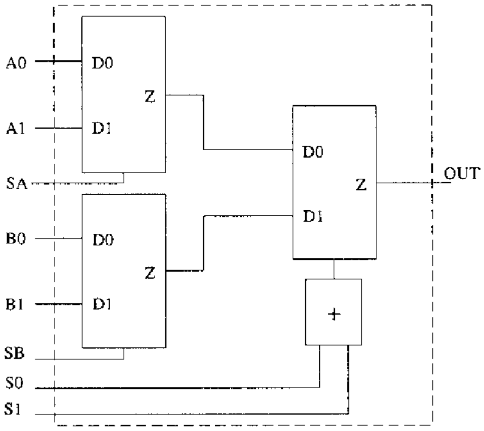


Figure 3.8: Basic logic block used in the ACT-1 series FPGA

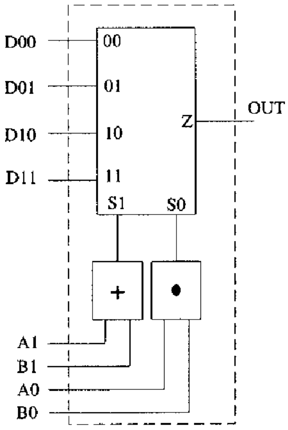


Figure 3.9: Basic logic block used in the ACT-2 and ACT-3 series FPGA

is determined and finally a merging algorithm is used that tries if it can merge two partially filled Xilinx blocks into one block.
All Karplus' heuristic algorithms are based on intuition and not really on the structure of the circuit. Besides the results obtained for Actel and Xilinx have been improved considerably (e.g. [LS95]), therefore this approach can be considered rather useless nowadays.

In [BBCS92], a ROBDD is constructed for a function, using lexicographical factorization to determine a good overall input ordering. The ROBDD network is then mapped on an ACT-1 network. The technology mapping algorithm is performed by a *pattern matching algorithm* (see Section 3.1.3): for all configurations of an ACTEL cell the corresponding ROBDD is constructed and the pattern matching algorithm tries to cover the ROBDD of the Boolean function with as few as possible cells. The matching algorithm uses only two different ROBDDs (ACT-1 cell configurations) for which it is claimed that they cover 90% of all nodes in an ROBDD. The algorithm works from the leaves to the roots and tries to match one of two patterns. If it succeeds, the corresponding nodes are marked. The nodes that are not matched at all are covered using a single ROBDD cell (which is expensive since only a single multiplexer from the ACT-1 cell is used). Experimental results show that the algorithm performs better compared to several other algorithms, for example, it produces better results than the Amap algorithm [Kar91a] mentioned above.

Kronecker Functional Decision Diagrams (KFDDs) have been successfully used for the synthesis of Atmel 6000 FPGAs [Atm95] [HP94][PCSS95]. Because Atmel's basic combinational cells [Atm95] cannot only implement a multiplexer but can also implement 3-input AND/EXOR cells, KDDs can be directly mapped on a network of Atmel building blocks. The benchmark results presented in [HP94] are compared to an older version of the proposed KDD algorithm. They show a significant improvement in the number of nodes in the KFDD compared to the predecessor of the proposed algorithm.

Recently, a very interesting approach has been introduced in [LPP96]. The method represents Boolean functions using OBDDs and is then able to find a disjoint and non-disjoint functional decomposition similar to the Ashenhurst-Curtis method (Section 3.3.1). The algorithm is able to find subfunctions in OBDDs and replaces such a subfunction by a single node. A nice feature is that the method can find subfunctions that are shared by any number of functions. The method is extended with a technology mapper so that Xilinx XC3000 and XC4000 devices can be used for the implementation. The technology mapper is able to use 9 different configurations of a XC4000 FPGA, thus making good use of the possibilities offered by the architecture of these FPGAs. Results show that this method is very effective, the method produces the best synthesis results currently available.

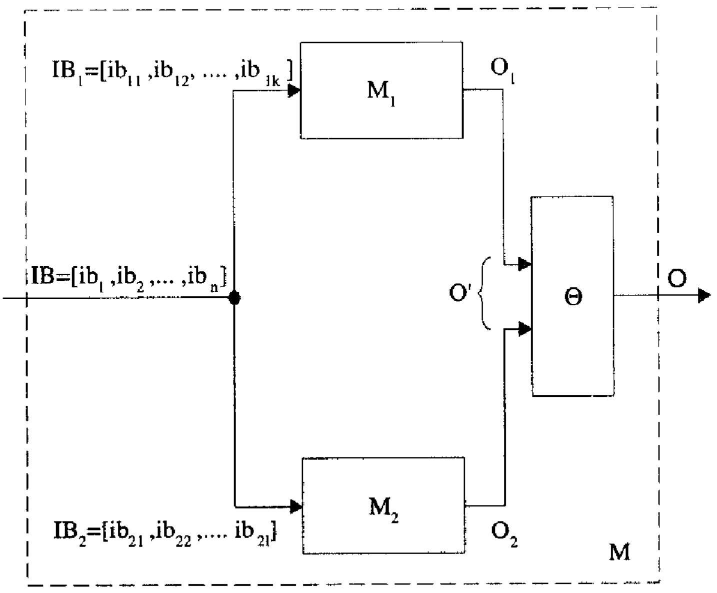

$IB_1=[ib_{11}, ib_{12}, \dots, ib_{1k}]$
M_1
O_1
$IB=[ib_1, ib_2, \dots, ib_n]$
O'
Θ
O
M_2
$IB_2=[ib_{21}, ib_{22}, \dots ib_{2l}]$
O_2
M

Figure 3.10: Input-bit parallel decomposition on two component machines

3.3 Special cases of general decomposition

In this section a number of special cases of the general decomposition will be discussed.

3.3.1 Input-bit parallel decomposition

In parallel decomposition, no information flows between the partial machines, and therefore the partitions π'_I and τ'_I of the general decomposition theorem are reduced to the identity partition. In input-bit decomposition, the input decoder Ψ is reduced to the distribution of the input bit lines and this results in the replacement of the input partitions π_I and τ_I by the bit-partitions π_{IB} and τ_{IB}. In this way, the following theorem was obtained from the general decomposition theorem.

Theorem 3.2 *Input-bit parallel decomposition with two component machines*
A combinational machine M has a non-trivial input-bit parallel decomposition with two component machines (see Figure 3.10) if and if only two partition doubles (π_{IB}, π_I^*) and (τ_{IB}, τ_I^*) exist that satisfy the conditions:

1.
$$\pi_I \leq \pi_I^* \text{ and } \tau_I \leq \tau_I^* \tag{3.6}$$

where

$$\pi_I = ind(\pi_{IB}) \text{ and } \tau_I = ind(\tau_{IB}) \tag{3.7}$$

2.

$$(\pi_I^* \cdot \tau_I^*, \pi_O(0)) \text{ is an I-O partition pair} \tag{3.8}$$

□

Of course this theorem can be easily extended to any number of component machines.

Two well-known special cases of input-bit parallel decomposition have been studied extensively in the past: functional decomposition and the use of input-encoders for PLA based synthesis. Both approaches will be discussed here in more detail.

The term *functional decomposition* is used to refer to techniques that split a Boolean function with many input variables into several Boolean functions each with fewer input variables. These new functions can be designed independently and if necessary recursively decomposed, thus reducing the complexity of the synthesis.
A number of classical functional decomposition algorithms have been developed in the 1960s [Ash59] [Cur61] [RK62] [Kar63]; recently these methods have gained interest, because they can be modified to be used for the logic synthesis targeted to field programmable gate arrays.
In [Ash59] a simple functional decomposition is presented, an n-input single-output Boolean function f can be decomposed as follows:

$$f(x_1, x_2, \ldots, x_b, x_{b+1}, \ldots, x_n) = g(h(x_1, x_2, \ldots, x_b), x_{b+1}, \ldots, x_n) \tag{3.9}$$

The set $B = \{x_1, x_2, \ldots, x_b\}$ is called the *bound set* and $F = \{x_{b+1}, \ldots, x_n\}$ is called the *free set*. Curtis [Cur61] has generalized the results obtained by Ashenhurst; in Curtis' decomposition algorithm s subfunctions can be used:

$$f(x_1, x_2, \ldots, x_b, x_{b+1}, \ldots, x_n) =$$

$$g(h_1(x_1, x_2, \ldots, x_b), h_2(x_1, x_2, \ldots, x_b), \ldots, h_s(x_1, x_2, \ldots, x_b), x_{b+1}, \ldots, x_n) \tag{3.10}$$

Normally, the above functional decomposition is referred to as the Ashenhurst-Curtis decomposition. In this method the functional decomposition is determined from the number of distinct columns in the Karnaugh map representation of the function. The Roth-Karp decomposition [RK62] is similar to that of Ashenhurst-Curtis, but uses a cube representation to find the decomposition and hence does not require the (potentially large) Karnaugh diagrams to be build.
The disadvantage of these approaches are three-fold:

1. The choice of the bound set is difficult. This is because no good relation between a bound set and the size of the decomposed has been found; that is, it is very hard to determine which bound set is a good set without the construction of the function.

2. The bound set is the same for all subfunctions; i.e. functional decomposition is able to find only special cases of the general decomposition model.

3. The structure of the function is not analyzed; for example the method does not attempt to find common subfunctions.

A special case of functional decomposition has been considered by W. Wan et al. [WP92]. Given a certain incompletely specified multiple output function $f(x_1, ..., x_n)$, the method presented in [WP92] searches for two disjoint subsets A and B of all inputs of f (i.e. $A, B \subset \{x_1, ..., x_n\}$, $A \cap B = \emptyset$), a multiple output function F and k single output functions $g_1, ..., g_k$ in order that the function $F(g_1(A), g_2(A), ..., g_k(A), B)$ realizes the "care" behaviour of f. The decomposition is targeted towards Xilinx FPGAs. The input set A is limited to 4 elements hence the functions g_k have no more than 4 inputs. Because g_k can implement any function of 4 variables with the same cost, the goal of the decomposition is to implement as much functionality as possible into the function g_k and make the function F as simple as possible. If the number of inputs of F is too large, the decomposition algorithm can be iteratively applied to F. Benchmark results presented in [WP92] show that the method performs well compared to previous methods that have been presented for synthesis on Xilinx logic blocks.

In the mis-pga algorithm [MNS+90] [MSBS91a] a number of different decomposition strategies have been implemented and Roth-Karp decomposition is one of them. The Roth-Karp decomposition however performs rather poor. This is due to the fact that the first feasible bound set that is found by the algorithm is selected; no attempt to find the best bound set is made; therefore this strategy has been abandoned in newer versions of mis-pga. More information on mis-pga can be found in Section 3.1.3.

Another well-known and extensively studied special case of the input-bit parallel decomposition is the *input-encoder problem*. In this case, the output decoder Θ is implemented as a PLA and the component machines M_i (referred to as input encoders) have multiple exclusive sets of input bits. The problem is often modeled using multiple-valued logic [Rud86] [RS87] [Sas84]. The general function of two-bit encoders is to replace the inputs a, $\overline{a}$, b and $\overline{b}$ of the PLA with the signals $a + b$, $a + \overline{b}$, $\overline{a} + b$ and $\overline{a} + \overline{b}$. It can be proved [Sas81] that the size of the new PLA (without the size of the two-bit encoders which generate these signals from a and b) cannot be larger than the size of the original PLA. However, the overall size (the size of the new PLA and the encoders) can be larger. Fortunately, in many practical cases, considerable gain in the total area can be obtained. The major problem with this method is the choice of pairs of inputs. In [Rud86] [Sas84] a heuristic algorithm is presented which tries to find suboptimal pairs of inputs. The drawback of this method is that it ignores interactions between pairs of inputs.

A similar, but more sophisticated and general way to solve the input-encoder problem was presented by Ciesielski et al. [CY92] [YC89]. They implement the input encoders using PLAs. The encoders can have any number of input signals and some of the input bits may be directly fed to the output decoder. In the first step, a set of inputs is partitioned into a number of disjoint subsets. Two heuristic approaches are presented for these: the first is based on integer programming whereas the second is based on a modified mincut algorithm. A classical multiple-valued minimization is then used to find the best implementation of the PLA Θ. The results that have been presented (in the

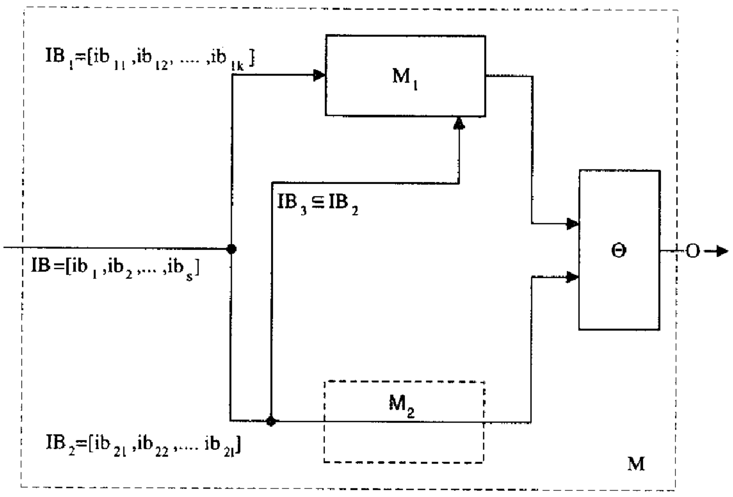

IB$_1$=[ib$_{11}$,ib$_{12}$, ... ,ib$_{1k}$]
M$_1$
IB$_3$ ⊆ IB$_2$
IB=[ib$_1$,ib$_2$,... ,ib$_s$]
Θ
O
M$_2$
IB$_2$=[ib$_{21}$,ib$_{22}$,.... ib$_{2l}$]
M

Figure 3.11: Input bit parallel decomposition as used by Łuba

form of benchmarks) are very promising, the only drawback of this approach is that the input sets are disjoint; the results show that the integer programming approach is better, but the graph-partitioning approach is faster.

Łuba et al. introduced yet another special case (see Figure 3.11) [ŁKJK91], where one of the component machines (M_2) is replaced by an identity function. The first step of this algorithm is to find the inputs IB_2 which have to be fed directly to output decoder Θ. The search algorithm tries to find the best set of inputs, such that the number of inputs of output decoder Θ does not exceed a user specified bound (for Xilinx CLBs this bound is set to 4). The algorithm then tries to find an implementation for machine M_1 using a minimum number of possible inputs (i.e. $IB_2 \cup IB_3$ contains a minimum number of elements). First a disjoint decomposition is used (i.e IB_3 has no elements). If this fails, inputs from IB_2 are added to IB_3 until machine M_1 can be constructed.
Unfortunately, no heuristics are described and no results on large benchmark sets are presented, therefore it is impossible to estimate the efficiency of this method for large circuits. However, the results that are presented are very promising. Using this special case, it is still possible to obtain the input bit parallel decomposition as shown in Figure 3.10, by recursively applying this special case to the output decoder Θ.

A number of methods for the more general input-bit parallel decomposition problem have been presented. In [HOI90], [HOI92] and [HOIW94] the set of input bits is partitioned in two disjoint subsets. A number of good heuristic procedures for partitioning exist. The goal of this method is to minimize the number of bits needed for communica-

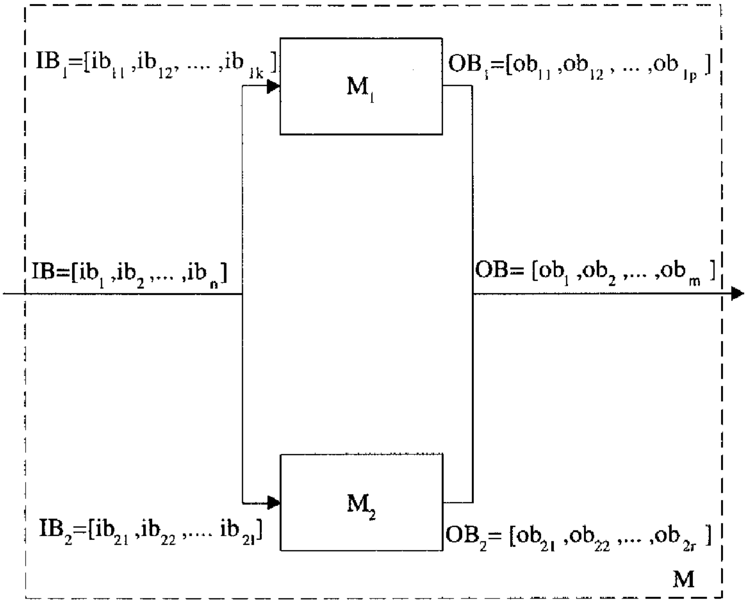

IB1=[ib11 ,ib12, ... ,ib1k]
M1
OB1=[ob11 ,ob12 , ... ,ob1p]
IB=[ib1 ,ib2 ,... ,ibn]
OB= [ob1 ,ob2 ,... ,obm]
M2
IB2=[ib21 ,ib22 ,.... ib2l]
OB2= [ob21 ,ob22 ,... ,ob2r]
M

Figure 3.12: Bit parallel decomposition with two component machines

tion between M_1 and M_2 and output decoder Θ. Although this method does not explicitly use the partition theory it can be easily formulated with this theory. The strength of this method is that it allows the estimation of communication complexity without having to construct the machines M_1 and M_2 and the output decoder Θ. This algorithm can have a linear complexity for circuits with low communication complexities.

3.3.2 Bit-parallel decomposition

In the bit-parallel decomposition, both the input decoder and the output decoder are reduced to the appropriate distribution of the input/output bit lines (see Figure 3.12). The theorem for this type of decomposition can be obtained from the input-bit parallel decomposition theorem by replacing the output partitions π_O and τ_O with bit partitions π_{OB} and τ_{OB}.

Theorem 3.3 *Bit parallel decomposition*
A combinational machine has a non-trivial bit parallel decomposition with two component machines (see Figure 3.12), if two partition doubles (π_{IB}, π_{OB}) and (τ_{IB}, τ_{OB}) exist that satisfy the conditions:

1.

$$\forall_{ob_k \in OB - dcb(\pi_{OB})} \quad (\pi_I, \pi_O(ob_k)) \text{ are I-O partition pairs} \tag{3.11}$$

where

$$\pi_I = ind(\pi_{IB}) \tag{3.12}$$

2.

$$\forall_{ob_k \in OB - dcb(\tau_{OB})} \quad (\tau_I, \tau_O(ob_k)) \text{ are I-O partition pairs} \tag{3.13}$$

where

$$\tau_I = ind(\tau_{IB}) \tag{3.14}$$

3.

$$\pi_{OB} \cdot \tau_{OB} = \pi_{OB}(0) \tag{3.15}$$

□

A number of solutions have been proposed for the bit-parallel decomposition. In [JV92] the problem is called the output decomposition problem. An output decomposition consists of partitioning the set of Boolean functions (outputs) into a number of disjoint subsets, each implemented by a separate component machine. The output decomposition in [JV92] aims at partitioning a multiple-output function into a minimal number of limited programmable logic building blocks (such as PLAs, PALs, etc.) and in minimizing the number of interconnections between the blocks. The problem is modeled as a multi-dimensional constrained optimization problem with constraints imposed on the number of inputs, outputs and terms. It is solved by a special multi-dimensional packing algorithm.

In the first step of the output decomposition algorithm, the information processing structure of the original combinational machine and its relation to the characteristics of building blocks are analyzed. From information about the correlations between the input, term and output variables as well as information about the constraints, the expected minimum number of building blocks and the expected number of input bits, output bits and terms per building block are computed. The expected values show how difficult it is to satisfy each of the constraints with a given number of blocks and indicate the amount of attention that must be paid to each of the constraints during the partitioning process. The active input bits and terms for each single-output function are also computed. Based on this information, affinities (from the viewpoint of a certain partitioning problem) between each two (single or multiple-output) functions can be computed.

With the above information, a limited number of near-optimal solutions are constructed in parallel by performing a multi-dimensional packing while using a beam-search algorithm. Since the decision making during the search is based on uncertain information, the search is guided by the heuristic elaborations of the rule of minimizing the uncertainty of choices. At each step, the decisions are taken which ensure the highest certainty of achieving the optimal solutions and, under this condition, the decisions that minimize the uncertainty of information for the future choices. Information that is used directly for decision making consists of relations between the characteristics of single-output

functions and constraints imposed by (partially constructed) building blocks and, correlations between the single-output functions and functions in (partially constructed) blocks.
Published experimental results show that the method is very effective and efficient; in almost all cases it was able to find the global optimum in reasonable time even for very complex functions (e.g. a functions with 131 inputs, 253 terms and 91 outputs (gpio) or a function with 45 inputs, 428 terms and 43 outputs (apex1)). The search algorithm has a number of parameters which enable a trade-off between the quality of solutions and the required computation time.
A similar method was published few months later in [HHC92]. It uses less information about the original multiple-output function than the method presented in [JV92] what can result in lower quality. An interesting concept not present in the method published in [JV92] is that of relaxing the term constraints and dynamically processing the terms.

Another approach to bit parallel decomposition is presented in [ŁKJ91]. In this paper the problem is referred to as parallel decomposition. The actual parallel decomposition is preceded by argument reduction. Argument reduction is a technique which minimizes the number of inputs of a Boolean function, as opposed to the classical minimization which aims at finding a minimum number of product terms. It is used to find function representations which use minimum number of input variables. This process is similar to term reduction which finds the minimum number of product terms.
The parallel decomposition algorithm uses the results of the argument reduction and constructs two-block parallel decompositions. Unfortunately in [ŁKJ91], only the idea of parallel decomposition is presented with no algorithms and heuristics. Only few results of experiments are shown however, these are very promising.

In two later papers [ŁSK93] [ŁS95], the decomposition method described in [ŁKJK91] (discussed in Section 3.3.1) is combined with the input-bit parallel decomposition method mentioned in the previous paragraph. This combination allows for the construction of complex networks of blocks. A heuristic is presented which shows how these two different decomposition approaches should be combined and which parameters should be used to tune these algorithms. Results presented for a large number of circuits show that this algorithm works well for ACTEL cells and very well for Xilinx cells. This algorithm has the potential to become one of the most useful and powerful synthesis algorithms for FPGAs.

3.4 Multiple valued logic and symbolic encoding

The concepts used in the symbolic design paradigm are in fact very similar to those used in the partition theory based general decomposition paradigm. This technique can be used if the inputs and/or the outputs of the combinational logic are represented by symbolic values, i.e. the binary representation of the inputs or outputs can be freely chosen. This approach is similar to the state encoding problem of sequential machines; in fact one of the first publications in this area [Mic86] has been derived from a state assignment algorithm introduced by the same authors [MBS84] [MBS85].

X Y	F
1 a	0
2 a	1
3 a	1
4 a	0
1 b	1
2 b	0
3 b	0
4 b	1
1 c	1
2 c	0
3 c	0
4 c	1

Table 3.1: Multiple-valued input binary valued output function F

Symbol	assignment 1 (x_1, x_2)	assignment 2 (x_1, x_2)
1	$(0,0)$	$(0,0)$
2	$(1,0)$	$(0,1)$
3	$(1,1)$	$(1,0)$
4	$(0,1)$	$(1,1)$

Table 3.2: Two different assignments for multiple valued variable X

In symbolic design, logic minimization is performed before the code assignment of the inputs and output symbols. That is, the Boolean function is represented as some kind of multiple-valued Boolean function and is then minimized using a multiple-valued minimizer. In a second phase, the input and output symbols are replaced with binary vectors that are compatible with the symbolic cover of the minimized multiple-value function. The symbolic design will be illustrated using the following example.

Example 3.4 *Symbolic design*
In this example, the multiple-valued input single-valued output function F (Table 3.1 is being considered. The function has one 4-valued input variable X and a 3-valued input variable Y. An expression for the on-cover of function F is

$$F = X^{\{2\}} \cdot Y^{\{a\}} + X^{\{3\}} \cdot Y^{\{a\}} + X^{\{1\}} \cdot Y^{\{b\}} +$$

$$X^{\{4\}} \cdot Y^{\{b\}} + X^{\{1\}} \cdot Y^{\{c\}} + X^{\{4\}} \cdot Y^{\{c\}}$$

Symbol	assignment 1 (y_1, y_2)	assignment 2 (y_1, y_2)
a	$(0,0)$	$(0,1)$
b	$(1,0)$	$(0,0)$
c	$(1,1)$	$(1,1)$

Table 3.3: Two different assignments for multiple valued variable Y

where the notation Z^T stands for

$Z^T = 1$ if and only if $Z \in T$

$Z^T = 0$ otherwise

Using a multiple valued Boolean minimizer the function F can be minimized to:

$$F' = X^{\{2,3\}} \cdot Y^{\{a\}} + X^{\{1,4\}} \cdot Y^{\{b\}} + X^{\{1,4\}} \cdot Y^{\{c\}}$$

and by factoring out the $X^{\{1,4\}}$

$$F'' = X^{\{2,3\}} \cdot Y^{\{a\}} + X^{\{1,4\}} \cdot Y^{\{b,c\}}$$

is obtained. The function F'' is the minimized multiple valued Boolean function. Now, the encoding of the symbols 1, 2, 3, 4 and a, b, c need to be found. In the Tables 3.2 and 3.3 two bit assignments for X and for Y are presented. Taken the first assignment for both X and Y and using the don't care in Y, the function becomes

$$F_1'' = x_1 \cdot \overline{y_1} + \overline{x_1} \cdot y_1$$

while using the second assignment for both X and Y the function becomes

$$F_2'' = (x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2) \cdot \overline{y_1} \cdot y_2 + (\overline{x_1} \cdot \overline{x_2} + x_1 \cdot x_2) \cdot (\overline{y_1} \cdot \overline{y_2} + y_1 \cdot y_2)$$

Thus, selecting the right input assignment is very important. □

For the two-level minimization of multiple-valued Boolean expressions, any ordinary two-level minimizer can be used, by using an input for each symbolic value [BHMS84, chapter 5]. However, also a number of dedicated multiple-valued minimizers have been introduced [RS87] [CYP89]. In [RS87] two minimizers are presented: Espresso-exact and Espresso-MV; Espresso-exact finds the solution with the fewest possible number of product terms (although it does not minimize the total number of literals). Espresso-MV is a heuristic minimizer dedicated to multiple valued functions. Results show that Espresso-MV finds the same results for multiple-valued functions as ordinary Boolean functions, but requires only half the processing time. In [CYP89] a graph coloring approach is used to determine the minimal cover; the results (both in quality and time) are comparable to

Espresso-MV.

From the example above it can be concluded that a bad choice of the encoding will produce too much logic; it can even produce erroneous results (if the chosen encodings intersect). In [MBS85] this problem has been formulated as an constrained encoding problem: the encoding of the inputs and the outputs so that they are compatible with the symbolic cover. Satisfying the encoding constraints is trivial (by encoding each symbol on a separate bit, all constraints are per definition satisfied); however finding an encoding that uses the minimal number of bits is intractable [SVBS94].
In [YC90] [YC91] the input encoding is considered. In this article the problem is formulated using dichotomies. A dichotomy is a two-block partition and it is used to determine which bit vectors are compatible (i.e. which bit vectors can be encoded using the same symbol). The proposed algorithm consists of a covering algorithm based on these dichotomies; it tries to find prime and essential prime dichotomies and uses a graph coloring algorithm to find a near-optimal cover of dichotomies. An interactive improvement algorithm with hill climbing and splitting of dichotomies is also used to improve the results. Results (measured in the number of bits used to encode the symbols) are better than those presented by [Mic86]. Unfortunately, comparisons based on more physical parameters (like chip area, number of look-up tables, delay times) have not been made. In [CSD91] the above method has been extended to simultaneous input and output encoding for sequential machines.
Devadas e.a [DN91] have addressed the problem of output decomposition. In their approach they use algorithms that find the optimal encoding. These algorithms are much faster than exhaustive search. They have applied these algorithms for the logic synthesis on PLAs; in this approach a subset of PLA inputs is selected and this subset is re-encoded using the approach mentioned above. This leads to a serial decomposition consisting of two PLAs: an input processor that processes part of the inputs and a main PLA that uses the output of the input processor and the other input bits to calculate the output function.
Because of the wide range of applications that use constrained encoding, a number of papers have been devoted to the theory and algorithms for this problem; the interested reader is referred to [SB93] [SVBS94].

Summarizing, the symbolic design approach is a very interesting contribution to logic synthesis, although it has some disadvantages. Disadvantages of this approach compared to the general decomposition paradigm are the following:

1. The symbolic design approach assumes that the optimal symbolic minimization followed by an optimal symbol encoding produces near-optimal overall results. This is not always true because these two steps are executed independently.

2. The search for the best input or output encoding is not based on the estimation of their influence on physical parameters (like area, speed, wires), but purely on desired compatibility of the patterns.

3. The general decomposition theory uses a more general and a more elaborate mathematical framework (partition product, M and m operators) than symbolic design

does. Therefore, algorithms based on decomposition theory have a better foundation.

Chapter 4

LUTSYN - Framework and Concepts

In the previous chapters the topic of this thesis has been introduced and placed in the context, basic definitions and concepts related to the topic have been presented and the state of the art in logic synthesis has been analyzed. This chapter and the next will focus on a new logic synthesis method for LUT based FPGAs. This new method has been named LUTSYN (Look-Up Table SYNthesis).
This chapter is subdivided into three parts. In the first part (Section 4.1) the logic synthesis problem being the topic of this thesis will be analyzed and defined more precisely. Also its computational complexity will be discussed. Using the results of this part and of the previous chapters, in the second part (Section 4.2), the mathematical framework that forms the basis of the problem solution will be introduced. In the last part of this chapter, an overview of the proposed synthesis method is provided (Section 4.3) and a number of concepts that are required for the near-optimal synthesis with this method are investigated.
The next chapters show how these concepts have been translated into algorithms and programs, discuss the results of some decomposition experiments performed with the developed prototype tool, and present some conclusions on the tool's and the method's performance.

4.1 Problem definition

4.1.1 Choice of an FPGA architecture

The LUTSYN algorithm is dedicated to LUT based FPGAs. This choice is not accidental. FPGAs in general have a number of important advantages compared to full custom, standard cell and mask programmable technologies. First of all, FPGAs are standard components. They are produced in large quantities, and therefore, are much cheaper (it is not necessary to create new masks for each FPGA implementation, as it is with the other implementation techniques). This fact enables the use of integrated circuits (in the form of FPGAs) in small series of products.
Another very important aspect of FPGAs is that they can be programmed by the user without involvement of the manufacturer. For this reason, production times of the design are significantly lower compared to the other technologies (hours instead of weeks). Also, it is no longer necessary to give the design to the integrated circuit manufacturer, and the valuable design can be kept in-house, safely secured from the eager eye of the

competitors. The fact that FPGAs can be programmed by users offers also a number of other advantages: fast prototyping, easy modifications of the functionality and reuse by simple reprogramming of the FPGA. In general, FPGAs offer complex functionality at a low cost. However, ASICs are generally faster than FPGAs and have a higher functional density.

Unfortunately, the traditional logic synthesis and technology mapping algorithms for the logic synthesis of Boolean functions cannot be applied effectively to FPGA architectures. This is due to the fact that FPGAs have a different architecture compared to the full custom, standard cell and mask programmable technologies and are completely prefabricated. An FPGA consists of a fixed number of blocks. Each of these blocks has a fixed number of inputs and outputs and it can implement a certain set of Boolean functions. The blocks can be connected by a programmable wiring mechanism, but once again this mechanism is restrictive and can implement only a small subset of all possible wiring schemes. In other words, FPGAs provide an architecture with very hard physical constraints, whereas the traditional logic synthesis methods do not consider any physical constraints. Moreover, technology mapping methods for other implementation techniques assume a much less constrained architecture (limited to only the cell library that is used and the overall chip area that is available).
For LUT based FPGA architectures the difference with the traditional ASIC architectures is even larger. A LUT block can implement any Boolean function, but with limited number of input variables (typically 4 variables). Thus the constraints are not imposed on the function type as it was by traditional synthesis, but on the physical dimensions. Also, LUT blocks are very small. This means, that decomposition is essential for the synthesis of almost every Boolean function. Furthermore, traditional synthesis is always based on a limited set of functions. In many cases the used operators are AND, OR and NOT [BRSW87], in other cases the EXOR and AND operators are used [BS93], but never the full scope of all possible Boolean functions with the number of inputs not larger than 4 is used in this synthesis. Even worse, traditional synthesis methods use this limited set of operators for CMOS circuit synthesis, even if the technology cell library consists of many more complex cells. This results in very complex technology mapping algorithms and bad synthesis results. For LUT based FPGAs it is extremely important to consider all available Boolean functions during the logic synthesis; because any function with a number of inputs and outputs not higher than the LUT input/output count limit can be implemented in a LUT, regardless the sort and complexity of the function's expression. This means, that the traditional logic synthesis, and in particular logic synthesis techniques based on literal counts cannot produce good results for LUT based FPGAs.
Based on these observations, it can be concluded that it is very useful to develop new logic synthesis methods dedicated to FPGA implementations.

A lot of different FPGAs with many different architectures are offered nowadays to the user. However, no logic synthesis method exists that is able to produce near-optimal implementations for all these different FPGA architectures. This is caused by the fact that many of the FPGA architectures differ vastly in the functionality of the logic blocks and in the flexibility of the interconnection mechanism. So, instead of trying to perform the logic synthesis for all existing FPGA architectures with a single synthesis method, in this work, a synthesis method will be developed that is able to find near-optimal implementations for a subset of all FPGA architectures (LUT based FPGAs). However, care will be taken that this synthesis method is flexible, so it will hopefully be easy to extend the method to some other FPGA architectures, and to other synthesis targets that have a strong analogy with LUT based FPGAs.

An important aspect of an FPGA architecture is the physical topology (or layout) of that FPGA. The physical topology of an FPGA consists of a number of blocks and wires with fixed positions and certain properties. The topology is different for each class of FPGAs and may even differ between different types of FPGAs of a single class. Many synthesis methods do not work on the actual FPGA physical topology but on a simplified model of the topology. Such a model is called a *topology model*. There are two reasons for making such a model. First of all, the physical topology can be too detailed and too complex for using it efficiently in synthesis algorithms. The second reason is that by using a model as the abstraction of a FPGA topology the synthesis can be performed for a larger class of FPGA topologies with the same method or algorithm, provided that the topology model is a good abstraction for each of these FPGA topologies.
The design step that translates the synthesis results obtained with the topology model to the actual physical FPGA topology is called *placement and routing*. The model is naturally inaccurate (if the topology model would provide the same information as the physical topology then the topology can be used directly and there would be no need for the model). Therefore, placement and routing can be a difficult problem to solve. It may even turn out that placement and/or routing is impossible. Therefore, it is important to find the break-even point so that the topology model is detailed and accurate enough for placement and routing but still can be used efficiently in higher level synthesis algorithms.
Often, in synthesis methods another level of abstraction is used: based on the topology model a, *logic model* is created that has nice mathematical properties so that the logic minimization can be mathematically formulated and solved. Methods that use only the logic model for synthesis, are called technology independent synthesis methods (e.g. division based methods). The transformation of designs synthesized with a logic model into the designs that account for a certain topological model is called *technology mapping*.

For this Ph.D. research project it has been decided to develop a synthesis method that uses a base lookup table FPGA architecture consisting of a set of identical small LUTs with programmable interconnections. The topology model used with this base architecture is a network of LUTs. To account for the limits in wiring resource, the method will minimize the number of long wires of the LUT network. There will be no separate logic model. That means that the logic and topological model are the same and technology mapping is not necessary.
There are at least two widely used FPGA architectures that correspond to this base architecture: Xilinx FPGAs [Xil93] and ALTERA's FLEX series [Alt95]. Both FPGA architectures consist of a programmable network of LUTs. The choice of this base architecture also seems to make sense in respect to other fine granularity FPGA architectures: the basic building blocks of all other architectures can be seen as some special cases of a LUT. After all, a LUT is able to implement any function of NI inputs and NO outputs,

so all other basic building blocks must be a special case of such a LUT. For example, the Actel architecture consists of network of programmable multiplexers. A multiplexer can be seen as a special limited "LUT", that is able to implement only a subset of all functions implementable by an actual LUT of the same input and output size. Similarly, Atmel FPGAs [Atm95] consist of a fine granularity block structure, where each block is able to implement a (large) subset of LUT functions.
Furthermore, the base logic synthesis method will also be able to produce standard cell implementations. If the standard cell library is complete (it implements all non-trivial Boolean functions of a certain number of input and output variables), the synthesis method can be used unmodified. Otherwise the synthesis method must be changed so that the missing functions are suppressed. The approach of extending the base architecture to other architectures can only be used if the logic blocks of the other architectures are largely similar to LUTs. If the difference between them becomes too large, then the model becomes inaccurate and the synthesis algorithm can no longer be guaranteed to produce near-optimal results. Small discrepancies between the logic blocks of an architecture and the LUT blocks used in the base architecture can be resolved by taking care that the synthesis algorithm will never construct or will remove blocks that implement one of the missing functions. Another approach would be that the missing functions will be build using two or more building blocks that are available in the architecture. For example, in [ŁSK93] it has been described how LUTs can be build by combining a number of Actel multiplexers. This way missing functions can be created as compound building blocks. However, if this approach is taken the synthesis method must be extended to utilize the fact that these compound cells are more expensive to use then the single cells (because they occupy more than one physical building block).

The synthesis method for the base architecture is aimed at small LUTs only. The typically maximal size of such a LUT is 4 inputs and 2 outputs or 5 inputs and 1 output. These values correspond to the size of Xilinx logic blocks; they have not been chosen arbitrary. In [FS94] it has been proved that the technology mapping of a Boolean network (where the vertices of the network correspond to any K-input 1-output Boolean function) onto a network of K-input LUTs is NP-complete for any $K \geq 5$. Furthermore, research performed by FPGA manufacturers has determined that for larger LUTs the unused logic in a LUT becomes too large [BFRV92, chapter 4].

4.1.2 Optimization criteria

Since the problem of optimal logic synthesis for LUT networks is computationally complex (see Section 4.1.4 for details), strictly optimal implementations cannot always be obtained using a realistic amount of computing resources. Therefore heuristic algorithms will be used that strive to find a solution as near as possible to the optimal in a reasonable time and using a reasonable amount of resources. The proposed heuristic algorithm will construct one or more circuit realizations that solve the synthesis problem in parallel. At one or more points in the construction process decisions have to be made which determine whether and how to continue the construction of the partial solutions. The heuristic algorithms will evaluate each partial solution by computing values for one of more evaluation parameters that predict the amount of resources that the best final solution reachable from a given partial solution will use. This computation is naturally inaccurate. The optimal solution is the solution that uses the fewest resources. Unfortunately, for large problems, it is virtually impossible to know whether the heuristic algorithm has found the optimal solution. In fact, it will be proved in Section 4.1.4 that it is impossible to make a precise quantitative statement about the distance (expressed as the difference in the resource usage) between the constructed solution and the optimal solution. However, it is possible to compare the partial solutions and make some comparative predictions on how much it will cost to develop these partial solution to final solutions.

The optimal solution of a synthesis problem is defined in terms of *optimization goals*. The optimization goals are the objectives of the synthesis and are part of the total design specification. Many different optimization goals can be thought of.
The most classical optimization goals of logic synthesis are *area optimization* and *delay optimization*. In area optimization, the goal is minimize the chip area occupied by the implementation. The area can be occupied by active elements and by wires. For LUT based FPGAs, area optimization is not interesting because the FPGAs are completely prefabricated and thus have a fixed area. It is however very interesting to minimize the number of cells that are used for the implementation and to minimize wiring (especially the number of long wires), since this allows to place more complex functions into an FPGA and helps the placement and routing phase to do a better job.
The goal of delay optimization is to minimize the critical path delay or maximize the operational frequency of the circuit. Delay optimization is very important for LUT based implementations, because the configurable switches used in LUT based FPGAs can contribute up to 60% of the overall design delay [XK96]. Delay optimization for FPGAs requires a topological model, because not all wires have the same characteristics and the delay of the wiring switches have a significant influence on the overall delay of the circuit. In [XK96] a method for the timing estimation for Xilinx FPGAs is proposed.
Design for low power has become a very important issue recently for a number of reasons. First of all, the demand of consumers for hi-tech portable products forces the designer to use as little power as possible. Secondly, to protect the environment, it is necessary to use as few as possible resources and finally, to further increase the density of integrated circuits, it is necessary to lower the power consumption to overcome heating problems and to reduce capacitances. Therefore *power consumption optimization* has become an important optimization goal. Minimizing the power consumption used by an FPGA, requires a model that models the power flows in an FPGA and a power estimation algorithm that calculates the amount of power consumed by the circuit. A very interesting approach to the synthesis for low power FPGAs can be found in [SSA94].
Another example of an optimization goal is *synthesis for testability*. The optimization goal here is to find an implementation that can be easily tested or includes self diagnostics. Since the internals of an FPGA cannot be reached, synthesis for testability just adds extra test functionality to the circuit to perform the testing (for example unused registers in the FPGA can be used to create a scan path). An overview of techniques for synthesis for testability techniques can be found in [KN96].

Unfortunately, optimization goals cannot always be directly expressed or their achievement directly measured. Often, they need to be translated into some estimators, i.e. (mathematical) statements related to the logical or topological model that is used for the synthesis. These estimators are called the *optimization criteria*: criteria that should be met as precisely as possible by the synthesis algorithms, because they express in a way the required circuit properties.
A number of optimization criteria for networks of LUTs are:

- **The number of LUTs**
  The number of LUTs is the primary optimization criterion for the amount of active area used in the implementation of the circuit, and the only useful estimation function for the active area of small granularity FPGAs. Values for this criterion can be computed easily from the LUT network by counting the number of LUTs.

- **The amount and the length of the interconnection nets**
  The amount of nets necessary to interconnect the LUTs and the length of each of these nets are two optimization criteria that measure the complexity of a circuit (together with the number of LUTs). The amount of nets provides the measure for the input and output capacity of the LUTs that are used. It is also an indication for the average number of times a LUT output is used as an input for another LUT. A small number of routing nets results in an easier placement and routing problem for the FPGA and has a positive effect on the number of look up tables used. The number of routing nets can be obtained easily from the LUT network by counting the interconnections.
  The length of the nets is a good measure for the (routing) complexity of a circuit because circuits with many long wires are generally more difficult to place and route. Short wires have a positive effect on the placement and routing of the circuit. Finding the length of the nets in the network of LUTs model is not straightforward, since this network does not take into account the position of the LUTs. One can analyze, however, which mechanisms create long nets, and which ones prevent the creation of many long nets.

- **Number of levels in the circuit**
  Since the delay characteristics for the LUTs may not be available at the logic synthesis level (because it is not a part of the used logic or topology model) the number of levels can be used as a measure for the speed of the circuit. This is especially true if the LUTs have the same delay time for each function they can implement. Minimizing the number of levels in a LUT network does not only minimize the critical path delay, but also makes the circuit faster in general. Unfortunately it has a negative impact on the number of LUTs in the network, since minimizing the network depth decreases the chance of reusing an output of a LUT.

- **Critical path delay**
  This optimization criterion is used to make the circuit as fast as possible. It cannot be computed accurately without making a topological model of the FPGA synthesis, because of the influence of delays in the wires. Furthermore, it requires accurate characteristics of the delay of the LUTs and the routing switches.

Of course it is also possible to create a trade-off between two or more of these optimization criteria. Such a trade-off can be static (specified by the user as a configuration parameter of the algorithm) or dynamically adapted by the algorithm based on the partial solution.
In the algorithm presented in this thesis the trade-off between the following optimization criteria will be used:

1. Minimize the number of LUTs in the circuit.
2. Minimize the number of levels in the circuit.

The trade-off algorithm will allow the construction of networks that have as few as possible LUTs. However it allows LUTs to be sacrificed to improve the speed of the circuit, if this is desirable or necessary.

4.1.3 Problem formulation

Based on the discussion in the previous section, it is now possible to formulate the exact problem statement for which a solution has been proposed in the scope of the reported research.

The goal of the research was to construct an effective and efficient logic synthesis method for the multiple-level implementation of multiple-output combinational Boolean functions using LUT based FPGAs. The method will construct the LUT network by performing a general decomposition (without loops), such that all possible loop-free network structures can be constructed. The following assumptions and decisions have been made:

1. The FPGA architecture will be modeled as a network of LUTs, so that technology mapping will not be necessary. Each LUT is able to implement any Boolean function that has no more than NI inputs and no more than NO outputs, where NI and NO are small positive integers (typically $NI \leq 5$ and $NO \leq 2$). All LUTs are assumed to have the same size.
2. The Boolean function to synthesize is specified as an incompletely specified multiple output Boolean function and represented as a function table. The method will be focused on functions for controller type circuits. The function tables of controllers can roughly be characterized as having many primary inputs and outputs, relatively few product terms compared to the number of all possible min terms, and many don't cares. The original function will not be preprocessed using two-level minimization techniques, so the total design freedom, including the complete don't care set is available to the actual multiple-level logic synthesis process.
3. The primary optimization criterion consists of minimization of the number of LUTs. However, a trade-off will be possible with the criterion "minimize the number of levels in the tree". Furthermore, the algorithm will try to minimize the number of long wires. So, the active area minimization will not be preferred at all costs.

The method will try to make the life of the "placement and router" process easy by limiting the number of long wires. The number of long wires is minimized using the following rationale: long wires are typical to larger circuits and they are primarily the result of the creation of common subfunctions or the use of the primary input signals or signals created in the circuit levels close to the primary inputs as inputs to the logic blocks located that are located close to the primary outputs. Namely, when creating common subfunctions all the fan-outs (except for one) cannot be processed further locally and their related signals have to be transported to another parts of the circuit. The reduction of long wires is therefore accomplished by limiting the creation of common subfunctions to those cases where the saving in the usage of LUT is considerable. Thus creation of small subfunctions being common subfunctions of many functions will be prohibited. This can be implemented easily by creating a threshold on the subfunction size: only subfunctions larger than the threshold will be created.

There are some problems that cannot be accounted for by the proposed synthesis method:

1. The model does not incorporate the topology of the FPGA; that means that it does not find a layout for the FPGA. So, floor planning, placement and routing of the LUT network into the FPGA need to be done independently as a postprocessing step of the method presented here. Adding topology to the proposed method is not straightforward.

2. The method assumes that each LUT has the same properties, regardless of the function it implements. Therefore, counting the number of LUTs is considered an accurate measure for these properties. This assumption is true for the area of a LUT (the area does not depend on the function the LUTs implements), but this is not necessarily true for electrical characteristics of the LUT, like the delay of the look-up cell or the power dissipated by that cell. If these characteristics are available, then extending the model with them and making it more accurate will be quite easy. Adapting the decision algorithms so that smart use of the extra information is made, is however a bit more difficult.

3. The synthesis method assumes LUTs with a fixed input and output size. Some FPGA architectures however have a number of different blocks available or have different modi operandi for these blocks (Xilinx XC3000 logic blocks can be programmed as 4-input, 2-output LUTs, or as 5-input, 1-output LUTs and these types can be mixed freely, Xilinx XC4000 series allow even more complex combinations). Extension of the method to cover such cases is however straightforward and consists just in considering in the place of a single set of maximal physical block dimensions a few such sets of maximal physical block dimensions in the same manner as the single set.

In the next sections a concept for solving the problem formulated above will be presented.

4.1.4 Complexity considerations

In this section the complexity of the logic synthesis problem stated in the previous sections will be discussed. There are two fundamental problems that need to be addressed:

1. Find an optimal realization r for a certain function f.
2. Find the distance between a realization r' of a function f and a best realization r for this function. The distance is expressed in terms of some quantity that is calculated by the quality function used to evaluate the realizations.

The first question simply solves the logic synthesis problem that has been formulated. Its complexity is important, because it shows if it is in practical sense possible to find the best solution for large instances of the problem with a reasonable time or if a heuristic solution for large problem instances is desirable. The second problem is also of some importance because it shows how far a realization lies from an optimal solution, hence it gives a measure for the quality of the solutions generated by a heuristic algorithm. It will be shown that both these problems are NP-hard [GJ79]. A problem is called NP-hard if a NP-complete problem can be transformed to it in polynomial time. This means that solving a NP-hard problem is at least as difficult as solving a NP-complete problem. Practically, this means that heuristic algorithms are acceptable and even essential for the synthesis of large functions and that determining the exact quality of a solution is as difficult as finding the optimal solution itself.

To prove the NP-hardness of these two problems the notion of realization space is necessary. The *realization space* $\mathcal{R}$ consists of all possible implementations of Boolean functions expressed in terms of a suitable description language. The realization space is extremely large, but it is finite because of economical, practical or physical limitations imposed by the implementation technology or the considered application.
On the realization space a *quality function* $Q : \mathcal{R} \rightarrow \mathbb{N} \cup \{\infty\}$ is defined, where $\mathbb{N}$ represents the set of natural numbers (including 0). Quality function Q_f creates an ordering on the realizations of function f. A realization $r \in \mathcal{R}$ is an optimal realization of f if there exists no realization $r' \in \mathcal{R}$ such that

$$Q_f(r') < Q_f(r)$$

Note that in general more than one optimal realization can exist. If a realization r does not implement the behaviour of function f then it is given an infinite quality, i.e. then $Q_f(r) = \infty$. For all other realizations it calculates a value representing the cost of the realization expressed in terms that the designer finds the most important (e.g. area, speed, trade-off between area and speed).
Proving the NP-hardness of a problem is done by showing that a well-known NP-complete problem is transformable to this problem in polynomial time [GJ79]. Here, it will be shown that the satisfiability problem (which is a well-known NP-complete problem [GJ79]) can be transformed in polynomial time to each of the fundamental problems described above.

Theorem 4.1 *Satisfiability problem*
Given a Boolean function $f : B^n \to B$, the problem of answering the question "does an input vector $\underline{x} = (x_1, x_2, ..., x_n) \in B^n$ exist such that $f(\underline{x}) = 1$?" is NP-complete; in other words determining the existence of a truth assignment for f is NP-complete. The proof can be found in [GJ79]. □

The proofs here will be restricted to single output functions. This is no severe restriction, because multiple output functions are more complex than single outputs. Therefore, if NP-hardness for single output functions is proved, then the problem for multiple output functions must also be NP-hard. Before the NP-hardness can be proved, it is necessary to make the following assumption: for each n-input null function $f_0^n : B^n \to B$

$$f_0^n : \underset{\underline{x} \in B^n}{\forall} f_0(\underline{x}) = 0$$

there exists precisely one optimal realization. This is by no way an unreasonable assumption: the n-input null function f_0^n is zero for all possible input values and hence can be realized by connecting the output to the low voltage pin of the power source. Any other implementation of that null function will cost more than this single wire and is thus less optimal. If this unique optimal realization of the null function is called r_{false}^n then this assumption can be formulated as follows:

1. $\forall n \in \mathbb{N} : r_{false}^n$ is *the best* realization of f_0^n.

2. For each optimal realization r of f : $Q_f(r) = Q_f(r_{false}^n)$ implies that $r = r_{false}^n$.

Theorem 4.2 *Finding an optimal realization r for f is NP-hard*
Suppose that an algorithm X exists that is able to find r for each Boolean function $f : B^n \to B$. Then X can be used to calculate r and r can be compared to r_{false}^n. If they are equal then f is the null function and hence f is not satisfiable. If r is not equal to r_{false}^n then f cannot be the null function (r_{false}^n is the unique best realization of the null function) thus at least one input vector $\underline{x}$ exists such that the $f(\underline{x}) = 1$. This means that the algorithm X could be used to solve the satisfiability problem for a function f by simply using algorithm X to find an optimal realization r for it and comparing it to r_{false}^n. The transformation of the satisfiability problem into this problem only involves the comparison of r and r_{false}^n which can be performed in time linear to the size of description of r_{false}^n. This completes the proof of the NP-hardness of finding the optimal realization of a Boolean function. □

Theorem 4.3 *For a given realization r', finding the distance $Q_f(r') - Q_f(r)$, where r is some optimal realization of f, is NP-hard*
Suppose an algorithm that can calculate $Q_f(r') - Q_f(r)$ exists. Then it can be used to calculate $p = Q_f(r_{false}^n) - Q_f(r)$. If $p = 0$ then $r = r_{false}^n$ and f is the null function f_0^n (this follows from the assumption made above). On the other hand, if $p \neq 0$ then an input vector $\underline{x}$ must exist such that $r(\underline{x}) = 1$. But since r is one of the optimal realizations of f, there must also be at least one input vector $\underline{x}$ such that $f(\underline{x}) = 1$. Therefore, the distance algorithm can be used to solve the satisfiability problem. Because the transformation

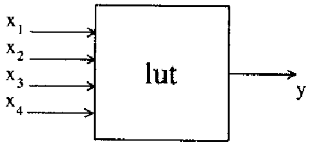


Figure 4.1: A 4-input, 1-output LUT

consists in comparing p with 0 (which costs constant time) it can be concluded that finding the distance is NP hard. □

Theorem 4.4 *Checking whether r is an optimal realization of some f is NP-hard*
The proof is analogous to the proof for finding an optimal realization for function f. □

Theorem 4.5 *For a given realization r' and natural number $n \in \mathbb{N}$, checking whether $Q_f(r') - Q_f(r) < n$, where r is some optimal realization of f is NP-hard*
It is an NP-hard problem to determine if the distance between the realization of a certain Boolean function and the best realization for that function is within a certain bound value n. The proof is analogous to the proof of finding the distance of realization r' and some optimal realization. □

The theorems in this section show that the use of heuristic algorithms is indispensable for the near-optimal logic synthesis of large Boolean functions, implemented as LUT networks. Also, the theorems state that it is impossible to decide in a reasonable time whether the synthesis algorithm has found the best possible realization.

4.2 Mathematical framework

4.2.1 Network representation using set systems

In the previous sections it has been discussed that a network of LUTs would be a good logic and topological model for the synthesis problem. The goal of this section is to develop a suitable mathematical framework for this model.
The mathematical framework consists of 2 parts. First, there is a definition of the network structure. The second part of the framework consists of an algebra that can be used to define the logic function of the blocks and defines operators on the graphs and the blocks. Defining the network structure is not really a problem: this will be a simple graph representation consisting of vertices (the LUTs) and edges (the interconnections). However, choosing a useful algebra is not-trivial at all. This will be a subject of this and the next sections.

When creating a mathematical model there are two important aspects to consider: the representation of information and the way this information can be processed. In the LUT case, the representation is the functional description of a single LUT and of networks of LUTs. The synthesis process of a LUT network uses that information to form a network that implements the desired functionality, so the operators defined in the mathematical model should facilitate the synthesis process. In general a mathematical model should have the following properties:

1. It must be an algebra with nice mathematical properties and the algebra should have useful and powerful operators defined in it.
2. The necessary information should be compactly represented. If this is not the case, then the model will not be satisfactory for large circuits.
3. It should model exactly the information necessary for the synthesis. Of course all the necessary information should be present in the model, but it should also not model too much information since this makes the model unnecessary complex and the superfluous information can distract or even mislead the synthesis process.
4. There should be a strong relation between the theory and practice; all the elements of the mathematical model must have a straightforward relation and interpretation with respect to the circuit that is being build.

A LUT is in fact a black box that can implement any Boolean function with no more than *NI* inputs and no more than *NO* outputs. Take for example the LUT in Figure 4.1, this LUT is able to implement any Boolean function of 4 or less inputs, e.g.:

$$y = x_1 + x_2$$

or

$$y = \overline{x_1} \cdot x_2 + x_2 \cdot x_3 \cdot x_4 + x_1 \cdot \overline{x_4}$$

but also

$$y = \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot x_4 + \overline{x_1} \cdot x_2 \cdot \overline{x_3} \cdot x_4 + x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_4 +$$

$$\overline{x_1} \cdot x_2 \cdot x_3 \cdot \overline{x_4} + x_1 \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot x_2 \cdot x_3 \cdot x_4$$

Although the above three functions suggest to be of a different complexity, from the LUT point of view their complexity is identical: they all fit into a single LUT. In fact the structure of the function implemented by a LUT is of no interest, because this does not give any useful information. This is exactly what the third property mentioned above, is all about: the sum-of-product representation is not a good representation for LUT blocks, because it contains too much information (about the structure of the function) and badly interpreting this information can give a false sense of complexity. Another example of the third property is that the phase of the input and output signals is not important, since inverters can be implemented at the inputs and outputs of every LUT at no extra cost.
The partition theory [HS66] (see Section 2.3) essentially implements what is just needed: input symbols are divided into a number of logic blocks. Each blocks represents the

I	$x_1x_2x_3$	y_1y_2	O
a	000	- -	W,X,Y,Z
b	001	01	X
c	010	11	Z
d	011	00	W
e	100	10	Y
f	101	10	Y
g	11-	1-	Y,Z

Table 4.1: Boolean function F for the realization example

symbols that need to be encoded with the same bit pattern and the operators defined on partitions can be used to build and verify networks of LUTs. Unfortunately, partitions have one fundamental drawback: they cannot handle don't care symbols correctly. In the above example, using partitions there is no way to say that the response on input pattern a is a don't care. Don't care processing is however essential for the construction of optimal and near-optimal Boolean networks.
There are a number of different approaches to extend the partition theory to incorporate don't cares. Each of these methods has its own peculiarities:

1. **Don't care blocks**
   In this concept, all don't care symbols are placed in a special don't care block and the partition sum and product operators are extended for use with these blocks. This approach is a generalization from the bit partitions introduced in Chapter 2. For example, a partition with a don't care block (π_{db}) could model the function in Table 4.1 like this:
   $$\pi_{db} = \left\{\overline{b};\ \overline{c};\ \overline{d};\ \overline{e,f,g};\ \overline{(a)}\right\} \tag{4.1}$$
   However, this representation can be too coarse, since it assumes that a symbol is a care symbol (belonging to exactly one block) or it is a don't care (that does belong into the don't care block). In general, it is also possible that a don't care belongs to a subset of all partition blocks, for example the symbol g is placed together with e and f in one block of π_{db}. This is however not completely correct, symbol g is partial don't care and can also belong in the block c. This cannot be expressed using don't care blocks.

2. **Set systems**
   A set system is a generalization of partition that allows each symbol to be placed in one or more blocks. Don't care conditions are modeled by adding a don't care symbol to each block whose value the don't care can take. The function above can be modeled using set systems as follows:
   $$\phi = \left\{\overline{a,b};\ \overline{a,c,g};\ \overline{a,d};\ \overline{a,e,f,g}\right\} \tag{4.2}$$
   This is exactly the information that needs to be modeled. Unfortunately, compared to partition theory set systems have a nasty property, they are not canonical, i.e.

the same information can be represented using a number of different set systems, for example instead of the block $\overline{a,c,g}$ the set system ϕ could also consist of the three blocks $\overline{a,c}$, $\overline{a,g}$ and $\overline{c,g}$ and model the same information.

3. **Using sets of incompatibility pairs**
This last way of handling don't cares represents all information by a set of incompatibility pairs. An incompatibility pair $a \mid b$ is a pair that indicates that the two elements a and b cannot be encoded by the same bit pattern, because difference between these symbols is required. Incompatibility sets implement information in a unique way, however this representation is not compact: a set of incompatibility pairs can easily become very large. For example the incompatibility set representing the function found in Table 4.1 is:

$$\{(b,c);\ (b,d);\ (b,e);\ (b,f);\ (b,g);\ (c,d);\ (c,e);\ (c,f);\ (d,e);\ (d,f);\ (d,g)\} \tag{4.3}$$

From these possibilities, the set of incompatibility pairs models exactly that what is needed: a representation that is able to distinguish a number of symbols. Unfortunately, for logic synthesis, it is not easy to deal with these sets, because there is no straightforward relation with bit patterns. In particular, it is not easy to determine how many bits are necessary to implement the information present in a set of incompatibility pairs. Classical set systems also model incompatibility pairs, but unfortunately they are not canonical: different set systems can model the same information (the same set of incompatibility pairs). For partitions with more than two blocks, it is possible to have a don't care, that belongs to a subset of the partition blocks. Therefore, partitions with don't care blocks are not useful because they cannot handle this kind of don't cares (in partitions with don't care blocks the don't care is always assumed to be don't care for all blocks).

In the next section a modified set system theory is presented that has all the properties of the classical set systems, but has the additional advantage of representing information in a unique canonical way. In this section the relation between incompatibility sets and set systems will also be further examined.

4.2.2 Set system theory

In this section the set system theory will be introduced. It will be shown that there is a large resemblance with partition theory but there are also a number of important differences. The theory in this section is partly based on the set system theory introduced by Hartmanis e.a. [HS66] but extended in a specific way for the LUT synthesis purpose. Development of the set system theory is not the goal of this thesis, therefore only those parts of the theory that are required for the purpose of the considered logic synthesis method will be described in this thesis.

Hartmanis e.a. [HS66] introduced an extension to their partition theory that they called set system theory. The advantage of this set system theory over partition theory is that it can model naturally don't cares which is essential for the efficient synthesis of

complex multiple-level Boolean functions. Unfortunately, the theory of Hartmanis e.a has a number of strange properties with respect to the information point of view. For example, the $\leq$ operator for set system does not always yield the result which should be expected from the information point of view (if two set systems ϕ and ψ implement the same information, then it may happen that $\phi \leq \psi$ but also that $\psi \not\leq \phi$). In [Kon95] an attempt has been made to fix it, by introducing a *Red*() operator. Unfortunately, there is only an informal definition of this operator.
In this section, the set system theory of Hartmanis [HS66] will be reformulated, such that the set system theory is consistent with the information point of view. It will be proved, that the *Red*() in fact represents the maximum element of a lattice.

Definition 4.1 *Compatibility relation*
Let V be a finite non-empty set of symbols. A binary relation $\simeq$ is called a compatibility relation on V if and only if it is reflexive and symmetric, i.e.:

$$\simeq \subseteq V \otimes V \tag{4.4}$$

such that

$$\underset{v \in V}{\forall} (v, v) \in \simeq \tag{4.5}$$

and

$$\underset{(v,w) \in \simeq}{\forall} (w, v) \in \simeq \tag{4.6}$$

A pair of compatible symbols is called a *compatibility pair*. The notation $a \simeq b$ will be used to denote that $(a, b) \in \simeq$. The set of all compatibility pairs of a certain compatibility relation $\simeq$ is called the *compatibility set*. □

A pair of symbols that is not compatible is called an *incompatibility pair* and the set of all incompatibility pairs is called the *incompatibility set*. In this section compatibility pairs are used; however later in this thesis, it is more convenient to use incompatibility pairs and incompatibility sets. This is of course no problem, since the incompatibility set is just the complement of the compatibility set. A compatibility pair is the most elementary unit of information (abstraction) that is used. In relation to the logic synthesis, a compatibility pair expresses that two compatible symbols will be encoded with the same bit pattern and thus that these symbols cannot be distinguished from each other. Similarly, the incompatibility set represents all pairs of symbols that will be encoded with different patterns and hence can be distinguished.

Example 4.1 *Compatibility and incompatibility set*
In this example the function presented in Table 4.1 is used again. The function table consists of the following input symbols (patterns):

$$V = \{a, b, c, d, e, f, g\} \tag{4.7}$$

The function can be represented with the compatibility relation $\simeq \subseteq V \otimes V$:

$$\begin{aligned} \simeq \; = \; & \{(a, b);\ (a, c);\ (a, d);\ (a, e);\ (a, f); \\ & (a, g);\ (c, g);\ (e, f);\ (e, g);\ (f, g)\} \end{aligned} \tag{4.8}$$

which represents the pairs of different input symbols that need not to be distinguished by the different output values. Note that for reasons of brevity the compatibility relation $\simeq$ is not completely enumerated: for all $x, y \in V$ the pairs (x, x) are missing and only one of the pairs (x, y) and (y, x) is present.

Each pair of input symbols in relation $\simeq$ can be encoded with the same output bit vector. The complement of $\simeq$ is the incompatibility set $|$:

$$|= \{(b,c);\ (b,d);\ (b,e);\ (b,f);\ (b,g);\ (c,d);\ (c,e);\ (c,f);\ (d,e);\ (d,f);\ (d,g)\} \quad (4.9)$$

the pairs in $|$ indicate that these symbols should be encoded using different output vectors. □

The relation between compatibility sets and set systems on one hand and encoding on the other hand will be more thoroughly examined in the Section 4.2.3.

Definition 4.2 *Compatible block*
A set $b \subseteq V$ is called a compatible block with respect to compatibility relation $\simeq \subseteq V \otimes V$ (denoted as $CB_{\simeq}(b)$) if and only if each pair of symbols in b is compatible:

$$CB_{\simeq}(b) \Leftrightarrow \forall_{v,w \in b}\ v \simeq w \quad (4.10)$$

□

Definition 4.3 *Maximal compatible block*
A set $b \subseteq V$ is called a maximal compatible block with respect to compatibility relation $\simeq \subseteq V \otimes V$ (denoted as $MB_{\simeq}(b)$) if and only if the block is compatible and no other symbol from V is compatible with all the symbols in b:

$$MB_{\simeq}(b) \Leftrightarrow CB_{\simeq}(b) \wedge \forall_{v \in V \setminus b}\ \exists_{w \in b}\ v \not\simeq w \quad (4.11)$$

□

Example 4.2 *Compatible and maximal compatible block*
In this example the symbol set V and the compatibility relation $\simeq$ from the previous example will be used again. The following blocks are some examples of compatible blocks:

$CB_{\simeq}(\{a\})$, $CB_{\simeq}(\{e,f\})$ and $CB_{\simeq}(\{e,f,g\})$

The following blocks are examples of maximal compatible blocks:

$MB_{\simeq}(\{a,b\})$, $MB_{\simeq}(\{a,c,g\})$ and $MB_{\simeq}(\{a,e,f,g\})$

□

Definition 4.4 *Set system*
Let V be a finite non-empty set of elements. A set system $\phi_{\simeq}$ is a representation for a compatibility relation $\simeq$ defined on V if and only if:

$$\phi_{\simeq} \subseteq 2^V \tag{4.12}$$

such that:

1. The subsets are compatible:
$$\forall_{b \in \phi_{\simeq}} \; CB_{\simeq}(b) \tag{4.13}$$
2. The set system is non redundant:
$$\forall_{b, b' \in \phi_{\simeq}} \; b \subseteq b' \Rightarrow b = b' \tag{4.14}$$
3. All pairs of $\simeq$ are present:
$$\forall_{(v,w) \in \simeq} \; \exists_{b \in \phi_{\simeq}} \; \{v, w\} \subseteq b \tag{4.15}$$

□

The subsets of a set system are called *blocks* and the notation convention for set systems is similar to that of partitions: the blocks are overlined and the symbols in the blocks are comma separated. The blocks itself are separated by semicolons. If there is no doubt for which compatibility relation the set system is a representation, then ϕ is used instead of $\phi_{\simeq}$. The notation $SS(V)$ will be used to denote all set systems that can be defined on a set V.

Example 4.3 *Notation of a set system*
Let $V = \{1, 2, 3, 4\}$ and let $\simeq \subseteq V \otimes V$ be a compatibility relation defined as follows:

$$\simeq = \{(1,2);\ (2,3);\ (2,4);\ (3,4)\}$$

Then

$$\phi_{\simeq} = \{\overline{1,2};\ \overline{2,3,4}\}$$

is a set system defined on V by $\simeq$. □

From these definitions and examples it follows that a set system is similar to a partition. In a set system however each symbol can be placed in more than one block. A partition is a special set system where each of the symbols is placed in exactly one block. Two special set systems will be distinguished: the set system with all elements in a single block will be called the *identity set system* (denoted as $\phi_{\simeq}(I)$) and the set system with each symbol in a singleton is called the *zero set system* (denoted as $\phi_{\simeq}(0)$).
A set system expresses information about the symbols allowing some symbols to be distinguished from some others. Two symbols c and d can be distinguished from each

other by a certain set system ϕ if and only if there is no block of ϕ that contains them both. If a symbol is an element of more than one subset than this expresses the loss of information. It is not known to which of the blocks this symbol belongs. As opposed to fuzzy logic, there is no weight associated with uncertain symbols, the symbol belongs with the same amount of certainty to each of the blocks. The fact that symbols can belong to more than one block, will be used to express don't cares in the circuit.
Unfortunately set systems are not unique representations for a compatibility relation $\simeq$. In fact, a whole set of set systems that implement $\simeq$ can be defined:

Definition 4.5 *Set system set Setsys*$(\simeq)$
$Setsys(\simeq)$ is the set of all set systems that represent a certain compatibility relation $\simeq$. □

Example 4.4 *Setsys*$(\simeq)$
Let $V = \{1,2,3,4\}$ and let $\simeq \subseteq V \otimes V$ be a compatibility relation defined as:

$$\simeq = \{(1,2);\ (1,3);\ (2,3);\ (2,4)\}$$

Then

$$Setsys(\simeq) = \left\{\{\overline{1,2,3};\ \overline{2,4}\};\ \{\overline{1,2};\ \overline{1,3};\ \overline{2,3};\ \overline{2,4}\}\right\}$$

□

$Setsys(\simeq)$ defines all set systems that implement the same compatibility information. This is in fact the source of all trouble in the Hartmanis set system theory; a number of set systems implement the same information, but the set system operators defined by Hartmanis do not acknowledge this fact and can produce different results for different representations of the same compatibility set.

If this problem is more closely examined, it turns out that the problem is caused by the fact that the blocks of a set system need not be maximal; in the example above the pairs $(1,2)$, $(1,3)$ and $(2,3)$ can be implemented in a single maximal block, or can be implemented as three blocks with two elements. Therefore, it makes sense to define a canonical set system with maximal blocks and to use this canonical set system as the representative of $Setsys(\simeq)$. Such a canonical representative can be defined as follows:

Definition 4.6 *Canonical representative of* $\simeq$
Let V be a finite non-empty set of symbols. $M_{\simeq}$ is called the canonical representative of a compatibility relation $\simeq \subseteq V \otimes V$. $M_{\simeq}$ is defined as

$$M_{\simeq} = \left\{b \subseteq 2^V \mid MB_{\simeq}(b)\right\} \tag{4.16}$$

□

Theorem 4.6 $(Setsys(\simeq), \sqsubseteq)$ *is a lattice*
Let V be a finite non-empty set of symbols. Let $\simeq \subseteq V \otimes V$ a compatibility relation defined on V. Define $\sqsubseteq$ as:

$$\forall_{\phi_{\simeq}, \psi_{\simeq} \in Setsys(\simeq)} \quad \phi_{\simeq} \sqsubseteq \psi_{\simeq} \text{ if and only if } \forall_{b \in \phi_{\simeq}} \exists_{b' \in \psi_{\simeq}} \ b \subseteq b' \tag{4.17}$$

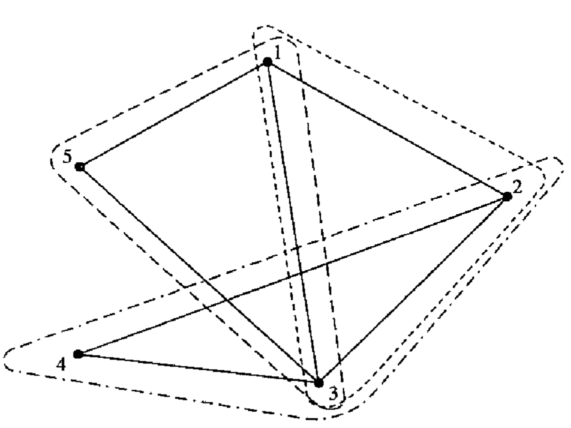


Figure 4.2: Compatibility graph for ϕ_1 and ϕ_2

then $(Setsys(\simeq), \sqsubseteq)$ forms a lattice and $M_{\simeq}$ and $m_{\simeq}$ defined by

$$M_{\simeq} = \left\{ b \subseteq 2^V \mid MB_{\simeq}(b) \right\} \tag{4.18}$$

$$m_{\simeq} = \{\{v, w\} \in V \otimes V \mid v \simeq w \wedge v \neq w\} \cup \left\{ \{v\} \in V \mid \forall_{w \in V \setminus \{v\}} w \not\simeq v \right\} \tag{4.19}$$

form the top and respectively the bottom of the lattice. □

The proof of this theorem can be found in appendix A.1. The importance of the theorem is that the choice of $M_{\simeq}$ as the canonical representative is not arbitrary but a logical choice, it contains all the compatibility pairs in the most compact form.
The question is how the canonical representative can be obtained from an arbitrary set system or compatibility set. This will be illustrated using the following example.

Example 4.5 *Calculation of* $M_{\simeq}$
Let $V = \{1, 2, 3, 4, 5\}$. V is a finite non-empty set. On V the compatibility relation $\simeq \subseteq V \otimes V$ is defined as follows:

$$\simeq = \{(1,2), (1,3), (1,5), (2,3), (2,4), (3,4), (3,5)\}$$

Set system $\phi_{\simeq}$ defined as:

$$\phi_{\simeq} = \left\{ \overline{1,2};\ \overline{1,3,5};\ \overline{2,3,4} \right\}$$

represents compatibility relation $\simeq$.
The maximal set system of $\simeq$ can be calculated by representing the information as a graph of compatibility pairs and calculate maximal connected subgraphs (also known

as cliques) of this graph. For example for the set systems $\phi_{\simeq}$ and compatibility relation $\simeq$ mentioned above the graph in Figure 4.2 can be drawn. In such a compatibility graph, each vertex represents a symbol and an edge between two vertices means that the associated two symbols are compatible. By using a standard algorithm for finding maximal connected subgraphs the following maximal set system is found:

$$\psi_{\simeq} = \left\{\overline{1,2,3};\ \overline{1,3,5};\ \overline{2,3,4}\right\}$$

It can be checked very easily that $\psi_{\simeq} \in Setsys(\simeq)$. □

In practice, the above algorithm will not often be needed. It will be proved later in this section, that the operators defined on set systems preserve canonicity.

Definition 4.7 *$\leq$ operator defined on set systems*
Let V be a non-empty set of symbols and let $\simeq \subseteq V \otimes V$ and $\simeq' \subseteq V \otimes V$ be two compatibility relations defined on V. Set systems $\phi_{\simeq} \in Setsys(\simeq)$ is called smaller or equal than set system $\psi_{\simeq'} \in Setsys(\simeq')$, written $\phi_{\simeq} \leq \psi_{\simeq'}$ if and only if: $\simeq \subseteq \simeq'$ □

In other words, a set system is called smaller or equal than another set system, if it implements fewer compatibility pairs. This makes sense from the informational point of view. Please note that if the set systems are in fact partitions then the $\leq$ corresponds to the $\leq$ operator for partitions.
The definition used here differs from the definition used by Hartmanis e.a [HS66]. The following theorem establishes the link.

Theorem 4.7 *$\leq$ operator defined on canonical set systems*
Let V be a non-empty set of symbols and let $\simeq \subseteq V \otimes V$ and $\simeq' \subseteq V \otimes V$ be two compatibility relations defined on V. Then

$$M_{\simeq} \leq M_{\simeq'} \Leftrightarrow \underset{b \in M_{\simeq}}{\forall} \ \underset{c \in M_{\simeq'}}{\exists} \ b \subseteq c \tag{4.20}$$

□

The proof of this theorem can be found in appendix A.2. The right hand of the expression is used by Hartmanis as the $\leq$ operator. The theorem states that the $\leq$ operator defined on canonical set systems is equal to the Hartmanis definition of $\leq$. However, if set systems are not canonical using the Hartmanis definition, it is possible to have two set systems $\phi_{\simeq}, \psi_{\simeq} \in Setsys(\simeq)$ such that $\phi_{\simeq} \leq \psi_{\simeq}$ but $\psi_{\simeq} \not\leq \phi_{\simeq}$, whereas both set systems implement the same information. This is not possible using the $\leq$ operator defined above.

Example 4.6 *Set system $\leq$ operator*
Let $V = \{1,2,3,4\}$. Then for example the following relations are satisfied:

$$\left\{\overline{1,2};\ \overline{3,4}\right\} \leq \left\{\overline{1,2};\ \overline{1,3,4};\ \overline{2,4}\right\} \tag{4.21}$$

$$\left\{\overline{1,2};\ \overline{2,3};\ \overline{1,3};\ \overline{4}\right\} \leq \left\{\overline{1,2,3};\ \overline{4}\right\} \tag{4.22}$$

$$\left\{\overline{1,2,3};\ \overline{4}\right\} \leq \left\{\overline{1,2};\ \overline{2,3};\ \overline{1,3};\ \overline{4}\right\} \tag{4.23}$$

$\square$

Note that the set systems in the second and third equation in the example above represent the same compatibility set. It should be noted that a set system that is the smaller (as defined by the $\leq$ operator) of two set systems can be the set system with the smallest number of blocks; for partitions this is impossible.

Definition 4.8 *Set system product*
Let $\phi_{\simeq} \in Setsys(\simeq)$ and $\psi_{\simeq'} \in Setsys(\simeq')$ two set systems. The set system product $\phi_{\simeq} \cdot \psi_{\simeq'}$ can then be defined as

$$\phi_{\simeq} \cdot \psi_{\simeq'} = \left\{ x \mid \underset{b_1 \in \phi_{\simeq}}{\exists} \ \underset{b_2 \in \psi_{\simeq'}}{\exists} \ x = b_1 \cap b_2 \wedge \underset{b_1' \in \phi_{\simeq}}{\forall} \ \underset{b_2' \in \psi_{\simeq'}}{\forall} \ x \subseteq b_1' \cap b_2' \Rightarrow x = b_1' \cap b_2' \right\} \tag{4.24}$$

$\phi_{\simeq} \cdot \psi_{\simeq'}$ is a set system and represents the following compatibility relation:

$$\simeq'' = \left\{ (v, w) \in V \otimes V \mid \underset{b \in \phi_{\simeq} \cdot \psi_{\simeq'}}{\exists} \{v, w\} \in b \right\} \tag{4.25}$$

i.e. $\phi_{\simeq} \cdot \psi_{\simeq'} \in Setsys(\simeq'')$ $\square$

The definition of the set system product is the same as the product defined by Hartmanis e.a [HS66].

Example 4.7 *Set systems product*
Let set V be $V = \{1, 2, 3, 4, 5\}$. Then the following set system product can be calculated:

$$\{\overline{1,2,3};\ \overline{1,4,5}\} \cdot \{\overline{1,2};\ \overline{3,4,5}\} = \{\overline{1,2};\ \overline{3};\ \overline{4,5}\} \tag{4.26}$$

$\square$

The set system product has the same practical meaning as the partition product. The set system product combines the information of the two set systems into one new set system. In the example, the first set system is able to distinguish symbol 3 from symbol 4. The second set system can distinguish symbol 2 from 3. So, the product set system is able to distinguish symbols from both pairs. On the other hand, neither set system is able to distinguish 1 from 2 so the set system product is unable to do it.

Theorem 4.8 *The set system product preserves canonicity*
Let $M_{\simeq} \in Setsys(\simeq)$ and $M_{\simeq'} \in Setsys(\simeq')$ be two *canonical* set systems, and let the set system product $M_{\simeq} \cdot M_{\simeq'} \in Setsys(\simeq'')$. Then

$$M_{\simeq''} = M_{\simeq} \cdot M_{\simeq'} \tag{4.27}$$

$\square$

The proof can be found in appendix A.3. The consequence of this theorem is extremely important. It states that the set system product of two canonical set systems is another canonical set system. This is important because the operators $\leq$ and $\cdot$ are unambiguously and naturally defined on set systems. Also, the $\leq$ operator can be efficiently checked for canonical set systems. Furthermore, it is not necessary to make set systems explicitly canonical (which can be a time consuming operation). Finally, from the information point of view canonical set systems are sound. In other words, canonical set systems provide a well defined, compact, unambiguous and easy to interpret mathematical framework for LUT based logic synthesis.

Like in partition theory, the symbol and bit representation of the circuit can be coupled using bit set systems. Unlike partitions, set systems are able to handle don't care bits. Let $B = \{b_1, b_2, .., b_{|B|}\}$ be a set of (input or output) bits. Let $T = \{t_1, t_2, ..t_{|T|}\}$ be a set of (input or output) symbols. Each input/output bit $b_k \in B$, introduces a two-block set system $\phi_T(b_k)$ on the set of symbols (bit value patterns) T: one block of $\phi_T(b_k)$ contains the symbols for which bit b_k has the value 0 and the other block contains the symbols for which b_k has the value 1. If a bit has the don't care value, then the symbol is placed in both blocks. The product of the set systems $\phi_T(b_k)$ for all the bits $b_k : b_k \in B$ will unambiguously define the set of all input/output symbols, i.e. it will be a zero set system on T.

Definition 4.9 *Symbol set system induced by a bit partition*
ϕ_T is a symbol set system induced by a bit partition π_B ($\phi_T \in ind(\pi_B)$) if and only if

$$\phi_T \geq \prod_{b_k \in (B - dcb(\pi_B))} \phi_T(b_k) \tag{4.28}$$

□

Definition 4.10 *The smallest symbol set system induced by a bit partition*
ϕ_T is the smallest symbol set system induced by a bit partition π_B ($\phi_T = ind^*(\pi_B)$) if and only if

$$\phi_T = \prod_{b_k \in (B - dcb(\pi_B))} \phi_T(b_k) \tag{4.29}$$

□

From the above to definitions it follows that any set system $\phi_T \geq ind^*(\pi_B)$ is an induced set system of π_B.

Definition 4.11 *Bit partition induced by a symbol set system*
π_B is a bit partition induced by a symbol set system ϕ_T ($\pi_B \in ind(\phi_T)$) if and only if

$$\forall_{b_k \in (B - dcb(\pi_B))} \quad \phi_T(b_k) \geq \phi_T \tag{4.30}$$

□

I	$x_1x_2x_3x_4$	y_1
a	0-00	0
b	0-01	1
c	-010	0
d	--11	0
e	0110	-
f	1-0-	1
g	1110	0

Table 4.2: Function table for bit partition and set systems example

If $\phi_T \in ind(\pi_B)$ then, having π_B the blocks of ϕ_T can be computed. If $\pi_B \in ind(\phi_T)$ then, having the block of ϕ_T the values of all the important bits from π_B can be computed.

Example 4.8 *Bit partitions and set systems*
In this example the function from Table 4.2 will be considered as a binary combinational machine. Bit-partitions on the input bits can be made. The set of the input bits is called X, i.e. $X = \{x_1, x_2, x_3, x_4\}$. The input symbols are labeled with lowercase letters from the alphabet, i.e. the set I of input symbols is defined as $I = \{a, b, c, d, e, f, g, h\}$. A binary input variable (bit) represents a symbolic two block set system such that all symbols for which the bit is zero are placed in one block, all symbol for which the bit is one are placed in the second block and symbols for which the bit is don't care are placed in both blocks: $\phi_I(x_1) = \left\{\overline{a, b, c, d, e};\ \overline{c, d, f, g}\right\}$, $\phi_I(x_2) = \left\{\overline{a, b, c, d, f};\ \overline{a, b, d, e, f, g}\right\}$ etc.
A bit partition on X is the partition $\pi_X = \{\overline{x_2};\ \overline{x_3};\ (\overline{x_1, x_4})\}$. Given any set system ψ, ψ is a symbol set system induced by π_X if and only if:

$$\psi \geq \prod_{b_x \in (X - dcb(\pi_X))} \phi_X(b_x)$$

which can be calculated as follows:

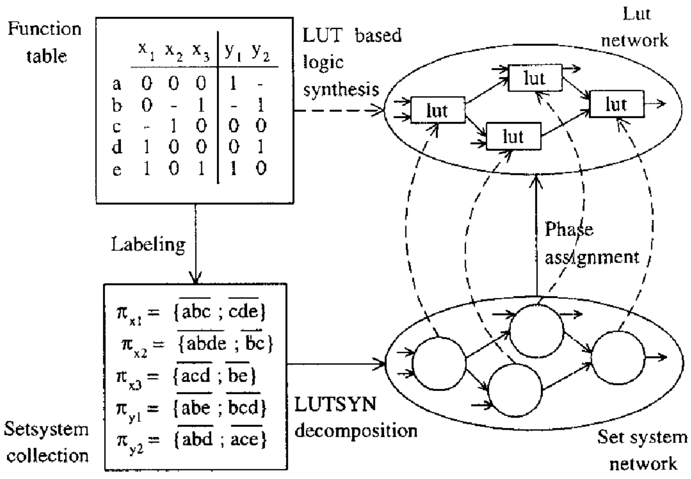
$$\psi \geq \phi_X(x_2) \cdot \phi_X(x_3) \Rightarrow$$

$$\psi \geq \left\{\overline{a, b, c, d, f};\ \overline{a, b, d, e, f, g}\right\} \cdot \left\{\overline{a, b, f};\ \overline{c, d, e, g}\right\} \Rightarrow$$

$$\psi \geq \left\{\overline{a, b, f};\ \overline{c, d};\ \overline{d, e, g}\right\}$$

Similarly, an example of a bit partition induced by the symbol set system $\left\{\overline{a};\ \overline{b};\ \overline{c, e};\ \overline{d};\ \overline{f};\ \overline{c, g}\right\}$ is the partition $\tau_X = \{\overline{x_1};\ \overline{x_3};\ \overline{x_4};\ (\overline{x_2})\}$. □

The set system theory presented here satisfies at least the first three of the four properties mentioned in Section 4.2.1: the defined set systems algebra has nice mathematical



	x_1	x_2	x_3	y_1	y_2
a	0	0	0	1	-
b	0	-	1	-	1
c	-	1	0	0	0
d	1	0	0	0	1
e	1	0	1	1	0

$\pi_{x1} = \{\overline{abc} ; \overline{cde}\}$
$\pi_{x2} = \{\overline{abde} ; \overline{bc}\}$
$\pi_{x3} = \{\overline{acd} ; \overline{be}\}$
$\pi_{y1} = \{\overline{abe} ; \overline{bcd}\}$
$\pi_{y2} = \{\overline{abd} ; \overline{ace}\}$

Figure 4.3: LUTSYN synthesis framework model

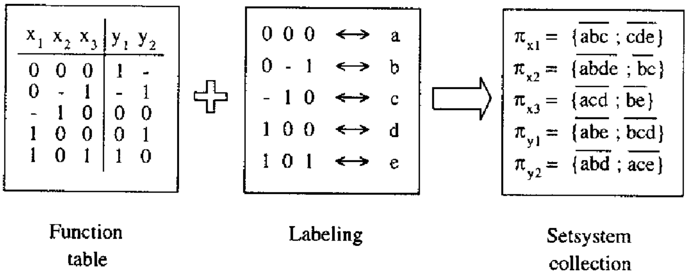
properties and has powerful operators. The set systems represent the information in a compact way and the information that is modeled is precisely all the information that is necessary. The fourth property, the relation between set system theory and the synthesis of Boolean functions using LUTs will be investigated in the next section.

4.2.3 Synthesis framework model

LUTSYN uses the set system representation for describing the information of primary inputs, of primary outputs, of intermediate signals and of the LUTs and uses the set system operators defined in the previous section to construct a network of LUTs that implements the required function. In Figure 4.3 the synthesis process is illustrated. The input of the algorithm consists of a function table of an incompletely specified Boolean function. Instead of a direct implementation into a LUT based FPGA, the function table representation is transformed into a collection of set systems. Details of the transformation process can be found in Section 4.2.4.

The set system collection is then used to construct a set system network. This transformation forms the core of LUTSYN. The transformations are expressed using set systems and operators on set systems. The main transformation step itself is outlined in Section 4.3 of this thesis.

A basic property of the so constructed set system network is that each of its nodes can be implemented using a single LUT. The transformation from the set system network to the corresponding LUT network is therefore straightforward. Because set systems do not model the phase of Boolean signals, this transformation can only be performed if for each of the set systems an encoding of the blocks has been chosen. This is however



x_1	x_2	x_3	y_1	y_2
0	0	0	1	-
0	-	1	-	1
-	1	0	0	0
1	0	0	0	1
1	0	1	1	0

0 0 0	$\longleftrightarrow$	a
0 - 1	$\longleftrightarrow$	b
- 1 0	$\longleftrightarrow$	c
1 0 0	$\longleftrightarrow$	d
1 0 1	$\longleftrightarrow$	e

$$\pi_{x1} = \{\overline{abc}\ ;\ \overline{cde}\}$$
$$\pi_{x2} = \{\overline{abde}\ ;\ \overline{bc}\}$$
$$\pi_{x3} = \{\overline{acd}\ ;\ \overline{be}\}$$
$$\pi_{y1} = \{\overline{abe}\ ;\ \overline{bcd}\}$$
$$\pi_{y2} = \{\overline{abd}\ ;\ \overline{ace}\}$$

Figure 4.4: Translation from function table to set system collection

a trivial problem, since LUTs can implement input and or output inverters at no extra cost. The translation from set system network to a LUT network can be found in 4.2.5.

In the next section the first transformation step (from function table to set system collection) will be discussed.

4.2.4 From function table to set systems

The basic transformation process is illustrated in Figure 4.4.The translation from function table to input and output set systems is performed similarly to the translation into partition representation. Each product term will be labeled by a unique symbol. For each input/output bit a two-block set system on this symbol set is then constructed by placing the symbols for which the bit is zero into one block and placing the symbols for which the bit has 1 as its value in another block. Symbols for which the bit is don't care (for which it is not important, whether it is 0 or 1) are placed in both blocks. A set system collection is composed of the set systems of all input and output bits of a certain Boolean function and describes the Boolean function in the set system notation.

Example 4.9 *From function table to set systems*
In Table 4.3 a 4-input, 2-output incompletely specified Boolean function is presented. From the function table the following set systems can be derived:

$$\phi_{x_1} = \{\overline{a, b, d, e, f, g};\ \overline{a, b, c, h}\}$$

$$\phi_{x_2} = \{\overline{a, b, c, d, e};\ \overline{b, c, d, f, g, h}\}$$

$$\phi_{x_3} = \{\overline{a, b, g, h};\ \overline{c, d, e, f}\}$$

$$\phi_{x_4} = \{\overline{a, c, e, f, g, h};\ \overline{b, c, d}\}$$

I	$x_1x_2x_3x_4$	y_1y_2
a	-010	01
b	--11	01
c	1-0-	10
d	0-01	11
e	0000	00
f	0100	00
g	0110	11
h	1110	--

Table 4.3: Non minimized function table

$$\phi_{y_1} = \{\overline{a, b, e, f, h};\ \overline{c, d, g, h}\}$$

$$\phi_{y_2} = \{\overline{a, b, d, g, h};\ \overline{c, e, f, h}\}$$

The four input bit set systems (ϕ_{x_1}, ϕ_{x_2}, ϕ_{x_3} and ϕ_{x_4}) induce together the zero symbol set system (which can be easily verified by calculating the set system product of the input bit set systems). This means that the information of the four bits is sufficient to unambiguously determine each product term. □

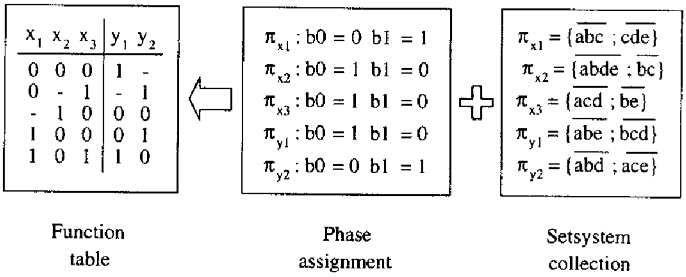
From this example it can be seen that the set system representation depends on the cubes that are used to represent the function. If the function could be expressed in min terms instead of some larger product terms, then 16 symbols would be required in the complete set system representation. The relation between the function table representation and the set system representation is discussed further in Section 4.3.4 where the preprocessor for LUTSYN will be described.

Theorem 4.9 *Existence of a set system representation*
For each NI-input, NO-output Boolean function a canonical set system representation exists. □

The proof of this theorem is trivial; each Boolean function can be represented with a function table of 2^{NI} min terms, and to this table, the set system construction process described above can be applied.

4.2.5 From set system networks to LUT network

In this section the transformation from a set system network to a LUT network will be discussed (see Figure 4.5). As mentioned before the LUTSYN algorithm will construct a set system network such that each of the nodes in this network can be implemented using a single LUT. The following theorem states the sufficient condition that a set system



x_1	x_2	x_3	y_1	y_2
0	0	0	1	-
0	-	1	-	1
-	1	0	0	0
1	0	0	0	1
1	0	1	1	0

π_{x1} : b0 = 0 b1 = 1
π_{x2} : b0 = 1 b1 = 0
π_{x3} : b0 = 1 b1 = 0
π_{y1} : b0 = 1 b1 = 0
π_{y2} : b0 = 0 b1 = 1

$\pi_{x1} = \{\overline{abc}\ ; \overline{cde}\}$
$\pi_{x2} = \{\overline{abde}\ ; \overline{bc}\}$
$\pi_{x3} = \{\overline{acd}\ ; \overline{be}\}$
$\pi_{y1} = \{\overline{abe}\ ; \overline{bcd}\}$
$\pi_{y2} = \{\overline{abd}\ ; \overline{ace}\}$

Figure 4.5: Translation from a set system collection to a function table

node should satisfy in order to be implementable by a single LUT.

Theorem 4.10 *Set system collection implementable by a LUT*
Let V be a set of symbols. A set system collection defined on V consisting of NOI two-block input set systems $\phi_i (1 \leq i \leq NOI)$

$$\phi_i \in SS(V) \wedge \|\phi_i\| = 2 \tag{4.31}$$

and of NOO two-block output set systems $\psi_j (1 \leq j \leq NOO)$

$$\psi_j \in SS(V) \wedge \|\psi_i\| = 2 \tag{4.32}$$

is implementable by a $LUTNOI$-input, $LUTNOO$-output LUT if and only if:

1. The number of inputs and outputs of the set system collection does not exceed the number of available inputs and outputs of the LUT, i.e.:

$$NOI \leq LUTNOI \wedge NOO \leq LUTNOO \tag{4.33}$$

2. Each of the outputs is a function of the inputs, i.e.:

$$\forall_{1 \leq i \leq NOO} \prod_{1 \leq j \leq NOI} \phi_j \leq \psi_i \tag{4.34}$$

□

Proof:
Because all the set systems have two blocks, each of them can be represented by a single bit. Therefore, a set system collection with NI inputs and NO outputs can be connected to a LUT with $LUTNOI$ inputs and $LUTNOO$ outputs only if Equation (4.33) is satisfied. The set system product of all the input set systems represents all the incompatibility pairs that are implemented by at least one of the input set systems. The output set systems can implement only such incompatibility pairs that are implemented by at least

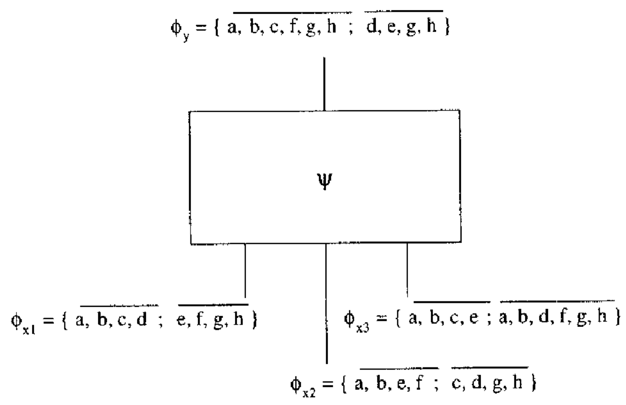

$\phi_y = \{ \overline{a, b, c, f, g, h} \; ; \; \overline{d, e, g, h} \}$
ψ
$\phi_{x1} = \{ \overline{a, b, c, d} \; ; \; \overline{e, f, g, h} \}$
$\phi_{x3} = \{ \overline{a, b, c, e} \; ; \; \overline{a, b, d, f, g, h} \}$
$\phi_{x2} = \{ \overline{a, b, e, f} \; ; \; \overline{c, d, g, h} \}$

Figure 4.6: A LUT block with the corresponding set system collection

one of the input set systems. This is expressed by Equation (4.34). □

LUTSYN will make sure that all the nodes in the set system network satisfy the theorem mentioned above. The translation from set system network to LUT network is therefore straightforward: each set system node description has to be translated into an equivalent LUT description. The network structure itself (the way the nodes are interconnected) does not change. Since set systems do not model the corresponding signal polarity for each set system an encoding of the blocks has to be chosen before the set system node can be translated to a function table. This is however a trivial problem, if the Boolean function is implemented using LUTs. LUTs can implement inverters on their inputs and outputs without any cost, so the phase of the Boolean signals is really unimportant and can be freely chosen, except for the primary inputs and output of the original function for which the phase is given by the function specification table. The following example shows, how the function implemented by a LUT is constructed from a given set system collection.

Example 4.10 *From set system collection to LUT function table*
Table 4.4 represents product terms of a certain function. Since, the outputs of this function are not necessary for this example, they are not presented. Each product term is labeled with a unique symbol. In Figure 4.6 a 3-input, 1-output LUT block and an associated set system collection is presented. The set system collection consists of 3 two-block input set systems and 1 two-block output symbol set system. To verify that this set system collection is implementable by the LUT, it must be shown that it satisfies the theorem stated in the previous section. Equation (4.33) can be verified very easily. To verify Equation (4.34) it is necessary to calculate the combined input set system

I	$x_1x_2x_3x_4$
a	-010
b	--11
c	1-0-
d	0-01
e	0000
f	0100
g	0110
h	1110

Table 4.4: Product terms and their symbol labels

$x_1x_2x_3$	y
000	0
001	0
010	0
011	1
100	1
101	0
110	0
111	1

Table 4.5: Tabular representation of the LUT function

information ψ:

$$\psi = \phi_{x_1} \cdot \phi_{x_2} \cdot \phi_{x_3}$$

$$\psi = \{\overline{a, b, c, d};\ \overline{e, f, g, h}\} \cdot \{\overline{a, b, e, f};\ \overline{c, d, g, h}\} \cdot \{\overline{a, b, c, e};\ \overline{a, b, d, f, g, h}\}$$

$$\psi = \{\overline{a, b};\ \overline{c};\ \overline{d};\ \overline{e};\ \overline{f};\ \overline{g, h}\}$$

To satisfy Equation (4.34) it must be shown that

$$\psi \leq \phi_y$$

which can be done simply by applying the definition of $\leq$.
Because, the set system collection satisfies (4.33) and (4.34) it can be implemented by a 3-input, 1-output LUT.

The input and output set systems together define unambiguously the function implemented by the LUT, with precision to the input and output phase assignment of the

function. In the case of LUT implementation, the input and output phase assignment is however not important and can be freely chosen (except for the primary inputs and outputs), because the implementation cost is the same for any assignment. In LUTSYN, when translating a set system network to a LUT network, the phase assignment of the LUT inputs will be predetermined, whereas the output phases can be randomly chosen. To determine the block function of the LUT, it is necessary to assign to each block from the input and output set systems the corresponding bit values 0 or 1. Suppose, that for ϕ_{x_1}, ϕ_{x_2}, ϕ_{x_3} and ϕ_y the first block is encoded with a 0 and the second block is encoded with a 1. Then the function Table 4.5 can be derived from the set systems as follows. $(0,0,0)$ designates those symbols that are in the first block of ϕ_{x_1}, ϕ_{x_2} and ϕ_{x_3}, i.e. the symbols a and b. Since a and b are in the first block of the output function, the function for that value is a zero. A similar argument holds for product term $(0,0,1)$.
Product term $(0,1,0)$ means the symbols that are in the first block of ϕ_{x_1} and ϕ_{x_3} and in the second block of ϕ_{x_2}. This is the symbol c which is in the first block of ϕ_y and hence the function is 0 for this input vector. This procedure can be repeated for all possible input vectors.
An interesting case is the term $(1,1,0)$. There are no symbols that are in the second block of ϕ_{x_1} and ϕ_{x_2} and in the first block of ϕ_{x_3}. Hence, this input combination will never occur and the function value in this case is a don't care. However, in the actual hardware implementation of a LUT, a zero or one has to be filled in in this case, because hardware blocks cannot implement don't care values. Here, the don't care is replaced by a 0.
Another interesting case is term $(1,1,1)$ that corresponds to the symbols g and h. Apparently, it is not of interest which of these bit values the function will take, since g and h are don't care symbols in the output set system. Once again, in the actual hardware implementation of the logic block a certain bit value (zero or one) needs to be chosen for these symbols In this example the value 1 is used. Maybe this extra information with respect to symbols g and h is not useful in the remaining of the construction process of the circuit, but it can certainly not harm anything and some extra information is provided at no extra hardware cost.

Table 4.5 represents the following Boolean function:

$$y = x_2 \cdot x_3 + x_1 \cdot \overline{x_2} \cdot \overline{x_3}$$

□

A final remark on the phase of the Boolean signals: set systems do not model the phase of the signals because this information is not useful for the synthesis with LUTs. If this information would be useful, then the phase can be simply added by labeling the set system blocks with the desired Boolean values.

4.2.6 Comparison to other representations

In the previous section the set system based LUT network representation has been introduced and the corresponding mathematical framework has been defined. In this section, this representation will be compared to some other well-known representations.

The most distinctive feature of the set system network representation is that it is used in both the logic and the topology model. In fact, in LUTSYN no distinction is made between these two models. This has the advantage that the effect of decisions made in the logic model on the structure can be seen immediately. In many methods, the topology model is derived from the logic model by a unidirectional transformation step. For example, in methods based on algebraic or Boolean decomposition of sum-of-product term functions, the sum-of-product term functions represent chiefly the logic model and the structural model is obtained from this model by technology mapping. In these methods often the error is made, that the structure of the Boolean expressions is identified with the structure of the model. The original reduced ordered binary decision diagrams represent also a logic view.
With respect to the LUTs, the set system representation is an abstraction of the phase of the functions and their inputs. This is just the abstraction which precisely corresponds to LUTs because the phase can be implemented at no cost by LUTs. Therefore, both the function and the inverse function can be used as an input for another LUT at no cost. A sum-of-product term function representation however cannot be complemented easily and efficiently (concurrent decomposition [RV92] is a notable exception, but that method is restricted to small and special cubes).

The set system based LUT network representation models don't cares efficiently and effectively. First of all, the input representation is not restricted in any way: every incompletely specified function can be used as an input of the synthesis methods. This is not the case with many Boolean or algebraic division algorithms, where two-level minimization is used before the function is given to the multiple-level logic synthesis algorithm. Don't cares are also modeled very naturally (there are no changes or exceptions in the set system theory necessary to model don't cares). Furthermore, all the set system operators naturally make use of available don't cares.

Finally, the set system theory satisfies the four important properties of mathematical models stated in Section 4.2.1.

4.3 LUTSYN overview and concepts

4.3.1 Introduction

In this section an overview of LUTSYN will be presented and the most important aspects of the algorithm will be discussed.

The most prominent concept that distinguishes LUTSYN from all other approaches is that the functions are constructed (composed) bottom-up, i.e. starting from the inputs, the function is built level by level until the outputs have been reached. Details regarding the bottom-up construction algorithm can be found in Section 4.3.2.

Another important aspect of LUTSYN is the way the heuristic algorithms make decisions. In LUTSYN, all heuristic decisions are based on a single framework: the similarity between set systems. The maximal similarity for set systems is obtained when the set systems are equal. For all other set systems the similarity expresses the degree of resemblance of these set systems. More information on the set system similarity can be found in Section 4.3.3.

The LUTSYN algorithm consists of three phases: a preprocessor phase, a clustering phase and a construction phase. The basic task of the *preprocessor phase* is to transform a two-level logic sum of term representation of an incompletely specified function into the set system notation. Details on the preprocessing algorithms can be found in Section 4.3.4.

The second phase of the LUTSYN algorithm is the *clustering phase*. In this phase a top-down clustering is performed. The goal of the clustering is to decide for which single output functions it seems to be useful to construct common subfunctions and which subfunctions should be constructed. The developed clustering algorithm is discussed in Section 4.3.5.

The *construction phase* forms the kernel of LUTSYN. This phase performs the actual multiple-level bottom-up construction of the LUT network based on the information obtained from the first two phases. LUTSYN will create a general decomposition without feedback loops. Details of the construction process can be found in Section 4.3.6.

4.3.2 Network construction approaches

In the network construction process subfunctions are constructed such that each of them can be implemented in a LUT and that together form they the desired function. In this context, a function is called feasible if it can be implemented by a single LUT and it is called infeasible if more than one LUT is needed for its implementation.
The construction of a multiple-level network is usually started at the outputs of the network and the construction process consists of a decomposition algorithm that recursively decomposes all non feasible functions. It is also possible to use a bottom-up construction: starting from the inputs a network is build that realizes the specified function. This bottom-up approach is rarely used in multiple-level logic synthesis, and therefore the bottom-up approach itself is worth an investigation. Furthermore, this approach has a number of interesting properties.

In the *top-down approach* the starting point of the synthesis are the two-block output set systems. Any set system that cannot be created using a single LUT, needs to be constructed as the set system product of two or more other set systems. The sharing of set systems between functions is allowed and encouraged. The goal is to find set systems that near-optimal fulfill the optimization goals.
In the *bottom-up approach* the construction process starts with the input set systems. The bottom-up approach is driven by the information that has to be computed, i.e. a new level of the network is constructed based on the set systems that are available (calculated on a previous levels) and with the output set systems of the function to construct and the common subfunctions as the target. The network is thus constructed level by level, such

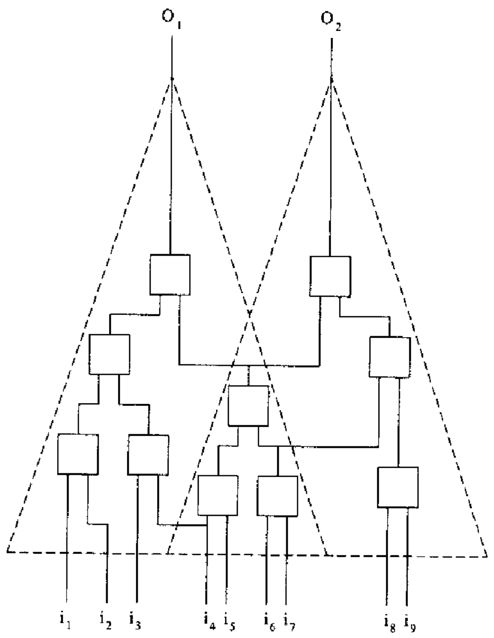


Figure 4.7: Typical logic function network structure

that each of the set systems of a certain level computes the desired output set systems more precisely. The construction process terminates when all the primary output set systems have been constructed. The bottom-up approach has two principle advantages over the top-down construction:

1. A network of fine granularity LUTs has usually the form of a set of overlapping triangles (see Figure 4.7). Each triangle represents the network of a single output. At the base of each triangle the inputs that are used for that output function can be found and the top of the triangles represent the primary outputs. Overlapping parts of the triangles are the common subfunctions. The most logic is present at the lower parts of the tree (near the leaves) and hence the largest gain in area minimization can also be expected to take place in this region. Therefore, it makes sense to use a bottom-up approach, since accurate information is available in the region where minimizations are most effective. Using the top-down approach it would be necessary to estimate the effect of decisions taken at the high levels of the tree on the lower levels of the tree. This can be difficult.

2. In the bottom-up approach, heuristic decisions have a clear goal: the output set

systems have to be constructed at the lowest cost. It is reasonable easy to find some measures that express the similarity of two set systems and hence it can be determined which set systems are likely to be included in the set system collection used to compute a certain output set system. In the top-down approach the goal is less clear, set systems should be split until an input set system is found, but how and which set system to use cannot be determined in a straightforward way.

The concept of bottom-up computational synthesis is successfully used in a number of goal directed compositional methods developed for decomposition of sequential machines [JK91][JV92][JV95]. This idea of bottom-up synthesis is also supported by results obtained by Saucier e.a. [SFA93]. Their lexicographical factorization algorithm provides a mechanism to select which inputs should be inserted at the leaves of the tree and which inputs can be inserted at a higher level. The lexicographical algorithm shows big improvement compared to non lexicographical factorization.

On the other hand, the top-down approach has one major advantage compared to the bottom-up approach: in the top-down approach it is much easier to find and create common subfunctions. After all, in the top-down approach Boolean functions are recursively decomposed and it is relatively easy to decompose them so that they have large common subfunctions. In the bottom-up approach it seems that subfunctions are created by mere chance (a subfunction is created if a set system is created that can be used for two output functions).
The bottom-up approach can however be extended with an analysis algorithm (clustering phase) that determines from the function's description which common subfunctions can have a large impact on the implementation and should therefore actually be constructed and used. The bottom-up construction algorithm will try to build these subfunctions in parallel with the construction of the primary output functions. However, while the 100% implementation of the primary output function is necessary, 100% implementation of the subfunctions is not required. Results of the clustering algorithm specify only some potentially useful (maximal) subfunctions and if during the actual network construction it turns out that such a subfunction is difficult or expensive to construct its smaller subfunctions can be constructed instead or even a common subfunction may not be constructed at all. An additional advantage is that only subfunctions are created that have a large impact on the active area size; this is important because the distribution of subfunction outputs often requires long or even global wires. For small subfunctions, the cost of these extra wires and the extra complexity of the routing problem can have a much larger negative global impact than the area savings that are obtained by using the subfunctions.

Because of the above reasons bottom-up synthesis is used in LUTSYN. To find useful subfunctions a clustering analysis will be performed (Section 4.3.5). The construction process will be discussed in Section 4.3.6.

4.3.3 Calculation of set system similarities

4.3.3.1 Incompatibility relations

A very important aspect of any heuristic search algorithm is the evaluation of the branches of the search tree that need to be traversed by the algorithm. In LUTSYN all information is represented using set systems, so it is necessary to find a way to evaluate and compare set systems. The evaluation process basically consists of comparing two set systems and determining a measure for the similarity that these two set systems have. The calculation of the similarity is one of the most important algorithms of LUTSYN because many decisions are based on it. In Section 4.2.2 canonical set system representatives have been introduced. The theory establishes a one-to-one link between set systems and compatibility sets. In this section the complement of the compatibility set called the incompatibility sets is used to find a measure of similarity between two set systems.

Definition 4.12 *Incompatibility relation*
Let ϕ be a set system on set S. Two symbols $a, b \in S$ are incompatible (denoted as $a \mid b$) if and only if

$$a \mid b \Leftrightarrow \underset{B \in \phi}{\forall} (a \in B \Rightarrow b \notin B) \wedge (b \in B \Rightarrow a \notin B) \tag{4.35}$$

The incompatibility relation | is symmetric, but it is not reflexive nor transitive. If a and b are incompatible, then $a \mid b$ is called a incompatibility pair. □

In other words, two symbols are incompatible if and only if the associated set system can be used to distinguish these two symbols. A set system introduces in fact a set of incompatibility pairs: each pair of symbols that the set system can distinguish is contained in this *incompatibility set*. The incompatibility set for a set system ϕ will be denoted $IC(\phi)$. Each set system uniquely defines an incompatibility set and each incompatibility set defines a unique canonical set system.

Example 4.11 *From set systems to incompatibility sets*
Given set $S = \{1,2,3,4,5\}$ and let ϕ be a set system defined on S:

$$\phi = \left\{\overline{1,2};\ \overline{1,3};\ \overline{2,4,5}\right\}$$

The incompatibility set defined by ϕ is:

$$IC(\phi) = \{(1,4);\ (1,5);\ (2,3);\ (3,4);\ (3,5)\}$$

□

For the determination of the maximal set system that is defined by an incompatibility set an algorithm for finding maximal connected subgraphs of a graph is necessary to determine the largest possible set system blocks. This will be shown using the following example:

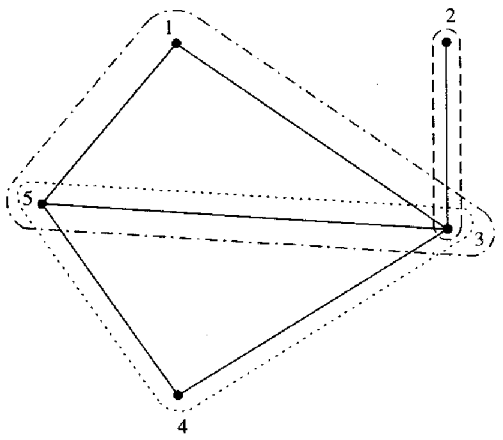


Figure 4.8: Compatibility graph associated with incompatibility set IC

Example 4.12 *From incompatibility set to set system*
Let $S = \{1, 2, 3, 4, 5\}$ be a set of symbols and let IC be the set of incompatibility pairs for some set system ϕ defined on S:

$$IC(\phi) = \{(1,2);\ (1,4);\ (2,5)\}$$

To find the maximal set system associated with $IC(\phi)$ a graph is drawn with an edge between each pair of symbols that are compatible (i.e. an edge for each pair missing from $IC(\phi)$). For $IC(\phi)$ the graph can be found in Figure 4.8. An algorithm for finding maximal connected subgraphs is then applied to the graph and a covering algorithm is used to cover all the edges of the graph using as few as possible maximal connected subgraphs. In Figure 4.8 3 maximal connected subgraphs are detected. Each subgraph is then transformed into a block of the set system, resulting in the following three block set system:

$$\phi = \left\{\overline{1,3,5};\ \overline{2,3};\ \overline{3,4,5}\right\}$$

The graph algorithm determines the largest possible blocks for the set systems and uses these blocks to cover the graph; if this algorithm would not have been used then the following set system ψ can also be derived from the incompatibility set $IC(\phi)$.

$$\psi = \left\{\overline{1,3};\ \overline{1,5};\ \overline{2,3};\ \overline{3,5};\ \overline{3,4,5}\right\}$$

ϕ is the unique maximal set system based on $IC(\phi)$, ψ is also based on $IC(\phi)$ however it is not a maximal set system. □

An efficient method for covering the compatibility graph with connected subgraphs is presented in [Kon95].

The concept of incompatibility pairs has been previously used for sequential machine decomposition [Kon95] and race-free state assignment for level-mode sequential machines [JS90]. It also resembles the concept of dichotomies used in [YC90] where dichotomies are used to find a near optimal multiple-valued encoding whereas the incompatibility pairs are used to determine the similarity between set systems. There is however one important difference between dichotomies on one side and incompatibility pairs and set systems on the other side: dichotomies are two block partitions, where the second block can only contain a single symbol. This way dichotomies are more general than incompatibility pairs (they can incorporate a number of incompatibility pairs that have the same second symbol) but are more restricted than maximal set systems.
In the next subsections the concept of incompatibility pairs will be used to introduce similarity measures for set systems.

4.3.3.2 Similarity of set systems

In the bottom-up construction process, a number of set systems are constructed in parallel. At some points of the algorithm it has to be decided which of these set systems should remain for further construction (because they are likely to be used in the final realization of the circuit) and which set systems can be removed (because it is very unlikely that these set systems will be part of the final set system network). Such a decision is based on the amount of information that is implemented by these set systems. If a set system implements a lot of information for a certain cluster or primary output set system, then it is a good candidate for further development. The similarity measure will be used to indicate the amount of common information.
A simple solution for the calculation of the similarity measure is computing the fraction of the number of incompatibility pairs that is common to both set systems. However, that would not be sufficient because the incompatibility pairs don't have the same importance. For example, if an incompatibility pair $a \mid b$ that is part of one or more output set systems is realized in only few set systems, then keeping the set systems that realize $a \mid b$ is much more important than keeping other set systems. On the other hand, if many set systems implement another incompatibility pair $c \mid d$ then there is no reason to prefer set systems that implement $c \mid d$. This way weighing of incompatibility pairs can be important when comparing set systems. Furthermore, if two set systems implement the same amount of incompatibility pairs of an output set system, one of them can implement all the easy incompatibility pairs whereas the other implements more difficult ones and thus be more valuable.

Therefore the similarity of two set systems will be calculated with weighed incompatibility pairs. The basic formula for the calculation of the similarity of set system s_1 compared to set system s_2 is stated in the following definition.

Definition 4.13 *Set system similarity*
Let S be a symbol set, and let s_1 and s_2 be two set systems defined on symbol set S. The set system similarity of set system s_1 compared to set system s_2 can be calculated as follows:

$$SIM_{s_2}(s_1) = \frac{\sum\limits_{a|b \in IC(s_1) \cap IC(s_2)} w_S(a \mid b)}{\sum\limits_{a|b \in IC(s_2)} w_S(a \mid b)} \tag{4.36}$$

where $w_S(a \mid b)$ is a weighing function for an incompatibility pair $a \mid b$. □

$w_S(a \mid b)$ is used to express the importance of the incompatibility pair $a \mid b$. A number of possible incompatibility weighing functions will be defined later in this section.

Example 4.13 *Set system similarity*
Let $S = \{1, 2, 3, 4\}$ and let

$$s_1 = \left\{\overline{1,2};\ \overline{3,4}\right\}$$

and let

$$s_2 = \left\{\overline{1,2};\ \overline{3};\ \overline{4}\right\}$$

be two set systems defined on S. Define $w_S(a \mid b)$ as the constant function 1. It can then easily be verified that

$$SIM_{s_2}(s_1) = 0.8$$

and

$$SIM_{s_1}(s_2) = 1.0$$

□

The function

$$w_S(a \mid b) : S \otimes S \rightarrow [0, ..., 1] \tag{4.37}$$

is called the *incompatibility weighing function*, it associates a real number between 0 and 1 inclusive to each incompatibility pair. If no confusion is possible the index S will not be used. The weighing function can be defined many different ways. In the most simple form equal functional weighing can be selected by $w(a, b) = 1$. With this weighing $SIM_{s_2}(s_1)$ gives a measure for the portion of information of s_2 present in s_1, without stressing importance of a specific information.
If the set system similarity is used to select a number of promising set systems from a large set of available set systems then this simple weighing function may not be sufficient. In such a case it makes more sense to give the pairs a weight according to the number of set systems that realize it. If an incompatibility pair $a \mid b$ is implemented by many different set systems in the LUT network then apparently it is not so important for a set system to implement this pair, since many alternative implementations exits. If a set system is the only set system that implements an incompatibility pair then this

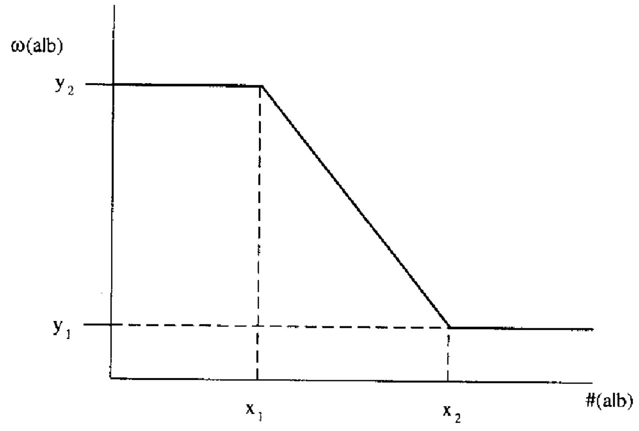


Figure 4.9: Linear availability weighing function

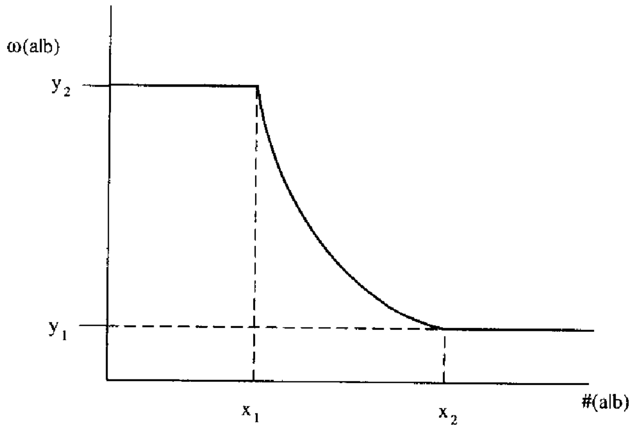


Figure 4.10: Availability weighing function based on $y = 1/x$

set system is very important, because not using this set system means that a long wire is necessary to get the information from lower levels or even primary inputs.
In Figure 4.9 the linear availability weighing function can be found. The X-axis contains the number of occurrences of an incompatibility pair. Two thresholds x_1 and x_2 are introduced. If an incompatibility pair $a \mid b$ has less than x_1 occurrences than this pair is so important, that it gets maximal weighing. On the other hand, if $a \mid b$ has more than x_2 occurrences than there are so many alternatives that it is not worth paying special attention to it. Between x_1 and x_2 the weighing is calculated by a linear function. In general $y_2 = 1$ so that the weighing of the incompatibility pairs is normalized. y_1 can be zero, but also a little higher, since it is always worth to implement an incompatibility relation, even if there are many alternatives. Best practice values for the parameters x_1, x_2, y_1 and y_2 can be determined using experiments. In different places of the algorithm, different parameters can be used. In some cases the difference between good and bad incompatibility pairs is not emphasized enough. In such a case the functions in Figure 4.10 (where a shifted version of the function $y = 1/x$ is used) can be used.

For each incompatibility pair the weight can be calculated a priori, but it is also possible to dynamically adjust it during the computations. For example in a set system selection process, the weights of incompatibility pairs that are implemented by already selected systems can be decreased in order to give set systems that implement slightly less important pairs also a chance in being selected. More complete information about information relationships of set systems and measures of the strength and importance of these relationships can be found in [Jóź97a].

4.3.4 Input representation and don't cares

In Section 4.2.4 it has been shown how the set system representation is derived from the function table. It has been noticed there, that the set system representation depends on the way that the function is expressed by product terms. In this section this relation is further investigated and the question will be asked whether it is necessary to use an input data preprocessor for the LUTSYN algorithm.

A preprocessor can be used to rearrange the input data to an algorithm so that this algorithm can use the input data more efficiently. Preprocessing can be passive (only changing the way the input information is represented) or active (using and changing the input information itself, to make new input data). Many multiple-level synthesis methods presented in the past, perform active preprocessing in the form of a two-level logic minimization before the actual multiple level synthesis is started. More recently, it has been recognized by Łuba e.a. [ŁKJ91] that often it is useful to minimize not the number of product terms but the number of input variables of a Boolean function. This concept will be illustrated using the next example.

Example 4.14 *Input and term minimization*
In Table 4.6 the 4-input, 1-output incompletely specified Boolean function F has been specified. This function can be minimized using the Karnaugh diagram in Figure 4.11. In a product term minimization the product terms a and b will be used and the minimized

$x_1x_2x_3x_4$	y
0000	0
0001	0
0010	0
0011	1
0100	0
0101	0
0110	0
0111	1

$x_1x_2x_3x_4$	y
1000	0
1001	-
1010	-
1011	0
1100	-
1101	1
1110	1
1111	-

Table 4.6: Boolean function F for input and term minimization example

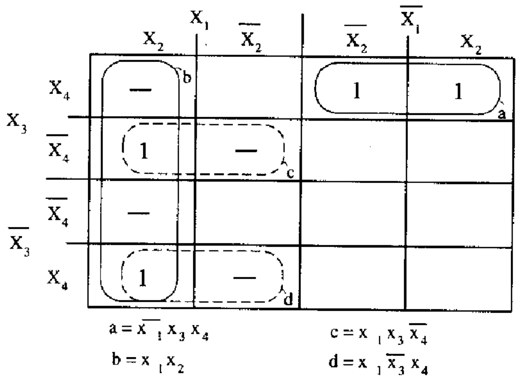


Figure 4.11: Karnaugh diagram for function F

function uses 2 product terms and 4 input variables:

$$F_1' = \overline{x_1}x_3x_4 + x_1x_2$$

Function F can also be minimized using input minimization techniques. In that case, the function uses the product terms a, c and d, which results into a minimized function with 3 product terms and 3 input variables:

$$F_2' = \overline{x_1}x_3x_4 + x_1x_3\overline{x_4} + x_1\overline{x_3}x_4$$

$\square$

In general, there is an input term minimization trade-off. In most cases, input minimization has a larger effect on the total complexity of a two-level function representation. Efficient near-optimal algorithms for the minimal input support problem can be found in [KJ95]. Unfortunately, both input and term minimization used as preprocessing steps will remove a lot of design freedom (in particular all don't cares) from the input specification even before the actual synthesis has started. It is expected that preserving the don't cares (so that they can be used by the actual multiple-level logic synthesis algorithm) will lead to better results. On the other hand, don't cares can be used to make the representation of the input data smaller and therefore more easily manageable.

For the reason mentioned above, the LUTSYN algorithm will use a passive preprocessor and will not sacrifice don't cares to make the representation of the input data more compact. From the example in Section 4.2.4 it follows that it is important how the data in the function table are arranged. After all, the function in Table 4.3 is expressed using 8 product terms. The function table in which these product terms are expanded to min terms specifies the same logic function and hence the same decomposition problem. In that case however, 16 different symbols have to be used instead of 8. Since, a network of LUTs is modeled by set systems on the same symbol set, more memory is necessary to store a network for larger symbol sets. In general, there are a many reasons why the number of symbols should be as low as possible:

1. Each symbol is stored in every set system of the network, therefore each unnecessary symbol is stored a lot of times in memory.

2. Operations on set systems with many symbols are generally slower than the same operations on set systems with fewer symbols. So, if the number of symbols can be removed without changing the meaning of the function and limiting the design freedom, there is no reason not to do this.

3. Unnecessary symbols give information that is not relevant for the problem at hand and can distract the algorithm from its work. In Table 4.3 the symbols e and f are distinguished by input x_2. This means, that in the whole network set systems are used with symbols e and f in it. However, there is no real need to know the difference between these two symbols (since they have the same output vector).Therefore, symbols e and f can better be replaced by one symbol

for the input pattern $(0, -, 0, 0)$. A similar situation can be found by looking at symbols a and b. Input x_4 is able to distinguish between symbol a and symbol b. However, for none of the outputs it is necessary to distinguish between these two symbols. Therefore, it is useful to write the product term associated with symbol a as $(-, 0, 1, -)$. If a symbol is don't care for all output components (like symbol h in the example), it is never necessary to distinguish this symbol from any other symbol and the symbol can be removed from the symbol set.

LUTSYN is primarily devoted to controller like circuits. This type of circuits can be characterized by having relatively few product terms in their original specification compared to all possible min terms and by many don't cares. Therefore, the number of symbols in the set systems can be quite easily kept low.
Based on this discussion the following task for the LUTSYN preprocessor has been defined: preprocess the input data so that as few as possible product terms are used and under this condition maximize the number of don't cares in each of these product terms. The preprocessor should be passive, i.e it does not limit the design freedom. The preprocessor can be implemented using well known operators on cubes, for details see Section 5.4.

4.3.5 Clustering process

During the analysis phase of LUTSYN a clustering process is performed. The clustering algorithm will cluster the outputs of the function to synthesize. It will place those outputs in the same cluster, that have a meaningful subfunction in common. The resulting clustering data is used to guide the bottom-up construction process to find set systems that are useful to implement as common subfunctions. The clustering is:

- **hierarchical**
  This defines a precedence relation on the clusters, i.e. clusters can be build from other clusters and a tree structure is constructed which shows how the clusters are related to each other.

- **with overlapping**
  Clusters may overlap and this has two functions. First of all, the use of overlapping clusters is a way to deal with uncertain decisions. If a function is well related to two clusters, it is hard to decide to which cluster it belongs. Allowing overlapping clusters (and adding this function to both clusters) postpones this decision until more accurate information is presented. A second reason for overlapping clusters is that it is very well possible that a function shares part of its functionality with one cluster of functions, whereas another part of the function is strongly related to another cluster. In such cases with both clusters it will form meaningful subfunctions.

- **non-decisive**
  The clustering is merely used to show the potential in creating useful subfunctions to the bottom-up construction process. Therefore the clustering is not decisive: the construction process can ignore some clusters or to construct some sub-functions

only partially if it turns out that their implementation is too difficult or too expensive.

- **similarity based**
  The clustering algorithm will create clusters based on the set system similarity of the output set systems, i.e. a cluster will only be constructed if the output set systems in the cluster have a high similarity. Please refer to Section 4.3.3 for more information on algorithms for calculating set system similarities.

In Section 5.5 the above clustering concepts will be translated in an algorithm.

4.3.6 Heuristic network construction process

The construction process of LUTSYN for building the LUT network is different from other known network building algorithms. The main distinctive feature of the construction algorithm is that it builds a number of solutions in parallel (i.e. the search algorithm is similar to breadth first search), while most other construction algorithms try to find alternative solutions sequentially (i.e. they use a depth first search based search strategy) or they find just a single solution. The choice for a parallel construction process has a number of advantages:

1. Because the network is build starting from the inputs, it is not certain in advance which of the constructed systems will be actually used for the calculation of a given output. Since, the number of possible set systems is very large, it makes sense to build the network with a number of the most promising set systems in parallel.

2. Because a number of solutions are build in parallel the heuristic algorithms are able to perform a *relative evaluation.* Normally, the evaluation of a set system is based on comparing the set system with the output and subfunction set systems. Relative evaluation means that the set systems can also be compared with each other. Relative evaluation is especially important in the early stage of the construction process, because there the constructed set systems are still very different from the output set systems and hence most set systems are evaluated with a low quality. By comparing these low quality set systems with each other, it is still possible to make a good judgement about their relative quality.

3. A special type of parallel search algorithm called a *double beam first search* has been proved to be efficient for a wide range of synthesis problems: state assignment [Kon90], output decomposition of Boolean functions [Vol91] [JV92], decomposition of sequential machines [JvD92] and test pattern generation for combinational circuits [Bos94]. The construction process in all these problems is very similar to the network building process considered here. Therefore, it is expected that this kind of parallel search algorithms can be applied to work efficiently and effectively for the LUT network construction.

The construction process will be illustrated with an example. For the sake of simplicity two-input one-output LUTs will be considered. In Figure 4.12 the construction process

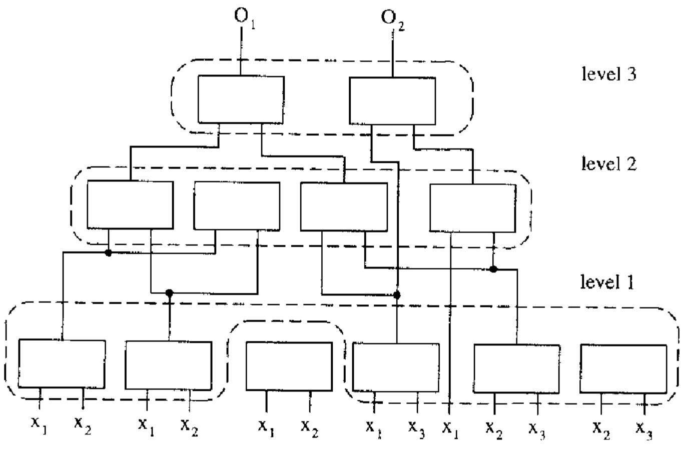


Figure 4.12: Network construction process of LUTSYN

for a three-input two-output Boolean function is given. The network is divided into a number of levels. The network is build level by level. All the set systems in a level are constructed in parallel.

In the first level all LUTs are constructed that are functions of the primary inputs. For each input combination one or more output functions can be generated. The algorithm will construct all non-trivial LUTs that it estimates to be most useful. In Figure 4.12 six LUTs have been constructed, three of them are different functions calculated from the inputs x_1 and x_2, one LUT uses input combination x_1 and x_3 and two LUTs use inputs x_2 and x_3 with each LUT a certain quality measure is associated, that indicates how useful the LUT is. If the number of LUTs in a level is too large, then this quality measure is used to keep only the best LUTs of the level. In Figure 4.12 the limit has been set to five LUTs per level, so one LUT has to be discarded.

The second level of the tree is constructed the same way as the first level: all input combinations are tried and useful functions for each of these combinations are constructed. As the inputs of the level 2 LUTs, the outputs of level 1 LUTs and the primary inputs are considered. It is possible that one or more LUT outputs of the first level cannot be meaningfully combined with another LUT output or a primary input. In that case the output of that LUT is never used and the LUT can eventually be removed from the network (the last LUT of level 1 is such a case). Since, level 2 has only 4 LUTs, no LUT needs to be removed.

Level 3 is then constructed the same way. In this case the LUTs of level 3 implement the output of the circuit. It is also possible that some primary outputs are calculated by more than one LUT.

In general, the LUTs of a certain level N have at least one input that is an output of a LUT of level $N - 1$. The remaining inputs can be freely chosen from any of the previously constructed levels including the primary inputs. The LUTs are also checked for redundancy: all of the inputs of a LUT must be required for computing the LUT's output function. Finally, for each level it is made sure that all incompatibility relations that are part of one of the outputs are implemented by at least one of the LUTs in that level, even if this means violating the limit on the maximal number of LUTs on a certain level.
If each output of the function is implemented at least once, then the construction process is terminated and the network can be cleaned up. All LUTs that are not needed for the calculation of the function outputs can be removed from the network. If an output is realized by more than one LUT, then one of them has to be selected and the other LUTs that implement that output can be removed. The selection process tries to minimize the overall number of LUTs in the LUT network.

The above description serves only to globally explain the construction process and not to explain the evaluation and selection functions. These functions will be discussed in Chapter 5.

Chapter 5

LUTSYN - Algorithms

For the further research of the concepts discussed in the previous chapter, it was necessary to create an experimental software environment that allows to study general decomposition concepts. The description of this environment is the goal of this chapter. This environment, called LUTSYN, will be introduced in Section 5.1. In Section 5.2 an example circuit will be introduced. This circuit is called EB. *Each of the algorithms discussed in this chapter will be illustrated using EB. In Section 5.3 a flow diagram and a global overview of the LUTSYN algorithm can be found. The remaining sections of this chapter discuss different parts of LUTSYN: the preprocessor (Section 5.4), the output clustering algorithm (Section 5.5), the network building algorithm (Section 5.6) and the postprocessor (Section 5.7).*
In the next chapter it will be shown how the LUTSYN environment can be used for the research of general decompositions.

5.1 The LUTSYN environment

For the further research of general decompositions it was necessary to create a software environment in which decomposition experiments can be performed and decomposition concepts (like those presented in the previous chapter) can be tested and evaluated. Such a software tool plays a role of a laboratory setup for the decomposition research. It is important to have such a tool for the following reasons:

1. Many concepts of the proposed decomposition approach are new or have never been used for the decomposition of Boolean functions. Although they are quite well motivated and it is to expect that these concepts will work; this assumption has to be confirmed by experimental research.

2. Researchers and designers do not have much experience with the decomposition structures offered by general decomposition. This kind of practical knowledge is however very important for the better understanding of decomposition problems and development of decomposition methods and tools. For example it gives researchers the feeling which things are typical and practical important (e.g. occur a lot in practical circuits) and which aspects have only theoretical importance.

3. Some decomposition concepts (such as the application of similarity measures and heuristic algorithms) cannot be fully developed without fine-tuning and experimentation; their effects must be studied in practice.

4. Some of the algorithms use parameters to determine some trade-offs. The default values for these parameters should be based on empirical data which can only be obtained from the decomposition of large classes of circuits.

5. The general decomposition of even quite small circuits involves a lot of computations. Doing these computations by hand is very time-consuming and error prone, and restricts the research to very small circuits only. Furthermore, some important aspects of decomposition can be made visible for only larger circuits. Such an experimental software tool accelerates the experimental research "thousands" times, what means that it actually enables such research, especially for large circuits.

Unfortunately, such a tool was not available and therefore needed to be created. Even worse, it was impossible to create this tool by combination and modification of the existing decomposition tools based for examples on the division paradigm (e.g. MIS [BRSW87], SIS [SSL$^+$92] or ASYL [SCS87]) because the general decomposition paradigm is much more general then the division paradigm and the set system based representation was incompatible with the network structure used in other tools. Therefore a new research environment called *LUTSYN* has been created. LUTSYN toolbox has the following properties:

1. It provides an environment in which all components necessary for the research of general decompositions are presents. This includes basic components (such as appropriate internal data structures and input/output routines), but also more complex components, such as the set system module, verification module and the network building modules.

2. It contains a framework synthesis tool. This framework tool is in fact an engine that is able to generate all possible decomposition structures. It can be used for the exhaustive search of the solution space; but even for small circuits this is not very practical. The actual use of the framework follows from the fact that it is that it is easy to add various heuristic search algorithms to it. This way, LUTSYN allows to implement, modify and test new heuristic algorithms in a flexible and fast way.

3. It allows to study the behaviour of the heuristic algorithms. This is extremely important for the development, evaluation and fine-tuning of heuristic algorithms. Basically, the toolbox is able to show the design decisions that have been made and allows the researcher to influence these decisions. It is also possible to show the current network structure. The toolbox has also an "automatic mode", in which the heuristic algorithms take all decisions.

4. It is robust in the sense that a lot of effort has been put in ensuring that the integrity of the network structure is preserved in all cases. For example, an integrity check will be performed on set systems after an operator is applied on them. Similarly, the LUTs and the network structures are checked for their functional correctness.

I	$x_0x_1x_2$	y_0y_1
0	0 - 0	0 0
1	0 0 1	1 1
2	0 1 1	0 1
3	1 0 0	0 0
4	1 0 1	- -
5	1 1 0	1 1
6	1 1 1	1 0

Table 5.1: Boolean function for example EB

The LUTSYN is designed as a research tool. This means that a lot of effort has been made to make a lot of internal data available to the researcher, however the speed and memory usage were not at the top priority.

In the remaining of this chapter the algorithms of the current version of LUTSYN will be presented. This version involves only one prototype heuristic search algorithm equipped with and only one similarity measure. It is a sound base for further experiments and follow-up research. It is not yet a final complete decompositional synthesis tool, but it has been developed to test the framework part of LUTSYN and to check the most important concepts of the proposed decomposition approach.

5.2 The EB circuit

In this section an example circuit will be introduced. The circuit is called *EB*. It is specially constructed to show as many properties of LUTSYN as possible and it will be used in the remaining of this thesis to illustrate the concepts used within LUTSYN.
EB is a circuit having of 3 binary inputs x_0, x_1 and x_2 and 2 binary outputs y_0 and y_1. The goal of the synthesis will be the implementation of EB using two-input, one-output LUTs. The function table of EB can be found in Table 5.1.
The size of the EB example may be considered rather small. That is true; however the size of EB is about the maximal size an example can have without falling into pages of calculations. This confirms again the huge solution space that is inherent to this type of synthesis problems.

The implementation of the function of *EB* specified in Table 5.1 found by LUTSYN contains 4 LUTs (see Figure 5.1). The functions that are implemented by each of the LUTs can be found in Tables 5.2, 5.3, 5.4 and 5.5. Note that block C contains a don't care in the output, this means that this blocks can be an EXOR cell (implementing the don't care as a zero) or an OR cell (by setting the don't care to one).

The following classes of set systems can be found in this example:

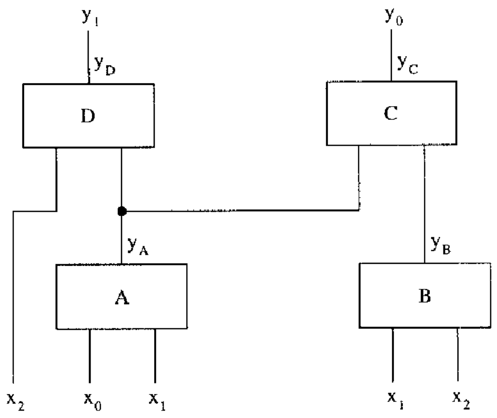


Figure 5.1: Implementation of the EB example

x_0x_1	y_A
0 0	0
0 1	0
1 0	0
1 1	1

Table 5.2: Boolean function for block A

x_1x_2	y_B
0 0	0
0 1	1
1 0	0
1 1	0

Table 5.3: Boolean function for block B

y_Ay_B	y_C
0 0	0
0 1	1
1 0	1
1 1	-

Table 5.4: Boolean function for block C

x_2y_A	y_D
0 0	0
0 1	1
1 0	1
1 1	0

Table 5.5: Boolean function for block D

- Input set systems

$$\phi_{x_0} = \left\{\overline{0,1,2};\ \overline{3,4,5,6}\right\} \tag{5.1}$$

$$\phi_{x_1} = \left\{\overline{0,1,3,4};\ \overline{0,2,5,6}\right\} \tag{5.2}$$

$$\phi_{x_2} = \left\{\overline{0,3,5};\ \overline{1,2,4,6}\right\} \tag{5.3}$$

- Output set systems

$$\phi_{y_0} = \left\{\overline{0,2,3,4};\ \overline{1,4,5,6}\right\} \tag{5.4}$$

$$\phi_{y_1} = \left\{\overline{0,3,4,6};\ \overline{1,2,4,5}\right\} \tag{5.5}$$

- Intermediate set systems

$$\phi_{y_A} = \left\{\overline{0,1,2,3,4};\ \overline{5,6}\right\} \tag{5.6}$$

$$\phi_{y_B} = \left\{\overline{0,2,3,5,6};\ \overline{1,4}\right\} \tag{5.7}$$

$$\phi_{y_C} = \left\{\overline{0,2,3};\ \overline{1,4,5,6}\right\} \tag{5.8}$$

$$\phi_{y_D} = \left\{\overline{0,3,6};\ \overline{1,2,4,5}\right\} \tag{5.9}$$

Due to space considerations the equivalent incompatibility sets will not be listed, they can however easily be obtained from the above set systems as explained in Section 4.3.3.

The set system collections for each block satisfy the conditions for set system collections as defined in Section 4.2.5 , furthermore ϕ_{y_C} and ϕ_{y_D} implement the outputs of EB:

$$\phi_{y_C} \leq \phi_{y_0} \text{ and } \phi_{y_D} \leq \phi_{y_1} \tag{5.10}$$

so these set systems form a correct implementation of EB.

The binary encoding of the set systems is as follows: the block containing symbol 0 is always encoded with binary value 0, the other block is encoding with binary value 1. In the sum-of-product representation this means that the following functions have been realized:

$$y_0 = \overline{x_0} \cdot x_2 + \overline{x_1} \cdot x_2 + x_0 \cdot x_1 \cdot \overline{x_2} \tag{5.11}$$

$$y_1 = x_0 \cdot x_1 + \overline{x_1} \cdot x_2 \tag{5.12}$$

5.3 Overview of the algorithm

The flow diagram of LUTSYN can be found in Figure 5.2. From this figure it is apparent that the LUTSYN algorithm consists of the following 4 major steps.

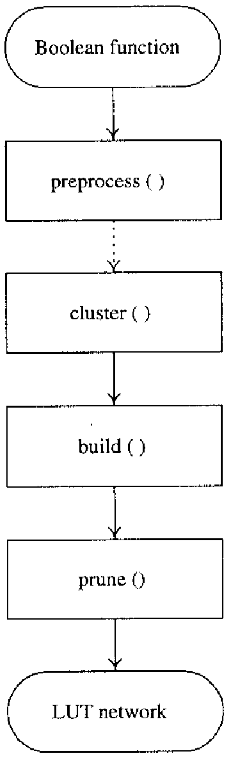


Figure 5.2: Top level flow diagram of LUTSYN

1. *Preprocess()* (Section 5.4)
   The preprocessor has not been integrated with the minimizing part of LUTSYN program, but is a separated program that has to be used prior to the real decomposition algorithm. The goal of the preprocessor is to prepare the input data so that the decomposition algorithm can make the most efficient use of it. The task of the preprocessor is to calculate a minimal product term representation of the input data without sacrificing don't cares.

2. *Cluster()* (Section 5.5)
   The clustering algorithm performs a hierarchical clustering on the output set (as discussed in Section 4.3.5). The result of this algorithm will be a tree of clusters containing output functions. Each cluster represents a common subfunction shared by the outputs of that cluster. The clustering tree will be used to help finding common subfunctions.

3. *Build()* (Section 5.6)
   The build function forms the kernel of the LUTSYN algorithm. This function is responsible for constructing the LUT network. Heuristic decision algorithms are used to guide the construction process. The build function continues constructing new levels of the network until for each output function at least one implementation has been found.

4. *Prune()* (Section 5.7)
   The prune function is necessary if one or more output functions have more than one realization. In such a case, for each output exactly one implementation has to be chosen, such that the network uses as few as possible LUTs in total. Selecting these outputs is the task of the prune algorithm.
   The result of the prune algorithm is the final LUT network that implements the circuit. This network will be written to an output file.

Each of the steps listed above will be discussed in more detail in the next few sections.

5.4 Preprocess() : Preparation of the input data

5.4.1 Introduction

The concept for the preprocessor has been introduced in Section 4.3.4. The preprocessor is necessary for two main reasons:

1. In order to create input and output set systems from a function table, this table must contain the ON-cover and the OFF-cover. Many benchmarks are however specified using the ON-cover and the DC-cover and the OFF-cover is implicitly specified as the complement of these two covers. In such a case, the preprocessor needs to calculate the OFF-cover explicitly.

2. Each product term in the function description becomes a symbol in the set systems. Therefore, it is important that the preprocessor minimizes the number of

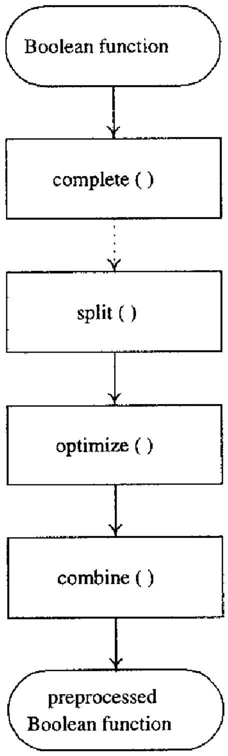


Figure 5.3: Flow diagram of the LUTSYN preprocessor

product terms in the function specification without exploiting don't cares, so that the decomposition algorithm uses set systems on as small as possible symbol sets.

The LUTSYN preprocessor will not sacrifice don't cares to make the function smaller, so it does not change the function itself, but only changes the way the function specification is organized. This way the complete don't care freedom is preserved and can be used by the decomposition algorithm to find better decompositions.

The flow diagram of the preprocessor function can be found in Figure 5.3. The preprocessor consists of four tasks, each of these tasks will be described in more detail in the next subsections.

5.4.2 Complete(): Calculate missing cover

The first step of the preprocessor is an algorithm that reads a function description from an input file into the internal data structure. The function is specified using the Espresso file format (the specification of the Espresso file format can be found in [Yan91]). The *complete()* function has two subtasks:

1. If a function specification does not contain the ON-cover and the OFF-cover then the missing cover will be calculated. This is necessary because the set system associated with an output bit, basically distinguishes the input patterns for which the output bit is zero, from those for which the output bit is one.
2. A consistency check will be performed on the function specification: if two product terms have overlapping input parts, then their output parts should be equal. Furthermore, it is checked if the whole input space is covered by the product terms. If inconsistencies have been detected, the program will abort with an error message.

The *complete()* function is implemented as a separate program. The algorithm is based on the sharp operator [Mil66]. The sharp operator (denoted as $A\#B$) is the difference (or subtraction) operator and calculates all those product terms that are part of cover A but that are not contained in cover B. The complement of a cover can be obtained by subtracting that cover from the universal cover (i.e. the cover containing all product terms). Here, a special version of the sharp operator called the *restricted sharp operator* [vW95][vH94] is used for complementing. The logical result of the restricted sharp is of course the same as the ordinary sharp operator defined in [Mil66]. The difference is that the restricted sharp does not contain all the prime implicants but only those prime implicants necessary for the covering. The sharp operator will be illustrated using the following example, taken from [vH94, Section 3.3].

Example 5.1 *Sharp operator*
In this example, a two-input, two-output cover will be used. A cover is specified as a set of product terms. A product term consists of an input part and an output part. Let

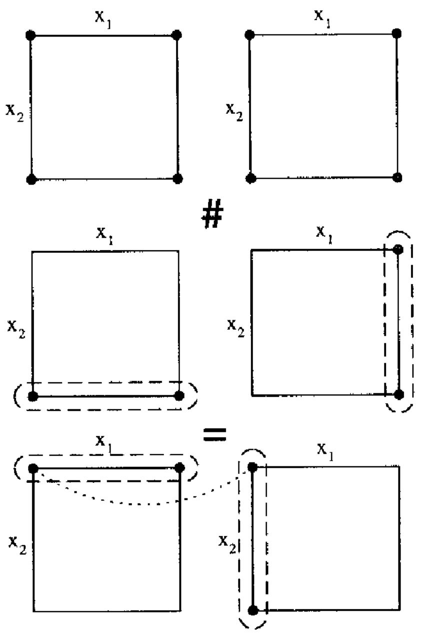


Figure 5.4: Calculation of $\{-- \mid 11\}\#\{-1 \mid 10;\ 1- \mid 01\}$

x_1, x_2 be the names of the input bits and let y_1, y_2 be the names of the output bits. The cover X defined as

$$X = \{-1 \mid 10;\ 1- \mid 01\}$$

represents the following functions:

$$y_1 = x_2 \text{ and } y_2 = x_1$$

Given the universal cover U:

$$U = \{-- \mid 11\}$$

Using the definition of the sharp operator from [Mil66] the following sharp difference can be calculated:

$$U\#X = \{0- \mid 01;\ -0 \mid 10;\ 00 \mid 11\}$$

This sharp operator has been visualized in Figure 5.4. A property of this sharp operator is that it calculates all prime implicants, even those that are not necessary for the cover.

In fact the last product term 00 | 11 is not necessary for the cover (which can be seen very easily from Figure 5.4. The restricted sharp [vW95] does not calculate these unnecessary prime implicants and finds the following solution:

$$U\#_{restricted}X = \{0- \mid 01;\ -0 \mid 10\}$$

□

In general, the restricted sharp operation describes the complemented cover using few product terms compared to the original sharp operator. The reader is referred to [vH94] for more details on the definition and implementation of the restricted sharp function.

5.4.3 Split() : Split into output covers

The output of *complete()*, is a description of the function using the ON-cover and the OFF-cover. This description will be optimized by *optimize()* (described in the next section). As discussed, the optimization will be passive, it will not use don't cares to minimize the number of product terms. This means that product terms can only be optimized using product terms that have the same output vector. Therefore, the function cover will be partitioned by *split()* in a number of subcovers, each subcover will contain product terms that have the same output vector. Each of these subcovers will then be minimized by *optimize()*.

5.4.4 Optimize() : Optimize each output cover

The *optimize()* block is quite complex, therefore a flow diagram of this block can be found in Figure 5.5. The input of *optimize()* is a cover composed of product terms that have the same output vector. The optimize algorithm performs iteratively three independent optimizations. If one of these optimizations results into a change of the cover, then a next iteration is started. If none of the optimizations changed the cover, then the cover is said to be optimized and the new cover is returned. The first optimization *merge_pterm()* is the most classical product term optimization, this function checks if two product terms form the cover can be merged to a new product term. For example, the product terms $a \cdot b$ and $a \cdot \overline{b}$ can be combined to a new product term a.
Redundant_pterm() is the second optimization, it tries to find product terms in the cover that are redundant, i.e. if such a redundant product term is removed from the cover, then the cover covers the same min terms. For example, the cover consisting of the product terms $a \cdot \overline{c}$, $\overline{b} \cdot c$ and $a \cdot \overline{b}$ is redundant, by removing the product term $a \cdot \overline{b}$ exactly the same min terms are covered.
The last optimization is performed by *sparse_pterm()*. The goal of this optimization is to remove literals from product terms. The classical example of the optimization performed by *sparse_pterm()* is $a + \overline{a} \cdot b = a + b$. *Sparse_pterm()* determines for every literal in the cover, if it can be removed without changing the min term covering. In principal, *redundant_pterm()* could also be performed as the by product of *sparse_pterm()*, however, *sparse_pterm()* is a rather expensive operation whereas *redundant_pterm()* is reasonable fast. This way, the optimizations obtained by *redundant_pterm()* help to reduce the

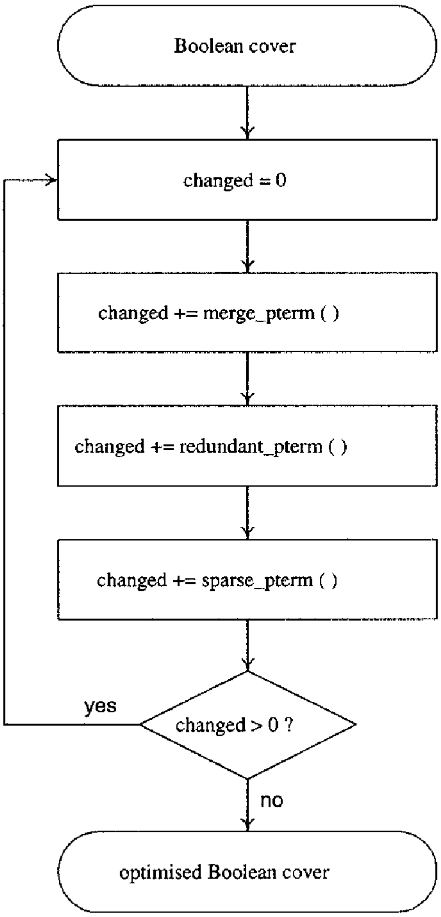

Boolean cover
changed = 0
changed += merge_pterm ( )
changed += redundant_pterm ( )
changed += sparse_pterm ( )
changed > 0 ?
yes
no
optimised Boolean cover

Figure 5.5: Optimization flow used in the LUTSYN preprocessor

computation time of *sparse_pterm()*.
The first two optimizations are useful because they decrease the number of product terms in the function table and hence the number of symbols in the set systems. The third optimization (*sparse_pterm()*) does not decrease the number of product terms, but decreases the number of literals. This is also of some importance, because such a removable literal suggests to the decomposition algorithm that the corresponding bit can be used to distinguish the product terms from some other product terms, whereas this is not the case.

5.4.5 Combine() : Combine output covers

The function of *combine()* is fairly simple, it collects the minimized subcovers processed by *optimize()*, makes a new cover from it and writes this cover to an output file. This output file contains then the final preprocessed Boolean function.

5.4.6 Example

In this section, the preprocessor will be illustrated using an example. Given the 4-input, 1-output function F defined in Table 5.6. This function contains the ON, OFF and DC-covers, so the *complete()* function only performs a consistency check. The *split()* algorithm then strips all product terms for which *all* output bits are don't care. The *split()* function results in two covers: subcover 1 (see Table 5.7) that contains all the product terms for which the output is 0, and subcover 2 (Table 5.9) that contains the product terms for which the output is 1. For each of these covers the *optimize()* algorithm is executed.
For subcover 1, first *merge_pterm()* is called. *Merge_pterm()* will combine the third and the fourth product term to a single product term $x_2 \cdot \overline{x_3} \cdot \overline{x_4}$. Then *redundant_pterm()* is called, this function removes the second product term, since this product term is entirely covered by the first and fifth product term. After these steps the subcover in Table 5.8 remains. Calling sparse_pterm does not change the subcover, nor does a new invocation of *merge_pterm()* and *redundant_pterm()*. Therefore, Table 5.8 contains the final optimized subcover 1.
Subcover 2 is optimized the same way. Here, only the *sparse_pterm()* algorithm contributes to a change of the cover as can be seen from the Karnaugh diagram in Figure 5.6. The optimized subcover 2 can be found in Table 5.10.
The optimized function F is obtained by combining the optimized subfunction and writing it to an output file. These two tasks are performed by *combine()*.

5.4.7 Implementation

The implementation of the LUTSYN preprocessor is based on a two-level logic synthesis toolbox. This toolbox is described in [vH94][dB95].
At this moment the preprocessor does not use any heuristic algorithms in optimizing the function cover. This is not so serious as it sounds, because the optimizations are performed on small subcovers obtained as a result of *split()* and furthermore, the preprocessor algorithm can be stopped anytime and the result found so far is a valid (but maybe not fully optimized) input for LUTSYN. It is a deliberate choice not to spent too

$x_1x_2x_3x_4$	y_1
0 0 0 -	0
- 0 0 1	0
0 1 0 0	0
1 1 0 0	0
1 0 - 1	0
- - 1 0	1
- 1 1 1	1
- 1 0 1	-
0 1 1 1	-
1 0 0 0	-

Table 5.6: Boolean function F for preprocessor

$x_1x_2x_3x_4$	y_1
0 0 0 -	0
- 0 0 1	0
0 1 0 0	0
1 1 0 0	0
1 0 - 1	0

Table 5.7: Subcover 1 for function F

$x_1x_2x_3x_4$	y_1
0 0 0 -	0
- 1 0 0	0
1 0 - 1	0

Table 5.8: Optimized subcover 1 for function F

$x_1x_2x_3x_4$	y_1
- - 1 0	1
- 1 1 1	1

Table 5.9: Subcover 2 for function F

$x_1x_2x_3x_4$	y_1
- - 1 0	1
- 1 1 -	1

Table 5.10: Optimized subcover 2 for function F

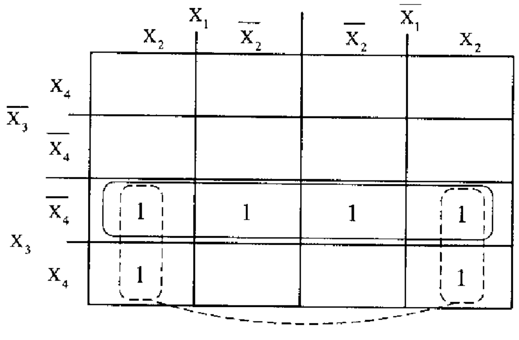


Figure 5.6: Karnaugh diagram of subcover 2

much work on the preprocessor algorithm at the moment. The main reason for this is that currently it is unknown what the influence of the preprocessor will be on the amount of resources that LUTSYN requires. Before a good preprocessor can be constructed, it is necessary to have an answer to the following two questions:

1. Does the preprocessor cause a significant decrease of the computation time used by LUTSYN? If the answer is yes, then preprocessing is useful, otherwise preprocessing can be considered a waste of time.

2. Has the preprocessor an influence on the quality of the LUT networks constructed by LUTSYN? If so, it should be determined which preprocessor steps have a positive influence on the quality.

Both these questions, can only be answered after the decomposition algorithm is completely finished and throughout experiments can be performed with it. Therefore, a near-optimal preprocessor is of lesser importance at the moment.

5.4.8 Preprocessing EB

Figure 5.7 contains a description of the EB circuit in Espresso format [Yan91]. The preprocessor accepts all cover combinations of the Espresso format. In this case the circuit is specified using the ON-cover, OFF-cover and DC-cover. Since, all covers are present the *complete()* algorithm performs only a consistency check. The don't care cover is then dropped by *split()* and the function is split in four subcovers based on the output vector:

$$subcover_{00} = \{0-0;\ 100\}$$

$$subcover_{01} = \{011\}$$

```
.i 3
.o 2
.type fdr
.p 7

0-0 00
001 11
011 01
100 00
101 --
110 11
111 10

.e
```

Figure 5.7: Preprocessor input file eb.pla

```
.i 3
.o 2
.type fr
.p 6

0-0 00
-00 00
111 10
110 11
001 11
011 01

.end
```

Figure 5.8: Preprocessor output file eb.pre

$$subcover_{10} = \{111\}$$

$$subcover_{11} = \{001;\ 110\}$$

Each of these covers is then minimized by *optimize()*. Only $subcover_{00}$ can be optimized, the other covers were already optimal. The output of *optimize()* is therefore the following lists of optimized covers:

$$subcover'_{00} = \{0-0;\ -00\}$$

$$subcover'_{01} = \{011\}$$

$$subcover'_{10} = \{111\}$$

$$subcover'_{11} = \{001;\ 110\}$$

Based on these covers *combine()* creates an output file in Espresso format that contains the preprocessed EB circuit. This output file can be found in Figure 5.8. This file will be used as the input for the cluster algorithm discussed in the next section.

5.5 Cluster() : Cluster algorithm

5.5.1 The algorithm

The concept of clustering has been discussed in Section 4.3.5. The goal of the *cluster()* algorithm is to determine for which outputs of the circuit it is useful to create common

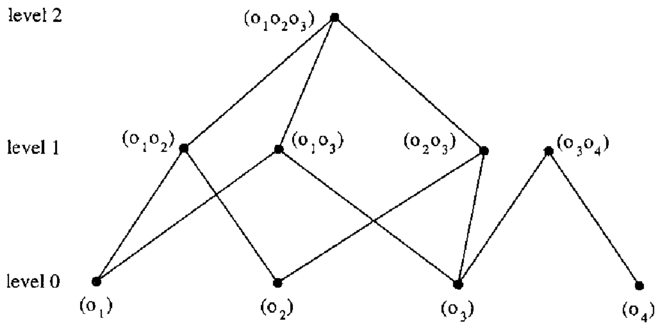


Figure 5.9: LUTSYN cluster algorithm

subfunctions (because these outputs will share much logic). The *build()* algorithm will only try to construct subfunctions for the clusters obtained by *cluster()* and will not create other subfunctions in order to prevent the creation of long wires for small subfunctions. A cluster of some output set systems is only constructed if the number of common incompatibility pairs is large compared to the number of incompatibility pairs of the output set system in the cluster with the largest number of incompatibility pairs (i.e. the cluster function must have a significant impact on all the set systems). The following formula is used to determine if a cluster C will be constructed:

$$\left\| \left\{ v \mid w \;\middle|\; \underset{\phi \in C}{\forall} v \mid_\phi w \right\} \right\| \geq CLUFAC \cdot max_{\phi \in C} \|\phi\| \tag{5.13}$$

where $\|S\|$ denotes the number of elements in set S, $\|\phi\|$ denotes the number of incompatibility relations in set system ϕ and $v \mid_\phi w$ denotes that v and w are incompatible with respect to set system ϕ. The left hand of this formula expresses the number of incompatibility pairs that are common to all output set systems in the cluster C. This number is compared to the maximum number of incompatibilities that the set systems in cluster C have. The user definable constant $CLUFAC$ is used as a threshold factor. Currently is has been set to 33%, but experiments have to confirm if this is a good value.

The cluster algorithm is illustrated in Figure 5.9. Clusters are constructed level by level and form a cluster tree. In Figure 5.9 the cluster tree of a 4 output Boolean function has been constructed. Level 0 of the cluster tree contains singleton clusters; there is a cluster for each output of the circuit.

In the first level all primary outputs are clustered in pairs. In theory using 4 level 0 clusters, 6 different combinations can be constructed. In Figure 5.9 only 4 clusters have been accepted (i.e. two clusters did not satisfy (5.13) and were deleted).

The second level of the cluster tree is then constructed based on combinations of level 1 clusters only.

At the moment the clusters on each level use the same $CLUFAC$ value. As an extension it is possible to let $CLUFAC$ be a function of the level of the cluster tree. For, at higher levels of the cluster tree the number of functions for which the cluster is a subfunction becomes larger and hence the amount of saved active area is larger. Therefore, it makes sense to use a lower $CLUFAC$ if the number of outputs in the cluster increases.

Each cluster is described using the set of common incompatibility relations. Construction of these sets will be performed by the *build()* algorithm described in the next section. There is however a difference between primary outputs and the cluster incompatibility sets. *Build()* is required to implement the primary output set systems, but it does not necessary have to construct logic for the entire cluster incompatibility set. If it is easier for the construction process, *build()* can decide to implement only a subset of the incompatibility pairs as the subfunction. This will be discussed in the next section.

The cluster algorithm has been implemented and is quite efficient, large Boolean functions (more than 15 outputs) can be clustered within few minutes.

5.5.2 Clustering EB

Clustering for the EB example is quite easy, since there are only two output set systems. When expressing the output set systems of EB as incompatibility sets, the following is obtained:

$$IC(\phi_{y_0}) = \{(0,1);\ (0,5);\ (0,6);\ (1,2);\ (1,3);\ (2,5);\ (2,6);\ (3,5);\ (3,6)\}$$

$$IC(\phi_{y_1}) = \{(0,1);\ (0,2);\ (0,5);\ (1,3);\ (1,6);\ (2,3);\ (2,6);\ (3,5);\ (5,6)\}$$

A cluster C can be obtained by intersecting the incompatibility sets of these two outputs; it contains 5 incompatibility pairs.

$$IC(C) = \{(0,1);\ (0,5);\ (1,3);\ (2,6);\ (3,5)\}$$

Using a $CLUFAC$ of 0.33 C is an acceptable cluster, which can be very easily verified using Equation (5.13). The set system ϕ_C that corresponds to $IC(C)$ is defined as follows (ϕ_C is obtained using the algorithm described in Section 4.3.3.1):

$$\phi_C = \left\{\overline{0,2,3,4};\ \overline{0,3,4,6};\ \overline{1,2,4,5};\ \overline{1,4,5,6}\right\}$$

This is however a 4 block set system, thus this set system cannot be implemented by a LUT (for EB only single output LUTs are used and thus subfunctions can only have two blocks). The *build()* algorithm described in the next section will use the information of $IC(C)$ to generate a subfunction ϕ'_C that implements only part of C, but does this in a two block set system:

$$\phi'_C = \left\{\overline{0,1,2,3,4};\ \overline{5,6}\right\}$$

ϕ'_C implements 3 out of 5 incompatibility pairs from $IC(C)$: $(0,5)$, $(2,6)$ and $(3,5)$.

In the next section the kernel of the LUTSYN algorithm will be discussed: the LUT network construction algorithm *build()*.

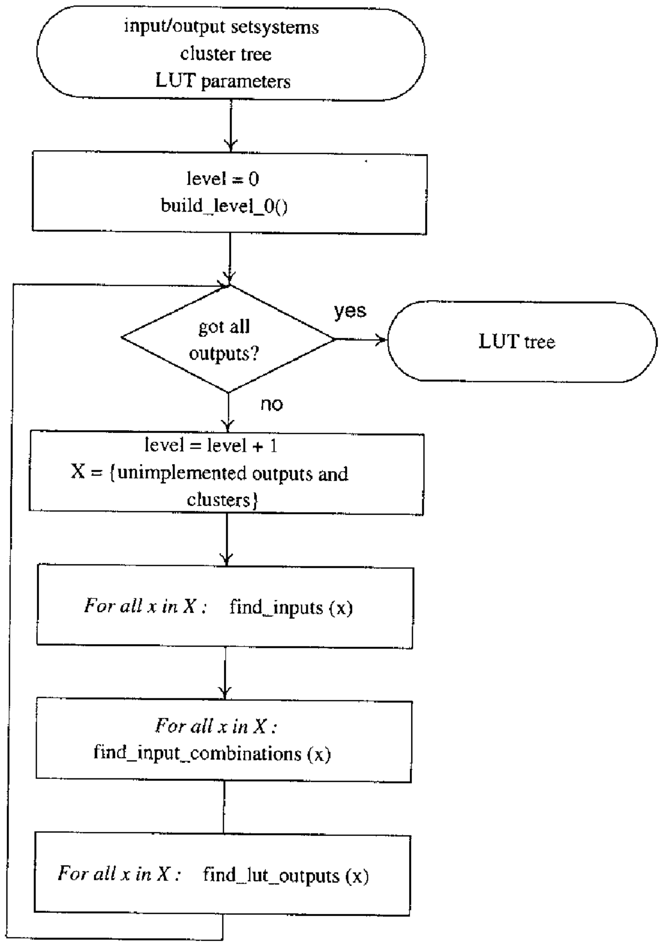

input/output setsystems
cluster tree
LUT parameters
level = 0
build_level_0()
got all outputs?
yes
LUT tree
no
level = level + 1
X = {unimplemented outputs and clusters}
For all x in X : find_inputs (x)
For all x in X : find_input_combinations (x)
For all x in X : find_lut_outputs (x)

Figure 5.10: Flow diagram of the *build()* algorithm

5.6 Build() : Network construction algorithm

5.6.1 Introduction

Build() forms the core of the synthesis algorithm. It is responsible for building the LUT network for a given Boolean function. An overview of the *build()* algorithm can be found in Figure 5.10.
The input of the algorithm consists of three types of information: the input and output set systems, the cluster tree and the LUT parameters. The input set systems represent the primary input of the LUT network, the output set systems are the functions that must be calculated by the LUT network. The cluster tree is used to guide the construction of subfunctions and the LUT parameters indicate the number of inputs and outputs of a considered LUT.
The function *build_level_0()* makes input LUTs (LUTs with no inputs and one output that corresponds to the input set system) for each of the primary input set systems. It is of course not necessary to implement these input LUTs in the final physical realization. These input LUTs are only used to simplify the algorithms: by using inputs LUTs the construction process does not need to distinguish between primary inputs and the outputs of the other LUTs. The input LUTs will be removed before the actual network implementation.
The main part of *build()* consists of a loop. In each iteration of this loop a level of the LUT tree is constructed. The loop process stops as soon as for each of the output set systems an implementation has been found. This way the algorithm prefers implementations with fewer levels above implementations that have more levels but fewer LUTs. The result of the build process is a LUT tree. This tree may contain too many LUTs, for example the tree may contain dead ends or it may contain more than one implementation for a certain function. Therefore the tree needs to be cleaned up. This is the task of the *prune()* algorithm described in Section 5.7. The result of *prune()* will be the final LUT network that implements the circuit.

The main loop of the build algorithm consists of three heuristic algorithms: *find_inputs()*, *find_input_combinations()* and *find_lut_outputs()*. Each of these algorithms operates on a single output or cluster set system that has not been implemented at one of the previous levels. This way each function is implemented using as few as possible levels. In rare cases it is possible that implementations of a set system will be found at higher levels in the tree as the by-product of the construction of some other set system. Such implementations with more than the minimal number of levels are permitted, but they will never actively be searched for.
Suppose the construction process is building level n of the LUT tree and set system o is an output or cluster set system that has not yet been implemented. The algorithm *find_inputs(o)* will find a list of inputs that are useful as the inputs for a level n LUT, such that this LUT is useful for the implementation of set system o. As inputs of this LUT the outputs of LUTs of one of the previous levels and the primary inputs are considered. If the list of inputs becomes too large (determined by a user definable threshold value) then a heuristic selection process will be performed that will select only the most promising input combinations. Details of the *find_inputs(o)* algorithm can be found in Section 5.6.3.

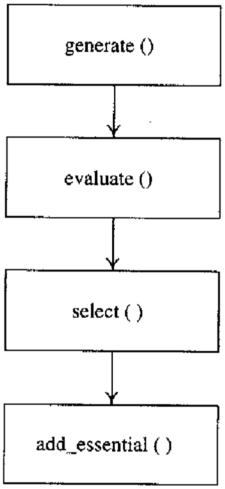


Figure 5.11: Framework of the heuristic decision algorithm

The algorithm *find_input_combinations(o)* uses the list of inputs created by *find_inputs(o)* to make useful input combinations for a LUT. An input combination is useful if it is non redundant and if it is likely to contribute to the implementation of function o. The exact algorithm can be found in Section 5.6.4. Of course, a heuristic selection algorithm will be used to keep the set of input combinations within practical limits.
Each input combination obtained from *find_input_combinations(o)* is transformed into a LUT. Calculating the output of this LUT is the task of the function *find_lut_outputs(o)*. This function calculates for each of the LUT one or more output functions that are likely to be a part of the implementation of function o. If one or more of the outputs implements function o then the construction of function o is completed and the function is marked done. Otherwise, the algorithm starts with building level $n+1$. *find_input_combinations(o)* is described in more detail in Section 5.6.5.

The heuristic algorithms *find_inputs()*, *find_input_combinations()* and *find_lut_outputs()* are all based on the same basic heuristic algorithm. This basic decision algorithm will be discussed in the next section. Following this section each of the three steps in *build()* will be discussed in detail. Each step will be illustrated using the EB example.

5.6.2 The basic heuristic decision algorithm

The three algorithms *find_inputs()*, *find_input_combinations()* and *find_lut_outputs()* use the same heuristic framework. This means that these algorithms all have the same structure and use similar functions for the evaluation and selection. This way it is guaranteed that

the independent steps *find_inputs()*, *find_input_combinations()* and *find_lut_outputs()* work in the same direction. A flow diagram for the heuristic decision framework can be found in Figure 5.11. The framework consists of 4 blocks. The framework operates on items. What an item exactly is depends on the function in which the heuristic algorithm is used: in *find_inputs()* an item is a signal that is a valid input for a LUT, in *find_input_combinations()* an item is a non redundant combination of inputs and in *find_lut_outputs()* an item is a set system that can be calculated from the LUT inputs and that does not exceed the maximal number of blocks.
The first block in Figure 5.11 is *generate()*. *Generate()* simply generates a list of all feasible items. This list is then passed to *evaluate()* that weighs each of items. *Evaluate()* is based on the similarity between the item and the set system that is being constructed and uses a decreasing incompatibility weighing function as described in Section 4.3.3.2.
Select() forms the core of the decision algorithm. It determines which items will be selected and which items will be removed. *Select()* uses two user definable threshold values $SELECT\_MAX$ (an integer) and $SELECT\_QUALITY$ (a floating point number between 0.0 and 1.0) to determine which items should be selected. $SELECT\_MAX$ indicates the maximal number of items that can be selected; this is an absolute limit. A relative limit is specified by $SELECT\_QUALITY$: the weight of an item should be at least $W_{best} * SELECT\_QUALITY$ in order to be selected. In this formula, W_{best} is the weight of the item with the highest weight. The goal of $SELECT\_MAX$ is to prevent a combinatorial explosion. $SELECT\_QUALITY$ is used to prevent the selection of items for which it can be safely assumed that they are not useful (because their quality is so much lower than the best item). The selection algorithm itself is then straightforward: the best $SELECT\_QUALITY$ items are selected provided that they satisfy the relative limit imposed by $SELECT\_QUALITY$.
Two variants of the *select()* algorithm exist: a static version that calculates the item's weights once before the selection process starts and a dynamic version that recalculates the weights after a single item has been selected. The reason for the introduction of the dynamic *select()* algorithm is that after selecting an item the weights of the not selected items change. Since, if an item is selected then selecting another item that implements almost the same incompatibility relations becomes less interesting and selecting an item that implements incompatibility relations that are not implemented by any of the selected items become more important. The dynamic *select()* algorithm is however much slower than the static version, because all weights must be recalculated after selecting an item and the list of not selected items must be resorted. If using the dynamic *select()* algorithm costs too much time, then it is of course possible to recalculate after selecting for example ten items, instead of recalculating after each selection.
The function *add_essential()* forms the last block of the heuristic decision framework. The goal of *add_essential()* is to add items that are not selected by *select()* but that are necessary for the implementation of any function. In LUTSYN it is made sure that each incompatibility relation of an output or a cluster set system is at least implemented by one of the selected items. If for some reason the *select()* algorithm misses this incompatibility relation then *add_essential()* will add a set system that implements it. This way, each level of the LUT tree can potentially implement the output or cluster set system. *Add_essential()* should not be necessary in normal cases. Its application by the construction algorithm generally indicates that the limits $SELECT\_MAX$ and $SELECT\_QUALITY$ are set to

high, so that good and necessary items are removed.

In LUTSYN the algorithm above is implemented with two options. It is possible to use this heuristic algorithm interactively. This way, the researcher can completely control the design decisions taken by the algorithm. He can intervene if it finds that the wrong or too many items have been selected and change the search parameters. This is very important for the analysis of existing heuristics and for developing new ones. It also helps in finding good empirical values for the search parameters. There is also an automatic option, in which the search parameters $SELECT\_MAX$ and $SELECT\_QUALITY$ are predefined and the search algorithms make all decisions fully automatically, without interacting with the user.

5.6.3 Find_inputs(o)

5.6.3.1 The algorithm

In this section the function *find_inputs(o)* will be discussed, where o is the output for which an implementation is to be found. Unless stated explicitly, o can be both a primary output or a subfunction obtained by the cluster algorithm discussed in Section 5.5. *Find_inputs(o)* will only be executed for outputs o which are not implemented on one of the previous levels of the LUT tree. It will be assumed that the algorithm is currently building level n ($n > 0$) of the LUT tree.

The task of *find_inputs(o)* is to make a list of inputs that can be used as the inputs of the level n LUTs. Generally, *find_inputs(o)* should not be necessary or at least should not be very restrictive. This is because *find_inputs(o)* selects as the inputs the outputs of LUTs of lower levels. However, the LUT outputs of lower levels are created because they were useful for the construction of the function o and it may be safely assumed that good outputs of a LUT are still good if they are used as inputs on the next level. There are three cases in which the operation performed by *find_inputs(o)* is really necessary:

1. Symmetric functions (like arithmetic operations) tend to produce a lot of LUTs with similarity in the same order of magnitude; in such a case an extra selection of inputs may be necessary to prevent a combinatorial explosion.

2. If the cluster algorithm has determined many useful subfunctions for function o selection may be necessary to prevent the list of inputs to become too large.

3. When studying the behaviour of *find_input_combinations(o)* or *find_lut_outputs(o)* it may be necessary to make the algorithm less restrictive in these functions. This can result in the generation of many LUTs such that the number of potential inputs on the subsequent level of the LUT tree can become too large.

The *find_inputs(o)* algorithm consists of the four basic steps of the heuristic framework described in the previous section. The flow diagram can be found in Figure 5.12. *Generate_inputs(o)* just makes a list of all permitted input signals. For inclusion on that list of inputs it considers the following signals:

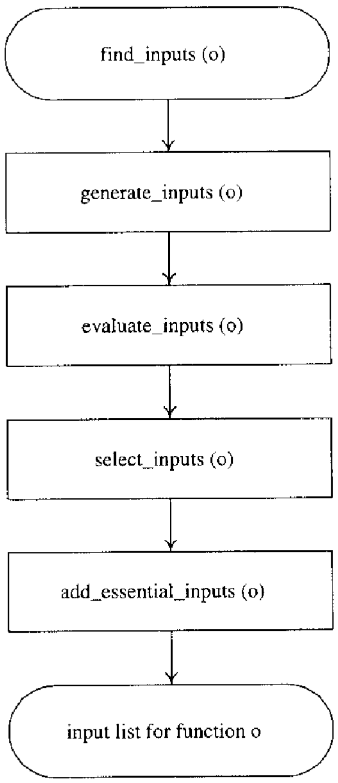


Figure 5.12: The *find_inputs(o)* algorithm

1. All primary inputs.
2. The outputs of level $m(m < n)$ LUTs if these LUTs are constructed for function o.
3. The outputs of level $m(m < n)$ LUTs if these LUTs are constructed for one of the subfunctions found by the cluster algorithm and function o is part of that particular cluster.

The consequence of this is that *find_inputs(o)* can only create subfunctions that are found by the cluster algorithm. *find_inputs(o)* will use LUTs that are constructed for a cluster if the output o is a part of the cluster even if that LUT does not implement the cluster yet. The goal of this approach is to allow the use of a subfunction that implements the cluster only partially. This is done because a cluster is only a tool in finding good subfunctions, the exact subfunction might be too difficult or too expensive to create, while its subfunction can be constructed much cheaper.

The list of all potential inputs constructed by *generate_inputs(o)* is then processed by *evaluate_inputs(o)*. *Evaluate_inputs(o)* simply calculates the set system similarity between the set system representation of the potential input and ϕ_o and sorts the list of inputs with respect to this value. The incompatibility pairs are weighed based on the number of input set systems that implement them. As the incompatibility weighing function, the function $weight = 1/noc$ is used, where noc is the number of occurrences of a certain incompatibility pair and $weight$ is the weighed value. This way set systems that implement many incompatibility pairs (and thus resemble ϕ_o) and set systems that implement important incompatibility pairs are preferred by the algorithm.
Then *select_inputs(o)* statically selects inputs using the heuristic algorithm described in section 5.6.2. It uses the values $SELECT\_INPUT\_QUALITY$ (which defaults to 0.33) and $SELECT\_INPUT\_MAX$ (which has a default value of 50) to limit the selection process.
Finally, *add_essential_inputs(o)* checks if all incompatibility pairs of set systems ϕ_o are implemented by at least one of the selected input set systems. If this is not the case then it will be impossible to completely construct ϕ_o at this level of the search tree. Therefore, *add_essential_inputs(o)* adds the best not selected input set system that implements the missing incompatibility pair. Generally, the adding of essential inputs indicates that the search process is deleting important set systems and the search parameters need to be adjusted.

5.6.3.2 The EB example

In this section the *find_inputs()* algorithm will be illustrated using the EB circuit. This circuit has tree primary inputs (x_0, x_1 and x_2) and two primary outputs (y_0 and y_1). See Section 5.2 for the details. Also a cluster C has been identified as a potential common subfunction for y_0 and y_1 (see Section 5.5.2). It will be assumed that the default values $SELECT\_INPUT\_QUALITY = 0.33$ and $SELECT\_INPUT\_MAX = 50$ will be used for the search. Please note that in this example, the function from Section 5.2 will be used and not the preprocessed format described in Section 5.4.6.

1	1.00					
2	1.00	1.00				
3	1.00	0.50	0.33			
4	0.50	1.00	0.50	1.00		
5	1.00	0.33	0.50	1.00	0.50	
6	0.50	0.50	1.00	0.50	1.00	1.00
$a \mid b$	0	1	2	3	4	5

Table 5.11: Weight of the incompatibility pairs in *find_inputs()*

The selection of the level 1 inputs is done for each output function and subfunction separately. For output y_0 the following selection decisions are made. Since at level 1 there is only a choice from the primary inputs *generate_inputs(y_0)* will find the following lists of input set systems: ϕ_{x_0}, ϕ_{x_1} and ϕ_{x_2}.
Evaluate_inputs(y_0) will calculate the similarity for each of these set systems with ϕ_o using the incompatibility weighing function $weight = 1/noc$. For example the incompatibility pair $0 \mid 1$ is implemented by ϕ_{x_2} but not by set systems ϕ_{x_0} and ϕ_{x_1}, so its number of occurrences is 1 and it weight will be 1/1=1. The pair $2 \mid 3$ is implemented by all three primary inputs, so its number of occurrences is 3 and its weight will be 0.33 (using two decimals). Calculating the weight for all incompatibility pairs results in Table 5.11. The similarity between ϕ_{y_0} and the input set systems can then easily be calculated using Equation (4.36):

$$SIM_{\phi_{y_0}}(\phi_{x_0}) = 0.50$$

$$SIM_{\phi_{y_0}}(\phi_{x_1}) = 0.36$$

$$SIM_{\phi_{y_0}}(\phi_{x_2}) = 0.43$$

This results in the following sorted list of potential inputs: ϕ_{x_0}, ϕ_{x_2} and ϕ_{x_1}. *Select_inputs(y_0)* will select all these inputs because the list is shorter than $SELECT\_INPUT\_MAX$ element and the similarity of all the set systems is within the 33% percent limit imposed by $SELECT\_INPUT\_QUALITY$.
In this case there exist no incompatibility pairs from ϕ_{y_0} that are not implemented by at least one of the selected input set systems. So the function *add_essential_inputs(y_0)* does not add new set systems.

For the second output and the subfunction cluster a similar calculation can be held. In all these cases the three primary inputs are selected as potential inputs without the aid of *add_essential_inputs()*. The construction of the second level of the LUT tree is done in a similar way. In that case however, *generate_inputs()* will also consider the outputs of level 1 LUTs as potential inputs.

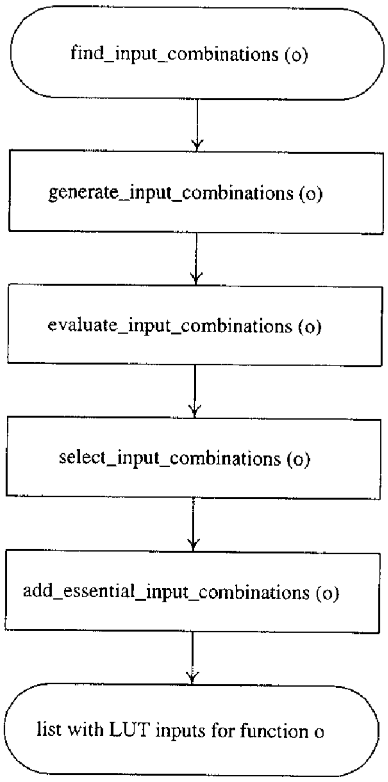

find_input_combinations (o)
generate_input_combinations (o)
evaluate_input_combinations (o)
select_input_combinations (o)
add_essential_input_combinations (o)
list with LUT inputs for function o

Figure 5.13: The *find_input_combination(o)* algorithm

5.6.4 Find_input_combinations(o)

5.6.4.1 The algorithm

The basic goal of *find_input_combinations(o)* is to find combinations of input set systems that will form the inputs of level n LUTs. The function will make the combinations based on the list of input set systems obtained by *find_inputs(o)*. The function *find_lut_outputs(o)*, which is discussed in the next section will then find the output set systems for each of these LUTs. The flow diagram of *find_lut_combinations(o)* can be found in Figure 5.13; it can be seen from this figure that *find_input_combinations(o)* closely follows the basic heuristic decision algorithm.

The *generate_input_combinations(o)* algorithm is rather straightforward. It takes the list of input set systems and it generates all possible input set system combinations that satisfy a number of conditions. Since all possible combinations of input set systems are tried, the list of combinations can become huge (this is also the reason why *find_inputs(o)* has been introduced). In order for an input combination to be accepted, it should satisfy the following conditions:

1. Each combination should consist of at least 2 and no more than $LUTNOI$ input set systems, where $LUTNOI$ defines the maximal number of inputs that a LUT can have.

2. Each pair of input set systems should be non redundant. This means that if ϕ_i and ϕ_j $(i \neq j)$ are two input set systems in the combination then the information present in ϕ_j should not be contained in the information of ϕ_i (i.e. $\phi_i \not\leq \phi_j$) and vice versa. This non redundancy check is not quite sufficient for LUTs that can have more than 2 inputs. For example, it is possible that a combination of three input set systems exist such that $\phi_i \cdot \phi_j \leq \phi_k$. However, due to the computational impact of checking of all this kind of combinations, only pairs of inputs will be checked for redundancy.

3. If the set system product of the input set systems only contains two blocks then it is checked if this set system has already been constructed on one of the previous levels. If such a set system is found and it was constructed for output o then the input set system combination is rejected.

The above rules construct a list of input set system combinations that are potentially useful for the construction of output o. This list of combinations is passed to *evaluate_input_combinations(o)* for the calculation of the weight factors. The evaluation of the input combination is based on the set system product of the input set systems in that combination. This set system product expresses the combined information of all the input set systems in the combination. Unfortunately, the input set system product will generally have more than two blocks. The output of a LUT is however a binary signal and can therefore only be represented by a two block set system. What is necessary is an abstraction of the input set system product to the output set system. Such an abstraction can be obtained by merging the set system blocks of the input set system product such that a two set block set system is left. This is the way in which *find_lut_outputs(o)* will find feasible output set systems for an input combination. The goal of *find_input_combinations(o)* is to look for input combinations that can generate useful set system outputs for the construction of ϕ_o. Finding these combinations is the target of *evaluate_input_combinations(o)*. This is accomplished by making a fast greedy prediction of the output set system that the *find_lut_outputs(o)* algorithm might construct. It is assumed that this greedy output set system will predict the set systems constructed by *find_lut_outputs(o)* well enough, so that decisions made on the greedy output set system are also valid for the output set systems that will be created later on by *find_lut_outputs(o)*. The prediction algorithm is illustrated using the following example.

Example 5.2 *Greedy output function prediction*
Let

$$\phi_I = \left\{\overline{1,2};\ \overline{3,4};\ \overline{5};\ \overline{6}\right\}$$

be the input set system product for a certain LUT and let

$$\phi_o = \left\{\overline{1,2};\ \overline{3,4,5,6}\right\}$$

be the function that is to be constructed. Since ϕ_I is a four block set system, it is first reduced to a three block set system. There are six ways to accomplish this:

1. Merging block 1 and 2 of ϕ_I : $\phi_1 = \left\{\overline{1,2,3,4};\ \overline{5};\ \overline{6}\right\}$
2. Merging block 1 and 3 of ϕ_I : $\phi_2 = \left\{\overline{1,2,5};\ \overline{3,4};\ \overline{6}\right\}$
3. Merging block 1 and 4 of ϕ_I : $\phi_3 = \left\{\overline{1,2,6};\ \overline{3,4};\ \overline{5}\right\}$
4. Merging block 2 and 3 of ϕ_I : $\phi_4 = \left\{\overline{1,2};\ \overline{3,4,5};\ \overline{6}\right\}$
5. Merging block 2 and 4 of ϕ_I : $\phi_5 = \left\{\overline{1,2};\ \overline{3,4,6};\ \overline{5}\right\}$
6. Merging block 3 and 4 of ϕ_I : $\phi_6 = \left\{\overline{1,2};\ \overline{3,4};\ \overline{5,6}\right\}$

For each of these set systems the set system similarity with ϕ_o is then calculated. The set system with the highest similarity is then selected and called ϕ'_I. Suppose that $SIM_{\phi_o}(\phi_4)$ is the best similarity then $\phi'_I = \left\{\overline{1,2};\ \overline{3,4,5};\ \overline{6}\right\}$. ϕ'_I is a three block set system that needs to be further reduced. This is done in a similar way:

1. Merging block 1 and 2 of ϕ'_I : $\phi_7 = \left\{\overline{1,2,3,4,5};\ \overline{6}\right\}$
2. Merging block 1 and 3 of ϕ'_I : $\phi_8 = \left\{\overline{1,2,6};\ \overline{3,4,5}\right\}$
3. Merging block 2 and 3 of ϕ'_I : $\phi_9 = \left\{\overline{1,2};\ \overline{3,4,5,6}\right\}$

Again the set system similarity between ϕ_o and each of the set systems ϕ_7, ϕ_8 and ϕ_9 is calculated. Since $\phi_9 = \phi_o$ that set system will have the highest set system similarity and that set system will be selected. In general it is of course not necessary that the greedy set system finds the primary output function. □

A few remarks with respect to this greedy selection process:

1. If there are more set systems with the highest similarity then one of them is picked at random (in *find_lut_outputs(o)* this is of course not desirable and a more sophisticated algorithm will be used there).
2. The block reduction step only reduces the number of blocks by one. So, if a ten block input set system product is calculated (which is very well possible using a four input LUT), it will take eight steps to come to a two set system block. This algorithm is however much faster than calculating the set system similarity for all possible block combinations directly (which is $O(2^b)$ where b represents the number of blocks).
3. The calculation of the set system similarities uses weighed incompatibility pairs using the formula: $weight = 1/noc$ where noc represents the number of occurrences of the incompatibility pair in the input set system products.

The algorithm performed by function *select_input_combinations(o)* is straightforward. It uses the parameters $SELECT\_INPUT\_COMBINATIONS\_QUALITY$ (default value 0.33) and $SELECT\_INPUT\_COMBINATIONS\_MAX$ (default value 100) to statically select the best input combinations using the algorithm described in Section 5.6.2.

Add_essential_input_combinations(o) acts as a safety net: if an incompatibility pair of ϕ_o is not implemented by the greedy input set system of the selected input combinations, then the best input combination that implements it is added to the selected set. This is however not so bad as it seems at the first sight: it is possible that the incompatibility pair is implemented by an input set system product, but it is not part of the greedy set system. However, it is better not to take any chance and add an extra combination to make sure that an implementation with this pair is possible.

5.6.4.2 The EB example

In this section the *find_input_combinations(o)* algorithm will be illustrated using the EB example circuit as described in Section 5.2.

As an example the input combinations for primary output ϕ_{y_0} will be calculated. In Section 5.6.3.2 it has been determined that the list of input set systems is: ϕ_{x_0}, ϕ_{x_2} and ϕ_{x_1}. Based on this list *generate_input_combinations(ϕ_{y_0})* will generate the following input set system products:

1. By combining ϕ_{x_0} and ϕ_{x_2}: $\phi_1 = \left\{\overline{0};\ \overline{1,2};\ \overline{3,5};\ \overline{4,6}\right\}$
2. By combining ϕ_{x_0} and ϕ_{x_1}: $\phi_2 = \left\{\overline{0,1};\ \overline{0,2};\ \overline{3,4};\ \overline{5,6}\right\}$

$a \mid b$	0	1	2	3	4	5
1	0.50					
2	0.50	0.50				
3	0.50	0.33	0.33			
4	0.33	0.50	0.33	0.50		
5	0.50	0.33	0.33	0.50	0.33	
6	0.33	0.33	0.50	0.33	0.50	0.50

Table 5.12: Weight of the incompatibility pairs in *find_input_combinations()*

3. By combining ϕ_{x_1} and ϕ_{x_2}: $\phi_3 = \left\{\overline{0,3};\ \overline{0,5};\ \overline{1,4};\ \overline{2,6}\right\}$

From these input set system product it is very easy to obtain Table 5.12 that contains the weighed number of occurrences of the incompatibility pairs in the input combination list. In this table the standard incompatibility weighing function $weight = 1/noc$ has been used. Using this table the blocks of the set systems can be reduced. For ϕ_1, six possible three block set systems can be constructed:

1. Merging block 1 and 2 of ϕ_1 (similarity 0.61): $\phi_{1,1} = \left\{\overline{0,1,2};\ \overline{3,5};\ \overline{4,6}\right\}$
2. Merging block 1 and 3 of ϕ_1 (similarity 0.61): $\phi_{1,2} = \left\{\overline{0,3,5};\ \overline{1,2};\ \overline{4,6}\right\}$
3. Merging block 1 and 4 of ϕ_1 (similarity 0.65): $\phi_{1,3} = \left\{\overline{0,4,6};\ \overline{1,2};\ \overline{3,5}\right\}$
4. Merging block 2 and 3 of ϕ_1 (similarity 0.56): $\phi_{1,4} = \left\{\overline{0};\ \overline{1,2,3,5};\ \overline{4,6}\right\}$
5. Merging block 2 and 4 of ϕ_1 (similarity 0.60): $\phi_{1,5} = \left\{\overline{0};\ \overline{1,2,4,6};\ \overline{3,5}\right\}$
6. Merging block 3 and 4 of ϕ_1 (similarity 0.65): $\phi_{1,6} = \left\{\overline{0};\ \overline{1,2};\ \overline{3,4,5,6}\right\}$

In this case there are two set systems with the maximal similarity: $\phi_{1,3}$ and $\phi_{1,6}$. LUTSYN takes the first one and performs the block merging algorithm again:

1. Merging block 1 and 2 of $\phi_{1,3}$ (similarity 0.39): $\phi_{1,3,1} = \left\{\overline{0,1,2,4,6};\ \overline{3,5}\right\}$
2. Merging block 1 and 3 of $\phi_{1,3}$ (similarity 0.43): $\phi_{1,3,2} = \left\{\overline{0,3,4,5,6};\ \overline{1,2}\right\}$
3. Merging block 1 and 4 of $\phi_{1,3}$ (similarity 0.48): $\phi_{1,3,3} = \left\{\overline{0,4,6};\ \overline{1,2,3,5}\right\}$

In this case, set system $\phi_{1,3,3}$ is selected. This set system will be called $\phi_{greedy_{y_1}}$. For the set systems ϕ_2 and ϕ_3 greedy algorithms can be obtained in a similar way. The sorted list of greedy set systems becomes:

$$\phi_{greedy_{y_2}} = \left\{\overline{0,2,3,4};\ \overline{0,1,5,6}\right\} \quad \text{similarity} = 0.65$$

$$\phi_{greedy_{y_3}} = \left\{\overline{0,3,5};\ \overline{1,2,4,6}\right\} \quad \text{similarity} = 0.48$$

$$\phi_{greedy_1} = \left\{\overline{0,4,6};\ \overline{1,2,3,5}\right\} \quad \text{similarity} = 0.48$$

This list also shows the drawback of a fast greedy search algorithm, the set system ϕ_{greedy_3} is the same set system as input set system ϕ_{x_2} and thus the proposed LUT function would be $f = x_2$. In the next section it will be shown using an example that a more sophisticated search algorithm will be able to find useful output set systems for input combination (ϕ_{x_1}, ϕ_{x_2}); in fact as can be seen from Figure 5.1, this combination is used in the final solution.

Select_input_combinations(ϕ_{y_0}) uses this list of greedy set systems to determine which input combinations should be removed. Using the default values 0.33 for *SELECT_INPUT_COMBINATIONS_QUALITY* and 100 for *SELECT_INPUT_COMBINATIONS_MAX* all input combinations will be accepted. Since all incompatibility relations of ϕ_{y_0} are implemented by at least one greedy set system, the algorithm *add_essential_input_combinations(ϕ_{y_0})* does not need to add other input combinations (this was not possible anyway, since all combinations already have been selected).
For the second output *select_input_combinations(ϕ_{y_1})* will of course find the same 3 input combinations. These input combinations are also found for the common subfunction expressed by cluster C.

5.6.5 Find_lut_outputs(o)

5.6.5.1 The algorithm

The output of *find_input_combinations(o)* is a list of combinations of input set systems that will be used as the input for a LUT. Each of these LUTs can implement a number of different output functions. It is the task of *find_lut_outputs(o)* to create a few promising output functions for each of these LUTs. The flow diagram of *find_lut_outputs(o)* is very similar to the other two heuristic decision algorithms (see Figure 5.14); in this case there is however an extra block *add_to_lut_tree(o)* which takes care of the actual creation of the LUTs and for adding these LUTs to level n of the LUT tree.

Generate_lut_outputs(o) takes as its input the list of selected input combinations obtained by *find_input_combinations(o)*. The goal of the generation process is to create few promising LUT output functions for each of these input combinations. This approach is a little different from *generate_inputs(o)* and *generate_input_combinations(o)* that generate all feasible items, while *generate_lut_outputs(o)* will use a heuristic decision process to only create a few promising output functions. The reason for this is, that generating all possible outputs is computationally expensive. For example, a 4-input LUT can result in a input set system product that contains 16 blocks and hence 2^{16} possible output functions. *Generate_lut_outputs(o)* will create a list of output set systems for each input combination by repeatedly merging two blocks of the set system and keeping only the most promising one. The algorithm is very similar to the greedy algorithm used in *find_input_combinations(o)* only here a number of good set systems will be selected instead of only the best one. The algorithm will be illustrated using the following example.

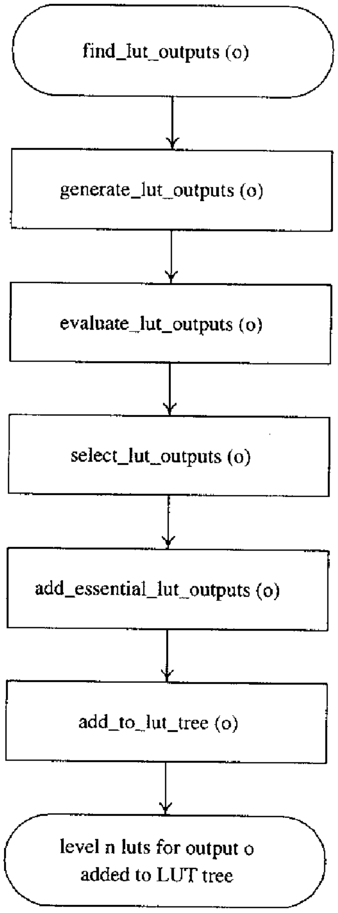


Figure 5.14: Flow diagram for *find_lut_outputs(o)*

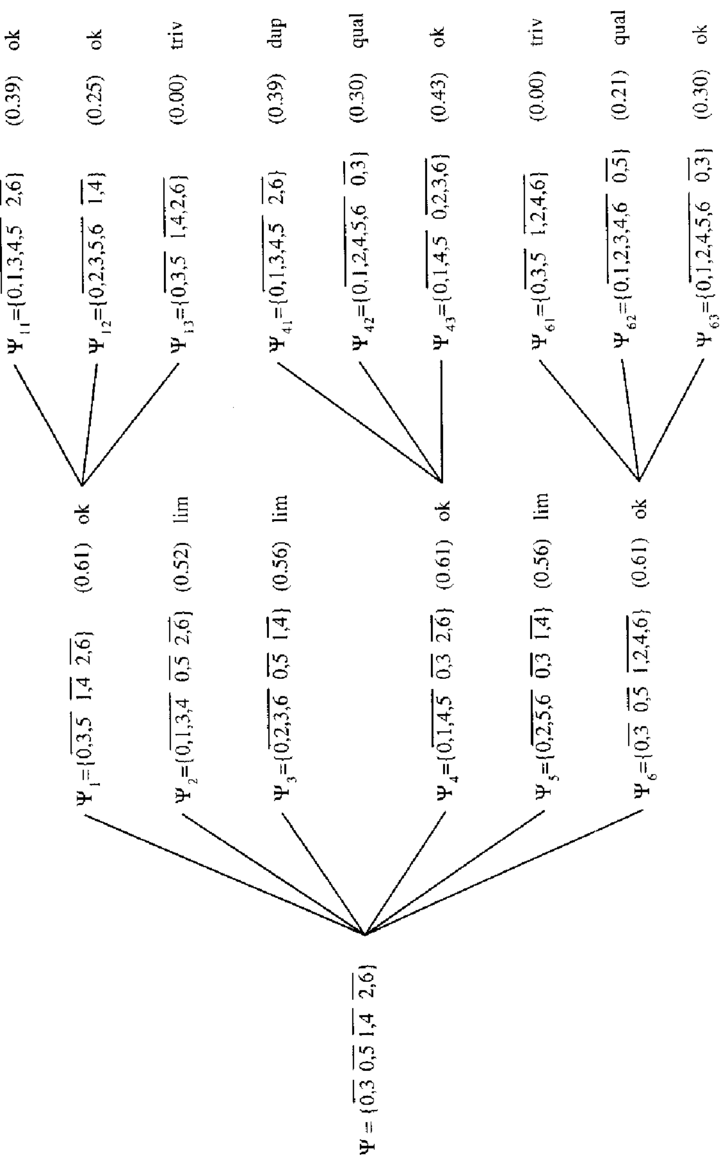

$\Psi = \{\overline{0,3}\ \overline{0,5}\ \overline{1,4}\ \overline{2,6}\}$
$\Psi_1=\{\overline{0,3,5}\ \overline{1,4}\ \overline{2,6}\}$ (0.61) ok
$\Psi_2=\{\overline{0,1,3,4}\ \overline{0,5}\ \overline{2,6}\}$ (0.52) lim
$\Psi_3=\{\overline{0,2,3,6}\ \overline{0,5}\ \overline{1,4}\}$ (0.56) lim
$\Psi_4=\{\overline{0,1,4,5}\ \overline{0,3}\ \overline{2,6}\}$ (0.61) ok
$\Psi_5=\{\overline{0,2,5,6}\ \overline{0,3}\ \overline{1,4}\}$ (0.56) lim
$\Psi_6=\{\overline{0,3}\ \overline{0,5}\ \overline{1,2,4,6}\}$ (0.61) ok
$\Psi_{11}=\{\overline{0,1,3,4,5}\ \overline{2,6}\}$ (0.39) ok
$\Psi_{12}=\{\overline{0,2,3,5,6}\ \overline{1,4}\}$ (0.25) ok
$\Psi_{13}=\{\overline{0,3,5}\ \overline{1,4,2,6}\}$ (0.00) triv
$\Psi_{41}=\{\overline{0,1,3,4,5}\ \overline{2,6}\}$ (0.39) dup
$\Psi_{42}=\{\overline{0,1,2,4,5,6}\ \overline{0,3}\}$ (0.30) qual
$\Psi_{43}=\{\overline{0,1,4,5}\ \overline{0,2,3,6}\}$ (0.43) ok
$\Psi_{61}=\{\overline{0,3,5}\ \overline{1,2,4,6}\}$ (0.00) triv
$\Psi_{62}=\{\overline{0,1,2,3,4,6}\ \overline{0,5}\}$ (0.21) qual
$\Psi_{63}=\{\overline{0,1,2,4,5,6}\ \overline{0,3}\}$ (0.30) ok

Figure 5.15: Generation of LUT output set systems for set system ψ

Example 5.3 *Finding output functions for a LUT*
In this example the set system

$$\psi = \left\{\overline{0,3};\ \overline{0,5};\ \overline{1,4};\ \overline{2,6}\right\}$$

will be used. This set system is the set system product of the input combination of primary inputs ϕ_{x_1} and ϕ_{x_2} as discussed in Section 5.6.4.2. The goal is to construct output set systems for output ϕ_{y_0}. The greedy set system algorithm was not able to find a useful output function for the LUT. In this example it will be shown that a more sophisticated search algorithm will be able to do find a useful output function.

The generation of the output set systems is done by repeatedly merging pairs of blocks of the input set system product until a two block set system remains. The difference with the greedy algorithm is however, that more than one set system may be selected. The selection process is based on the *select()* algorithm described in Section 5.6.2. As an absolute limit the parameter $SELECT\_OUTPUT\_SS\_MAX$ with a value of 3 is used and as the relative limit the parameter $SELECT\_OUTPUT\_SS\_QUALITY = 0.80$ will be used. The select algorithm is illustrated in Figure 5.15. Since, ψ is a 4-block set system, 6 3-block set systems can be computed from it by merging different combination of two blocks. These set systems are the set systems from ψ_1 to ψ_6. For each of these set systems the similarity with ϕ_{y_0} is calculated. $SELECT\_OUTPUT\_SS\_MAX$ will allow only three set systems to be selected. Therefore the 3 set systems with the highest similarity are selected (in Figure 5.15 these are marked with "ok"), the other three set systems will be removed (marked with "lim" because they violate the maximal set system limit).
For each of the selected 3-block set systems, the number of blocks is further reduced by merging two blocks. For set system ψ_1 three two-block set systems $\psi_{1,1}$, $\psi_{1,2}$ and $\psi_{1,3}$ are constructed. From this list $\psi_{1,3}$ is removed because it is the same as one of the LUT inputs (i.e. it implements the trivial function $f = x_2$); this set system is marked with "triv" (trivial). The other two set systems satisfy the parameters $SELECT\_OUTPUT\_SS\_MAX$ and $SELECT\_OUTPUT\_SS\_QUALITY$ and are therefore accepted. For set system ψ_4 also three two-block set systems can be constructed: $\psi_{4,1}$, $\psi_{4,2}$ and $\psi_{4,3}$. However, $\psi_{4,1} = \psi_{1,1}$ so this set system will not be duplicated. Set system $\psi_{4,3}$ is accepted, but $\psi_{4,2}$ is destroyed because it does not satisfy the relative limit $SELECT\_OUTPUT\_SS\_QUALITY = 0.80$; this set system is marked with "qual". For ψ_6 similar considerations apply.
Summarizing, the following four output set systems have been selected by *generate_lut_outputs*(ϕ_{y_0}) for the input combination (ϕ_{x_1}, ϕ_{x_2}):

$$\psi_{1,1} = \psi_{4,1} = \left\{\overline{0,1,3,4,5};\ \overline{2,6}\right\}$$

$$\psi_{1,2}\left\{\overline{0,2,3,5,6};\ \overline{1,4}\right\}$$

$$\psi_{4,3}\left\{\overline{0,1,4,5};\ \overline{0,2,3,6}\right\}$$

and

$\psi_{6,3}\left\{\overline{0,1,2,4,5,6};\ \overline{0,3}\right\}$

□

By repeating this algorithm for each of the input combinations a list of LUTs is created. This list is passed to *evaluate_lut_outputs(o)* which has a very simple task: since the similarities of the LUTs already have been calculated, the LUT lists only needs to be sorted with respect to these similarities. The sorted list is then passed to a dynamic selection *algorithm select_lut_outputs(o)* that uses the parameters $SELECT\_LUT\_OUTPUTS\_MAX = 200$ and $SELECT\_LUT\_OUTPUTS\_QUALITY = 0.33$. If necessary, *add_essential_lut_outputs(o)* will add extra LUTs for incompatibility relations of ϕ_o that not have been implemented. This is not strictly necessary, because level $n+1$ can obtain the missed incompatibility relations from a level $m : m < n$. However, getting information from lower levels will result in long wires, which is undesirable from a routing point of view. Adding these incompatibility pairs, gives the build process for level $n+1$ at least the possibility to obtain this information from the level n LUTs.

The last step of the *find_lut_outputs(o)* algorithm is the block *add_to_lut_tree(o)*. This block performs two tasks. First of all, it adds the selected LUTs to the LUT tree, so that they can be used in the build process for the next levels. Secondly, it will check if one or more of the added LUTs implements the function o. If this is the case then the LUT is marked with this fact and output o is marked done.

In the next subsection *find_lut_outputs(o)* will be illustrated using the EB example function.

5.6.5.2 The EB example

In this example, the construction of the level 1 LUTs for output y_0 of function EB will be described. In Section 5.6.4.2 it has been shown that *find_input_combinations(y_0)* has selected three different input combinations. For easy reference, the set system product of these set systems have been copied here:

1. By combining ϕ_{x_0} and ϕ_{x_2}: $\phi_1 = \left\{\overline{0};\ \overline{1,2};\ \overline{3,5};\ \overline{4,6}\right\}$
2. By combining ϕ_{x_0} and ϕ_{x_1}: $\phi_2 = \left\{\overline{0,1};\ \overline{0,2};\ \overline{3,4};\ \overline{5,6}\right\}$
3. By combining ϕ_{x_1} and ϕ_{x_2}: $\phi_3 = \left\{\overline{0,3};\ \overline{0,5};\ \overline{1,4};\ \overline{2,6}\right\}$

Using these set systems, the weights of the incompatibility pairs can be calculated. Since this are the same set systems as used in *find_input_combinations(y_0)* the weights of the pairs are equal and Table 5.12 can be used to calculate the similarities.
For set systems ϕ_1, ϕ_2 and ϕ_3 *generate_lut_outputs(y_0)* will then find a list of LUTs. For ϕ_3 this process has been detailed in the example from the previous section; the same algorithm can be applied for ϕ_1 and ϕ_2. The result of *generate_lut_outputs(y_0)* is a list of ten different output set systems. This list is then sorted by *evaluate_lut_outputs(y_0)*

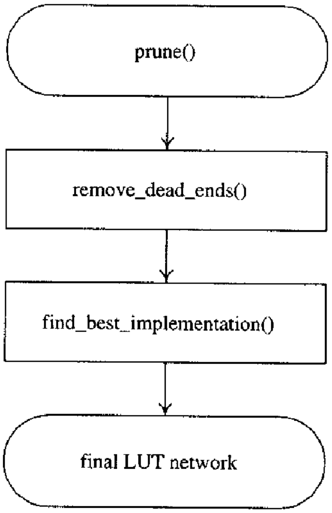


Figure 5.16: The flow diagram of the prune algorithm

and handed to *select_lut_outputs(y_0)*. Since the best LUT's similarity is 0.65 and the lowest similarity is 0.30 all set systems are accepted using the default search parameters: $SELECT\_LUT\_OUTPUTS\_MAX = 200$ and $SELECT\_LUT\_OUTPUTS\_QUALITY = 0.33$. Finally, *add_to_lut_tree(y_0)* will add these LUTs to the LUT tree and check if any of these set systems implements an output or a cluster. Unfortunately this is not the case and an extra level of the tree is required for the implementation of output y_0.
Looking at the other output *find_lut_outputs(y_1)* will result in a list of 7 LUTs. For the cluster C also 7 LUTs are found, which makes a total of 24 level 1 LUTs.

5.7 Prune() : Find best of feasible realizations

5.7.1 The algorithm

The output of the heuristic *build()* algorithm is a level n LUT tree that implements all the primary output functions. *Prune()* performs two postprocessing steps to transform the LUT tree into the final LUT network. The flow diagram of *prune()* can be found in Figure 5.16. The task of *remove_dead_ends()* is simple. The LUT tree will contain LUTs that are not used for the implementation of any of the primary output functions. These LUTs were created because the heuristic algorithms in *build()* determined that they were promising, but in the end they were not needed. They are removed by *remove_dead_ends()*, so that only the LUTs actually used for the calculation of one of the outputs remain.

It is possible that more than one implementation for a primary output has been found. In that case, it is necessary to choose for each primary output one implementation, such that the overall number of LUTs is minimal. This is the task of the function *find_best_implementation()*. At the moment *find_best_implementation()* is an exhaustive search algorithm that finds the optimal number of LUTs. Until today it was not necessary to create a heuristic version of this algorithm, because the computation times of *find_best_implementation()* were considerably lower than the computation times of *build()*. The result of *find_best_implementation()* is a LUT tree that implements all the output functions such that overall number of LUTs is minimal. This tree is saved in an output file and is the realization found by the LUTSYN algorithm.

5.7.2 Pruning EB

With respect to the EB circuit, the LUT tree constructed by *build()* consists of 24 level-1 LUTs and 234 level-2 LUTs. It has found 12 different realizations for output y_0 and 10 realizations for output y_1.
After all unused LUTs have been removed from the LUT tree by *remove_dead_ends()*, 11 level-1 LUTs and 22 level-2 LUTs remain. *find_best_implementation()* then finds the minimal number of blocks that is needed in a realizations. Since y_0 is implemented 12 different ways and y_1 has 10 different realizations the total number of possible realizations comes to 120. *find_best_implementation()* finds 8 implementations that use 4 LUTs, 16 implementations that use 5 LUTs and 96 implementations that require 6 LUTs. From the 8 implementations that require 4 LUTs one implementation is selected at random which will be the implementation found by LUTSYN. This implementation can be found in Figure 5.1 in Section 5.2.

This completes the discussion of the LUTSYN algorithms. Some practical results obtained using the LUTSYN environment will be discussed in the next chapter.

Chapter 6

Experimental results

The algorithms described in Chapter 5 have been implemented in the form of a C++ program. Experimental results obtained with this program can be found in this chapter.

6.1 Experimental results

Although LUTSYN is not yet a complete automated synthesis tool, it is interesting to look at some synthesis results that can be obtained from it, to see what results can be produced by such a first prototype and to have a first impression on the proposed decomposition approach. In Table 6.1 some synthesis results for 2-input, 1-output LUTs have been presented for a number of Boolean functions from the MCNC logic synthesis and optimization benchmark set [Yan91]. The first three columns after the circuit name represent the number of inputs, outputs and the number of product terms in the function table of the description. These function tables have been preprocessed using the preprocessor described in Section 4.3.4 and Section 5.4. As can be seen from the table, the preprocessor was not able to reduce the number of product term of the circuits *rd53*, *shift* and *xor5*. This is due to the fact that these circuits have orthogonal product terms, so it is impossible to collapse them into larger product terms.
The fifth column in the table presents synthesis results obtained fully automatically by LUTSYN, using the default search parameters described in the previous chapter. In this column the number of levels of the circuit have also been presented. The results for LUTSYN are compared to DEMAIN [LS95], one of the best logic synthesis algorithms for look-up table based architectures currently available. This algorithm, developed at the University of Warsaw is in a way similar to LUTSYN: both algorithms are being used for the research on Boolean function decomposition and both algorithms have both an interactive and an automated mode. Both use the set system based approach. The difference is however, that DEMAIN uses a top-down reduction approach in which it combines a bit parallel decomposition with an parallel (disjoint) input bit decomposition (see Section 3.3.2) whereas LUTSYN strives to find general decompositions using the bottom-up construction approach. Since obtaining the number of levels from DEMAIN is not straightforward, they have been calculated by hand for two small circuits. As can be seen from Table 6.1, the results of LUTSYN and DEMAIN are about comparable, except for the the symmetric function *rd53*, where DEMAIN produces much better re-

Benchmark	#inputs	#outputs	#terms	#LUTs LUTSYN	#LUTs DEMAIN
a06	5	5	12	29/4	27
bbtas	5	5	19	29/4	29
con1	7	2	19	36/6	37
dk27	4	5	12	23/3	24
eb	3	2	7	4/2	5/3
lion	4	3	11	15/3	13/4
rd53	5	3	32	36/6	26
shift	4	4	16	23/3	24
tra04	4	3	14	11/4	11
xor5	5	1	16	4/3	4/3

Table 6.1: LUTSYN results for 2-input, 1-output LUTs

sults. This is very encouraging since LUTSYN is only a prototype research environment which will be used for the development of adequate heuristic algorithms. Apparently, the already implemented simple and yet incomplete heuristics of LUTSYN are already able to find good decompositions for many circuits from the MCNC benchmark set.

In the remaining of this section a number of benchmark results will be examined in more detail.

The interesting aspect of the *a06* benchmark are the preprocessor results. It turned out that this benchmark had a redundant input variable. This fact is not easy to detect in the original function table (Figure 6.1), but the preprocessor discovered it and used it to reduce the input file, what is evident from the output file of the preprocessor (Figure 6.2). From this figure it can be clearly seen that for all care conditions *I*1 and *I*2 have the same value.

For the *eb* circuit, the difference between the LUTSYN and the DEMAIN solution can be explained by the fact that DEMAIN needs an extra level for the implementation. This level is necessary, because as the first step, DEMAIN performs a parallel decomposition that splits the two-output function describing *eb* into two single-output function. This reduces the decomposition problem for *eb* in two independent (but simpler) synthesis problems. These two parts can however not share logic, and sharing logic is necessary to find the 4 block solution. Furthermore, DEMAIN uses a parallel (disjoint) input bit decomposition but for the 4 block solution the non-disjoint decomposition is required (see Figure 5.1). This shows once again the advantages of the general decomposition paradigm.

The effect of one of the optimization criteria of LUTSYN ("try to minimize the number of levels in the realization") can be seen from the results obtained for the *lion* benchmark. Although DEMAIN has a solution that uses 13 LUTs (while LUTSYN requires 15), the

```
.i 5
.o 5
.ilb I1 I2 PS2 PS1 PS0
.ob NS2 NS1 NS0 O1 O2
.p 28
10000   01000
01000   00100
00000   -----
11000   -----
10100   11000
01100   10100
00100   -----
11100   -----
10101   11000
01101   01100
00101   -----
11101   -----
10001   01000
01001   01010
00001   -----
11001   -----
10110   00010
01110   10010
00110   -----
11110   -----
10010   00001
01010   10001
00010   -----
11010   -----
10011   11000
01011   01010
00011   -----
11011   -----
.e
```

Figure 6.1: Preprocessor input file a06.pla

```
.i 5
.o 5
.ilb I1 I2 PS2 PS1 PS0
.ob NS2 NS1 NS0 O1 O2
.type fr
.p 12

--111 00000
1000- 01000
1010- 11000
10011 11000
010-1 01010
01000 00100
01100 10100
01101 01100
10110 00010
01110 10010
10010 00001
01010 10001
.end
```

Figure 6.2: Preprocessor output file a06.pre

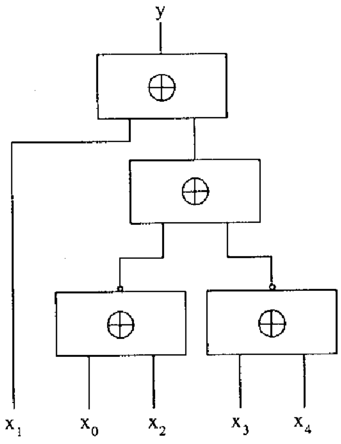


Figure 6.3: Implementation found by LUTSYN for *xor5*

solution offered by LUTSYN uses only 3 levels, while DEMAIN needs 4. For small circuits this does not seem very important, but extra levels do have an impact on the delay, routability and the complexity of large circuits.

In Figure 6.3 the network structure found by LUTSYN for benchmark *xor5* can be found. From this figure it can be seen that LUTSYN discovers the only sensible network structure, but uses the complement of an EXOR in two cases. Since

$$a \oplus b = \overline{a} \oplus \overline{b}$$

the network with and without the complements are functionally equivalent. The complements are due to the fact that the phase of intermediate signals is arbitrarily assigned. If the negated phase assignment had been used then these complements would not have been necessary.

The network constructed by LUTSYN for benchmark *tra04* can be found in Figure 6.4. In this figure, the function implemented by each block is represented by a Boolean expression. In this expression a and b represent the inputs of that block. This network is interesting for two different reasons. First of all it shows that LUTSYN really uses a rich set of Boolean blocks: in the implementation of *tra04* five different block functions have been used. Secondly, from this network it can be seen that the subfunction creation strategy works. Two clusters have been formed and the subfunctions constructed for these clusters have been used: in this case one subfunction is shared by all output

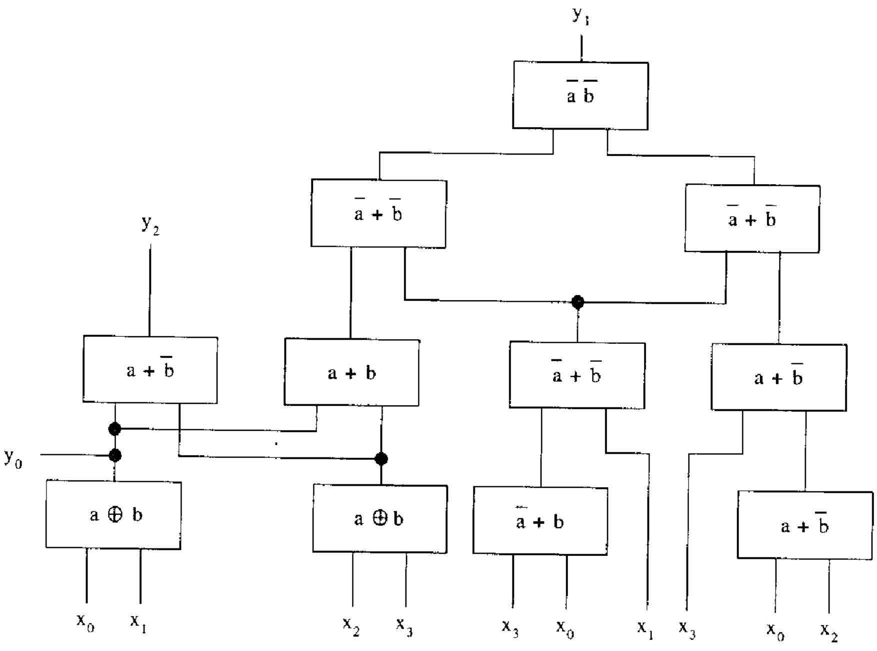


Figure 6.4: Implementation found by LUTSYN for *tra04*

functions and another subfunction is shared by y_1 and y_2.

LUTSYN does not give good results for the symmetric function *rd53*. This is caused by the fact that LUTSYN is able to construct the functions for two of the three outputs in the first few levels, but fails to construct the last function. This is obviously a flaw in the heuristic algorithm. Ways to improve LUTSYN for symmetric functions can be found in the next section.

6.2 Symmetric functions and LUTSYN

It has been known for a long time that special care has to be taken when doing a multiple level synthesis of symmetric combinational functions [KD91]. The special care consists of exploiting the following property of symmetric functions: the inputs are disjointedly decomposed over the tree, i.e. one half of the inputs is used in the left half of the function tree, whereas the other half of the inputs is exclusively used on the right. Since DEMAIN is very good in finding parallel decompositions [ŁKJ91] and because it is working top-down it is able to exploit the special properties of symmetric functions very well.

The same special properties prevent LUTSYN from finding good solutions for symmetric functions. This is caused by the following reason: when LUTSYN selects inputs or input combinations for the generation of the next level of LUTs, these inputs and input combinations are selected independently; their similarity with the output set system is calculated and the best are selected. However, due to the symmetric nature these similarities are about equal. This means that LUTSYN selects more or less randomly a number of combinations, without keeping the overall result into view. What needs to be done in LUTSYN is to create a new heuristic algorithm that selects the inputs and input combinations not independently, but in groups. Each group consists of a number of set systems that together implement all the input combinations of a Boolean function. The set systems in such a group need to be related, i.e. it should be easy to combine the information that they possess. These groups will be kept together across the construction of levels, so the next level will use the information in the group and tries to strengthen the group by adding new promising set systems to it or by replacing set systems in the group by better ones.

Such an algorithm will not only benefit symmetric functions, but will benefit the synthesis of all combinational functions. This will be the first problem that will be addressed in the follow-up research, and in fact further research in this area has already been started.

6.3 Larger LUT blocks

LUTSYN is also able to construct networks for LUTs that have more than 2 inputs. Unfortunately, the construction of these networks takes at the moment an unpractical amount of computing resources because of too weak and incomplete heuristics. The bottle-neck for 4 input LUTs is the huge increase in the number of candidate input combinations for the LUTs. For 4 inputs, all combinations of 2, 3 and 4 inputs must be checked. The grouping algorithm suggested in the previous section will certainly help, but that is not all.

During some experiments with LUTSYN it has been found that when the number of blocks in a set system increases the important information can be stored in more ways. For example, when using two-block set systems there are not many ways incompatibility relations can be stored in the set system. However, using large set systems (for example, 4-input LUTs can have a 16-block product set system) the incompatibility relations that represent the necessary information can be distributed over many different set system blocks. The set system product of two set systems with very different set system blocks results in a product set system with many blocks and abstraction is necessary to reduce the block count so that the set system product can be encoded with the available number of output bits. The set system product of two set system with similar blocks, is a set system with relatively few blocks and requires therefore only a little abstraction. In other words, combining and preserving the information present in two set systems becomes easier if the blocks of these set systems are more alike.
This means that the simple information driven similarity measure defined in Section 4.3.3.2 needs to be extended to include a structural component: a value that indicates how much the blocks of the two set systems are similar. A number of algorithms for this are currently under investigation. With such an algorithm it will be undoubtedly possible to generate general decompositions for LUTs with any number of inputs in a reasonable time.

6.4 Improvements

One of the first things that need to be done with LUTSYN is to extend and refine the similarity measures and heuristics. Two examples of extensions for which research is ongoing have been proposed in the previous sections. A few other possible improvements are the following:

1. LUTSYN is not the fastest algorithm and uses a lot of memory. This is the result of its software implementation that aimed to provide not a final synthesis tool for practical applications, but a research laboratory setup. It is slow because it performs a lot of integrity checks on the internal data structures. It uses a lot of memory because it keeps a lot of information that is not required by the program, but which is useful information for the researcher that uses the program. Both these properties are valid in research and useful in research software, but make the program unsuitable for the synthesis of large circuits. Therefore, an option that disables these features would be very desirable.

2. Extra heuristics based on the similarity measures between the input/output set systems and between the input set systems should be implemented for better conducting the search for solutions; especially for symmetric and strong asymmetric functions. These heuristics will both limit the search space (and search time), and result in better decompositions.

3. LUTSYN is currently unable to work with LUTs that have more than one output. For architectures that allow more than one output (such as Xilinx FPGA), this means that not all possibilities offered by the FPGA are used. Adding more outputs to LUTSYN is not very difficult. If LUTSYN is extended in this respect, it is important to realize that it is better to model the outputs with a number of two block set systems, than to model it using one $n(n > 2)$ block set system. In the first case, it is much easier to use only subsets of the LUT outputs as the input of another LUT.

4. Another point of improvement of LUTSYN is the way that subfunctions are created. The concept itself (explicitly search for useful subfunctions) is very good, but the current clustering algorithm itself is not so good. It has the following weak points: first of all, the threshold for creating a cluster should not be a constant (as it is now) but should take into account for how many functions the cluster is a subfunction. If a cluster is common to many output functions, than it is acceptable to lower the threshold, because overall gain is still possible. Secondly, it sometimes creates strange subfunctions. This is caused by the fact, that instead of a real subfunction a set of common incompatibility relations is used in the algorithm and this set of incompatibility relations not always translates to a nice set system.
   Another way of tackling this problem is by combining the bottom-up with the top-down approach. The main advantage of the top-down approach that it is able to create good subfunctions in an easier way.

Although still a lot of research needs to be performed, the preliminary results from the first prototype version of LUTSYN with very simple and not complete heuristics show the power and abilities that are offered by general decomposition and therefore, we are encouraged to continue the research in this field.

Chapter 7

Conclusions and recommendations

In this chapter the discussion in this thesis will be recapitulated and some suggestions for improvements and future research will be given.
In Section 7.1 the most characteristic features of LUTSYN are listed and the differences with other logic synthesis methods are shown. Then, in Section 7.2, the contributions of the Ph.D. work are enumerated. The last section of this chapter contains a look into the future; in Section 7.3 a few possible improvements for the LUTSYN algorithm will be discussed.

7.1 Characteristic features of LUTSYN

In this thesis a new general decomposition synthesis approach is considered and a synthesis method based on this approach and the associated tool called LUTSYN (*Look-Up Table SYN*thesis) are presented. LUTSYN deals with the multiple-level synthesis for Boolean functions using look-up table based Field-Programmable Gate Arrays (FPGAs) for the implementation. The method is targeted for controller-like circuits and has a number of characteristic features that distinguishes it from other logic synthesis methods.

The most prominent feature of LUTSYN is that it creates general decompositions being acyclic networks, i.e. without loops (see Section 2.4). A general decomposition in the form of an acyclic network means that the method does not impose any restrictions on the implementation (other than the physical limits imposed by the FPGA and the fact that no feedback loops can be created); i.e. every possible block functions, network structures, input combination etc. is permitted and can be potentially found by the decomposition algorithm. This is different from all methods and algorithms described in Chapter 3; which are all special cases of general decomposition. For example the division based synthesis methods (Section 3.1) limit the block functions to AND, OR and NOT only. The graph based synthesis algorithms in Section 3.2 limit themselves to multiplexer type block functions. The special cases of general decomposition (Section 3.3) are as the name suggests, only able to find special cases of the general decomposition structure and thus are not general. The functional decomposition based methods and algorithms (Section 3.3.1), that are the most closely related to LUTSYN, restrict the way in which the decompositions, and in particular their free and bound input sets can be created. Their current implementations involve a generator that is a special case of the LUTSYN generator.

A second important aspect of LUTSYN is that it is based on the canonical set system theory introduced in Section 4.2. The set system theory has two important aspects. First of all, set systems are used as the single representation model within LUTSYN: set systems are used for the representation of input and output signals and for the representation of the building block functions. Network structures are modeled using networks of set systems. Secondly, the operators defined in set systems are used as a mathematical framework for the analysis and construction of LUT networks. The set system representation has a number of important advantages over other representations:

1. Set systems provide an efficient and compact way of representing Boolean functions and especially the controller-type functions. The number of symbols in set systems grows linearly with the number of terms and the number of set systems used to model a function is equal to the number of its input and output variables. Thus, the only factor which considerably influences the size of the representation is the number of term symbols because these symbols are in all input and output set systems. Fortunately, the controller type logic is specified always by a relatively small number of terms (control predicates), this is true for a number of other types of logic also. Only some special type functions, like EXOR, have a number of terms which grows exponentially with the number of inputs, but the considered decomposition method is not devoted to such functions. For other representations like the product term representation or the OBDD representation the size of the representation can significantly differ for practical Boolean functions, and heavily depends of the number of input and output variables.

2. Set systems provide a compact maximal functionally complete representation of logic blocks and Boolean functions. This means that any possible function defined in the representation algebra is used. For example, for 2-input, 1-output LUTs this means that any 2-input, 1-output function can be used at the same cost. Other representations (like those used by division based synthesis algorithms) use only minimal functionally complete representations: they consider only the operators AND, OR and NOT as block function and require technology mapping to form more complex cells.

3. Set system and set system operators model the concept of don't cares very naturally and nothing special has to be done for exploiting don't cares. Their exploitation follows from the synthesis target directed decomposition decisions. This is different from division based logic synthesis methods that use a two-level minimizer to remove all original don't cares before starting the actual synthesis. By keeping all the don't cares and passing them to the real synthesis algorithm, the whole design freedom is given to the actual synthesis process, and used to synthesize better circuits.

4. Set systems also model exactly what is needed for the synthesis with LUTs: they model the block functions but they don't model the phase of the signals because that information is not important for the synthesis.

5. The synthesis performed by LUTSYN uses only three operators (the set system less or equal operator, the set system product operator and the merging of set system blocks). All these operators can be very efficiently computed on set systems.

LUTSYN models the FPGAs as networks of LUTs. This means that LUTSYN synthesizes the Boolean functions using building blocks that can be mapped one to one to the blocks used by the implementation, and thus it does not require technology mapping. This is a major advantage since in many cases the technology mapping phase must perform the actual logic synthesis starting only from some input network that realizes the function in a way randomly related to the actual synthesis target, and with a removed design freedom. Furthermore, it has to alter the network structure obtained by the technology independent logic synthesis algorithm in order to fit it into the building blocks used in the implementation. In fact, technology mapping must often locally resynthesize a part of the network in order to find a feasible mapping to the given building blocks.

A very characteristic feature of LUTSYN is that it uses a clustering algorithm that searches explicitly for common subfunctions, whereas other methods find subfunctions only implicitly as the by-product of the synthesis process. Furthermore, only these explicitly found subfunctions will be used as subfunctions. This is done to prevent the creation of small subfunctions that need to be distributed all over the FPGA and therefore cause long wires.

LUTSYN uses a bottom-up construction algorithm. Starting from the inputs, it constructs a network of LUTs until all outputs have been implemented. Also, the algorithm constructs a number of potential solutions in parallel. This is different from most other synthesis algorithms that use a top-down reduction technique (a special case of decomposition where the problem is decomposed into trivial and nontrivial part, that is further decomposed) for the network construction and that construct solutions serially or do not create more than one solution at all.

Since the problem of optimal logic synthesis is computationally complex, strictly optimal implementations cannot always be obtained using a realistic amount of resources. Therefore heuristic algorithms are used that strive to find a solution that would be as near as possible to the optimal implementation. All heuristic decisions made by the search algorithm are based on a single heuristic decision framework that is applied consequently everywhere, thus ensuring that all decisions work in the same direction. The heuristic algorithms are highly controllable by the user, e.g they allow the user to specify a trade-off between the required computation time and the quality of the implementation found. They can also easily be changed or replaced because the heuristic part is separated from the framework part of LUTSYN. A set of default values for all search parameters is defined, that allows to find good decompositions without deep knowledge of the heuristic algorithms.

7.2 Contributions of the work

To the best of my knowledge the work presented in this thesis is highly original. This opinion has been confirmed by a number of researchers with an outstanding reputation in this research area. In this section the contributions of the work will be summarized.

- The formalization of the set system theory is new. The formulation presented here introduces the canonical representation of set systems. This overcomes the problem of having several set systems that represent the same information, a problem that was present in earlier descriptions of the set system theory [HS66], [Kon95]. Furthermore, by proving that $(Setsys(\simeq), \sqsubseteq)$ forms a lattice and defining the canonical representative of $\simeq$ as the top of this lattice we were able to show that all necessary set system operators preserve the canonical property.
  In general, the formulation presented here, has increased the mathematical insight in the set system theory and has made this theory more consistent and easier to use.
- LUTSYN is the first algorithm that really performs a general decomposition, other logic synthesis algorithms for LUT based FPGAs all use special cases of general decomposition (Section 4.3.6).
- The explicit search for common subfunctions and the way this information is used in a *bottom-up construction algorithm* is novel (see Section 4.3.5).
- The way set system similarities are used for decomposition is new and developed especially for LUTSYN (Section 4.3.3).
- A new short proof that the synthesis problem is NP-hard has been presented in Section 4.1.4.
- A C++ class has been developed that implements set systems and the operators defined on set systems. This class is general and can be used for all problems involving set systems (Chapter 5).
- A C++ program LUTSYN has been developed that implements the algorithms presented in this thesis. LUTSYN provides an experimental environment which is currently being used for the further research of general decompositions and for the development and testing of heuristic algorithms for the near optimal decompositional synthesis. The program is specially designed to provide a lot of internal information, so the program can be experimentally used for research purposes, and especially for analysis and fine-tuning of all heuristic parts of the algorithm (Chapter 5).

7.3 Future work

The future work has already been started: the continuation of the research presented in this thesis. The goal of this work is the further research of general decompositions with much more experimentation (which is enabled by the generated research environment) and the creation of effective and efficient heuristic algorithms so that LUTSYN environment can be really used for synthesizing large circuits using the general decomposition paradigm. In this research also the improvements suggested in Chapter 6 will be taken into account.

Finally, many ideas for further work with LUTSYN have came up in the past. Here are some of them:

1. The set system representation of the Boolean function is obtained from any of a large number of cube representations of the function. The influence of the choice of the cube representation on the decomposition results should be investigated more closely. Especially, it is interesting to know if it is possible to determine the best cube representation for a function.

2. LUTSYN has been developed for FPGAs that use one type of cells: n-input, m-output look-up tables. It is interesting to investigate how LUTSYN can be extended to a broader class of FPGA architectures. It is believed that it is fairly simple to extend LUTSYN for the use with the FPGA architectures of Actel [Act95] and Atmel [Atm95].

3. In Section 5.6.2 the framework for the heuristic decision algorithm of LUTSYN is presented. The last two steps of this algorithm are *select()* (that selects a number of potential good items) and *add_essential()* that adds items that are necessary for the construction but that are forgotten by *select()*. The reason for this order is internal speed optimizations in LUTSYN (for example certain lists of items are automatically sorted because set systems are selected in order of their similarity). It would however be interesting to experiment with switching the order of these two blocks, such that all incompatibility pairs of the output set system are selected at least once by *add_essential()* and then using *select()* to add other good items to the list. Such an algorithm will be slower, but the results might be better.

4. LUTSYN uses 9 different parameters that tune different parts of the heuristic algorithm. Although a setting has been found that works fairly well, the influence of these parameters should be more closely investigated.

5. The blocks of some FPGA based architectures have more capabilities then the simple look up table. These capabilities can be used in the implementation of circuits. For example, the logic blocks in the Xilinx FPGAs include a large number of registers that are unused by the current LUTSYN version. These registers can be used to create a pipeline architecture, thus increasing the speed of the combinational machine at no extra cost. Another possible way of using these registers is the creation of a scan path to improve the testability of the design.

6. In LUTSYN the topology of the FPGA is modeled by a network of LUT blocks. This model is inaccurate. It is possible (however not much probable, because LUTSYN does a lot for good routability) that LUTSYN finds a LUT network that does not exceed the number of LUTs in the FPGA, but cannot be implemented

into the FPGA because of routing issues. By giving LUTSYN a better knowledge of the topology of the FPGA, or by adding a placement-and-routing algorithm to LUTSYN this problem can be solved.

Bibliography

[ABS+91] P. Abouzeid, L. Bouchet, K. Sakouti, G. Saucier, and P. Sicard. Lexicographical expression of Boolean function for multilevel synthesis of high speed circuits. In R.W. Dutton, editor, *VLSI Logic Synthesis and Design*, pages 31–39, Kyoto, Japan, 1991. IOS, Amsterdam, the Netherlands.

[Act95] *FPGA Data Book and Design Guide.* Actel Corporation, 1995.

[Ake78] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[Alt95] *Data Book.* Altera Corporation, March 1995.

[AMD95] *Mach 1,2,3 and 4 Family Data Book - High density EE CMOS Programmable Logic.* Advanced Micro Devices, incorporated, 1995.

[Ash59] R.L. Ashenhurst. The decomposition of switching functions. In *Proceedings International Symposium on the Theory of Switching Functions*, pages 74–116, April 1959.

[ASP91] P. Abouzeid, G. Saucier, and F. Poirot. Lexicographical factorization minimizing the critical path and the routing factor of multilevel logic. In P. Michel and G. Saucier, editors, *Logic and Architecture Synthesis. Proceedings of the IFIP TC10/WG10.5 Workshop*, pages 219–228, Paris, France, 1991. North-Holland, Amsterdam, the Netherlands.

[ASSP90] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. Multilevel synthesis minimizing the routing factor. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 365–368, Orlando, Florida, June 1990. IEEE, New York, USA.

[Atm95] *Configurable Logic Design and Application Book.* Atmel Corporation, San Jose, USA, August 1995.

[BBCS92] T. Besson, H. Bouzouzou, M. Crastes, and G. Saucier. Synthesis on multiplexer-based programmable devices using (ordered) binary decision diagrams. In *Proceedings EURO ASIC '92*, pages 8–13, Paris, France, June 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[BCH$^+$82] R.K. Brayton, J.D. Cohen, G.D. Hachtel, B.M. Trager, and Y.Y. Yun. Fast recursive Boolean function manipulation. In *Proceedings 1982 International Symposium of Circuits and Systems*, volume 1, pages 58–62, Rome, Italy, May 1982.

[BFRV92] S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic. *Field-Programmable Gate Arrays*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1992.

[BHMS84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1984.

[BHS90] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[BM82] R.K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proceedings 1982 International Symposium of Circuits and Systems*, volume 1, pages 49–54, Rome, Italy, May 1982.

[Bos94] W.F.A. Bosch. Implementation of an OR-BDD based TPG algorithm for combinational circuits. Master thesis report EB 496, Eindhoven University of Technology, Faculty of Electrical Engineering, Digital Systems Group, Eindhoven, the Netherlands, February 1994.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proceeding 27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, USA, June 1990. IEEE, New York, NY, USA.

[BRSW87] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. School of Computer Science technical report CMU-CS-92-160, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, July 1992.

[BS93] D. Brand and T. Sasao. Minimization of and-exor expressions using rewrite rules. *IEEE Transactions on Computers*, 42(5):568–576, May 1993.

[CJZT91] N. Calazans, R. Jacobi, Q. Zhang, and C. Trullemans. Improving BDDs manipulation through incremental reduction and enhanced heuristics. In *The European Conference on Design Automation. Proceedings*, pages 11.3.1 – 11.3.5, San Diego, California, USA, May 1991. IEEE, New York, NY, USA.

[Cou94] O. Coudert. Two-level logic minimization: an overview. *the VLSI journal*, 17(2):97–140, October 1994.

[CSD91] M.J. Ciesielski, J. Shen, and M. Davio. A unified approach to input-output encoding for FSM state assignment. In *28th ACM/IEEE Design Automation Conference, Proceedings 1991*, pages 176–181, San Francisco, California, USA, June 1991. ACM, New York, USA.

[CSS91] M. Crastes, K. Sakouti, and G. Saucier. A technology maping method based on perfect and semi-perfect matchings. In *28th ACM/IEEE Design Automation Conference. Proceedings 1991*, pages 93–98, San Francisco, California, USA, June 1991. ACM, New York, USA.

[Cur61] H.A. Curtis. A generalized tree circuit. *Journal of the Assocation for Computing Machinery*, 8:484–496, 1961.

[CY92] M.J. Ciesielski and S. Yang. PLADE: A two-stage PLA decomposition. *IEEE Transactions on Computer-Aided Design*, 11(8):943–954, August 1992.

[CYP89] M. Ciesielski, S. Yang, and M.A. Perkowski. Multiple-valued Boolean minimization based on graph coloring. In *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 262–265, Cambridge, Cambridge, Massachusetts, October 1989. IEEE Computer Society Press, Washington, DC, USA.

[Cyp94] *Programmable Logic Data Book 1994/1995*. Cypress Semiconductor, San Jose, USA, July 1994.

[CZJ+92] N. Calazans, Q. Zhang, R. Jacobi, B. Yernaux, and A. Trullemans. Advanced ordering and manipulation techniques for binary decision diagrams. In *The European Conference on Design Automation Proceedings*, pages 452–457, Brussels, Belgium, March 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[DAR86] M.R. Dagenais, V.K. Agarwal, and N.C. Rumin. McBoole: A new procedure for exact logic minimization. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):229–238, January 1986.

[dB95] F. de Bruyn. Multiple output minimizing algorithms for the logic synthesis shell. Graduation report EB-606, Eindhoven University of Technology, Department of Electrical Engineering, Section of Digital Information Systems, Eindhoven, the Netherlands, December 1995.

[DN91] S. Devadas and A.R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):13–27, January 1991.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *31st Design Automation Conference. Proceedings 1994*, pages 415–419, San Diego, California, USA, June 1994. ACM, New York, USA.

[FRC90] R.J. Francis, J. Rose, and K. Chung. Chortle: A technology mapping program for lookup table-based field programmabl gate arrays. In *27th ACM/IEEE Design Automation Conference. Proceedings 1990*, pages 613–619, Orlando, Florida, USA, June 1990. IEEE, New York, New York, USA.

[FRV91a] R. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *28th ACM/IEEE Design Automation Conference. Proceedings 1991*, pages 227–233, San Francisco, California, USA, June 1991. ACM, New York, New York, USA.

[FRV91b] R. Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGAs for performance. In *1991 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 568–571, Santa Clara, California, USA, November 1991. IEEE Computer Society Press, Los Alamitos, California, USA.

[FS94] A.H. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, November 1994.

[FSP92] B.J. Falkowski, I. Schäfer, and M.A. Perkowski. Effective computer methods for the calculation of rademacher-walsh spectrum for completely and incompletely specified Boolean functions. *IEEE Transactions on Computer-Aided Design*, 11(10):1207–1226, October 1992.

[GHB93] J. Greene, E. Hamdy, and S. Beal. Antifuse field programmable gate arrays. *Proceedings of the IEEE*, 81(7):1042–1056, July 1993.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability, a guide to the theory of NP-completeness*. A series of books in the mathematical sciences. W.H. Freeman and company, New York, 1979.

[GM93] J. Gergov and C. Meinel. Analysis and manipulation of Boolean functions in terms of decision graphs. In E.W. Mayer, editor, *Graph-Theoretic Concepts in Computer Science*, number 657 in Lecture Notes in Computer Science, pages 310–320. Springer-Verlag, Berlin, Germany, 1993.

[HHC92] Z. Hasan, D. Harrison, and M. Ciecielski. A fast partitioning method for pla-based FPGAs. *IEEE Design & Test of Computers*, 9(4):34–39, December 1992.

[HOI90] T. Hwang, R.M. Owens, and M. Irwin. Exploiting communication complexity for multilevel logic synthesis. *IEEE Transactions on Computer-Aided Design*, 9(10):1017–1027, October 1990.

[HOI92] T. Hwang, R.M. Owens, and M. Irwin. Efficiently computing communication complexity for multilevel logic synthesis. *IEEE Transactions on Computer-Aided Design*, 11(5):545–554, May 1992.

[HOIW94] T. Hwang, R.M. Owens, M. Irwin, and K. Wang. Logic synthesis for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(10):1280–1287, October 1994.

[HP94] P. Ho and M.A. Perkowski. Free kronecker decision diagrams and their application to atmel 6000 FPGA mapping. In *Proceedings EURO-DAC 94 with EURO-VHDL '94*, pages 8–13, Grenoble, France, September 1994. ACM, New York, USA.

[HS66] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentince-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1966.

[JB62] E.J. McCluskey Jr. and T.C. Bartee. *A Survey of Switching Circuit Theory*. McGraw-Hill Book Company, inc., 1962.

[JK91] L. Jóźwiak and J.C. Kolsteren. An efficient method for sequential general decomposition of sequential machines. *The EUROMICRO Journal on Microprocessing and Microprogramming*, 32(1-5):657–664, August 1991.

[Jóź89] L. Jóźwiak. The bit full-decomposition of sequential machines. Eindhoven University of Technology Research Reports 89-E-223, Eindhoven University of Technology, Faculty of Electrical Engineering, Eindhoven, the Netherlands, May 1989.

[Jóź95a] L. Jóźwiak. General decomposition and its use in digital circuit synthesis. *VLSI Design*, 3(3-4):225–248, 1995.

[Jóź95b] L. Jóźwiak. Logic synthesis. In *EUROMICRO 95, Design of Hardware/Software Systems*, pages 6–7, Como, Italy, September 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[Jóź95c] L. Jóźwiak. Modern concepts of quality and their relations to model libraries. In *Workshop on Libraries, Component Modeling and Quality Assurance*, pages 97–116, Nantes, France, April 1995. IFIP WG/ESPRIT.

[Jóź96a] L. Jóźwiak. Modern concepts of quality and their relationship to design reuse and model libraries. In O. Levia J-M. Bergé and J. Rouillard, editors, *Hardware Component Modeling*, Current Issues in Electronic Modeling, pages 107–130. Kluwer Academic Publishers, Boston, USA, 1996.

[Jóź96b] L. Jóźwiak. Quality-driven design space exploration in electronic system design. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, pages 1049–1054, Warsaw, Poland, June 1996. IESE, IEEE, New York, New York, USA.

[Jóż97a] L. Jóźwiak. Information relationships and measures: An analysis apparatus for efficient information system synthesis. In *23rd Euromicro Conference*, Budapest, Hungary, September 1997. To appear.

[Jóż97b] L. Jóźwiak. On the use of term trees for effective and efficient test pattern generation. In *23rd Euromicro Conference*, Budapest, Hungary, September 1997. To appear.

[JP95] L. Jóźwiak and A. Postula. Automatic hardware synthesis with FPGAs, tutorial accompaying the easa/ieee conference on electronic technology directions, May 1995.

[JS90] L. Jóźwiak and R. Specker. Crafal: Algorithms for Critical RAce Free Assignment of Level mode sequential circuits. Internal report, Eindhoven Universtity of Technology, Faculty of Electrical Engineering, Eindhoven, the Netherlands, April 1990.

[JV92] L. Jóźwiak and F. Volf. An efficient method for decomposition of multiple-output Boolean functions and assigned sequential machines. In *Proceedings The European Conference on Design Automation*, pages 114–122, Brussels, Belgium, March 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[JV95] L. Jóźwiak and F. Volf. Efficient decomposition of assigned sequential machines and boolean functions for PLD implementations. In *EASA/IEEE Conference on Electronic Technology Directions*, pages 258–266, Adelaide, Australia, May 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[JvD92] L. Jóźwiak and A.P.H. van Dijk. A method for general simultaneous full decomposition of sequential machines: Algorithms and implementation. EUT Report 92-E-267, Eindhoven University of Technology, Faculty of Electrical Engineering, Eindhoven, the Netherlands, December 1992.

[Kar53] M. Karnaugh. A map method for synthesis of combinational logic circuits. *Transactions American Institute of Electrical Engineers*, 72:593–599, 1953.

[Kar63] R.M. Karp. Function decomposition and switching circuit design. *Journal Society of Industrial Applied Mathematics*, 11(2):291–335, June 1963.

[Kar88a] K. Karplus. Representing Boolean functions with if-then-else DAGs. Baskin Center for Computer Engineer & Information Sciences technical report UCSC-CRL-88-28, University of California, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, November 1988.

[Kar88b] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. Baskin Center for Computer Engineer & Information Sciences technical report UCSC-CRL-88-29, University of California, Baskin Center for Computer

Engineering & Information Sciences, University of California, Santa Cruz, November 1988.

[Kar90] K. Karplus. Using if-then-else DAGs to do technology mapping for field-programmable gate arays. Baskin Center for Computer Engineer & Information Sciences technical report UCSC-CRL-90-43, University of California, Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz, September 1990.

[Kar91a] K. Karplus. Amap: a technology mapper for selector-based field-programmable gate arrays. In *28th ACM/IEEE Design Automation Conference. Proceedings 1991*, pages 244–247, San Francisco, California, USA, June 1991. ACM, New York, USA.

[Kar91b] K. Karplus. Amap: a technology mapper for table-lookup field-programmable gate arrays. In *28th ACM/IEEE Design Automation Conference. Proceedings 1991*, pages 240–243, San Francisco, California, USA, June 1991. ACM, New York, USA.

[Kar92] K. Karplus. ITEM: an if-then-else minimizer for logic synthesis. In *EURO-ASIC 92 proceedings*, pages 2–7, Paris, France, June 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[KD91] G. Kim and D. Dietmeyer. Multilevel logic synthesis of symmetric switching functions. *IEEE Transactions on Computer-Aided Design*, 10(4):436–446, April 1991.

[KJ95] P.A. Konieczny and L. Jóźwiak. Minimal input support problem and algorithms to solve it. EUT Report 95-E-289, Eindhoven University of Technology, Faculty of Electrical Engineering, Eindhoven, the Netherlands, April 1995.

[KN96] A. Kraśniewski and M. Nowicka. Application-dependent testability of FPGA-based circuits designed using functional decomposition. In *Proceedings of IFIP TC10/WG10.5 International Workshop on Logic and Architecture Synthesis*, pages 167–175, Grenoble, France, December 1996.

[Kon90] P. Konieczny. Implementation of a predictive dynamic beam search algorithm in a program for intelligent state assignment of sequential machines. Internal report, Eindhoven University of Technology, Section of Digital Information Systems, Eindhoven, the Netherlands, October 1990.

[Kon95] P.A. Konieczny. *General Decomposition of Sequential Machines - Algorithms and Programs*. AIO-EB 036. Eindhovense School voor Technologische Ontwerpen, Eindhoven, the Netherlands, June 1995.

[KSR92] U. Kebschull, E. Schubert, and W. Rosentiel. Multilevel logic synthesis based on functional decision diagrams. In *Proceedings. The European Conference on Design Automation*, pages 43–47, Brussels, Belgium, March 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[Lee59] C.Y. Lee. Representation of switching circuits by binary-decision diagrams. *The Bell System Technical Journal*, 38:985–999, July 1959.

[ŁKJ91] T. Łuba, J. Kalinowski, and K. Jasiński. PLATO - a CAD tool for logic synthesis based on decomposition. In *Proceedings of the European Conference on Design Automation*, pages 65–69, Amsterdam, the Netherlands, February 1991. IEEE Computer Society Press, Los Alamitos, California, USA.

[ŁKJK91] T. Łuba, J. Kalinowski, K. Jasiński, and A. Kraśniewski. Combining serial decomposition with topological partitioning for effective multi-level PLA implementations. In P. Michel and G. Saucier, editors, *Logic and Architecture Synthesis. Proceedings of the IFIP TC10/WG10.5 Workshop*, pages 243–252, Paris, France, 1991. North-Holland, Amsterdam, the Netherlands.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall Software Series. Prentice-Hall International, Inc., Englewood Cliffs, NJ 07632, 1981.

[LPP96] Y. Lai, K.R. Pan, and M. Pedram. OBDD-based function decomposition: Algorithms and implementatino. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):977–990, August 1996.

[ŁS95] T. Łuba and H. Selvaraj. A general approach to Boolean function decomposition and its application in FPGA-based synthesis. *VLSI Design*, 3(3-4):289–300, 1995.

[ŁSK93] T. Łuba, H. Selvaraj, and A. Kraśniewski. A new approach to FPGA-based logic synthesis. In *Workshop on Design Methodologies for Microelectronics and Signal Processing*, pages 135–142, October 1993.

[MBS84] G. De Micheli, R. Brayton, and A. Sanginvanni-Vincentelli. KISS: A program for optimal state assignment of finite state machines. In *IEEE International Conference of Computer-Aided Design. Digest of Technical Papers*, pages 209–211, Santa Clara, California, USA, November 1984. IEEE, New York, New York, USA.

[MBS85] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, CAD-4(3):269–285, July 1985.

[MBS92] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential synthesis for table look up FPGA's. In *Proceedings Euro ASIC '92*, pages 32–37, Paris, France, June 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[MF89] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 556–559, Santa Clara, California, USA, November 1989. IEEE Computer Society Press, Los Alamitos, California, USA.

[Mic86] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):597–616, October 1986.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill, Inc., Highstown, NJ, 1994.

[Mil66] R.E. Miller. *Switching Theory*, volume I: Combinational Circuits. John Wiley & Sons, Inc., New York, USA, second edition, March 1966.

[MKLC89] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney. The transduction method - design of logic networks based on permissible functions. *IEEE Transactions on Computers*, 38(10):1404–1424, October 1989.

[MLBS92] S. Malik, L. Lavagno, R.K. Brayton, and A. Sangiovanni-Vincentelli. Symbolic minimization of multilevel logic and the input encoding problem. *IEEE Transactions on Computer-Aided Design*, 11(7):825–843, July 1992.

[MNS$^+$90] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Proceedings 27th ACM/IEEE Design Automation Conference*, pages 620–625, Orlando, Florida, USA, June 1990. IEEE, New York, New York, USA.

[MSBS91a] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In *IEEE International Conference on Computer-Aided Design. Digest of Technical Papers 1991*, pages 564–567, Santa Clara, California, USA, November 1991. IEEE Computer Society Press, Los Alamitos, California, USA.

[MSBS91b] R. Murgai, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Performance directed synthesis for table look up programmable gate arrays. In *IEEE International Conference on Computer-Aided Design. Digest of Technical Papers 1991*, pages 572–575, Santa Clara, California, USA, November 1991. IEEE Computer Society Press, Los Alamitos, California, USA.

[MSBS93] P.C. McGeer, J.V. Sanghavi, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, 1(7):432–440, December 1993.

[MWBS88] S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *IEEE International Conference on Computer-Aided Design, ICCAD-89. Digest of Technical Papers*, pages 6–9, Santa Clare, California, USA, November 1988. IEEE Computer Society Press, Washington DC, USA.

[OCF93] J.C. Madre O. Coudert and H. Fraisse. A new viewpoint on two-level logic minimization. In *Proceedings 30th ACTM/IEEE Design Automation Conference*, pages 625–630, Dallas, Texas, USA, June 1993. ACM, New York, USA.

[PCSS95] M.A. Perkowski, M. Chrzanowska-Jeske, A. Sarabi, and I. Schäfer. Multi-level logic synthesis based on kronecker decision diagrams and Boolean ternary decision diagrams for incompleteley specified functions. *VLSI Design*, 3(3-4):301–313, 1995.

[Phi90] *Semi-custom Programmable Logic Devices (PLD).* Philips Components Division, 1990.

[RGS93] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate array. *Proceedings of the IEEE*, 81(7):1013–1029, July 1993.

[RK62] J.P. Roth and R.M. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(4):227–238, April 1962.

[RS87] R.L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on computer-aided design*, CAD-6(5):727–750, September 1987.

[Rud86] R.L. Rudell. *Multiple-valued logic minimization for PLA synthesis.* Ph.D. dissertation, University of California, Berkeley, California, USA, 1986. Memorandum no. UCB/ERL M86/65.

[RV90] J. Rajski and J. Vasudevamurthy. Testability preserving transformations in multi-level logic synthesis. In *Proceedings International Test Conference 1990*, pages 265–273, Washington DC, USA, 1990. IEEE Computer Society Press, Los Alamitos, California, USA.

[RV92] J. Rajski and J. Vasudevamurthy. The testability-preserving concurrent decomposition and factorization of Boolean expressions. *IEEE Transactions on Computer-Aided Design*, 11(6):778–793, June 1992.

[Sas81] T. Sasao. Mutiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30(9):635–643, September 1981.

[Sas84] T. Sasao. Input variable assignment and output phase optimization of PLA's. *IEEE Transactions on Computers*, C-33(10):879–894, October 1984.

[SB93] C. Shi and J.A. Brzozowski. An efficient algorithm for constrained encoding and its applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1813–1826, December 1993.

[SCS87] G. Saucier, M. Crastes, and P. Sicard. ASYL: A rule-based system for controllor synthesis. *IEEE Transactions on Computer-Aided Design*, 6(6):1088–1097, November 1987.

[SDG95] A. Shen, S. Devadas, and A. Ghosh. Probabilistic manipulation of Boolean functions using free Boolean diagrams. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):87–95, January 1995.

[SFA93] G. Saucier, J. Fron, and P. Abouzeid. Lexicographical expressions of Boolean functions with application to multilevel synthessis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(11):1642–1654, November 1993.

[SGR93] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose. Synthesis methods for field programmable gate arrays. *Proceedings of the IEEE*, 81(7):1057–1083, July 1993.

[SSA94] P.H. Schneider, U. Schlichtmann, and K.J. Antreich. A new power estimation technique with application to decomposition of Boolean functions for low power. In *Proceedings EURO-DAC'94 with EURO-VHDL'94*, pages 388–393, Grenoble, France, September 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[SSL+92] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Memorandum UCB/ERL M92/41, University of California, Department of Electrical Engineering and Computer Science, Berkeley, California, USA, May 1992.

[SVBS94] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Satisfaction of input and output encoding constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):589–602, May 1994.

[Tri93] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, July 1993.

[vH94] J.G.M. van Houtert. Espresso file format for the logic synthesis shell. Graduation report EB-547, Eindhoven University of Technology, Department of Electrical Engineering, Section of Digital Information Systems, Eindhoven, the Netherlands, November 1994.

[VJ95] F.A.M. Volf and L. Jóźwiak. Decompositional logic synthesis approach for look up table FPGAs. In W.A. Cook, R.A. Hull, and C. Traver, editors, *Proceedings Eighth Annual IEEE International ASIC Conference and Exhibit*, pages 358–361, Austin, Texas, September 1995. IEEE Computer Society Press.

[VJS95] F. Volf, L. Jóźwiak, and M. Stevens. Division-based versus general decomposition-based multiple-level logic synthesis. *VLSI Design*, 3(3-4):267–287, 1995.

[Vol91] F.A.M. Volf. DeCC - an implementation of a new method for the decomposition of Boolean functions and assigned sequential machines. Graduation report EB-291, Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, Eindhoven, the Netherlands, February 1991.

[VR90] J. Vasudevamurthy and J. Rajski. A method for concurrent decompositions and factorization of Boolean expressions. In *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 510–513, Santa Clara, California, USA, November 1990. IEEE Computer Society Press, Washington DC, USA.

[vW95] M.J.M. van Weert. Private communication, 1995.

[WP92] W. Wan and M.A. Perkowski. A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping. In *Proceedings EURO-DAC'92*, pages 230–235, Hamburg, Germany, September 1992. IEEE Computer Society Press.

[Xil93] *The Programmable Logic Data Book*. Xilinx Incorporated, 1993.

[XK96] M. Xu and F.J. Kurdahi. Area and timing estimation for lookup table based FPGAs. In *Proceedings of the European Design & Test Conference 1996*, pages 151–157, Paris, France, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[Yan91] S. Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. MCNC technical report, Microelectronics Center of North Carolina, Reseach Triangle Park, North Carolina, USA, January 1991.

[YC89] S. Yang and M.J. Ciesielski. PLA decomposition with generalized decoders. In *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 312–315, Santa Clara, California, USA, November 1989. IEEE Computer Society Press, Los Alamitos, California, USA.

[YC90] S. Yang and M.J. Ciesielski. On the relationship between input encoding and logic minimization. In *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Science*, volume 4, pages 377–386, Kailua-Kona, Hawaii, USA, January 1990. IEEE Computer Society Press, Los Alamitos, California, USA.

[YC91] S. Yang and M.J. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems*, 10(1):4–12, January 1991.

Appendix A

Proof of set system properties

The goal of this appendix is to prove some of the fundamental theorems introduced in Chapter 4. In the first part of this appendix (Section A.1) it will be proven that by defining the correct $\sqsubseteq$, $(Setsys(\simeq), \sqsubseteq)$, forms a lattice. The top of this lattice will be written $M_{\simeq}$ and is called the canonical set system representative of $Setsys(\simeq)$. In Section A.2 the $\leq$ operator on canonical set systems will be defined and it will be proved that this definition is equivalent to the definition used for non-canonical set systems. The last section (Section A.3) is devoted to the proof of the theorem that states that the set system product of two canonical set systems results in a canonical set system.

A.1 Properties of set system sets

A.1.1 Introduction

The following definitions are copied from Section 4.2.2 for the sake of easy reference. For the interpretation of these definitions the reader is referred to Section 4.2.2.

Definition A.1 *Compatibility relation*
Let V be a finite non-empty set of symbols. A binary relation $\simeq$ is called a compatibility relation on V if and only if it is reflexive and symmetric, i.e.:

$$\simeq \subseteq V \otimes V \tag{A.1}$$

such that

$$\underset{v \in V}{\forall} (v, v) \in \simeq \tag{A.2}$$

and

$$\underset{(v,w) \in \simeq}{\forall} (w, v) \in \simeq \tag{A.3}$$

□

The relation $\simeq$ is called the *compatibility operator* and a pair of compatible symbols is called a *compatibility pair*. The notation $a \simeq b$ will be used to denote that $(a, b) \in \simeq$.

Definition A.2 *Compatible block*
A set $b \subseteq V$ is called a compatible block with respect to compatibility relation $\simeq \subseteq V \otimes V$ (denoted as $CB_{\simeq}(b)$) if and only if all pairs of symbols in b are compatible:

$$CB_{\simeq}(b) \Leftrightarrow \forall_{v,w \in b} \; v \simeq w \tag{A.4}$$

□

Definition A.3 *Maximal compatible block*
A set $b \subseteq V$ is called a maximal compatible block with respect to compatibility relation $\simeq \subseteq V \otimes V$ (denoted as $MB_{\simeq}(b)$) if and only if the block is compatible and no other symbol from V is compatible with all the symbols in b:

$$MB_{\simeq}(b) \Leftrightarrow CB_{\simeq}(b) \wedge \forall_{v \in V \setminus b} \; \exists_{w \in b} \; v \not\simeq w \tag{A.5}$$

□

Definition A.4 *Extendible block*
A set $b \subseteq V$ is called an extendible block with respect to compatibility relation $\simeq \subseteq V \otimes V$ (denoted as $EB_{\simeq}(b)$) if and only if a maximal compatible block can be found that contains b:

$$EB_{\simeq}(b) \Leftrightarrow \exists_{b' \subseteq V} \; b \subseteq b' \wedge MB_{\simeq}(b') \tag{A.6}$$

□

Definition A.5 *Set system*
Let V be a finite non-empty set of elements. A set system $\phi_{\simeq} : \phi_{\simeq} \subseteq 2^V$ is a representation for a compatibility relation $\simeq$ defined on V if and only if the following conditions are met:

1. The subsets are compatible:
$$\forall_{b \in \phi_{\simeq}} \; CB_{\simeq}(b) \tag{A.7}$$

2. The set system is non-redundant:
$$\forall_{b, b' \in \phi_{\simeq}} \; b \subseteq b' \Rightarrow b = b' \tag{A.8}$$

3. All compatibility pairs of $\simeq$ are present in the set system:
$$\forall_{(v,w) \in \simeq} \; \exists_{b \in \phi_{\simeq}} \; \{v, w\} \subseteq b \tag{A.9}$$

□

Definition A.6 *Set system set Setsys*$(\simeq)$
Setsys$(\simeq)$ is the set of all set systems that represent compatibility relation $\simeq$. □

In Section 4.2.2 the following theorem has been introduced:

Theorem A.1 *$(Setsys(\simeq), \sqsubseteq)$ is a lattice*
Let V be a finite non-empty set of symbols. Let $\simeq \subseteq V \otimes V$ be a compatibility relation defined on V. Define $\sqsubseteq$ as:

$$\underset{S,T \in Setsys(\simeq)}{\forall} \quad S \sqsubseteq T \text{ if and only if } \underset{b \in S}{\forall} \ \underset{b' \in T}{\exists} \ b \subseteq b' \tag{A.10}$$

then $(Setsys(\simeq), \sqsubseteq)$ forms a lattice and $M_{\simeq}$ and $m_{\simeq}$ defined as

$$M_{\simeq} = \left\{ b \subseteq 2^V \mid MB_{\simeq}(b) \right\} \tag{A.11}$$

and

$$m_{\simeq} = \{\{v, w\} \in V \otimes V \mid v \simeq w \wedge v \neq w\} \cup \left\{ \{v\} \in 2^V \mid \underset{w \in V \setminus \{v\}}{\forall} \ w \not\simeq v \right\} \tag{A.12}$$

form the top respectively the bottom of the lattice. Because $M_{\simeq}$ is the most compact set system in $Setsys(\simeq)$, it is often referred to as *the canonical representative* of $Setsys(\simeq)$. □

This section is devoted to the proof of this theorem. The proof consists of three parts. First, it will be proved that $(Setsys(\simeq), \sqsubseteq)$ is a lattice (Section A.1.2). Secondly, it will be proved that $M_{\simeq}$ is the top of this lattice (Section A.1.3). Finally, in Section A.1.4 it will be proved that $m_{\simeq}$ is the bottom of this lattice. In these sections, the following theorem is used:

Theorem A.2 *Non-redundant blocks*
Let V be a finite non-empty set of symbols. For $B \subseteq 2^V$ define the set of non-redundant blocks of B (denoted as $NR(B)$) by

$$NR(B) = \left\{ b \in B \mid \underset{b' \in B}{\forall} \ b \subseteq b' \Rightarrow b = b' \right\} \tag{A.13}$$

Then

$$\underset{B \subseteq 2^V}{\forall} \ \underset{b \subseteq V}{\forall} \left\{ \left(\underset{b' \in B}{\exists} \ b \subseteq b' \right) \Rightarrow \left(\underset{b'' \in NR(B)}{\exists} \ b \subseteq b'' \right) \right\} \tag{A.14}$$

□

Proof:
The proof is by mathematical induction on $n = \#(B)$. The induction predicate is defined as:

$$P(n) = \left(\underset{B \subseteq 2^V}{\forall} \ \#(B) = n \right) \Rightarrow \left(\underset{b \subseteq V}{\forall} \left\{ \left(\underset{b' \in B}{\exists} \ b \subseteq b' \right) \Rightarrow \left(\underset{b'' \in NR(B)}{\exists} \ b \subseteq b'' \right) \right\} \right) \tag{A.15}$$

- **Base :** $P(0)$
  Let $B \subseteq 2^V$ such that $\#(B) = n = 0$. Then $B = \varnothing$. Let $b \subseteq V$. Then
  $$\underset{b' \in B}{\exists} \; b \subseteq b'$$
  is false and thus
  $$\left(\underset{b' \in B}{\exists} \; b \subseteq b' \right) \Rightarrow \left(\underset{b'' \in NR(B)}{\exists} \; b \subseteq b'' \right)$$
  is true.

- **Induction :** $P(n) \Rightarrow P(n+1)$
  Let $b \subseteq 2^V$ such that $\#(B) = n+1$. Let $b \subseteq V$ and assume that for some $b' \in B : b \subseteq b'$. Fix such a b'. If $n = 0$ then $NR(B) = B$ and hence for some $b'' \in NR(B) : b \subseteq b''$ holds. If $n > 0$ define $B' = B \setminus \{b''\}$, where $b'' \in B \setminus \{b'\}$. Then $\#(B') = n$ and since $b' \in B'$ and $b \subseteq b'$ we have that
  $$\underset{b' \in B'}{\exists} \; b \subseteq b'$$
  By induction we then also have
  $$\underset{c \in NR(B')}{\exists} \; b \subseteq c$$
  Fix such a c. If
  $$\underset{d \in B}{\forall} \; c \subseteq d \Rightarrow c = d$$
  then $c \in NR(B)$ and then
  $$\underset{b'' \in NR(B)}{\exists} \; b \subseteq b''$$
  holds. If
  $$\neg(\underset{d \in B}{\forall} \; c \subseteq d \Rightarrow c = d)$$
  then for some $d \in B, c \subset d$. Then $d \notin B'$ because otherwise $c \notin NR(B')$, which would be a contradiction. Hence $d = b''$. But then $b'' \in NR(B)$ because otherwise there would be an $e \in B$ with $b'' \subset e$. But then $e \in B', c \subset e$ and thus $c \notin NR(B')$, which is a contradiction. Hence
  $$\underset{b'' \in NR(B)}{\exists} \; b \subseteq b''$$

We thus have
$$\underset{n \in \mathbb{N}}{\forall} \; P(n)$$
Now let $B \subseteq 2^V$ then for some $n \in \mathbb{N}$, $\#(B) = n$. Since $P(n)$ holds we have
$$\underset{b \subseteq V}{\forall} \left\{ \left(\underset{b' \in B}{\exists} \; b \subseteq b' \right) \Rightarrow \left(\underset{b'' \in NR(B)}{\exists} \; b \subseteq b'' \right) \right\}$$
and we have (A.14). This ends the proof of this theorem. □

A.1.2 Set systems form a lattice

Theorem A.3 *$\sqsubseteq$ is a partial order on $Setsys(\simeq)$*

□

Proof:
To prove that relation $\sqsubseteq$ is a partial order it has to be shown that the relation is reflexive, anti-symmetric and transitive. The proofs of the reflexivity and transitivity properties are straightforward, so only the property of anti-symmetry is outlined here. Let $\phi_{\simeq}, \psi_{\simeq} \in Setsys(\simeq)$, let $\phi_{\simeq} \sqsubseteq \psi_{\simeq}$, $\psi_{\simeq} \sqsubseteq \phi_{\simeq}$ and let $b \in \phi_{\simeq}$. Then

$$\underset{b' \in \psi_{\simeq}}{\exists} \; b \subseteq b' \tag{A.16}$$

and by fixing such a b'

$$\underset{b'' \in \phi_{\simeq}}{\exists} \; b' \subseteq b'' \tag{A.17}$$

Fix b'', then based on (A.16) and (A.17) it can be concluded that

$$b \subseteq b' \subseteq b'' \tag{A.18}$$

By definition of $Setsys(\simeq)$ one has that $b \subseteq b'' \Rightarrow b = b''$. Therefore $b = b'$ and thus $b \in \psi_{\simeq}$. So for each $b \in \phi_{\simeq}$ we have that also $b \in \psi_{\simeq}$. Hence $\phi_{\simeq} \subseteq \psi_{\simeq}$. By a symmetric argument it can be shown that $\psi_{\simeq} \subseteq \phi_{\simeq}$, so $\phi_{\simeq} = \psi_{\simeq}$. □

Definition A.7 *Block collection of a set system set*
The block collection $\cup X$ of a set of set systems $X \subseteq Setsys(\simeq)$ consists of all the set system blocks of the set systems in X:

$$\cup X = \left\{ b \mid \underset{\phi_{\simeq} \in X}{\exists} \; b \in \phi_{\simeq} \right\} \tag{A.19}$$

□

Theorem A.4 *Least upper bounds of $(Setsys(\simeq), \sqsubseteq)$*
Let $X \subseteq Setsys(\simeq)$ be a non-empty set of set systems. Then $\sqcup X$ defined as

$$\sqcup X = \left\{ b \in \cup X \mid \underset{b' \in \cup X}{\forall} \; b \subseteq b' \Rightarrow b = b' \right\} \tag{A.20}$$

is the least upper bound of X. It is further contained in $Setsys(\simeq)$. □

Proof:
To prove the least upper bound, three properties must be checked:

1. The least upper bound operator is closed:

$$\sqcup X \in Setsys(\simeq) \tag{A.21}$$

2. It is an upper bound:

$$\forall_{\phi_\simeq \in X} \ \phi_\simeq \sqsubseteq \sqcup X \tag{A.22}$$

3. $\sqcup X$ is the smallest of all upper bounds:

$$\forall_{\phi_\simeq \in Setsys(\simeq)} \left(\left(\forall_{\psi_\simeq \in X} \ \psi_\simeq \sqsubseteq \phi_\simeq \right) \Rightarrow \sqcup X \sqsubseteq \phi_\simeq \right) \tag{A.23}$$

(A.21) will be proved first. It has to be shown that $\sqcup X$ satisfies the definition of $Setsys(\simeq)$. It will first be shown that the blocks of $\sqcup X$ are non-redundant. Let $b_1, b_2 \in \sqcup X$ and let $b_1 \subseteq b_2$. By definition of $\sqcup X$:

$$\forall_{b' \in \cup X} \ b_1 \subseteq b' \Rightarrow b_1 = b' \tag{A.24}$$

Since, $b_2 \in \sqcup X \subseteq \cup X$ it follows $b_1 = b_2$. Next, it should be shown that all the blocks of $\sqcup X$ are compatible. This is trivial because the blocks of $\sqcup X$ are blocks of set systems from $Setsys(\simeq)$ and the blocks of $Setsys(\simeq)$ are compatible by definition. Finally, it should be shown that each collection $\{v, w\}$ for which $v \simeq w$, is contained in some block of $\sqcup X$. To this end let $v \simeq w$. Since $X \neq \varnothing$ and since (by definition of $Setsys(\simeq)$) each set system in X contains a block b for which $v, w \in b$, we have that

$$\exists_{b \in \cup X} \ \{v, w\} \subseteq b$$

But then we also have

$$\exists_{b \in \sqcup X} \ \{v, w\} \subseteq b$$

by applying Equation (A.14). This concludes the proof of $\sqcup X \in Setsys(\simeq)$.

To prove (A.22) let $\phi_\simeq \in X$ and let $b \in \phi_\simeq$. Then by definition of $\sqcup X$ it follows that there exists a block $c \in \sqcup X$, such that $b \subseteq c$. Hence $\phi_\simeq \sqsubseteq \sqcup X$ and thus (A.22) holds.

The third part of the proof consists of proving (A.23). Let $\phi_\simeq \in Setsys(\simeq)$ be an upper bound of set X, i.e. let $\phi_\simeq$ be such that

$$\forall_{\phi'_\simeq \in X} \ \phi'_\simeq \sqsubseteq \phi_\simeq \tag{A.25}$$

Assume $b \in \sqcup X$. Then by definition of $\sqcup X$ a set system $\phi''_\simeq \in X$ exists such that $b \in \phi''_\simeq$. Since $\phi''_\simeq \sqsubseteq \phi_\simeq$ there exists a block $b' \in \phi_\simeq$ such that $b \subseteq b'$. Hence

$$\forall_{b \in \sqcup X} \ \exists_{b' \in \phi_\simeq} \ b \subseteq b' \tag{A.26}$$

and thus $\sqcup X \sqsubseteq \phi_\simeq$. This concludes the proof of (A.23). □

Definition A.8 *Block intersection of a set system set*
The block intersection $\cap X$ of a set of set systems $X \subseteq Setsys(\simeq)$ is defined as:

$$\cap X = \left\{ b \mid \underset{\phi_{\simeq} \in X}{\forall} \; \underset{b' \in \phi_{\simeq}}{\exists} \; b \subseteq b' \right\} \tag{A.27}$$

□

Theorem A.5 *Greatest lower bounds of* $(Setsys(\simeq), \sqsubseteq)$
Let $X \subseteq Setsys(\simeq)$ be a non-empty set of set systems. Then $\sqcap X$ defined as

$$\sqcap X = \left\{ b \in \cap X \mid \underset{b' \in \cap X}{\forall} \; b \subseteq b' \Rightarrow b = b' \right\} \tag{A.28}$$

is the greatest lower bound of X. It is further contained in $Setsys(\simeq)$. □

Proof:
To prove the greatest lower bound, three properties must be checked:

1. The greatest lower bound operator is closed:

$$\sqcap X \in Setsys(\simeq) \tag{A.29}$$

2. $\sqcap X$ is a lower bound:

$$\underset{\phi_{\simeq} \in X}{\forall} \; \sqcap X \sqsubseteq \phi_{\simeq} \tag{A.30}$$

3. $\sqcap X$ is the largest of all lower bounds:

$$\underset{\phi_{\simeq} \in Setsys(\simeq)}{\forall} \left(\underset{\phi'_{\simeq} \in X}{\forall} \; \phi_{\simeq} \sqsubseteq \phi'_{\simeq} \right) \Rightarrow \phi_{\simeq} \sqsubseteq \sqcap X \tag{A.31}$$

(A.29) will be proved first. It has to be shown that $\sqcap X$ satisfies the definition of $Setsys(\simeq)$. The proofs of (A.7) and (A.8) are straightforward. (A.9) remains to be proved. Let $v \simeq w$. Then for each $\phi_{\simeq} \in X$ there exists a block $b \in \phi_{\simeq}$ such that $\{v, w\} \subseteq b$. Hence $\{v, w\} \in \cap X$. But then by (A.14) there exists a block $b' \in \sqcap X$ such that $\{v, w\} \in b'$.

(A.30) will be proved next. Let $\phi_{\simeq} \in X$ and let $b \in \sqcap X$. Then $b \in \cap X$. But then there exists a $b' \in \phi_{\simeq}$ such that $b \subseteq b'$. Hence $\sqcap X \sqsubseteq \phi_{\simeq}$.

Finally, we will prove (A.31). Let $\phi_{\simeq}$ be a lower bound of X. It has to be shown that $\phi_{\simeq} \sqsubseteq \sqcap X$. Let $b \in \phi_{\simeq}$. Then for each $\phi'_{\simeq} \in X$ there exists a $b' \in \phi'_{\simeq}$ such that $b \subseteq b'$. But then by definition of $\cap X$, $b \in \cap X$. Hence, by (A.14), for some $c \in \sqcap X, b \subseteq c$. We thus have $\phi_{\simeq} \sqsubseteq \sqcap X$. □

Theorem A.6 $(Setsys(\simeq), \sqsubseteq)$ *is a lattice*
$(Setsys(\simeq), \sqsubseteq)$ is a lattice, $\sqcup Setsys(\simeq)$ is the maximum element and $\sqcap Setsys(\simeq)$ it the minimum element of this lattice. □

Proof:
A lattice is defined as a partial ordered set in which each non-empty set of elements has an upper bound and a lower bound. It has been proved before that $\sqsubseteq$ is a partial order and that $\sqcup X$ is the least upper bound of a non-empty set $X \in Setsys(\simeq)$, and that $\sqcap X$ is the greatest upper bound. It has also been proved that $\sqcup X, \sqcap X \in Setsys(\simeq)$. We thus have that $\sqcup Setsys(\simeq)$ is the maximum element of the lattice and that $\sqcap Setsys(\simeq)$ is the minimum element of the lattice. □

This concludes the first part of the proof of theorem A.1.

A.1.3 $M_{\simeq}$ is the maximum of $(Setsys(\simeq), \sqsubseteq)$

The goal of this section is to show that $M_{\simeq}$ is the maximum of $(Setsys(\simeq), \sqsubseteq)$. Before this theorem can be proved it is first necessary to prove another theorem.

Theorem A.7 *Existence of maximal compatible block*
Let $b \subseteq V$ be a set system block. Then

$$CB_{\simeq}(b) \Rightarrow EB_{\simeq}(b) \qquad (A.32)$$

□

Proof:
The proof is by mathematical induction on $n = \#(V \setminus b)$. The induction predicate is defined by:

$$P(n) = \forall_{b \in V} \#(V \setminus b) = n \wedge CB_{\simeq}(b) \Rightarrow EB_{\simeq}(b) \qquad (A.33)$$

- **Base :** $P(0)$
  If $\#(V \setminus b) = 0$ then $b = V$. Assume $CB_{\simeq}(b)$ holds. Then trivially there exists no $c \in V \setminus V = \varnothing$ such that $v \simeq w$ for all $w \in V$ and thus $MB_{\simeq}(b)$ holds. Since $b \subseteq b$, $EB_{\simeq}(b)$ must also hold.

- **Induction :** $P(n) \Rightarrow P(n+1)$
  Let $b \subseteq V$ such that

  $$\#(V \setminus b) = n + 1 \wedge CB_{\simeq}(b) \qquad (A.34)$$

  Suppose $MB_{\simeq}(b)$. Then $EB_{\simeq}(b)$ holds trivially. Suppose $\neg MB_{\simeq}(b)$. Then

  $$\exists_{x \in V \setminus b} \forall_{w \in b} x \simeq w \qquad (A.35)$$

  Fix such an x. Let block $b' = b \cup \{x\}$. Then

  $$\#(V \setminus b') = \#(V \setminus b \cup \{x\}) = \#(V \setminus b) - 1 = n \qquad (A.36)$$

  Since $CB_{\simeq}(b')$ holds because of (A.35) and $CB_{\simeq}(b)$, $EB_{\simeq}(b')$ can be concluded using the induction hypothesis. Hence, there exists a $b'' \subseteq V$ such that $b' \subseteq b''$ and $MB_{\simeq}(b'')$. But since $b \subseteq b''$ and $MB_{\simeq}(b'')$ we then also have $EB_{\simeq}(b)$.

We thus have that

$$\forall_{n \in \mathbb{N}} \forall_{b \in V} \#(V \setminus b) = n \wedge CB_{\simeq}(b) \Rightarrow EB_{\simeq}(b)$$

Now let $b \subseteq V$ such that $CB_{\simeq}(b)$. Then for some $n \in \mathbb{N}, \#(V \setminus \{b\}) = n$ and thus $EB_{\simeq}(b)$. Hence for each $b \subseteq V$ with $CB_{\simeq}(b), EB_{\simeq}(b)$ also holds. This concludes the proof of (A.32). □

Theorem A.8 *$\sqcup Setsys(\simeq)$ is the maximum of $Setsys(\simeq)$*

$$\sqcup\, Setsys(\simeq) = M_{\simeq} \tag{A.37}$$

□

Proof:
The proof consists of two parts:

1. $M_{\simeq}$ is a set system:
$$M_{\simeq} \in Setsys(\simeq) \tag{A.38}$$
2. $M_{\simeq}$ is the maximum:
$$\forall_{S \in Setsys(\simeq)} \quad S \sqsubseteq M_{\simeq} \tag{A.39}$$

Proof of (A.38):
It must be shown that $M_{\simeq}$ satisfies the Equations (A.7), (A.8) and (A.9). (A.7) is satisfied trivially because the blocks of $M_{\simeq}$ are maximal compatible. (A.8) can be proved as follows. Let $b_1, b_2 \in M_{\simeq}$ and let $b_1 \subseteq b_2$. Now assume $b_1 \subset b_2$. Then a symbol $p \in b_2$ can be found such that $p \notin b_1$. By definition of $MB_{\simeq}(b_1)$ we have

$$\forall_{u \in V \setminus b_1} \exists_{v \in b_1} u \not\simeq v \tag{A.40}$$

However, $p \in V \setminus b_1$ and $p \simeq v$ for all $v \in b_1$ (since $MB_{\simeq}(b_2)$ holds). This is a contradiction and therefore b_1 cannot be a subset of b_2. Hence $b_1 = b_2$. For the proof of (A.9) let $v \simeq w$. Then $CB_{\simeq}(\{v, w\})$ and thus, by (A.32), there exists a block $c \subseteq V$ with $MB_{\simeq}(c)$ and with $\{v, w\} \subseteq c$. By definition of $M_{\simeq}$, $c \in M_{\simeq}$, and thus (A.9) holds.

Proof of (A.39):
Let $b \in S$ so $CB_{\simeq}(b)$. Then, $CB_{\simeq}(b)$ and thus by (A.32) a set $b' \subseteq V$ with $b \subseteq b'$ can be found such that $MB_{\simeq}(b')$. Clearly $b' \in M_{\simeq}$. Hence $S \sqsubseteq M_{\simeq}$. □

This completes the second part of the proof of theorem A.1.

A.1.4 $m_{\simeq}$ is the minimum of $(Setsys(\simeq), \sqsubseteq)$

Theorem A.9 *$m_{\simeq}$ is the minimum of $Setsys(\simeq)$*

$$m_{\simeq} = \sqcap Setsys(\simeq) \tag{A.41}$$

□

Proof:
The proof consists of two parts:

1. $m_{\simeq}$ is a set system:
$$m_{\simeq} \in Setsys(\simeq) \tag{A.42}$$

2. $m_{\simeq}$ is the minimum:
$$\forall_{S \in Setsys(\simeq)} \quad m_{\simeq} \sqsubseteq S \tag{A.43}$$

Proof of (A.42):
The properties (A.7), (A.8) and (A.9) can be checked trivially using the definition of $m_{\simeq}$.

Proof of (A.43):
Let $S \in Setsys(\simeq)$ and let $b \in m_{\simeq}$. Then $b = \{v, w\}$ with $v \simeq w$. By the definition of a $Setsys(\simeq)$ there exists a $b' \in S$ such that $b \subseteq b'$. Hence $m_{\simeq} \subseteq S$. □

This completes the last part of the proof of theorem A.1.

A.2 $\leq$ operator

In Section 4.2.2 the following definition and theorem have been stated.

Definition A.9 *$\leq$ operator defined on set systems*
Let V a finite non-empty set of symbols and let $\simeq \subseteq V \otimes V$ and $\simeq' \subseteq V \otimes V$ be two compatibility relations defined on V. Set systems $\phi \in Setsys(\simeq)$ is called smaller or equal than set system $\psi \in Setsys(\simeq')$, written $\phi \leq \psi$, if and only if $\simeq \subseteq \simeq'$:

$$\phi \leq \psi \Leftrightarrow \simeq \subseteq \simeq' \tag{A.44}$$

□

Theorem A.10 *$\leq$ operator defined on canonical set systems*
Let V be a finite non-empty set of symbols and let $\simeq \subseteq V \otimes V$ and $\simeq' \subseteq V \otimes V$ be two compatibility relations defined on V. Then

$$M_{\simeq} \leq M_{\simeq'} \Leftrightarrow \underset{b \in M_{\simeq}}{\forall} \; \underset{c \in M_{\simeq'}}{\exists} \; b \subseteq c \tag{A.45}$$

□

Proof:
By definition of the canonical set system:

$$M_{\simeq} = \left\{ b \subseteq 2^V \mid MB_{\simeq}(b) \right\} \tag{A.46}$$

$$M_{\simeq'} = \left\{ b \subseteq 2^V \mid MB_{\simeq'}(b) \right\} \tag{A.47}$$

The proof consists of two parts:

1. **Proof of $M_{\simeq} \leq M_{\simeq'} \Rightarrow \underset{b \in M_{\simeq}}{\forall} \; \underset{c \in M_{\simeq'}}{\exists} \; b \subseteq c$**

   Let $M_{\simeq} \leq M_{\simeq'}$. Let $b \in M_{\simeq}$. Then $CB_{\simeq}(b)$. But since by (A.46) $\simeq \subseteq \simeq'$ we also have $CB_{\simeq'}(b)$. By (A.32) there exists a c with $b \subseteq c$ and $MB_{\simeq'}(c)$.

2. **Proof of $M_{\simeq} \leq M_{\simeq'} \Leftarrow \underset{b \in M_{\simeq}}{\forall} \; \underset{c \in M_{\simeq'}}{\exists} \; b \subseteq c$**

   Assume

   $$\underset{b \in M_{\simeq}}{\forall} \; \underset{c \in M_{\simeq'}}{\exists} \; b \subseteq c$$

   We have to show that $\simeq \subseteq \simeq'$. To this end let $v \simeq w$. Then for some $b \in M_{\simeq}$, $\{v, w\} \subseteq b$. But then for some $c \in M_{\simeq'}$, $\{v, w\} \subseteq c$. Since $CB_{\simeq'}(c)$ we then have $v \simeq' w$.

□

A.3 Closeness of set system product

In Section 4.2.2 the following definition has been stated.

Definition A.10 *Set system product*
Let $\phi_{\simeq} \in Setsys(\simeq)$ and $\psi_{\simeq'} \in Setsys(\simeq')$ be two set systems. The set system product $\phi_{\simeq} \cdot \psi_{\simeq'}$ is then defined as

$$\phi_{\simeq} \cdot \psi_{\simeq'} = \left\{ x \mid \underset{b_1 \in \phi_{\simeq}}{\exists} \; \underset{b_2 \in \psi_{\simeq'}}{\exists} \; x = b_1 \cap b_2 \wedge \underset{b'_1 \in \phi_{\simeq}}{\forall} \; \underset{b'_2 \in \psi_{\simeq'}}{\forall} \; x \subseteq b'_1 \cap b'_2 \Rightarrow x = b'_1 \cap b'_2 \right\} \tag{A.48}$$

It is not hard to show that $\phi_{\simeq} \cdot \psi_{\simeq'}$ is a set system representing the following compatibility relation:

$$\simeq'' = \left\{ (v, w) \in V \otimes V \mid \underset{b \in \phi_{\simeq} \cdot \psi_{\simeq'}}{\exists} \; \{v, w\} \in b \right\} \tag{A.49}$$

where $\simeq''$ is defined by

$$a \simeq'' b \Leftrightarrow a \simeq b \wedge a \simeq' b \tag{A.50}$$

We thus have $\phi_{\simeq} \cdot \psi_{\simeq'} \in Setsys(\simeq'')$ □

The goal of this section is to prove the following theorem:

Theorem A.11 *The set system product preserves canonicity*

$$M_{\simeq''} = M_{\simeq} \cdot M_{\simeq'} \tag{A.51}$$

□

Proof:
The proof consists of two parts:

1. **Proof of $M_{\simeq''} \subseteq M_{\simeq} \cdot M_{\simeq'}$**

   Let $b \in M_{\simeq''}$. Then $CB_{\simeq''}(b)$ and thus by (A.50) $CB_{\simeq}(b)$ and $CB_{\simeq'}(b)$. Then for some c with $b \subseteq c$, $MB_{\simeq}(c)$ and for some d with $b \subseteq d$, $MB_{\simeq'}(d)$. Then $CB_{\simeq''}(c \cap d)$. If $b \subset c \cap d$, we have a contradiction, hence $b = c \cap d$. Now let $b' \in M_{\simeq}$ and $b'' \in M_{\simeq'}$ such that $b \subset b' \cap b''$. Then $CB_{\simeq''}(b' \cap b'')$. This contradicts the fact that $MB_{\simeq''}(b)$. Hence $b \in M_{\simeq} \cdot M_{\simeq'}$.

2. **Proof of $M_{\simeq''} \supseteq M_{\simeq} \cdot M_{\simeq'}$**

   Let $b \in M_{\simeq} \cdot M_{\simeq'}$. Then for some $b' \in M_{\simeq}$ and $b'' \in M_{\simeq'}$, $b = b' \cap b''$ and there are no $c' \in M_{\simeq}$ and $c'' \in M_{\simeq'}$ such that $b \subset c' \cap c''$. Further, we have $CB(b)$. Suppose $\neg MB_{\simeq}(b)$ then for some $v \notin b$:

   $$\forall_{w \in b} \ v \simeq w$$

   But then $CB_{\simeq}(b' \cup \{v\})$ and $CB_{\simeq'}(b'' \cup \{v\})$ and $(v \notin b' \vee v \notin b'')$. Hence $\neg MB_{\simeq}(b')$ or $\neg MB_{\simeq'}(b'')$ which is a contradiction. We thus have $MB_{\simeq''}(b)$ and hence $b \in M_{\simeq''}$.

□

Curriculum vitae

Frank Volf was born on 15 December 1967, in Eindhoven, the Netherlands.

From September 1986 he studied at the Faculty of Electrical Engineering of the Eindhoven University of Technology. He completed his Master's degree in Electrical Engineering (specialization Information Technology) in February 1991. His graduation work was performed in the Section of Digital Information Systems under supervision of dr.ir. L. Jóźwiak and concerned the output decomposition of Boolean functions and assigned sequential machines.

From March 1991 to May 1996 he was employed as a Ph.D. student in the same section as he graduated for his Master's degree. Under supervision of prof.ir. Stevens and dr.ir. Jóźwiak he worked towards his Ph.D., which resulted in this thesis. In this period he also obtained a second degree qualification to teach mathematics.

Since 1 June 1996 he has worked as a network and security consultant for the Intranet Services Division of Origin. He is a member of the firewall team, a group of people responsible for the the design, implementation and operation of the firewalls of Origin and its customers.

Stellingen

Behorende bij het proefschrift

A bottom-up approach to multiple-level logic synthesis for look-up table based FPGAs

door

Franciscus Adrianus Maria Volf

1. LUTSYN is er het levend bewijs van dat logische synthese en technology mapping beter als één probleem beschouwd kunnen worden [dit proefschrift].
2. Don't care condities horen gebruikt te worden om de implementatie te optimaliseren en niet om de specificatie te versimpelen [dit proefschrift].
3. De setsysteemtheorie is sluitend gemaakt door de canonieke setsystemen. Dit is goed bruikbaar voor het beschrijven van decompositiestructuren [dit proefschrift].
4. De logische synthese is een van de belangrijkste stappen in het ontwerptraject van digitale systemen. Zij vertaalt een mens georiënteerde symbolische beschrijving naar een machine georiënteerde binary beschrijving.
5. In de ideale synthesemethode voor LUT-based FPGAs wordt de bottom-up netwerkconstructie van LUTSYN gecombineerd met de traditionele top-down aanpak.
6. Tegenwoordig compenseren bedrijven een matig product door een agressieve marketingstrategie.
7. Het is vreemd dat de TUE 's avonds en in het weekend zijn deuren voor haar medewerkers sluit. Dit past niet in het imago van een research instituut.
8. Het is verstandig om na de promotie te gaan werken in een ander vakgebied.
9. Als de voortekenen van CNN ons niet bedriegen, staan er straks meer verslaggevers dan soldaten aan het front.
10. De digitale snelweg begint steeds meer te lijken op een gewone snelweg. Verschijnselen als files, opstoppingen, spitsuur en omleidingen doen zich tegenwoordig ook op de digitale snelweg voor.
11. In het huidige ruimtevaarttijdperk, waarin spionagesatellieten de score op de golfbaan kunnen bijhouden, is het verstandig bij sommige activiteiten een paraplu te gebruiken, ook als het niet regent.
12. Een *stelling* is een plaats waar een partij eike*n*hakhouttakken geschild wordt (van Dale Groot woordenboek der Nederlandse taal).