Variable Ordering for ROBDD-based FPGA Logic Synthesis

by

Jacqueline E. Crow
BSc., University of Victoria, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MSc.

in the
Department of Computer Science

We accept this thesis as conforming
to the required standard

Dr. Micaela Serra, co-supervisor (Dept. of Computer Science)

Dr. Jon Muzio, co-supervisor (Dept. of Computer Science)

Dr. John Ellis, Departmental Member (Dept. of Computer Science)

Dr. Inna Sharf, Outside Member (Dept. of Mechanical Engineering)

Dr. G. McLean, External Examiner (Dept. of Mechanical Engineering)

Supervisor: Drs. Micaela Serra and Jon Muzio

Abstract

The field programmable gate array (FPGA) is becoming a more and more popular replacement for application specific integrated circuits (ASICs). Logic synthesis techniques for this technology have been adopted and modified from those used for regular ASICs. In this thesis, a technique using Reduced Ordered Binary Decision Diagrams (ROBDDs) for logic synthesis for look-up table (LUT)-based FPGAs is investigated. The ROBDD is a useful representation for logic functions as it is a canonical form and it is relatively straight-forward to map this form to LUT-based FPGAs. A major problem with this representation, however, is that the variable ordering can have a very large effect on the size of the ROBDD. Another way of representing a function is with its autocorrelation coefficients. These provide information about areas of similarity in a function. This thesis introduces a method of generating variable orderings for ROBDDs based on the autocorrelation coefficients of the function. The questions we investigate are whether this method of ordering consistently produces ROBDDs of a reasonable size, and whether these ROBDDs can be mapped to LUT-based FPGAs to result in a smaller number of logic blocks than on average.

Examiners:

Dr. Micaela Serra, co-supervisor (Dept. of Computer Science)

Dr. Jon Muzio, co-supervisor (Dept. of Computer Science)

Dr. John Ellis, Departmental Member (Dept. of Computer Science)

Dr. Inna Sharf, Outside Member (Dept. of Mechanical Engineering)

Dr. G. McLean, External Examiner (Dept. of Mechanical Engineering)

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank everyone who has helped me out with this thesis, and given me lots of wonderful advice! In particular, Dr. Micaela Serra, Dr. Jon Muzio, and Dr. Mike Miller, and all the grad students in the VLSI group. Thanks, everyone!

For Kevin.

Chapter 1

Introduction

1.1 Motivation

Field programmable gate arrays (FPGAs) are a relatively new product in the VLSI industry. A FPGA is a chip whose function can be programmed without the use of large manufacturing equipment. This new technology has become very important in the field of VLSI ASIC design[CD94]. Today FPGAs are used in many areas, including prototyping for education. Their usability is due to their field-programmability, relative inexpense, and speed of production.

In order to implement a function using a FPGA, some sort of logic synthesis is required. This usually involves two steps; the function is described in a standard form and some manipulation is done, followed by technology mapping to the target technology.

If the target technology is known to be a FPGA, a representation can be chosen to simplify the minimization and technology mapping steps. One such representation is a reduced ordered binary decision diagram (ROBDD). A ROBDD has many advantages, two of which are that ROBDDs are canonical representations, and that there is a relatively straight-forward mapping from the ROBDD representation of the function to look-up table-based (LUT-based) symmetrical FPGA architectures. However, it is known that the size of a ROBDD depends heavily on the chosen ordering of the variables[BSPR93]. It is this problem that this research investigates.

Another way to represent a function is to use its autocorrelation coefficients. This representation provides a measure of similarity between the function in question and the same function with some modifications. By using this information, it may be possible to generate a variable ordering that results in a ROBDD of optimal size (*i.e.* the fewest possible nodes).

1.2 Goals

In this thesis a number of methods for generating variable orderings based on a function's autocorrelation coefficients are developed. The goal of this work is to develop an ordering based on the autocorrelation coefficients such that the ROBDDs built using this ordering will never have more nodes than a ROBDD built using a random ordering. This ordering should also perform better than current ordering methods can achieve.

1.3 Outline

Chapter 2 describes the different architectures for FPGAs. The different methods of implementing both the logic blocks and programmable connections between the logic blocks are also discussed. Finally, the steps involved in designing a FPGA to implement a specific function are briefly outlined.

In Chapter 3 we discuss different function representations, in particular, ROBDDs. The steps in technology mapping are discussed, and the technology mapping tool used in this work is examined in detail.

Chapter 4 defines the autocorrelation coefficients and discusses in detail how to calculate them. The meaning of these coefficients is also presented.

Chapter 5 relates the autocorrelation coefficients to the problem of variable ordering for ROBDDs. A number of in-depth examples are given with a detailed discussion. After presenting the motivation for this work, the different methods of generating the autocorrelation coefficient-based orderings are described.

In Chapter 6 we describe the metrics and methods used for producing our results. For each autocorrelation coefficient-based ordering, a comparison to two standards is given, followed by discussions of these comparisons.

Chapter 7 concludes this thesis by summarizing the results. Some recommendations and areas for future research are also presented.

Chapter 2

Introduction to FPGAs

Before looking at the details of this work, some background material must first be introduced. This chapter presents an overview of FPGAs. The term FPGA is explained, and different types of FPGAs are introduced. Also, a discussion of the design flow for programming FPGAs is presented. Finally, a general overview of logic synthesis closes the chapter.

2.1 FPGAs

FPGA stands for *Field Programmable Gate Array*. As the name suggests, a FPGA is an array of devices whose interconnects can be programmed by a user without the use of large and expensive equipment. A more formal definition for the term FPGA can be stated as[BFRV92]:

> "A FPGA is a device in which the final logic structure can be directly configured by the end user, without the use of an integrated circuit fabrication facility."

Currently there is much interest in FPGAs. Some of the many reasons for this are listed below:

- Realization times are measured in hours rather than months, providing a new mechanism for rapid prototyping [SKC94].
- FPGAs provide both large scale integration and user programmability [FRV91].
- The short design cycle and low manufacturing costs have made FPGAs an important technology for VLSI application specific integrated circuit (ASIC) designs [CD94].
- FPGAs are readily scalable, and can take advantage of new processes without a lot of design or architectural changes [Pat90].
- According to Xilinx, one of the main FPGA manufacturers, FPGAs have
  - a flexible architecture,
  - considerably shortened development time as compared to usual ASICs,
  - high density,
  - advanced development tools, and
  - user programmability.

2.2 General Layout

FPGAs consist of two basic elements:

1. elements to implement the logic (logic blocks), and
2. programmable interconnections between the logic blocks.

All FPGAs also have elements to facilitate input and output, and usually some type of storage elements.

Different types of FPGAs employ different methods for implementing the logic blocks and programmable interconnects. There are also different architectures combining these elements.

The main possibilities for FPGA architectures include the following [BFRV92]:

- Symmetrical array architectures,
- Row-based architectures,
- Sea-of-gates architectures, and
- Hierarchical Programmable Logic Device (PLD) architectures.

Figure 2.1 shows diagrams describing each of these four architectures. For each of the architectures shown, the I/O and storage elements are usually incorporated along the outer edge of the FPGA.

Symmetrical array architectures.
This layout consists of logic blocks laid out on the chip in an array format. The logic blocks are connected by groups of interconnection wires that run along the length and the width of the chip, between the columns and rows of the logic blocks. Programmable interconnection points are at the intersections of the interconnection wires. Two commercial FPGAs that make use of this architecture are the Xilinx families of FPGAs and the QuickLogic FPGAs.

Row-based architectures.
In this architecture, the logic blocks are laid out in rows which are separated by

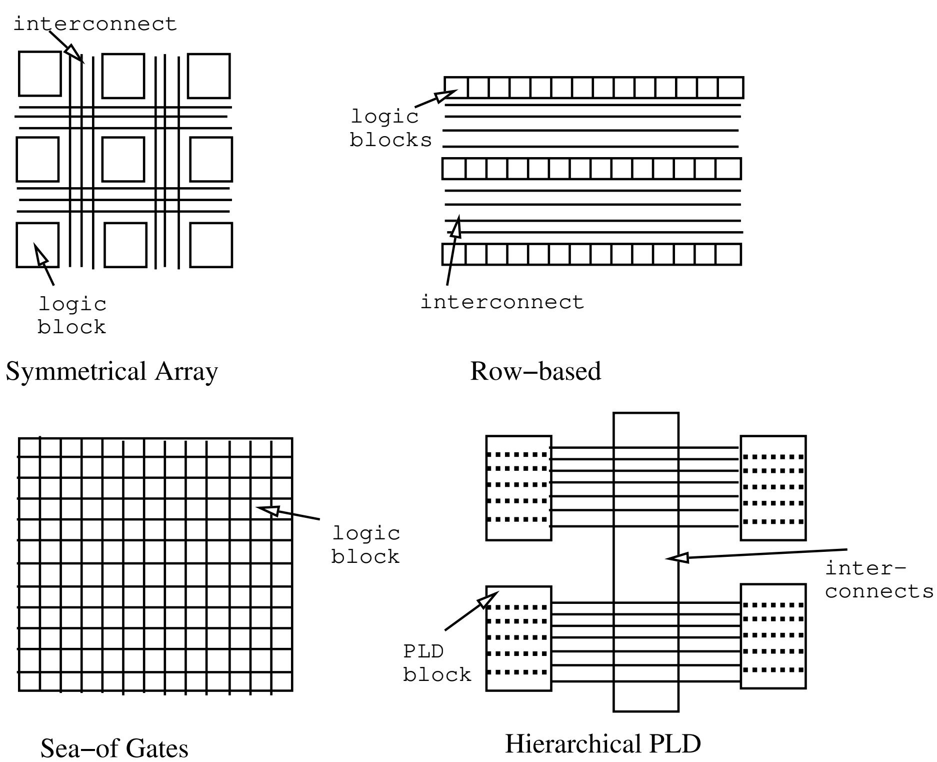


Figure 2.1: The four basic architectures for FPGAs.

horizontal wiring segments. Vertical wiring segments are also available for connecting the rows. Crosspoint Solutions and Actel/TI are two companies that make use of this architecture in their FPGAs.

Sea-of-gates architectures.
This layout consists of an array of logic blocks overlayed with a dense interconnect resource. This architecture is used by the Algotronix Corporation in their FPGAs.

Hierarchical programmable logic device-based (PLD) architectures.
This layout consists of a number of separate programmable logic devices. These devices are each connected to a central connector, where all programmable connecting is done. Altera and AMD make use of this architecture[1] in their FPGAs.

2.2.1 Logic Blocks

As mentioned above, there are different methods for implementing both the logic blocks of a FPGA and the programmable interconnects.

The most common models for FPGA logic blocks are summarized here.

- **Look-Up Tables (LUTs).** A LUT with k inputs and m outputs is referred to as a *k-input m-output LUT*. One of the Xilinx families of FPGA is based on a 5-input 2-output LUT. Such a look-up table can implement any function with a maximum of 5 inputs and 2 outputs.

- **Combinations of programmable array logic, programmable logic arrays, or programmable macro logic.** Each of these generally consist of some

[1] Xilinx, QuickLogic, Crosspoint Solutions, Actel, Texas Instruments (TI), Algotronix, Altera, and Advanced Micro Devices (AMD) are each trademark names of their respective companies.

type of programmable array that combines product and sum terms in order to implement a function. Further details are given by [Mil93]. The Altera and AMD FPGAs use a logic block that follows this model.

- **Multiplexors (MUXes).** A multiplexor is a block of logic that selects m input lines from n possible choices, and allows the signals on these m inputs to be propagated through to the m outputs of the multiplexor. Usually small MUXes, such as 2-1 (2-input, 1-output) MUXes are used in logic blocks. Networks of such MUXes are also sometimes used. This particular model is used in the logic blocks of the Altera/TI FPGAs.

- **Basic gates.** The logic block may consist of either a simple gate (*e.g.* AND, OR, NAND, etc.) or it may consist of a combination of these. The Plessey FPGAs have logic blocks consisting of NAND gates[2].

- **Transistors.** The logic blocks of the Crosspoint FPGAs consist of transistor pairs.

- **Combinations of the above.** Some FPGA architectures have combined some of the above models for their logic blocks. For example, the Algotronix logic blocks consist of multiplexors and basic gates.

This research concentrates on the LUT-based design for logic blocks that is used in the Xilinx 3000 series of FPGAs. The main reason for this is the current popularity of this type of FPGA.

A LUT is a digital memory with k address (input) lines that can implement any function of those k inputs by placing the truth table into memory[BFRV92].

[2]Plessey is a trademark name of the Plessey company.

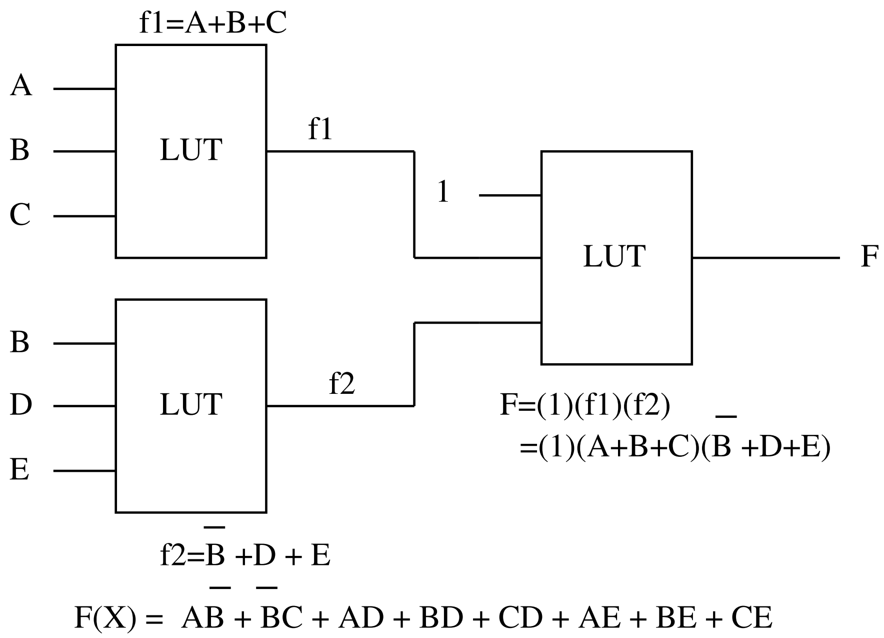


Figure 2.2: Three 3-input LUTs used to implement a function with 5 variables.

Figure 2.2 shows how two LUTS, each with 3 inputs and 1 output, can be connected to implement the function

$$f = A\overline{B} + \overline{B}C + AD + BD + CD + AE + BE + CE$$

A LUT with more inputs can implement more functions, and so fewer blocks are needed to implement the desired circuit. However, a LUT with more inputs also needs more routing and area per block.

2.2.2 Programmable Interconnections

The routing architecture of a FPGA consists of wire segments that are connected with programmable switches. The programmable switches can be implemented as:

- EPROMs (Erasable Programmable Read Only Memories),

- EEPROMs (Electrically Erasable PROMs),
- static RAM, and
- anti-fuses.

Anti-fuses and static RAM are the most often used.

According to the Actel/TI designers, the choice of a logic module (after designing the antifuse interconnect)

> ...turned out to be a very difficult question, involving subtle trade-offs among routability, the logical capability of the module as perceived by the user, and delays due to the capacitive loading in the routing segments [GGR+89].

So although any type of logic block could be paired with a given programmable switch, flexibility and routability issues must be taken into account when making a choice.

The Actel/TI FPGAs make use of the anti-fuse technology in the programmable interconnects. The idea of the anti-fuse is that there is no connection through the anti-fuse until it is fused by a high voltage. This type of interconnect is only programmable once.

The Altera FPGAs use EPROM programmable interconnects. Although this type of interconnect is re-programmable, re-programming cannot be done in-circuit [BFRV92].

The EEPROM approach is used in the AMD programmable interconnects. These can be re-programmed in-circuit, but require more space than the EPROM version [BFRV92].

The Xilinx families of FPGAs make use of static RAM (SRAM) programming technology, along with a number of other companies. This type of connection is re-programmable in-circuit, but requires a relatively large amount of space. The SRAM interconnects also require a permanent storage to store the FPGA configuration, as the SRAM is volatile [BFRV92].

2.3 Design Flow

When designing a FPGA to perform the desired function, two basic steps are followed:

1. Perform logic synthesis on the function. The purpose is to transform the function into a FPGA-compatible format.
2. Download the transformed function to the FPGA.

With the Xilinx FPGA software called XAct[3], the design flow follows this outline:

i. Design Entry - describe the function using a schematic capture tool or a text-based description.

ii. Design Verification - simulate the design to verify that it is functionally correct.

iii. Design Implementation - break down the design so that it fits in the logic blocks of the FPGA, then program the logic blocks and the programmable interconnects to connect routes between appropriate logic blocks. This is known as *partitioning*, *placing*, and *routing*.

[3]XAct is a trademark of the Xilinx company.

iv. In-circuit Verification - simulate working with the programmed FPGA to ensure that the implemented circuit behaves correctly.

It is in the first step, design entry, that logic synthesis is done.

2.4 Logic Synthesis

When used in reference to VLSI design, logic synthesis is most commonly defined as a two-step process consisting of[MM93]:

1. the optimization of a technology-independent logic representation, and
2. technology mapping.

In this research only logic synthesis for combinational logic is examined.

The above steps are usually broken down into more detail as follows:

1. A standardized representation of the desired function is produced. Standard formats may vary from graphs such as binary decision diagrams (see section 3.1), to equations describing the logic or languages such as Register Transfer Language (RTL).
2. The standard format is manipulated in order to minimize the logic. Minimizing the logic consists of removing any redundancies and attempting to reduce the number of logic components.
3. Having reached a minimal or near minimal representation, the logic description must now be transformed to a format that is implemented in the desired technology. This format can vary from a list of basic gates to layouts that describe transistor structures.

Steps 1 and 2 are part of the technology independent optimization phase, while step 3 is the technology dependent step known as technology mapping.

Step 1 - produce a representation of the function in a standard format Languages such as VHDL[4] or RTL (Register Transfer Language) are often used to initially specify the function. In order to perform the next step of minimizing the logic, this description is often transformed into a two-level or multi-level representation of the function. A *sum-of-products* representation, as shown below, is one example of a two-level representation.

$$F = ac + ad + bc + bd + e$$

A *factored form* representation, show here, is an example of a multi-level representation.

$$F = (a + b)(c + d) + e$$

The same function is shown in each example.

Function representations are discussed in more detail in section 3.1.

Step 2 - manipulate the function in order to minimize the logic Depending on the representation chosen in step 1, either two-level or multi-level logic minimization is done.

When performing two-level minimization, the goal is to find a sum-of-products expression which is a cover[5] for a given logic function and which has the least number of product terms.

The objective of multi-level logic synthesis is to find the "best" multi-level structure, where "best" in this case means an equivalent representation that is optimal

[4]VHDL stands for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuits.

[5]see Appendix A for definitions of terminology.

with respect to some cost function. Five basic operations are used in order to reach this goal:

i. Decomposition. This is the process of re-expressing a single function as a collection of new functions.

ii. Extraction. This is the process of identifying and creating some intermediate functions and variables, and re-expressing the original functions in terms of the intermediate plus the original variables. The process is used to identify the common sub-expressions.

iii. Factoring. This is the process of deriving a factored form from a sum-of-products form. The reason for this is to get the minimum number of literals possible in the expression.

iv. Substitution. This is the process of expressing a function F as a function of a second function, G, plus the original inputs to the function F. This is done by substituting G into F where ever possible.

v. Collapsing. This is also known as elimination, or flattening, and is the inverse of substitution.

These manipulations are repeated until the "best" structure (or close to it) is achieved. It is possible to use either *algebraic* or *Boolean* methods to perform the five operations listed above. Details and algorithms for both methods are given by [BHSV90].

Step 3 - technology mapping

Technology mapping is defined as a process of transforming a technology independent (optimized) Boolean network into a technology-based circuit [Woo91]. Traditional techniques for technology mapping use a library of basic cells [SVGR93]. The Boolean network representing the circuit is transformed so that it uses only cells that exist in the library.

2.5 Summary

The purpose of this chapter is to provide some background information about FPGAs for the work this thesis presents.

This chapter introduces a new technology called field programmable gate arrays (FPGAs). The chapter gives a description of the different FPGA architectures and the different methods for implementing the combinational logic blocks and programmable interconnects. In this research only LUT-based FPGAs are considered.

A brief description of the design process for FPGAs is also given. One of the steps in this process is the synthesis of the logic function to be implemented. Logic synthesis and function representations are discussed in detail in the following chapter.

Chapter 3

Function Representations and Logic Synthesis

This chapter examines more closely different ways of representing the Boolean function to be synthesized. In particular, the representation used in this research is presented. Also, a tool used for performing logic synthesis and technology mapping is introduced.

3.1 Traditional Representations of Boolean Functions

Before minimizing the logic during logic synthesis, a decision on how to represent the function must be made.

The most straightforward way to represent a Boolean function is using its *truth table*. This is a table in which all possible input values are given, along with the corresponding output value(s). A truth table of a simple function with three inputs

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 3.1: the truth table corresponding to the sum-of-products expression $A + BC$

and one output is shown in Figure 3.1. Unfortunately, the representation is very large, since for n inputs, there are 2^n possible input values. For this reason, other methods for representing a function are usually chosen.

As discussed in Chapter 2, a function can be represented using a two-level or multi-level representation.

One two-level representation is the *sum-of-products* representation. Another two-level representation is the *Espresso*, or *pla* format [Yan91]. A pla representation of the sum-of-products expression $A + BC$ would look like this:

```
.i 3
.o 1
.p 2
-11 1
1-- 1
.e
```

The *.i*, *.o*, and *.p* indicate how many inputs, outputs, and product terms the file contains. Next, each position in the input plane, that is, the group of numbers on the left, corresponds to an input variable. A "1" implies that the input literal appears uncomplemented in the product term and a "0" implies that the input literal appears in complemented form. A "-" implies a *don't care* value, meaning that the input literal does not appear in that product term. See [Yan91] for further details on this representation. As the name indicates, this representation is mainly used for specifying functions to be implemented using programmable logic arrays (PLAs)[1].

Multi-level representations are also called *Boolean networks*. These can be described as a directed acyclic graph in which each node is associated with a variable and a representation of some logic function [BHSV90]. In this type of representation, each node can represent one specific function, one of a small set of functions, or any arbitrary function.

3.2 An Introduction to ROBDDs and ITE-DAGs

One way of representing a function, as mentioned above, is by its truth table. However, since a truth table lists all possible input values, this is a rather large representation. One way of describing a function's truth table is to use a *Binary Decision Diagram* (BDD). In order to avoid the size problem, methods of reducing the BDD have been found. Another, similar multi-level representation is if-then-else directed, acyclic graphs (ITE-DAGs). Both the reduced version of BDDs and ITE-DAGs have great potential for use in representing logic expressions [F$^+$93].

3.2.1 BDDs

Binary decision diagrams, or BDDs, are graph representations of Boolean functions. A BDD is defined as [Kar88b]

[1]see Appendix for definition.

> *a binary directed acyclic graph with two leaves* TRUE *and* FALSE*, in which each non-leaf node is labeled with a variable and has two out-edges, one pointing to the subgraph that is evaluated if the node label evaluates to* TRUE *and the other pointing to the subgraph that is evaluated if the node label evaluates to* FALSE*.*

Every node in the BDD represents either a literal in the Boolean function, or its complement. Every internal node has two outward edges leading to two other nodes. If the node has a value of "1" (TRUE) then, to obtain the value of the expression, one follows the edge marked "1" and evaluates that node. Similarly, if the node has a value of "0" (FALSE), one follows the edge marked "0" and evaluates that node. Eventually an output with the value "1" or "0" is reached, and the evaluation is complete. In all diagrams of BDDs and ITE-DAGs the direction of the edges leading from each node is not explicitly marked, but rather implied.

3.2.2 ROBDDs

BDDs are often used in their canonical form. This simplified form has the property that any two equivalent Boolean functions have the same canonical BDD if the same variable ordering is used. This canonical form is called a *ROBDD*, or *R*educed *O*rdered *B*inary *D*ecision *D*iagram.

A ROBDD is a reduced BDD with a specified ordering of variables. A ROBDD meets two main specifications:

- a BDD is a reduced BDD if it contains no vertex whose left subdiagram is equal to its right subdiagram, nor does it contain distinct vertices v and v' such that the subdiagrams rooted by v and v' are isomorphic.

row	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

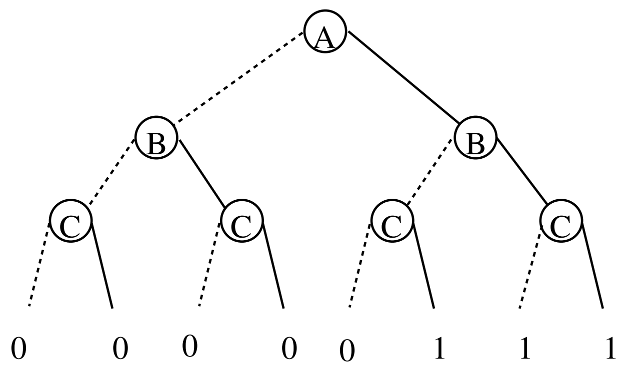


Figure 3.2: the truth table and corresponding BDD for the function $f = AB + AC$

- a BDD is an ordered BDD if on every path from the root node to an output, the variables are encountered in the specified order.

One can produce a BDD directly from the truth table for the function, as shown in Figure 3.2[2]. The function shown in Figure 3.2 is $f = AB + AC$, and the truth table on the left-hand side of the Figure gives the behaviour of the function. To create the BDD from the truth table, a path leading from the root node to a leaf is created for each line in the truth table. In Figure 3.2, the root node is chosen to be A. As an example, in row 3, the value of the function F is "0". To create a path in the BDD reflecting this, we follow left or right edges in the diagram, depending on whether the node we are currently visiting has a value of "1" or "0" in the third row of the truth table. A has the value "0", so we follow the left edge to the next node, B, which, in the truth table, has the value "1", so the next edge we follow is the right edge. C also has the value "1", so again we follow the right edge to reach a leaf node. Since this leaf node represents the value of the function with the variable assignments as given in the third row of the table, the leaf node must have the value given in the

[2]in this Figure, and all future Figures of BDDs, the path to follow if the node evaluates to a "0" is marked by a dotted line, while the path to follow if the node evaluates to a "1" is marked with a solid line. Edges to the final outputs may be shared in all ROBDDs; however, in order to simplify the diagrams, this is not always shown.

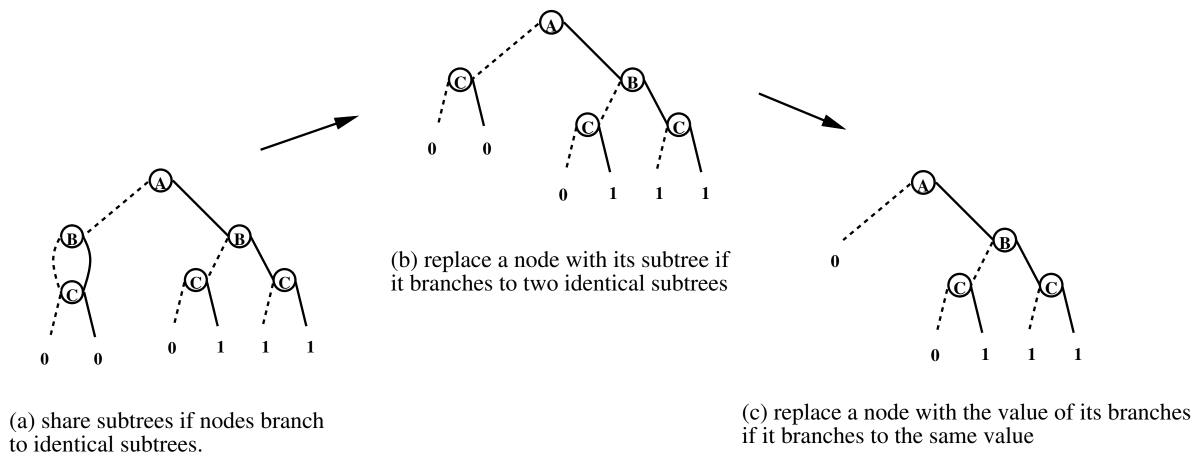


Figure 3.3: simplifying a BDD

third row of the truth table, that is, a "0". A similar process is followed for each row in the truth table.

Once the BDD has been created from the truth table, it is often necessary to simplify it in order to save space, or to get a ROBDD. The main steps in simplifying BDDs are as follows:

- Check to see if any two (or more) nodes branch to identical subdiagrams. If so, then the subdiagrams are shared. The way this is done is simply by eliminating one of the subdiagrams, and having both (all) of the nodes branch to the same subdiagram. This step is shown in part (a) of Figure 3.3. The original BDD is shown in Figure 3.2.

- Check to see if any one node branches to two identical subdiagram. If so, then that node is redundant, and is replaced with its subdiagram. This is shown in part (b) of Figure 3.3.

- Finally, check to see if a node branches to two output nodes with the same value. If so, then that node is replaced with the value, as shown in part (c) of

Figure 3.3.

- If one subdiagram is the inverse of another, *inverted edges* may be used in the ROBDD in order to further reduce its size [Ake78].

For functions with multiple-outputs a BDD is created for each output. When simplifying the BDDs for each output, sharing of subdiagrams may be done between the different ROBDDs.

3.2.3 Implementing ROBDDs

Since a BDD has two outward edges from each internal node, it is reasonable to represent this structure as a node with two pointers and an identification field, where each pointer points to a node or to a function output. The outputs in this case are the nodes containing the value "1" or "0" and pointers pointing to nil.

However, the problems of comparing subdiagrams of the ROBDD in order to produce the canonical form can become quite complex and time consuming, so other structures must also be used. One such structure suggested to solve this problem is a *hash table* [BRB90]. The hash table is used to save a copy of each node in the BDD, where the node is described as a triple consisting of the parent of the node and its left and right children. Before any new node is added when constructing the ROBDD, the hash table is checked to see if the node is already in the table, in which case the existing node is used instead of creating a new one.

In [Bry86] the following structure is used:

```
type vertex = record
    low, high: vertex;
```

BDD Operations		
Procedure Name	Function	Time Complexity
Reduce	an arbitrary BDD G is reduced to the canonical ROBDD	$O(\|G\| \log(\|G\|))$
Apply	takes two BDDs G_1 and G_2 representing functions f and g and produces the BDD of f *op* g, where *op* is a Boolean operation	$O(\|G_1\| \cdot \|G_2\|)$
Restrict	takes in a BDD G and produces the resulting BDD for a *restriction* of one of its variables; that is, for a given assignment of one of its variables	$O(\|G\| \log(\|G\|))$
Compose	takes in two BDDs G_1 and G_2 and produces the BDD for the *composition* of the two functions. Composition is defined below.	$O(\|G_1\|^2 \cdot \|G_2\|)$

Table 3.1: Table showing possible operations on BDDs, along with the time complexity for each operation.

index: $l \ldots n+1$;

val: $(0, 1, X)$;

id: integer;

mark: Boolean;

end;

Table 3.2.3 shows some of the operations performed on BDDs and their time complexities [Bry86].

Composition is defined as

$$f_1|_{x_i=f_2} = f_2 \cdot f_1|_{x_i=1} + (\bar{f_2}) \cdot f_1|_{x_i=0}$$

where $f_1|_{x_i=a}$ is the restriction of f_1 where $x_i = a$.

ROBDDs are a good choice for representing functions for a number of reasons [Bry86]:

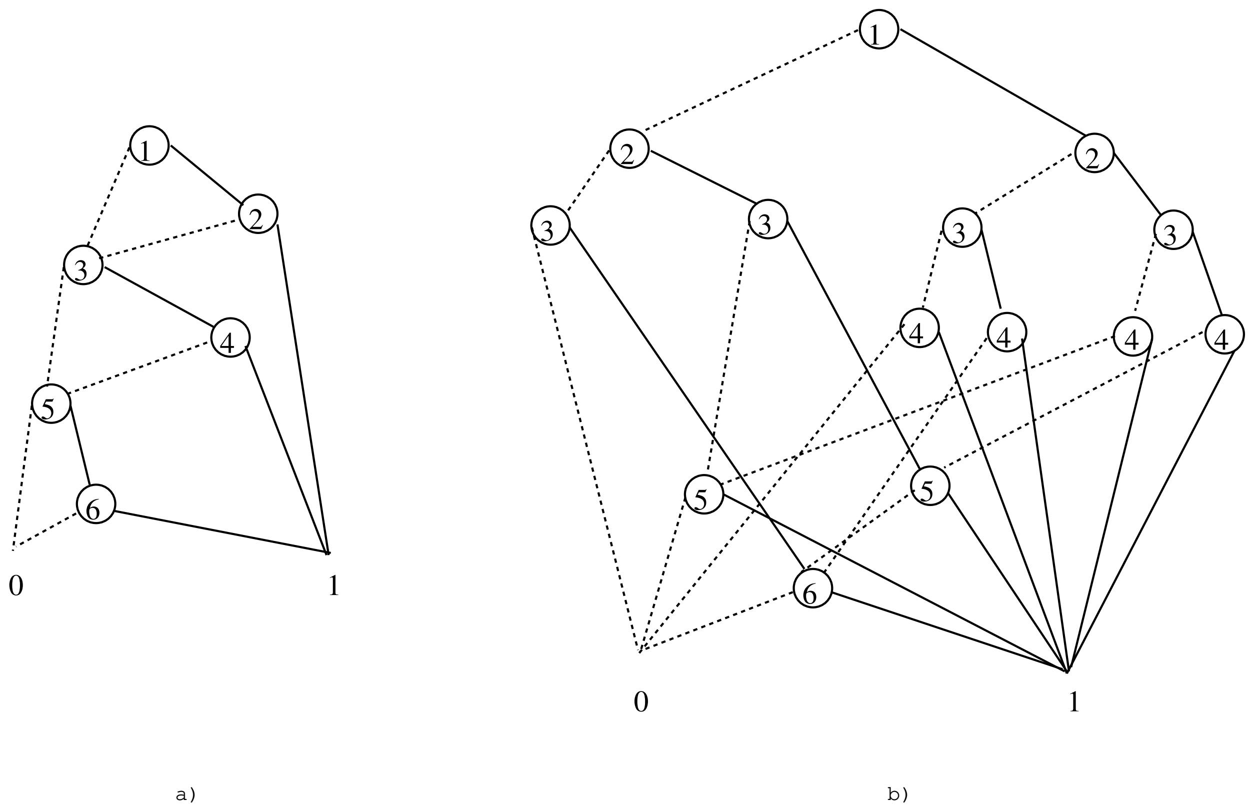


Figure 3.4: Two ROBDDs representing the same function, with the order of the input variables permuted.

- most commonly encountered functions have reasonable ROBDD representations,
- for many practical functions, the time complexity of any single operation is bounded by the product of the graph sizes for the functions being operated on, and
- ROBDDs are a canonical representation.

However, some ordering of the variables must be chosen before processing the function, and for some functions, the size of the ROBDD representing the function is highly sensitive to the ordering. This is demonstrated in Figure 3.4, where the first

ROBDD represents the function $f_1 = x_1x_2 + x_3x_4 + x_5x_6$ and the second ROBDD represents the function $f_2 = x_1x_4 + x_2x_5 + x_3x_6$. These functions differ from each other only by a permutation of the input variables, yet it is clear that the first ordering results in a much smaller ROBDD representation. The problem of finding a variable ordering that minimizes the size of the corresponding ROBDD is itself a co NP-complete problem [B$^+$86].

According to Bryant, for most practical functions a reasonable ordering can generally be found in a relatively short amount of time by a human with some knowledge of the function. The functions for which there is no good ordering are seldom used in practical applications, with the exception of integer multiplication [Bry86].

3.2.4 ITE-DAGs

Another method of representing Boolean functions is to use If-Then-Else directed acyclic graphs (ITE-DAGs).

ITE-DAGs can be described in terms of a ROBDD that has been split into parts. This is easiest to see with a trivial ROBDD consisting of one variable, as is shown in Figure 3.5. In the transition from the BDD in part (b) of Figure 3.5 to the ITE-DAG in part (c) of Figure 3.5, we can see that the node labeled A has moved from being the parent of the nodes labeled "0" and "1" to being a sibling of these two nodes. If the second and third leaves of the ITE-DAG are labeled B and C respectively, then we get the description of how to evaluate an ITE-DAG:

if A **then** B **else** C

The formal definition of an ITE-DAG is as follows [Kar88a]:

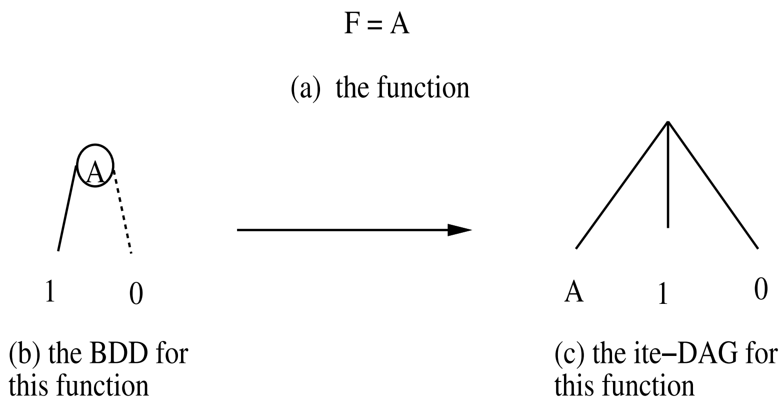


Figure 3.5: Describing an ITE-DAG in terms of a BDD

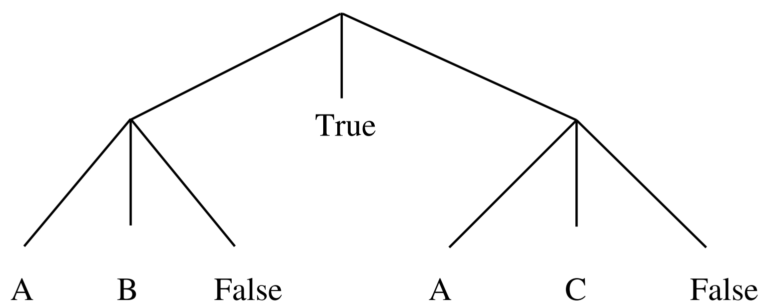


Figure 3.6: an ITE-DAG for the function $f = AB + AC$

Definition 3.2.1 *An if-then-else DAG is a ternary directed acyclic graph in which each leaf is labeled with TRUE, FALSE, or a literal, and each internal node has three out-edges pointing to the* **if-**, **then-**, *and* **else-** *parts. The meaning of a leaf node is the label on the node and the meaning of an internal node is described recursively as*

if *meaning(***if***-part)* **then** *meaning(***then***-part)* **else** *meaning(***else***-part) [Kar88a]*

An example of an ITE-DAG for the function $f = AB + AC$ is shown in Figure 3.6.

Another way of describing an ITE-DAG is as a structure describing a 3-input multiplexor. Two of the inputs are the data inputs, and the third is the select input. Figure 3.7 shows how the if-then-else structure is related to a 3-input multiplexor.

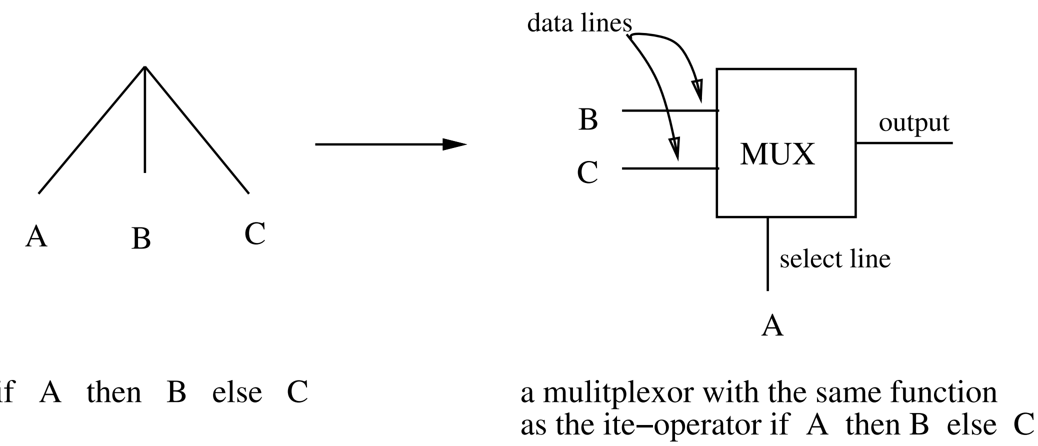


Figure 3.7: How the if-then-else structure can be described as a three-input multiplexor.

We noted in the previous section that an advantage of using ROBDDs is that they are a canonical representation of the function. There is also a canonical form of ITE-DAGs.

A canonical form of ITE-DAGs is defined by [Kar88a]. Seven restrictions on the expressions allowed in the **if**, **then**, and **else** parts. Details are given by [Kar88a].

For multiple-output functions there is a separate ITE-DAG for each output. When reducing the ITE-DAGs to their canonical form, nodes may be shared between the different ITE-DAGs.

3.3 Technology Mapping to FPGAs

This section gives an introduction to different technology mapping algorithms, and then shows an example using Item for minimization and technology mapping. (The reader is reminded that in this work only FPGAs with look-up table logic blocks architectures are considered, therefore only methods for logic synthesis to these architectures are discussed.) Item is a tool for logic synthesis and technology mapping which uses ITE-DAGs to internally represent functions. Details on this tool are given in the following section.

In general, technology mapping approaches consist of three steps [Woo91]:

1. Decomposition – transform the initial Boolean network that is input so that every node in the network has k or less distinct inputs. k is the number of inputs for each logic block.

2. Reduction – reduce the number of nodes and/or edges in the network representing the function.

3. Packing – combine two nodes satisfying certain input constraints into one block.

The first step, decomposition, can be done in a number of ways.

The mapping algorithm called Xmap [Kar91] works with ITE-DAGs used as the internal representation of the circuit. The ITE-DAGs then become a natural decomposition. Another technology mapper known as Chortle [FRC90] does something similar by dividing the input network into a forest of trees. An extension to MIS (Multiple-level Logic Optimization System) [B+86] called Mis-pga [MNS+90] uses kernel extraction with Roth-Karp decomposition to decompose a possibly *infeasible* function into a *feasible* function. A function is called *feasible* if all of its nodes can be implemented using FPGA logic blocks, otherwise it is *infeasible*. Kernel extraction[3] consists of extracting the kernel from an infeasible node. The node and the kernel are then recursively decomposed. The Roth-Karp decomposition is done if the first method fails.

[3]See Appendix A for definitions.

Roth-Karp decomposition, also known as *disjoint decomposition*, consists of partitioning the inputs to the node into two sets called the *bound* set and the *free* set. The bound set can be of size no greater than the number of inputs to the logic block being mapped to. If we denote the bound set as X and the free set as Y, then a function F decomposed using Roth-Karp decomposition takes on this form:

$$F(X,Y) = g(a_1(X), a_2(X), \ldots, a_t(X), Y)$$

Mis-pga-new [MSBSV91] uses improved methods for decomposition. The four methods that Mis-pga-new uses are:

- cube packing,
- cofactoring,
- And-Or decomposition, and
- disjoint decomposition (Roth-Karp decomposition).

The main method used is *cube packing*. In cube packing, the cubes of the functions are considered to be "items" that are to be packed into "bins", where the logic blocks of the FPGA are the bins. The weight of each cube is the number of literals it contains, and the capacity of each bin is the number of inputs to the logic block. The problem is to use the minimum number of bins to hold all of the cubes. Since this is a NP-complete problem, heuristics are used [MSBSV91]. However, this method may produce poor results if the cubes share many literals. In this case, *cofactoring* is used.

Cofactoring is also known as *Shannon decomposition*. In this case, a function f is decomposed into a form such as the following:

$$f = x_i \cdot f_{x_i} + \bar{x}_i \cdot f_{\bar{x_i}}$$

f_{x_i} is the function f with the variable x_i assigned the value of "1" while $f_{\bar{x_i}}$ is the function f with the variable x_i assigned the value of "0". Each decomposed function f_{x_i} and $f_{\bar{x_i}}$ has one less variable than the original function, so the process is applied recursively until each decomposed function has fewer variables than the number of inputs to the logic blocks of the FPGA. It should be pointed out that this is also a method used for building BDDs.

And-Or decomposition is used when the function is better described using factored forms instead of cubes.

Disjoint, or *Roth-Karp decomposition* is described above. The problem with disjoint decomposition is that there may be too many possible partitions of the input variables. However, it is useful for symmetric functions since all partitions of the variables are equally good [MSBSV91].

After decomposition, reduction and packing is done.

In the reduction step the goal is to minimize either the number of nodes or the number of edges in the network. The reason for this is that the number of nodes in the network correlates to the *area* that the mapped circuit occupies [SVGR93], and the number of edges correlates to the *delay* of the mapped circuit [CD94]. A third goal in technology mapping is to improve *routability*. When placement and routing are performed on a mapped circuit, it is not always possible to make the necessary connections using the programmable interconnects. For this reason, it is sometimes necessary to perform a technology mapping algorithm that optimizes the routability of the circuit.

The first two goals for technology mapping, mapping for minimal area and mapping for minimal delay, need little explanation. It should be clear that in a VLSI circuit, high delay between signals being put on the inputs and the correct signals being propagated to the outputs is undesirable. Minimizing the area is also necessary since an implementation that is not minimized for area may have too many logic blocks to fit on the FPGA, while an implementation with the number of nodes minimized may fit. Usually, mapping for both minimal area and delay is done, if possible. However, there are trade-offs between the two. When minimizing the number of nodes in the network, often the number of edges, or the depth of the circuit is increased. If delay occurs in the connections between the blocks, then the more levels of blocks there are, the higher the delay. So in mapping to minimize both delay and area, both the number of blocks and the number of connections between them are minimized. Chortle-d [FRV91] is an example of a technology mapper that maps for both delay and for area. This mapper locates the *critical paths* of the circuit, or paths where the longest delay occurs. It then performs a depth-optimizing algorithm on these paths. Once this is complete, a second mapper which optimizes for area is then run on the circuit.

The reduction and packing steps are sometimes combined and called *node minimization.* [SVGR93] describes three methods for this:

1. local elimination (sometimes called partitioning),
2. covering, and
3. merging.

Local elimination consists of looking at pairs (i, j) of nodes where node i is a fan-in to node j. A node is a *fan-in* to another node if the output of the first node is the input to the second. If the node obtained by collapsing node i into node j is feasible, then the resulting node can be implemented with a logic block. In other words, if node i and node j are combined and the number of inputs to the resulting node is still less than the number of inputs to the logic block, then the resulting node can be implemented with one logic block and is considered *feasible*. However, it should be taken into account that this process increases the number of wiring connections that are necessary, and possibly the delay.

Covering is a type of global elimination; that is, instead of looking a local pairs of nodes, it tries to find a global solution. This method of node minimization attempts to identify *supernodes* in the network. A supernode of node i is called S_i, and is a cluster of nodes that consists of the node i and some other nodes that are fan-ins to node i. The maximum inputs to the supernode S_i must be less than the number of inputs to the logic block, and if node j is in the supernode S_i, then all nodes on the path from node j to node i must also be in S_i. The goal is to find the smallest set of supernodes covering the entire network. However, in finding the optimal set of supernodes, the set is constrained such that each input to each supernode in the set is either a primary input or an output from another supernode in the set. This becomes a *binate covering problem*, which is a constrained form of the NP-hard set covering problem [SVGR93]. There are no general effective heuristics or fast exact algorithms for this problem, although greedy heuristics are used with reasonable results.

The final method, *merging*, is implementation and FPGA architecture dependent. If the exact architecture of the logic block is known, then nodes can be merged to take advantage of the properties of the logic blocks.

After decomposition, reduction, and packing, the resulting network of logic blocks is then placed and routed.

Placement consists of assigning each logic block to a specific location in the FPGA. Routing consists of selecting the wire segments that must be connected on the FPGA in order to realize the desired function.

3.4 ITEM - A Tool for Logic Synthesis and Technology Mapping

In the previous chapter, logic synthesis in general was introduced. Having examined in detail two methods of representing Boolean functions, we now re-examine logic synthesis, specifically to FPGA target technologies.

In this research, the tool chosen for performing logic synthesis is a package called Item. Item stands for *If-Then-Else-Minimizer*. Item is a software package that is part of a project from the University of California, Santa Cruz. The project consists of building a multi-level logic minimizer that is based on ITE-DAGs.

Item splits logic synthesis into a technology independent minimization phase and a technology mapping phase. The first phase tries to find improvements that would apply to almost any target technology. The second phase attempts to find a circuit that can be implemented in the target technology and also takes advantage of the improvements found in the first phase.

Item uses ITE-DAGs internally to represent the logic functions.

3.4.1 Technology Independent Minimization in Item

In Item, the method of minimization is specified by the user. The function is automatically converted to ITE-DAGs when it is read from the input file. When the function has been converted and stored in memory, the user can invoke one or more of the following commands for minimizing the function:

transform This command transforms the given expressions or ITE-DAG(s) in memory as specified by the user. The transformations used can consist of transforming to canonical ITE-DAGs, BDDs or ROBDDs, using what the designers called

Local Factor transformations[4], or stripping off all nodes except principal inputs and outputs.

bcov This command performs block covering. This extracts common subexpressions from a network of functions using the two-column rectangle replacement algorithm. Details are given by [Kar89].

obdd This command converts the expressions given or the ITE-DAG(s) in memory to canonical BDDs.

order This command computes a new variable order for the expressions given or for the ITE-DAG(s) in memory. It is then stored as the current variable order to be worked with. A number of methods can be specified for this, or an explicit ordering for the variables can be given.

In this research the methods used for minimization consist of converting the function to ROBDD representation and then testing different orderings of the variables.

The step following the minimization is the technology mapping step.

3.4.2 Technology Mapping in Item

The command for technology mapping in Item is the *map* command.

When using this command a method must be specified, along with a number of possible options:

map <method> [-h #] [-f #] [-p #] [-r #] [-a areacostfcn] [-d delaycostfcn] [-v] [-m]

[4]the algorithms used in the Local Factor transformations are described in [Kar89].

The mapping methods available are *xmap, xcmap, xtmap, fanout*, and *internal*.

Here we examine the xmap algorithm.

Xmap is described in the Item help files as follows:

> [xmap] – map to table-lookup blocks (fan-in <= lutwidth) treat nodes with fanout>=param($-h$) as high-fanout nodes [Kar91]

This means that a simple mapping technique is used in which each node in the graph representation of the function is examined. If a particular node has fanout greater than or equal to a certain parameter, this node is treated as a *high fanout node*. In other words, if this node is an input to a large number of other nodes, where this number is greater than the parameter given, the node is marked in some way for special consideration. The reason for this is to give the algorithm some guidance in choosing which nodes should be marked as combinational logic blocks.

The options available with xmap are described in the Item help files as follows:

$-v$ verify that fan-in for each gate is <= the parameter $-f$; default is off

$-m$ do not merge table-lookup gates into CLBs (default is to do merge)

$-h$ # specify that nodes with fanout >= # are high fanout nodes; default is 3

All other options are only for use with other mappers.

The option $-v$ enables the verification that each gate, in this case look-up-table blocks, has fewer inputs than the parameter f. The option $-m$ turns off the last phase of the algorithm, which is to merge look-up-table blocks into one CLB wherever possible. The final option, $-h\#$ gives a parameter which is used in selecting nodes for marking during the marking pass of the algorithm. The parameter $-f\#$ is also

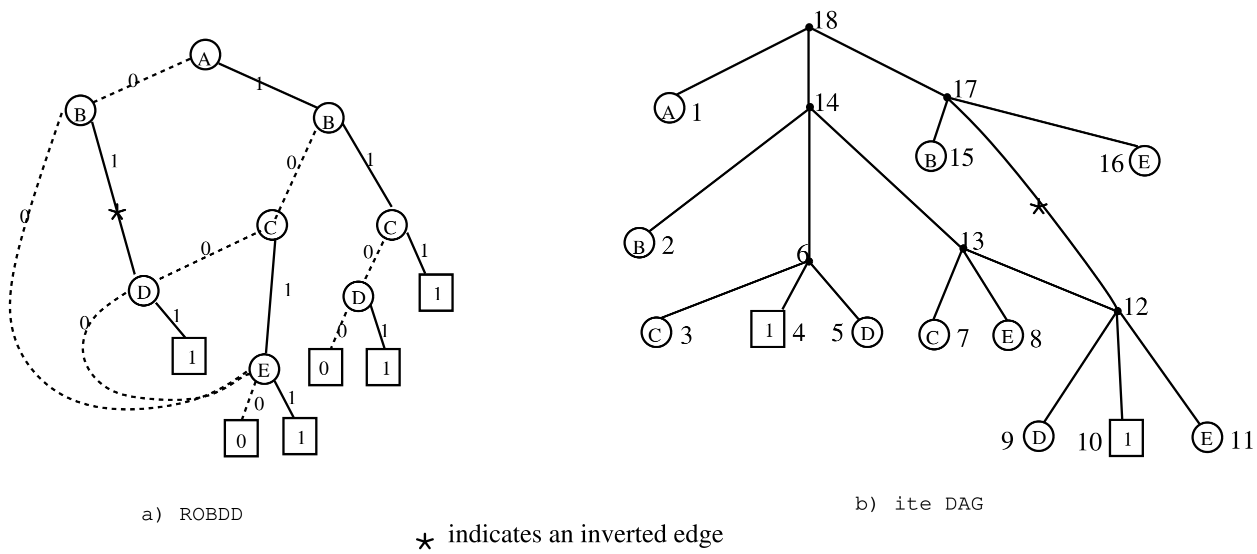


Figure 3.8: The ROBDD and corresponding to the ITE-DAG for the function $F = abc + de + ad\overline{c} + \overline{b}e$

available with xmap and this parameter indicates how many inputs each CLB has. If not indicated, the default is five.

In [Kar91], the technology mapping algorithm is summarized by saying

> ...logic blocks found by xmap are (possibly overlapping) sub-DAGs of the ITE-DAG for the entire circuit. Because of this *direct mapping*, xmap preserves any path-delay-fault testability of the underlying DAG.

The details become clearer upon closer investigation of the algorithm, given below with an example. The function being mapped for this example is the function

$$F = abc + de + ad\overline{c} + \overline{b}e$$

We assume that the parameter f is equal to three for this example. The ROBDD and corresponding ITE-DAG for this function are shown in Figure 3.8. In this example we show the steps as performed on the ITE-DAG although the algorithm can also be performed on the ROBDD.

The xmap algorithm does, in the worst case, four passes of the ITE-DAG representing the function. In the following description of the four passes, f is the number of inputs to the look-up table block used in the FPGA we are mapping to.

1. If $f = 2$, the entire ITE-DAG must be preprocessed, replacing all three-input if-then-else triples (*e.g.* if a then b else c) with either $ab + \overline{a}c$ or $(a + c)(\overline{a} + b)$. This is to guarantee that each node has at most two children. In this case only, the underlying DAG is not preserved. Since $f = 3$ for this example, we can skip immediately to the second pass.

2. The second pass is the *marking phase*. This consists of marking some nodes as outputs of logic blocks, and recording for each marked node N a set of f or fewer marked nodes that could be used as inputs to the logic block. This set forms a vertex cut of marked nodes that separates the node N from the primary inputs. A vertex cut is a subset of vertices in a graph such that the original graph minus this subset of vertices is disconnected. It is good to have this set as small as possible so that it is easier to merge with another cut in a shared logic block. Details on how nodes are chosen to be marked are outlined below.

   This pass is a straightforward traversal of the ITE-DAG in which nodes are visited only after all their inputs have been visited. As each node is visited, if it is a primary input or output, it is automatically marked. Otherwise, a set called a *signal set* is recorded. A signal set lists all of the marked nodes that are needed to compute the function of the node in consideration. Once a node is marked, its signal set consists only of itself, and the marked nodes that were previously in its signal set are recorded as gate inputs. If a node is

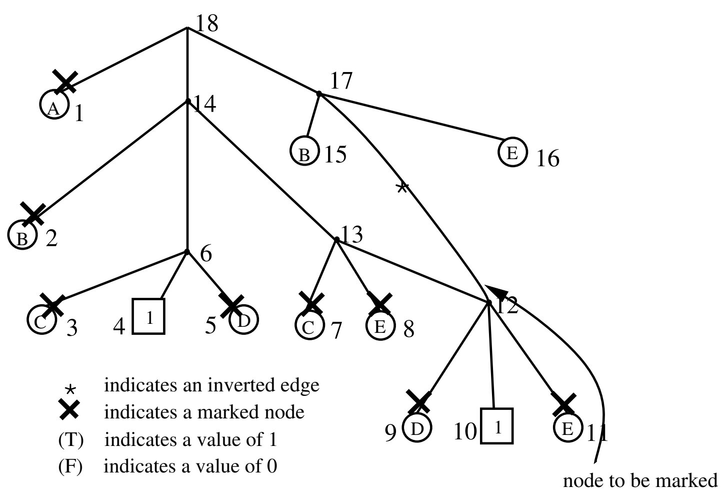


Figure 3.9: The ITE-DAG for the function $F = abc + de + ad\overline{c} + \overline{b}e$ after reducing the fan-in on node 13.

node	signal set	gate inputs (only for marked nodes)
1	{1}	{A}
2	{2}	{B}
⋮	⋮	⋮
12	{9, (T), 11}	
13	{7, 8, 9, (T), 11}	
after node 13's signal set by marking node 12:		
12	{12}	{D, (T), E}
13	{7, 8, 12}	

Figure 3.10: Table showing how the signal sets for the marked nodes change during the marking pass of the xmap algorithm.

reached whose signal set size is greater than f, this node is called an *overflow node*. When an overflow node is reached, the size of its signal set is reduced by marking one or more of its descendants, until its signal set size is less than or equal to f. In choosing the descendants to mark, the nodes that have been marked as high-fanout nodes are chosen first.

In Figure 3.9 the ITE-DAG for the function $F = abc + de + ad\overline{c} + \overline{b}e$ is shown with the nodes labelled in the order they are encountered as the ITE-DAG is traversed. At node 13, the traversal encounters the first node in which the fan-in

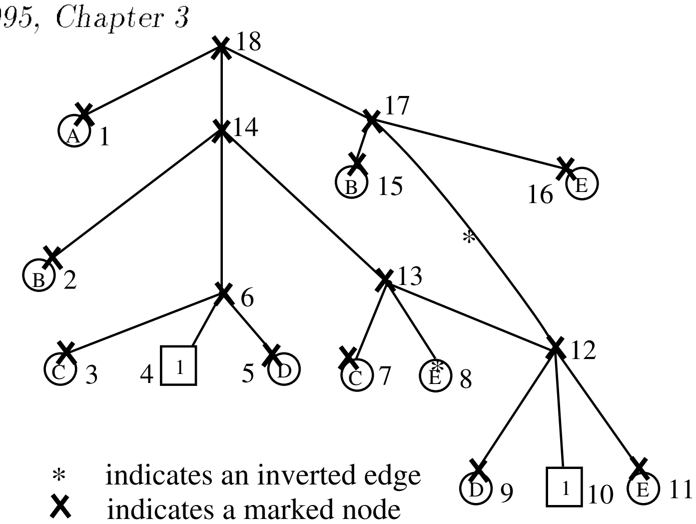


Figure 3.11: The ITE-DAG for the function $F = abc + de + ad\overline{c} + \overline{b}e$ after the entire marking pass.

is greater than three. The next step is to mark some of its descendant nodes in order to reduce the fan-in to the node in question. When the nodes are marked, the signal sets for all nodes above that node are updated, as are the gate input sets for the nodes. Figure 3.9 shows the ITE-DAG after reducing the fan-in of node 13 to less than three, and the chart in Figure 3.10 shows the changes to the pertinent signal and gate input sets.

This step is repeated for each node in the traversal.

When the marking pass is complete, each node is either marked or has a signal set size less than or equal to f. All of the primary inputs and output are marked. Figure 3.11 shows the resulting ITE-DAG after completion of the marking pass.

3. The third pass is the logic block selection pass. The polarity for each marked node is chosen and the gate function needed to compute the node is determined.

   In this pass, inverters are created when a primary output is a negation of a primary input, or when both polarities of a primary output are needed. For each marked node that is not a primary input, a logic block must be created

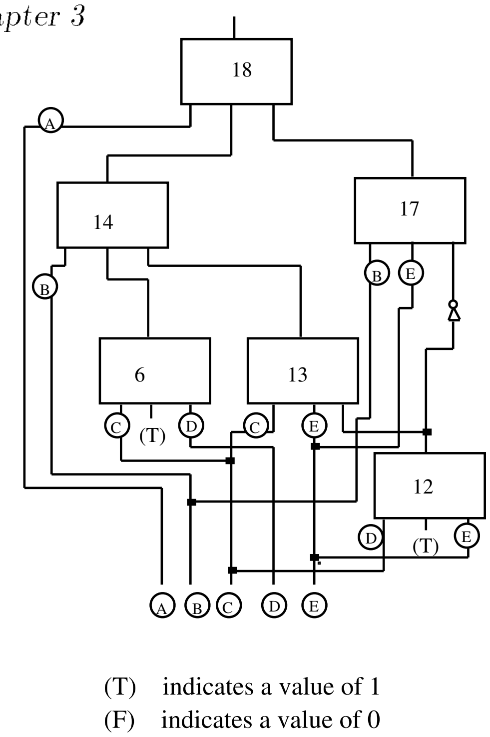


Figure 3.12: The LUT-circuit for the function $F = abc + de + ad\overline{c} + \overline{b}e$.

and the appropriate inputs chosen. These inputs can either be the set of gate inputs as recorded when the node was marked, or another set of legal inputs can be created by looking at all the marked descendants of the node closest to it. The algorithm chooses the smaller of the two sets at the inputs to the logic block.

For our example, the circuit corresponding to the marked ITE-DAG that resulted from step 2 is shown in Figure 3.12. Note that no merging has been performed before realizing this circuit of LUTs.

4. The fourth and final pass is the merge pass. This consists of trying to find nodes that can be merged into one logic block, wherever possible. This is done using a greedy algorithm to get a good, but usually not optimal, matching.

3.5 Summary

The purpose of this chapter is to provide some background information about synthesizing a function from an initial description to a FPGA mapping.

The first step is to describe the function using some standard form. This chapter discusses a number of representations, in particular reduced ordered binary decision diagrams (ROBDDs). The next step is the technology mapping, where decomposition, reduction, and packing are performed. This work employs a technology mapping tool called Item which is also described.

Chapter 4

Introduction to Autocorrelation Coefficients

In the previous chapter, different methods of representing functions in the Boolean domain are discussed. In this chapter we present two function representations that lie in the *spectral domain*. Unlike the Boolean domain, values in the spectral domain can be any integer. Transforms exist for converting from one domain to the other, and much work has been done on the use of spectral representations [Kar76][HMM85].

A function can be described in the spectral domain through the use of *spectral function coefficients*. Based on these spectral coefficients we can derive the values of a set of *autocorrelation coefficients*. In this chapter, these different types of coefficients are described and defined.

The reason for this interest in another way of describing functions is to find some easy way of getting information about the function that can be used in finding variable orderings. When discussing the ROBDD of a function, it is mentioned that this is a useful structure as an intermediate representation in mapping the function to FPGAs.

However, the ordering of the function's variables has, in some cases, a very large effect on the size of the ROBDD. Thus we are very interested in ways of finding orderings that reduce this size.

In the following chapters, the information about a function that is available in its autocorrelation coefficients is applied to finding variable orderings that result in smaller, if not optimal, ROBDDs.

We first present an introduction to the spectral and autocorrelation coefficients.

4.1 Spectral Coefficients

Previously in this work we assumed that a circuit of n inputs is represented by a function $f(X)$ where $X = \{x_n x_{x-1} ... x_1\}$ is a set of Boolean variables, $x \in \{0, 1\}$. However, a circuit can be represented also in the Boolean domain where the Boolean variables are $x \in \{+1, -1\}$. This is simply another representation of the function describing the circuit in question.

Using either of these representations, there is a unique output pattern, also in the Boolean domain, for each unique input pattern. If we consider each input pattern to be a description of the function at one point in space, then one can see that the function is only described at each of these points.

It may, however, be useful to discern information about the entire function from examining a single output, instead of only information about the corresponding input. What is needed is an alternate representation, using more than two values. Such a representation using integers is said to be contained in the *spectral domain*.

In order to transform the data from the Boolean domain to the spectral domain, a transform matrix $\mathbf{T}$ is required. There must be no loss of any information in the transformation, and the process must be easily reversible. In the process of recoding the data, some patterns may become more apparent.

4.1.1 Definition of the Spectral Coefficients

The spectrum of $f(X)$ is given by $\mathbf{R} = \mathbf{T}^n \cdot \mathbf{Z}$ where $\mathbf{R}$ consists of the resulting spectral coefficients, $\mathbf{Z}$ is the output of the function $f(X)$ written as a vector, and $\mathbf{T}^n$ is the nth order transform matrix, of size $2^n \cdot 2^n$. The simplest form of transform matrix $\mathbf{T}^n$ is given by the recursive definition of $\mathbf{T}$ in the Hadamard ordering, as shown below.

$$\mathbf{T}^0 = [1] \quad \mathbf{T}^n = \begin{bmatrix} \mathbf{T}^{n-1} & \mathbf{T}^{n-1} \\ \mathbf{T}^{n-1} & -\mathbf{T}^{n-1} \end{bmatrix}$$

As an example the Hadamard transform matrices for $\mathbf{T}^0$, $\mathbf{T}^1$, and $\mathbf{T}^2$ are shown below.

$$\mathbf{T}^0 = [1] \quad \mathbf{T}^1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \mathbf{T}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The *Rademacher-Walsh spectrum* is obtained using the same transform $\mathbf{R} = \mathbf{T}^n \cdot \mathbf{Z}$ except that the transform matrix $\mathbf{T}^n$ is the Rademacher-Walsh matrix. The Rademacher-Walsh matrix is a complete set of functions that are generated by taking combinations of a reduced set of functions known as the Rademacher functions [Hur78]. Figure 4.1 graphically describes the Rademacher functions for $n = 3$.

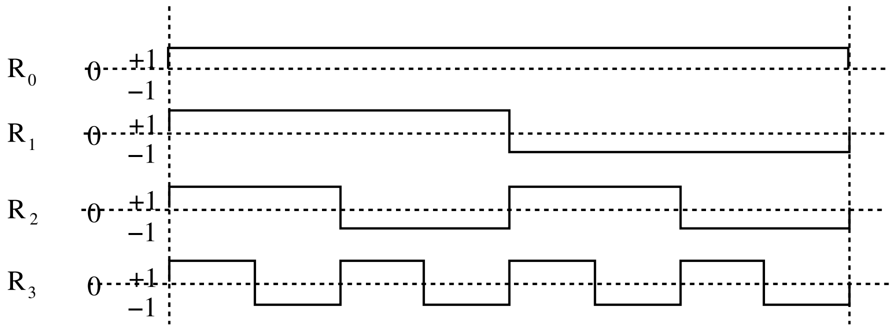


Figure 4.1: The Rademacher functions for n=3.

The Rademacher-Walsh matrix obtained from these functions is shown below.

$$\begin{array}{l} R_0 \\ R_1 \\ R_2 \\ R_3 \\ R_1R_2 \\ R_1R_3 \\ R_2R_3 \\ R_1R_2R_3 \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

4.1.2 Calculating the Spectral Coefficients

The transform for obtaining the spectral coefficients is $\mathbf{R} = \mathbf{T}^n \cdot \mathbf{Z}$. This is simply a matrix multiplication. However, for large values of n, this becomes quite a lengthy process. For this reason, a fast transform method is used, which is described here using what is often referred to as "butterfly" diagrams [HMM85]. The process is shown here for the Hadamard transform, but can be modified for transforms in alternative row orderings such as the Rademacher-Walsh transform.

The fast transform consists of adding or subtracting the elements of $\mathbf{Z}$ in a pre-

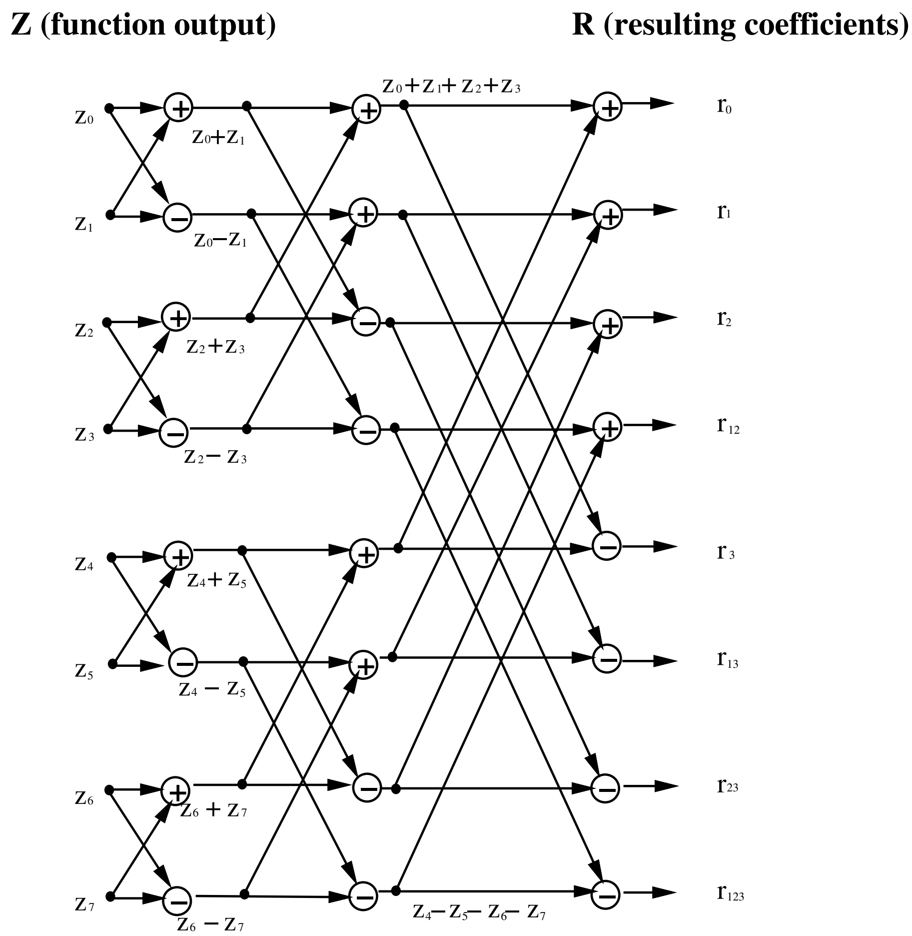


Figure 4.2: The fast transform for the Hadamard transform for $n = 3$.

scribed order. By doing this, the number of operations that must be performed can be reduced from $2^n \cdot 2^n$ to $2^n \cdot n$ [HMM85]. The process is shown in Figure 4.2. In this Figure, the original elements of **Z** are given as z_i, $i = 0...2^n - 1$, and the resulting spectral coefficients are marked as r_j, where j corresponds to the labels for each row in the Rademacher-Walsh matrix on the previous page. For each $+$ or $-$, the resulting value is the addition or subtraction of the values indicated by the two arrows pointing towards the operation symbol. By counting the number of operations, we can see that only 24 operations are required for $n = 3$, which is $2^n \cdot n$.

row	x_1	x_2	x_3	x_4	Z_0
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	1

Figure 4.3: The truth table for the function $f_0 = \overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4} + \overline{x_1}\ \overline{x_2}\ x_3\ x_4 + x_2\ \overline{x_3}\ x_4 + x_1\ x_2\ x_4 + x_1\ x_3\ \overline{x_4}$.

4.1.3 An Example of Calculating the Rademacher-Walsh Spectrum

As an example we calculate the Rademacher-Walsh coefficients for the function

$$f_0 = \overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4} + \overline{x_1}\ \overline{x_2}\ x_3\ x_4 + x_2\ \overline{x_3}\ x_4 + x_1\ x_2\ x_4 + x_1\ x_3\ \overline{x_4}$$

The truth table for this function is shown in Figure 4.3. The truth vector for $f_0(X)$ is referred to as $\mathbf{Z}_0$.

The resulting set of spectral coefficients to define the function f_0 are deduced using $\mathbf{R}_0 = \mathbf{T}^4 \cdot \mathbf{Z}_0$. Figure 4.4 shows these coefficients.

$$
\begin{array}{l}
R_0 \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_{12} \\ R_{13} \\ R_{14} \\ R_{23} \\ R_{24} \\ R_{34} \\ R_{123} \\ R_{124} \\ R_{134} \\ R_{234} \\ R_{1234}
\end{array}
\cdot
\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}
=
\begin{bmatrix} 7 \\ -1 \\ -1 \\ -1 \\ -1 \\ 3 \\ 3 \\ -1 \\ -1 \\ 3 \\ -1 \\ -1 \\ -1 \\ 3 \\ 3 \\ 3 \end{bmatrix}
$$

Figure 4.4: The spectral coefficient vector $\mathbf{R}_0$ for the function vector $\mathbf{Z}_0$.

4.2 Autocorrelation Coefficients

The *autocorrelation coefficients* of a function are another representation in the spectral domain. Again, there is no loss of data in the transformation. The new representation is simply a recoding of the data. The autocorrelation coefficients of a function are calculated using the *autocorrelation function.*

4.2.1 The Definition of the Autocorrelation Function

The autocorrelation function is

$$B(u) = \sum_{v=0}^{2^n-1} f(v)f(v \oplus u) \tag{4.1}$$

where the function is $f(X)$, $X = x_n x_{n-1} ... x_1$, $u = \sum_{i=1}^{n} u_i 2^{i-1}$, $v = \sum_{i=1}^{n} v_i 2^{i-1}$, and n is the number of inputs. The symbol $\oplus$ represents the bitwise exclusive-or function. The number of ones in u is called the *weight* of u and is denoted by $\parallel u \parallel$[Kar76].

Simply put, the autocorrelation function computes a measure of similarity between the function in question and the same function that has been modified.

For multiple output functions a second step must be performed. In this case the autocorrelation function is referred to as the *total autocorrelation function*, and is defined as

$$B(u) = \sum_{i=0}^{m-1} B_i(u) = \sum_{i=0}^{m-1} \sum_{v=0}^{2^n-1} f_i(v) f_i(v \oplus u) \tag{4.2}$$

where m is the number of outputs and the function $F = f_0 f_1 ... f_{m-1}$.

4.2.2 Calculating the Autocorrelation Coefficients

Using Equation 4.2 to calculate the autocorrelation coefficients for a function requires $O(m2^{2n})$ operations. However, there is a significantly faster method for calculating the autocorrelation coefficients. Using the spectrum $\mathbf{R}_i$ of the function, as calculated above, the autocorrelation function can be computed as

$$\mathbf{B}_i = \frac{1}{2^n} \mathbf{T}^n \mathbf{R}_i^2 \tag{4.3}$$

It should be pointed out that ${\mathbf{R}_i}^2$ is defined as the spectral vector $\mathbf{R}$ for the ith function output with each term squared. Each corresponding element of $\mathbf{B}_i$ is then summed over the number of outputs to calculate the total autocorrelation coefficients, $\mathbf{B}$. Using this method requires $O(mn2^n)$ operations, assuming a fast method is used to calculate $\mathbf{R}_i$[Kar76].

4.2.3 An Example of Calculating the Autocorrelation Spectrum

When using Equation 4.3 to calculate the autocorrelation coefficients is useful to break the process down into 5 steps.

1. Calculate $\mathbf{R}_i$, where $i = 0..m-1$ and m is the number of outputs of the function.

row	x_1	x_2	x_3	x_4	f_0	f_1	f_2
0	0	0	0	0	1	0	1
1	0	0	0	1	0	0	0
2	0	0	1	0	0	0	0
3	0	0	1	1	1	0	1
4	0	1	0	0	0	1	1
5	0	1	0	1	1	1	0
6	0	1	1	0	0	0	0
7	0	1	1	1	0	1	1
8	1	0	0	0	0	1	1
9	1	0	0	1	0	0	0
10	1	0	1	0	1	1	0
11	1	0	1	1	0	1	1
12	1	1	0	0	0	0	0
13	1	1	0	1	1	0	1
14	1	1	1	0	1	0	1
15	1	1	1	1	1	1	0

Figure 4.5: The truth table for the multiple output function F.

2. For each $\mathbf{R}_i$ calculate $\mathbf{R}_i^2$.

3. For each $\mathbf{R}_i^2$ premultiply the transform matrix $\mathbf{T}^n$.

4. Multiply the results of the previous step by $\frac{1}{2^n}$. Each resulting matrix is referred to as $\mathbf{B}_i$.

5. Finally, sum the corresponding elements of each $\mathbf{B}_i$. The resulting matrix is the matrix $\mathbf{B}$, the total autocorrelation coefficients.

An example is shown in the following Figures. Figure 4.5 gives the truth table of the function F for which the total autocorrelation coefficients are being calculated.

Step 1 is to calculate $\mathbf{R}_i$ for each of the outputs. In this case, there are 3 outputs, f_0, f_1, and f_2. The resulting matrices are shown in Figure 4.6.

$$\mathbf{R}_0 = \begin{bmatrix} 7 \\ -1 \\ -1 \\ -1 \\ -1 \\ 3 \\ 3 \\ -1 \\ -1 \\ 3 \\ -1 \\ -1 \\ -1 \\ 3 \\ 3 \\ 3 \end{bmatrix} \quad \mathbf{R}_1 = \begin{bmatrix} 7 \\ -1 \\ -1 \\ -1 \\ -1 \\ -5 \\ 3 \\ -1 \\ -1 \\ 3 \\ 3 \\ -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \quad \mathbf{R}_2 = \begin{bmatrix} 8 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 4 \\ 4 \\ -4 \end{bmatrix}$$

Figure 4.6: The matrices $\mathbf{R}_0$, $\mathbf{R}_1$, and $\mathbf{R}_2$ as calculated using the Rademacher-Walsh transform.

$$\mathbf{R}_0^2 = \begin{bmatrix} 49 \\ 1 \\ 1 \\ 1 \\ 1 \\ 9 \\ 9 \\ 1 \\ 1 \\ 9 \\ 1 \\ 1 \\ 1 \\ 9 \\ 9 \\ 9 \end{bmatrix} \quad \mathbf{R}_1^2 = \begin{bmatrix} 49 \\ 1 \\ 1 \\ 1 \\ 1 \\ 25 \\ 9 \\ 1 \\ 1 \\ 9 \\ 9 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{R}_2^2 = \begin{bmatrix} 64 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 16 \\ 0 \\ 0 \\ 16 \\ 16 \\ 16 \end{bmatrix}$$

Figure 4.7: The matrices $\mathbf{R}_0^2$, $\mathbf{R}_1^2$, and $\mathbf{R}_2^2$.

Step 2 is to square each matrix $\mathbf{R}_i$. The results of this step are shown in Figure 4.7. Step 3 consists of multiplying the transform matrix $\mathbf{T}^n$ by the squared $\mathbf{R}_i$'s. This step can also be performed using the fast method described above. The results are shown in Figure 4.8. Step 4 is to multiply each matrix $\mathbf{B}_i$ by $\frac{1}{2^n}$. The resulting matrices are shown in Figure 4.9.

The final step is only necessary for multiple output functions. It consists of summing the corresponding elements of each of the m matrices $\mathbf{B}_i$. The process and its results are shown in Figure 4.10.

$$2^n \cdot \mathbf{B}_0 = \begin{bmatrix} 112 \\ 32 \\ 16 \\ 48 \\ 32 \\ 48 \\ 48 \\ 64 \\ 64 \\ 48 \\ 16 \\ 32 \\ 48 \\ 48 \\ 64 \\ 64 \end{bmatrix} \quad 2^n \cdot \mathbf{B}_1 = \begin{bmatrix} 112 \\ 64 \\ 32 \\ 64 \\ 32 \\ 0 \\ 64 \\ 32 \\ 32 \\ 64 \\ 0 \\ 32 \\ 64 \\ 32 \\ 64 \\ 96 \end{bmatrix} \quad 2^n \cdot \mathbf{B}_2 = \begin{bmatrix} 128 \\ 0 \\ 32 \\ 32 \\ 64 \\ 96 \\ 96 \\ 64 \\ 64 \\ 96 \\ 32 \\ 64 \\ 32 \\ 96 \\ 64 \\ 64 \end{bmatrix}$$

Figure 4.8: The matrices $\mathbf{B}_0$, $\mathbf{B}_1$, and $\mathbf{B}_2$ before being multiplied by $\frac{1}{2^n}$.

$$\mathbf{B}_0 = \begin{bmatrix} 7 \\ 2 \\ 1 \\ 3 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \\ 3 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \end{bmatrix} \quad \mathbf{B}_1 = \begin{bmatrix} 7 \\ 4 \\ 2 \\ 4 \\ 2 \\ 0 \\ 4 \\ 2 \\ 2 \\ 4 \\ 0 \\ 2 \\ 4 \\ 2 \\ 4 \\ 6 \end{bmatrix} \quad \mathbf{B}_2 = \begin{bmatrix} 8 \\ 0 \\ 2 \\ 2 \\ 4 \\ 6 \\ 6 \\ 4 \\ 4 \\ 6 \\ 2 \\ 4 \\ 2 \\ 6 \\ 4 \\ 4 \end{bmatrix}$$

Figure 4.9: The matrices $\mathbf{B}_0$, $\mathbf{B}_1$, and $\mathbf{B}_2$.

$$\begin{bmatrix} 7 \\ 2 \\ 1 \\ 3 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \\ 3 \\ 1 \\ 2 \\ 3 \\ 3 \\ 4 \\ 4 \end{bmatrix} + \begin{bmatrix} 7 \\ 4 \\ 2 \\ 4 \\ 2 \\ 0 \\ 4 \\ 2 \\ 2 \\ 4 \\ 0 \\ 2 \\ 4 \\ 2 \\ 4 \\ 6 \end{bmatrix} + \begin{bmatrix} 8 \\ 0 \\ 2 \\ 2 \\ 4 \\ 6 \\ 6 \\ 4 \\ 4 \\ 6 \\ 2 \\ 4 \\ 2 \\ 6 \\ 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 22 \\ 6 \\ 5 \\ 9 \\ 8 \\ 9 \\ 13 \\ 10 \\ 10 \\ 13 \\ 3 \\ 8 \\ 9 \\ 11 \\ 12 \\ 14 \end{bmatrix} = \mathbf{B}$$

Figure 4.10: Summing the matrices $\mathbf{B}_i$ to calculate the total autocorrelation coefficients, $\mathbf{B}$.

$$\begin{array}{l} R_0 \\ R_1 \\ R_2 \\ R_3 \\ R_{12} \\ R_{13} \\ R_{23} \\ R_{123} \end{array} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

Figure 4.11: The Rademacher-Walsh transform for $n = 3$.

4.3 The Meaning of the Spectral and Autocorrelation Coefficients

This section discusses the meaning of the spectral and autocorrelation coefficients. It is necessary to understand what the coefficients mean before attempting to make use of them.

4.3.1 Spectral Coefficients

To get some idea of what the spectral coefficients mean, we investigate the Rademacher-Walsh functions for $n = 3$. The Rademacher-Walsh transform, for $n = 3$ is shown in $\{+1, -1\}$ notation in Figure 4.11.

Figure 4.12 again shows the Rademacher-Walsh transform for $n = 3$ recoding $+1, -1$ as $0, 1$, so that the relationship between the Rademacher-Walsh functions that form the rows of the matrix and the three input variables x_1, x_2, and x_3 is easier to see. The Figure also shows the three input variables, and on the right, the function that each row of the transform matrix represents.

The resulting coefficients are a numeric measure of similarity between the original x1x2x3

		000	001	010	011	100	101	110	111	
	R'_0	0	0	0	0	0	0	0	0	constant 0
Converted Rademacher– Walsh functions	R'_1	0	0	0	0	1	1	1	1	x1
	R'_2	0	0	1	1	0	0	1	1	x2
	R'_3	0	1	0	1	0	1	0	1	x3
	R'_{12}	0	0	1	1	1	1	0	0	x1 xor x2
	R'_{13}	0	1	0	1	1	0	1	0	x1 xor x2
	R'_{23}	0	1	1	0	0	1	1	0	x2 xor x3
	R'_{123}	0	1	1	0	1	0	0	1	x1 xor x2 xor x3

Figure 4.12: The Rademacher-Walsh transform matrix for $n = 3$, with the corresponding function for each row.

function and the function represented by the matrix row. Coefficients that compare the original function to a Rademacher-Walsh function that corresponds to only one variable, *e.g.* $x1$, are called *first order* coefficients. Coefficients that compare the function to a function corresponding to a combination of two variables, *e.g.* $x_1 \oplus x_2$ are referred to as *second order* coefficients. This notation continues up to n^{th} *order* coefficients, where n is the number of input variables to the original function.

4.3.2 Autocorrelation Coefficients

The autocorrelation coefficients of a function are coefficients that give a measure of similarity between the function f being examined and the same function f that has been modified in some way. In this case, the first order coefficients indicate the similarity between the functions f and g, where $g(x_1, ...x_i, ...x_n) = f(x_1, ...\overline{x_i}, ...x_n)$.

The second order coefficients indicate the similarity between f and h where h is function f with two variables inverted, and so on. This can be demonstrated by examining the formal definition for the autocorrelation function:

$$B(u) = \sum_{i=0}^{m-1} B_i(u) = \sum_{i=0}^{m-1} \sum_{v=0}^{2^n-1} f_i(v) f_i(v \oplus u) \tag{4.4}$$

where m is the number of outputs, n is the number of inputs, and the function $F = f_0 f_1 ... f_{m-1}$.

As an example we use the function

$$f_0 = \overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4} + \overline{x_1}\ \overline{x_2}\ x_3\ x_4\ + x_2\ \overline{x_3}\ x_4 + x_1\ x_2\ x_4 + x_1\ x_3\ \overline{x_4}$$

The truth vector for this function is shown in Figure 4.5 under the column marked f_0. This truth vector is referred to as Z_0.

For $u = 0$ the result is

$$\begin{aligned} B_0(0) &= f_0(0000) f_0(0000 \oplus 0000) + f_0(0001) f_0(0001 \oplus 0000) + ...+ \\ &\quad f_0(1111) f_0(1111 \oplus 0000) \\ &= 1+0+0+1+0+1+0+0+0+0+1+0+0+1+1+1 \\ &= 7 \end{aligned}$$

This shows that there are 7 minterms in this function. For $u = 1$ the result is

$$\begin{aligned} B_0(1) &= f_0(0000) f_0(0000 \oplus 0001) + f_0(0001) f_0(0001 \oplus 0001) + ...+ \\ &\quad f_0(1111) f_0(1111 \oplus 0001) \\ &= 2 \end{aligned}$$

which shows that at a distance of 1, the function is not significantly similar to itself.

Another way of describing the autocorrelation coefficient is as follows. Each autocorrelation coefficient of a given order gives a measure of the similarity between the values the function takes when a given number of function variables are assigned values that are the inverse of the original values. The number of variables whose assigned values are varied corresponds to the order of the resulting coefficient.

For example, again using the function from Figure 4.5, if $u = 3$ then the calculation of $B_0(3)$ looks like this:

$$\begin{aligned} B_0(3) &= f_0(0000)f_0(0000 \oplus 0011) + f_0(0001)f_0(0001 \oplus 0011) + \ldots + \\ &\quad f_0(1111)f_0(1111 \oplus 0011) \\ &= 4 \end{aligned}$$

The resulting value of 4 is the second order coefficient obtained by varying the variables x_3 and x_4 and comparing the varied function value to the original function value.

It seems reasonable that the information inherent in the autocorrelation coefficients can be put to use in logic synthesis. Specifically, the focus of this research is to investigate whether grouping variables of a function according to the function's autocorrelation coefficients results in a smaller ROBDD representation of the function.

4.4 Summary

In this chapter we introduce the spectral and autocorrelation coefficients of a function. These are another way of representing of a function. The spectra and autocorrelation coefficients are, however, not restricted to the Boolean domain.

The spectral coefficients provide a measure of how similar the function is to another function. The autocorrelation coefficients provide a measure of how similar a function is to itself. The reason for our interest in the autocorrelation coefficients is that the information they provide may be used in the reduction step of technology mapping.

If the function to map to a FPGA is represented by a ROBDD, some variable orderings may result in ROBDDs with fewer nodes. The autocorrelation coefficients may provide information that we can use in finding these variable orderings.

This hypothesis is investigated in the following chapter.

Chapter 5

Experimental Work

The purpose of Chapters 1 through 4 has been to provide the reader with the necessary background material. Chapter 5 goes on to discuss the motivation behind this research.

This research is based on the hypothesis that the autocorrelation coefficients of a function contain information that can be used to generate variable orderings resulting in smaller ROBDDs. In this chapter we consider a number of small, in-depth examples to see whether they offer any support to this hypothesis.

In later sections we discuss the goals of this research, followed by descriptions of the methods developed for using the autocorrelation coefficients to generate variable orderings.

5.1 Motivation

In Chapter 3, a useful method of representing functions for synthesis to FPGAs is introduced. This representation is called a ROBDD. However, the ROBDD representation has one major drawback: the ordering of a function's variables can have a large effect on the size of the resulting ROBDD. The problem of how to find a good ordering is a difficult one, but according to Bryant, a relatively good solution can usually be found by a human with some knowledge of the function in a relatively short amount of time[Bry86].

This is only feasible for relatively small functions, therefore it would be useful for this process to be done automatically. An algorithm for variable ordering based on the autocorrelation coefficients of a function could be a very valuable step in logic synthesis systems.

5.1.1 Autocorrelation Coefficients

Autocorrelation coefficients are presented in detail in Chapter 4. Briefly, they give a measure of how similar a function is to itself when certain input variable(s) are inverted. Thus, this information could be very useful in determining which variables should be grouped together in order to share the most subdiagrams in the ROBDD. If two or more variables have high magnitude first-order coefficients, grouping these variables may result in the similar parts of the function being shared instead of spread apart. Similarly, if two variables have high magnitude second-order coefficients, grouping these variables may also result in more subdiagram-sharing being possible.

5.1.2 Examples

Here some of the above ideas are considered with a number of examples. We first concentrate on one example. In the process of discussing this function we examine the autocorrelation coefficients, Karnaugh map, and finally the ROBDDs for this

x3x4 \ x1x2	00	01	11	10
00				
01				
11		1	1	
10			1	

a)

(0)=3			
(1)=2	(12)=0	(123)=0	(1234)=0
		(124)=0	
	(13)=0	(134)=0	
	(14)=2		
(2)=0	(23)=0	(234)=0	
	(24)=0		
(3)=0	(34)=0		
(4)=2			

b)

Figure 5.1: The Karnaugh map and autocorrelation coefficients for the function $f = x_1x_2x_3 + x_2x_3x_4$.

function. The ROBDDs shown have different orderings in order to demonstrate how the autocorrelation coefficients may be used. After a discussion of the relationship between the autocorrelation coefficients and the function's BDD, a number of other functions are presented in order to see if the same observations made for the initial function hold true.

Example 1

The first function we examine has the sum-of-products expression

$$f = x_1x_2x_3 + x_2x_3x_4$$

and the Karnaugh map as shown in Figure 5.1 part a).

The autocorrelation coefficients for this function are shown in part b) of Figure 5.1. The first column of figures are the first-order coefficients, the second column are the second-order coefficients, and so on. In each column the value in brackets on the left is the variable(s) corresponding to the coefficient value, which is given on the right.

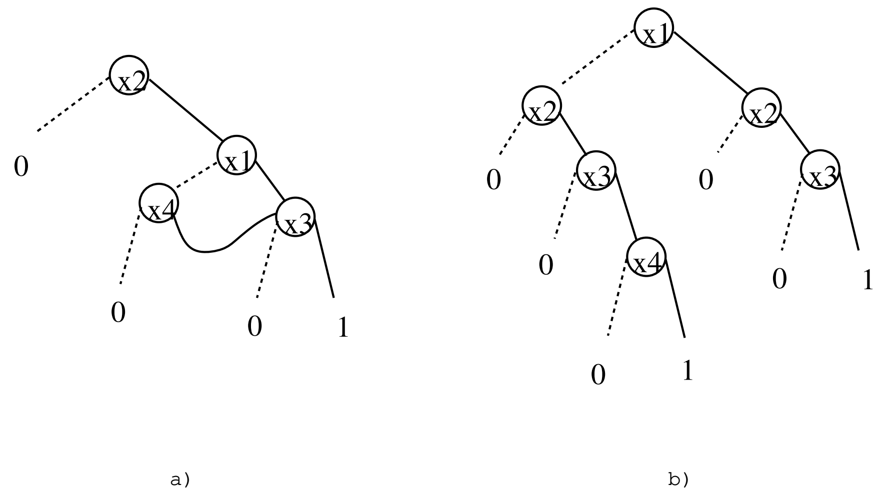


Figure 5.2: Two ROBDDs for the function $f = x_1x_2x_3 + x_2x_3x_4$.

In the autocorrelation coefficients there are high magnitude values for the variables x_4 ($u = 1000$), x_1 ($u = 0001$), and the pair of variables x_1x_4 ($u = 1001$). In the list of coefficients in Figure 5.1 we notice also a coefficient value of 3 corresponding with what appears to be the variable x_0. x_0 is not a variable, but instead corresponds to a coefficient value indicating how similar the function is to itself. In this case, the value 3 informs us that the function has three input combinations for which the output has the value 1.

If a variable ordering is determined such that the two variables x_1 and x_4 are adjacent in the ordering and the resulting ROBDD is built, this ROBDD has only 5 nodes. To calculate this number we assume one node for each decision point in the ROBDD plus one node that is shared between all outputs, that is, 1s and 0s. This ROBDD is shown in Figure 5.2 part a).

In Figure 5.2 part b) the ROBDD resulting from the variable ordering $x_1x_2x_3x_4$ is shown. Clearly the ROBDD in part a) is smaller (fewer nodes).

There are also other possible orderings that group the variables x_1 and x_4 together. To show that it is the pairing of x_1 and x_4 that results in the smaller size and not some

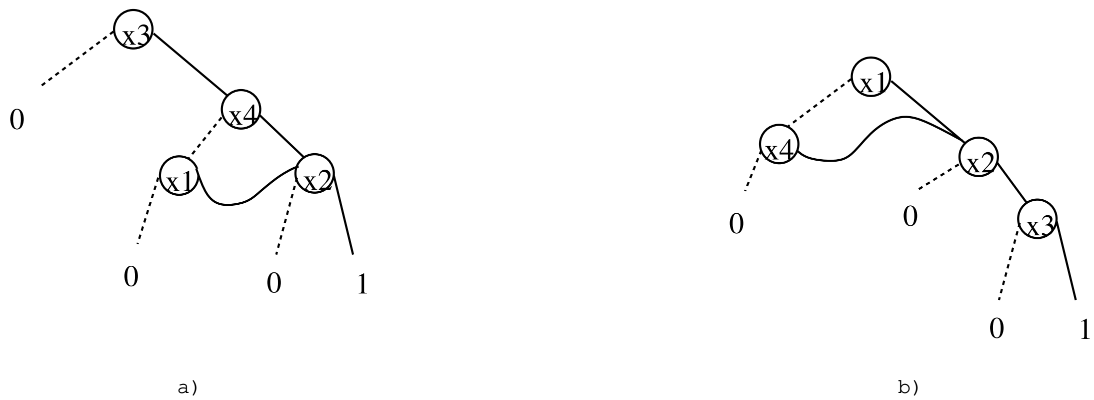


Figure 5.3: Two other ROBDDs for the function $f = x_1x_2x_3 + x_2x_3x_4$.

other feature of the above ordering, the ROBDDs resulting from two other orderings that pair up these two variables are shown in Figure 5.3.

Table 5.1 shows all the possible orderings for the four input variables for this function, and the corresponding number of nodes in the ROBDD built with that ordering.

This table shows that not only do all the orderings with the variables x_1 and x_4 paired together result in ROBDDs with only 5 nodes, but also that all of the ROBDDs with only 5 nodes have the variables x_1 and x_4 grouped together.

Discussion

Before examining other functions we attempt to explain why pairing together variables with high magnitude autocorrelation coefficients results in a smaller ROBDD for this example.

To do this, it is useful to understand the relationship between a function's autocorrelation coefficients and Karnaugh map. Figure 5.4 shows the Karnaugh map and

ordering	number of nodes	ordering	number of nodes
1 2 3 4	7	1 2 4 3	6
1 3 2 4	7	1 3 4 2	6
1 4 2 3	5	1 4 3 2	5
4 2 3 1	7	4 2 1 3	6
4 3 2 1	7	4 3 1 2	6
4 1 2 3	5	4 1 3 2	5
3 2 1 4	5	3 2 4 1	5
3 1 2 4	6	3 1 4 2	5
3 4 2 1	6	3 4 1 2	5
2 1 3 4	6	2 1 4 3	5
2 3 1 4	5	2 3 4 1	5
2 4 1 3	5	2 4 3 1	6

Table 5.1: All possible orderings for four inputs, with the corresponding number of nodes for the ROBDD for the function $f = x_1x_2x_3 + x_2x_3x_4$.

autocorrelation coefficients for the example we are discussing.

The First-Order Autocorrelation Coefficients

The first-order autocorrelation coefficient 2 that is associated with the variable x_4 can be said to correspond to the squares labeled **b** and **c** in the Karnaugh map in Figure 5.4. This is because x_4 has the value 1 in square **b** and 0 in square **c** while the value of the function remains 1 for both squares while all other variable values remain constant. Similarly, the first-order autocorrelation coefficient for the variable x_1 is explained by the function value remaining the same in squares **a** and **b** while the value x_1 takes on in square **a** is the inverse of that in **b**.

The Second-Order Autocorrelation Coefficients

The only remaining high magnitude coefficient is the second-order autocorrelation coefficient for the variable pair x_1x_4. This is explained by looking at the values in squares **a** and **c**. For square **a**, x_1 takes the value 0 and x_4 takes the value 1, while the

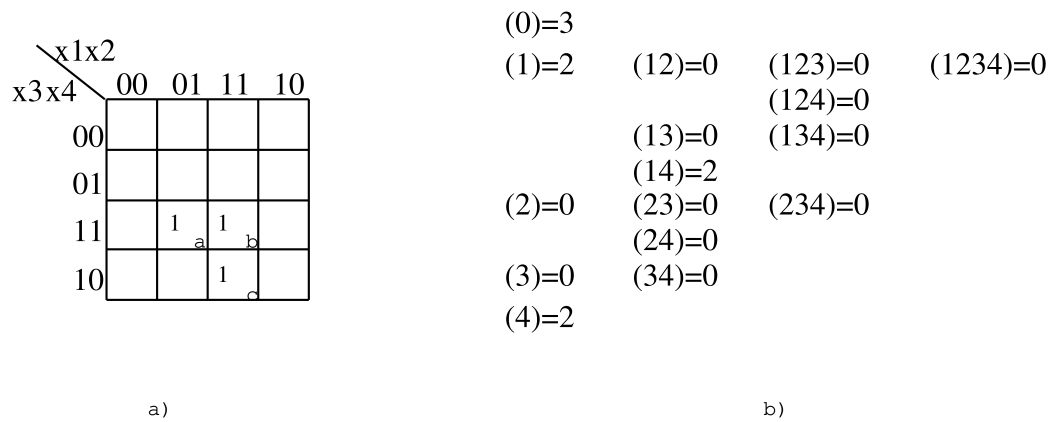


Figure 5.4: The Karnaugh map and autocorrelation coefficients for the function $f = x_1x_2x_3 + x_2x_3x_4$.

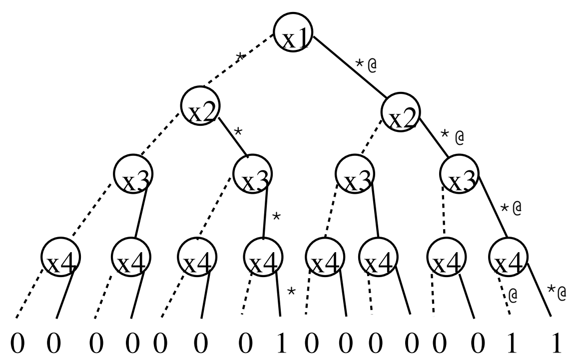


Figure 5.5: The unreduced BDD for the function $f = x_1x_2x_3 + x_2x_3x_4$.

function value is 1. For square **c**, x_1 takes the value 1, x_4 takes the value 0, and again, the function value is 1. To summarize this, from square **a** to square **c**, the values for both x_1 and x_4 are inverted while the function value remains a 1. This results in the value of the 2 for the second-order autocorrelation coefficient for the variable pairing x_1x_4.

The next question is, how does this relate to the ROBDD representing the function? To answer this we examine two unreduced BDDs. The first of these is shown in Figure 5.5. This BDD uses the variable ordering $x_1x_2x_3x_4$. In Figure 5.5, there are 3

$x_1 = 1$ or 0		$x_1 = 1$
$x_2 = 1$		$x_2 = 1$
$x_3 = 1$	OR	$x_3 = 1$
$x_4 = 1$		$x_4 = 1$ or 0
(these paths marked		(these paths marked
with a *)		with a @)

Figure 5.6: Three possible paths through the ROBDD in the previous Figure that reach an output of 1.

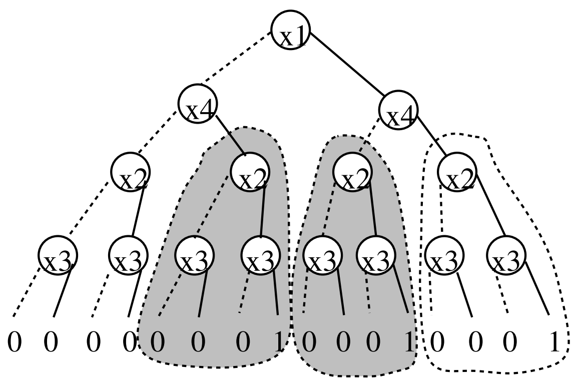


Figure 5.7: A reordered unreduced BDD for the function $f = x_1x_2x_3 + x_2x_3x_4$.

paths from the root (variable x_1) through the BDD that lead to function values of 1. These paths can be described as shown in Figure 5.6.

This corresponds with the information given in the autocorrelation coefficients where x_1 and x_4 both have coefficients with high magnitude values, and to the squares **ab** and **bc** in the Karnaugh map in Figure 5.4.

By reordering the variables for the BDD in shown in Figure 5.7, it is easier to see how the second-order autocorrelation coefficients correspond to the shape of the BDD. The two shaded subdiagrams have identical terminals, and can therefore be shared when reducing the BDD. The terminals with the value 1 in each of the two shaded subdiagrams correspond to the squares **a** and **c** in the Karnaugh map in Figure 5.4. For the shaded subdiagram on the left, the values that x_1 and x_4 take on are 0 and 1, respectively, while for the shaded subdiagram on the right, both variables' values are the inverse of these (*i.e.* x_1 takes the value 1 and x_4 takes the value 0).

So by grouping x_1 and x_4 together because of their large second-order autocorrelation coefficient, there are two subdiagrams that can be shared. Also, by putting x_4 near the top of the ordering (and therefore near the top of the ROBDD) we can see that there is a third subdiagram that also can be shared (shown outlined but not shaded in Figure 5.7).

Thus making use of the first and second-order autocorrelation coefficients, for this example we are able to order the variables so as to share more subdiagrams in the BDD which results in fewer nodes in the final ROBDD.

Example 2

This example is a function with the sum of products expression

$$f = \overline{x_2}\,\overline{x_4} + x_2\overline{x_3}x_4 + x_1x_2x_4 + \overline{x_1}x_3\overline{x_4}$$

The Karnaugh map and autocorrelation coefficients for this function are shown in Figure 5.8.

Looking at the autocorrelation coefficients in Figure 5.8 we see that in the first-order coefficients there are two variables, x_1 and x_3 that have coefficients with a value of 6, while all the other first-order coefficients are of a lesser magnitude. Therefore pairing the variables x_1 and x_3 may result in more sharing of subdiagrams in the resulting ROBDD.

In the second-order autocorrelation coefficients we see that the value of the coefficient for the pairings of x_1x_3 is also of a high magnitude, thus giving another reason

x3x4 \ x1x2	00	01	11	10
00	1			1
01		1	1	
11			1	
10	1	1		1

a)

(0)=8
(1)=6 (12)=2 (123)=2 (1234)=6
(124)=6
(13)=6 (134)=2
(14)=2
(2)=2 (23)=2 (234)=6
(24)=6
(3)=6 (34)=2
(4)=0

b)

Figure 5.8: The Karnaugh map and autocorrelation coefficients for the function $f = \overline{x_2}\,\overline{x_4} + x_2\overline{x_3}x_4 + x_1x_2x_4 + \overline{x_1}x_3\overline{x_4}$.

to group together these variables. However, another second-order coefficient also has the value 6, indicating that possibly the variables x_2 and x_4 should be paired together. To check these hypotheses all possible orderings of the variables have been generated, along with the resulting ROBDD. These values are shown in Table 5.2.

Examining Table 5.2 indicates that all of the orderings resulting in only 5 nodes have the variables x_1 and x_3 adjacent to each other. There is no ordering with these variables grouped together that has more than 6 nodes. However, the other pairing suggested by the autocorrelation coefficients, x_2x_4 seems to have little or no effect on the size of the resulting ROBDD.

Example 3

The third example has the sum-of-products expression

$$f = x_1x_2\overline{x_3} + \overline{x_1}x_2x_3 + \overline{x_1}x_3x_4$$

The Karnaugh map and autocorrelation coefficients for this function are shown in Figure 5.9.

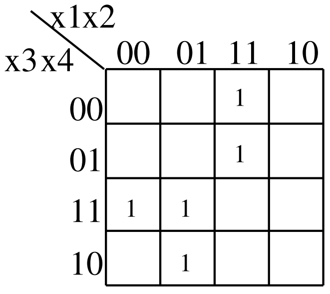
ordering	number of nodes	ordering	number of nodes
1 2 3 4	6	1 2 4 3	7
1 3 2 4	5	1 3 4 2	6
1 4 2 3	7	1 4 3 2	6
4 2 3 1	5	4 2 1 3	5
4 3 2 1	6	4 3 1 2	5
4 1 2 3	6	4 1 3 2	5
3 2 1 4	6	3 2 4 1	7
3 1 2 4	5	3 1 4 2	6
3 4 2 1	7	3 4 1 2	6
2 1 3 4	5	2 1 4 3	6
2 3 1 4	5	2 3 4 1	6
2 4 1 3	6	2 4 3 1	6

Table 5.2: All possible orderings for four inputs, with the corresponding number of nodes for the ROBDD for the function $f = \overline{x_2}\,\overline{x_4} + x_2\overline{x_3}x_4 + x_1x_2x_4 + \overline{x_1}x_3\overline{x_4}$.

In Figure 5.9, the largest first-order autocorrelation coefficients are for the variables x_4 and x_2. The largest second-order coefficient is for the variable pairings x_1x_3, with the pairing x_2x_4 having the second largest second-order coefficient.

This would tend to suggest grouping x_2 and x_4 and grouping x_1 and x_3, but it is not clear which pairing should have the highest priority.

Table 5.3 shows all possible orderings and the corresponding ROBDD size. In this table, the smallest ROBDD has 6 nodes. For most of the orderings resulting in ROBDDs with 6 nodes, the variables x_2 and x_4 are paired together. Out of 24 orderings, there are 10 that result in ROBDDs with 6 nodes. Of these 10 orderings, 8 contain the pairing x_2x_4 or x_4x_2, and 4 contain the pairing x_1x_3 or x_3x_1. Two of



a)

(0)=5
(1)=0 (12)=0 (123)=2 (1234)=2
(124)=0
(13)=4 (134)=4
(14)=0
(2)=2 (23)=0 (234)=0
(24)=2
(3)=0 (34)=0
(4)=4

b)

Figure 5.9: The Karnaugh map and autocorrelation coefficients for the function $f = x_1 x_2 \overline{x_3} + \overline{x_1} x_2 x_3 + \overline{x_1} x_3 x_4$.

ordering	**number of nodes**	**ordering**	**number of nodes**
1 2 3 4	7	1 2 4 3	6
1 3 2 4	7	1 3 4 2	6
1 4 2 3	6	1 4 3 2	7
4 2 3 1	7	4 2 1 3	7
4 3 2 1	7	4 3 1 2	8
4 1 2 3	7	4 1 3 2	8
3 2 1 4	7	3 2 4 1	6
3 1 2 4	7	3 1 4 2	6
3 4 2 1	6	3 4 1 2	7
2 1 3 4	7	2 1 4 3	6
2 3 1 4	7	2 3 4 1	6
2 4 1 3	6	2 4 3 1	6

Table 5.3: All possible orderings for four inputs, with the corresponding number of nodes for the ROBDD for the function $f = x_1 x_2 \overline{x_3} + \overline{x_1} x_2 x_3 + \overline{x_1} x_3 x_4$.

x3x4 \ x1x2	00	01	11	10
00				
01		1	1	
11		1	1	1
10			1	1

a)

(0)=7			
(1)=4	(12)=2	(123)=2	(1234)=2
		(124)=2	
	(13)=4	(134)=2	
	(14)=2		
(2)=4	(23)=2	(234)=2	
	(24)=4		
(3)=4	(34)=2		
(4)=4			

b)

Figure 5.10: The Karnaugh map and autocorrelation coefficients for the function $f = x_1x_3 + x_2x_4$.

the 10 have neither of these pairings.

Example 4

The final example has the sum-of-products expression

$$f = x_1x_3 + x_2x_4$$

The autocorrelation coefficients and Karnaugh map for this function are shown in Figure 5.10.

This function differs from the previous functions in that the first-order autocorrelation coefficients are all of the same magnitude, thus providing little information as to a good ordering. However, in the second-order coefficients there is more information. Here we see that the pairings x_1x_3 and x_2x_4 have larger magnitude coefficients than the rest of the pairings. This would suggest that to result in a smaller ROBDD, a variable ordering should consist of some combination of these two pairs. Once again, Table 5.4 shows all possible orderings and the corresponding ROBDD size.

In Table 5.4 it is clear that the smallest ROBDD has 5 nodes. Also, every ordering consisting of some combination of the variables pairs x_2x_4 and x_1x_3 results in 5 nodes

ordering	number of nodes	ordering	number of nodes
1 2 3 4	7	1 2 4 3	7
1 3 2 4	5	1 3 4 2	5
1 4 2 3	7	1 4 3 2	7
4 2 3 1	5	4 2 1 3	5
4 3 2 1	7	4 3 1 2	7
4 1 2 3	7	4 1 3 2	7
3 2 1 4	7	3 2 4 1	7
3 1 2 4	5	3 1 4 2	5
3 4 2 1	7	3 4 1 2	7
2 1 3 4	7	2 1 4 3	7
2 3 1 4	7	2 3 4 1	7
2 4 1 3	5	2 4 3 1	5

Table 5.4: All possible orderings for four inputs, with the corresponding number of nodes for the ROBDD for the function $f = x_1x_3 + x_2x_4$.

while conversely, every ordering with only 5 nodes consists of some ordering in which one of x_2 and x_4 or x_1 and x_3 appear consecutively.

In this work only the first and second-order autocorrelation coefficients are considered. The main reason for this is that to determine orderings, variables are paired up until all variables are used. With the third-order coefficients and higher, one coefficient value will present $n!$ possible rearrangements of the variables within that grouping (where n is the order of the coefficients), making it infeasible for this to give immediate ordering information.

5.1.3 Expected Results

Thus far the examples shown have been not been completely symmetric and have had at least one ordering that results in a ROBDD of a reasonable size. Completely symmetric functions do not result in any improvement in the size of their ROBDD representation when using autocorrelation coefficients to find an ordering. This is clear for two reasons:

1. If a function is completely symmetric, rearranging the order of its variables does not change the function, so any reordering does not change the size of the ROBDD representation.

2. When calculating the autocorrelation coefficients for a completely symmetric function, they are all equal. This is useful in determining whether or not a function is symmetric before attempting to generate a variable ordering.

There are some examples for which the ROBDD representation of a function always results in an exponential size ROBDD. One such type of function includes the functions that represent the outputs of integer multiplication. For a multiplier with word size n (thus two sets of inputs, each with n bits), there will be 2^n outputs. It has been proven that at least one of these 2^n outputs will require a graph with at least $2^{n/8}$ nodes.[Bry86].

For functions that do not fall into either of these categories, it appears that in the case of these three examples, using the autocorrelation coefficients to order the input variables of a function may often result in a smaller ROBDD representation, if not the optimal size. Moreover, by examining the autocorrelation coefficients for the functions to be synthesized, it may be possible to quickly identify whether or not it is possible to find a good ordering for the function.

5.2 The Experimental Work

In this section we discuss the algorithms that we developed to make use of the autocorrelation coefficients in finding variable orderings that would result in smaller ROBDD function representations.

The logical progression of this research is to go from the simplest method on to the more complicated ones. We begin with an ordering method based only on the first-order autocorrelation coefficients, discussed in the first section below. The section following this describes two methods based on the second order coefficients, and the final section describes an ordering method that makes use of an average value composed of both the first and second order coefficients for each variable.

For each ordering method a brief outline of the steps in the algorithm is given, followed by a small example of the working of the algorithm.

5.2.1 Using the First Order Autocorrelation Coefficients to Find Variable Orderings

This first method used to find orderings is based on only the first-order autocorrelation coefficients of a function, and follows a relatively simple algorithm. Briefly, the variables are sorted according to their first-order autocorrelation coefficients. During the initial testing period for this algorithm it was found that a slightly better result is often achieved if the variables with largest magnitude coefficients are put at the end of the ordering, rather than at the beginning.

An example of the first-order autocorrelation coefficients and the resulting ordering for the function $f = x_1x_2x_3 + x_2x_3x_4$ is shown in Figure 5.11.

5.2.2 Using Second Order Autocorrelation Coefficients to Find Variable Orderings

The methods used to find variable orderings based on the second order autocorrelation coefficients are a little more complex. In fact, two versions of this are developed.

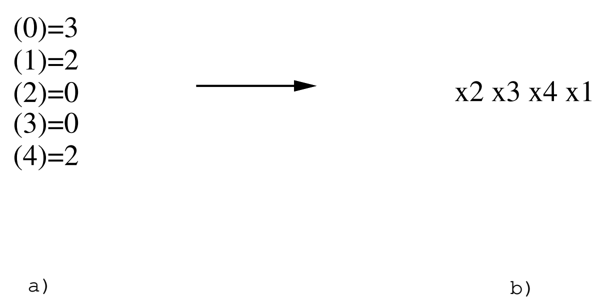


Figure 5.11: The first-order autocorrelation coefficients and resulting ordering for the function $f = x_1x_2x_3 + x_2x_3x_4$.

Method 2 Version a)

The first version of this method initially uses the first-order coefficients in deciding the ordering of the variables. *i.e.*, the first variable to be put into the ordering list is the variable with the largest magnitude first-order coefficient. The next step is to use the second order coefficients to choose a variable to go adjacent to this first variable. This is achieved by searching all the second-order coefficients, and choosing the variable pair that has the largest second-order coefficient and which also contains the first variable. This process of searching for the largest second-order coefficient of a pair containing the previous variable is repeated until all the variables are used.

Here is a brief summary of these steps:

1. Choose the first variable according to which has the largest first-order autocorrelation coefficient.

2. Choose another variable to go adjacent to this variable according to which pair containing the first variable has the largest second-order autocorrelation coefficient.

(0)=14	
(1)=12	(12)=0
	(13)=12
	(14)=0
	(15)=12
(2)=0	(23)=0
	(24)=14
	(25)=0
(3)=12	(34)=0
	(35)=12
(4)=0	(45)=0
(5)=12	

Figure 5.12: The first and second-order autocorrelation coefficients for the function $F = (x_4 \oplus x_2)(x_3 + x_1 + x_5)$.

3. Repeat the previous step until all the variables have been added to the ordering. No variables can be repeated.

For example, the first and second order autocorrelation coefficients for the function $f = (x_4 \oplus x_2)(x_3 + x_1 + x_5)$ are shown in Figure 5.12.

According to the steps of this algorithm, the first variable to add to the ordering is either x_1, x_3, or x_5, as they all have the largest magnitude first-order coefficient, 12. In this example, we choose x_1. The next step is to search the second-order coefficient list for the largest magnitude coefficient of a pair containing x_1. Again, there is more than one choice, so we select the variable pairing $(x_1 x_3)$ which has the coefficient 12. Thus the variable ordering is so far as follows:

$$x_3 \quad x_1$$

Note that the first variable to be selected is actually the last variable in the ordering. Repeating the previous step, we find that the next largest magnitude coefficient for a pair containing x_3 is again the coefficient 12, for the pairing $(x_3 x_5)$. The ordering is now

$$x_5 \;\; x_3 \;\; x_1$$

Once all of the variables have been used, the resulting ordering is

$$x_4 \;\; x_2 \;\; x_5 \;\; x_3 \;\; x_1$$

The steps through this example are summarized here:

1. The first variable is chosen according to the size of the first-order autocorrelation coefficients; in this case the variable chosen is x_1.
2. A second variable is chosen according to which pair containing x_1 has the largest second-order autocorrelation coefficients; the variable chosen here is x_3.
3. Step 2 is repeated, only this time we are looking for a pair containing x_3 and having the largest second-order coefficient; the variable selected is x_5.
4. Step 2 is repeated again to find a pair containing x_5; the pair matching this condition would be x_1x_5 but since x_1 is already in the list, x_2 is chosen.
5. Step 2 is repeated again to find a pair containing x_2; the variable selected is x_4.
6. All variables have been added to the ordering, and the final ordering is

$$x_4 \;\; x_2 \;\; x_5 \;\; x_3 \;\; x_1$$

In building an ROBDD for the function using the default ordering of x_1, x_2, x_3, x_4, x_5 the number of nodes is 11. Using the new ordering, the number of nodes is 8.

Method 2 Version b)

The second version of the ordering algorithm based on the second-order autocorrelation coefficients is slightly more complicated, but similar to the above algorithm. This second version was developed to see if a slightly different method of combining the variables based on their second-order coefficients would produce different results.

This ordering method is based entirely on the second-order coefficients. For this method, the steps are outlined as follows:

1. Find the pair of variables with the largest second-order autocorrelation coefficient.
2. Check to see if either of these variables have already been added to the ordering; if one of them has been then:
   - if the variable is at the beginning of the list, add its pair to the beginning (put it adjacent to the one already in the list)
   - if the variable is at the end of the list, add its pair to the end
   - if the variable is in the middle of the list, add its pair to the beginning of the ordering
3. If neither of the pair is in the ordering, add them to the head of the pairing.
4. If both of the pair are already in the ordering, discard them.
5. Go back to step 1 and continue until either all variables are in the ordering, or the end of the list of coefficients is reached. Each time a pair is used, it is removed from the list.

As an example this ordering method is used on the autocorrelation coefficients from Figure 5.13.

(24)=14
(13)=12
(15)=12
(35)=12
(12)=0
(14)=0
(23)=0
(25)=0
(34)=0
(45)=0

Figure 5.13: The sorted list of second-order autocorrelation coefficients for the function $f = (x_4 \oplus x_2)(x_3 + x_1 + x_5)$.

1. The list of second-order autocorrelation coefficients is first sorted from largest to smallest, as shown in Figure 5.13.

2. The next step is to find the pair with the largest coefficient. Since the list is sorted, this is the first pair in the list. To ensure we don't use this pair again, we remove the this variable pairing from the list. Since this is the first pair, they are simply added to the ordering list, which would then be $x_2 \ x_4$.

3. This step is repeated until all the variables have been added to the ordering list. The variable pair with the next largest coefficient value is found by simply getting the next pair in the sorted list, $x_1 x_3$. This pair is removed from that list and the following steps are performed:

   (a) Check to see if either of the variables in this pair is already in the ordering; if one of them is, then check to see if it is at the head or tail of the ordering; if it is, then add the other variable (the one not in the ordering) to the ordering adjacent to its pair. If this is not possible, add it to the head of the ordering.

(b) If neither of the pair are in the ordering, add them to the head of the pairing. This is the case for the variable pair x_1x_3.

(c) If both of the pair are already in the ordering, discard them.

4. Go back to step 3; continue until either all variables are in the ordering, or the end of the list of coefficients is reached. After the first repetition of these steps the new ordering is as follows:

$$x_1 \;\; x_3 \;\; x_2 \;\; x_4$$

The final ordering, once all the variables have been used is, is as follows:

$$x_5 \;\; x_1 \;\; x_3 \;\; x_2 \;\; x_4$$

If we build a ROBDD based on this ordering, the number of nodes is now 7, one less node than in the previous ordering method's ROBDD, and 4 less nodes than the ROBDD based on the default ordering. 7 is also the smallest possible number of nodes for a ROBDD representing this function.

5.2.3 Using Both the First and Second Order Autocorrelation Coefficients to find Variable Orderings

In the first method, the metric used for ordering the variables is the magnitude of the first order autocorrelation coefficients. In the second two methods, the metric used for ordering the variables is the magnitude of the second order autocorrelation coefficients. In both cases, information is left unused. Thus for this fourth method of ordering, the metric used is a combination of the two. Each first order coefficient is denoted r_i for $i = 1..n$, where n is the number of input variables. Each second order

(0)=14			
(1)=12	(12)=0	(12)=12	
	(13)=12	(13)=36	
	(14)=0	(14)=12	
	(15)=12	(15)=36	
(2)=0	(23)=0	(23)=12	$\longrightarrow$ x4 x2 x5 x3 x1
	(24)=14	(24)=14	
	(25)=0	(25)=12	
(3)=12	(34)=0	(34)=12	
	(35)=12	(35)=36	
(4)=0	(45)=0	(45)=12	
(5)=12			
a)		b)	c)

Figure 5.14: The first and second order autocorrelation coefficients for the function $F = (x_4 \oplus x_2)(x_3 + x_1 + x_5)$, followed by the value $r_i + r_j + r_{ij}$ calculated for each variable pairing and the resulting ordering.

coefficient is denoted r_{ij} for $i = 1..n$ and $j = 1..n$. The metric used in this method is calculated for each pair of variables, and can be described as

$$r_i + r_j + r_{ij}$$

We should note that alternate weights for the different order coefficients could also be tested, but were not in this work. An example is shown in Figure 5.14. In part a) of the figure are the first and second order autocorrelation coefficients for the function. In part b) the metric given above has been calculated for each variable pairing, and in part c), the resulting ordering is given. This ordering is calculated using the same method as in version 2 of the second-order coefficient method, except that it is based on this new metric. The ordering calculated here results is a ROBDD with 8 nodes, which is not optimal, but is 3 less nodes than the achieved by default ordering.

5.3 Summary

The ordering of the variables is very important when representing functions with ROBDDs. Some orderings may cause a ROBDD to have an exponential number of nodes for some functions. To avoid this, it has been hypothesized that the information in the autocorrelation coefficients may be made use of in order to find optimal variable orderings.

For the examples tested in this chapter, in most cases orderings based on the autocorrelation coefficients result in optimal ROBDDs. Where the ROBDDs were not optimal, they were still better than the worst case. In general, it is hypothesized that ordering the variables based on the autocorrelation coefficients is, in most cases, better than average. For certain types of functions, in particular, symmetric functions, no ordering is better than any other. However, symmetric functions can be detected by examining their spectral coefficients [HMM85].

The second half of this chapter describes the autocorrelation coefficient-based ordering methods used in the following chapters. The first method is based on only the first-order autocorrelation coefficients. The second two methods are both based on the second-order autocorrelation coefficients, and the final method orders the variables based on the average of the first and second-order coefficients for each variable.

In the following chapter, a number of practical circuits are tested to see if the above hypothesis holds true in general.

Chapter 6

Experimental Results

The previous chapter explains the algorithms developed to generate variable orderings based on the autocorrelation coefficients. Chapter 6 presents the experiments based on these algorithms and the results achieved.

In the first section, the measures used for the comparison of each ordering method are discussed. These measures are the number of nodes in the ROBDD for the function, the number of combinational logic blocks in the FPGA after the function is mapped to it, and the time required to determine the variable ordering.

There are six methods of ordering that are discussed; four methods based on the autocorrelation coefficients, a randomly generated ordering, and an ordering generated by the package Item. In this chapter, only results generated after running the mapper called xmap are presented. Tables containing values for each of the other mappers in Item are given in Appendix E.

The results presented here demonstrate that orderings based on the autocorrelation coefficients are, on average, as good as, or better than the orderings generated by Item. These results also show that these methods are considerably faster than Item.

Before further discussion, it should be noted that a number of restrictions were applied for these experiments. One restriction is the limitations of the program used for calculating the autocorrelation coefficients. This program limits the number of function inputs to 20 or less. Also, since this is the initial research into this area, it seems logical to begin with simpler functions. Therefore, these experiments use only single-output functions. Table D.1 in Appendix D gives tables of all the functions used, with the number of inputs, number of products, and the source of each function.

6.1 Experimental Procedure

6.1.1 Measures Used

As mentioned above, there are three measures used in this work to determine whether, in general, the ordering methods presented are useful. These measures are:

- the number of nodes in the ROBDD representing the function, when the given ordering is used. This is a logical measure to use, as the ordering methods have been developed in order to reduce this number. The value is calculated by the package called Item, which is discussed in Chapter 3.

- the number of combinational logic blocks in the FPGA implementing the function. This is counted after merging all possible pairs of look-up tables that are used to implement the function. This is a more practical measure than the one above since the target technology in this research is intended to be look-up table-based FPGAs. In this case, Item is used to map the ROBDD representing the function to a FPGA using 5-input 2-output lookup-tables as its combinational logic blocks. The number of CLBs is a commonly used measure in determining the area a FPGA mapping requires.

- the speed of determining the variable ordering. This can be a very important measure for practical uses. In this research the *time* command in UNIX was used to measure the seconds taken generate a variable ordering for the function. All tests are performed on a SparcStation 10.

Another commonly-used measure in this area of research is the unit-delay of the mapped FPGA. In this research, we have not measured this value since our work is aimed solely at reducing the area of the the FPGA that is required.

6.1.2 Other Methods Used for Comparisons

The following five methods are each based on the function's autocorrelation coefficients:

- the first order autocorrelation coefficient method,
- the second order autocorrelation coefficient methods, versions 1 and 2,
- the ordering method based on the average of the first and second autocorrelation coefficients for each pair, and
- the reverse of each of these.

The reversed ordering is generated for each of these because it requires very little time but in some cases, produces quite dramatic effects on the measures.

For comparison purposes the same measures are also determined using the following orderings for each function:

- a randomly-generated ordering, and

- the best ordering that Item can determine.

The randomly-generated ordering is used to represent what results can be achieved if little or no work is done in determining an ordering. Item's best ordering is the goal that we are attempting to reach or surpass using one of the five autocorrelation coefficient-based orderings.

The steps for determining the autocorrelation coefficient-based orderings are discussed in detail in the previous chapter. The reverse orderings are found by using Item to reverse the ordering. The script used for generating Item's best ordering is given in Appendix C. The cost function given for Item to use in generating this ordering is the number of nodes in the ROBDD for the function.

6.1.3 Steps Performed

To find the desired measures for each function using each different ordering method, a number of steps were followed:

1. Produce all files that are needed for determining the variable ordering,
2. determine the variable ordering using the appropriate method,
3. make use of Item to create a ROBDD for the function, then map the function to an XC3000 FPGA, and finally,
4. output the resulting measures from the Item mapping.

These steps are almost identical for all of the ordering methods used. The main differences are that, for the methods of ordering based on the autocorrelation coefficients, the autocorrelation coefficients must be calculated before the variable ordering can be determined.

In order to calculate the autocorrelation coefficients a program called *acspec* was used. This program was made available for this research by Randal Tomczuk [Tomed]. As previously mentioned, this program has a built-in limitation restricting the input-functions to 20 or fewer inputs. Also, the input format required for acspec is the two-level, pla format. Hence, all functions are described in this format.

6.2 Summary of Results

In the following three sections, results from each of the autocorrelation coefficient-based ordering methods are presented. For each method, a table of the ROBDD nodes and CLBs required for a small selection of functions is presented, followed by a summary of the results. The summary of results is given as the average percent improvement for each method over the best values that Item can achieve, and the average percent improvement for each method over the values from the randomly generated orderings. The truncated average is also given. This is the same calculation as the average percent improvement only with the best and worst values discarded. The average percent improvement of method A over method B is calculated as follows:

$$percent\ \ improvement = 100 * \frac{(resultfromB - resultfromA)}{(resultsfromB)}$$

At the end of the chapter the results for 10 randomly generated functions are presented, followed by the results for a small selection of multiple-output functions.

6.2.1 Method 1

The first method of ordering to be discussed is the method based on the first-order autocorrelation coefficients. Table 6.1 shows comparisons between the number of

function	Method 1	Method 1, reversed	Item	randomly generated orderings
	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)
9sym.pla	(26, 8)	(26, 8)	(26, 8)	(26, 8)
add61.pla	(4, 1)	(4, 1)	(4, 1)	(4, 1)
add62.pla	(6, 1)	(6, 1)	(6, 1)	(6, 1)
add63.pla	(9, 2)	(9, 3)	(11, 2)	(13, 3)
add64.pla	(12, 3)	(12, 4)	(20, 3)	(22, 4)
add65.pla	(15, 4)	(15, 6)	(39, 10)	(31, 8)
add66.pla	(18, 4)	(18, 8)	(78, 37)	(91, 44)
add67.pla	(19, 4)	(19, 8)	(93, 34)	(81, 35)
co14.pla	(73, 24)	(71, 24)	(71, 24)	(71, 24)
life.pla	(884, 412)	(857, 418)	(857, 418)	(884, 413)
max46.pla	(83, 27)	(78, 27)	(79, 24)	(83, 27)
parity.pla	(10,2)	(10, 2)	(10, 2)	(10, 2)
ryy6.pla	(31, 10)	(33, 10)	(85, 25)	(122, 49)

Table 6.1: The ROBDD nodes and CLBs required for orderings generated by Method 1 for a selection of single-output functions.

ROBDD nodes resulting from each ordering method, and also comparisons between the number of CLBs required by the mapped function resulting from each ordering method.

Table 6.1 shows results for only 13 out of 34 functions tested. The table shows that in most cases, the randomly generated orderings are not as good as either Item's best ordering, or the orderings generated by Method 1. There are also four functions in this table for which the results are identical for all four ordering methods. The reason for this is that each of these functions is either completely or partially symmetric.

The timing results for this method are given in Table E.2 in Appendix E. They show clearly that for nearly all functions, the time required by this first ordering method is always either almost 0 seconds, or if not, still far smaller than the time required by Item. In fact, the average percent improvement of this method over Item's times is 93.04%.

To have a better understanding of the overall performance of this method, Table 6.2 shows the average percent improvement of Method 1 over Item and over the randomly generated orderings. These results are based on the results from 34 func-

ordering method	improvement over Item		improvement over Random	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
first-order	11.15	9.09	23.54	24.32
truncated avg.	10.20	9.22	22.82	26.12
reverse	-58.11	-92.68	6.75	6.00
truncated avg.	3.23	-4.82	15.83	11.42
	Item's improvement over Random		**speedup over Item**	
	% for bdd nodes	% for CLBs	%	
	11.04	13.83	~90	
truncated avg.	10.62	12.95	~90	

Table 6.2: Summary of results for Method 1 orderings for single-output functions, given as percentage improvement.

ordering method	number better than Item		number worse than Item		number equal to Item	
	bdd nodes	CLBs	bdd nodes	CLBs	bdd nodes	CLBs
Method 1	10	8	10	4	14	22
Method 1, reversed	10	6	13	11	11	17

Table 6.3: Summary of results for Method 1 orderings for single-output functions, given as the number of results better, equal to, and worse than Item's.

tions tested. The average improvement over both Item's results and the randomly generated ordering results is given in the first row, with the average improvement minus the best and worst results given in the following row, labeled **truncated avg.**. Item's average improvement over the randomly generated orderings is also given in the bottom part of the table. The reason for giving the truncated averages is that for some orderings (see Table E.1 in Appendix E for the complete table of results) the average results are overwhelmed by one extremely poor value.

It is interesting to note that, for this ordering method, the reversed ordering performs significantly worse than the original Method 1 ordering, even when the truncated averages are compared. This indicates that for these functions, when ordering based on the first-order autocorrelation coefficients, it is best to have the variables with large coefficients at the *end* of the ordering.

Another comparison is shown in Table 6.3.

Table 6.3 shows the number of results from Method 1 orderings that are better,

function	Method 2a)	Method 2b)	Item	randomly generated orderings
	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)
9sym.pla	(26, 8)	(26, 8)	(26, 8)	(26, 8)
add61.pla	(4, 1)	(4, 1)	(4, 1)	(4, 1)
add62.pla	(6, 1)	(8, 1)	(6, 1)	(6, 1)
add63.pla	(9, 3)	(11, 3)	(11, 2)	(13, 3)
add64.pla	(12, 5)	(12, 4)	(20, 3)	(22, 4)
add65.pla	(15, 7)	(15, 6)	(39, 10)	(31, 8)
add66.pla	(18, 9)	(18, 8)	(78, 37)	(91, 44)
add67.pla	(19, 4)	(19, 8)	(93, 34)	(81, 35)
co14.pla	(73, 24)	(71, 24)	(71, 24)	(71, 24)
life.pla	(857, 418)	(857, 418)	(857, 418)	(884, 413)
max46.pla	(83, 29)	(85, 30)	(79, 24)	(83, 27)
parity.pla	(10, 2)	(10, 2)	(10, 2)	(10, 2)
ryy6.pla	(29, 12)	(29, 9)	(85, 25)	(122, 49)

Table 6.4: The ROBDD nodes and CLBs required for orderings generated by Methods 2a) and 2b) for a selection of single-output functions.

worse, and the same as Item's results for the 34 functions tested. This shows that for the majority of these functions, Method 1 performs about the same as does Item's ordering method. For the final mapped function, Method 1 achieves better results than Item on 8 functions, and is worse than Item on 4 functions. The values in this table also confirm that that the reversed ordering does not perform as well as the original.

6.2.2 Methods 2a) and 2b)

The second two methods of ordering are based on the second-order autocorrelation coefficients. The following tables show a selection of the results obtained when using these methods to obtain variable orderings. Tables 6.4 and 6.5 compare the number of ROBDD nodes and the number of CLBs required by each mapped function for each ordering method.

For the results shown here it appears that both ordering methods 2a) and 2b) perform better than the randomly generated orderings. It is not clear, however, which is the overall best method; Item, Method 2a), Method 2b), or one of 2a) or 2b)

functions	Method 2a) reversed	Method 2b) reversed	Item	randomly generated orderings
	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)	(bdd nodes, CLBs)
9sym.pla	(26, 8)	(26, 8)	(26, 8)	(26, 8)
add61.pla	(4, 1)	(4, 1)	(4, 1)	(4, 1)
add62.pla	(6, 1)	(8, 1)	(6, 1)	(6, 1)
add63.pla	(9, 2)	(11, 3)	(11, 2)	(13, 3)
add64.pla	(12, 3)	(12, 4)	(20, 3)	(22, 4)
add65.pla	(15, 5)	(15, 4)	(39, 10)	(31, 8)
add66.pla	(18, 7)	(18, 4)	(78, 37)	(91, 44)
add67.pla	(19, 8)	(19, 4)	(93, 34)	(81, 35)
co14.pla	(71, 24)	(73, 24)	(71, 24)	(71, 24)
life.pla	(884, 412)	(884, 412)	(857, 418)	(884, 413)
max46.pla	(80, 27)	(84, 29)	(79, 24)	(83, 27)
parity.pla	(10, 2)	(10, 2)	(10, 2)	(10, 2)
ryy6.pla	(29, 10)	(27, 9)	(85, 25)	(122, 49)

Table 6.5: The ROBDD nodes and CLBs required by the orderings obtained by reversing Methods 2a) and 2b).

reversed.

Table 6.6 shows the average and truncated average percent improvement for both methods 2a) and 2b) over Item's results. Item's improvement over the randomly generated orderings is also given in the bottom part of Table 6.6.

For Method 2a), one function (see Table E.3 in Appendix E) results in a very poor ROBDD, and this value overwhelms the resulting averages. This effect is also seen in the average improvement for the reverse of Method 2b).

Both methods a) and b) are very fast as compared to Item. Method 2a) times are 92.94% faster than Item while Method 2b) times are 92.80% faster than Item.

Another comparison is shown in Table 6.7. This table shows the number of results from methods 2a) and b) that are better, worse, and the same as those from Item. Method 2a) results in ROBDD sizes that are worse than those achieved by Item for a majority of the functions, but results in CLB counts that are greatly improved. This trend is actually reversed in Table 6.6. Method 2b), however, performs about the same as Item for a majority of the functions. A trend that can be noticed here for all ordering methods is that, after mapping, the number of functions with the same

ordering method	improvement over Item		improvement over Random	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
Method 2a)	-58.09	-89.13	5.51	8.15
truncated avg.	3.24	-2.61	14.51	13.53
2a) reversed	9.38	8.72	20.78	23.52
truncated avg.	8.93	8.04	20.61	24.24
Method 2b)	11.89	5.16	24.24	20.53
truncated avg.	10.98	4.81	23.56	20.07
2b) reversed	-60.88	-84.82	5.20	9.99
truncated avg.	0.29	-0.47	14.81	15.17
	Item's improvement over Random		**Method 2a)**	**Method 2b)**
			speedup over Item	**speedup over Item**
	% for bdd nodes	% for CLBs	%	%
	11.04	13.83	~90	~90
truncated avg.	10.62	12.95	~90	~90

Table 6.6: Summary of results for Method 2a) and 2b) orderings for single-output functions, given as percentage improvement.

ordering method	**number better than Item**		**number worse than Item**		**number equal to Item**	
	bdd nodes	CLBs	bdd nodes	CLBs	bdd nodes	CLBs
Method 2a	9	6	14	9	11	19
Method 2a, reversed	11	8	9	5	14	21
Method 2b	11	8	9	4	14	22
Method 2b, reversed	10	8	14	9	10	17

Table 6.7: Summary of results for Method 2a) and 2b) orderings for single-output functions, given as the number of results better, equal to, and worse than Item's.

results as those achieved by Item increases as compared to before mapping. This indicates that the mapping algorithm Xmap is often able to find a good mapping, even if the ROBDD size is not optimal.

6.2.3 Method 3

The third method of ordering uses an average of the first and second-order autocorrelation coefficients for each variable pair to determine a variable ordering. Table 6.8 shows a comparison of the results achieved by this method to those achieved by Item and the randomly generated orderings.

Again, it appears that this method of ordering shows a definite improvement over

function	Method 3	Method 3 reversed	Item	randomly generated orderings
	(bdd nodes, CLBS)	(bdd nodes, CLBS)	(bdd nodes, CLBS)	(bdd nodes, CLBS)
9sym.pla	(26, 8)	(26, 8)	(26, 8)	(26, 8)
add61.pla	(4, 1)	(4, 1)	(4, 1)	(4, 1)
add62.pla	(6, 1)	(6, 1)	(6, 1)	(6, 1)
add63.pla	(9, 2)	(9, 3)	(11, 2)	(13, 3)
add64.pla	(12, 3)	(12, 4)	(20,3)	(22, 4)
add65.pla	(22, 4)	(21, 9)	(39, 10)	(31, 8)
add66.pla	(20, 4)	(19, 8)	(78, 37)	(91, 44)
add67.pla	(21, 4)	(20, 8)	(93, 34)	(81, 35)
co14.pla	(71, 24)	(73, 24)	(71, 24)	(71, 24)
life.pla	(884, 412)	(857, 418)	(857, 418)	(884, 413)
max46.pla	(83, 26)	(78, 28)	(79, 24)	(83, 27)
parity.pla	(10, 2)	(10, 2)	(10, 2)	(10, 2)
ryy6.pla	(31, 10)	(33, 10)	(85, 25)	(122, 49)

Table 6.8: The ROBDD nodes and CLBs required for orderings generated by Method 3 for a selection of single-output functions.

ordering method	improvement over Item		improvement over Random	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
$r_i + r_j + r_{ij}$	10.71	8.85	23.00	25.01
truncated avg.	9.79	9.00	22.25	25.81
reverse	-58.63	-90.50	6.25	5.02
truncated avg.	2.72	-6.41	15.33	9.96
	Item's improvement over Random		**speedup over Item**	
	% for bdd nodes	% for CLBs	%	
	11.04	13.83	~90	
truncated avg.	10.62	12.95	~90	

Table 6.9: Summary of results for Method 3 orderings for single-output functions, given as percentage improvement.

the randomly generated orderings. The average percent improvement over both Item and the random orderings is shown in Table 6.9. Also shown in Table 6.9 is Item's average percent improvement over the randomly generated orderings.

Table 6.9 shows how method 3 compares to Item's ordering results and to the randomly generated orderings. For this method also there is one function resulting in a very poor ROBDD with the reverse ordering, and it is clear how this has affected the average performance. Another comparison is shown in Table 6.10. This table shows the number of results from this method that are better, worse, and the same as Item's results.

ordering method	**number better than Item**		**number worse than Item**		**number equal to Item**	
	bdd nodes	CLBs	bdd nodes	CLBs	bdd nodes	CLBs
Method 3	10	8	9	5	15	21
Method 3, reversed	11	6	12	11	11	17

Table 6.10: Summary of results for Method 3 orderings for single-output functions, given as the number of results better, equal to, and worse than Item's.

6.2.4 Discussion of Single Output Results

From the above tables of averages, it seems that for most functions, these ordering methods improve upon the best results that Item achieves, if only slightly. The speed at which these methods generate orderings far outstrips Item. If we compare the autocorrelation coefficient-based orderings to randomly generated orderings, on average there is significant improvement. Of the four autocorrelation-coefficient based orderings, the method most often resulting in the smallest size ROBDD is Method 2 version a), with 11.9% improvement over Item's orderings and 24.24% improvement over the randomly generated orderings.

The method most often resulting in the smallest number of CLBs in the mapped function is Method 3, ordering based on the average of both the first and second order autocorrelation coefficients for each pair. For the number of CLBs, the average percent improvement of this method over Item is 8.85% and the average percent improvement of this method over the random orderings is 25.01%.

Another observation that should be made here is that for the functions *add64.pla* through *add67.pla*, all of the autocorrelation-based ordering methods resulted in significantly smaller ROBDDs and number of CLBs in the mapped functions than the ordering generated by Item. In fact, for these functions, the average overall percent improvement over Item is 36.81% for the number of ROBDD nodes, and 23.70% for the number of CLBs. However, these results indicate not that our methods perform exceedingly well, but instead that for these examples, Item performs exceedingly

function	Method 1	Method 2a)	Method 2b)
cm152a.pla	(17, 5,4)	(384, 146,125)	(17, 5,4)

function	Method 3	Item	random ordering
cm152a.pla	(17, 5,4)	(17, 5,4)	(84, 45,37)

Table 6.11: Results for the function cm152a.pla.

poorly. This can be determined by comparing the results from Item's ordering to the results from the randomly generated orderings.

While the autocorrelation coefficient-based orderings appear to perform well for single-output functions such as *add7.pla*, their performance is more unpredictable for functions such as *cm152a.pla*. The results for this function using each of the 4 methods are shown in Table 6.11.

The function *cm152a.pla* is shown below in sum-of-products form:

$$
\begin{aligned}
f \;=\; & v0\ \overline{v8}\ \overline{v9}\ \overline{v10} + v1\ v8\ \overline{v9}\ \overline{v10} + \\
& v2\ \overline{v8}\ v9\ \overline{v10} + v3\ v8\ v9\ \overline{v10} + \\
& v4\ \overline{v8}\ \overline{v9}\ v10 + v5\ v8\ \overline{v9}\ v10 + \\
& v6\ \overline{v8}\ v9\ v10 + v7\ v8\ v9\ v10
\end{aligned}
$$

6.2.5 Some Results on Randomly Generated Functions

In this section, the results from testing each of the ordering methods from above on a number of randomly generated functions are presented. Each of the functions used has 10 inputs and 1 output, and were generated using the random number generator available with the C Library functions (see man 3 random), seeded with the value 1.

For the randomly generated functions, none of the autocorrelation coefficient-based orderings result in any appreciable improvement, if any at all. There is also little improvement over the randomly generated orderings for these functions. This seems to indicate that ROBDDs may not be the best representation to use when

function	Method 1	Item	random orderings
	(bdd nodes, clbs)	(bdd nodes, clbs)	(bdd nodes, clbs)
1.pla	(197, 56)	(193, 69)	(202, 68)
10.pla	(109, 42)	(105, 41)	(109, 41)
2.pla	(180, 61)	(181, 70)	(187, 67)
3.pla	(131, 47)	(129, 49)	(133, 54)
4.pla	(120, 44)	(120, 44)	(124, 48)
5.pla	(201, 67)	(199, 65)	(208, 70)
6.pla	(210, 67)	(200, 67)	(202, 68)
7.pla	(104, 39)	(107, 42)	(109, 41)
8.pla	(186, 66)	(181, 61)	(191, 65)
9.pla	(191, 65)	(180, 62)	(186, 69)
Method 1's % improvement over Item		**Method 1's % improvement over the random orderings**	
bdd nodes	clbs	bdd nodes	clbs
-1.90	2.44	1.49	6.04

Table 6.12: The ROBDD nodes and CLBs required for orderings generated by Method 1 for 10 randomly generated functions.

function	Method 2a)	Method 2b)	Item	random orderings
	(bdd nodes, clbs)	(bdd nodes, clbs)	(bdd nodes, clbs)	(bdd nodes, clbs)
1.pla	(200, 73)	(196, 74)	(193, 69)	(202, 68)
10.pla	(107, 39)	(106, 37)	(105, 41)	(109, 41)
2.pla	(184, 64)	(185, 62)	(181, 70)	(187, 67)
3.pla	(135, 51)	(137, 53)	(129, 49)	(133, 54)
4.pla	(117, 42)	(125, 45)	(120, 44)	(124, 48)
5.pla	(203, 68)	(201, 69)	(199, 65)	(208, 70)
6.pla	(207, 63)	(202, 73)	(200, 67)	(202, 68)
7.pla	(105, 42)	(110, 38)	(107, 42)	(109, 41)
8.pla	(191, 65)	(180, 68)	(181, 61)	(191, 65)
9.pla	(189, 66)	(179, 66)	(180, 62)	(186, 69)
ordering method	**% improvement over Item**		**% improvement over Random**	
	bdd nodes	clbs	bdd nodes	clbs
Method 2a)	-2.35	-0.35	1.06	3.21
Method 2b)	-1.88	-2.00	1.49	1.76

Table 6.13: The ROBDD nodes and CLBs required for orderings generated by Methods 2a) and 2b) for 10 randomly generated functions.

function	Method 3	Item	random orderings
	(bdd nodes, clbs)	(bdd nodes, clbs)	(bdd nodes, clbs)
1.pla	(193, 63)	(193, 69)	(202, 68)
10.pla	(103, 39)	(105, 41)	(109, 41)
2.pla	(184, 63)	(181, 70)	(187, 67)
3.pla	(127, 46)	(129, 49)	(133, 54)
4.pla	(120, 42)	(120, 44)	(124, 48)
5.pla	(208, 63)	(199, 65)	(208, 70)
6.pla	(205, 72)	(200, 67)	(202, 68)
7.pla	(105, 38)	(107, 42)	(109, 41)
8.pla	(185, 57)	(181, 61)	(191, 65)
9.pla	(191, 67)	(180, 62)	(186, 69)
Method 3's % improvement over Item		**Method 3's % improvement over the random orderings**	
bdd nodes	clbs	bdd nodes	clbs
-1.17	3.79	2.19	7.22

Table 6.14: The ROBDD nodes and CLBs required for orderings generated by Method 3 for 10 randomly generated functions.

ordering method	% improvement over Item		% improvement over Random	
	bdd nodes	clbs	bdd nodes	clbs
Method 1	-20.01	-21.44	16.16	15.96
Method 2a)	-41.08	-53.21	4.97	1.00
Method 2b)	-12.37	-13.87	20.98	20.91
Method 3	-19.50	-17.56	18.96	22.77

Table 6.15: Summary of results for 22 multiple-output functions.

representing random functions, as there is little similarity present in the different parts of the function, and thus less sharing of subtrees can take place.

6.2.6 Some Results on Multiple Output Functions

In this section, the results from testing each of the ordering methods from above on a number of multiple-output functions are presented. Details on each of the functions used for these tests are given in Appendix D.

The results in Table 6.15 indicate that for this small selection of multiple-output functions, Item does very well in determining orderings for the smallest possible ROBDDs; in fact, better than any of the autocorrelation coefficient-based orderings could do. However the coefficient-based orderings still perform far better than the randomly

ordering: $x_4 x_2 x_1 x_3$		ordering: $x_2 x_1 x_3 x_4$	
cellcount (CLBs)	gatecount (LUTs)	cellcount (CLBs)	gatecount (LUTs)
4	4	8	8

Table 6.16: The differing number of CLBs and look-up tables needed for two variables orderings resulting in the same size ROBDD for the function $f = \overline{x_2}\,\overline{x_4} + \overline{x_1} x_3 \overline{x_4} + x_2 \overline{x_3} x_4 + x_1 x_2 x_4$.

generated orderings, and are much faster than Item.

6.3 Summary

All results given in this chapter are based on the mapping algorithm called xmap. Results using the other mapping algorithms available in Item are given in Appendix E.

It is very important to know how the technology mapper works. This is demonstrated by the following example.

In Table 6.16 it appears that different orderings for this particular example result in double the number of CLBs required to map this function.

The ROBDDs, as shown in Figure 6.1, have the same number of nodes.

Thus it appears that there is some other characteristic of the ROBDD other than the number of nodes that is affecting the mapping results. However, these results were generated using the mapping algorithm xmap, with a look-up table width of two. With look-up table widths of two or less, the underlying structure is changed, and so the number of nodes in the ROBDD is no longer the same.

In this research we assume that, with the above exception, none of Item's mappers modify the underlying structure of the function representation. This assumption is taken from [Kar90]. Based on this, one of our hypotheses is that building a small ROBDD for the function results in a fewer number of CLBs required in the final

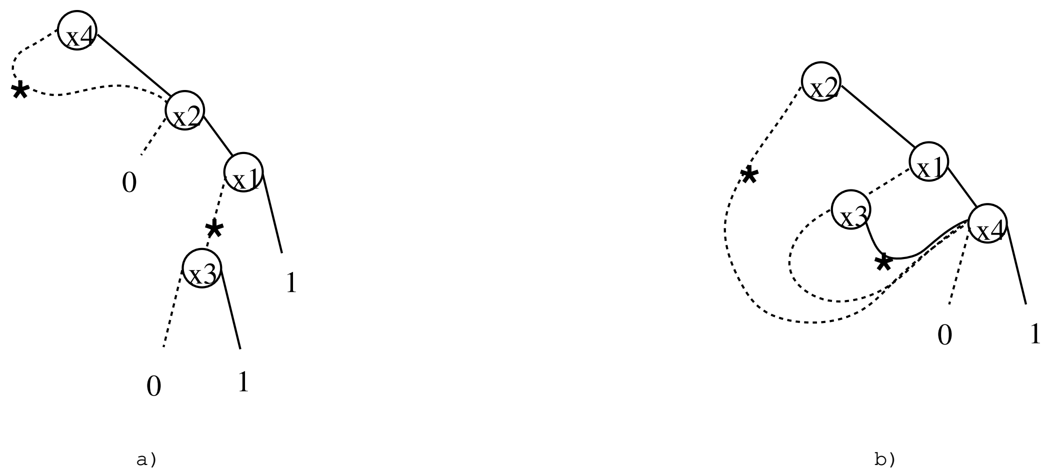


Figure 6.1: Two ROBDDs for the function $f = \overline{x_2}\,\overline{x_4} + \overline{x_1}x_3\overline{x_4} + x_2\overline{x_3}x_4 + x_1x_2x_4$.

FPGA mapping. Based on the results in this chapter, this hypothesis does seem to be valid. Our other hypothesis is that using the autocorrelation coefficients to order the variables for the ROBDD will result in fewer nodes in the ROBDD. The results presented in this chapter confirm this hypothesis for some cases, but not all.

For practical use, the results in this chapter indicate that each of the autocorrelation coefficient-based ordering methods developed are effective for certain functions, and that they are all very fast. Method 3 has the best overall performance, but generated an ordering resulting in an exponential size ROBDD for one example. Therefore, the most practical way to make use of these methods would be to initially use Method 3 to order the variables, but also generate the reverse ordering, and use the best of the two.

Chapter 7

Conclusions

7.1 Overview

In Chapter 2 we present a new technology called FPGAs. These are becoming very popular because of their relatively low cost and the speed with which they can be programmed. To implement a function with a FPGA, the function must be described in some standard format, which can then be mapped to the FPGA.

In Chapter 3 we suggest using a ROBDD to represent the function. This representation simplifies many steps in the technology mapping process. However, the ordering of the function's variables can have a large effect on the size of the ROBDD.

Chapter 4 introduces autocorrelation coefficients. The autocorrelation coefficients of a function provide information about the function in a different format. We hypothesize that the information given by a function's autocorrelation coefficients can be used to generate a variable ordering that will result in a ROBDD of optimal size.

In Chapters 5 and 6 we examine this hypothesis in more detail. Chapter 5 examines a small number of in-depth examples for which the information in the autocorrelation coefficients corresponds to ROBDDs with fewer nodes. Chapter 6 tests our ordering-generation methods on a number of single-output functions.

7.2 Summary of Results

The four autocorrelation coefficient-based orderings all produce significantly smaller ROBDDs than the randomly generated orderings for nearly all single-output functions tested. The same is true for the number of CLBs.

When compared to the ordering generated by Item, our methods do not perform as well. For the single-output functions tested, Method 1, the reverse of Method 2a), Method 2b) and Method 3 show from 5.2% to 9.2% improvement over Item for CLBs, and from 8.9% to 11.9% improvement over Item for the ROBDD sizes. The other methods result in negative improvements over the orderings generated by Item.

All of the autocorrelation coefficient-based ordering methods perform on average at least 90% faster than Item.

In Chapter 6 we also compare our methods to Item's orderings using 10 randomly generated functions and a number of multiple-output functions. For the randomly generated functions, our methods do not achieve any significant improvement over Item's ordering method, and only slight improvements over the randomly generated orderings. For the multiple-output functions, the autocorrelation coefficient-based orderings perform very poorly in comparison to Item's orderings, but still show fair improvements over the randomly generated orderings.

For practical use we recommend that the designer should have some knowledge of the function to be mapped before selecting an ordering method. If a function is symmetrical, no ordering changes will result in any increase or decrease in the size of the function's ROBDD. For a single-output function, the best ordering method to use is Method 3. Since it is a very fast ordering method, generating the reverse of the ordering and comparing the two is recommended in order to achieve better results. For multiple-output functions, the small number of tests in this work indicate that Item's ordering method is the recommended method for achieving the best results.

7.3 Future Work

There are many aspects of this work into which further research could be done. One of the restrictions of this research is that the main experiments used only single-output functions. An immediate area for future work is to develop ordering methods that perform equally well for multiple-output functions. Further, no investigation is done into the usefulness of the larger order autocorrelation coefficients or the $1, -1$ encoded autocorrelation coefficients. Either or both of these could be of use in developing autocorrelation coefficient-based ordering methods for multiple-output functions.

A further area of research is in improving not only the area required by the mapped function, but also the time delays and the routability of the function. Ordering a function's variables to reduce the number of edges in the ROBDD may be an idea to consider for this.

7.4 Concluding Remarks

The results in this work lead us to conclude that the autocorrelation coefficient-based ordering methods developed here do have some usefulness when used to generate orderings for single-output functions that are to be mapped to LUT-based FPGAs.

However, these methods do not perform as well in our tests on multiple-output functions. Future work in this area may produce promising results.

Appendix A

Glossary

A.1 Glossary

A.1.1 Acronyms

ASIC Application Specific Integrated Circuit

BDD Binary Decision Diagram

DAG Directed Acyclic Graph

FPGA Field Programmable Gate Array

LUT Look-up Table

NP Non-deterministic Polynomial

PAL Programmable Array Logic

PLA Programmable Logic Array

PLD Programmable Logic Device

PML Programmable Macro Logic

POS Product of Sums

ROBDD Reduced Ordered Binary Decision Diagram

SOP Sum of Products

A.1.2 Logic Definitions

cover - a cover F of an incompletely specified function (*i.e.* a function with a don't care set) is a completely specified function f such that all minterms in the on-set of f are also in F and no minterm in F is in the off-set of f

cube - product

cube-free - an expression that does not contain a cube of the function in question

don't care set - the set of all minterms dc of a Boolean function f for which $f(dc)$ is not defined

factored forms - a Boolean function representation consisting of arbitrary parenthesized expressions with the single constraint that complementation be applied only to single variables

implicant - a cube or product q of a Boolean function f that is contained in the union of the on-set of f and the don't care set of f and that is not in the off-set of f

Karnaugh map - a tabular method of representing Boolean functions that gives all possible input combinations and the resulting function outputs, but which allows grouping of the outputs so that minterms can be found

kernel - k is a kernel of an algebraic expression f if $k = \frac{f}{c}$ where c is a cube of f and k is cube-free

literal - a Boolean variable x when it has the value "1" or the value "0"

LUT (Look-up Table) - a memory table that can look up any of 2^{2n} possible functions, where n is the number of inputs

minterm - a product of a function in which no variable appears more than once

multi-level representations - representations of Boolean functions that use arbitrary nesting of operators through the use of parentheses; factored forms are an example of multi-level representations

off-set - the set of all minterms u of a Boolean function f that satisfy $f(u) = 0$

on-set - the set of all minterms v of a Boolean function f that satisfy $f(v) = 1$

POS (product of sums) - a Boolean function representation in which a number of expressions are combined with the "and" operator. Each combined expression consists of a number of literals that are combined with the "or" operator

SOP (sum of products) - a Boolean function representation in which a number of expressions are combined with the "or" operator. Each combined expression consists of a number of literals that are combined with the "and" operator

product - one of the sub-expressions in a SOP function representation that consists of literals combined with the "and" operator

truth table - a tabular method of representing Boolean functions in which all possible input combinations and the resulting function output(s) are listed

two-level representations - representations of Boolean functions that use only two levels of operators; the sum of products representation is one example of a two-level representation

Appendix B

Item Scripts

```
# script based on example script included with
# ITEM, modified by me (Jackie Crow) December 7/93

INITIAL
print "Script started " date() "\n"
print "Comparing mappers: (lookup tables, CLBs, unitdelay)\n"
print "Initial network: (gates, unitdelay)\n"
print "bonuses read: \n "
read -bonuses /package/Item/src/Mapping/Tmap-ad.params
print bonuses() "\n\n"
print Filename column(20) Initial
print column(36) Xmap column(52) Xtmap column(68) Xcmap
print column(84) cputime
print "\n"
set lutwidth 5
ALWAYS
```

```
set cputime 0
read -blif
print filename()
print column(20) "(" gatecount() ", " unitdelay() ")"
map xmap
print column(36) "(" gatecount() ", " cellcount() ", " unitdelay() ")"
# write -blif outxmap.blif
# write -pla outxmap.pla
map xtmap -p -1
# -p -1 means try all the bonus sets that were read in to get the best mapping
print column(52) "(" gatecount() ", " cellcount() ", " unitdelay() ")"
# write -blif outxtmap.blif
# write -pla outxtmap.pla
map xcmap
print column(68) "(" gatecount() ","  cellcount() ","  unitdelay() ")"
print column(84) cputime()
# write -blif outxcmap.blif
# write -pla outxcmap.pla
print "\n"
FINAL
#  write the bonuses out again, if any tuning was done (-r parameter > 0)
# write -bonuses /a/projects/logic-min/lib/item/Tmap-ad.params
quit

# script based on example script included with
# ITEM, modified by me (Jackie Crow) December 7/93
```

```
INITIAL
print "Script started " date() "\n"
print "Comparing Xmap, fanout, and internal: (lookup tables, CLBs, unitdelay)\n"
print "Initial network: (gates, unitdelay)\n"
# print "bonuses read: \n "
# read -bonuses /package/Item/src/Mapping/Tmap-ad.params
# print bonuses() "\n"
print Filename column(20) Initial
print column(36) Xmap column(52) fanout column(68) internal
print column(84) cputime
print "\n"
set lutwidth 5
ALWAYS
set cputime 0
read -blif
print filename()
print column(20) "(" gatecount() ", " unitdelay() ")"
map xmap
print column(36) "(" gatecount() ", " cellcount() ", " unitdelay() ")"
# write -blif outxmap.blif
# write -pla outxmap.pla
map fanout
print column(52) "(" gatecount() ", " cellcount() ", " unitdelay() ")"
# write -blif outfanout.blif
# write -pla outfanout.pla
map internal
```

```
print column(68) "(" gatecount() ","  cellcount() ","  unitdelay() ")"
print column(84) cputime()
# write -blif outinternal.blif
# write -pla outinternal.pla
print "\n"
FINAL
#  write the bonuses out again, if any tuning was done (-r parameter > 0)
# write -bonuses /a/projects/logic-min/lib/item/Tmap-ad.params
quit
```

Appendix C

Scripts for Calculating Results

The following is the script used for iterating through each input file and calling the appropriate programs. This script is the one using the second version of the ordering method requiring the second order autocorrelation coefficients.

```
#!/bin/csh

# myorder.script file

set output = "`ls -1 *.pla`"
@ count = 0
while ($count < $#output)
@   count ++
    set file = ($output[$count])
    echo ""
    echo "file:" $file "number:" $count

#  here we read the num of inputs...
```

```
   set infile = "`cat $file | grep .i`"
   set line1 = ($infile[1])
   set inputs = ($line1[2])
   if ($inputs > 20) then
echo $file has $inputs inputs ;
echo skipping $file ;
# cleanup
\rm $file* ;
continue
   endif
# in other words, don't bother if the file is too big

   echo running acspec on $file
   /usr/bin/time acspec -o2 $file > $file.acspec
   # gets the ac coeffs for the next step...first check if it worked or not
   set infile = "`cat $file.acspec`"
   set line1 = ($infile[1])
   set firstch = ($line1[1])
   if ($firstch =~ %*) then
echo acspec encountered invalid function definition ;
echo cannot calculate ordering ;
echo skipping $file ;
# cleanup
\rm $file* ;
continue
    endif
    # in other words, if acspec results in an error message, skip the file
```

```
   echo running myorder on $file
   /usr/bin/time myorder 2 $file.acspec
    # uses the ac coeffs to get a new ordering for the circuit,
    # saves it in $file.order, so we must move it to a place that item can find
   \cp $file.ord* my.order
   \rm $file.acspec $file.ord*

    echo running Item on $file
    /usr/bin/time /package/Item/bin/Item -f itemmap.script $file > MYORDER.RES/$file.results

    echo "" >> MYORDER.RES/$file.results
    echo "calculated ordering: " >> MYORDER.RES/$file.results
    cat my.order >> MYORDER.RES/$file.results

    \rm $file  *.strip* *.temp*
end
```

The following script is the script used by Item to specify what commands are to be done.

```
# itemmap.script
# this script will run the Item mapping algs (all of them!!)
# using the ordering in the file called my.order

# written Jan. 17 1995
# modified May 2 1995
# j. crow
# intended for use in mapping a file according to the ordering
# calculated using acspec.  notice that other than transforming the
# circuit into a ROBDD and reordering the vars, NO OTHER MIMIZATION
# IS DONE!

INITIAL
    print "Script started on " date() "\n"
    print "Reporting (siscount, lookup tables, CLBs) for different ordering methods\n"
    print "and different mapping methods. "
    print "Max fanin for all three methods is 5, and \n"
    print "fanout >= 3 is considered a high-fanout node for all methods\n"
    print "except fanout -h 2 and internal \n\n"
    print column(0) filename()
    print column(25) "my ordering\n\n"

ALWAYS
```

```
print column(0) "xmap"

read -pla
read -varorder my.order
obdd
map xmap
print column(25) "("siscount() ", " gatecount() " ," cellcount() ")"

print "\n"

print column(0) "xtmap"

read -pla
read -varorder my.order
obdd
map xtmap
print column(25) "(" siscount() ", " gatecount() ", " cellcount() ")"

print "\n"

print column(0) "xcmap"

read -pla
read -varorder my.order
obdd
map xcmap
print column(25) "(" siscount() ", " gatecount() ", " cellcount() ")"
```

```
print "\n"

print column(0) "fanout -h 3"

read -pla
read -varorder my.order
obdd
map fanout -h 3
print column(25) "(" siscount() ", " gatecount() ", " cellcount() ")"

print "\n"

print column(0) "fanout -h 2"

read -pla
read -varorder my.order
obdd
map fanout -h 2
print column(25) "(" siscount() ", " gatecount() ", " cellcount() ")"

print "\n"

print column(0) "internal (fanout -h 1)"

read -pla
read -varorder my.order
```

```
    obdd

    map internal

    print column(25) "(" siscount() ", " gatecount() ", " cellcount() ")"


FINAL

    quit
```

The following script is the script used by Item to find a variable ordering that results in a small number of nodes in the function's ROBDD.

```
# item.script
# written Jan. 17 1995
# modified May 2 1995
# modified May 5 1995
# j. crow
# only the ordering is done here, so that I can compare the
# time it takes to find a good ordering.

INITIAL

ALWAYS

    read -pla
    order depthbest -c siscount -b
    print varorder()
    print "\n"

FINAL
    quit
```

Appendix D

Test Files

file name	number of inputs	number of products	type of function
9sym.pla	9	87	counts ones
add6	12	1092	adds two 6 bit numbers*
adr4	8	256	4-bit adder*
b9	16	123	*
cm152a.blif	11	8	transformed to pla file
co14.pla	14	47	4-color problem**
ex1.pla	5	16	the xor function
ex2.pla	5	7	modified xor function
ex3.pla	5	4	sparse function
ex4.pla	5	8	non-sparse function
life	9	512	
log8mod	8	47	*
max46	9	46	
parity.pla	8	128	output is one if odd number of input ones
ryy6	16	112	
sym10	10	837	

Table D.1: List of the files used in the main tests for this research.

Files marked with * are files that are multiple output files, but were broken down into one file for each output for this research. The output in question is indicate by a number appended to the end of the file name, *e.g.*, the first output of add6 becomes add61.pla.

** indicates that more information is found in /package/sis-1.1/examples/doc.

file name	**location**
9sym.pla	/package/sis-1.1/examples/comb/mcnc91/tlex
add6	/package/sis-1.1/examples/math
adr4	/package/sis-1.1/examples/math
b9	/package/sis-1.1/examples/indust
cm152a.blif	/package/sis-1.1/examples/comb/mcnc91.mlex
co14.pla	/package/sis-1.1/examples/math
ex1.pla	xor-function
ex2.pla	modified xor-function
ex3.pla	sparse function
ex4.pla	non-sparse function
life	/package/sis-1.1/examples/math
log8mod	/package/sis-1.1/examples/math
max46	/package/sis-1.1/examples/indust
parity.blif	/package/sis-1.1/examples/comb/mcnc91/mlex (modified)
ryy6	/package/sis-1.1/examples/indust
sym10	/package/sis-1.1/examples/math

Table D.2: Locations of the files used in the main tests for this research.

Appendix E

Tables of Results

Results using the Item technology mapping method xmap

The xmap algorithm works in three passes. On the first pass, a traversal of the internal function representation is done, with each node being marked as the output of a logic block if necessary and a list of the inputs to each logic block is kept. On the second pass, polarites are chosen (to take care of any inverted edges) for each marked node and the function of its gate is determined. The third pass attempts to share logic blocks where ever possible [Kar90].

function	Method 1	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8,8)	(26, 8,8)	(26, 8,8)	(26, 8,8)
add61.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
add62.pla	(6, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
add63.pla	(9, 2,2)	(9, 3,3)	(11, 2,2)	(13, 3,3)
add64.pla	(12, 3,3)	(12, 5,4)	(20, 3,3)	(22, 6,4)
add65.pla	(15, 4,4)	(15, 7,6)	(39, 11,10)	(31, 11,8)
add66.pla	(18, 4,4)	(18, 9,8)	(78, 41,37)	(91, 50,44)
add67.pla	(19, 4,4)	(19, 9,8)	(93, 36,34)	(81, 40,35)
adr41.pla	(13, 3,3)	(13, 5,4)	(32, 14,11)	(29, 13,12)
adr42.pla	(12, 3,3)	(12, 5,4)	(25, 10,9)	(22, 7,7)
adr43.pla	(9, 2,2)	(9, 3,3)	(13, 3,3)	(9, 3,3)
adr44.pla	(6, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
adr45.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
b91.pla	(19, 6,6)	(25, 6,6)	(15, 5,4)	(18, 3,3)
b92.pla	(22, 7,7)	(28, 8,8)	(21, 6,4)	(70, 29,23)
b93.pla	(27, 8,8)	(33, 11,11)	(27, 9,8)	(56, 25,23)
b94.pla	(30, 10,10)	(36, 12,12)	(32, 10,8)	(114, 35,33)
b95.pla	(19, 6,6)	(19, 8,7)	(15, 6,6)	(52, 19,18)
cm152a.pla	(17, 5,4)	(384, 154,127)	(17, 5,4)	(84, 45,37)
co14.pla	(73, 42,24)	(71, 42,24)	(71, 42,24)	(71, 42,24)
ex1.pla	(7, 1,1)	(7, 1,1)	(7, 1,1)	(7, 1,1)
ex2.pla	(12, 1,1)	(12, 1,1)	(11, 1,1)	(13, 1,1)
ex3.pla	(12, 1,1)	(12, 1,1)	(12, 1,1)	(11, 1,1)
ex4.pla	(12, 1,1)	(12, 1,1)	(11, 1,1)	(12, 1,1)
life.pla	(884, 412,412)	(857, 428,418)	(857, 428,418)	(884, 413,413)
log8mod1.pla	(11, 2,2)	(11, 3,3)	(11, 2,2)	(10, 2,2)
log8mod2.pla	(15, 3,3)	(18, 3,3)	(15, 3,3)	(17, 3,3)
log8mod3.pla	(17, 3,3)	(21, 3,3)	(17, 3,3)	(17, 3,3)
log8mod4.pla	(24, 4,4)	(32, 6,6)	(22, 4,4)	(26, 4,4)
log8mod5.pla	(27, 4,4)	(31, 4,4)	(24, 4,4)	(28, 6,6)
max46.pla	(83, 31,27)	(78, 33,27)	(79, 27,24)	(83, 35,27)
parity.pla	(10, 2,2)	(10, 2,2)	(10, 2,2)	(10, 2,2)
ryy6.pla	(31, 10,10)	(33, 11,10)	(85, 30,25)	(122, 59,49)
sym10.pla	(32, 12,12)	(32, 12,12)	(32, 12,12)	(32, 12,12)

Table E.1: All results for the 1st-order coefficient-based method using xmap as the mapper.

function	Method 1	Item	random orderings
9sym.pla	0.0	38.8	0.0
add61.pla	0.0	0.0	0.1
add62.pla	0.0	0.0	0.1
add63.pla	0.0	0.2	0.1
add64.pla	0.0	1.4	0.1
add65.pla	0.0	7.4	0.1
add66.pla	0.0	49.4	0.1
add67.pla	0.0	33.2	0.1
adr41.pla	0.0	7.6	0.0
adr42.pla	0.0	7.9	0.0
adr43.pla	0.0	7.9	0.0
adr44.pla	0.0	7.3	0.0
adr45.pla	0.0	7.7	0.0
b91.pla	1.6	27.0	0.0
b92.pla	1.8	29.3	0.0
b93.pla	1.6	31.6	0.0
b94.pla	1.6	50.8	0.0
b95.pla	1.6	22.9	0.0
cm152a.pla	0.0	1.2	0.0
co14.pla	0.9	11.2	0.0
ex1.pla	0.0	2.2	0.0
ex2.pla	0.0	1.1	0.0
ex3.pla	0.0	0.8	0.0
ex4.pla	0.0	1.0	0.0
life.pla	0.0	96.0	0.0
log8mod1.pla	0.0	1.1	0.0
log8mod2.pla	0.0	1.2	0.0
log8mod3.pla	0.0	1.2	0.0
log8mod4.pla	0.0	1.1	0.0
log8mod5.pla	0.0	1.2	0.0
max46.pla	0.0	17.9	0.0
parity.pla	0.0	36.7	0.0
ryy6.pla	1.5	103.4	0.0
sym10.pla	0.1	742.6	0.0

Table E.2: All timing results for the 1st-order coefficient-based method using xmap as the mapper.

function	Method 2a)	Method 2b)	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8,8)	(26, 8,8)	(26, 8,8)	(26, 8,8)
add61.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
add62.pla	(8, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
add63.pla	(11, 3,3)	(9, 4,3)	(11, 2,2)	(13, 3,3)
add64.pla	(12, 5,4)	(12, 6,5)	(20, 3,3)	(22, 6,4)
add65.pla	(15, 7,6)	(15, 8,7)	(39, 11,10)	(31, 11,8)
add66.pla	(18, 9,8)	(18, 10,9)	(78, 41,37)	(91, 50,44)
add67.pla	(19, 9,8)	(19, 4,4)	(93, 36,34)	(81, 40,35)
adr41.pla	(13, 5,4)	(13, 3,3)	(32, 14,11)	(29, 13,12)
adr42.pla	(14, 5,4)	(12, 6,5)	(25, 10,9)	(22, 7,7)
adr43.pla	(11, 3,3)	(9, 4,3)	(13, 3,3)	(9, 3,3)
adr44.pla	(8, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
adr45.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
b91.pla	(20, 4,4)	(16, 5,4)	(15, 5,4)	(18, 3,3)
b92.pla	(28, 8,8)	(24, 6,6)	(21, 6,4)	(70, 29,23)
b93.pla	(31, 10,10)	(27, 7,7)	(27, 9,8)	(56, 25,23)
b94.pla	(34, 12,12)	(28, 9,9)	(32, 10,8)	(114, 35,33)
b95.pla	(15, 6,6)	(19, 6,6)	(15, 6,6)	(52, 19,18)
cm152a.pla	(384, 146,125)	(17, 5,4)	(17, 5,4)	(84, 45,37)
co14.pla	(71, 42,24)	(73, 42,24)	(71, 42,24)	(71, 42,24)
ex1.pla	(7, 1,1)	(7, 1,1)	(7, 1,1)	(7, 1,1)
ex2.pla	(12, 1,1)	(11, 1,1)	(11, 1,1)	(13, 1,1)
ex3.pla	(11, 1,1)	(11, 1,1)	(12, 1,1)	(11, 1,1)
ex4.pla	(12, 1,1)	(12, 1,1)	(11, 1,1)	(12, 1,1)
life.pla	(857, 428,418)	(857, 428,418)	(857, 428,418)	(884, 413,413)
log8mod1.pla	(11, 3,3)	(11, 2,2)	(11, 2,2)	(10, 2,2)
log8mod2.pla	(18, 3,3)	(15, 3,3)	(15, 3,3)	(17, 3,3)
log8mod3.pla	(21, 3,3)	(18, 3,3)	(17, 3,3)	(17, 3,3)
log8mod4.pla	(27, 6,6)	(25, 4,4)	(22, 4,4)	(26, 4,4)
log8mod5.pla	(29, 4,4)	(28, 4,4)	(24, 4,4)	(28, 6,6)
max46.pla	(85, 36,30)	(83, 36,29)	(79, 27,24)	(83, 35,27)
parity.pla	(10, 2,2)	(10, 2,2)	(10, 2,2)	(10, 2,2)
ryy6.pla	(29, 10,9)	(29, 12,12)	(85, 30,25)	(122, 59,49)
sym10.pla	(32, 12,12)	(32, 12,12)	(32, 12,12)	(32, 12,12)

Table E.3: All results for the 2nd-order coefficient-based methods using xmap as the mapper.

function	Method 2a), reversed	Method 2b), reversed	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8,8)	(26, 8,8)	(26, 8,8)	(26, 8,8)
add61.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
add62.pla	(8, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
add63.pla	(11, 3,3)	(9, 2,2)	(11, 2,2)	(13, 3,3)
add64.pla	(12, 3,3)	(12, 3,3)	(20, 3,3)	(22, 6,4)
add65.pla	(15, 4,4)	(15, 6,5)	(39, 11,10)	(31, 11,8)
add66.pla	(18, 4,4)	(18, 8,7)	(78, 41,37)	(91, 50,44)
add67.pla	(19, 4,4)	(19, 9,8)	(93, 36,34)	(81, 40,35)
adr41.pla	(13, 3,3)	(13, 5,4)	(32, 14,11)	(29, 13,12)
adr42.pla	(14, 3,3)	(12, 3,3)	(25, 10,9)	(22, 7,7)
adr43.pla	(11, 3,3)	(9, 2,2)	(13, 3,3)	(9, 3,3)
adr44.pla	(8, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
adr45.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
b91.pla	(22, 5,5)	(32, 6,6)	(15, 5,4)	(18, 3,3)
b92.pla	(22, 6,6)	(36, 9,8)	(21, 6,4)	(70, 29,23)
b93.pla	(25, 7,7)	(39, 11,11)	(27, 9,8)	(56, 25,23)
b94.pla	(28, 9,9)	(36, 12,12)	(32, 10,8)	(114, 35,33)
b95.pla	(15, 6,6)	(19, 8,7)	(15, 6,6)	(52, 19,18)
cm152a.pla	(17, 5,4)	(384, 148,122)	(17, 5,4)	(84, 45,37)
co14.pla	(73, 42,24)	(71, 42,24)	(71, 42,24)	(71, 42,24)
ex1.pla	(7, 1,1)	(7, 1,1)	(7, 1,1)	(7, 1,1)
ex2.pla	(11, 1,1)	(12, 1,1)	(11, 1,1)	(13, 1,1)
ex3.pla	(11, 1,1)	(12, 1,1)	(12, 1,1)	(11, 1,1)
ex4.pla	(13, 1,1)	(10, 1,1)	(11, 1,1)	(12, 1,1)
life.pla	(884, 412,412)	(884, 412,412)	(857, 428,418)	(884, 413,413)
log8mod1.pla	(11, 2,2)	(12, 3,3)	(11, 2,2)	(10, 2,2)
log8mod2.pla	(15, 3,3)	(18, 3,3)	(15, 3,3)	(17, 3,3)
log8mod3.pla	(18, 3,3)	(21, 3,3)	(17, 3,3)	(17, 3,3)
log8mod4.pla	(22, 4,4)	(31, 6,6)	(22, 4,4)	(26, 4,4)
log8mod5.pla	(24, 4,4)	(31, 4,4)	(24, 4,4)	(28, 6,6)
max46.pla	(84, 40,29)	(80, 33,27)	(79, 27,24)	(83, 35,27)
parity.pla	(10, 2,2)	(10, 2,2)	(10, 2,2)	(10, 2,2)
ryy6.pla	(27, 9,9)	(29, 11,10)	(85, 30,25)	(122, 59,49)
sym10.pla	(32, 12,12)	(32, 12,12)	(32, 12,12)	(32, 12,12)

Table E.4: All results for the 2nd-order coefficient-based methods (reversed) using xmap as the mapper.

function	Method 2b)	Method 2a)	Item	random orderings
9sym.pla	0	0	38.8	0.1
add61.pla	0.0	0.0	0.0	0.1
add62.pla	0.0	0.0	0.0	0.1
add63.pla	0.0	0.0	0.2	0.1
add64.pla	0.0	0.0	1.4	0.1
add65.pla	0.0	0.0	7.4	0.1
add66.pla	0.0	0.0	49.4	0.1
add67.pla	0.0	0.0	33.2	0.1
adr41.pla	0	0	7.6	0.1
adr42.pla	0	0	7.9	0.1
adr43.pla	0	0	7.9	0.1
adr44.pla	0	0	7.3	0.1
adr45.pla	0	0	7.7	0.1
b91.pla	2.1	1.8	27.0	0.1
b92.pla	2.1	1.7	29.3	0.1
b93.pla	2.0	1.8	31.6	0.1
b94.pla	1.8	1.8	50.8	0.1
b95.pla	1.9	1.8	22.9	0.1
cm152a.pla	0	0	1.2	0.1
co14.pla	1.1	1.0	11.2	0.1
ex1.pla	0	0	2.2	0.1
ex2.pla	0	0	1.1	0.1
ex3.pla	0	0	0.8	0.1
ex4.pla	0	0	1.0	0.1
life.pla	0	0	96.0	0.1
log8mod1.pla	0	0	1.1	0
log8mod2.pla	0	0	1.2	0
log8mod3.pla	0	0	1.2	0
log8mod4.pla	0	0	1.1	0
log8mod5.pla	0	0	1.2	0
max46.pla	0	0	17.9	0.1
parity.pla	0	0	36.7	0.1
ryy6.pla	1.9	1.7	103.4	0.1
sym10.pla	0.1	0.1	742.6	0.1

Table E.5: All timing results for the 2nd-order coefficient-based methods using xmap as the mapper.

function	Method 3	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8,8)	(26, 8,8)	(26, 8,8)	(26, 8,8)
add61.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
add62.pla	(6, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
add63.pla	(9, 2,2)	(9, 3,3)	(11, 2,2)	(13, 3,3)
add64.pla	(12, 3,3)	(12, 5,4)	(20, 3,3)	(22, 6,4)
add65.pla	(22, 5,4)	(21, 9,9)	(39, 11,10)	(31, 11,8)
add66.pla	(20, 4,4)	(19, 9,8)	(78, 41,37)	(91, 50,44)
add67.pla	(21, 4,4)	(20, 9,8)	(93, 36,34)	(81, 40,35)
adr41.pla	(16, 3,3)	(15, 5,4)	(32, 14,11)	(29, 13,12)
adr42.pla	(12, 3,3)	(12, 5,4)	(25, 10,9)	(22, 7,7)
adr43.pla	(9, 2,2)	(9, 3,3)	(13, 3,3)	(9, 3,3)
adr44.pla	(6, 1,1)	(6, 1,1)	(6, 1,1)	(6, 1,1)
adr45.pla	(4, 1,1)	(4, 1,1)	(4, 1,1)	(4, 1,1)
b91.pla	(19, 5,5)	(27, 6,6)	(15, 5,4)	(18, 3,3)
b92.pla	(24, 7,7)	(30, 9,8)	(21, 6,4)	(70, 29,23)
b93.pla	(27, 8,8)	(33, 11,11)	(27, 9,8)	(56, 25,23)
b94.pla	(28, 9,9)	(34, 12,12)	(32, 10,8)	(114, 35,33)
b95.pla	(19, 9,9)	(19, 8,8)	(15, 6,6)	(52, 19,18)
cm152a.pla	(17, 5,4)	(384, 148,122)	(17, 5,4)	(84, 45,37)
co14.pla	(71, 42,24)	(73, 42,24)	(71, 42,24)	(71, 42,24)
ex1.pla	(7, 1,1)	(7, 1,1)	(7, 1,1)	(7, 1,1)
ex2.pla	(12, 1,1)	(12, 1,1)	(11, 1,1)	(13, 1,1)
ex3.pla	(11, 1,1)	(10, 1,1)	(12, 1,1)	(11, 1,1)
ex4.pla	(11, 1,1)	(11, 1,1)	(11, 1,1)	(12, 1,1)
life.pla	(884, 412,412)	(857, 428,418)	(857, 428,418)	(884, 413,413)
log8mod1.pla	(11, 2,2)	(11, 3,3)	(11, 2,2)	(10, 2,2)
log8mod2.pla	(15, 3,3)	(18, 3,3)	(15, 3,3)	(17, 3,3)
log8mod3.pla	(17, 3,3)	(21, 3,3)	(17, 3,3)	(17, 3,3)
log8mod4.pla	(24, 4,4)	(32, 6,6)	(22, 4,4)	(26, 4,4)
log8mod5.pla	(27, 4,4)	(31, 4,4)	(24, 4,4)	(28, 6,6)
max46.pla	(83, 33,26)	(78, 34,28)	(79, 27,24)	(83, 35,27)
parity.pla	(10, 2,2)	(10, 2,2)	(10, 2,2)	(10, 2,2)
ryy6.pla	(31, 10,10)	(33, 11,10)	(85, 30,25)	(122, 59,49)
sym10.pla	(32, 12,12)	(32, 12,12)	(32, 12,12)	(32, 12,12)

Table E.6: All results for the ordering method based on the average of first and second coefficients and using xmap as the mapper.

function	Method 3	Item	random orderings
9sym.pla	0	38.8	0.1
add61.pla	0.0	0.0	0.1
add62.pla	0.0	0.0	0.1
add63.pla	0.0	0.2	0.1
add64.pla	0.0	1.4	0.1
add65.pla	0.0	7.4	0.1
add66.pla	0.0	49.4	0.1
add67.pla	0.0	33.2	0.1
adr41.pla	0	7.6	0.1
adr42.pla	0	7.9	0.1
adr43.pla	0	7.9	0.1
adr44.pla	0	7.3	0.1
adr45.pla	0	7.7	0.1
b91.pla	2.0	27.0	0.1
b92.pla	2.0	29.3	0.1
b93.pla	2.0	31.6	0.1
b94.pla	1.8	50.8	0.1
b95.pla	1.9	22.9	0.1
cm152a.pla	0.1	1.2	0.1
co14.pla	1.1	11.2	0.1
ex1.pla	0	2.2	0.1
ex2.pla	0	1.1	0.1
ex3.pla	0	0.8	0.1
ex4.pla	0	1.0	0.1
life.pla	0	96.0	0.1
log8mod1.pla	0	1.1	0
log8mod2.pla	0	1.2	0
log8mod3.pla	0	1.2	0
log8mod4.pla	0	1.1	0
log8mod5.pla	0	1.2	0
max46.pla	0	17.9	0.1
parity.pla	0	36.7	0.1
ryy6.pla	2.0	103.4	0.1
sym10.pla	0.1	742.6	0.1

Table E.7: All timing results for the ordering method based on the average of first and second coefficients and using xmap as the mapper.

Results using the Item technology mapping method xtmap

Xtmap is an algorithm based on a generate-and-test paradigm [Kar89]. The algorithm generates many possible single-cell implementations for a node, chooses the best one, then recursively applies the algorithm to the inputs to the chosen cell. The heuristics for choosing the best node can be set by parameters given by the user or determined by the algorithm.

function	**Method 1**	**reverse**	**Item**	**random orderings**
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
add63.pla	(9, 6, 3)	(9, 6, 3)	(11, 8, 4)	(13, 9, 5)
add64.pla	(12, 9, 5)	(12, 9, 5)	(20, 16, 8)	(22, 17, 9)
add65.pla	(15, 12, 6)	(15, 12, 7)	(39, 34, 17)	(31, 27, 14)
add66.pla	(18, 15, 8)	(18, 15, 9)	(78, 72, 37)	(91, 87, 46)
add67.pla	(19, 16, 8)	(19, 16, 9)	(93, 86, 43)	(81, 77, 39)
adr41.pla	(13, 10, 5)	(13, 10, 5)	(32, 29, 15)	(29, 25, 13)
adr42.pla	(12, 9, 5)	(12, 9, 5)	(25, 22, 12)	(22, 19, 10)
adr43.pla	(9, 6, 3)	(9, 6, 3)	(13, 10, 6)	(9, 6, 3)
adr44.pla	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 14, 8)	(25, 20, 10)	(15, 11, 6)	(18, 12, 6)
b92.pla	(22, 17, 9)	(28, 23, 12)	(21, 15, 8)	(70, 66, 33)
b93.pla	(27, 21, 11)	(33, 28, 14)	(27, 20, 10)	(56, 52, 26)
b94.pla	(30, 24, 13)	(36, 31, 16)	(32, 24, 12)	(114, 105, 53)
b95.pla	(19, 15, 8)	(19, 16, 9)	(15, 12, 6)	(52, 48, 25)
cm152a.pla	(17, 7, 4)	(384, 379, 190)	(17, 7, 4)	(84, 81, 41)
co14.pla	(73, 58, 29)	(71, 58, 29)	(71, 58, 29)	(71, 58, 29)
ex1.pla	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)
ex2.pla	(12, 8, 4)	(12, 9, 5)	(11, 7, 4)	(13, 10, 5)
ex3.pla	(12, 8, 4)	(12, 9, 5)	(12, 8, 4)	(11, 8, 4)
ex4.pla	(12, 7, 4)	(12, 9, 5)	(11, 7, 4)	(12, 9, 5)
life.pla	(884, 880, 440)	(857, 854, 427)	(857, 854, 427)	(884, 881, 441)
log8mod1.pla	(11, 7, 4)	(11, 8, 4)	(11, 6, 3)	(10, 6, 3)
log8mod2.pla	(15, 11, 6)	(18, 15, 8)	(15, 11, 6)	(17, 14, 7)
log8mod3.pla	(17, 12, 6)	(21, 18, 9)	(17, 14, 7)	(17, 13, 7)
log8mod4.pla	(24, 19, 10)	(32, 28, 14)	(22, 17, 9)	(26, 22, 11)
log8mod5.pla	(27, 22, 11)	(31, 27, 14)	(24, 19, 10)	(28, 24, 12)
max46.pla	(83, 79, 40)	(78, 75, 38)	(79, 75, 38)	(83, 80, 41)
parity.pla	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)
ryy6.pla	(31, 27, 14)	(33, 28, 14)	(85, 77, 39)	(122, 115, 58)
sym10.pla	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)

Table E.8: Results after ordering based on the first order coefficients and mapping with the xtmap method.

function	Method 2a)	Method 2b)	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(8, 4, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
add63.pla	(11, 7, 4)	(9, 6, 4)	(11, 8, 4)	(13, 9, 5)
add64.pla	(12, 9, 5)	(12, 9, 6)	(20, 16, 8)	(22, 17, 9)
add65.pla	(15, 12, 7)	(15, 12, 7)	(39, 34, 17)	(31, 27, 14)
add66.pla	(18, 15, 9)	(18, 15, 9)	(78, 72, 37)	(91, 87, 46)
add67.pla	(19, 16, 9)	(19, 16, 8)	(93, 86, 43)	(81, 77, 39)
adr41.pla	(13, 10, 5)	(13, 10, 5)	(32, 29, 15)	(29, 25, 13)
adr42.pla	(14, 10, 6)	(12, 9, 6)	(25, 22, 12)	(22, 19, 10)
adr43.pla	(11, 7, 4)	(9, 6, 4)	(13, 10, 6)	(9, 6, 3)
adr44.pla	(8, 4, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(20, 16, 9)	(16, 11, 6)	(15, 11, 6)	(18, 12, 6)
b92.pla	(28, 24, 12)	(24, 17, 9)	(21, 15, 8)	(70, 66, 33)
b93.pla	(31, 27, 14)	(27, 20, 10)	(27, 20, 10)	(56, 52, 26)
b94.pla	(34, 30, 15)	(28, 23, 12)	(32, 24, 12)	(114, 105, 53)
b95.pla	(15, 12, 6)	(19, 15, 8)	(15, 12, 6)	(52, 48, 25)
cm152a.pla	(384, 379, 190)	(17, 7, 4)	(17, 7, 4)	(84, 81, 41)
co14.pla	(71, 58, 29)	(73, 58, 29)	(71, 58, 29)	(71, 58, 29)
ex1.pla	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)
ex2.pla	(12, 9, 5)	(11, 7, 4)	(11, 7, 4)	(13, 10, 5)
ex3.pla	(11, 7, 4)	(11, 8, 4)	(12, 8, 4)	(11, 8, 4)
ex4.pla	(12, 8, 4)	(12, 8, 4)	(11, 7, 4)	(12, 9, 5)
life.pla	(857, 854, 427)	(857, 854, 427)	(857, 854, 427)	(884, 881, 441)
log8mod1.pla	(11, 8, 4)	(11, 6, 3)	(11, 6, 3)	(10, 6, 3)
log8mod2.pla	(18, 15, 8)	(15, 11, 6)	(15, 11, 6)	(17, 14, 7)
log8mod3.pla	(21, 17, 9)	(18, 13, 7)	(17, 14, 7)	(17, 13, 7)
log8mod4.pla	(27, 23, 12)	(25, 19, 10)	(22, 17, 9)	(26, 22, 11)
log8mod5.pla	(29, 25, 13)	(28, 23, 12)	(24, 19, 10)	(28, 24, 12)
max46.pla	(85, 82, 41)	(83, 79, 40)	(79, 75, 38)	(83, 80, 41)
parity.pla	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)
ryy6.pla	(29, 25, 13)	(29, 26, 14)	(85, 77, 39)	(122, 115, 58)
sym10.pla	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)

Table E.9: Results after ordering based on the second order coefficients and mapping with the xtmap method.

function	Method 3	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)	(26, 23, 12)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
add63.pla	(9, 6, 3)	(9, 6, 3)	(11, 8, 4)	(13, 9, 5)
add64.pla	(12, 9, 5)	(12, 9, 5)	(20, 16, 8)	(22, 17, 9)
add65.pla	(22, 19, 10)	(21, 18, 10)	(39, 34, 17)	(31, 27, 14)
add66.pla	(20, 16, 8)	(19, 16, 11)	(78, 72, 37)	(91, 87, 46)
add67.pla	(21, 17, 9)	(20, 17, 10)	(93, 86, 43)	(81, 77, 39)
adr41.pla	(16, 13, 7)	(15, 12, 6)	(32, 29, 15)	(29, 25, 13)
adr42.pla	(12, 9, 5)	(12, 9, 5)	(25, 22, 12)	(22, 19, 10)
adr43.pla	(9, 6, 3)	(9, 6, 3)	(13, 10, 6)	(9, 6, 3)
adr44.pla	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)	(6, 3, 2)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 14, 7)	(27, 23, 12)	(15, 11, 6)	(18, 12, 6)
b92.pla	(24, 18, 10)	(30, 25, 13)	(21, 15, 8)	(70, 66, 33)
b93.pla	(27, 21, 11)	(33, 28, 15)	(27, 20, 10)	(56, 52, 26)
b94.pla	(28, 23, 12)	(34, 30, 15)	(32, 24, 12)	(114, 105, 53)
b95.pla	(19, 15, 8)	(19, 16, 8)	(15, 12, 6)	(52, 48, 25)
cm152a.pla	(17, 7, 4)	(384, 379, 190)	(17, 7, 4)	(84, 81, 41)
co14.pla	(71, 58, 29)	(73, 58, 29)	(71, 58, 29)	(71, 58, 29)
ex1.pla	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)	(7, 4, 2)
ex2.pla	(12, 8, 4)	(12, 9, 5)	(11, 7, 4)	(13, 10, 5)
ex3.pla	(11, 8, 4)	(10, 7, 4)	(12, 8, 4)	(11, 8, 4)
ex4.pla	(11, 7, 4)	(11, 8, 4)	(11, 7, 4)	(12, 9, 5)
life.pla	(884, 880, 440)	(857, 854, 427)	(857, 854, 427)	(884, 881, 441)
log8mod1.pla	(11, 7, 4)	(11, 8, 4)	(11, 6, 3)	(10, 6, 3)
log8mod2.pla	(15, 11, 6)	(18, 15, 8)	(15, 11, 6)	(17, 14, 7)
log8mod3.pla	(17, 13, 7)	(21, 17, 9)	(17, 14, 7)	(17, 13, 7)
log8mod4.pla	(24, 19, 10)	(32, 28, 14)	(22, 17, 9)	(26, 22, 11)
log8mod5.pla	(27, 22, 11)	(31, 27, 14)	(24, 19, 10)	(28, 24, 12)
max46.pla	(83, 79, 40)	(78, 75, 38)	(79, 75, 38)	(83, 80, 41)
parity.pla	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)	(10, 7, 4)
ryy6.pla	(31, 27, 14)	(33, 28, 14)	(85, 77, 39)	(122, 115, 58)
sym10.pla	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)	(32, 29, 15)

Table E.10: Results after ordering based on the average of first and second order coefficients and mapping with the xtmap method.

ordering method	improvement over Item		improvement over Random	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
1st-order	11.15	11.18	23.54	25.80
truncated avg.	10.20	10.38	22.82	25.63
2nd-order v1	-58.09	-133.49	5.51	5.62
truncated avg.	3.26	1.01	14.51	14.82
2nd-order v2	11.89	10.96	24.24	24.89
truncated avg.	10.98	10.15	23.56	24.66
avg-order	10.71	9.97	23.00	24.27
truncated avg.	9.79	9.17	22.25	24.22

Table E.11: Summary of results using xtmap.

Results using the Item technology mapping method xcmap

According the the Item help files, this mapping algorithm is similar to xmap except that it uses the mimimum height for computing the expression using lutwidth-wide lookup tables.

function	**Method 1**	**reverse**	**Item**	**random ordering**
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
add63.pla	(9, 2, 2)	(9, 3, 3)	(11, 2, 2)	(13, 3, 3)
add64.pla	(12, 3, 3)	(12, 6, 5)	(20, 3, 3)	(22, 6, 4)
add65.pla	(15, 4, 4)	(15, 9, 7)	(39, 11, 10)	(31, 13, 8)
add66.pla	(18, 4, 4)	(18, 12, 9)	(78, 47, 37)	(91, 62, 49)
add67.pla	(19, 4, 4)	(19, 12, 9)	(93, 42, 34)	(81, 34, 29)
adr41.pla	(13, 3, 3)	(13, 6, 5)	(32, 10, 9)	(29, 15, 12)
adr42.pla	(12, 3, 3)	(12, 6, 5)	(25, 10, 9)	(22, 7, 7)
adr43.pla	(9, 2, 2)	(9, 3, 3)	(13, 3, 3)	(9, 3, 3)
adr44.pla	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 7, 6)	(25, 7, 6)	(15, 4, 3)	(18, 3, 3)
b92.pla	(22, 13, 9)	(28, 9, 8)	(21, 5, 4)	(70, 32, 21)
b93.pla	(27, 12, 9)	(33, 11, 10)	(27, 12, 9)	(56, 27, 21)
b94.pla	(30, 14, 11)	(36, 13, 12)	(32, 13, 9)	(114, 48, 38)
b95.pla	(19, 7, 7)	(19, 11, 11)	(15, 6, 6)	(52, 22, 20)
cm152a.pla	(17, 7, 4)	(384, 127, 96)	(17, 7, 4)	(84, 43, 30)
co14.pla	(73, 45, 27)	(71, 45, 27)	(71, 45, 27)	(71, 45, 27)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(12, 1, 1)	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)	(12, 1, 1)
life.pla	(884, 413, 413)	(857, 417, 416)	(857, 417, 416)	(884, 413, 413)
log8mod1.pla	(11, 3, 3)	(11, 3, 3)	(11, 3, 2)	(10, 2, 2)
log8mod2.pla	(15, 3, 3)	(18, 3, 3)	(15, 3, 3)	(17, 3, 3)
log8mod3.pla	(17, 3, 3)	(21, 3, 3)	(17, 3, 3)	(17, 3, 3)
log8mod4.pla	(24, 4, 4)	(32, 7, 6)	(22, 4, 4)	(26, 4, 4)
log8mod5.pla	(27, 4, 4)	(31, 4, 4)	(24, 4, 4)	(28, 7, 6)
max46.pla	(83, 31, 24)	(78, 26, 22)	(79, 30, 23)	(83, 31, 24)
parity.pla	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)
ryy6.pla	(31, 14, 13)	(33, 13, 12)	(85, 39, 32)	(122, 62, 48)
sym10.pla	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)

Table E.12: Results after ordering based on the first order coefficients and mapping with the xcmap method.

function	Method 2a)	Method 2b)	Item	random ordering
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(8, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
add63.pla	(11, 3, 3)	(9, 3, 3)	(11, 2, 2)	(13, 3, 3)
add64.pla	(12, 6, 5)	(12, 6, 5)	(20, 3, 3)	(22, 6, 4)
add65.pla	(15, 9, 7)	(15, 8, 7)	(39, 11, 10)	(31, 13, 8)
add66.pla	(18, 12, 9)	(18, 10, 9)	(78, 47, 37)	(91, 62, 49)
add67.pla	(19, 12, 9)	(19, 4, 4)	(93, 42, 34)	(81, 34, 29)
adr41.pla	(13, 6, 5)	(13, 3, 3)	(32, 10, 9)	(29, 15, 12)
adr42.pla	(14, 6, 5)	(12, 6, 5)	(25, 10, 9)	(22, 7, 7)
adr43.pla	(11, 3, 3)	(9, 3, 3)	(13, 3, 3)	(9, 3, 3)
adr44.pla	(8, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(20, 4, 4)	(16, 4, 3)	(15, 4, 3)	(18, 3, 3)
b92.pla	(28, 9, 8)	(24, 6, 5)	(21, 5, 4)	(70, 32, 21)
b93.pla	(31, 11, 10)	(27, 7, 6)	(27, 12, 9)	(56, 27, 21)
b94.pla	(34, 13, 12)	(28, 9, 8)	(32, 13, 9)	(114, 48, 38)
b95.pla	(15, 7, 7)	(19, 7, 7)	(15, 6, 6)	(52, 22, 20)
cm152a.pla	(384, 127, 96)	(17, 7, 4)	(17, 7, 4)	(84, 43, 30)
co14.pla	(71, 45, 27)	(73, 45, 27)	(71, 45, 27)	(71, 45, 27)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 1, 1)	(11, 1, 1)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(11, 1, 1)	(11, 1, 1)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)	(12, 1, 1)
life.pla	(857, 417, 416)	(857, 417, 416)	(857, 417, 416)	(884, 413, 413)
log8mod1.pla	(11, 3, 3)	(11, 2, 2)	(11, 3, 2)	(10, 2, 2)
log8mod2.pla	(18, 3, 3)	(15, 3, 3)	(15, 3, 3)	(17, 3, 3)
log8mod3.pla	(21, 3, 3)	(18, 3, 3)	(17, 3, 3)	(17, 3, 3)
log8mod4.pla	(27, 7, 6)	(25, 4, 4)	(22, 4, 4)	(26, 4, 4)
log8mod5.pla	(29, 4, 4)	(28, 4, 4)	(24, 4, 4)	(28, 7, 6)
max46.pla	(85, 30, 23)	(83, 31, 24)	(79, 30, 23)	(83, 31, 24)
parity.pla	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)
ryy6.pla	(29, 12, 11)	(29, 15, 14)	(85, 39, 32)	(122, 62, 48)
sym10.pla	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)

Table E.13: Results after ordering based on the second order coefficients and mapping with the xcmap method.

function	**Method 3**	**reverse**	**Item**	**random ordering**
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)	(26, 8, 8)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
add63.pla	(9, 2, 2)	(9, 3, 3)	(11, 2, 2)	(13, 3, 3)
add64.pla	(12, 3, 3)	(12, 6, 5)	(20, 3, 3)	(22, 6, 4)
add65.pla	(22, 5, 4)	(21, 9, 7)	(39, 11, 10)	(31, 13, 8)
add66.pla	(20, 4, 4)	(19, 12, 9)	(78, 47, 37)	(91, 62, 49)
add67.pla	(21, 4, 4)	(20, 12, 9)	(93, 42, 34)	(81, 34, 29)
adr41.pla	(16, 3, 3)	(15, 6, 5)	(32, 10, 9)	(29, 15, 12)
adr42.pla	(12, 3, 3)	(12, 6, 5)	(25, 10, 9)	(22, 7, 7)
adr43.pla	(9, 2, 2)	(9, 3, 3)	(13, 3, 3)	(9, 3, 3)
adr44.pla	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 7, 5)	(27, 7, 6)	(15, 4, 3)	(18, 3, 3)
b92.pla	(24, 13, 9)	(30, 9, 8)	(21, 5, 4)	(70, 32, 21)
b93.pla	(27, 12, 9)	(33, 11, 10)	(27, 12, 9)	(56, 27, 21)
b94.pla	(28, 10, 8)	(34, 13, 12)	(32, 13, 9)	(114, 48, 38)
b95.pla	(19, 6, 6)	(19, 11, 10)	(15, 6, 6)	(52, 22, 20)
cm152a.pla	(17, 7, 4)	(384, 127, 96)	(17, 7, 4)	(84, 43, 30)
co14.pla	(71, 45, 27)	(73, 45, 27)	(71, 45, 27)	(71, 45, 27)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(11, 1, 1)	(10, 1, 1)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(11, 1, 1)	(11, 1, 1)	(11, 1, 1)	(12, 1, 1)
life.pla	(884, 413, 413)	(857, 417, 416)	(857, 417, 416)	(884, 413, 413)
log8mod1.pla	(11, 3, 3)	(11, 3, 3)	(11, 3, 2)	(10, 2, 2)
log8mod2.pla	(15, 3, 3)	(18, 3, 3)	(15, 3, 3)	(17, 3, 3)
log8mod3.pla	(17, 3, 3)	(21, 3, 3)	(17, 3, 3)	(17, 3, 3)
log8mod4.pla	(24, 4, 4)	(32, 7, 6)	(22, 4, 4)	(26, 4, 4)
log8mod5.pla	(27, 4, 4)	(31, 4, 4)	(24, 4, 4)	(28, 7, 6)
max46.pla	(83, 31, 24)	(78, 26, 22)	(79, 30, 23)	(83, 31, 24)
parity.pla	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)	(10, 2, 2)
ryy6.pla	(31, 14, 13)	(33, 13, 12)	(85, 39, 32)	(122, 62, 48)
sym10.pla	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)	(32, 12, 12)

Table E.14: Results after ordering based on the average of first and second order coefficients and mapping with the xcmap method.

ordering method	**improvement over Item**		**improvement over Random**	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
1st-order	11.15	4.29	23.54	21.91
truncated avg.	10.20	5.68	22.82	25.54
2nd-order v1	-58.09	-69.92	5.51	6.86
truncated avg.	3.26	-4.78	14.51	11.61
2nd-order v2	11.89	7.15	24.24	21.78
truncated avg.	10.98	6.92	23.56	21.21
avg-order	10.71	6.74	23.00	23.27
truncated avg.	9.79	8.28	22.25	24.94

Table E.15: Summary of results using xcmap.

Results using the Item technology mapping method fanout -h 2

The fanout algorithm decomposes the network so that there is a clb for every node whose fanout >= the value given after the -h flag.

function	Method 1	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 3)	(6, 2, 2)	(6, 3, 3)	(6, 3, 3)
add63.pla	(9, 4, 4)	(9, 4, 4)	(11, 4, 4)	(13, 4, 4)
add64.pla	(12, 5, 5)	(12, 7, 7)	(20, 6, 6)	(22, 10, 10)
add65.pla	(15, 6, 6)	(15, 10, 10)	(39, 12, 12)	(31, 13, 13)
add66.pla	(18, 7, 7)	(18, 13, 13)	(78, 26, 26)	(91, 26, 26)
add67.pla	(19, 6, 6)	(19, 14, 14)	(93, 24, 24)	(81, 22, 22)
adr41.pla	(13, 4, 4)	(13, 8, 8)	(32, 7, 7)	(29, 9, 9)
adr42.pla	(12, 5, 5)	(12, 7, 7)	(25, 8, 8)	(22, 11, 11)
adr43.pla	(9, 4, 4)	(9, 4, 4)	(13, 4, 4)	(9, 4, 4)
adr44.pla	(6, 3, 3)	(6, 2, 2)	(6, 2, 2)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 3, 3)	(25, 8, 8)	(15, 2, 2)	(18, 3, 3)
b92.pla	(22, 5, 5)	(28, 11, 11)	(21, 3, 3)	(70, 19, 19)
b93.pla	(27, 5, 5)	(33, 13, 13)	(27, 6, 6)	(56, 23, 23)
b94.pla	(30, 6, 6)	(36, 15, 15)	(32, 7, 7)	(114, 43, 43)
b95.pla	(19, 5, 5)	(19, 8, 8)	(15, 5, 5)	(52, 17, 17)
cm152a.pla	(17, 1, 1)	(384, 125, 125)	(17, 1, 1)	(84, 19, 19)
co14.pla	(73, 45, 45)	(71, 45, 45)	(71, 45, 45)	(71, 45, 45)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 2, 2)	(12, 3, 3)	(11, 2, 2)	(13, 2, 2)
ex3.pla	(12, 1, 1)	(12, 2, 2)	(12, 1, 1)	(11, 3, 3)
ex4.pla	(12, 2, 2)	(12, 3, 3)	(11, 2, 2)	(12, 3, 3)
life.pla	(884, 753, 753)	(857, 756, 756)	(857, 756, 756)	(884, 751, 751)
log8mod1.pla	(11, 1, 1)	(11, 3, 3)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(15, 2, 2)	(18, 5, 5)	(15, 2, 2)	(17, 5, 5)
log8mod3.pla	(17, 3, 3)	(21, 6, 6)	(17, 2, 2)	(17, 2, 2)
log8mod4.pla	(24, 2, 2)	(32, 8, 8)	(22, 4, 4)	(26, 7, 7)
log8mod5.pla	(27, 5, 5)	(31, 9, 9)	(24, 2, 2)	(28, 7, 7)
max46.pla	(83, 17, 17)	(78, 20, 20)	(79, 18, 18)	(83, 18, 18)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(31, 8, 8)	(33, 9, 9)	(85, 11, 11)	(122, 30, 30)
sym10.pla	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)

Table E.16: Results after ordering based on the first order coefficients and mapping with the fanout -h 2 method.

function	Method 2a)	Method 2b)	Item	random ordering
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(8, 2, 2)	(6, 2, 2)	(6, 3, 3)	(6, 3, 3)
add63.pla	(11, 5, 5)	(9, 3, 3)	(11, 4, 4)	(13, 4, 4)
add64.pla	(12, 7, 7)	(12, 4, 4)	(20, 6, 6)	(22, 10, 10)
add65.pla	(15, 10, 10)	(15, 5, 5)	(39, 12, 12)	(31, 13, 13)
add66.pla	(18, 13, 13)	(18, 6, 6)	(78, 26, 26)	(91, 26, 26)
add67.pla	(19, 14, 14)	(19, 6, 6)	(93, 24, 24)	(81, 22, 22)
adr41.pla	(13, 8, 8)	(13, 4, 4)	(32, 7, 7)	(29, 9, 9)
adr42.pla	(14, 8, 8)	(12, 4, 4)	(25, 8, 8)	(22, 11, 11)
adr43.pla	(11, 5, 5)	(9, 3, 3)	(13, 4, 4)	(9, 4, 4)
adr44.pla	(8, 2, 2)	(6, 2, 2)	(6, 2, 2)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(20, 6, 6)	(16, 2, 2)	(15, 2, 2)	(18, 3, 3)
b92.pla	(28, 12, 12)	(24, 5, 5)	(21, 3, 3)	(70, 19, 19)
b93.pla	(31, 14, 14)	(27, 6, 6)	(27, 6, 6)	(56, 23, 23)
b94.pla	(34, 16, 16)	(28, 9, 9)	(32, 7, 7)	(114, 43, 43)
b95.pla	(15, 7, 7)	(19, 5, 5)	(15, 5, 5)	(52, 17, 17)
cm152a.pla	(384, 125, 125)	(17, 1, 1)	(17, 1, 1)	(84, 19, 19)
co14.pla	(71, 45, 45)	(73, 45, 45)	(71, 45, 45)	(71, 45, 45)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 4, 4)	(11, 2, 2)	(11, 2, 2)	(13, 2, 2)
ex3.pla	(11, 1, 1)	(11, 2, 2)	(12, 1, 1)	(11, 3, 3)
ex4.pla	(12, 4, 4)	(12, 2, 2)	(11, 2, 2)	(12, 3, 3)
life.pla	(857, 756, 756)	(857, 756, 756)	(857, 756, 756)	(884, 751, 751)
log8mod1.pla	(11, 3, 3)	(11, 1, 1)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(18, 5, 5)	(15, 2, 2)	(15, 2, 2)	(17, 5, 5)
log8mod3.pla	(21, 5, 5)	(18, 4, 4)	(17, 2, 2)	(17, 2, 2)
log8mod4.pla	(27, 9, 9)	(25, 2, 2)	(22, 4, 4)	(26, 7, 7)
log8mod5.pla	(29, 9, 9)	(28, 4, 4)	(24, 2, 2)	(28, 7, 7)
max46.pla	(85, 16, 16)	(83, 18, 18)	(79, 18, 18)	(83, 18, 18)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(29, 10, 10)	(29, 7, 7)	(85, 11, 11)	(122, 30, 30)
sym10.pla	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)

Table E.17: Results after ordering based on the second order coefficients and mapping with the fanout -h 2 method.

function	Method 3	reverse	Item	random ordering
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)	(26, 17, 17)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 3)	(6, 2, 2)	(6, 3, 3)	(6, 3, 3)
add63.pla	(9, 4, 4)	(9, 4, 4)	(11, 4, 4)	(13, 4, 4)
add64.pla	(12, 5, 5)	(12, 7, 7)	(20, 6, 6)	(22, 10, 10)
add65.pla	(22, 9, 9)	(21, 10, 10)	(39, 12, 12)	(31, 13, 13)
add66.pla	(20, 6, 6)	(19, 12, 12)	(78, 26, 26)	(91, 26, 26)
add67.pla	(21, 5, 5)	(20, 13, 13)	(93, 24, 24)	(81, 22, 22)
adr41.pla	(16, 5, 5)	(15, 6, 6)	(32, 7, 7)	(29, 9, 9)
adr42.pla	(12, 5, 5)	(12, 7, 7)	(25, 8, 8)	(22, 11, 11)
adr43.pla	(9, 4, 4)	(9, 4, 4)	(13, 4, 4)	(9, 4, 4)
adr44.pla	(6, 3, 3)	(6, 2, 2)	(6, 2, 2)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 5, 5)	(27, 9, 9)	(15, 2, 2)	(18, 3, 3)
b92.pla	(24, 4, 4)	(30, 11, 11)	(21, 3, 3)	(70, 19, 19)
b93.pla	(27, 5, 5)	(33, 13, 13)	(27, 6, 6)	(56, 23, 23)
b94.pla	(28, 7, 7)	(34, 16, 16)	(32, 7, 7)	(114, 43, 43)
b95.pla	(19, 6, 6)	(19, 9, 9)	(15, 5, 5)	(52, 17, 17)
cm152a.pla	(17, 1, 1)	(384, 125, 125)	(17, 1, 1)	(84, 19, 19)
co14.pla	(71, 45, 45)	(73, 45, 45)	(71, 45, 45)	(71, 45, 45)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 2, 2)	(12, 3, 3)	(11, 2, 2)	(13, 2, 2)
ex3.pla	(11, 2, 2)	(10, 2, 2)	(12, 1, 1)	(11, 3, 3)
ex4.pla	(11, 2, 2)	(11, 2, 2)	(11, 2, 2)	(12, 3, 3)
life.pla	(884, 753, 753)	(857, 756, 756)	(857, 756, 756)	(884, 751, 751)
log8mod1.pla	(11, 1, 1)	(11, 3, 3)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(15, 2, 2)	(18, 5, 5)	(15, 2, 2)	(17, 5, 5)
log8mod3.pla	(17, 4, 4)	(21, 5, 5)	(17, 2, 2)	(17, 2, 2)
log8mod4.pla	(24, 2, 2)	(32, 8, 8)	(22, 4, 4)	(26, 7, 7)
log8mod5.pla	(27, 5, 5)	(31, 9, 9)	(24, 2, 2)	(28, 7, 7)
max46.pla	(83, 16, 16)	(78, 20, 20)	(79, 18, 18)	(83, 18, 18)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(31, 8, 8)	(33, 9, 9)	(85, 11, 11)	(122, 30, 30)
sym10.pla	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)	(32, 21, 21)

Table E.18: Results after ordering based on the average of first and second order coefficients and mapping with the fanout -h 2 method.

ordering method	**improvement over Item**		**improvement over Random**	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
1st-order	11.15	1.25	23.54	30.93
truncated avg.	10.20	3.68	22.82	31.47
2nd-order v1	-58.09	-420.47	5.51	-19.49
truncated avg.	3.26	-60.81	14.51	-5.35
2nd-order v2	11.89	3.26	24.24	33.80
truncated avg.	10.98	4.19	23.56	36.08
avg-order	10.71	-6.88	23.00	25.84
truncated avg.	9.79	-5.10	22.25	27.62

Table E.19: Summary of results using fanout -h 2.

Results using the Item technology mapping method fanout -h 3

function	Method 1	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
add63.pla	(9, 1, 1)	(9, 2, 2)	(11, 1, 1)	(13, 1, 1)
add64.pla	(12, 1, 1)	(12, 2, 2)	(20, 3, 3)	(22, 2, 2)
add65.pla	(15, 1, 1)	(15, 2, 2)	(39, 7, 7)	(31, 3, 3)
add66.pla	(18, 1, 1)	(18, 2, 2)	(78, 15, 15)	(91, 6, 6)
add67.pla	(19, 1, 1)	(19, 1, 1)	(93, 1, 1)	(81, 5, 5)
adr41.pla	(13, 1, 1)	(13, 1, 1)	(32, 7, 7)	(29, 1, 1)
adr42.pla	(12, 1, 1)	(12, 2, 2)	(25, 4, 4)	(22, 1, 1)
adr43.pla	(9, 1, 1)	(9, 2, 2)	(13, 2, 2)	(9, 2, 2)
adr44.pla	(6, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 1, 1)	(25, 2, 2)	(15, 1, 1)	(18, 3, 3)
b92.pla	(22, 1, 1)	(28, 2, 2)	(21, 1, 1)	(70, 9, 9)
b93.pla	(27, 1, 1)	(33, 4, 4)	(27, 2, 2)	(56, 9, 9)
b94.pla	(30, 1, 1)	(36, 5, 5)	(32, 2, 2)	(114, 7, 7)
b95.pla	(19, 2, 2)	(19, 6, 6)	(15, 1, 1)	(52, 7, 7)
cm152a.pla	(17, 1, 1)	(384, 6, 6)	(17, 1, 1)	(84, 15, 15)
co14.pla	(73, 34, 34)	(71, 34, 34)	(71, 34, 34)	(71, 34, 34)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 1, 1)	(12, 2, 2)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(12, 1, 1)	(12, 2, 2)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(12, 1, 1)	(12, 2, 2)	(11, 1, 1)	(12, 2, 2)
life.pla	(884, 376, 376)	(857, 380, 380)	(857, 380, 380)	(884, 377, 377)
log8mod1.pla	(11, 1, 1)	(11, 2, 2)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(15, 1, 1)	(18, 3, 3)	(15, 1, 1)	(17, 1, 1)
log8mod3.pla	(17, 1, 1)	(21, 4, 4)	(17, 1, 1)	(17, 1, 1)
log8mod4.pla	(24, 1, 1)	(32, 4, 4)	(22, 1, 1)	(26, 1, 1)
log8mod5.pla	(27, 1, 1)	(31, 4, 4)	(24, 2, 2)	(28, 2, 2)
max46.pla	(83, 8, 8)	(78, 12, 12)	(79, 12, 12)	(83, 11, 11)
parity.pla	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)
ryy6.pla	(31, 5, 5)	(33, 4, 4)	(85, 10, 10)	(122, 13, 13)
sym10.pla	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)

Table E.20: Results after ordering based on the first order coefficients and mapping with the fanout -h 3 method in Item.

function	Method 2a)	Method 2b)	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(8, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
add63.pla	(11, 1, 1)	(9, 2, 2)	(11, 1, 1)	(13, 1, 1)
add64.pla	(12, 2, 2)	(12, 2, 2)	(20, 3, 3)	(22, 2, 2)
add65.pla	(15, 2, 2)	(15, 2, 2)	(39, 7, 7)	(31, 3, 3)
add66.pla	(18, 2, 2)	(18, 2, 2)	(78, 15, 15)	(91, 6, 6)
add67.pla	(19, 1, 1)	(19, 1, 1)	(93, 1, 1)	(81, 5, 5)
adr41.pla	(13, 1, 1)	(13, 1, 1)	(32, 7, 7)	(29, 1, 1)
adr42.pla	(14, 1, 1)	(12, 2, 2)	(25, 4, 4)	(22, 1, 1)
adr43.pla	(11, 1, 1)	(9, 2, 2)	(13, 2, 2)	(9, 2, 2)
adr44.pla	(8, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(20, 3, 3)	(16, 1, 1)	(15, 1, 1)	(18, 3, 3)
b92.pla	(28, 3, 3)	(24, 1, 1)	(21, 1, 1)	(70, 9, 9)
b93.pla	(31, 4, 4)	(27, 1, 1)	(27, 2, 2)	(56, 9, 9)
b94.pla	(34, 5, 5)	(28, 1, 1)	(32, 2, 2)	(114, 7, 7)
b95.pla	(15, 5, 5)	(19, 2, 2)	(15, 1, 1)	(52, 7, 7)
cm152a.pla	(384, 6, 6)	(17, 1, 1)	(17, 1, 1)	(84, 15, 15)
co14.pla	(71, 34, 34)	(73, 34, 34)	(71, 34, 34)	(71, 34, 34)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 2, 2)	(11, 1, 1)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(11, 1, 1)	(11, 1, 1)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(12, 1, 1)	(12, 1, 1)	(11, 1, 1)	(12, 2, 2)
life.pla	(857, 380, 380)	(857, 380, 380)	(857, 380, 380)	(884, 377, 377)
log8mod1.pla	(11, 2, 2)	(11, 1, 1)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(18, 3, 3)	(15, 1, 1)	(15, 1, 1)	(17, 1, 1)
log8mod3.pla	(21, 3, 3)	(18, 1, 1)	(17, 1, 1)	(17, 1, 1)
log8mod4.pla	(27, 2, 2)	(25, 2, 2)	(22, 1, 1)	(26, 1, 1)
log8mod5.pla	(29, 4, 4)	(28, 2, 2)	(24, 2, 2)	(28, 2, 2)
max46.pla	(85, 9, 9)	(83, 11, 11)	(79, 12, 12)	(83, 11, 11)
parity.pla	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)
ryy6.pla	(29, 5, 5)	(29, 4, 4)	(85, 10, 10)	(122, 13, 13)
sym10.pla	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)

Table E.21: Results after ordering based on the second order coefficients and mapping with the fanout -h 3 method in Item.

function	Method 3	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)	(26, 1, 1)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
add63.pla	(9, 1, 1)	(9, 2, 2)	(11, 1, 1)	(13, 1, 1)
add64.pla	(12, 1, 1)	(12, 2, 2)	(20, 3, 3)	(22, 2, 2)
add65.pla	(22, 1, 1)	(21, 3, 3)	(39, 7, 7)	(31, 3, 3)
add66.pla	(20, 1, 1)	(19, 4, 4)	(78, 15, 15)	(91, 6, 6)
add67.pla	(21, 1, 1)	(20, 3, 3)	(93, 1, 1)	(81, 5, 5)
adr41.pla	(16, 1, 1)	(15, 1, 1)	(32, 7, 7)	(29, 1, 1)
adr42.pla	(12, 1, 1)	(12, 2, 2)	(25, 4, 4)	(22, 1, 1)
adr43.pla	(9, 1, 1)	(9, 2, 2)	(13, 2, 2)	(9, 2, 2)
adr44.pla	(6, 1, 1)	(6, 2, 2)	(6, 1, 1)	(6, 1, 1)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 1, 1)	(27, 3, 3)	(15, 1, 1)	(18, 3, 3)
b92.pla	(24, 1, 1)	(30, 4, 4)	(21, 1, 1)	(70, 9, 9)
b93.pla	(27, 1, 1)	(33, 4, 4)	(27, 2, 2)	(56, 9, 9)
b94.pla	(28, 1, 1)	(34, 5, 5)	(32, 2, 2)	(114, 7, 7)
b95.pla	(19, 1, 1)	(19, 4, 4)	(15, 1, 1)	(52, 7, 7)
cm152a.pla	(17, 1, 1)	(384, 6, 6)	(17, 1, 1)	(84, 15, 15)
co14.pla	(71, 34, 34)	(73, 34, 34)	(71, 34, 34)	(71, 34, 34)
ex1.pla	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)	(7, 1, 1)
ex2.pla	(12, 1, 1)	(12, 2, 2)	(11, 1, 1)	(13, 1, 1)
ex3.pla	(11, 2, 2)	(10, 1, 1)	(12, 1, 1)	(11, 1, 1)
ex4.pla	(11, 1, 1)	(11, 2, 2)	(11, 1, 1)	(12, 2, 2)
life.pla	(884, 376, 376)	(857, 380, 380)	(857, 380, 380)	(884, 377, 377)
log8mod1.pla	(11, 1, 1)	(11, 2, 2)	(11, 1, 1)	(10, 1, 1)
log8mod2.pla	(15, 1, 1)	(18, 3, 3)	(15, 1, 1)	(17, 1, 1)
log8mod3.pla	(17, 1, 1)	(21, 2, 2)	(17, 1, 1)	(17, 1, 1)
log8mod4.pla	(24, 1, 1)	(32, 4, 4)	(22, 1, 1)	(26, 1, 1)
log8mod5.pla	(27, 1, 1)	(31, 4, 4)	(24, 2, 2)	(28, 2, 2)
max46.pla	(83, 8, 8)	(78, 12, 12)	(79, 12, 12)	(83, 11, 11)
parity.pla	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)	(10, 1, 1)
ryy6.pla	(31, 5, 5)	(33, 4, 4)	(85, 10, 10)	(122, 13, 13)
sym10.pla	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)	(32, 1, 1)

Table E.22: Results after ordering based on the average of the first and second order coefficients and mapping with the fanout -h 3 method in Item.

ordering method	improvement over Item		improvement over Random	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
1st-order	11.15	17.38	23.54	29.82
truncated avg.	10.20	18.67	22.82	28.77
2nd-order v1	-58.09	-55.08	5.51	-5.93
truncated avg.	3.26	-45.61	14.51	-2.55
2nd-order v2	11.89	-0.13	24.24	8.63
truncated avg.	10.98	0.28	23.56	9.38
avg-order	10.71	17.38	23.00	27.30
truncated avg.	9.79	18.67	22.25	29.22

Table E.23: Summary of results using fanout -h 3.

Results using the Item technology mapping method internal

This mapping algorithm is described in the Item help files as simply creating a clb output for every if-then-else triple in the the internal representation (equivalent to "fanout -h 1").

function	**Method 1**	**reverse**	**Item**	**random orderings**
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
add63.pla	(9, 6, 6)	(9, 6, 6)	(11, 8, 8)	(13, 9, 9)
add64.pla	(12, 9, 9)	(12, 9, 9)	(20, 16, 16)	(22, 17, 17)
add65.pla	(15, 12, 12)	(15, 12, 12)	(39, 34, 34)	(31, 27, 27)
add66.pla	(18, 15, 15)	(18, 15, 15)	(78, 72, 72)	(91, 87, 87)
add67.pla	(19, 16, 16)	(19, 16, 16)	(93, 86, 86)	(81, 77, 77)
adr41.pla	(13, 10, 10)	(13, 10, 10)	(32, 29, 29)	(29, 25, 25)
adr42.pla	(12, 9, 9)	(12, 9, 9)	(25, 22, 22)	(22, 19, 19)
adr43.pla	(9, 6, 6)	(9, 6, 6)	(13, 10, 10)	(9, 6, 6)
adr44.pla	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 14, 14)	(25, 20, 20)	(15, 11, 11)	(18, 12, 12)
b92.pla	(22, 17, 17)	(28, 23, 23)	(21, 15, 15)	(70, 66, 66)
b93.pla	(27, 21, 21)	(33, 28, 28)	(27, 20, 20)	(56, 52, 52)
b94.pla	(30, 24, 24)	(36, 31, 31)	(32, 24, 24)	(114, 105, 105)
b95.pla	(19, 15, 15)	(19, 16, 16)	(15, 12, 12)	(52, 48, 48)
cm152a.pla	(17, 7, 7)	(384, 379, 379)	(17, 7, 7)	(84, 81, 81)
co14.pla	(73, 58, 58)	(71, 58, 58)	(71, 58, 58)	(71, 58, 58)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 8, 8)	(12, 9, 9)	(11, 7, 7)	(13, 10, 10)
ex3.pla	(12, 8, 8)	(12, 9, 9)	(12, 8, 8)	(11, 8, 8)
ex4.pla	(12, 7, 7)	(12, 9, 9)	(11, 7, 7)	(12, 9, 9)
life.pla	(884, 880, 880)	(857, 854, 854)	(857, 854, 854)	(884, 881, 881)
log8mod1.pla	(11, 7, 7)	(11, 8, 8)	(11, 6, 6)	(10, 6, 6)
log8mod2.pla	(15, 11, 11)	(18, 15, 15)	(15, 11, 11)	(17, 14, 14)
log8mod3.pla	(17, 12, 12)	(21, 18, 18)	(17, 14, 14)	(17, 13, 13)
log8mod4.pla	(24, 19, 19)	(32, 28, 28)	(22, 17, 17)	(26, 22, 22)
log8mod5.pla	(27, 22, 22)	(31, 27, 27)	(24, 19, 19)	(28, 24, 24)
max46.pla	(83, 79, 79)	(78, 75, 75)	(79, 75, 75)	(83, 80, 80)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(31, 27, 27)	(33, 28, 28)	(85, 77, 77)	(122, 115, 115)
sym10.pla	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)

Table E.24: First order autocorrelation coefficient ordering results after mapping with the internal method.

function	Method 2a)	Method 2b)	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(8, 4, 4)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
add63.pla	(11, 7, 7)	(9, 6, 6)	(11, 8, 8)	(13, 9, 9)
add64.pla	(12, 9, 9)	(12, 9, 9)	(20, 16, 16)	(22, 17, 17)
add65.pla	(15, 12, 12)	(15, 12, 12)	(39, 34, 34)	(31, 27, 27)
add66.pla	(18, 15, 15)	(18, 15, 15)	(78, 72, 72)	(91, 87, 87)
add67.pla	(19, 16, 16)	(19, 16, 16)	(93, 86, 86)	(81, 77, 77)
adr41.pla	(13, 10, 10)	(13, 10, 10)	(32, 29, 29)	(29, 25, 25)
adr42.pla	(14, 10, 10)	(12, 9, 9)	(25, 22, 22)	(22, 19, 19)
adr43.pla	(11, 7, 7)	(9, 6, 6)	(13, 10, 10)	(9, 6, 6)
adr44.pla	(8, 4, 4)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(20, 16, 16)	(16, 11, 11)	(15, 11, 11)	(18, 12, 12)
b92.pla	(28, 24, 24)	(24, 17, 17)	(21, 15, 15)	(70, 66, 66)
b93.pla	(31, 27, 27)	(27, 20, 20)	(27, 20, 20)	(56, 52, 52)
b94.pla	(34, 30, 30)	(28, 23, 23)	(32, 24, 24)	(114, 105, 105)
b95.pla	(15, 12, 12)	(19, 15, 15)	(15, 12, 12)	(52, 48, 48)
cm152a.pla	(384, 379, 379)	(17, 7, 7)	(17, 7, 7)	(84, 81, 81)
co14.pla	(71, 58, 58)	(73, 58, 58)	(71, 58, 58)	(71, 58, 58)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 9, 9)	(11, 7, 7)	(11, 7, 7)	(13, 10, 10)
ex3.pla	(11, 7, 7)	(11, 8, 8)	(12, 8, 8)	(11, 8, 8)
ex4.pla	(12, 8, 8)	(12, 8, 8)	(11, 7, 7)	(12, 9, 9)
life.pla	(857, 854, 854)	(857, 854, 854)	(857, 854, 854)	(884, 881, 881)
log8mod1.pla	(11, 8, 8)	(11, 6, 6)	(11, 6, 6)	(10, 6, 6)
log8mod2.pla	(18, 15, 15)	(15, 11, 11)	(15, 11, 11)	(17, 14, 14)
log8mod3.pla	(21, 17, 17)	(18, 13, 13)	(17, 14, 14)	(17, 13, 13)
log8mod4.pla	(27, 23, 23)	(25, 19, 19)	(22, 17, 17)	(26, 22, 22)
log8mod5.pla	(29, 25, 25)	(28, 23, 23)	(24, 19, 19)	(28, 24, 24)
max46.pla	(85, 82, 82)	(83, 79, 79)	(79, 75, 75)	(83, 80, 80)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(29, 25, 25)	(29, 26, 26)	(85, 77, 77)	(122, 115, 115)
sym10.pla	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)

Table E.25: Second order autocorrelation coefficient ordering results after mapping with the internal method.

function	Method 3	reverse	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
9sym.pla	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)	(26, 23, 23)
add61.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
add62.pla	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
add63.pla	(9, 6, 6)	(9, 6, 6)	(11, 8, 8)	(13, 9, 9)
add64.pla	(12, 9, 9)	(12, 9, 9)	(20, 16, 16)	(22, 17, 17)
add65.pla	(22, 19, 19)	(21, 18, 18)	(39, 34, 34)	(31, 27, 27)
add66.pla	(20, 16, 16)	(19, 16, 16)	(78, 72, 72)	(91, 87, 87)
add67.pla	(21, 17, 17)	(20, 17, 17)	(93, 86, 86)	(81, 77, 77)
adr41.pla	(16, 13, 13)	(15, 12, 12)	(32, 29, 29)	(29, 25, 25)
adr42.pla	(12, 9, 9)	(12, 9, 9)	(25, 22, 22)	(22, 19, 19)
adr43.pla	(9, 6, 6)	(9, 6, 6)	(13, 10, 10)	(9, 6, 6)
adr44.pla	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)	(6, 3, 3)
adr45.pla	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)	(4, 1, 1)
b91.pla	(19, 14, 14)	(27, 23, 23)	(15, 11, 11)	(18, 12, 12)
b92.pla	(24, 18, 18)	(30, 25, 25)	(21, 15, 15)	(70, 66, 66)
b93.pla	(27, 21, 21)	(33, 28, 28)	(27, 20, 20)	(56, 52, 52)
b94.pla	(28, 23, 23)	(34, 30, 30)	(32, 24, 24)	(114, 105, 105)
b95.pla	(19, 15, 15)	(19, 16, 16)	(15, 12, 12)	(52, 48, 48)
cm152a.pla	(17, 7, 7)	(384, 379, 379)	(17, 7, 7)	(84, 81, 81)
co14.pla	(71, 58, 58)	(73, 58, 58)	(71, 58, 58)	(71, 58, 58)
ex1.pla	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)	(7, 4, 4)
ex2.pla	(12, 8, 8)	(12, 9, 9)	(11, 7, 7)	(13, 10, 10)
ex3.pla	(11, 8, 8)	(10, 7, 7)	(12, 8, 8)	(11, 8, 8)
ex4.pla	(11, 7, 7)	(11, 8, 8)	(11, 7, 7)	(12, 9, 9)
life.pla	(884, 880, 880)	(857, 854, 854)	(857, 854, 854)	(884, 881, 881)
log8mod1.pla	(11, 7, 7)	(11, 8, 8)	(11, 6, 6)	(10, 6, 6)
log8mod2.pla	(15, 11, 11)	(18, 15, 15)	(15, 11, 11)	(17, 14, 14)
log8mod3.pla	(17, 13, 13)	(21, 17, 17)	(17, 14, 14)	(17, 13, 13)
log8mod4.pla	(24, 19, 19)	(32, 28, 28)	(22, 17, 17)	(26, 22, 22)
log8mod5.pla	(27, 22, 22)	(31, 27, 27)	(24, 19, 19)	(28, 24, 24)
max46.pla	(83, 79, 79)	(78, 75, 75)	(79, 75, 75)	(83, 80, 80)
parity.pla	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)	(10, 7, 7)
ryy6.pla	(31, 27, 27)	(33, 28, 28)	(85, 77, 77)	(122, 115, 115)
sym10.pla	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)	(32, 29, 29)

Table E.26: Average of first and second order autocorrelation coefficient ordering results after mapping with the internal method.

ordering method	**improvement over Item**		**improvement over Random**	
	% for bdd nodes	% for CLBs	% for bdd nodes	% for CLBs
1st-order	11.15	11.78	23.54	27.05
truncated avg.	10.20	10.82	22.82	26.40
2nd-order v1	-58.09	-154.26	5.51	5.58
truncated avg.	3.26	-0.38	14.51	15.17
2nd-order v2	11.89	13.10	24.24	28.09
truncated avg.	10.98	12.16	23.56	26.99
avg-order	10.71	10.51	23.00	25.61
truncated avg.	9.79	9.51	22.25	24.88

Table E.27: Summary of results using internal.

Multiple Output Functions

A small number of multiple output functions have also been tested. These results are shown in the following tables, Table E.28, Table E.29, Table E.30, Table E.31, and Table E.32.

function	**Method 1**	**Method 3**	**Item**
	(seconds)	(seconds)	(seconds)
hline 5xp1.pla	0.0	0.0	1.5
9sym.pla	0.0	0.0	5.8
Z5xp1.pla	0.0	0.0	10.8
alu4.pla	11.7	12.3	217.7
apex4.pla	0.1	0.1	82.5
b12.pla	68.6	70.2	
clip.pla	0.0	0.0	8.0
con1.pla	0.0	0.0	0.1
ex1010.pla	0.3	0.5	89.0
ex5.pla	0.1	0.1	280.3
inc.pla	0.0	0.0	1.9
misex1.pla	0.0	0.0	0.4
misex3.pla	8.5	9.2	257.8
rd53.pla	0.0	0.0	0.7
rd73.pla	0.0	0.0	4.6
rd84.pla	0.0	0.0	15.0
sao2.pla	0.0	0.0	3.9
squar5.pla	0.0	0.0	1.0
table3.pla	1.7	2.0	35.0
table5.pla	17.2	17.8	35.0
xor5.pla	0.0	0.0	0.3

Table E.28: Timing results for 22 multiple output functions. The mapper xmap was used for these tests.

function	Method 2a)	Method 2b)	Item
	(seconds)	(seconds)	(seconds)
hline 5xp1.pla	0.0	0.0	1.5
9sym.pla	0.0	0.0	5.8
Z5xp1.pla	0.0	0.0	10.8
alu4.pla	6.3	12.3	217.7
apex4.pla	0.1	0.1	82.5
b12.pla	40.0	70.2	
clip.pla	0.0	0.0	8.0
con1.pla	0.0	0.0	0.1
ex1010.pla	0.2	0.5	89.0
ex5.pla	0.0	0.1	280.3
inc.pla	0.0	0.0	1.9
misex1.pla	0.0	0.0	0.4
misex3.pla	4.6	9.2	257.8
rd53.pla			
rd73.pla			
rd84.pla			
sao2.pla			
squar5.pla			
table3.pla			
table5.pla			
xor5.pla			

Table E.29: The remaining timing results for 22 multiple output functions. The mapper xmap was used for these tests.

function	Method 1	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
5xp1.pla	(87, 27 ,23)	(43, 18 ,13)	(81, 30 ,22)
9sym.pla	(26, 8 ,8)	(26, 8 ,8)	(26, 8 ,8)
Z5xp1.pla	(87, 27 ,23)	(43, 18 ,13)	(70, 25 ,19)
alu4.pla	(996, 517 ,407)	(610, 260 ,217)	(860, 435 ,355)
apex4.pla	(923, 536 ,399)	(930, 532 ,399)	(994, 591 ,449)
bw.pla	(127, 37 ,36)	(129, 37 ,36)	(136, 37 ,36)
clip.pla	(96, 45 ,38)	(78, 33 ,30)	(180, 110 ,91)
con1.pla	(19, 4 ,4)	(18, 3 ,3)	(25, 4 ,4)
ex1010.pla	(1834, 992 ,756)	(1796, 988 ,749)	(1834, 994 ,757)
ex5.pla	(283, 139 ,100)	(268, 133 ,96)	(369, 196 ,136)
inc.pla	(77, 29 ,22)	(80, 29 ,22)	(96, 36 ,28)
misex1.pla	(42, 17 ,12)	(42, 17 ,12)	(48, 19 ,14)
misex3.pla	(1704, 878 ,658)	(638, 283 ,226)	(2899, 1653 ,1295)
rd53.pla	(18, 3 ,3)	(18, 3 ,3)	(18, 3 ,3)
rd73.pla	(32, 9 ,8)	(32, 9 ,8)	(32, 9 ,8)
rd84.pla	(47, 16 ,15)	(47, 16 ,15)	(47, 16 ,15)
sao2.pla	(94, 35 ,29)	(90, 36 ,31)	(143, 54 ,46)
squar5.pla	(43, 12 ,10)	(42, 10 ,9)	(48, 10 ,9)
t481.pla	(39, 14 ,12)	(76, 21 ,21)	(274, 143 ,97)
table3.pla	(889, 439 ,346)	(897, 467 ,367)	(1285, 642 ,505)
table5.pla	(905, 439 ,360)	(769, 371 ,305)	(2286, 1239 ,1041)
xor5.pla	(7, 1 ,1)	(7, 1 ,1)	(7, 1 ,1)

Table E.30: Results on 22 multiple output functions after using Method 1 to generate orderings and then mapping the function using xmap.

function	Method 2b)	Method 2a)	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
5xp1.pla	(76, 28 ,23)	(43, 18 ,13)	(43, 18 ,13)	(81, 30 ,22)
9sym.pla	(26, 8 ,8)	(26, 8 ,8)	(26, 8 ,8)	(26, 8 ,8)
Z5xp1.pla	(76, 28 ,23)	(43, 18 ,13)	(43, 18 ,13)	(70, 25 ,19)
alu4.pla	(605, 267 ,211)	(1135, 619 ,492)	(610, 260 ,217)	(860, 435 ,355)
apex4.pla	(1000, 582 ,445)	(1062, 652 ,491)	(930, 532 ,399)	(994, 591 ,449)
clip.pla	(104, 41 ,39)	(118, 44 ,37)	(78, 33 ,30)	(180, 110 ,91)
con1.pla	(17, 3 ,3)	(20, 4 ,4)	(18, 3 ,3)	(25, 4 ,4)
ex1010.pla	(1819, 990 ,754)	(1812, 1006 ,760)	(1796, 988 ,749)	(1834, 994 ,757)
ex5.pla	(264, 133 ,95)	(348, 170 ,123)	(268, 133 ,96)	(369, 196 ,136)
inc.pla	(80, 29 ,22)	(113, 43 ,34)	(80, 29 ,22)	(96, 36 ,28)
misex1.pla	(42, 17 ,12)	(72, 30 ,23)	(42, 17 ,12)	(48, 19 ,14)
misex3.pla	(1495, 798 ,606)	(1040, 578 ,456)	(638, 283 ,226)	(2899, 1653 ,1295)
rd53.pla	(18, 3 ,3)	(18, 3 ,3)	(18, 3 ,3)	(18, 3 ,3)
rd73.pla	(32, 9 ,8)	(32, 9 ,8)	(32, 9 ,8)	(32, 9 ,8)
rd84.pla	(47, 16 ,15)	(47, 16 ,15)	(47, 16 ,15)	(47, 16 ,15)
sao2.pla	(93, 40 ,33)	(109, 52 ,40)	(90, 36 ,31)	(143, 54 ,46)
squar5.pla	(42, 10 ,9)	(44, 10 ,9)	(42, 10 ,9)	(48, 10 ,9)
t481.pla	(22, 6 ,6)	(41, 17 ,15)	(76, 21 ,21)	(274, 143 ,97)
table3.pla	(931, 465 ,364)	(1910, 1101 ,893)	(897, 467 ,367)	(1285, 642 ,505)
table5.pla	(782, 369 ,295)	(3851, 2178 ,1798)	(769, 371 ,305)	(2286, 1239 ,1041)
xor5.pla	(7, 1 ,1)	(7, 1 ,1)	(7, 1 ,1)	(7, 1 ,1)

Table E.31: Results on 22 multiple output functions after using Methods 2a) and 2b) to generate orderings and then mapping the function using xmap.

function	Method 3	Item	random orderings
	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)	(bdd nodes, luts, clbs)
5xp1.pla	(84, 26 ,22)	(43, 18 ,13)	(81, 30 ,22)
9sym.pla	(26, 8 ,8)	(26, 8 ,8)	(26, 8 ,8)
Z5xp1.pla	(85, 27 ,22)	(43, 18 ,13)	(70, 25 ,19)
alu4.pla	(605, 266 ,211)	(610, 260 ,217)	(860, 435 ,355)
apex4.pla	(924, 519 ,393)	(930, 532 ,399)	(994, 591 ,449)
bw.pla	(115, 28 ,27)	(129, 37 ,36)	(136, 37 ,36)
clip.pla	(132, 65 ,50)	(78, 33 ,30)	(180, 110 ,91)
con1.pla	(22, 3 ,3)	(18, 3 ,3)	(25, 4 ,4)
ex1010.pla	(1339, 803 ,603)	(1796, 988 ,749)	(1834, 994 ,757)
ex5.pla	(265, 130 ,90)	(268, 133 ,96)	(369, 196 ,136)
inc.pla	(76, 30 ,23)	(80, 29 ,22)	(96, 36 ,28)
misex1.pla	(42, 17 ,12)	(42, 17 ,12)	(48, 19 ,14)
misex3.pla	(1835, 1000 ,773)	(638, 283 ,226)	(2899, 1653 ,1295)
rd53.pla	(18, 3 ,3)	(18, 3 ,3)	(18, 3 ,3)
rd73.pla	(32, 9 ,8)	(32, 9 ,8)	(32, 9 ,8)
rd84.pla	(43, 14 ,13)	(47, 16 ,15)	(47, 16 ,15)
sao2.pla	(85, 36 ,28)	(90, 36 ,31)	(143, 54 ,46)
squar5.pla	(40, 8 ,7)	(42, 10 ,9)	(48, 10 ,9)
t481.pla	(53, 22 ,20)	(76, 21 ,21)	(274, 143 ,97)
table3.pla	(896, 437 ,353)	(897, 467 ,367)	(1285, 642 ,505)
table5.pla	(1153, 539 ,438)	(769, 371 ,305)	(2286, 1239 ,1041)
xor5.pla	(7, 1 ,1)	(7, 1 ,1)	(7, 1 ,1)

Table E.32: Results on 22 multiple output functions after using Method 3 to generate orderings and then mapping the function using xmap.

Randomly Generated Functions

Tables E.33 and E.34 show the times taken for each ordering method when run on 10 randomly generated functions. The times for generating the random orderings are not shown since they are all 0.

function	**Method 1**	**Method 3**	**Item**
	seconds	seconds	seconds
1.pla	0.0	0.0	6.8
10.pla	0.0	0.0	2.1
2.pla	0.0	0.0	6.4
3.pla	0.0	0.0	3.0
4.pla	0.0	0.0	2.2
5.pla	0.0	0.0	7.2
6.pla	0.0	0.0	8.9
7.pla	0.0	0.0	1.9
8.pla	0.0	0.0	5.5
9.pla	0.0	0.0	6.6

Table E.33: Timing results for 10 randomly generated functions using Methods 1 and 3 to generate orderings.

function	**Method 2a)**	**Method 2b)**	**Item**
	seconds	seconds	seconds
1.pla	0.0	0.0	6.8
10.pla	0.0	0.0	2.1
2.pla	0.0	0.0	6.4
3.pla	0.0	0.0	3.0
4.pla	0.0	0.0	2.2
5.pla	0.0	0.0	7.2
6.pla	0.0	0.0	8.9
7.pla	0.0	0.0	1.9
8.pla	0.0	0.0	5.5
9.pla	0.0	0.0	6.6

Table E.34: Timing results for 10 randomly generated functions using Methods 2a and 2b to generate orderings.

Bibliography

[Ake78] S. Akers. Binary Decision Diagrams. *IEEE Transaction on Computers*, vol. C-27(no. 6), June 1978.

[B+86] R. Brayton et al. Multiple-Level Logic Optimization System. In *IEEE International Conference on Computer Aided Design*, pages 356–359, 1986.

[BFRV92] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[BHSV90] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of 27th ACM IEEE Design Automation Conference*, 1990.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, vol. C-35(no. 8):677–691, August 1986.

[BSPR93] H. Bouzouzou, G. Saucier, F. Poiroit, and R. Roane. Use of Binary Decision Diagrams for Logic Synthesis. In *International Workshop on*

Logic Synthesis 1993 Workshop Notes, pages P3b–1 – P3b–6, 1993.

[CD94] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer Aided Design*, vol. 13(no. 1):1–11, January 1994.

[F$^+$93] M. Fujita et al. Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation. *IEEE Transactions on Computer Aided Design*, vol. 12(no. 1):6–12, January 1993.

[FRC90] R. Francis, J. Rose, and K. Chung. Chortle: a Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays. In *Proceedings of the 27th ACM IEEE Design Automation Conference*, pages 613–619, 1990. Paper 37.3.

[FRV91] R. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 512–519, June 1991. Paper 15.1.

[GGR$^+$89] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. El-ayat, and A. Mohsen. An Architecture for Electrically Reconfigurable Gate Arrays. *IEEE Journal of Solid State Circuits*, vol. 24(no. 2):394–398, April 1989.

[HMM85] S. L. Hurst, D. M. Miller, and J. C. Muzio. *Spectral Techniques in Digital Logic*. Academic Press, Inc., Orlando, Florida, 1985.

[Hur78] S. Hurst. *The Logical Processing of Digital Signals*. Crane Russak, 1978.

[Kar76] M. Karpovsky. *Finite Orthogonal Series in the Design of Digital Devices.* John Wiley and Son, 1976.

[Kar88a] K. Karplus. Representing Boolean Functions with If-Then-Else DAGs. Technical Report UCSC-CRL-88-28, Universtiy of California Santa Cruz, November 1988. ftp'd from ftp.cse.ucsc.edu, 22 pages.

[Kar88b] K. Karplus. Using if-the-else DAGs for Multi-Level Logic Minimization. Technical Report UCSC-CRL-88-29, University of California Santa Cruz, November 1988. ftp'd from ftp.cse.ucsc.edu, 20 pages.

[Kar89] K. Karplus. ITEM: an If-Then-Else Minimizer for Logic Synthesis. In *Proceedings of Euroasic 1989*, 1989. ftp'd from ftp.cse.ucsc.edu.

[Kar90] K. Karplus. Using if-then-else DAGs to do Technology Mapping for Field-Programmable Gate Arrays. Technical report, Universtiy of California Santa Cruz, 1990. ftp'ed from ftp.cse.ucsc.edu.

[Kar91] K. Karplus. Xmap: a Technology Mapper for Table-Lookup Field-Programmable Gate Arrays. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 240–243, June 1991.

[Mil93] D. M. Miller. Multi-Level Synthesis and Technology Mapping for FPGA's. In *Proceedings of the First Canadian Workshop on Field Programmable Gate Arrays*, pages 1–10, June 1993.

[MM93] F. Mailhot and G. De Micheli. Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations. *IEEE Transactions on Computer Aided Design*, vol. 12(no. 3):599–620, May 1993.

[MNS+90] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Synthesis for Programmable Gate Arrays. In *Proceedings of the 27th ACM IEEE Design Automation Conference*, 1990.

[MSBSV91] R. Murgai, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli. Improved Logic Synthesis Algorithms for Table Look Up Architectures. In *IEEE Conference on Computer Aided Design Digest of Technical Papers*, pages 564–567, 1991.

[Pat90] W. Patterson. Field Programmable Gate Arrays Get Enough Speed and Density for Computer Applications. In *COMPCON Spring 1990 35th IEEE International Conference*, pages 477–480, March 1990.

[SKC94] M. Schlag, J. Kong, and P. Chan. Routability-Driven Technology Mapping for Lookup Table-Based FPGA's. *IEEE Transactions on Computer Aided Design*, vol. 13(no. 1):13–26, January 1994.

[SVGR93] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose. Synthesis Methods for Field Programmable Gate Arrays. *Proceedings of the IEEE*, vol. 81(no. 7):1057–1083, July 1993.

[Tomed] Randal Tomczuk. *Ph. D. Thesis*. PhD thesis, University of Victoria, still to be completed.

[Woo91] N. Woo. A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 248–251, June 1991. Paper 15.5.

[Yan91] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. Microelectronics Center of North Carolina, North Carolina, USA, 1991.