IMPLEMENTATION OF SWITCHING CIRCUIT MODELS
AS VECTOR SPACE TRANSFORMATIONS

Approved by:

Dr. Mitchell A. Thornton
Dissertation Committee Chairman

Dr. Jennifer Dworak

Dr. Ping Gui

Dr. Theodore Manikas

Dr. Sukumaran Nair

IMPLEMENTATION OF SWITCHING CIRCUIT MODELS AS VECTOR SPACE TRANSFORMATIONS

A Dissertation Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Engineering

by

David Kebo Houngninou

M.S., Comp. Engineering, Washington University in St. Louis
B.S., Comp. Engineering, University of Evansville

December 16, 2017

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor Dr. Mitchell Thornton for his continuous support during my Ph.D. study and my research, for his patience, motivation, and great knowledge. His guidance helped me throughout conducting my research and writing this thesis. I am grateful for his mentorship and the opportunity to develop my teaching skills over the past few years.

Besides my advisor, I would like to thank the rest of my thesis committee, Dr. Sukumaran Nair, Dr. Jennifer Dworak, Dr. Ping Gui, and Dr. Theodore Manikas, for their insightful comments during my proposal which led me to explore further in my research from various perspectives.

I would like to thank my wife Sheila Sagbo for her words of encouragement and her continuous support over all these years.

Last but not least, I would like to thank my parents Denise and Etienne Houngninou for the words of wisdom and for supporting me morally and spiritually throughout writing this thesis and my life in general.

Houngninou, David Kebo M.S., Comp. Engineering, Washington University in St. Louis
B.S., Comp. Engineering, University of Evansville

Implementation of Switching Circuit Models
as Vector Space Transformations

Advisor: Dr. Mitchell A. Thornton
Doctor of Philosophy degree conferred December 16, 2017
Dissertation completed December 11, 2017

Modeling of switching circuits is the foundation for many Electronic Design Automation (EDA) tasks and is commonly used at various phases of the design flow for tasks such as simulation, justification, and other analyses. State-of-the-art simulation tools are based on discrete event algorithms using switching algebraic models and are highly optimized and mature. Symbolic simulation may also be implemented using a discrete event approach, or other approaches based on extracted functional models. The common foundation of modern simulation tools is that of a switching or Boolean algebraic model that may be augmented with timing information. Justification using switching circuit models are often based on solving the satisfiability problem and can be computationally expensive. Alternative models, such as the one proposed here have the potential to allow for advances in performance and storage requirements in applications such as simulation and justification.

Recently, an alternative foundational model for conventional digital electronic circuits has been proposed where the circuits are modeled as transfer functions in the form of matrices. The essence of the new model is to represent information as an element in a vector space rather than as a switching function variable. In this way, switching circuits are likewise modeled as transformations from one vector space to another. We demonstrate that the vector space model can be effectively used as the basis for symbolic simulation, justification, and other applications.

A central issue in using the vector space model is that representations and manipulations

of the models must not incur complexity any worse than that of algorithms based upon traditional switching algebraic approaches. In particular we show that Algebraic Decision Diagrams (ADDs) can be used to represent vector space models thus allowing the advantages of the vector space approach to be realized while also ensuring the complexity of the underlying algorithms are no worse than that of conventional switching algebraic models. Spatial complexity is significantly reduced through the use of ADDs to represent the transfer functions as compared to explicit representations and they serve to illustrate the viability of the linear algebraic model in EDA applications.

A transfer function is a mathematical function relating the output or response of a system to the input or stimulus. It is a concise mathematical model representing the input/output behavior of a system, and it is widely used in many areas of engineering including system theory and signal analysis. We implement a framework to build transfer function models of digital switching functions using ADDs and demonstrate their application to simulation, justification, and the computation of the algebraic normal form (ANF).

Cryptographic primitives may be composed of collections of switching functions. The Algebraic Normal Form (ANF) of a cryptographic switching function is of general interest since this form allows for the computation of many characteristics of interest to the cryptography community. One interesting property of the ANF is that it allows for direct observation of the algebraic degree of a switching function. We present a technique to determine the ANF of switching functions through the traversal of a structural netlist with complexity $O(N)$.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

This is dedicated to:

Chapter 1

INTRODUCTION

Switching theory provides a rich theoretical basis for modeling digital logic circuits. Traditionally, digital logic circuits are modeled using the axioms and postulates of switching theory formulated in terms of a binary-valued Boolean algebra over discrete scalar-valued switching functions. The switching theory framework has led to an extensive set of analysis and synthesis methods that continue to be commonly used in all facets of modern digital circuit design activities. Using this new approach, we reformulate these mathematical models in terms of linear transforms over vector spaces. Transfer functions describe the input-output behavior of a system. We can obtain the system response with respect to a particular input stimulus through a multiplicative operation among the stimulus and transfer function. In our model, we use the transfer function model to represent a switching circuit and show how the transfer function can be formulated based upon the topology of a structural representation of the switching circuit as well as other representations. This technique is described in further details in Chapter 4. The inverse transfer function can be used to determine a corresponding input stimulus given an output response through a multiplicative operation. In terms of digital logic network operations, these tasks are commonly referred to as simulation and justification respectively.

1.1. The Study Contributions

In Chapter 2, we provide background information on linear algebra and the vector space representation method. The conventional switching theory framework has led to an extensive set of synthesis methods that are commonly used in modern digital circuit design. In our approach, we reformulate these mathematical models in terms of linear transforms over vector spaces. The first prototype used to compute the transfer function was implemented using

sparse matrices. To obtain the output response, we can multiply the input stimulus by the transfer matrix. The fact that matrices can grow exponentially in size makes them unsuitable for representing large functions with multiple variables. We show that it is more efficient to represent transfer functions using Algebraic Decision Diagrams due to their compactness and canonicity. We also explain how this new theory can be used as a part of an EDA tool for representing and manipulating switching functions as transfer functions. The experimental results validate our hypothesis by showing the timing and memory improvements achieved by using decision diagrams.

Previous work [27] described a new theory for representing switching functions with linear algebraic transfer functions. We have described the application of this theory to common operations such as simulation and justification. Chapter 4 provides the detailed steps involved in building the transfer function starting with a structural netlist. The theory is equally applicable to combinational circuits and sequential circuits because the structure of the underlying transfer functions remains the same.

Chapter 3 describes the use of our model to build a prototype simulation tool. We describe how the simulator is implemented including relevant matrix-based models and ADD-based algorithms that are employed to perform the computations. An important contribution of this work is the creation of graph-based algorithms to efficiently implement the required linear algebraic operations for tasks such as simulation, justification, and ANF computation. The system response with respect to a particular input stimulus is unique and can be obtained through a multiplicative operation of the stimulus and the transfer function. In Chapter 3, we describe three different approaches for the computation of simulation responses. Following the description of each implementation, we compare their performances in terms of computation time and storage requirements.

Justification is the inverse problem of simulation. Knowing the output response and the characterization of a logic network, we develop ADD-based methods to compute the corresponding input stimuli. In Chapter 5, we demonstrate that the same transfer function can be reused to perform justification using a single multiplication. Following the description of our

justification method and its implementation, we report performance in terms of computation time and storage requirements.

Cryptographic primitives serve as the building blocks of larger cryptographic systems. It is common to represent or model a cryptographic primitive of n inputs and m outputs as a collection of r Boolean or switching functions. We are interested in computing the Algebraic Normal Form (ANF) of a cryptographic switching function. This expression allows us to extract the algebraic degree of a switching function in linear time. In cryptography, knowing the degree of a switching function can aid in various cryptanalysis tasks and is a valuable piece of information. However, computing the degree of a switching function using the common switching algebraic model is well-known to require extensive computational resources. Another contribution of this research is the provision of a technique that recovers the ANF of switching functions through traversals of a structural netlist based upon use of the vector space model. In Chapter 6, we provide background information, a definition of the ANF, and a new method for its computation using the vector space model. In particular, we show how a single ANF coefficient can be extracted through a single traversal of a structural netlist.

Chapter 2

BACKGROUND

2.1. Matrices, BDDs, and Related Operators

Switching theory is based on the mathematics of Boole as originally devised for symbolic logic manipulation. We develop an alternative to the traditional switching theory model for digital network representation and manipulation, using matrices as transfer function. Binary Decision Diagrams (BDDs) are data structures that are widely used in the CAD industry. In 1980, R. E. Bryant [4] demonstrated that we can use a BDD as a canonical representation of a Boolean function. He also demonstrated how to perform binary Boolean operations on two BDDs. Previous research has also shown that binary decision diagrams are an efficient data structure to represent almost any common Boolean function. They are usually smaller than any other representation. As an example, for the $n \times n$ Walsh matrix, the BDD representation is of size complexity $O(n \log n)$ [12]. Later in 1988, Malik et al. [19] proposed a faster way to carry out formal verification for a larger set of combinational networks compared to existing verification systems. The authors proposed new variable ordering techniques that are based on the topology of the multi-level network to improve the speed of testing. In 1992, further experiments revealed a relationship between binary decision diagrams and matrices. Any BDD can be represented as a vector of length 2^n or as a sparse matrix of size $2^{n-1} \times 2^{n-1}$. BDDs are folded representations of the Shannon Cofactor Tree. Supposing a Boolean function F of n variables $x_1, x_2, \ldots, x_n$:

$$F : \{0,1\}^n \rightarrow \{0,1\}$$

We define new Boolean functions of $n-1$ variables as follows:

$$F_{x1}(x_1, x_2, \ldots, x_n) = F(1, x_2, x_3, \ldots, x_n)$$

$$F_{x1'}(x_1, x_2, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n)$$

F_{xi} and $F_{xi'}$ are the cofactors of F, and the Shannon Expansion is written as:

$$F(x_1, x_2, \ldots, x_n) = x_i.Fx_i + x_{i'}.Fx_{i'} \tag{2.1}$$

The cofactors F_{x_i} and $F_{x_i'}$ can be represented as a tree which is also a BDD. Each internal node in the BDD is a subfunction. Each internal node has two children; the left child is the cofactor with respect to x_i', and the right child is the cofactor with respect to x_i. The overall function is a larger but compressed tree, because all isomorphic subtrees are folded together into a single structure. The compression means that we merge internal nodes that represent the same function into a single node. We can also write cofactors of a function as a sparse matrix filled with binary numbers. Identical functions will map to the same matrix. A Multi-Terminal Binary Decision Diagram (MTBDD) [12] is a variant of BDDs that has arbitrary integer values at the leaf nodes instead of two leaves of 0 and 1. MTBDDs represent functions from a Boolean space $\mathbb{B}^n$ onto a finite set $R\widetilde{R}$. For a vector v of size m, v is a function from the Boolean space $\mathbb{B}^{\log n}$ onto the range of the vector, and can be represented as a MTBDD. This background work shows that we can represent vectors as BDDs and matrices as MTBDDs. The size of the MTBDD and its matrix are correlated. Each path in the tree traverses $\log n$ nodes, where n is the number of rows in the matrix and the space complexity is $O(n)$.

We also investigated the type of operations over these data structures such as addition, inner product, outer product, scalar multiplication, and composition. The work of Clarke and Fujita [12] covered in details a set of operations such as vector multiplication, multiplication of a vector by a vector, and multiplication of a matrix by a vector, multiplication of a matrix by a matrix. [12] uses a procedure called *Apply* which takes as input two BDDs and

an operator. Randal E. Bryant [4] developed the procedure *Apply*. It provides the basic method to perform operations on two Boolean functions. *Apply* is fully implemented in CUDD, a package for BDD manipulation written in c that we use later for experimental results in our research. Supposing two Boolean functions f_1, f_2 and a binary operator $\langle op \rangle$, we define the function $f_1 \langle op \rangle f_2$ as:

$$[f_1 \langle op \rangle f_2](x_1, \ldots, x_n) = f_2(x_1, \ldots, x_n) \langle op \rangle f_2(x_1, \ldots, x_n)$$

The work of Randal E. Bryant [4] in 1986 introduced more algorithms to perform basic operations on Boolean functions represented as BDDs; we summarized these operations in Table 1. These algorithms use graph algorithms techniques such as ordered traversal, table lookup, and vertex encoding. The time complexity of these algorithms closely depends on the size of the graphs.

Procedure	**Result**	**Time Complexity**
Reduce	G reduced by canonical form	$O(\|G\| \cdot \log(\|G\|)$
Apply	$f_1 \langle op \rangle f_2$	$O(\|G_1\| \cdot \|G_2\|)$
Restrict	$f\|_{x_i=b}$	$O(\|G\| \cdot \log\|G\|)$
Compose	$f1\|_{x_i=f2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Satisfy-one	some element of S_f	$O(n)$
Satisfy-all	S_f	$O(n \cdot \|S_f\|)$
Satisfy-count	$\|S_f\|$	$O(\|G\|)$

Table 2.1. Algorithms for basic operations on BDDs

This background work is essential for our research because we reuse some of these operations to compute the output response of a circuit after building a transfer function. Supposing a vector g and a matrix f, the multiplication of f and g is expressed as:

$$h(x_1, \ldots, x_m) = f(x_1, y_1, \ldots, x_m, y_n) \circ g(y_1, \ldots, y_n)$$

The matrix f performs a transformation of the vector g which represents our input vector and the result becomes a new vector h which represents the output vector. The vector by matrix multiplication looks like this:

$$\begin{pmatrix} h_{x1'} \\ \\ h_{x1} \end{pmatrix} = \begin{pmatrix} f_{x1'y1'} & f_{x1'y1} \\ \\ f_{x1y1'} & f_{x1y1} \end{pmatrix} \begin{pmatrix} g_{y1'} \\ \\ g_{y1} \end{pmatrix}$$

Each element of the vector becomes:

$$h_{x1'} = f_{x1'y1'} \circ g_{y1'} + f_{x1'y1} \circ g_{y1}$$

$$h_{y1} = f_{x1y1'} \circ g_{y1'} + f_{x1y1} \circ g_{y1}$$

2.2. Variable Reordering Methods

A binary decision diagram is ordered if each variable is encountered at most once on each path from the root to a leaf. We consider a decision diagram to be fully-reduced if it does not contain duplicate nodes for a given level. The nodes at each level of the diagram represent a variable. Variable reordering is essential especially for large directed acyclic graphs because it helps reduce the number of nodes. Finding the optimal variable ordering is an NP-complete problem [3]. There are two types of techniques for variable reordering: static variable ordering and dynamic variable ordering. Static variable ordering determines the order before constructing the BDD when dynamic variable ordering reorders when building the BDD. Static variable ordering is faster but not as efficient as dynamic reordering. Dynamic reordering can be more time consuming because it is done at runtime but is very handy when it comes to size optimization.

Static variable reordering: In static reordering, algorithms search for the best order by extracting topological data from the graph. We surveyed four main groups of static reordering: Graph search algorithms, graph evaluation algorithms, decomposition algorithms, and sample-based algorithms. In the graph search algorithm proposed by [19], we start by assigning a level of zero to vertices with no edges and perform a breadth-first search to assign an order to the other vertices. [11] also proposed some methods using depth-first and breadth-first traversal from the outputs of the circuit to its inputs. Some of these algorithms have limitations. For instance, the graph search algorithm proposed above was designed for BDDs representing a single Boolean function. This approach would not be beneficial for model checking nowadays, since we deal with large size BDDs. The research in [10] proposed some reordering methods such as variable appending and variable interleaving. For the variable appending method, we start by keeping track of a predefined priority order. The outputs are reordered based on the order of the inputs. The variable interleaving method interleaves the primary input variables that occur in multiple primary output lists. The algorithm starts with the primary outputs in some predefined priority order. For each primary output, we map a primary input that changes the value of that primary output and order them in a list.

Dynamic variable reordering: For n variables there are $n!$ different orders Dynamic reordering techniques iteratively improve variable orders. If the size of the BDD exceeds a threshold at runtime, we interrupt the operation, and we perform reordering. A popular method of dynamic reordering uses the sifting minimization algorithm. This algorithm finds the best position for a variable, assuming that the other variables are fixed. For n variables in the BDD, there are n possible positions for a variable. The purpose of the algorithm is to find the best position to reduce the size of the BDD. We gradually sift a variable down or up in the tree structure. Sifting means that we swap a variable with its successor or predecessor variable until we find the best size for the BDD [23]. Sifting has a complexity of $O(n^2)$. Other dynamic variable ordering methods were also implemented and are widely used.

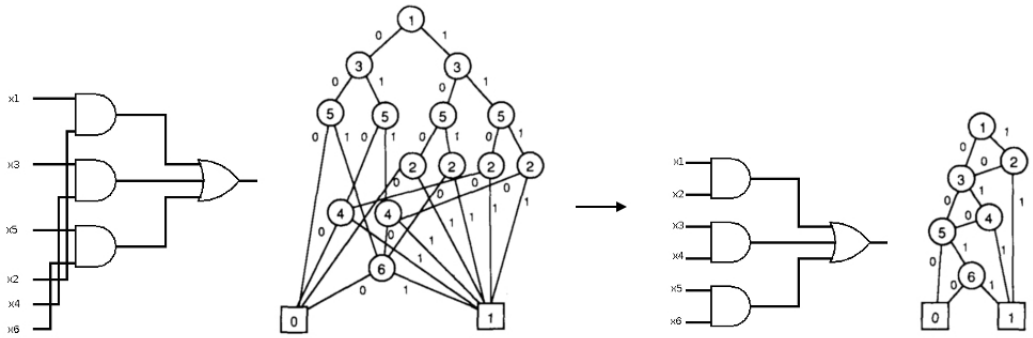

Figure 2.1. BDD variable reordering

[25] discussed a model for building BDDs. It starts by building a decision diagram for all the input variables, then builds the BDD for the output of the gate fed by the primary inputs. The resulting outputs are used to compute the BDDs of the next gates and so on until we reach the outputs. Moving forward in this process, we compute the BDD of multiple intermediate subfunctions. The technique developed in [25] relies on garbage collection for memory optimization. It keeps a reference count of each internal node and each terminal node. Since multiple functions can share the same subgraph, we increment the reference count any time a new arc points to the node. In the same way, any time a node is freed, we decrement the reference count. Garbage collection becomes useful when we need to remove all unnecessary nodes with a reference count of 0. Later in our work, we propose another method to build BDDs of combinational circuits using segments of the circuit called partitions.

We surveyed and analyzed previous work about key topics such as formal logic verification, canonical representation of Boolean functions, operations on Boolean functions, variable reordering, and memory optimization. This background provides a solid set of references for implementing our new method to build a transfer function model that uses binary decision diagrams as the underlying data structure.

2.3. Definitions and Mathematical Notations

Building a transfer function consists of multiple steps starting from parsing a structural netlist. The textual description of our benchmark circuits is in Verilog. The main steps involved in the parsing process are: fanout detection, netlist levelization, netlist partitioning, crossover detection, translating nets to sparse matrices or binary decision diagrams and building partitions representing intermediate functions. Before we describe each step of the process, we provide some definitions of the mathematical terms and notations used recurrently in this work.

2.3.1. The Vector Space

Linear algebra is defined over a vector space. A vector space consists of a set of k- dimensional vectors and the operations of scaling and addition. In other words, it is characterized by a dimension and a set of vectors. The scaling operation is a multiplicative operation with operands consisting of a scalar and a vector. The addition operation is performed over two operand vectors within the space. Both operations yield a resultant vector. Vectors are one-dimensional arrays of values or components, and the number of values comprising a vector defines the vector space dimension in which they are members. The vector space model offers a framework for binary networks, since they may be formulated using vectors to represent units of information. For this research, the vector spaces we are focusing on are finite-dimensioned Hilbert Spaces. For simulations using our transfer function model, we will use canonical basis vectors which are vectors whose components are all zero-valued except for a single-valued component.

2.3.2. The Hilbert Space

A Hilbert space is a particular vector space defined for an arbitrary dimension k, including an infinite dimension, and that has a norm and inner product associated with it. Column vectors that are members of a Hilbert space are denoted as $v \in \mathbb{H}^k$ and the corresponding row vector as v^T when the components of v are not complex-valued.

2.3.3. The Dirac Notation

We use the Dirac notation or "bra-ket" notation [9] to represent abstract vectors and linear forms in mathematics. Rather than using over arrows conventionally used in physics ($\vec{A}$), Dirac's notation for a vector uses vertical bars and angular brackets: $|A\rangle$ .We express a row vector v as $\langle v|$ referred to as "bra-v" and a column vector w as $|w\rangle$ referred to as "ket-w". The "bra-ket" notation is convenient since we express the inner product as $\langle v \,|\, w\rangle$ and the outer product as $|v\rangle \langle w|$. These expressions are useful in the formulation of transfer matrices that model digital logic networks. In this work we use a canonical vector space basis consisting of ket-0 ($|0\rangle$) and ket-1 ($|1\rangle$). The Bra-ket notation is convenient when using the vector space for simulation because the orientation of the bracket can indicate whether the vector multiplication is an inner product or an outer product.

$$|0\rangle = \begin{bmatrix} 1 \\ \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ \\ 1 \end{bmatrix}$$

2.3.4. The Inner Product

The inner product or dot product is a multiplication operation that produces a scalar product of two operand vectors of the same dimension. In a vector space, it is a way to multiply vectors together, with the result of this multiplication being a scalar.
The conventional notation for r-dimensional vectors x and y is:

$$x \cdot y = x^T \cdot y = \sum_{i=0}^{r-1} x_i \cdot y_i \qquad x^T = \begin{bmatrix} x_0 \; x_1 \ldots x_n \end{bmatrix} \qquad y^T = \begin{bmatrix} y_0 \; y_1 \ldots y_n \end{bmatrix}$$

The following expression represents the inner product using bra-ket notation:

$$\langle x \,|\, y\rangle = \sum_{i=0}^{r-1} x_i \cdot y_i \qquad \langle x| = x^T \qquad |y\rangle = y$$

A vector v may undergo a linear transformation that maps it to another space. Linear

transformations are expressed as vector-matrix multiplications where matrices are denoted with capital letters such as A. The inner product product $x^T \cdot y$ can be written as $\langle x \,|\, y \rangle$.

2.3.5. The Outer Product

The outer product is a multiplicative operation that can be used to multiply two tensors regardless of their order. It can be performed for two vectors, or tensors of order one, of any size. When we perform the outer product on a pair of vectors, we obtain a matrix or tensor of order two.

Using bra-ket notation, the following expressions denote an outer product:

$$\langle x \cdot y| = \langle x| \otimes \langle y| \qquad |x \cdot y\rangle = |x\rangle \otimes |y\rangle \qquad \langle x| \langle y| = |x\rangle \otimes \langle y|$$

The outer product is a non-commutative operation since $\langle x| \otimes \langle y| \neq \langle y| \otimes \langle x|$.
A comparison of conventional versus braket notation is shown in Table 2.2.

Table 2.2. Linear algebra and bra-ket notation

Operation	**Linear Algebra**	**Bra-Ket**
Inner product	$a \cdot b = b \cdot a$	$\langle a \,|\, b\rangle = \langle b \,|\, a\rangle$
Outer product	$a \otimes b$	$|a\rangle \langle b|$
Direct product	AB	AB
Outer product	$A \otimes B$	$|A\rangle \langle B|$
Vector/matrix product	$c = Ab$	$|c\rangle = A\,|b\rangle$
Vector/matrix product	$c^T = b^T A$	$\langle c| = \langle b|\, A$

Chapter 3

BUILDING THE TRANSFER FUNCTION MODEL

3.1. Parsing the Netlist

To compute a transfer function representation of a switching function into a matrix represented as an ADD, we first parse a structural netlist. In the work presented here, we used the Verilog language to represent structural netlists. The Verilog parser is a program that extracts information from structural multi-level combinational logic circuits written in Verilog. This parser is developed in *C* language. It tokenizes every line in a Verilog file, and invokes various callback methods.

The purpose of the parser is to read every line in the Verilog file and extract all the relevant information from the netlist. The parser identifies all the fanouts in the netlist and rewrites the netlist to include those fanouts. Next, the parser creates a *C*-language data structure with variables such as a unique ID, the number of inputs, the number of outputs, the number of gates, the number of partitions, and the number of crossovers. Statistical information collected from the Verilog netlist can be used to estimate the amount of memory required when building the transfer function. Each gate and each wire identified in the Verilog netlist is represented internally in the *C*-language data structure. A gate has attributes such as a unique ID, a name, the type of gate, the number of inputs, the number of outputs, and a matrix representation in the form of a small ADD representation.

The information that the parser returns includes:

- The module name
- The list of instantiations in the module
- The list of inputs

- The list of outputs
- The list of internal wires
- The list of logic gates

3.2. Fanout Detection

Typically, the output of a logic gate is connected to the input(s) of one or more logic gates. Whenever such an output drives two or more inputs of other gates, a structure known as a fanout is present in the netlist. Fanout points are treated as network elements since these structures have differing numbers of outputs. To obtain the correct transfer function, we must account for fanouts encountered in the circuit. The input Verilog netlists are in the form of a set of Boolean operators, and these netlists do not explicitly define fanouts as is the case in other netlist languages like ISCAS85. The first step of the process is to identify all the fanouts in the netlist and rewrite the netlist to include those fanouts. This identification is done by parsing all the Boolean operators in the netlist and grouping all the gates that have identical nets in their input port list. For every set of duplicate nets found, we create a new fanout node with a unique identifier. The outputs of the fanouts have the same values as the input. We assign each output wire of a fanout with a unique ID number.

3.3. Netlist Levelization

During event-driven simulation, gates are not always simulated in the order they are listed in a netlist. To simulate a circuit, we start by assigning binary values to the primary inputs and proceed by propagating those values until they reach the outputs. A single gate is not simulated until all of its input values are set. If the output of a gate named 'A' drives another gate named 'B', then the output of gate 'B' depends on the output of gate 'A'. To obtain the correct output value for gate 'B', the order of the simulation matters. In the case above, gate 'A' must be simulated before gate 'B'. The process of ordering and determining the proper gate arrangement for simulation is called levelization. We begin this process by

assigning all primary inputs with an initial level value of zero. During a netlist traversal from the primary inputs toward the outputs, the levelization process assigns a level number to each net that is encountered. This identical to the method used in many structural netlist simulation algorithms in conventional EDA tools. To levelize a structural circuit netlist, we apply three rules:

1. A net or a wire can be assigned a level number only if its driving gate has been assigned a level number.

2. A gate output net can be assigned a level number only if all its input nets have been assigned level numbers.

3. The level number of a gate output net is the maximum of all its inputs' level numbers incremented by one. For instance, if a gate output net 'D' has two input net values 'B' and 'C' with level numbers 4 and 7 respectively, then the level number of gate 'D' is 8, i.e., $max(4, 7) + 1$.

To accomplish levelization, we implemented a recursive approach by applying the three rules above starting from the primary inputs.

The purpose for performing levelization is to identify cuts in the netlist such that partitions are formed. Levelization values identify where such cuts occur in the netlist through grouping all nets that have identical levelization indices [27].

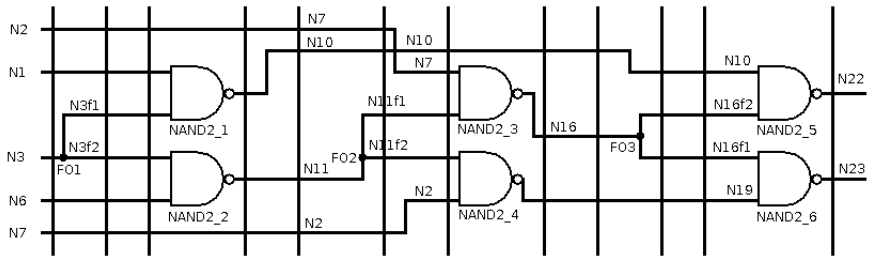

Figure 3.1. Schematic of benchmark circuit c17.v with partition cuts as vertical lines

3.4. Netlist Serial Partitioning

We group nets and gates with the same level numbers in parallel stages called partitions. Partitioning separates the network into series or partitions of subcircuits. A partition is made of the following types of elements: gates, fanins, fanouts and pass-through wires. All the primitive logic gates whose output nets have the same level numbers are identified and grouped in the same partition. Pass-through wires are wires that cross through a partition. Completing the serial partitioning process requires two or more passes through the netlist and is thus of temporal complexity $O(n)$ where n is the number of nets. The spatial complexity is also $O(n)$ as the structural Verilog netlist is parsed into an internal graph memory structure where nodes represent gates, primary inputs, and outputs. Graph edges correspond to the topological nets in the circuit.

3.5. Crossover Detection and Rows Permutations

Crossovers are the intersections of conducting wires in a structural netlist. We can represent multiple crossovers as a series of single crossovers. Levelization does not detect crossovers, so we need an additional step to account for crossovers before the calculation of the overall transfer function. The intermediate transfer functions in the form of permutation matrices for crossovers are injected in between existing partition stages. In the case where there is no crossover, the permutation matrix is an identity matrix.

3.5.1. Crossover Detection using Linear Equations

To identify crossovers between stages we use a set of linear equations (Figure 3.2). As an example, consider two serial partitions: an origin partition and a destination partition. All nets in the origin partition must have a mapping in the destination partition (from outputs of stage m to inputs of stage $m + 1$). First, we assign an order to every net in the origin partition, starting from the topmost element. Then, we assign an order to every net in the destination partition, starting from the topmost element. The orders are used as y-coordinates in a two-dimensional Cartesian coordinate map. Using these coordinates,

we compute a linear equation $y = ax + b$ using each pair of nets mapping from the origin to the destination partition. The equations are used to find the intersections of the lines. Each line intersection represents a crossover. By using the linear equation, we compute the coordinates of each intersection. All crossovers must be processed in the order they occur. Therefore, we use the x coordinates of each intersection point to sort the crossovers.

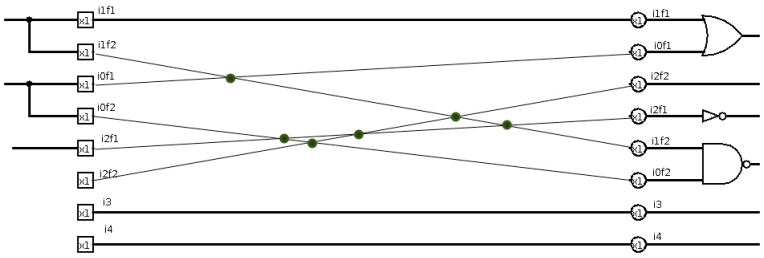

Figure 3.2. Crossover detection using linear equations

3.5.2. Computation of the Permutation Matrices

Once we detect crossovers, we need to construct the corresponding permutation matrices as their models. To obtain the correct transfer function, crossovers must be processed in the order in which they occur. Figure 3.3 shows an example arrangement of two crossings. We process multi-wire crossovers one at a time. Lines labeled 'I' represent wires, lines labeled 'C' represent crossovers, and lines labeled 'FO' represent fanouts. The transfer function for a wire is the identity matrix. The transfer function for a crossover is a predefined crossover matrix.

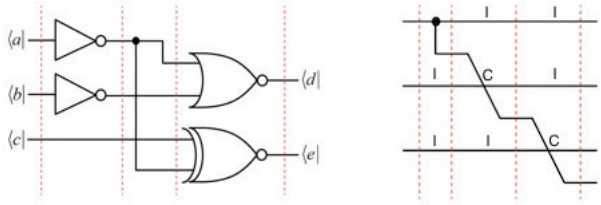

Figure 3.3. Computation of a crossover matrix

In Figure 3.3, we have two crossovers. To compute the permutation matrix T we use the following equation:

$$T = (FO \otimes I \otimes I) \cdot (I \otimes C \otimes I) \cdot (I \otimes I \otimes C)$$

3.6. Combining the Intermediate Partitions

The previous process of serial partitioning groups all nets and gates with the same level numbers into a set of parallel stages. After the partitions are formed, there is a need to compute their corresponding transfer function. To compute the transfer function for a partition, we compute the outer product of all the parallel network elements starting from the topmost element. Each partition transfer matrix requires p outer product operations where p is the number of parallel elements. To build the overall transfer function, we multiply all these partitions together in a particular order [27]. Starting from the leftmost partition, the first partition is multiplied by the next one to form an intermediate transfer function. The intermediate transfer function is then multiplied by the next partition in the serial sequence and so on. A transfer function requires m direct product operations where m is the number of partitions. When a crossover partition is encountered, it must also be inserted into the sequence at the appropriate point and multiplied as well. When calculating the overall monolithic transfer function, we also take into account memory management. At every iteration, once an intermediate function is calculated, we discard the previous partition to free up unused memory. This technique is especially beneficial when dealing with large netlists.

3.7. Building the Transfer Function using Sparse Matrices

A sparse matrix is a matrix in which many of the elements are zero valued. The characteristics of sparse matrices can be exploited to optimize the size and speed of computational resources when they are manipulated. Since there are fewer non-zero elements than zeros, less memory is required to store the data. Since our theory uses principles of linear algebra,

sparse matrices are an efficient data structure. In particular, using sparse matrices with a larger number of zeros saves a significant amount of memory and speeds up the processing of that data. Sparse matrices also have other significant advantages in terms of computational efficiency. There is no need to perform unnecessary low-level arithmetic since the set is limited to $\{0, 1\}$. This increase in efficiency allows for performance improvements when dealing with large netlists.

A transfer function matrix X is isomorphic to the truth table representation of that function [27]. We use this principle to build a library of sub-matrices. We refer these blocks as sub-matrices because we reuse them as building blocks to construct larger transfer matrices. The most commonly used gates are `AND, OR, XOR, BUF, NAND, NOR, XNOR, INV`. In addition to the traditional gates, we also account for fanouts and crossovers. Fanout points are treated as network elements since these structures have differing numbers of outputs. To represent a fanout as a matrix, we must first write a truth table, then perform the outer product of the row vectors on each output. Crossovers are the intersections of conducting wires. The permutation of the order of wires affects the correctness of the transfer function. If we ignore crossovers during the transformation of the input vector into an output vector, the network output response becomes incorrect because the order of the row vectors is interchanged. To fix this problem, we must include some intermediate matrices representing crossovers. These intermediate matrices are not intended to perform Boolean operations; their purpose is to rearrange the order of the row vectors between two cascades. Crossover transfer matrices are permutation matrices that are orthogonal rotations in the Hilbert space $\mathbb{H}^n$, or a vector space projection onto itself. To represent a crossover as a matrix, we must write a truth table first then perform the outer product of the row vectors on each output.

The transfer function T representing the input-output relationship of a logic network F is of the form in equation 3.6:

$$T = \sum_{i=1}^{2^n} |x_i\rangle \langle f_i| \tag{3.1}$$

To compute T, we must understand how to perform the conversion of a truth table to a matrix. For example, Table 3.1 below shows how to convert a truth table for functions f and g

x_1	x_2	f	g
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 3.1. Example of truth table for f and g

We can compute the transfer function for this truth table using Equation 3.2. We can derive T from the truth table using row vectors.

$$T = \begin{bmatrix} \langle 10| \\ \langle 10| \\ \langle 10| \\ \langle 01| \end{bmatrix} = \begin{bmatrix} \langle 1|\otimes\langle 0| \\ \langle 1|\otimes\langle 0| \\ \langle 1|\otimes\langle 0| \\ \langle 0|\otimes\langle 1| \end{bmatrix} = \begin{bmatrix} [0\ 1]\otimes[1\ 0] \\ [0\ 1]\otimes[1\ 0] \\ [0\ 1]\otimes[1\ 0] \\ [1\ 0]\otimes[0\ 1] \end{bmatrix} = \begin{bmatrix} [0\ 0\ 1\ 0] \\ [0\ 0\ 1\ 0] \\ [0\ 0\ 1\ 0] \\ [0\ 1\ 0\ 0] \end{bmatrix} \tag{3.2}$$

$$T = \sum_{i=0}^{3} |x_i\rangle \langle f_i| = |0\rangle \langle 2| + |1\rangle \langle 2| + |2\rangle \langle 2| + |3\rangle \langle 1| \tag{3.3}$$

$$T = \begin{bmatrix} 0\ 0\ 1\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 1\ 0\ 0 \end{bmatrix} \tag{3.4}$$

The tables below summarizes the operations performed to obtain the sub-matrices for each gate.

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 0\|$	[1 0]	[1 0]	[1 0]	$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 0\|$	[1 0]	[0 1]	[1 0]	
$\langle 1\|$	$\langle 0\|$	$\langle 0\|$	[0 1]	[1 0]	[1 0]	
$\langle 1\|$	$\langle 1\|$	$\langle 1\|$	[0 1]	[0 1]	[0 1]	

Table 3.2. `AND` truth table using elements of elements $\mathbb{H}$

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 0\|$	[1 0]	[1 0]	[1 0]	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 1\|$	[1 0]	[0 1]	[0 1]	
$\langle 1\|$	$\langle 0\|$	$\langle 1\|$	[0 1]	[1 0]	[0 1]	
$\langle 1\|$	$\langle 1\|$	$\langle 1\|$	[0 1]	[0 1]	[0 1]	

Table 3.3. `OR` truth table using elements of elements $\mathbb{H}$

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 0\|$	[1 0]	[1 0]	[1 0]	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 1\|$	[1 0]	[0 1]	[0 1]	
$\langle 1\|$	$\langle 0\|$	$\langle 1\|$	[0 1]	[1 0]	[0 1]	
$\langle 1\|$	$\langle 1\|$	$\langle 0\|$	[0 1]	[0 1]	[1 0]	

Table 3.4. `XOR` truth table using elements of elements $\mathbb{H}$

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 1\|$	[1 0]	[1 0]	[0 1]	$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 1\|$	[1 0]	[0 1]	[0 1]	
$\langle 1\|$	$\langle 0\|$	$\langle 1\|$	[0 1]	[1 0]	[0 1]	
$\langle 1\|$	$\langle 1\|$	$\langle 0\|$	[0 1]	[0 1]	[1 0]	

Table 3.5. `NAND` truth table using elements of elements $\mathbb{H}$

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 1\|$	[1 0]	[1 0]	[0 1]	$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 0\|$	[1 0]	[0 1]	[1 0]	
$\langle 1\|$	$\langle 0\|$	$\langle 0\|$	[0 1]	[1 0]	[1 0]	
$\langle 1\|$	$\langle 1\|$	$\langle 0\|$	[0 1]	[0 1]	[1 0]	

Table 3.6. `NOR` truth table using elements of elements $\mathbb{H}$

a	b	$a \otimes b$	a	b	$a \otimes b$	Matrix
$\langle 0\|$	$\langle 0\|$	$\langle 0\|$	[1 0]	[1 0]	[0 1]	$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$
$\langle 0\|$	$\langle 1\|$	$\langle 1\|$	[1 0]	[0 1]	[1 0]	
$\langle 1\|$	$\langle 0\|$	$\langle 1\|$	[0 1]	[1 0]	[1 0]	
$\langle 1\|$	$\langle 1\|$	$\langle 0\|$	[0 1]	[0 1]	[0 1]	

Table 3.7. `XNOR` truth table using elements of elements $\mathbb{H}$

a	a	Matrix
$\langle 0\|$	[1 0]	$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$
$\langle 1\|$	[0 1]	

Table 3.8. `BUFFER` truth table using elements of elements $\mathbb{H}$

a	$\neg a$	Matrix
$\langle 0\|$	$[0\ 1]$	$\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$
$\langle 1\|$	$[1\ 0]$	

Table 3.9. `NOT` truth table using elements of elements $\mathbb{H}$

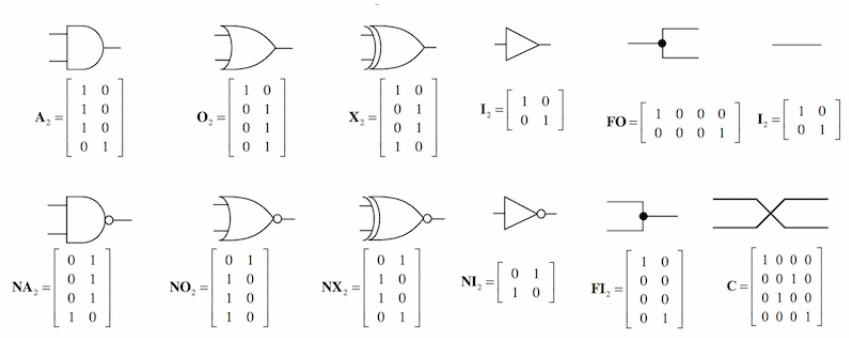

Figure 3.4. Summary of primitive operator matrices

The levelization process described in section 3.3 allows us to align network elements in groups called partitions. There are two types of partition matrices; partitions of Boolean functions and partitions of crossovers. Both matrices are built in the same manner. To build a partition matrix, we start from the topmost gate in the partition and perform an outer product of all sub-matrices in the partition. We can express the outer product as a matrix product composed of elements that are scaled matrices as in the following equation.

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & a_{13}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & a_{23}B & \dots & a_{2n}B \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{d1}B & a_{d2}B & a_{d3}B & \dots & a_{dn}B \end{bmatrix}$$

The outer product is non-commutative: $A \otimes B \neq B \otimes A$. The dimensions of the two matrices being multiplied together do not need to have any relation to each other, therefore we can multiply any gate by any other.

Once all the partitions are computed, we multiply all the partitions matrices together using the matrix inner product. In the equation below, the matrix F represents the transfer

function.

$$matrixF = matrixA \times matrixB \times \ldots \times matrixZ$$

To multiply two matrices, A and B, the number of columns in A must equal the number of rows in B. This requirement enables us to identify all faulty cascade matrices when a row/column mismatch occurs during the product. To demonstrate this process, we generated transfer function matrices using benchmarks from the ISCAS85 collection. Figure C.1 in the appendix shows the corresponding output matrix for the circuit c17.v. Figure 3.5 describes the model for a logic network characterized as a function F in the transfer function framework. The inputs are denoted by an n-dimensional vector, $|x_i\rangle \in \mathbb{H}^n$ and the outputs by a vector $|f_i\rangle \in \mathbb{H}^m$. We represent the functional behavior of the circuit by the switching function $f(x_1, \ldots, x_n)$ and the $n \times m$ transformation matrix F that serves as the specification of the network transfer function.

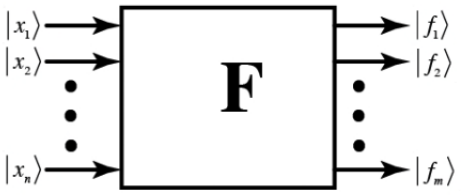

Figure 3.5. A transfer function framework for F

As an example of computing the monolithic transfer function given a structural netlist, consider the example circuit in Figure 3.6 that is composed of one `AND` gate and one inverter. We partition the network into series or cascade. In this example, we identify three partitions θ_1, θ_2, θ_3. Because each partition is composed of a set of parallel elements, the signals on each parallel line must be combined into a single element in $\mathbb{H}^w$ where $\log w$ is the number of parallel network signals in a partition.We build each of the partition matrix using the outer product of each network element. Next, we multiply each partition matrix using the direct matrix multiplication operation.

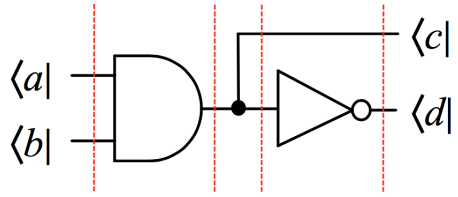

Figure 3.6. Sample circuit of 3 partitions

We write the resulting transfer matrix as:

$$T = T_{\theta_1} \cdot T_{\theta_2} \cdot T_{\theta_3}$$

After partitioning, we calculate each partition transfer matrix using the outer product:

$$T_{\theta_1} = AND = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$T_{\theta_2} = FANOUT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{\theta_3} = INV \otimes I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Next, we perform compute the direct product of the partition transfer matrices

$$T_{\theta_1} \cdot T_{\theta_2} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{\theta_1} \cdot T_{\theta_2} \cdot T_{\theta_3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The resulting transfer matrix is $T = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$

3.7.1. Conversion of Matrices to Algebraic Decision Diagrams

In the previous section the description of the process to compute the overall monolithic transfer matrix for a netlist utilized explicit matrices. While moderate computational efficiency increases can be obtained through the exploitation of sparse matrix methods, the resulting algorithms are still inferior to methods based upon conventional switching algebra models. For the vector space method to be useful in a practical manner it is necessary to reformulate these computations such that computational efficiencies exceed, or at least are equal to, those used in conventional EDA methods based upon switching algebraic foundations. A key contribution of this research is to utilize efficient graph-based algorithms where matrices and vectors are represented as ADDs. To achieve this result, we formulated and implemented all the linear algebraic calculations as efficient graph algorithms. Due to the property of truth table isomorphism, the computational storage requirements for transfer matrices are never worse than those used in conventional switching algebraic models since every representation used in that ubiquitous theory is equally useful in the vector space model.

ADDs represent transfer matrices with explicit row vectors. Shared Binary Decision Diagrams (SBDDs) represent transfer matrices with factored row vectors. In both representations, we interpret non-terminal nodes as matrix row vector indices. In the structure of the SBDD, multiple graphs can share the same terminal nodes. In Figure 3.7, we first interpret a sample circuit as a matrix, then as an SBDD, and finally as an ADD.

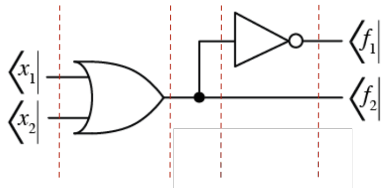

Figure 3.7. Sample circuit of 3 partitions

Using the method explained in Section 5.7.2 we compute the monolithic transfer matrix representing the outputs f_1 and f_2 in the form of decision diagrams.

$$T_{f_1} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \langle 1| \\ \langle 1| \\ \langle 1| \\ \langle 0| \end{bmatrix} \qquad T_{f_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \langle 0| \\ \langle 0| \\ \langle 0| \\ \langle 1| \end{bmatrix}$$

A 0 or 1 path from a vertex in a BDD is a decision on a variable. By reading each row of the truth table of the above functions f_1 and f_2, we can draw the corresponding shared BDDs. Each terminal node at the bottom of the graph is a mapping to a row in the matrix of f_1 and f_2.

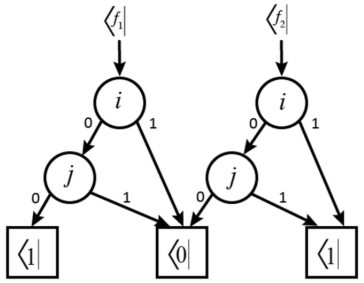

Figure 3.8. Corresponding SBDD for f_1 and f_2

The Shared BDD can be merged into an ADD where the terminal nodes can have values different from $\langle 0|$ and $\langle 1|$. It is a representation of the overall transfer function.

$$T = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \langle 2| \\ \langle 1| \\ \langle 1| \\ \langle 1| \end{bmatrix} \qquad \Rightarrow$$

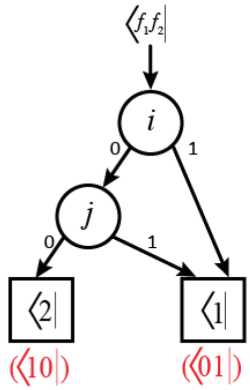

Figure 3.9. Corresponding ADD for $f_1 f_2$

3.8. Building the Transfer Function using Algebraic Decision Diagrams: the

Radix Polynomial Method

For the vector space method to have practical usefulness, it must not require worse computational complexity in either runtime or storage requirements as compared to traditional switching algebraic models. Two approaches for efficiently representing switching functions are cube list representations and binary decision diagrams. Binary decision diagrams (BDD) are widely used in logic synthesis and formal verification of integrated circuits. A BDD is a graph representation that is in the form of a directed acyclic graph [5]. It has one root, branch nodes, and terminal nodes. The root node represents the Boolean function, the leaf nodes are either 0 or 1 and correspond to the constant Boolean functions. A BDD must obey two main rules. First, the diagram must be ordered: this means that we must encounter variables in the same order along the paths. Second, variables may occur at most once along a given path. Additionally, the diagram must be reduced meaning that all redundant nodes are removed, and that isomorphic subgraphs are shared. Figure 3.10 illustrates the difference between a decision tree and a binary decision diagram.

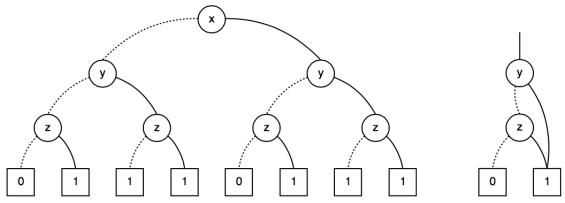

Figure 3.10. A decision tree converted to a Binary Decision Diagram

Many tasks in synthesis, optimization, testing tools, design, and verification of digital systems already manipulate large Boolean functions. However, to add value and improvements to existing EDA tools, we need efficient ways of representing and manipulating such large functions. For the implementation of switching circuit models as transfer functions, we focus on the use of binary decision diagrams. Binary decision diagrams offer a canonical representation of Boolean functions and can be compressed by using the reduction and reordering rules. These attributes make BDDs suitable to save storage and improve efficiency

when dealing with large expressions. The worst-case complexity for BDD representations is $O(2^n)$ where n is the number of dependent variables for switching functions. However, it is well-known that most BDDs are very compact for functions of interest when they are properly represented in reduced form. One motivation of this work is to take advantage of the reordering and reduction rules and provide a compact and reduced representation of functions. Due to truth table isomorphism, a compact BDD representation of a switching function likewise can serve as a compact representation of the vector space transfer matrix.

BDDs have multiple extensions that were devised to further reduce storage requirements. In our implementation, we use Algebraic Decision Diagrams (ADDs) also referred to as Multi-Terminal Binary Decision Diagrams [12]. As an experimental tool, we use `CUDD`: the Colorado University Decision Diagram Package written by Fabio Somenzi [25]. `CUDD` is a $C/C++$ library for creating different types of decision diagrams including binary decision diagrams (BDD), Zero-suppressed BDDs (ZDD), and algebraic decision diagrams (ADD).

3.8.1. Building a Library of BDDs using the `CUDD` Package

Using routines provided by `CUDD`, we built a library of BDDs corresponding to the most common netlist elements. We can create BDDs for primitive logic gates such as `AND, OR, XOR, NOT` using routines for conjunction, disjunction, and complementation. Algorithms of polynomial complexity are already available in `CUDD` and referred to as `Cudd_bddAnd, Cudd_bddOr, Cudd_bddXor, Cudd_bddNot`. These functions can be used to iteratively construct new BDDs from existing ones. The operation is performed by creating a unique variable for each gate input, referencing it, and applying the above routine to the inputs. The functions return a pointer to the resulting BDD if successful. These small BDDs are used as building blocks to compute larger partitions of parallel elements. Since we are now dealing with binary decision diagrams rather than matrices, we implemented a new method to compute the outer product of BDDs. We note that this form of the outer product is also known as the Kronecker product and hence we use these terms interchangeably. This multiplication is carried out by using a novel algorithm based upon a radix polynomial

interpretation. This is one of our key results in this research and is described in [14].

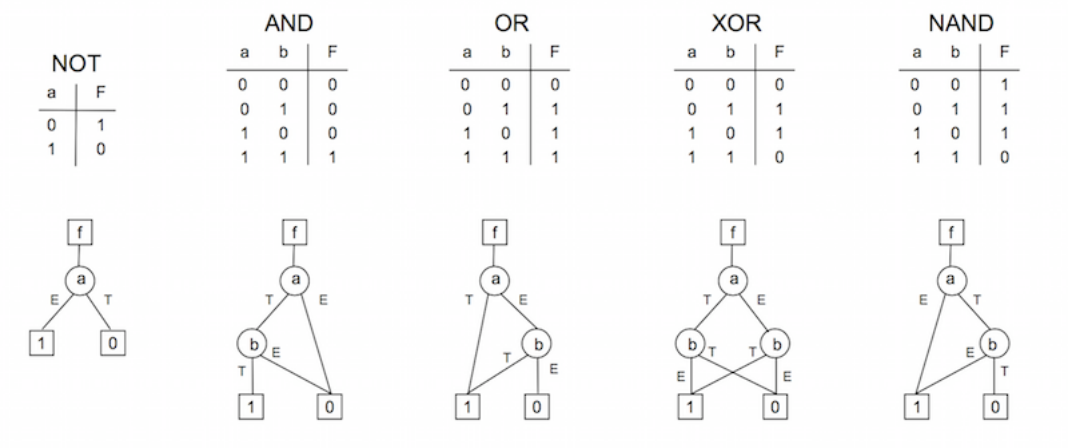

Figure 3.11. Primitive operator BDDs

3.8.2. Building the Partitions BDDs

After applying serial partitioning to the netlist, we obtain partitions containing gates, fanins, fanouts and pass-through wires. To compute the transfer function for a partition, we compute the Kronecker product of each element's transfer matrix in BDD form. The outer product of all the element matrices in BDD form is produced by starting from the topmost element. In linear algebra, the outer product is the tensor product of two elements. Therefore, the product $u \otimes v$ is equivalent to a matrix multiplication $u \cdot v^T$. Starting from the matrix representing the topmost partition element, we multiply it by the transpose of the next matrix in the stage. Each operation is carried in pair, so the resulting matrix is then multiplied by the transpose of the following matrix and so on until we reach the bottommost element in the partition. The BDD representing a switching function is isomorphic to the matrix representation of the same function. Since our implementation focuses more on nodes reduction and reordering, we interpret each network element as BDDs rather than matrices. For the multiplication of BDD, we use a radix polynomial [14] as described in detail in a following section.

3.8.3. Crossovers and Variable Reordering

The transfer function relies on the topology of the network, therefore, we must account for crossovers. Between partitions, we can detect one or more crossovers. The occurrence of crossovers is higher in circuits with more wires. When additional crossing wires are encountered, we can reorder the variables by inserting permutation matrices. Another way to process crossovers is by permuting variables in the partitions following the crossovers. This is can be achieved using function in `CUDD` function `Cudd_addPermute`. This function takes an array containing the wires order and creates a new ADD with permuted variables. Each entry in the array corresponds to a unique variable in the manager. `Cudd_addPermute` returns a pointer to the resulting ADD.

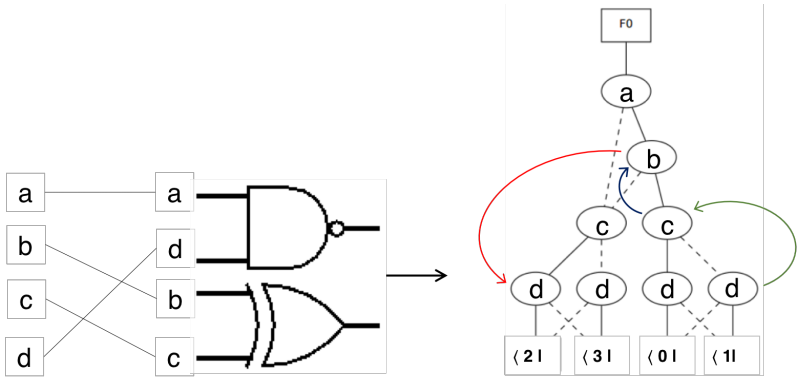

Figure 3.12. ADD Variable permutations

Crossovers are translated into row permutations. The rows in the decision diagram of figure 3.12 represent the variables of the function. The diagram contains four rows a, b, c and d. Our variable reordering technique consists of swapping the position of the variables every time a crossover occurs. When one our more crossing wires are identified between two partitions, we create array entries. These entries are the variables in both partitions. The order of the entries in the array will determine how to reorder the following partitions accordingly. The i^{th} entry of the array is the index of the variable that is to substitute the i^{th} variable. After the permutations are completed, `CUDD` returns a pointer to the new ADD with permuted variables and discards the previous one.

3.8.4. Algebraic Decision Diagrams Kronecker Product using a Radix Polynomial

An algebraic decision diagram is a binary decision diagram whose terminal nodes can be arbitrary integer values instead of just 0 and 1 [2]. It is a suitable data structure to represent and manipulate large sparse matrices efficiently. Since it shares similar attributes with BDDs, we can easily convert one diagram type to another and vice versa. `CUDD` provides integer and floating point multiplication in algebraic decision diagrams. If f and g are two 0-1 ADDs, the function returns the inner product $f \cdot g$. This operation is done using a routine called `Cudd_AddTimes`. We modify the latter function to implement the Kronecker product.

For example, let F be an ADD representing a function of n_1 variables and m_1 outputs. Let G be an ADD representing a function of n_2 variables and m_2 outputs. The resulting Kronecker product $Z = F \otimes G$ is an ADD of $n_1 + n_2$ variables. For each path in ADD Z, the corresponding terminal node is calculated using the following expression:

$$Z_{terminal} = 2^{m_2} \cdot F_{terminal} + G_{terminal}$$

Using the `APPLY` procedure developed by Bryant [4] with the above operator, we build the resultant graph Z. The procedure `APPLY` takes two decision diagrams and an operator $\langle op \rangle$ and generates the reduced graph $F \langle op \rangle G$. The algorithm proceeds from the root nodes of each input graph downward to the terminal nodes. The advantage of this procedure is that it provides a canonical and compressed tree as a result. The time complexity of the Kronecker product is $O(|F|.|G|)$ where $|F|$ and $|G|$ represent the number of vertices in the graph F and G respectively.

To get the diagram representing the outer product of two ADDs we use the following formula:

$$cuddV(Z) = 2^{m_2} \cdot cuddV(F) + cuddV(G) \tag{3.5}$$

`cuddV(F)` and `cuddV(G)` represent the two operands ADDs. `cuddV(Z)` is a pointer to the resulting ADD and represents the result of the multiplication. The following algorithm

shows the required steps for performing the Kronecker product of two Decision Diagram F and G:

Algorithm 1: Kronecker product of two ADDs

Input: ADD F and ADD G
Output: ADD Z representing the resultant graph $F \otimes G$

```
1 m_2 ← number of outputs of G
2 foreach paths in Z from root to terminal do
3     F[i] ← terminal node in F
4     G[i] ← terminal node in G
5     Z[i] ← 2^{m_2} · F[i] + G[i]
6     Build ADD Z
7 return Z;
```

Figure 3.13 illustrates the Kronecker product of an `OR` gate with an `AND` gate.

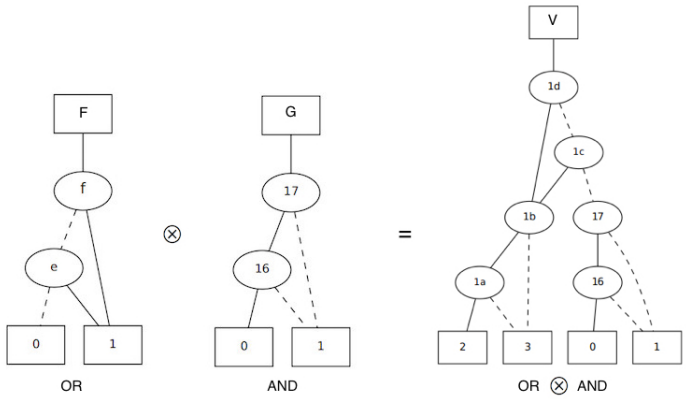

Figure 3.13. Kronecker product or two ADDs

3.8.5. Algebraic Decision Diagrams Direct Product

The direct product is also a necessary operation to obtain the overall transfer function. In the same way as for matrices, we multiply all partitions together starting from the leftmost one. The multiplication of two decision diagrams is a row transformation of the multiplier diagram by the multiplicand diagram. Since we formulated our transformation over the vector space, the values of the rows in the multiplicand ADD are used as pointers to the

rows in the multiplier ADD. The following rules apply to the multiplication of two decision diagrams:

- The number of variables in the resulting decision diagram is equal to the number of variables in the multiplicand diagram
- The direct multiplication of two decision diagrams is non-commutative
- The direct multiplication of decision diagrams is associative

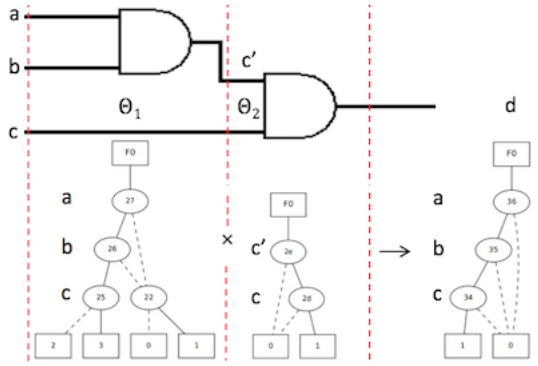

Figure 3.14. Decision diagrams multiplication of circuit partitions

Figure 3.14 shows the partition cuts of a circuit made of two `AND` logic gates. This circuit is parsed in two cascades. The first cascade is composed of an `AND` gate and a wire. It is represented by an ADD of three variables a, b, c and four terminal constants 0, 1, 2, 3. The second cascade is composed of an `AND` gate. It is represented by an ADD of two variables c' and c and two terminal constants 0 and 1. Both partitions are multiplied together to produce an ADD of three variables and two terminal constants 0 and 1. The result represents the transfer function of the logic circuit and is isomorphic to the truth table of a three-input AND gate.

The multiplication algorithm includes two cases: the multiplication of an ADD by an ADD (matrix-by-matrix product), and the multiplication of a constant by an ADD (vector-by-matrix product). n_1 is the number of variables in the multiplicand node F and n_2 is the number of variables in the multiplier node G. Building the resulting ADD involves several

calls to the `Cudd_addIte` method. `Cudd_addIte( f, g, h)` is an algorithm that builds the graph for the composing two functions. It allows us to derive the functions for a logic network or expression containing repeated structures. The ITE Boolean operation stands for if-then-else. It takes three arguments. The two arguments g and h are the Boolean functions to be combined, and the argument f is the resulting function. In `CUDD`, the unique table of nodes is implemented by a hash table. The pointer to f and the two children functions g and h is stored in the entry corresponding to the key (f, g, h). The composition is defined as follow:

$$\texttt{Cudd\_addIte}(f, g, h) = f \cdot g + f^{'} \cdot h \tag{3.6}$$

Algorithm 2: ADD multiplication of two nodes

Input: ADD F and ADD G
Output: ADD Z representing the resultant graph $F \times G$

```
1 if n1 is equal to 0 then
      ▷ F is a constant node
2     Z ← terminal node of G for variable assignment F
3 else if n1 is greater than 0 then
4     foreach paths in F from root to terminal do
5         Z_paths[i] ← variable assignment F
6         Z_terminals[i] ← terminal node of G for variable assignment F
7     Build ADD Z
8 return Z;
```

In our prototype simulator, we experiment with two approaches for computing the output response. In the first approach, we formulate the overall circuit transfer function in the form of a single graph that we refer to as the "monolithic ADD". The second method omits the step of computing the transfer function as a block and instead retains an array of multiple ADDs where each represents the transfer function for an individual serial stage of the partitioned netlist. This implementation and comparison of two vector space simulation methods using decision diagram representations is another key result of this research and is described in detail below as well as in [15].

3.8.6. Additional Structures Added to the CUDD Package

To represent all network elements as binary decision diagrams, we added some additional structures to the CUDD library. Fanout and crossovers are treated as network elements since these structures have differing numbers of outputs. These functions are essential for building the partitions. Fanouts are electrical nodes in which a single conducting wire carries a signal that drives two or more conductors. The fanout ADD represents all the inputs driven by the output of a logic gate. Fanout functions take one argument n, where n is the number of fanout wires and returns the corresponding ADD.

```
DdNode *retval = Cudd_addIte(gbm, var, Cudd_addConst (gbm,
    (CUDD_VALUE_TYPE)coef), Cudd_addConst(gbm, (CUDD_VALUE_TYPE)0));
```

For every fanout, if the input is 0, the output is 0, else if the input is 1, the output is $2^N - 1$. Figure 3.15 shows the diagrams for some 2, 3 and 4 outputs' fanouts.

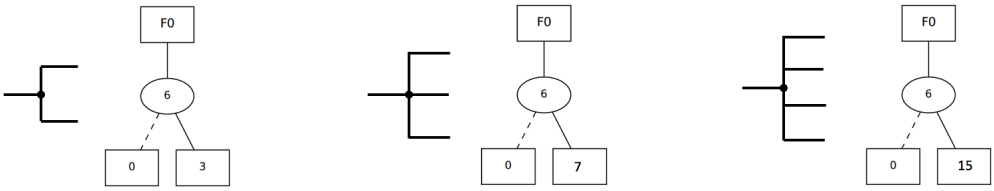

Figure 3.15. ADD representation of a 2, 3 and 4 outputs fanout

The next structure that we added is the ADD for a crossover. Crossovers perform a permutation of two or more rows in the transfer function. The crossover function returns a ADD representing two crossing wires.

Figure 3.16 shows both functions and their corresponding diagram.

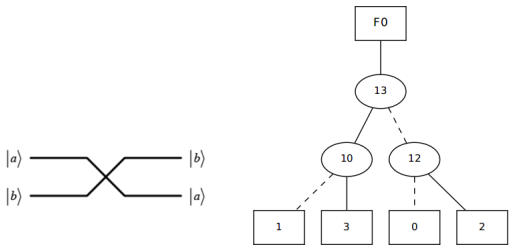

Figure 3.16. ADD representation of a crossover

Chapter 4

FUNCTIONAL SIMULATION USING THE TRANSFER FUNCTION MODEL

4.1. Functional Simulation using the Transfer Function Model

Simulation and justification are core operations in EDA tools. In our proposed method, they can be formulated within the context of a linear algebraic circuit model. Logic simulation is widely used to check that a synthesized netlist meets the expected functional specifications and meets some timing constraints. It can be applied to test generation and fault simulation as well. In a compiled-code simulation, every gate is evaluated once at each simulation. In an event-driven simulation, gates are evaluated only when an event occurs at their inputs. Our goal with this model is to use a transfer function model to perform simulation. After computing the transfer function for a netlist, we can determine its output response using a vector-matrix direct product. The output response of a logic network stimulated by input $\langle x_q|$ and modeled by transfer matrix T is denoted by $\langle f_q|$ and is computed using the following equation:

$$\langle f_q| = \langle x_q| .T \tag{4.1}$$

The output response can be decomposed to determine individual output elements by expressing the output response in terms of a bra notation and then converting the value to a binary string [27]. Deriving the transfer matrix using Equation 4.1 is exponentially complex since it involves the determination of 2^n terms through the use of a simulation tool or some other means. Fortunately, we can determine the transfer matrix of a given logic network through the use of transfer matrices of individual network elements and their corresponding interconnections. We use benchmarks from the ISCAS85 for experimental results.

For simulation using BDD, we use a similar approach. The technique with BDD uses a monolithic Boolean combination. For this purpose, we use the *compose* operator developed

by [4]. The algorithm composes two independent functions and generates a unified BDD. Supposing the graphs f and g represent the BDD for two independent partitions, we can compose f and g by replacing each vertex v in f by the graph of g. Next, we simultaneously replace each branch to a terminal vertex in g with a branch to the children of v depending on the value of the terminal vertex. The composition can be expressed in terms of Boolean operations, according to the following expression, derived from the Shannon expansion theorem:

$$f|_{x_i=g} = g.f|_{x_i=1} + (\neg g).f|_{x_i=0}$$

In our proposed method, each partition in the netlist represents a function on its own. Starting from the primary inputs, we will compose of all the partition graphs until reaching the outputs. The graph obtained from the output represents the output response. For a netlist made of n serial partitions, $n+1$ composition must be performed to obtain the output response.

The concept of a transfer function model for digital circuits is devised such that the input stimulus and the output response are represented by an element in a finite dimensioned Hilbert vector space. The following section on simulation of switching circuits describes the use of the transfer function model to implement and evaluate a prototype simulation tool. The prototype parses a structural netlist in Verilog and constructs the transfer matrix for the netlist in the form of a BDD. Constructing the transfer function of a structural circuit description requires partitioning the netlist into a serial cascade of parallel stages, constructing the transfer matrices of each stage, and combining the stages using a matrix direct product [14]. The advantage of our simulation approach is that it supports symbolic simulation wherein any of the outputs, or subsets of the outputs, can be represented as taking on both binary values simultaneously. In one extreme, all possible input values can be symbolically simulated with one vector-matrix computation. In the other extreme, a single input assignment can be simulated with one vector-matrix product.

The transfer function concept as described in [27] provided the theoretical background for simulation. The corresponding transformation from the input stimulus vector space to the

output response vector space is given by a matrix. In [28] these theoretical results are further extended to cover non-binary switching circuits and to characterize the transfer functions representing switching circuits in spectral domains. To reduce the spatial complexity and improve the performance of applications based on the linear algebraic approach, we use binary decision diagrams (BDDs) to represent vectors and matrices. The implementation of the theory using BDDs is described in [14] where we provided algorithms for parsing a structural netlist into a BDD transfer function, and included required operations such as the inner product of vectors, the direct vector-matrix product, and the outer product of matrices. [14] also provided some experimental results for the generation of transfer functions as Binary Decision Diagrams and made a comparison of the compactness of the diagrams using variable ordering techniques such as sifting.

The following sections describe how the simulator is implemented including the relevant matrix-based and BDD-based algorithms. We experimented with different methods to perform simulation using transfer functions and vectors [15]. Following the implementation, we describe the evaluation of each simulation method in terms of time and storage requirements.

4.1.1. Simulation using a Monolithic Transfer Function

The monolithic method consists of formulating the overall circuit in the form of a single decision diagram. After applying serial partitioning to the netlist and building all partitions, we can combine all the partitions by multiplying them together. This method uses the ADD-by-ADD multiplication algorithm presented in equation 2 to combine all the partitions of the netlist into a single block, then multiplies it by the input stimulus to obtain the output stimulus. The advantage of this method is that we can represent the entire function as one diagram, and simulate for all input combinations in a single iteration. The downside of building a monolithic transfer function is the high memory usage. The monolithic transfer function does not take full advantage of garbage collection.

All nodes in an algebraic decision diagram are stored in hash tables also called unique tables. The hash table guarantees that each node is canonical in the function. This property

also makes decision diagrams canonical. The CUDD manager or $DdManager$ is the collection of all the unique tables and other auxiliary data structures. Prior to simulation, we initialize the $DdManager$ with parameters such as the number of variables, the cache size and the maximum memory size allocatable to the nodes. To build the resulting decision diagram, the CUDD memory manager must keep track of all previous partitions. The spatial complexity for multiplying two partitions is $O(|F|.|G|)$; therefore, simulating netlists with a large number of variables and a large number of unique nodes can consume substantial amounts of physical memory.

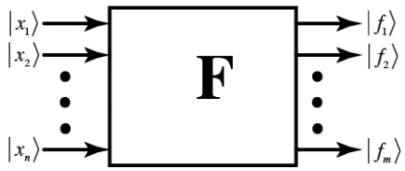

Figure 4.1. A transfer function framework for F (Monolithic method)

Figure 4.1 shows a high-level overview of the transfer function framework. For a circuit of n variables, we incrementally built an ADD of n variables. The input vector $\langle x|$ is composed of $\langle x_1|$, $\langle x_2|$, ..., $\langle x_n|$ and determines the variable assignment for the simulation. We use the variable assignment from vector $\langle x|$ to traverse the ADD F downward starting from variable x_1 and ending on variable x_n. The simulation consists of a vector by matrix multiplication, and the result is a vector. The output vector of the simulation, $\langle f|$, is composed of $\langle f_1|$, $\langle f_2|$, ..., $\langle f_n|$.

4.1.2. Simulation using an Array of Transfer Functions

Rather than building the entire transfer function at once and in one block, as described in the monolithic method, this method performs multiple simulations incrementally starting from the primary inputs. The array method consists of performing multiple vector-matrix multiplications over each partition to obtain the output response. The input vector $\langle x|$ is composed of $\langle x_1|$, $\langle x_2|$, ..., $\langle x_n|$ and determines the variable assignment for the simulation.

A circuit with m partitions requires m intermediate simulations.

CUDD frequently uses garbage collection for better memory management and to reclaim memory that is no longer in use. After initialization, the CUDD manager keeps track of all unique nodes (internal and terminal) using a reference count. The count is incremented every time a new branch points to a node and it is decremented when a node is released. When a node reaches a reference count of zero, it is considered dead. Every time we build a new partition, the $DdManager$ automatically increases the reference count through a method called $Cudd\_Ref$. After that partition i is simulated, we obtain an intermediate output vector f_i and move forward to the next partition $i + 1$. The previous diagram is no longer needed and can be released or dereferenced by making a call to the method $Cudd_RecursiveDeref$. The dereferencing scheme is very useful when dealing with netlists with multiple partitions because only one partition is stored in memory at a time.

The benefit of this technique is that we can free up the nodes of previous cascades after each iteration. Since we use only one partition at a time, building the output response incrementally is ideal for reducing memory usage and increasing the speed of simulation; however, we never end up computing the entire function. In contrast to the monolithic method, we must run a new simulation for every variable assignment of the inputs to test. The experimental results show a comparison of both techniques by providing time and storage requirements.

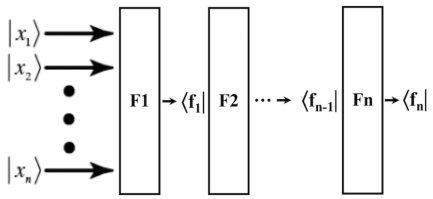

Figure 4.2. A transfer function framework for F (Array method)

Figure 4.4 shows a high-level overview of the array of transfer functions. The array is comprised of smaller diagrams with fewer nodes. The number of inputs and outputs varies between each partition. With every new simulation, we are required to build a new array of

functions.

4.1.3. Simulation using the Distributed Factored Form

The transfer function characterizing a switching network can be expressed in distributed factored form and represented by a set of interconnected transfer functions that are the network elements. This form, when represented graphically, has the same topology as the switching network netlist. Simulation using the distributed vectors consists of traversing the circuit in a distributed manner. Starting from the inputs, we traverse the circuit by simulating each logic gate one at a time. Each gate has its own function and can be simulated individually. As presented in Figure 3.11, we reuse our own library of BDDs comprising all logic gates. The distributed vectors are an efficient way to perform simulation for multiple reasons. This method does not require building intermediate diagrams for partitions; it uses the smallest transfer function possible, which are the transfer functions of the logic gates. The previous simulation methods, the monolithic transfer function (see 4.1.1) and the array of transfer functions (see 4.1.2) required the need for multiple row permutations to process crossover wires. With the distributed distributed factored form there is no need to process crossovers since we are not multiplying partitions. Intermediate crossover partitions are not needed because there is no row transformation involved.

The method works as follows: According to the variable assignment, every input is assigned a constant value either $\langle 0|$ or $\langle 1|$. In `CUDD` these values can be set using calls to `Cudd_ReadOne` or `Cudd_ReadZero`. Next, we simulate all the logic gates within the first partition encountered. By referring to the levelization process described in section 3.3, we can ensure that all the gates are simulated in the order they are listed in the netlist. The output row vectors of the first partition are fed to the logic gates of the next partition. The same steps are repeated for the subsequent partitions until the traversal reaches the outputs. Figure 4.3 shows the simulation of logic gates in the first two partitions of circuit c17 for an arbitrary variable assignment.

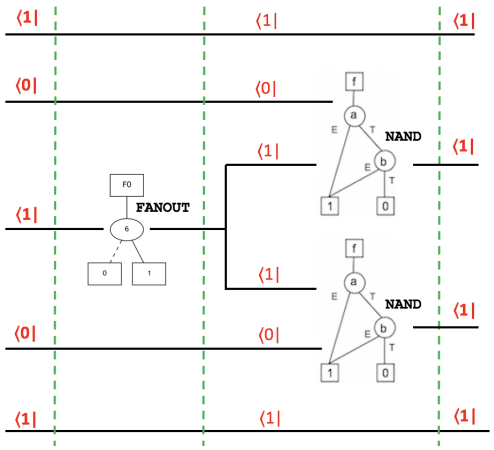

Figure 4.3. Distributed factored form

We experimented with two ways to perform simulation using the distributed factored form. Starting from the primary inputs, we can simulate each logic gate individually. As with an event-driven simulation, we must make sure that every logic gets simulated at the right time using the correct net value. Each partition is simulated in order so that all the nets are updated at the same time.

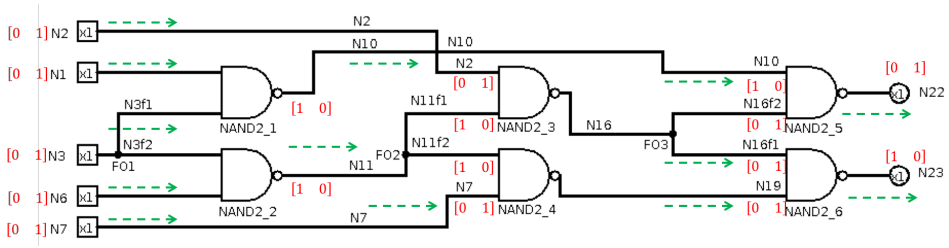

Figure 4.4. Simulation using the distributed vectors

Another way to distribute the vectors for simulation is to start from the outputs and traverse the circuit backwards using recursion. Starting from the outputs, we recurse over all inputs of the gates encountered until reaching the primary inputs. As opposed to the monolithic and the array methods, the distributed method creates a unique decision diagram for each output. Because there is only one output for each diagram the terminal nodes for each decision diagram are $\langle 0|$ and $\langle 1|$.

4.2. Experimental Results

We experimented with each simulation method and collected data such as memory usage and timing. Because the partitioning process is identical for all methods, the number of partitions remains the same. To test the prototype simulator, we use a set of benchmark circuits. Two-level benchmark netlists are converted into multi-level combinational logic circuits in the form of a Verilog structural netlist. For all tests, the inputs are represented as row vectors. The output response is obtained using two different methods. The two-level benchmark netlists are converted into multi-level combinational logic circuits in the form of a Verilog structural netlist before we apply the technique to them. We convert the native .pla files into corresponding Verilog files using Synopsys Design Compiler. The converted files are then in the form of a set of two-level Boolean equations expressed in Verilog syntax. The multilevel netlists are saved as structural Verilog descriptions and used as input to a parser that computes the corresponding binary decision diagrams. For this experiment, we use variable ordering algorithms for building the BDDs. The `CUDD` package offers multiple dynamic reordering algorithms. BDDs and ADDs, which share the same unique table, are simultaneously reordered for efficiency. These algorithms iteratively improve variable order to avoid the BDDs size grows out of boundaries during computation. The first algorithm uses variable reordering and the second one does not use variable reordering. The table below summarizes timing data, the total number of nodes, and memory usage for the use of sifting variable reordering.

Table 4.1. Simulation output response (Monolithic method)

Benchmark	Inputs/ Outputs	# of partitions	# of nodes	Memory (MB)	Time to build partitions (ms)	Time to build ADD (ms)	Time to simulate (ms)
xor5.v	5/1	6	11	8.77	0.44	2.67	0.01
c17.v	5/2	12	12	8.90	0.57	4.88	0.02
majority.v	5/1	12	9	8.98	1.09	5.10	0.01
test1.v	3/3	16	10	8.92	0.86	5.49	0.01
rd53.v	5/3	18	21	9.29	1.21	10.48	0.04
con1.v	7/2	14	15	19.07	2.14	175.67	0.01
radd.v	8/5	28	109	19.12	4.91	296.04	0.03
rd73.v	7/3	24	71	19.34	5.56	76.96	0.01
mux.v	21/1	26	145	33.77	7.61	43.47	0.01
c432.v	36/7	57	451	41.08	240.60	945.89	0.08
c499.v	41/32	16	442	43.14	246.50	850.11	0.09
c1355.v	41/32	16	451	48.12	291.14	928.19	0.12
c880.v	60/26	67	895	67.90	1412.62	6580.10	0.21
c5315.v	178/123	80	1286	83.47	3150.11	7783.62	0.37
c2670.v	233/140	99	1560	97.01	6521.43	8195.09	0.58

Table 4.2. Simulation output response (Array method)

Benchmark	Inputs Outputs	# of partitions	Memory (MB)	Time to build partitions (ms)	Time to build ADDs (ms)	Time to simulate (ms)
xor5.v	5/1	6	8.70	0.44	0.17	0.01
c17.v	5/2	12	8.79	0.57	0.70	0.03
majority.v	5/1	12	8.84	1.09	1.01	0.04
test1.v	3/3	16	8.81	0.86	0.84	0.03
rd53.v	5/3	18	9.06	1.21	2.70	0.07
con1.v	7/2	14	18.82	2.14	65.11	0.44
radd.v	8/5	28	21.08	4.91	195.43	0.54
rd73.v	7/3	24	11.72	5.56	21.37	0.19
mux.v	21/1	26	16.19	7.61	429.69	1.90
c432.v	36/7	57	18.42	240.60	619.09	2.78
c499.v	41/32	16	17.22	246.50	1150.11	2.01
c1355.v	41/32	16	21.15	291.14	1142.30	45.29
c880.v	60/26	67	25.42	1412.62	7100.60	5.21
c5315.v	178/123	80	37.17	3150.11	8752.80	9.75
c2670.v	233/140	99	49.01	6521.43	9450.20	12.87

Table 4.1 and table 4.2 summarize timing data, the total number of nodes, and memory usage for the monolithic method and the array method. The motivation for comparing these two approaches is that the monolithic ADD is larger and requires more memory for representation but only requires a single vector-matrix product computation to obtain the output response. In contrast, the array of ADDs method results in less required storage but requires k vector-matrix computations to compute an output response vector. Furthermore, it is not necessary to compute the entire array of ADDs in the latter method since only a single serial stage is computed at a time. For both methods, crossovers between partitions are handled by reordering variables accordingly in the corresponding ADD.

Table 4.3. Simulation using the distributed factored form

Benchmark	Inputs Outputs	# of partitions	Time to build partitions (ms)	Time to simulate (ms)
xor5.v	5/1	6	0.44	0.12
c17.v	5/2	12	0.57	0.13
majority.v	5/1	12	1.09	0.23
test1.v	3/3	16	0.86	0.25
rd53.v	5/3	18	1.21	0.36
con1.v	7/2	14	2.14	0.53
radd.v	8/5	28	4.91	0.52
rd73.v	7/3	24	5.56	0.56
mux.v	21/1	26	7.61	2.84
c432.v	36/7	57	240.60	48.06
c499.v	41/32	16	246.50	51.82
c1355.v	41/32	16	291.14	45.29
c880.v	60/26	67	1412.62	422.19
c5315.v	178/123	80	3150.11	5837.03
c2670.v	233/140	99	6521.43	9387.50

Table 4.3 shows timing requirements for the distributed method. Except for the netlist traversal, no additional computation such as fanout detection, crossover detection, or row permutations is required.

4.3. Application of the Transfer Function Model to Sequential Circuits

Sequential circuits are another type of logic circuits in which the output depends on the input variable assignment and the previous state of the circuits. The states, previous state, present state, and next state, are logic values of the circuit, which are temporarily stored. The states are stored in memory elements such as registers or flip-flops. There is a finite number of states that the circuit can be in. At every clock cycle, the next state

can be determined as a function of the current state and the current inputs. We extended the transfer function model to apply to sequential circuit as well. The topology of the circuit presented above remains the same. Figure 4.5 below illustrates a state machine that includes a combinational block and a sequential block. The sequential block uses memory and registers as storage elements. The circuit has both external inputs and internal inputs that depend on the previous state output.

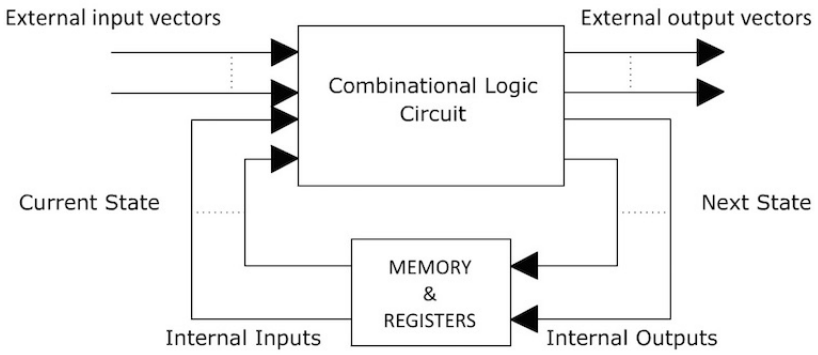

Figure 4.5. Combinational block and sequential block

To adapt our model to sequential circuits, we can unroll the sequential circuit into a larger combinational circuit, then perform the composition or multiplication of the blocks at every cycle. The multiplication of every sub-function can be achieved using the algorithm defined in equation 2. Knowing there are a finite number of states in the circuit, the number of multiplication is equal to n, where n represents the number of states. Figure 4.6 illustrates the process of unrolling the sequential circuit.

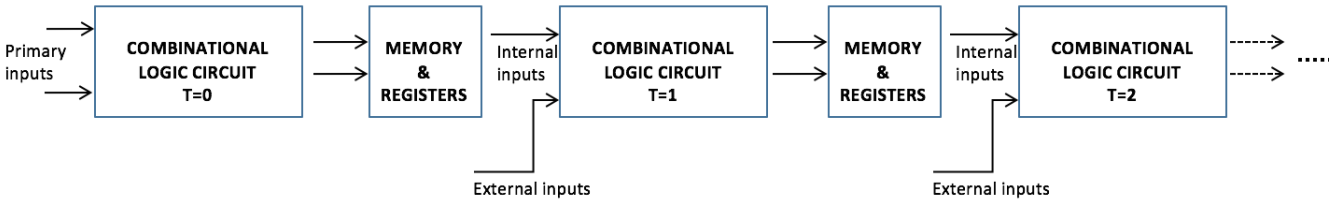

Figure 4.6. Unrolling of the sequential circuit

Starting with an initial input vector representing the primary inputs, we build the transfer function for the combinational logic block using the proposed method in [14]. On the first cycle, we perform the multiplication of the input by the transfer function representing the combinational circuit. Multiplying a vector by a matrix performs a linear transformation of the input vector, so the output of the transfer function is also a vector. On the next cycle, we reuse that output vector as the input to the same transfer function combined with some new external inputs. We repeat the same process for multiple cycles.

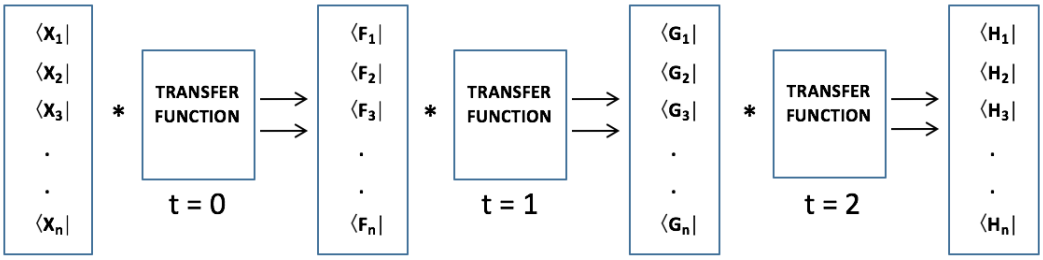

Figure 4.7. Iterations of vector multiplications

Example 4.1 The example in Figure 4.8 is a basic synchronous sequential circuit. Let's assume an execution of 3 clock cycles. In this example, we are modeling the combinational circuit as a transfer matrix to show the linear transformation on the input vectors in each cycle.

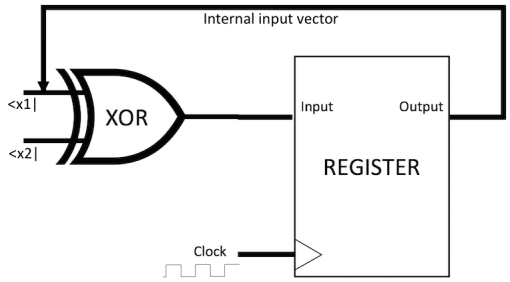

Figure 4.8. Basic synchronous sequential circuit

Figure 4.9 shows how the sequential circuit is unrolled from time $t = 0$ to $t = 3$.

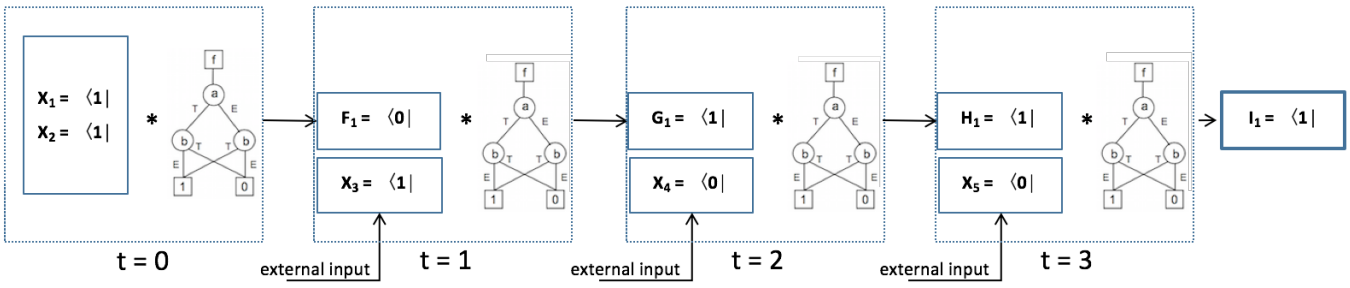

Figure 4.9. Synchronous sequential circuit unrolled on 3 cycles

At time t=0, let $\langle x_1| = \langle 1|$ and $\langle x_2| = \langle 1|$ be the primary input vectors.

We multiply the input vectors by the transfer function representing the Exclusive-OR:

$$\langle f_1| = \langle 11| \cdot T = [\,0\;0\;0\;1\,] \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [\,1\;0\,] = \langle 0|$$

At time t=1, an external input vector $\langle x_3| = \langle 1|$ comes in.

We multiply the input vector by the transfer function representing the Exclusive-OR:

$$\langle g_1| = \langle 01| \cdot T = [\,0\;1\;0\;0\,] \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [\,0\;1\,] = \langle 1|$$

At time t=2, an external input vector $\langle x_3| = \langle 0|$ comes in.

We multiply the input vector by the transfer function representing the Exclusive-OR:

$$\langle h_1| = \langle 10| \cdot T = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} = \langle 1|$$

At time t=3, the resulting output vector for the sequential circuit is: $\langle 1|$ □

Example 4.2 The example in Figure 4.10 is a modified version of the benchmark circuit c17. Both outputs are registered. In this example, we will model the combinational circuit as a decision diagram to show the linear transformation on the input vectors in each cycle.

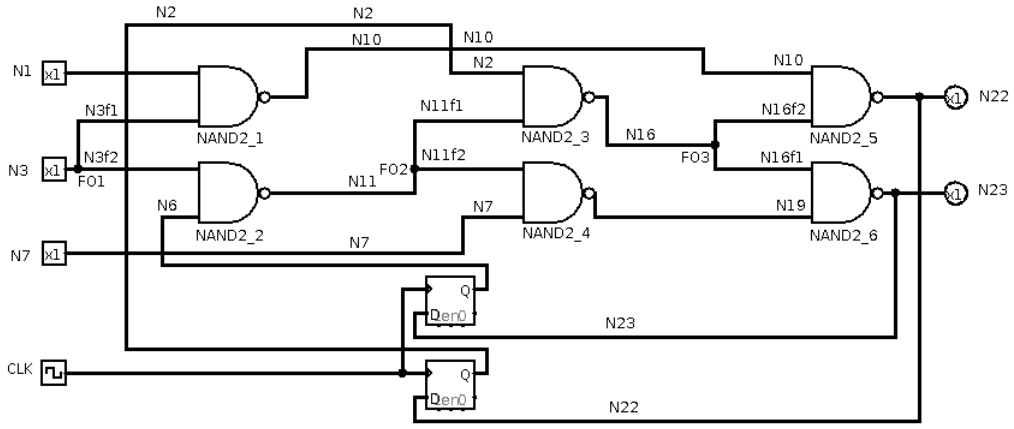

Figure 4.10. Registered version of c17

□

When unrolling the sequential circuit, we are performing the same operation over multiple cycles. The complexity of the transfer function is $O(n^2)$ for n variables. Assuming we are running for t cycles, the absolute complexity would increase to $t \times O(n^2)$. The relative complexity remains $O(n^2)$.

For sequential circuits that have multiple registers breaking the combinational path, we need to build transfer functions for each of the combinational blocks in the circuit. To apply our current topology to sequential circuits, we would need to modify the parser to handle states. While reading the netlist, the parser must be able to identity all the combinational

logic block that are separated by registers. Each combinational logic block is treated as a sub-circuit; therefore, it must have an independent transfer function. The challenge during the parsing process is to locate all the sub-circuits separated by registers.

We experimented with benchmark circuits for ISCAS89. These netlists where written to test sequential test pattern generation algorithms. They are relatively larger and more complex than the previous benchmarks from ISCAS85. "s" means that the circuit is synchronous, sequential, and the suffix number is the number of interconnect lines among the circuit primitives. There is no functional description of these circuits. The type of circuits in the benchmark: are 4-bit multipliers, traffic-light controllers, PLD devices. Other circuits such as `s1238` are combinational circuits with randomly inserted flip-flops.

During the traversal, the parser splits the netlist into multiple sub-circuits. Each combinational block has two types of inputs; the primary inputs and the feedback inputs from blocks. After identifying each subcircuits we can build the transfer function using the methods presented in Chapter 4.1.1

Example 4.3 We experimented with circuit `s27` which is a sequential circuit of 10 gates and 3 D Flip-Flops. During the traversal, the parser splits the netlist into 2 sub-circuits. The first transfer function includes the logic before the registers and the second function includes the logic after the registers. Figure 4.11 shows the schematic of `s27` with both sub-circuits.

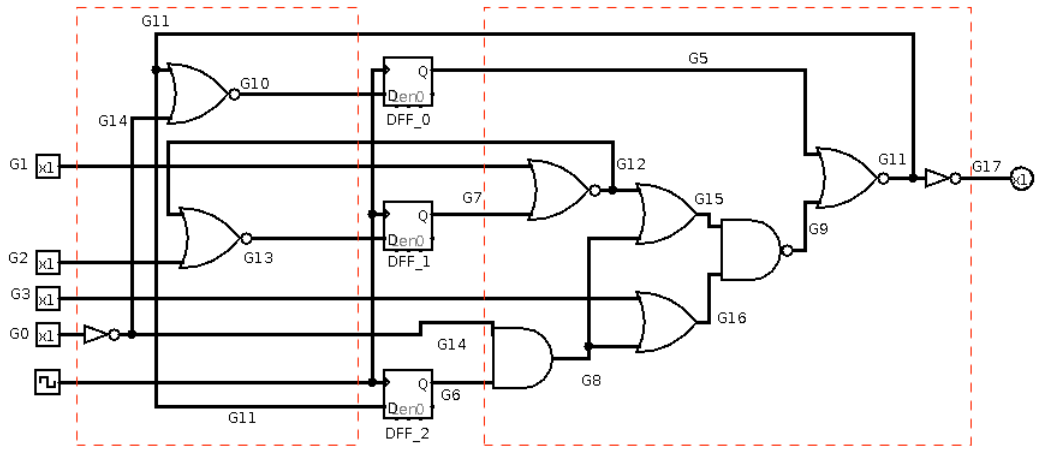

Figure 4.11. s27 combinational logic blocks

To simulate `s27` over n cycles, we need to unroll the circuit n. Since `s27` has 2 monolithic functions, we would perform $2 \times n$ simulations to get the output. The primary inputs are $G2$, $G0$, $G11$, and $G13$. The feedback inputs are $G11$ and $G12$. Figure 4.12 shows the schematic of `s27` with both sub-circuits.

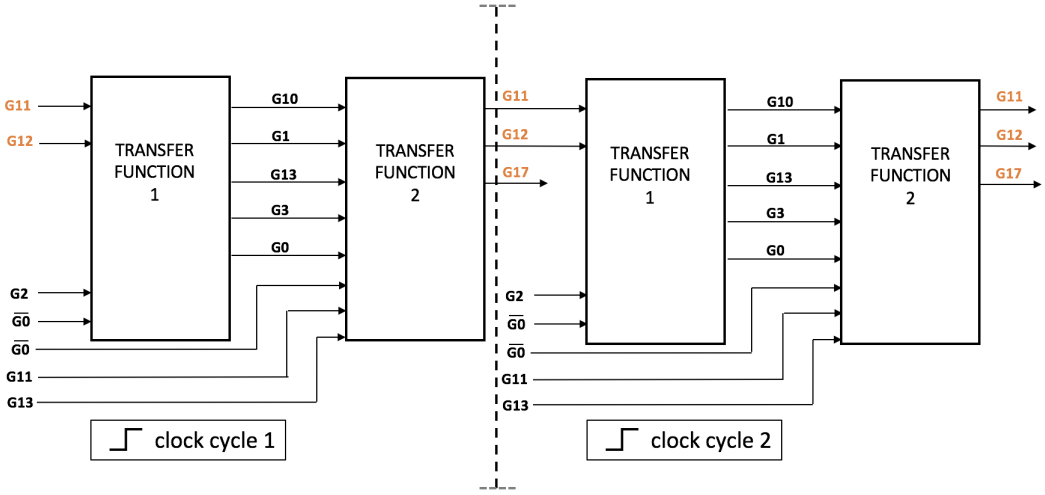

Figure 4.12. s27 loop unrolling of two transfer functions

The ADD of transfer function 1 and transfer function 2 are displayed below.

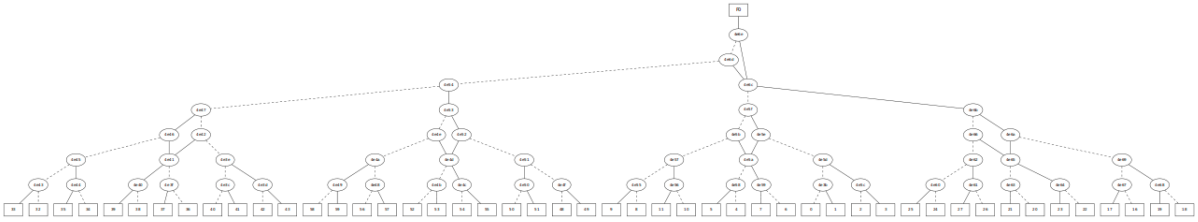

Figure 4.13. s27 transfer function 1

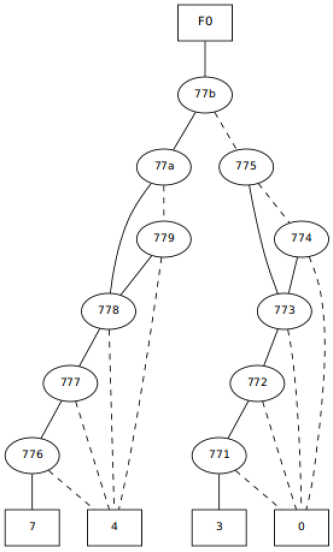

Figure 4.14. s27 transfer function 2

□

Chapter 5

JUSTIFICATION USING THE TRANSFER FUNCTION MODEL

Justification is the inverse problem of simulation. Given the output response and the characterization of a logic network the objective is to compute the corresponding input stimuli. The advantage of using a transfer function matrix is that we can easily perform justification by using the transpose of the matrix. By formulating a method for justification within the linear algebraic framework, we obtain the input vector $\langle x|$ from the following equation:

$$\langle x| = \langle f| \, T^{-1} \tag{5.1}$$

The theory for performing justification using the vector space model is given in [27]. In our implementation of the theory, we use the ADD representation to perform justification. A significant outcome of this portion of the research is that we show justification is performed with the same complexity as that of simulation as described in the previous Chapters. When justification is performed using traditional switching theory models, high complexity is often required when satisfiability algorithms are employed which are known to have inefficiencies in the worst case.

5.1. Justification using the Transfer Matrices

Some EDA tasks require the determination of an input stimulus given a transfer function and an output response. Using an output response and the characterization of a logic network, we are interested in computing the corresponding input stimuli. Solving the justification problem contributes to multiple design and analysis applications such as synthesis, verification, and test. We can perform justification using the vector space model without resorting to creating multiple variable assignments or backtracking. In the vector space model,

we perform simulation using a single vector-matrix multiplication. The same method applies to justification which can also be performed using a vector-matrix multiplication.

To formulate a method for justification within the linear algebraic framework, we solve Equation 5.5 for the input vector $\langle x|$.

$$\langle x| = \langle f|\, T^{-1} \tag{5.2}$$

A naive approach to the justification problem requires the inverse matrix T^{-1} be formulated. However, this is generally not possible since T is usually not of full rank and T^{-1} only exists if the network is reversible. In other words, the transfer matrix T which is also the simulation matrix, must be of full rank to obtain T^{-1}. A large number of switching networks do not meet these requirements; therefore, justification would not be possible using this method because we cannot compute the inverse matrix T^{-1}. Our first approach consists of using the Moore-Penrose pseudo-inverse T^{+}. Depending on the number of rows and columns, N and M respectively, the system can be over-specified or under-specified. To obtain the pseudo-inverse T^{+}, we solve Equation 5.3:

$$T^{+} = \begin{cases} (T^{*} \cdot T)^{-1} T^{*} \\ T^{*} \cdot (T T^{*})^{-1} \end{cases} \tag{5.3}$$

T^{*} represents the transpose of T. The multiplication $(T^{*}T)$ represents the Gram matrix and is denoted as $gram(T)$. The Gram matrix is always invertible and is in the form of a diagonal matrix. Because $gram(T)$ is diagonal, the inverse $gram(T)^{-1}$ is also diagonal. All the components of the $gram(T)^{-1}$ are the multiplicative inverses of the values on $gram(T)$. Due to this fact, the multiplicative inverses of the Grammian are simply constants and can be ignored for the the purpose of computing input justifications. We can avoid computing the pseudo-inverse altogether by using the matrix transpose T^{*}. The justification T^{J} matrix is thus obtained using Equation 5.4:

$$T^J = T^T \tag{5.4}$$

The example below shows how we can use the justification matrix for a 2-input `AND` gate and an output vector $\langle 0|$.

Example 5.1

$$T_{NAND} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$T^J = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\langle x_1 x_2| = \langle f| \cdot T^J = \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix}$$

□

The row vector resulting from the multiplication shows that 3 input vectors cause an output vector $\langle 0|$.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix} = \langle 00| + \langle 01| + \langle 10|$$

5.1.1. Justification using Column Vectors

The first method to perform justification consisted of using the transpose matrix T^T. Another approach to justification consists of representing the input stimulus and output response as column vectors instead of row vectors. By reusing the same transfer function T, we can determine the corresponding input stimulus. This method avoids the need for extra computation to build the transpose matrix.

$$|x\rangle = T \cdot |f\rangle \tag{5.5}$$

Example 5.2

$$T_{NAND} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$|x_1x_2\rangle = T \cdot |f\rangle = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

□

The column vector resulting from the multiplication shows that 3 input vectors cause an output vector $|0\rangle$.

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = |00\rangle + |01\rangle + |10\rangle$$

5.2. Justification using Algebraic Decision Diagrams

5.2.1. Background

Just as with matrices, we can perform justification using decision diagrams and their terminal node vectors. An ADD has a root, branch nodes, and terminal nodes. The root is the Boolean function, the branch nodes represent the variables, and terminal nodes, which are constants, represent the output. Our objective with justification is to get the combination of all possible input vectors that can evaluate to an output.

Previous backtracking algorithms have been proposed to extract the cubes of a Decision Diagram. Because of the compactness of decision diagrams, multiple variable assignments are merged to the same path. Redundant nodes are also removed for optimization. [25] proposed a recursive algorithm called OneSat to find one satisfiable assignment for a formula. The procedure takes three arguments and computes one cube in the function. The first argument is the root node. The second argument is the complementation parity of the path. This value is generally set to 1, because the `IF` decision is 1 and the `THEN` decision is 0. The third argument is an array containing the variables. The OneSat algorithm is presented below:

In the following section, we propose a method to perform justification without any additional traversal of the graph. However, this procedure requires that we modify the structure of the ADD nodes by including some additional information in the leaf node or terminal

Algorithm 3: OneSat algorithm to find one variable assignment

```
1 if v is terminal node then
2 |   return p;
3 sat[v → index] = 1
4 if OneSat(v → T, p, sat) then
5 |   return 1;
6 sat[v → index] = 0
7 if v → E is complemented then
8 |   complement p;
9 return OneSat(v → E, p, sat);
```

nodes.

5.2.2. The Vector Space

The transfer function model presented in chapters 2 and 4 represented inputs variables as row vectors $\langle i|$ and the outputs as row vectors $\langle f|$. When performing simulation, we traverse the graph according to the variable assignment and the return value in a row vector of n elements, where n is the number of variables. To perform justification using the vector space, we will represent the terminal nodes as column vectors $|f\rangle$.

Example 5.3 The following circuit in figure 5.1 has three inputs and two outputs.

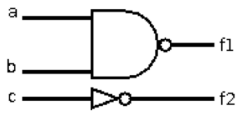

Figure 5.1. Sample circuit

Figure 5.5 shows the circuit's transfer matrix and its corresponding Algebraic Decision Diagram. The function has four output vectors $\langle 0|$,$\langle 1|$,$\langle 2|$, and$\langle 3|$.

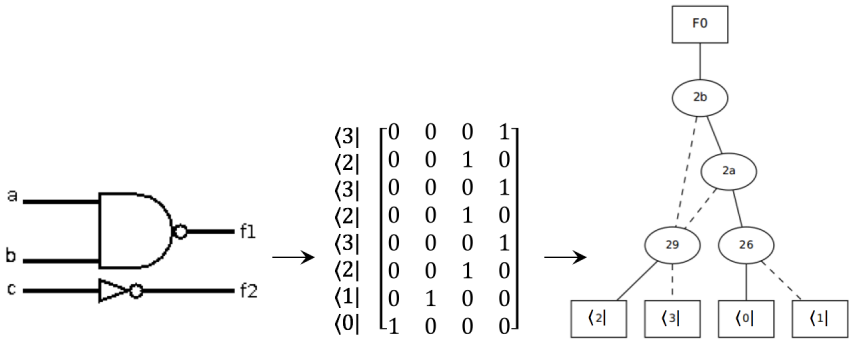

Figure 5.2. Sample circuit

The next step consists of interpreting the terminal row vectors as column vectors. While building the decision diagram, we update each terminal node with the row indexes of each cube. This is achieved by modifying the node struct in `CUDD` by adding an array of indices to each terminal node. The array is of type `char`, and each index is of 1-bit size.

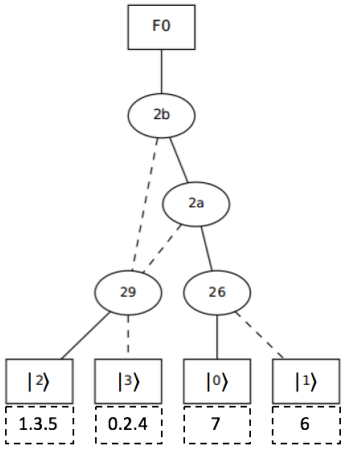

Figure 5.3. Column vectors with row indices

The advantage of using column vector is that we can extract all the satisfiable variable assignments combined into a single row vector. Figure 5.4 shows how we can extract the corresponding input vector using the justification ADD.

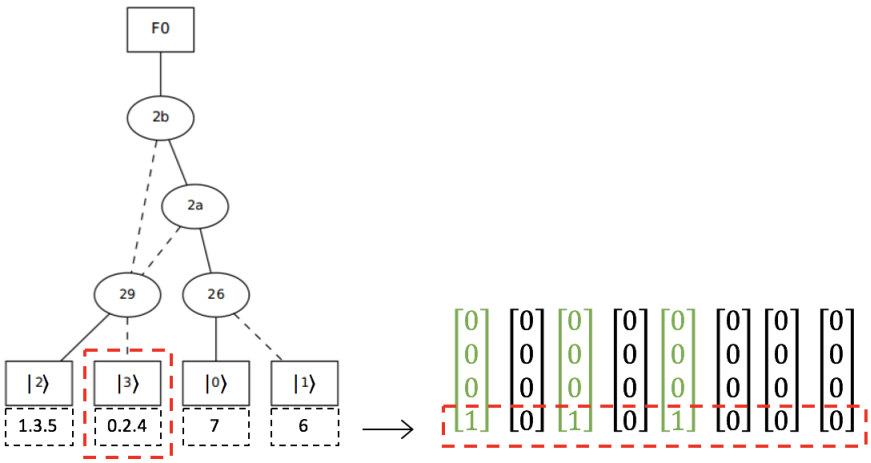

Figure 5.4. Justification on the output column vector $|3\rangle$

The corresponding input vectors are extracted as follows:

$$[\,1\,0\,1\,0\,1\,0\,0\,0\,] = \langle 0| + \langle 2| + \langle 4|$$

□

5.3. Justification using the Distributed Factored Form

The method described above, based on a monolithic ADD would work for circuits with a small number of inputs. When dealing with larger circuits with a large number of inputs, the overall ADD would grow exponentially in size because all the row indexes would be stored in the terminal nodes. Because the size of decision diagram structure matters in our implementation, we use an alternative method based on the distributed factored form. Using the distributed factored form, we perform a backward traversal at the level of each logic gate. For justification of an output, we traverse the circuit backward starting from that output and distribute vectors all the way to the primary inputs. The advantage of the distributed factored form is that we use smaller independent decision diagrams without the need to build the entire transfer function. Before proceeding, we need to modify the existing library of decision diagrams. Each decision diagram from the library is now updated to contain row

indices. For each 2-input logic gate, there is a maximum of 4 row indices to add to the terminal nodes.

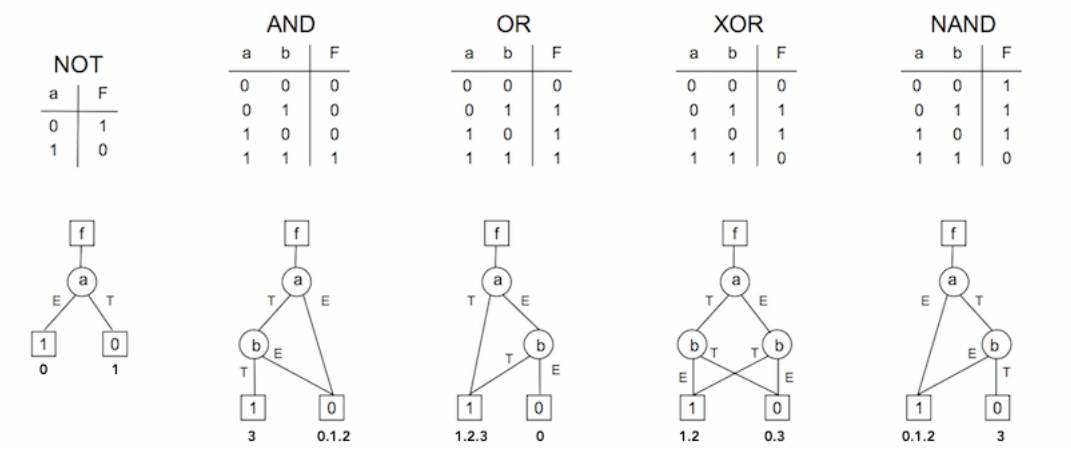

Figure 5.5. Primitive operator BDDs

The following example shows a backward traversal of circuit c17. During the parsing step, we replace all logic gates by their corresponding ADD.

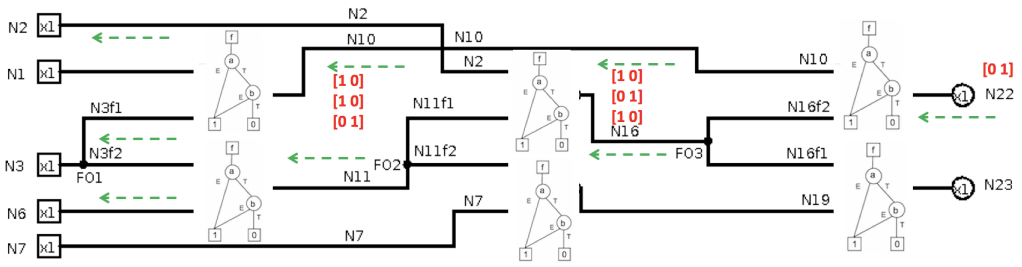

Figure 5.6. Backward traversal of circuit c17

Table 5.1. Justification using the distributed factored form

Benchmark	Inputs Outputs	# of partitions	Time to build partitions (ms)	output to justify	Time to justify (ms)
xor5.v	5/1	6	0.44	xor5	9.18
c17.v	5/2	12	0.57	n22	10.21
majority.v	5/1	12	1.09	o0	11.29
test1.v	3/3	16	0.86	o1	12.34
rd53.v	5/3	18	1.21	o0	12.47
con1.v	7/2	14	2.14	f0	16.62
radd.v	8/5	28	4.91	o0	19.59
rd73.v	7/3	24	5.56	o2	19.66
mux.v	21/1	26	7.61	q	21.89
c432.v	36/7	57	240.60	N223	63.12
c499.v	41/32	16	246.50	N724	69.29
c1355.v	41/32	16	291.14	G1324	59.90
c880.v	60/26	67	1412.62	N388	536.98
c5315.v	178/123	80	3150.11	G5193	6141.21
c2670.v	233/140	99	6521.43	N398	10182.03

Table 5.1 shows timing requirements for the distributed method. Except for the backward traversal of the netlist, no additional computation such as fanout detection, crossover detection, or row permutations is required. The values above are based on the justification for output values $|1\rangle$. We can verify the correctness of the computed inputs but performing simulation using each row vector. The computed justified input values are represented as a list of row indices.

Example 5.4 c17 justified inputs for n22 = $|1\rangle$ are [0.1.4.5.8.9.12.13.16.17.20.21.22.23] □

5.4. Representation of the Justified Inputs as an ADD

An alternative to represent all justified inputs is to prune the Algebraic Decision Diagram of the monolithic transfer function. We can prune the monolithic transfer function using a backtracking algorithm to include all the satisfiable variable assignments in a single decision diagram. The resulting decision diagram would have one terminal node representing the output under test. This method is based on a combination of the recursive algorithms `OneSat` and `SatHowMany` developed by Somenzi in [25].

Algorithm 4: OneSat algorithm to find one variable assignment

```
1  if  v is terminal node then
2  |   return p;
3  sat[v → index] = 1
4  if  OneSat(v → T, p, sat) then
5  |   return 1;
6  sat[v → index] = 0
7  if  v → E is complemented then
8  |   complement p;
9  return OneSat(v → E, p, sat);
```

Now that we can identify variable assignments, we need to identify the shortest traversal to prune the Decision Diagram.

Algorithm 5: SatHowMany algorithm to find the number of traversals (variable assignments)

```
1  if  v is a terminal node then
2  |   return 2^n;
3  if  v is in the unique table then
4  |   return result from the unique table;
5  countT = SatHowMany[v → T, n]
6  countE = SatHowMany[v → E, n]
7  if  v → E is complemented then
8  |   countE = 2^n − countE
9  count = (countT + countE) / 2;
10 insert (v, count) in table;
11 return count;
```

The complexity of this algorithm relies on the total number of nodes visited. The number of visited nodes is at most $2n + 1$; therefore the complexity is n.

Example 5.5 On a circuit such a c17 we can use the backtracking algorithm presented above to pruning the transfer functions for each justification of the output values $|0\rangle$, $|1\rangle$, $|2\rangle$ and $|3\rangle$

Figure 5.7. c17.v transfer function ADD

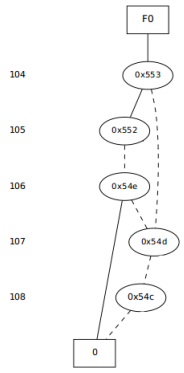

Figure 5.8. Justified inputs for output $|0\rangle$

Justification for the output $|0\rangle$ involves the traversal of 5 internal nodes if the transfer function.

Figure 5.9. c17.v transfer function ADD

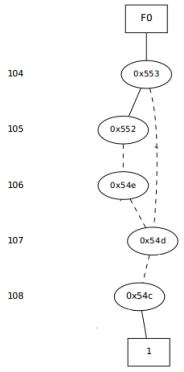

Figure 5.10. Justified inputs for output $|1\rangle$

Justification for the output $|1\rangle$ involves the traversal of 5 internal nodes if the transfer function.

Figure 5.11. c17.v transfer function ADD

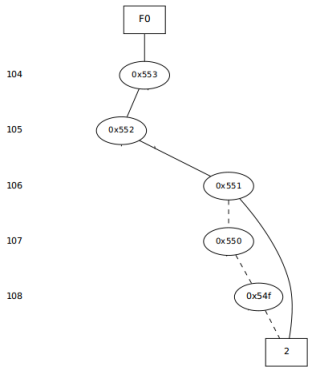

Figure 5.12. Justified inputs for output $|2\rangle$

Justification for the output $|2\rangle$ involves the traversal of 5 internal nodes if the transfer function.

Figure 5.13. c17.v transfer function ADD

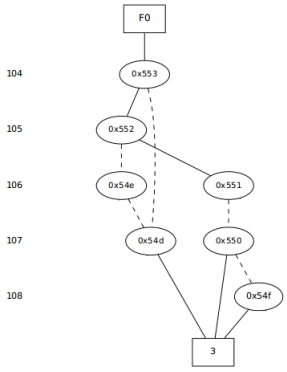

Figure 5.14. Justified inputs for output $|3\rangle$

Justification for the output $|3\rangle$ involves the traversal of 7 internal nodes if the transfer function.

□

The following table shows the timing requirements for the pruning algorithm applied to some benchmark circuits.

Table 5.2. ADD Pruning algorithm runtime

Benchmark	Inputs/ Outputs	# of partitions	# of nodes	Memory (MB)	Time to build partitions (ms)	Time to build ADD (ms)	Time to prune ADD (ms)
xor5.v	5/1	6	11	8.77	0.44	2.67	1.12
c17.v	5/2	12	12	8.90	0.57	4.88	1.21
majority.v	5/1	12	9	8.98	1.09	5.10	1.34
test1.v	3/3	16	10	8.92	0.86	5.49	1.40
rd53.v	5/3	18	21	9.29	1.21	10.48	1.59
con1.v	7/2	14	15	19.07	2.14	175.67	1.81
radd.v	8/5	28	109	19.12	4.91	296.04	1.93
rd73.v	7/3	24	71	19.34	5.56	76.96	2.21
mux.v	21/1	26	145	33.77	7.61	43.47	2.98
c432.v	36/7	57	451	41.08	240.60	945.89	3.83
c499.v	41/32	16	442	43.14	246.50	850.11	3.21
c1355.v	41/32	16	451	48.12	291.14	928.19	3.01
c880.v	60/26	67	895	67.90	1412.62	6580.10	4.08
c5315.v	178/123	80	1286	83.47	3150.11	7783.62	4.30
c2670.v	233/140	99	1560	97.01	6521.43	8195.09	4.87

The pruning algorithm based one backtracking is efficient because of the canonicity of the unique table used in `CUDD` (See [26]). The unique table consists of as many hash tables as there are variables in use. The above method is able to recurse starting from the terminals to the root nodes. From a performance point of view, without the hash tables in the `CUDD` framework, the justification procedure would have a runtime of $O(2^n)$. According to the experimental results, the unique table of the decision diagram enables us to keep the complexity at $O(n)$ for pruning the ADD.

Chapter 6

ALGEBRAIC NORMAL FORM DEGREES COMPUTATION

6.1. Background on the Algebraic Normal Form

Cryptographic primitives serve as the building blocks of larger cryptographic systems. It is common to represent or model a cryptographic primitive of n inputs and m outputs as a collection of r Boolean or switching functions of the form $f_i : \mathbb{B} \rightarrow \mathbb{B}$. B represents the binary set of scalars, $\mathbb{B} = \{0, 1\}$.

The Algebraic Normal Form (ANF) allows for direct observation of the algebraic degree of the switching function. We can obtain the ANF by parsing and traversing a structural netlist. Due to both the usefulness of the ANF and the complexity in extracting it, we are motivated to find a technique that allows for the extraction of the ANF from a structural netlist. In this chapter, we present a technique whereby an ANF coefficient can be extracted through a traversal of a netlist of N gates or operators with complexity $O(N)$. Another difficulty is that computation of the ANF is a computationally expensive process. A function of the form $f_i : \mathbb{B} \rightarrow \mathbb{B}$ is characterized by an ANF coefficient vector a that is comprised of 2^n elements, a_i where $a_i \in \mathbb{B}$. Therefore, explicit storage of a results in an exponentially sized vector. For the largest values of n, explicit computation methods run exponentially and are prohibitively expensive.

Our switching function may be an exact model of a portion of an electronic circuit or software algorithm comprising a cryptographic primitive, or it may be a switching function used to model a portion of a primitive. In some cases, the switching function may not be fully specified.

6.1.1. The Algebraic Normal Form

Switching functions are of the form $f_i : \mathbb{B} \rightarrow \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$ and n is a positive integer representing the number of dependent variables of f. Each switching function f, can be characterized in a variety of normal forms. Normal forms are canonical in the sense that any unique fully specified switching function has one and only one normal form. Due to the canonicity property, normal forms are convenient for use in equivalence proofs and other common tasks in the design and analysis of information processing methods.

As an example, a small switching function where $n = 3$ is used to illustrate these common representations where each ith valuation of f_i is denoted by m_i where $m_i \in \mathbb{B}$. The specific symbolic form for the sum of minterms (SOM) representation of a switching function where $n = 3$ is given in Equation 6.1.

$$\begin{aligned} f = m_0 \cdot \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + m_1 \cdot \overline{x_1} \cdot \overline{x_2} \cdot x_3 + m_2 \cdot \overline{x_1} \cdot x_2 \cdot \overline{x_3} + m_3 \cdot x_1 \cdot \overline{x_2} \cdot \overline{x_3} \\ + m_4 \cdot \overline{x_1} \cdot x_2 \cdot x_3 + m_5 \cdot x_1 \cdot \overline{x_2} \cdot x_3 + m_6 \cdot x_1 \cdot x_2 \cdot \overline{x_3} + m_7 \cdot x_1 \cdot x_2 \cdot x_3 \end{aligned} \tag{6.1}$$

The algebraic normal form utilizes two operators only, the conjunctive logical-AND operator as a product and the disjunctive modulo-2 additive operator commonly referred to as the Exclusive-OR. All literals are present in positive polarity form only so the unary inversion/logical-NOT operator is not present. The symbolic ANF for a general switching function where $n = 3$ can also be written where the coefficients are $a_i \in \mathbb{B}$. In this case, each conjunctive product term or monomial formed by the function literals takes on a different form as shown in Equation 6.2.

$$\begin{aligned} f = a_0 \cdot (1) \oplus a_1 \cdot x_1 \oplus a_2 \cdot x_2 \oplus a_3 \cdot x_3 \oplus a_{12} \cdot x_1 \cdot x_2 \\ \oplus a_{13} \cdot x_1 \cdot x_3 \oplus a_{23} \cdot x_2 \cdot x_3 \oplus a_{123} \cdot x_1 \cdot x_2 \cdot x_3 \end{aligned} \tag{6.2}$$

According to Equation 6.2, the overall degree of a particular switching function is the maximum degree monomial present in the ANF representing that function. We can represent a linear transformation using the linear algebraic notation where m represents a column vector whose components are $m_i \in \mathbb{B}$ and where a represents a corresponding column vector whose components are $a_i \in \mathbb{B}$. The transformation then takes the form $a = R \cdot m$ where R is

the characterizing linear transformation matrix. In many cases, structural netlist representations of switching functions are much more compact than other forms of representation, particularly as the number of dependent variables n increases. Representing switching functions as a structural netlist causes the n dependent variables to be represented as n inputs to the netlist and the value of the switching function f_i is represented as the netlist output.

The matrices R are also known as the positive-polarity Reed-Muller (PPRM) transformation matrices, and the vector of ANF coefficients, as the PPRM spectrum. From this point of view, the vector a characterizing a switching function is referred to as the ANF spectrum and is the same as the PPRM spectrum.

For a circuit with a single variable, the Reed-Muller matrix is represented as follows:

$$R_1 = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$$

For a circuit with three variables, the Reed-Muller matrix is represented as follows:

$$R_3 = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right] \otimes \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right] \otimes \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$$

For a circuit with n variables, the Reed-Muller matrix is represented as follows:

$$R_n = \bigotimes_{\mathrm{j}=1}^{\mathrm{n}} \cdot R_1$$

6.2. Method for Extracting the ANF from a Netlist

6.2.1. Constants Modeled in the Switching Domain and the ANF Domain

In our linear algebraic model presented here, we model constants as elements of $\mathbb{H}$. We model the constant element $\langle 0| \in \mathbb{H}$ as $\langle 0| = [1, 0]$ and $\langle 1| \in \mathbb{H}$ as $\langle 1| = [0, 1]$. We also amend the elements of $\mathbb{H}$ resulting in $\mathbb{H}^+ = \{\mathbb{H}, \langle \emptyset|, \langle t|\}$. The additional elements $\langle \emptyset|$ and $\langle t|$ are included for the purpose of developing the transfer function model for a logic network and

their inclusion is convenient in analysis of the modeled network. Qualitatively, the element $\langle\emptyset|$ can be considered as the inexistence of either $\langle 0|$ or $\langle 1|$, while the element $\langle t|$ represents an element that is simultaneously both $\langle 0|$ and $\langle 1|$ so that $\langle t| = \langle 0| + \langle 1|$. In using $\mathbb{H}^+$, we denote a lattice algebra by a Hasse diagram. Figure 6.1 contains the Hasse diagram for the elements of the set $\mathbb{H}^+$.

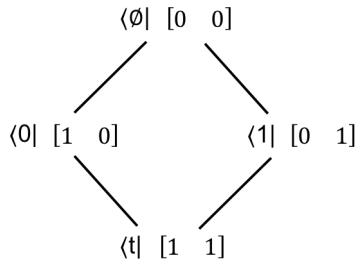

Figure 6.1. Hasse diagram of values in the switching domain

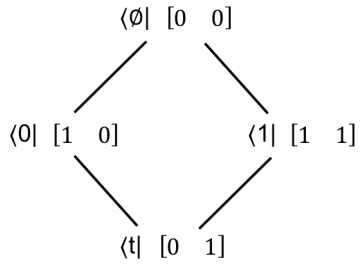

Figure 6.2. Hasse diagram of constant values in the ANF domain

6.2.2. Graph Traversal

Our methodology allows for a structural netlist to be modeled as a transfer function over Hilbert vector spaces of finite dimension rather than the more common model of collections of Boolean algebraic switching functions [14]. In this alternative formulation, individual gates are represented as transfer matrices, and the structural interconnection dictates whether the individual gate or operator transfer matrices are combined using either the direct matrix product or the outer matrix product. This model requires that binary values be modeled as

bra or ket vectors rather than constants in $\mathbb{B}$.

We represent the elements in $\mathbb{B}$ in the linear algebraic framework as elements in a finite-dimensioned Hilbert vector space, $\mathbb{H}$, where $\mathbb{B} = 0, 1$ and $\mathbb{H}_1 = 0, 1$. The notation $\langle 0|$ and $\langle 1|$ represent the row vectors or "bras" $[\langle 1| \langle 0|]$ and $[\langle 0| \langle 1|]$ respectively in accordance with the bra-ket notation of [9].

The ANF can be calculated from a transfer matrix representing a switching function in linear algebraic form by a direct matrix product of an appropriately dimensioned Reed-Muller transformation matrix followed by computing the modulus-2 of each component in the resultant vector [16].

Example 6.1 ANF Computation of circuit c17

Consider the example switching function of output n22 of benchmark circuit c17 whose transfer matrix is given in Equation 6.4. Since T_{n22} is of dimension 32×4, the appropriately dimensioned Reed-Muller transformation matrix is R_5. The direct product is given in Equation 6.4.

$$R_5 T_{n22} = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0\\
1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0\\
1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0\\
1&1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&1&1&1&1&0&0&0&0\\
1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0\\
1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0\\
1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0\\
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}
\begin{bmatrix}
1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1\\1&0\\1&0\\0&1\\0&1
\end{bmatrix} =
\begin{bmatrix}
1&0\\2&0\\1&1\\2&2\\2&0\\4&0\\2&2\\4&4\\2&0\\4&0\\2&2\\4&4\\4&0\\8&0\\4&4\\8&8\\2&0\\4&0\\2&2\\4&4\\4&0\\8&0\\5&3\\10&6\\3&1\\6&2\\3&5\\6&10\\6&2\\12&4\\7&9\\14&18
\end{bmatrix} \tag{6.3}$$

□

The resulting 32×2 matrix contains the ANF values when mapped into the linear al-

gebraic domain. The rightmost column vector of the resultant product matrix represents the ANF of the candidate function, and the leftmost represents the ANF of the complement of the candidate function. To obtain the actual ANF vector of the candidate function, the modulus-2 of the rightmost column vector is computed as shown below.

$$\begin{bmatrix} a_0 \\ a_5 \\ a_4 \\ a_{45} \\ a_3 \\ a_{35} \\ a_{34} \\ a_{345} \\ a_2 \\ a_{25} \\ a_{24} \\ a_{234} \\ a_{23} \\ a_{235} \\ a_{234} \\ a_{2345} \\ a_1 \\ a_{15} \\ a_{14} \\ a_{145} \\ a_{13} \\ a_{135} \\ a_{134} \\ a_{1345} \\ a_{12} \\ a_{125} \\ a_{124} \\ a_{1245} \\ a_{123} \\ a_{1235} \\ a_{1234} \\ a_{12345} \end{bmatrix} = \begin{bmatrix} 0 \mod 2 \\ 0 \mod 2 \\ 1 \mod 2 \\ 2 \mod 2 \\ 0 \mod 2 \\ 0 \mod 2 \\ 2 \mod 2 \\ 4 \mod 2 \\ 0 \mod 2 \\ 0 \mod 2 \\ 2 \mod 2 \\ 4 \mod 2 \\ 0 \mod 2 \\ 0 \mod 2 \\ 4 \mod 2 \\ 8 \mod 2 \\ 0 \mod 2 \\ 0 \mod 2 \\ 2 \mod 2 \\ 4 \mod 2 \\ 0 \mod 2 \\ 0 \mod 2 \\ 3 \mod 2 \\ 6 \mod 2 \\ 1 \mod 2 \\ 2 \mod 2 \\ 5 \mod 2 \\ 10 \mod 2 \\ 2 \mod 2 \\ 4 \mod 2 \\ 9 \mod 2 \\ 18 \mod 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{6.4}$$

To extract a single ANF value, we perform a traversal of the graph representing the circuit diagram. The technique involves interpreting the netlist in a hybrid form where individual gates are represented as an interconnection of small transfer matrices of each gate. As presented in figure 6.3, we can replace each logic gates by its corresponding matrix. A particular ANF coefficient is then computed by propagating (i.e., simulating) a particular variable assignment when propagated through the network [16]. However, the values must be transformed into the ANF domain. This is accomplished by inserting the R_1 matrix at each primary input. Figure 6.4 illustrates the example gate level diagram of c17 with each gate replaced by its corresponding transfer matrix and the primary inputs and outputs represented by appropriate matrices that enable the calculation of ANF coefficients. The following example illustrates how the netlist can be traversed to extract ANF coefficients.

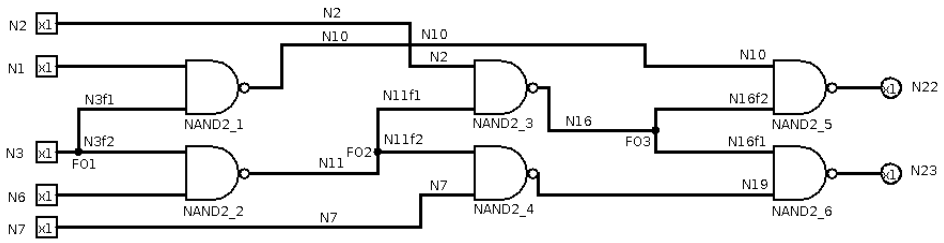

Figure 6.3. Benchmark circuit c17

The following figure shows the replacement of each logic gate by its corresponding transfer matrix.

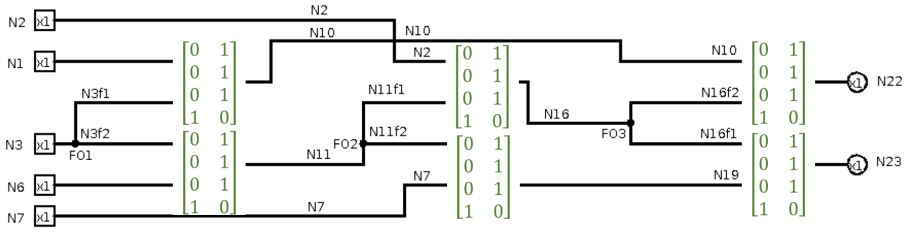

Figure 6.4. Example of a Hybrid Netlist for ANF Computation

Example 6.2 Extracting a_{12345} Coefficient from Netlist

In Figure 6.6 we extract the ANF coefficient a_{12345} by initially assigning the input values $\langle x_1x_2x_3x_4x_5| = \langle 11111|$ and prepending the netlist primary inputs with the Reed-Muller matrix R_5 to ensure that the input value assignments are transformed to the ANF domain. The input assigned values are then propagated through the network by multiplying each row vector with the transfer matrix encountered. For those transfers matrices with j multiple inputs, the individual values on each line are combined into a single vector of dimension 2^j (i.e., a vector in $\mathbb{H}_j$) through the use of the outer product operator $\otimes$. Because the outer product operation is not commutative, it is important to order the operands by using the topmost value in the netlist as the leftmost factor and the bottommost value in the netlist as the rightmost factor. The values that are propagated through the netlist are highlighted in red font.

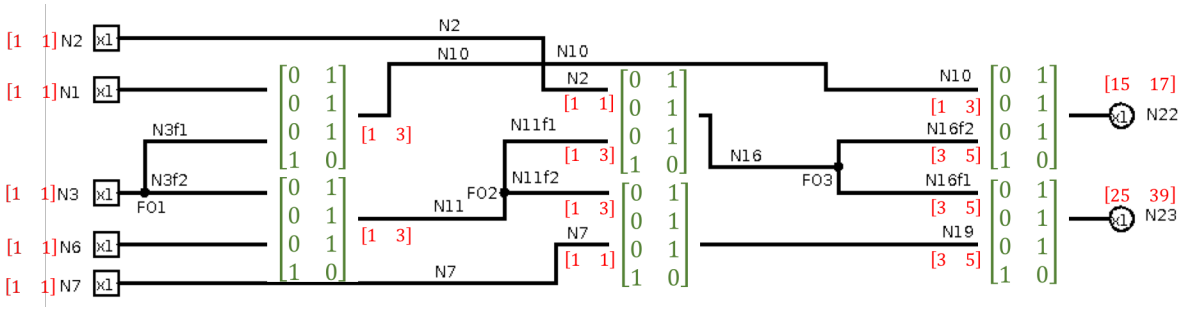

Figure 6.5. Example of a Hybrid Netlist for ANF Computation a_{12345}

□

Example 6.3 : Extracting a_{124} Coefficient from Netlist

In Figure 6.6 the ANF coefficient a_{124} is extracted by initially assigning the input values $\langle x_1 x_2 x_4| = \langle 11010|$ and prepending the netlist primary inputs with the R_5 matrix to ensure that the input value assignments are transformed to the ANF domain. The input assigned values are then propagated through the network by multiplying each row vector with the transfer matrix encountered. The values that are propagated through the netlist are shown in red font.

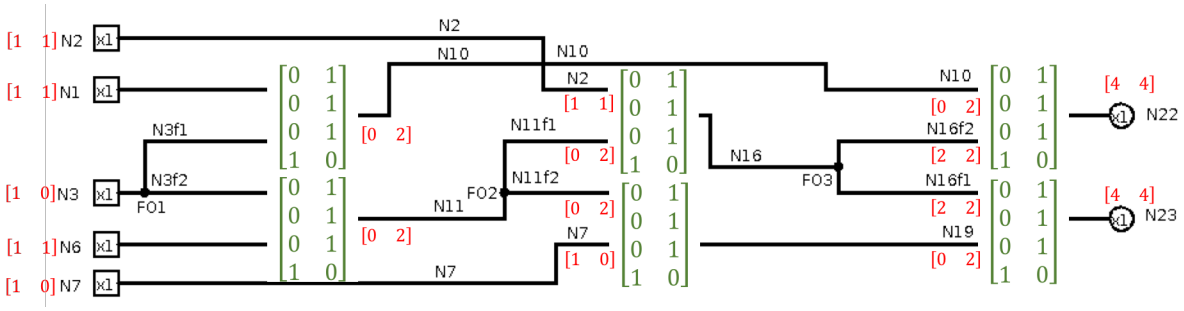

Figure 6.6. Example of a Hybrid Netlist for ANF Computation a_{124}

□

6.3. Computation of the Maximum Algebraic Degree

To determine the maximum algebraic degree of a netlist, we propagate multiple input variable assignments through the network. For each assignment of n input variables, we set k variables to $\langle 1|$ with k representing the algebraic degree. The number of test cases follows a binomial distribution. The i^th row of the Pascal's triangle is the number of combinations C_k (n choose k) with i going from 1 to n. The binomial coefficients ${}^nC_k$ represent the number of variable assignments required for each degree k computation.

$$\binom{n}{k} = {}^nC_k = \frac{n!}{k!\,(n-k)!} \tag{6.5}$$

Figure 6.7 shows the distribution of ANF coefficients in the Pascal Triangle for the 5 variables of circuit c17.

VARIABLES											
0						**1**					
1					**1**		**1**				
2				**1**		**2**		**1**			
3			**1**		**3**		**3**		**1**		
4		**1**		**4**		**6**		**4**		**1**	
5	**1**		**5**		**10**		**10**		**5**		**1**
	0		1		2		3		4		5
	ANF DEGREE										

Figure 6.7. ANF coefficients in the Pascal Triangle for circuit c17

To reduce the computation time of the degree search, one preliminary step is to prune the search space by identifying all input variables that the output depends on. The pruning method consists of performing n traversals, where n is the number of input variables. The simulations are performed in the switching domain using a combination of the total vectors $\langle t|$ and the null vector $\langle \emptyset|$. Example 6.4 shows how to find the dependent input variables among x_1, x_2, x_3, x_4, and x_5 for an output f.

Example 6.4 : Extracting dependent input variables for output f

Assuming the netlist has five input variables x_1, x_2, x_3, x_4, and x_5 each of the input

variables is assigned to $\langle\emptyset|$, one at a time, while the remaining variables are assigned to $\langle t|$. The advantage of the pruning method is that it runs in linear time. For $n = 5$ variables we perform five simulations successively in the switching domain by assigning the following input values to the input vectors:

$$\langle x_1x_2x_3x_4x_5| = \langle \emptyset tttt|$$

$$\langle x_1x_2x_3x_4x_5| = \langle t\emptyset ttt|$$

$$\langle x_1x_2x_3x_4x_5| = \langle tt\emptyset tt|$$

$$\langle x_1x_2x_3x_4x_5| = \langle ttt\emptyset t|$$

$$\langle x_1x_2x_3x_4x_5| = \langle tttt\emptyset|$$

For each set of assigned values, if the output f evaluates to the null vector $\langle\emptyset|$, then the f depends on the variable x_i assigned to $\langle\emptyset|$. □

Figure 6.8 shows the traversal of c17 for the variable in $\langle\emptyset tttt|$ in the switching domain:

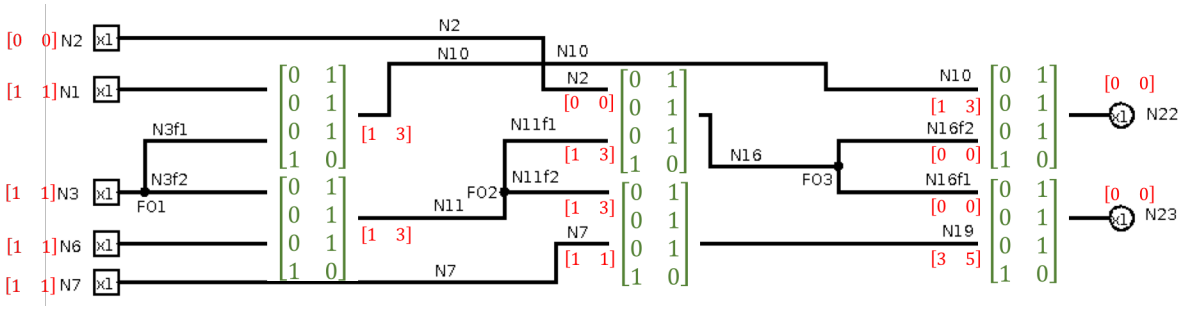

Figure 6.8. Example of a Hybrid Netlist for ANF Computation

After the traversal, both outputs N_{22} and N_{23} of c17 become 0; this means that outputs N_{22} and N_{23} depend on input N_2.

6.3.1. Binomial Distribution of ANF Coefficients

The overall degree of a particular switching function is the maximum degree monomial present in the ANF representing that function. To find the maximum degree, we must perform multiple traversals using multiple combinations of variable assignments. The number of

elements in each equivalence class, or the cardinality, is a set of integers that are binomially distributed. We collected the runtime to generate 2^n coefficients for circuits with n coefficients. The binomial distribution is a discrete probability distribution; therefore, a random switching function should have ANF coefficients a_i that are binomially distributed.

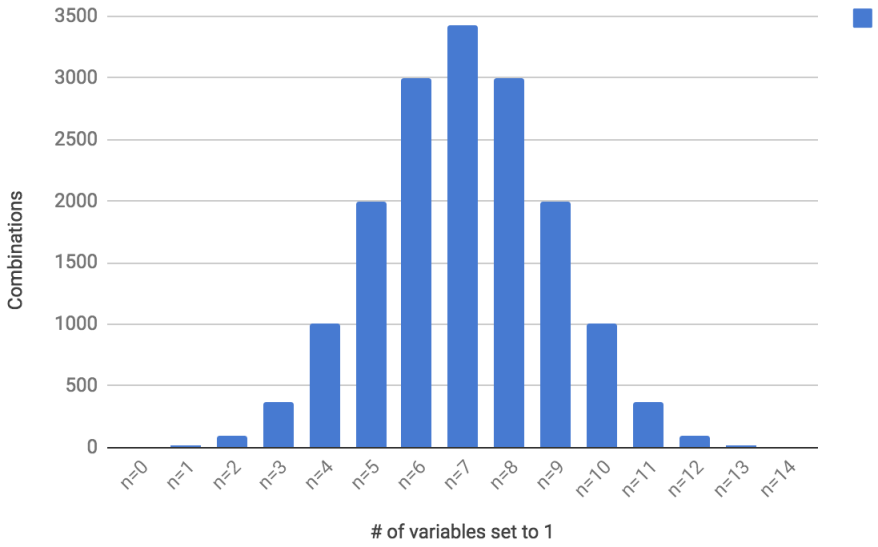

Figure 6.9. Binomial distribution

Figure 6.9 shows the distribution of the number of test cases required for a graph with n variables. When the number of input variable set to $\langle 1|$ approaches $\frac{n}{2}$ the number of combinations gets larger. The pruning method presented can help improve the maximum degree search time by reducing the number of variables. Figure 6.10 shows the binomial distribution after pruning with a 50% variable reduction for the same circuit.

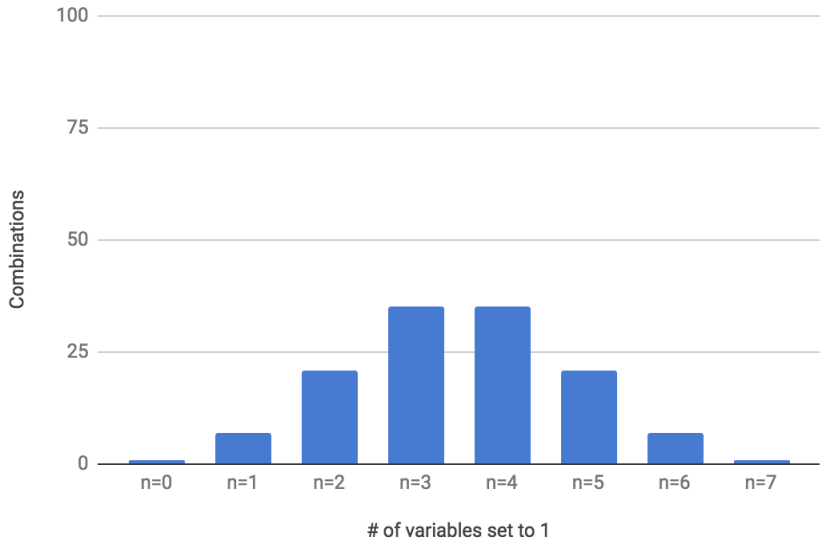

Figure 6.10. Binomial distribution with a 50% variable reduction

6.3.2. Experimental Results

The experimental results in Table 6.1 show the maximum algebraic degrees for some outputs of ISCAS85 benchmark circuits. The reduction scheme allows to lower the number of variable assignments before proceeding to the degree search. Only the dependent variables of the tested output are changed, while all the independent variables remain fixed to one. The reduction of the number of variable assignments has allowed noticeable timing improvement on some outputs.

Table 6.1. Computation of the maximum algebraic degree

Benchmark circuit	Inputs/ Outputs	#gates	Output	Maximum degree	Degree search time (ms)	Optimized degree search time (ms)	Speedup
majority.v	5/1	13	o0	5	2.69	2.26	16%
test1.v	3/3	14	o2	n/a	n/a	n/a	n/a
rd53.v	5/3	22	o1	1	32.03	31.41	2%
con1.v	7/2	28	f0	7	4.04	2.92	28%
radd.v	8/5	33	o2	6	59.09	55.12	7%
cm163a.v	16/5	45	t	15	50.29	48.06	4%
dk17.v	10/11	154	_o0	6	2456.93	2453.89	0%
pcle.v	19/9	61	v	18	25.7	24.69	4%
mux.v	21/1	55	v	21	10.92	10.71	2%
cm85a.v	11/3	59	_o1	10	55.57	54.45	2%
x2.v	10/7	60	l	9	7.67	7.23	6%
sct.v	19/15	100	b0	18	64.67	63.15	2%
misex2.v	25/18	130	z	25	4.87	2.71	44%
alu2b.v	10/8	142	_o2	9	66.16	65.89	1%
c432.v	36/7	160	n223	19	19210.74	3146.91	84%
c499.v	41/32	202	N724	41	110.32	106.22	4%
c880a.v	60/26	383	n388	60	0.96	0.889	7%
c1355.v	41/32	546	g1324	41	89.12	85.12	4%
c1908	33/25	880	n2753	33	140.52	138.47	1%
c2670	233/140	1269	n398	233	1.6	1.47	8%
c3540	50/22	1669	n1713	50	2.84	2.72	4%
c5315	178/123	2307	n709	178	3.15	2.8	11%
c6288	32/32	2416	N545	32	1.58	0.55	65%
c7552	207/108	3513	N387	207	4.2	3.84	9%

Chapter 7

CONCLUSION

We have described how we can use the theory in [27] to efficiently manipulate switching circuits using BDDs and ADDs. Our approach consists of representing a digital logic network as a transfer function using a sparse matrix or a decision diagram. To obtain an output response for a simulation, we represent the input stimulus as a vector, and the transfer function can linearly transform that input vector into an output vector. The advantage of computing transfer functions is that we can use the same function framework for both digital network simulation and the reverse process, justification. Our method represents the resulting transfer function as a binary decision diagram by merging all the BDD partitions either by multiplication or composition operations. The attributes of these diagrams are the ability to have a more compressed representation of the transfer function especially when dealing with large netists, the ability to use multiple variable reordering algorithms, and the benefits of a smaller memory footprint.

Two versions of the simulator were implemented with one optimizing runtime and the other optimizing memory usage. In order to make use of the theoretical results of [27] in a practical manner, ADDs are used to represent the matrices and vectors. A new tensor multiplication algorithm was formulated as an operation over matrices represented as ADDs and was shown to be very efficient in that it only required the traversal of a single path in each of the two operands ADDs. This tensor multiplication algorithm enabled the two candidate simulation methods to have reasonable and competitive runtimes and memory usage statistics. These results indicate that the linear algebraic theory can be used as a practical and reasonable alternative to conventional switching algebra models for digital circuit EDA tools. We also demonstrated how to adapt our model to simulate sequential circuits by performing a simulation for each sequential input. The complexity remains the same for

combinational and sequential circuits with the sequential circuit requiring a simulation to be performed for each new input vector, just as is the case for combinational circuits.

ANF (Algrebraic Normal Form) coefficients are very useful in analyzing switching function models of cryptographic primitives. They can be used to determine how closely the function approximates a random function or the linearity. A method for the calculation of ANF coefficients both as a complete set or as individual coefficients through a traversal of a structural netlist model of a cryptographic switching function is described. Furthermore, we have described a method for estimating the ANF coefficients when the candidate switching function is only partially known. Input/output observation pairs of a candidate and unknown switching function may be used to construct a netlist that estimates the switching function of interest.

Future research involves applying the vector space method to other applications common in EDA tools. An immediate area of research is to incorporate timing into the switching circuit models. This can potentially be accomplished by modifying the transfer matrix elements to contain values corresponding to time delays rather than the value '1.'

One very promising avenue for future work involves incorporating this approach into mixed-signal simulators. Currently mixed-signal simulators use a hybrid approach where the digital portion uses conventional discrete event simulation and the analog portion uses traditional techniques such as SPICE. Then these two often disparate simulation results must somehow be combined into a single output. The vector space method offers the possibility to produce a truly unified approach since it is based upon the use of transfer functions which is a common model for analog circuitry. Thus the potential to produce an overall mixed-signal transfer function and hence a unified simulation process is promising.

Appendix A

Transfer function: xor5.v

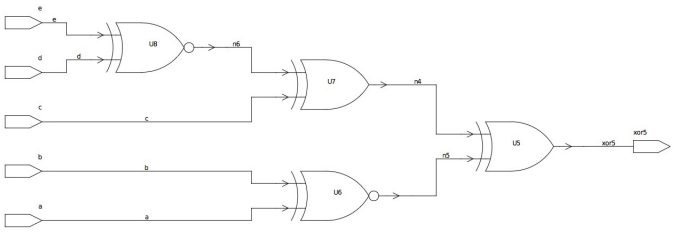

Figure A.1. xor5.v schematic

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Figure A.2. xor5.v matrix

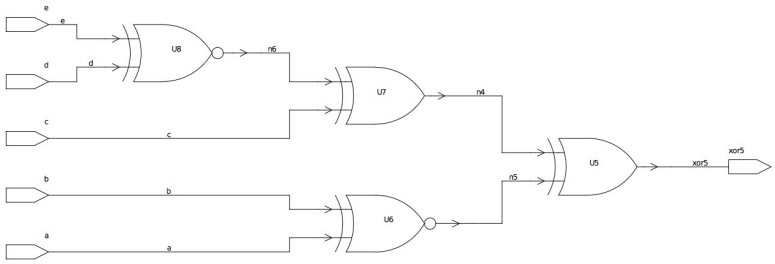

Figure A.3. xor5.v schematic

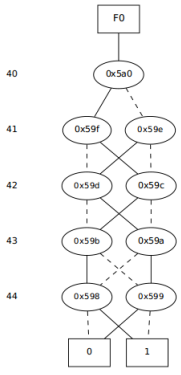

Figure A.4. xor5.v ADD

Appendix B

Transfer function: majority.v

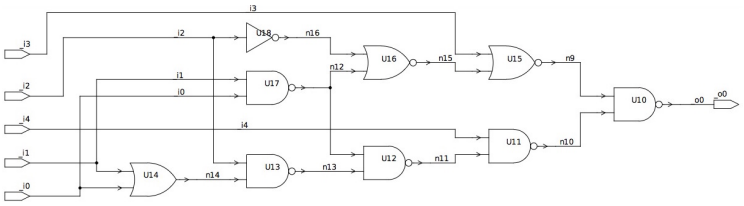

Figure B.1. majority.v schematic

$$F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Figure B.2. majority.v matrix

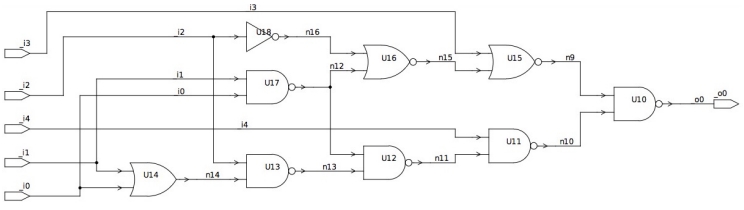

Figure B.3. majority.v schematic

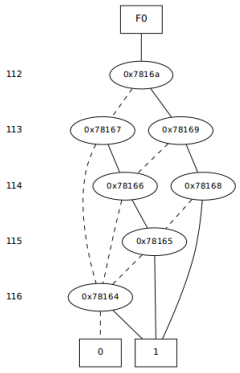

Figure B.4. majority.v ADD

Appendix C

Transfer function: c17.v

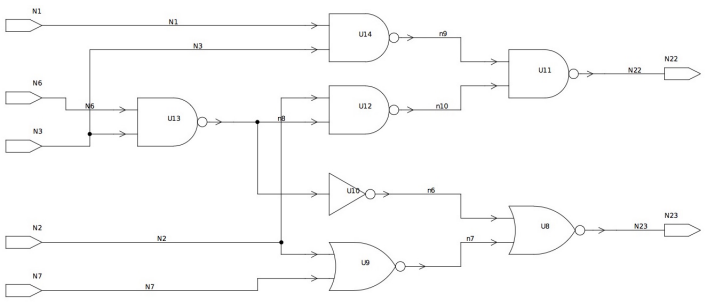

Figure C.1. c17.v schematic

$$F = \begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 1&0&0&0 \\ 1&0&0&0 \\ 1&0&0&0 \\ 1&0&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \\ 0&0&0&1 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&0&1&0 \\ 0&0&1&0 \\ 0&0&1&0 \end{bmatrix}$$

Figure C.2. c17.v matrix

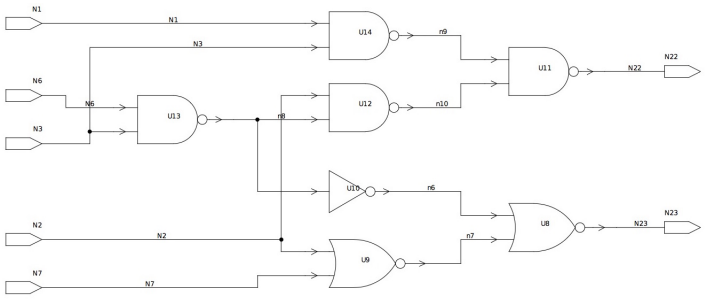

Figure C.3. c17.v schematic

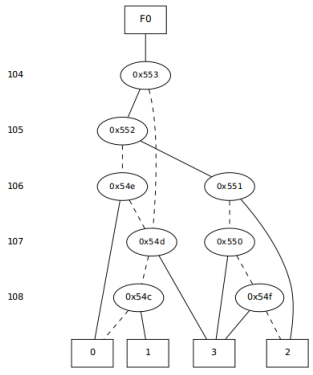

Figure C.4. c17.v ADD

Appendix D

Transfer function: rd53.v

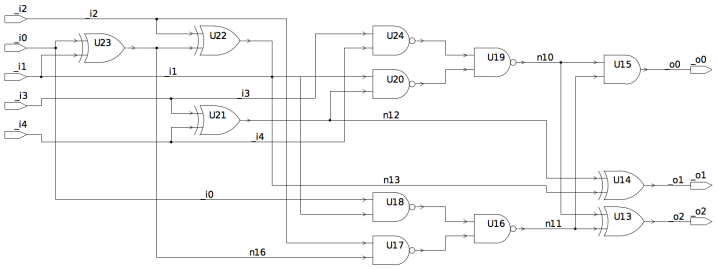

Figure D.1. rd53.v schematic

$$F = \begin{bmatrix}
1&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0\\
0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0\\
0&1&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0
\end{bmatrix}$$

Figure D.2. rd53.v matrix

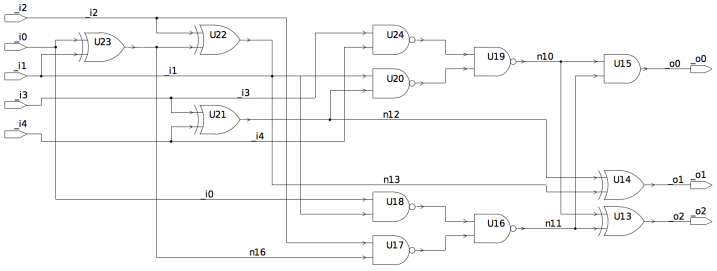

Figure D.3. rd53.v schematic

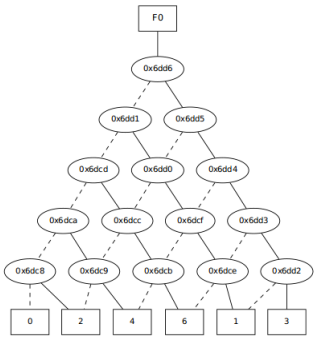

Figure D.4. rd53.v ADD

Appendix E

Transfer function: radd.v

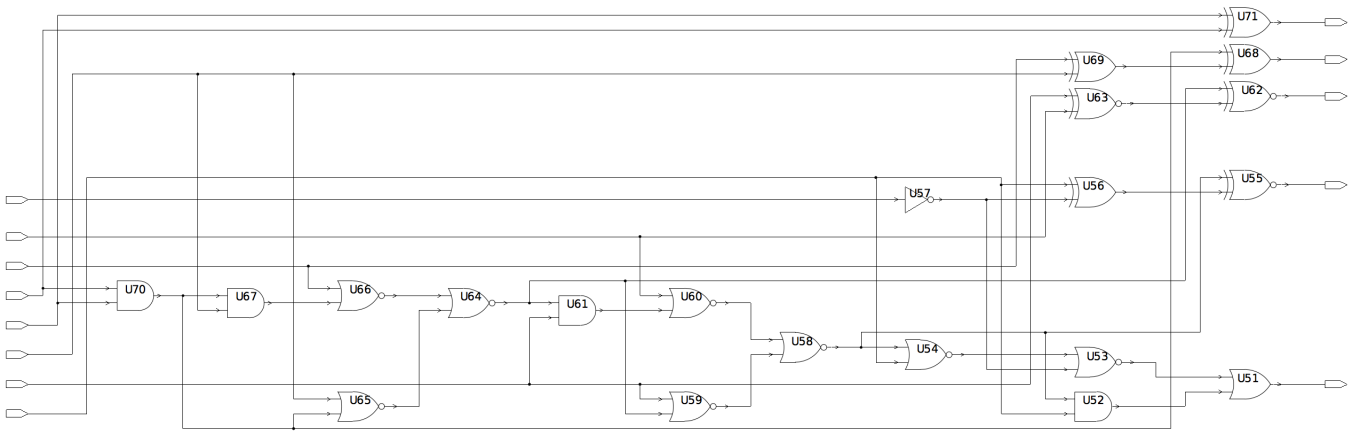

Figure E.1. radd.v schematic

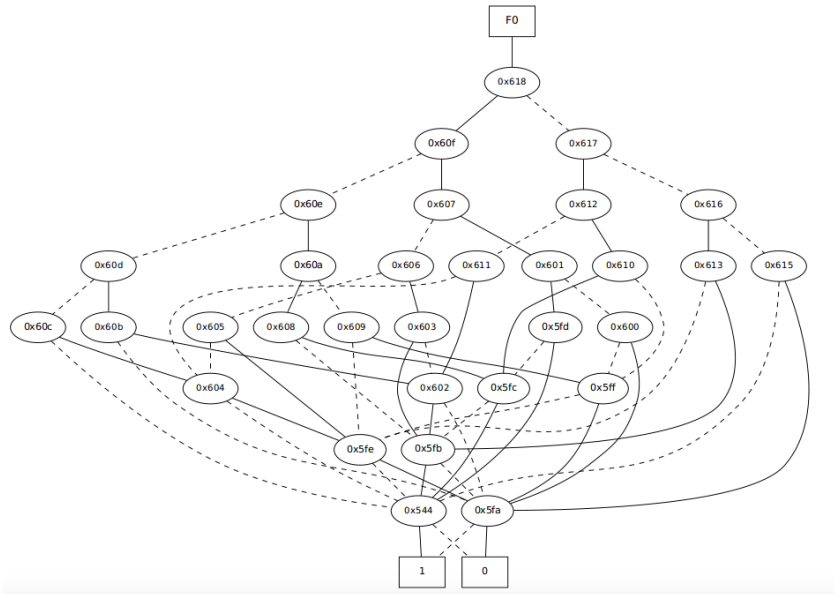

Figure E.2. radd.v Output o4

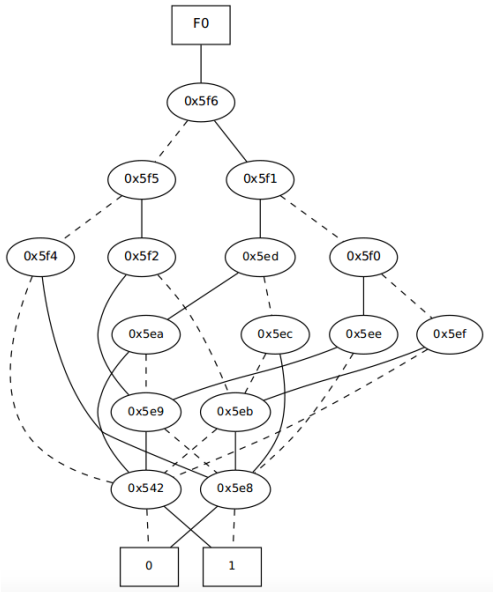

Figure E.3. radd.v Output o3

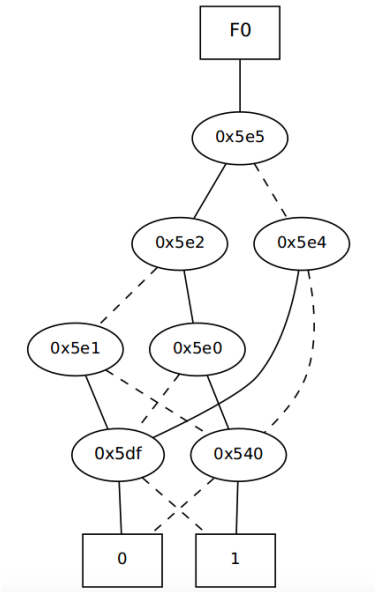

Figure E.4. radd.v Output o2

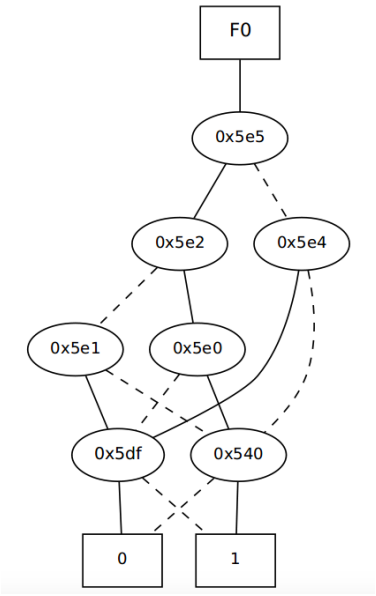

Figure E.5. radd.v Output o1

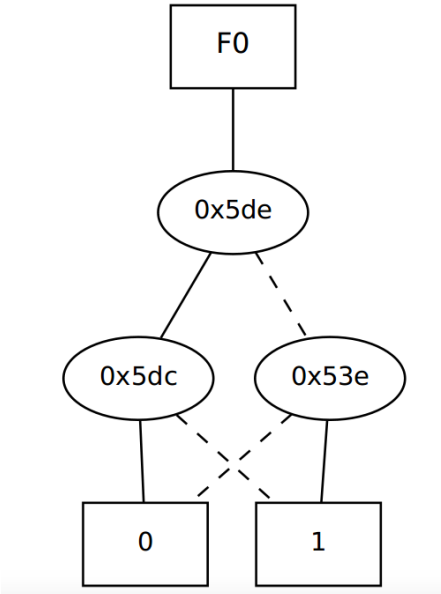

Figure E.6. radd.v Output o0

Appendix F

Transfer function: i3.v

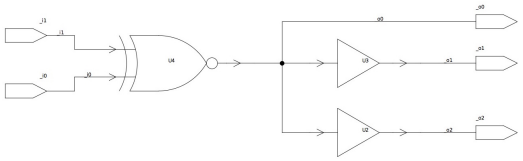

Figure F.1. i3.v schematic

$$F = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure F.2. i3.v matrix

Appendix G

Code listing

```
/**
 * Main program
 * @param argc, *argv[]
 */
int main (int argc, char *argv[])
{
    int input_row=0, output_row=0, reed_muller=0, round=2, i=0, j=0;
    int id=0, crossover_mode=0, option=0, simulation=-1, justification=0;
    int n, vars, found;
    int inputs[99];
    int mode = MATRIX_MODE; /*By default the mode is MATRIX_MODE*/
    char filename[30], verilog[100], format[50];
    struct timeval startTime;
    double endTime=0, partitionsTime=0, ddTime=0, simulationTime=0, justificationTime=0,
         cumulativeTime=0;
    long rows=0, decimal=0;
    kroneckerTime=0, reorderingTime_arr=0, simulationTime_arr=0;
    debug=false; /*Testing mode*/
    verbosity=4; /*Level of details in DD printing*/
    DdNode *output;
    DdNode *tmp=NULL;
    coefficient* anf_coefficients;

    if (argc != 2) {
        printf("Usage: ./transfer <config file>\n"); /* Check for a verilog input file */
        //exit(1);
    }
```

```
read_config(argv[1], verilog, format, &option, &input_row, &output_row, &crossover_mode,
    &simulation, &justification, &reed_muller); /*Read a configuration file from the command
    line*/
printf("\n**** CONFIGURATION ****\n \tVerilog file: %s\n \tTransfer function format: %s\n
    \tSimulation type: %d\n \tInput row: %d\n********************\n\n", verilog, format,
    simulation, input_row);

circuit c = (circuit)calloc(1,sizeof(struct circuit_)); /*Declare an instance of a circuit */
c->name = strdup(verilog);                              /*Set circuit name*/
system("exec rm -r cascades/* inputs/* transparent/* add/* bdd/*"); /*Clean up directories*/

if (strcmp(format, "dd") ==0) /*In mode 1 no matrix is created*/
    mode = 1;

parse_verilog_file (c, c->name, mode);  /*Parse the verilog file */

levelizer_r1 (c);         /*Levelize the circuit round #1*/
while (!levelization_completed (c)) {  /*Keep levelizing until full completion*/
    levelizer_rn (c, round);  /*Levelize the circuit fully with n rounds*/
    round++;
}

gettimeofday(&startTime, NULL); // Start timer
partitions (c); /*Create an array of partitions*/
stopTimer (&partitionsTime, startTime); // Stop timer

gbm = Cudd_Init(0,0,CUDD_UNIQUE_SLOTS,CUDD_CACHE_SLOTS,0); /* Initialize a new BDD manager. */

Cudd_ReduceHeap(gbm, CUDD_REORDER_SYMM_SIFT, 3000); // Dynamic reordering by sifting method

Cudd_SetBackground(gbm, Cudd_ReadPlusInfinity(gbm)); // Change the background to infinity
    (Default value was zero)

switch (simulation) {

case DEGREE_SEARCH:
  vars = c->inputcount;         // Number of input variables
```

```
  int out = atoi(argv[2]);      // Output under test
  int k = atoi(argv[3]);        // Degree
  int *active = (int *) malloc(vars * sizeof(int)); // Array of active inputs

  /* OPTIMIZED DEGREE SEARCH */

  for (out=0; out < c->outputcount; out++){          // Perform a degree search on each output
     found =0;
     int active_vars = reduce_search_space (c, out, active);   // reduce the search space for
         each output
     k = active_vars;                              // Set the degree for the search equal to the
         number of active variables
     cumulativeTime = 0;                             // Reset time

     gettimeofday(&startTime, NULL);    // Start the timer for simulation
     for (i=k; i>0; i--) {
        found = degree_search_opt (c, active_vars, i, active, out);  // Degree search from
            higher degrees
        fprintf(stderr, "******* FOUND = %d ***********\n\n", found);
        if (found) break;
     }
     stopTimer(&cumulativeTime, startTime);  // Stop the timer for simulation
     fprintf(stderr,"Cumulative time: %.6f ms\n\n\n", cumulativeTime);
  }

  /* END OPTIMIZED DEGREE SEARCH */

  free (active);  // Free memory
break;


case MATRIX_MONOLITHIC:
    matrix_direct_product (c); /*Direct matrix product of each cascade stages*/
    matrix inputVector = (matrix)malloc(sizeof(struct matrix_)); /*Input vector for the
        simulation*/
    build_vector (inputVector, c->inputcount, input_row); /*Build input row vector*/
    gettimeofday(&startTime, NULL); // Start timer
```

```
    direct (inputVector, c->transfer); /*Simulation: multiply the input row vector by the
        transfer function matrix*/
    stopTimer(&simulationTime, startTime); // Stop timer

    if (justification==MATRIX_JUSTIFICATION) {
        matrix transposeMatrix = (matrix)malloc(sizeof(struct matrix_)); /*Transpose matrix*/
        transpose (c->transfer, transposeMatrix); // Build the transpose matrix
        matrix outputMatrix = (matrix)malloc(sizeof(struct matrix_)); /*Output matrix*/
        build_vector (outputMatrix, c->outputcount, input_row);
        gettimeofday(&startTime, NULL); // Start timer
        direct (outputMatrix, transposeMatrix); /*Multiply the input row vector by a transfer
            matrix*/
        stopTimer(&simulationTime, startTime); // Stop timer
        free_matrix (transposeMatrix, transposeMatrix->rows); /*Free the transpose matrix*/
        free_matrix (outputMatrix, outputMatrix->rows); /*Free the output matrix*/
    }

    free_matrix (c->transfer, ROWS); /*Free the transfer matrix*/
    free_matrix (inputVector, inputVector->rows); /*Free the input matrix*/
    break;

case MATRIX_DEBUG:
    gettimeofday(&startTime, NULL); // Start timer
    debug=true;
    matrix_direct_product (c); /*Direct matrix product of each cascade stages*/
    stopTimer(&simulationTime, startTime); // Stop timer
    break;

case DD_MONOLITHIC:
    gettimeofday(&startTime, NULL); // Start timer
    transferFunction (c, DD_MONOLITHIC, 0, crossover_mode); /*Build the entire transfer
        function DD first*/
    stopTimer(&ddTime, startTime); // Stop timer
    gettimeofday(&startTime, NULL); // Start timer
    output = Cudd_addMultiply(gbm, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)input_row),
        c->transfer_dd); /*Simulation: Get the corresponding terminal constant in the DD
        Transfer function DD*/
```

```
    stopTimer(&simulationTime, startTime); // Stop timer
    print_all (output, "./add/output.dot", 4,"\n\nOutput response (Monolithic method) \n");

    if (justification==DD_JUSTIFICATION) {
        gettimeofday(&startTime, NULL); // Start timer
        Cudd_ddJustification(gbm, c->transfer_dd, output_row);
        stopTimer(&justificationTime, startTime); // Stop timer
    }

    if (NULL != c->transfer_dd)
        Cudd_RecursiveDeref(gbm, c->transfer_dd);
    if (NULL != output)
        Cudd_RecursiveDeref(gbm, output);
    break;

case DD_ARRAY1:
    gettimeofday(&startTime, NULL); // Start timer
    transferFunction (c, DD_ARRAY1, input_row, crossover_mode); /*Build the output response
        incrementally*/
    stopTimer(&simulationTime, startTime); // Stop timer
    if (NULL != c->transfer_dd)
        Cudd_RecursiveDeref(gbm, c->transfer_dd);
    printf ("\n >>>>>>>>>>>>>>>>>>>>. %.6f ms \n", kroneckerTime);
    ddTime = kroneckerTime + reorderingTime_arr;
    simulationTime = simulationTime_arr;
    break;

case DD_ARRAY2:
    csc=0;
    nodeArray = (DdNode **)calloc(999, sizeof(DdNode*));
    output = Cudd_addConst (gbm, (CUDD_VALUE_TYPE)input_row);
    gettimeofday(&startTime, NULL); // Start timer
    transferFunction (c, DD_ARRAY2, 0, crossover_mode); /*Build the output response
        incrementally*/
    stopTimer(&ddTime, startTime); // Stop timer
    gettimeofday(&startTime, NULL); // Start timer
    for (i=0; i < csc; i++) {
```

```
            tmp = Cudd_addMultiply(gbm, output, nodeArray[i]);
            Cudd_Ref(tmp);
            Cudd_RecursiveDeref(gbm, output);
            Cudd_RecursiveDeref(gbm, nodeArray[i]);
            output = tmp;
        }
        stopTimer(&simulationTime, startTime); // Stop timer
        print_all (output, "./add/output.dot", 4,"\n\nOutput response (Array 2 method) \n");
        free(nodeArray);
        if (NULL != c->transfer_dd)
            Cudd_RecursiveDeref(gbm, c->transfer_dd);
        if (NULL != output)
            Cudd_RecursiveDeref(gbm, output);
        break;

    case DD_DISTRIBUTED:

        switch (option) {

        case CONSTANT: // Set inputs to be constants (Simulation)
            for (i=0; i < c->inputcount; i++) {// Create BDD variables for the inputs.
                inputs[i] = 1;            // Temporary set all inputs to 1
                id = getID(c->inputs[i], c); // Get the ID number of an input variable
                printf("Input wire name: %s \n", getWire(id,c)->name);
                getWire(id,c)->dd = (inputs[i] == 0) ? Cudd_ReadZero(gbm) : Cudd_ReadOne(gbm); //
                    Input is either constant 0 or constant 1
                Cudd_Ref(getWire(id,c)->dd);
                sprintf(filename, "./inputs/input_%d.dot", i);
                print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 4, "Input");
            }

            gettimeofday(&startTime, NULL); // Start timer
            distributed_dd (c); /*Crawl in the circuit, starting from the primary inputs*/
            stopTimer(&simulationTime, startTime); // Stop timer

            for (i=0; i < c->outputcount; i++) {
                id = getID(c->outputs[i], c); // Get the ID number of an output variable
```

```
        printf("\nOutput wire name: %s \n", getWire (id,c)->name);
        sprintf(filename, "./bdd/distributed_bdd_%d.dot", i);
        print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 4, "Distributed
            method"); // Convert the BDD to an ADD
    }
    break;

case VARIABLE: // Set inputs to be variables
    /*Create a DD variable for all the primary inputs*/
    for (i=0; i < c->inputcount; i++) {
        id = getID(c->inputs[i], c); // Get the ID number of an input variable
        printf("Input wire name: %s \n", getWire(id,c)->name);
        getWire(id,c)->dd = Cudd_bddNewVar(gbm); // Create a unique DD node representing a
            primary input variable
        Cudd_Ref(getWire(id,c)->dd);
        sprintf(filename, "./inputs/input_%d.dot", i);
        print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 4, "Input");
    }
    /*End Create a DD variable for all the primary inputs*/

    /*Build a DD for each output of the circuit*/
    gettimeofday(&startTime, NULL); // Start timer
    distributed_dd (c); /*Crawl in the circuit, starting from the primary inputs*/
    stopTimer(&simulationTime, startTime); // Stop timer
    /*End Build a DD for each output of the circuit*/

    for (i=0; i < c->outputcount; i++) {
        id = getID(c->outputs[i], c); // Get the ID number of an output variable
        printf("\nOutput wire name: %s \n", getWire (id,c)->name);
        sprintf(filename, "./bdd/distributed_bdd_%d.dot", i);
        print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 4, "Distributed
            method"); // Convert the BDD to an ADD
    }
    break;

default:
    break;
```

```
    }
    /*End distributed traversal*/

    fprintf(stderr,"\n\nSimulation time: %.6f ms\nCumulative time: %.6f ms\n\n",
        simulationTime, cumulativeTime);
    break;

case DD_RECURSIVE:
    for (i=0; i < c->inputcount; i++) {// Create BDD variables for all the primary inputs.
        id = getID(c->inputs[i], c); // Get the ID number of an input variable
        printf("Input wire name: %s \n", getWire(id,c)->name);
        getWire(id,c)->dd = Cudd_bddNewVar(gbm); // Create a unique DD node representing a
            primary input variable
        Cudd_Ref(getWire(id,c)->dd);
        sprintf(filename, "./inputs/input_%d.dot", i);
        print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 4, "Input");
    }
    printf("\n\n******** Creating BDD recursively starting from the outputs ********\n\n");
    for (i=0; i < c->outputcount; i++) {
        id = getID(c->outputs[i], c); // Get the ID number of an output variable
        printf("\n\n\n---- Output wire name: %s ----\n", getWire (id,c)->name);
        gettimeofday(&startTime, NULL); // Start timer
        recursive_bdd (getWire(id,c), c); // Build the BDD recursively
        stopTimer(&endTime, startTime); // Stop timer
        ddTime += endTime; // Accumulate DD computation time
        sprintf(filename, "./bdd/bdd_%d.dot", i);
        print_all (Cudd_BddToAdd(gbm, getWire(id,c)->dd), filename, 2, "Recursive method"); //
            Convert the BDD to an ADD
    }
    printf("\n\n******** End of creating BDD recursively starting from the outputs
        ********\n\n");

    if (justification==DD_JUSTIFICATION) {
        print_all (getWire(id,c)->dd, filename, 4, "getWire(id,c)->dd");
        gettimeofday(&startTime, NULL); // Start timer
        Cudd_ddJustification(gbm, getWire(id,c)->dd, output_row);
        stopTimer(&justificationTime, startTime); // Stop timer
```

```
    }
    break;

case SEQUENTIAL:
    transferFunction (c, DD_MONOLITHIC, 0, crossover_mode); /*Build the entire transfer
        function DD first*/
    int cycles = 4;
    int invars = c->inputcount;
    int outvars = c->outputcount;
    int *seq_inputs = (int *) malloc(invars * sizeof(int)); // Inputs
    int *seq_outputs = (int *) malloc(outvars * sizeof(int)); // Outputs

    getBinary (input_row, invars, seq_inputs, 0);   // Convert a row to binary

    for (i=0; i < cycles; i++) {
      output = Cudd_addMultiply(gbm, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)input_row),
          c->transfer_dd); /*Simulation: Get the corresponding terminal constant in the DD
          Transfer function DD*/
      getBinary ((int) cuddV(output), outvars, seq_outputs, 0);
      seq_inputs[0] = seq_outputs[1]; // Update inputs

      input_row = getDecimal (invars, seq_inputs);
      print_all (output, "./add/output.dot", 4,"\n\nOutput response (Monolithic method) \n");
  }

    if (NULL != c->transfer_dd)
        Cudd_RecursiveDeref(gbm, c->transfer_dd);
    if (NULL != output)
        Cudd_RecursiveDeref(gbm, output);

    free (seq_inputs);
    free (seq_outputs);
    break;

case DD_CASCADES_ONLY:
    c->transfer_dd = Cudd_ReadZero(gbm);
```

```
    transferFunction (c, DD_CASCADES_ONLY, 0, 0); /*Build the entire transfer function DD
        first*/
    break;

default:
    printf("\n\n NO OPERATION PERFORMED \n\n"); /* Check for a verilog input file */
}

printf("\nNodes in the unique table: %d | Manager memory: %lu bytes | Nodes with non-zero
    reference counts: %d\n", Cudd_ReadKeys(gbm), Cudd_ReadMemoryInUse(gbm),
    Cudd_CheckZeroRef(gbm));
info_dd (gbm, "./info.txt"); //Print out statistics and settings for a CUDD manager
Cudd_Quit(gbm);/*Shut down the DdManager*/

printf("\n Partitioning time: %.6f ms\n DD building time: %.6f ms\n Simulation time: %.6f ms\n
    Justification time: %.6f ms\n\n", partitionsTime, ddTime, simulationTime,
    justificationTime);
fprintf(stderr,"\n\nSimulation time: %.6f ms\nCumulative time: %.6f ms\n\n", simulationTime,
    cumulativeTime);

/*Dereferencing and deallocations*/
for (i=0; i < c->outputcount; i++)
    free (c->outputs[i]);

for (i=0; i < c->inputcount; i++)
    free (c->inputs[i]);

for (i=0; i < c->nodecount; i++)
    free (c->nodes[i]);
free(c->nodes);

for (i=0; i < c->wirecount; i++) {
    if (mode==MATRIX_MODE) { // Only in matrix mode, free the allocated matrix and vector
        free_matrix(c->wires[i]->m, c->wires[i]->m->rows); /*Free the wire matrix*/
        free_matrix(c->wires[i]->v, c->wires[i]->v->rows); /*Free the wire vector*/
    }
    free (c->wires[i]->name);
```

```
        free (c->wires[i]->type);
        free (c->wires[i]);
    }
    free(c->wires);

    for (i=0; i < STATIC_ARR_MAX; i++) {
        for(j=0; j < STATIC_ARR_MAX; j++) {
            free (c->crossovers[i][j]);
        }
    }

    for (i=0; i < STATIC_ARR_MAX; i++) {
        for(j=0; j < STATIC_ARR_MAX; j++) {
            if(NULL != c->partition[i][j]) {
                free (c->partition[i][j]->name);
                free (c->partition[i][j]->type);
            }
            free (c->partition[i][j]);
        }
    }

    free (c->name);
    free (c); /*Deallocate memory used by the circuit*/

    /*End Dereferencing and deallocations*/

    return 0;
}



/**Function********************************************************************

  Synopsis    [Create a ADD from different sizes of fanouts]

  Description [The Else branch is always a zero (0)
   Depending on the number of fanouts the Then branch varies between: (3 or 7 or 15 or 31 etc.)]
```

```
  @param the number of fanout branches
******************************************************************************/
DdNode * Cudd_fanout (int fanouts)
{
    DdNode *var = Cudd_addNewVar(gbm); /*Root node of the fanout*/
    Cudd_Ref(var);
    int coef = pow(2, fanouts)-1; /*Calculate the max value depending on the number of fanouts*/
    DdNode *retval = Cudd_addIte(gbm, var, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)coef),
        Cudd_addConst(gbm, (CUDD_VALUE_TYPE)0)); /*Else branch=0, Then branch=(3 or 7 or 15 or 31
        etc.)*/
    Cudd_RecursiveDeref(gbm,var);
    return retval;
}

/**Function********************************************************************

  Synopsis   [Creates a ADD for a crossover]

  Description [A crossover is the intersection (crossing) of two wires
  The terminal nodes of a crossover ADD are 0, 2, 1 and 3]

  @param none
******************************************************************************/
DdNode * Cudd_crossover ()
{
    DdNode *var1 = Cudd_addNewVar(gbm); /*First variable in the crossover DD*/
    Cudd_Ref(var1);
    DdNode *var2 = Cudd_addNewVar(gbm); /*Second variable in the crossover DD*/
    Cudd_Ref(var2);
    DdNode *node1 = Cudd_addIte(gbm,var2, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)2),
        Cudd_addConst(gbm, (CUDD_VALUE_TYPE)0)); /*Else branch=0, Then branch=2*/
    Cudd_Ref(node1);
    DdNode *node2 = Cudd_addIte(gbm,var2, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)3),
        Cudd_addConst(gbm, (CUDD_VALUE_TYPE)1)); /*Else branch=1, Then branch=3*/
    Cudd_Ref(node2);
```

```
    DdNode *retval = Cudd_addIte(gbm,var1, node2 ,node1);
    Cudd_RecursiveDeref(gbm,var1);
    Cudd_RecursiveDeref(gbm,var2);
    Cudd_RecursiveDeref(gbm,node1);
    Cudd_RecursiveDeref(gbm,node2);
    return retval;
}


/**Function************************************************************************

  Synopsis    [Prints a dd summmary]

  Description [pr = 0 : prints nothing
  pr = 1 : prints counts of nodes and minterms
  pr = 2 : prints counts + disjoint sum of product
  pr = 3 : prints counts + list of nodes
  pr > 3 : prints counts + disjoint sum of product + list of nodes]

  @param the dd node
************************************************************************************/
void print_dd (DdNode *dd, int n, int pr, char *name)
{
    printf("%s\n", name);
    printf("DdManager nodes: %ld | ", Cudd_ReadNodeCount(gbm)); /*Reports the number of live nodes
        in BDDs and ADDs*/
    printf("DdManager vars: %d | ", Cudd_ReadSize(gbm) ); /*Returns the number of BDD variables in
        existance*/
    if(NULL !=dd) {
        printf("DdNode nodes: %d | ", Cudd_DagSize(dd)); /*Reports the number of nodes in the BDD*/
        printf("DdNode vars: %d | ", Cudd_SupportSize(gbm, dd) ); /*Returns the number of variables
            in the BDD*/
    }
    printf("DdManager reorderings: %d | ", Cudd_ReadReorderings(gbm) ); /*Returns the number of
        times reordering has occurred*/
    printf("DdManager memory: %ld bytes\n\n", Cudd_ReadMemoryInUse(gbm) ); /*Returns the memory in
        use by the manager measured in bytes*/
```

```
    if (Cudd_SupportSize(gbm, dd) > 20 && verbosity >1) {
        pr = 1;
        printf("\nCaution: DD is too large to print (%d vars), verbosity has been reduced to 1
            !\n\n", Cudd_SupportSize(gbm, dd));
    }
    Cudd_PrintDebug(gbm, dd, n, pr); // Prints to the standard output a DD and its statistics:
        number of nodes, number of leaves, number of minterms.
}

/**Function********************************************************************

  Synopsis    [Writes a dot file representing the argument DDs]

  Description []

  @param the dd node and the filename
********************************************************************************/
void write_dd (DdNode *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    DdNode **ddnodearray = (DdNode**)malloc(sizeof(DdNode*)); // initialize the function array
    ddnodearray[0] = dd;
    if (NULL != dd) {
        Cudd_DumpDot(gbm, 1, ddnodearray, NULL, NULL, outfile); // dump the function to .dot file
    }
    fclose (outfile); // close the file */
    free(ddnodearray);
}

/**Function********************************************************************

  Synopsis    [Prints a dd summary and writes a dot file representing the argument DDs]

  Description []
```

```
  @param the dd node and the filename
******************************************************************************/
void print_all (DdNode *dd, char* filename, int pr, char *name)
{
    print_dd(dd,2,pr, name); /*Print expansion dd to standard out*/
    write_dd(dd, filename);  /*Write current expansion ADD to file*/
}

/**Function********************************************************************

  Synopsis   [Prints out statistics and settings for a CUDD manager]

  Description []

  @param the dd manager and the filename
******************************************************************************/
void info_dd (DdManager *dd, char* filename)
{
    FILE *outfile; // output file pointer for .dot file
    outfile = fopen(filename,"w");
    Cudd_PrintInfo (gbm, outfile);
    fclose (outfile); // close the file
}



/**Function********************************************************************

  Synopsis   [Build a crossover DD for one cascade according to the position of the crossover DD]

  Description [Given a permutation in an array 'permut', this function creates a new ADD with
      permuted variables.
  There should be an entry in the array 'permut' for each variable in the manager. The i-th entry
      of permut holds the index of the variable that is to substitute the i-th variable.
  Returns a pointer to the resulting ADD if successful; NULL otherwise.
  permut [index] = mapping; this formula doesnt work
  permut [mapping] = index; this formula works]
```

```
  @param a circuit c, the current cascade n, the mode 0:monolithic / 1:array
*****************************************************************************/
DdNode * crossover_reordering (circuit c, int n, DdNode *node)
{
    int order=0, k=0;
    int rootIndex = (int) Cudd_NodeReadIndex(node); // Index of the first variable in the node
    int managerVars = Cudd_ReadSize(gbm); // Number of variables in the manager
    int* permut = malloc((managerVars) * sizeof(int)); // Array of permutation variables
    for(k=0; k < managerVars; k++) // Initialization: there should be an entry in array permut for
         each variable in the manager
        permut[k]= k;
    int index = rootIndex;

    printf("\nnode index: %d, node vars: %d\n", rootIndex, Cudd_SupportSize(gbm, node));
    // This loop fills out the array of permutation variables with variables indexes
    while (c->partition[n][order] != NULL) {
        printf("\nc->partition[%d][%d]: %s, order: %d, mapping: ", n, order,
             c->partition[n][order]->name, c->partition[n][order]->order);
        for(k=0; k < c->partition[n][order]->mappingCount; k++) { /*Print out the array of
             mappings*/
            printf("%d ", c->partition[n][order]->mapping[k]);
            permut[rootIndex + c->partition[n][order]->mapping[k]] = index; //
            //permut[index] = rootIndex + c->partition[n][order]->mapping[k]; // Normal operation
                 from cudd documentation
            index++;
        }
        order++;
    }
    /*
    printf("\n Number of variables in the manager = %d\n", managerVars );
    for(k=0; k < managerVars; k++) // Initialization: there should be an entry in array permut for
         each variable in the manager
        printf("permut[%d] = %d\n", k, permut[k]);
    */
    DdNode *reordered_dd = Cudd_addPermute(gbm, node, permut);
    free(permut);
```

```
    return reordered_dd;
}


/**Function********************************************************************

  Synopsis    [Build cascades of a circuit using the Kronecker product of all parallel elements]

  Description []

  @param the circuit c, and the current cascade n
********************************************************************************/
DdNode * cascade_kronecker (circuit c, int n)
{
    int i=0, j=0;
    char filename[40];        /*Name of the files that hold the cascades*/
    struct timeval startTime;
    double diffTime=0;
    DdNode *tmp;            /*Initialize temporary dd*/
    DdNode *cascade_dd = Cudd_ReadZero(gbm);  /*Initialize the cascade holding the Kronecker
        product*/
    Cudd_Ref(cascade_dd);

    printf ("\n******** CASCADE DD KRONECKER PRODUCT STAGE %d *********\n", n);

    gettimeofday(&startTime, NULL); // Start timer
    while (i<c->wirecount && c->wires[i] != NULL) {  /*Iterate through all wires*/
        if (c->wires[i]->partition == n || pass_through (c,c->wires[i], n))
        {
            if (gate (c->wires[i]->type) || pass_through (c,c->wires[i], n) ||
                c->wires[i]->primary) /*Only gates and pass-throughs are allowed in a cascade*/
            {
                printf("\nOuter product by: WIRE %s, number of inputs:%d ...\n", c->wires[i]->name,
                    c->wires[i]->inputcount);
                c->wires[i]->dd = get_add(c->wires[i], c); /*Build the BDD for the wire*/
                gbm->fanout_coef = pow(2, c->wires[i]->outputcount); /*Hack to pass eventual fanout
                    coefficients to the kronecker function*/
```

```
            tmp = Cudd_addApply(gbm, Cudd_addKronecker, c->wires[i]->dd, cascade_dd);
                /*Kronecker Product of 2 ADDs*/

            //sprintf(filename, "./cascades/add_%d_%d_%s.dot", n, j, c->wires[i]->name);
            //print_all (c->wires[i]->dd, filename, 4, "");

            Cudd_Ref(tmp);
            Cudd_RecursiveDeref(gbm, c->wires[i]->dd);
            Cudd_RecursiveDeref(gbm, cascade_dd);
            cascade_dd = tmp;
            j++;
        }
    }
    i++;
  }
  stopTimer(&diffTime, startTime); // Stop timer
  kroneckerTime += diffTime; // Accumulate DD kronecker building time

  sprintf(filename, "./cascades/cascade_%d.dot", n);
  //print_all (cascade_dd, filename, verbosity, "");
  printf ("\n******** END CASCADE BDD KRONECKER PRODUCT STAGE %d | Number of nodes with non-zero
      reference counts: %d ********\n\n", n, Cudd_CheckZeroRef(gbm) );

  return cascade_dd; /*Return the built dd for the cascade*/
}

/**Function********************************************************************

  Synopsis    [Build cascades of a circuit using the Kronecker product of all parallel elements]

  Description [Inject eventual crossover matrices in between]

  @param the circuit c, the mode
******************************************************************************/
void
transferFunction (
  circuit c, /*The circuit to test*/
```

```
    int mode, /*The simulation type: 0:Monolithic, 1:Array*/
    int row,  /*If array method, the input vector to test for*/
    int crossover_mode)
{
    int n=0;
    char filename[40]; /*Name of the file that holds the result*/
    struct timeval startTime;
    double diffTime=0;
    DdNode *cascade_dd = Cudd_ReadZero(gbm); /*Initialize the individual cascade dd*/
    Cudd_Ref(cascade_dd);
    DdNode *crossover_dd=NULL; /*Intermediate DD*/
    DdNode *tmp=NULL;
    DdNode *reordered_cascade = NULL;

    printf( "\n\n\n**************** DIRECT PRODUCTS (.) *****************\n");

    for(n=0; n<c->cascades ; n++) {
        printf( "\n\n******** IN CASCADE: %d ********\n", n);
        if (NULL != cascade_dd)
            Cudd_RecursiveDeref(gbm, cascade_dd);
        cascade_dd = cascade_kronecker (c, n); /*Operation to compute the cascade dd using
            Kronecker product*/

        if (mode==DD_MONOLITHIC && crossover_mode==CROSSOVER_REORDERING) { /*Monolithic new
            crossover method*/
            if (n == 0) { //First cascade is untouched, no node multiplication needed
                c->transfer_dd = cascade_dd;
                Cudd_Ref(c->transfer_dd);
            }
            else {
                if (isCrossover(c, n-1)) { //Check for crossover in the previous cascade ( We check
                    the previous casacades to reorder input variables (Not possible on the output
                    variables))
                    printf("\n\n << Crossover detected in cascade %d: start reordering cascade
                        %d\n", n-1, n);
                    reordered_cascade = crossover_reordering (c, n-1, cascade_dd);  // Reorder the
                        cascade DD
```

```
            Cudd_Ref(reordered_cascade);
            Cudd_RecursiveDeref(gbm, cascade_dd);
            cascade_dd = reordered_cascade;
        }
        tmp = Cudd_addMultiply(gbm, c->transfer_dd, cascade_dd);
        Cudd_RecursiveDeref(gbm, c->transfer_dd);
        c->transfer_dd = tmp;
    }
}// End monolithic function with crossover_reordering method

else if (mode==DD_ARRAY1 && crossover_mode==CROSSOVER_REORDERING) { /*Arrray method 1*/
    if (n == 0) { /*Multiply the row vector by the first cascade*/
        c->transfer_dd = Cudd_addMultiply(gbm, Cudd_addConst (gbm, (CUDD_VALUE_TYPE)row),
             cascade_dd);
        Cudd_Ref(c->transfer_dd);
    }
    else {
        if (isCrossover(c, n-1)) { //Check for the eventual crossover injection. If a
             crossover is detected in the current cascade, then we multiply the current
             vector by the crossover DD
            printf( "\n\n << CROSSOVER INJECTION\n");

            gettimeofday(&startTime, NULL); // Start timer
            reordered_cascade = crossover_reordering(c, n-1, cascade_dd);  // Reorder the
                 cascade DD
            Cudd_Ref(reordered_cascade); // Reference the reordered cascade
            stopTimer(&diffTime, startTime); // Stop timer
            reorderingTime_arr += diffTime; // Accumulate DD reordering time

            Cudd_RecursiveDeref(gbm, cascade_dd);
            cascade_dd = reordered_cascade; // Point the casade DD to the reordered DD
        }
        else {
            gettimeofday(&startTime, NULL); // Start timer
            tmp = Cudd_addMultiply(gbm, c->transfer_dd, cascade_dd);
            stopTimer(&diffTime, startTime); // Stop timer
            simulationTime_arr += diffTime; // Accumulate DD simulation time
```

```
                Cudd_Ref(tmp);
                Cudd_RecursiveDeref(gbm, c->transfer_dd);
                c->transfer_dd = tmp;
            }
        }
    } // End array 1 with crossover_reordering method

    else { /*Defaut case: testing*/
        printf("\n\nCascade debugging %d\n", n);
        print_dd(cascade_dd, 2, 4, "new cascade_dd built from cascade_kronecker"); // Print
            cascade dd to standard out
    }

    printf( "\n******** END CASCADE %d | Number of nodes with non-zero reference counts: %d
        ********\n\n", n, Cudd_CheckZeroRef(gbm) );
}

sprintf(filename, "./add/transfer_dd_%d.dot", n);
print_all (c->transfer_dd, filename, 4,"\n\n++++++ TRANSFER ADD ++++++ \n");

if (NULL != cascade_dd)
    Cudd_RecursiveDeref(gbm, cascade_dd);

printf( "\n******** END DIRECT PRODUCTS *********\n\n");
}


/**Function************************************************************************

  Synopsis    [Inputs crawl through all the cascades until reaching the final output]

  Description [Starting from the primary inputs the algorithm evaluates the resulting BDD constant
      in every cascade.
  This method ends up providing constant vectors for each output]

  @param the circuit c, and the current cascade n
```

```
**************************************************************************/
void distributed_dd (circuit c)
{
    int i=0, n=0;
    for(n=0; n<=c->cascades ; n++) { /*For all the cascades in the circuit*/
        printf( "\n\n******** IN CASCADE: %d ********\n", n);

        while (i<c->wirecount && c->wires[i] != NULL) {  /*Iterate through all wires*/
            if (c->wires[i]->partition == n || pass_through (c,c->wires[i], n))
            {
                if (gate (c->wires[i]->type) || pass_through (c,c->wires[i], n) ||
                     c->wires[i]->primary) /*Only gates and pass-throughs are allowed in a cascade*/
                {
                    printf("\nSimulating %s, Type:%s ...\n", c->wires[i]->name, c->wires[i]->type);
                    get_bdd(c->wires[i], c); /*Simulate a single gate at a time*/
                }
            }
            i++;
        }
        i=0;
        printf( "\n******** END CASCADE %d | Number of nodes with non-zero reference counts: %d
             ********\n\n", n, Cudd_CheckZeroRef(gbm) );
    }
}
/**Function*************************************************************

  Synopsis    [Build corresponding ADD for a wire]

  Description []

  @param the wire object, the circuit c
**************************************************************************/
DdNode * get_add (wire w, circuit c)
{
    int i;
    DdNode *dd, *tmp1, *tmp2, *tmp3, *var;
    dd = Cudd_ReadOne(gbm); /*Initialize dd*/
```

```
Cudd_Ref(dd);

int type_num = convert (w->type); /*Convert type to number*/

switch (type_num) {
case INPUT:
    fprintf(stderr,"Error: Uninitialized primary input.\n");
    break;

case AND:
    printf("Building AND %s ADD, id: %d\n", w->name, w->id);
    tmp1 = Cudd_ReadOne(gbm);
    Cudd_Ref(tmp1);
    for (i=0; i < w->inputcount; i++) {
        var = Cudd_bddNewVar(gbm);
        Cudd_Ref(var);
        tmp2 = Cudd_bddAnd(gbm, tmp1, var);
        Cudd_Ref(tmp2);
        Cudd_RecursiveDeref(gbm,var);
        Cudd_RecursiveDeref(gbm,tmp1);
        tmp1 = tmp2;
    }
    Cudd_RecursiveDeref(gbm,dd);
    dd=tmp1;
    break;

case NAND:
    printf("Building NAND %s ADD, id: %d\n", w->name, w->id);
    tmp1 = Cudd_ReadOne(gbm);
    Cudd_Ref( tmp1 );
    for (i=0; i < w->inputcount; i++) {
        var = Cudd_bddNewVar(gbm);
        Cudd_Ref(var);
        tmp2 = Cudd_bddAnd(gbm, tmp1, var);
        Cudd_Ref(tmp2);
        Cudd_RecursiveDeref(gbm,var);
        Cudd_RecursiveDeref(gbm,tmp1);
```

```
            tmp1 = tmp2;
        }
        Cudd_RecursiveDeref(gbm,dd);
        dd = Cudd_Not(tmp1);
        break;

    case OR:
        printf("Building OR %s ADD, id: %d\n", w->name, w->id);
        tmp1 = Cudd_ReadLogicZero(gbm);
        Cudd_Ref(tmp1);
        for (i=0; i < w->inputcount; i++) {
            var = Cudd_bddNewVar(gbm);
            Cudd_Ref(var);
            tmp2 = Cudd_bddOr(gbm,tmp1, var);
            Cudd_Ref(tmp2);
            Cudd_RecursiveDeref(gbm,var);
            Cudd_RecursiveDeref(gbm,tmp1);
            tmp1 = tmp2;
        }
        Cudd_RecursiveDeref(gbm,dd);
        dd = tmp1;
        break;

    case NOR:
        printf("Building NOR %s ADD, id: %d\n", w->name, w->id);
        tmp1 = Cudd_ReadLogicZero(gbm);
        Cudd_Ref(tmp1);
        for (i=0; i < w->inputcount; i++) {
            var = Cudd_bddNewVar(gbm);
            Cudd_Ref(var);
            tmp2 = Cudd_bddOr(gbm,tmp1,var);
            Cudd_Ref(tmp2);
            Cudd_RecursiveDeref(gbm,var);
            Cudd_RecursiveDeref(gbm,tmp1);
            tmp1 = tmp2;
        }
        Cudd_RecursiveDeref(gbm,dd);
```

```
    dd = Cudd_Not(tmp1);
    break;

case XOR:
    printf("Building XOR %s ADD, id: %d\n", w->name, w->id);
    tmp1 = Cudd_ReadLogicZero(gbm);
    Cudd_Ref(tmp1);
    for (i=0; i < w->inputcount; i++) {
        var = Cudd_bddNewVar(gbm);
        Cudd_Ref(var);
        tmp2 = Cudd_bddXor(gbm,tmp1,var);
        Cudd_Ref(tmp2);
        Cudd_RecursiveDeref(gbm,var);
        Cudd_RecursiveDeref(gbm,tmp1);
        tmp1 = tmp2;
    }
    Cudd_RecursiveDeref(gbm,dd);
    dd = tmp1;
    break;

case XNOR:
    printf("Building XNOR %s ADD, id: %d\n", w->name, w->id);
    tmp1 = Cudd_ReadOne(gbm);
    Cudd_Ref(tmp1);
    for (i=0; i < w->inputcount; i++) {
        var = Cudd_bddNewVar(gbm);
        Cudd_Ref(var);
        tmp2 = Cudd_bddXor(gbm,tmp1,var);
        Cudd_Ref(tmp2);
        Cudd_RecursiveDeref(gbm,var);
        Cudd_RecursiveDeref(gbm,tmp1);
        tmp1 = tmp2;
    }
    Cudd_RecursiveDeref(gbm,dd);
    dd = tmp1;
    break;
```

```
case NOT:
    printf("Building NOT %s ADD, id: %d\n", w->name, w->id);
    tmp1 = Cudd_bddNewVar(gbm);
    Cudd_Ref(tmp1);
    Cudd_RecursiveDeref(gbm,dd);
    dd = Cudd_Not(tmp1);
    break;

case BUF:
    printf("Building BUF %s ADD, id: %d\n", w->name, w->id);
    Cudd_RecursiveDeref(gbm,dd);
    dd = Cudd_bddNewVar(gbm);
    Cudd_Ref(dd);
    break;

case I:
    printf("Building I %s ADD, id: %d\n", w->name, w->id);
    Cudd_RecursiveDeref(gbm,dd);
    dd = Cudd_bddNewVar(gbm);
    Cudd_Ref(dd);
    break;

case FO:
    printf("Building FO %s ADD, id: %d\n", w->name, w->id);
    Cudd_RecursiveDeref(gbm,dd);
    dd = Cudd_fanout (w->outputcount);
    Cudd_Ref(dd);
    return dd;
    break;

case FI:
    printf("Building FI %s ADD, id: %d\n", w->name, w->id);
    Cudd_RecursiveDeref(gbm,dd);
    dd = Cudd_bddNewVar(gbm);
    Cudd_Ref(dd);
    break;
```

```
    default:
        fprintf(stderr,"Error: Illegal wire type (DD)\n");
        exit(1);
    }

    tmp3 = Cudd_BddToAdd(gbm, dd);
    Cudd_Ref(tmp3);
    Cudd_RecursiveDeref (gbm, dd);
    return tmp3;
}

#endif // _BDD_H_
```

BIBLIOGRAPHY

[1] Agrawal, D., Baktir, S., Karakoyunlu, D., and Pankaj Rohatgi, B. S. Trojan detection using ic fingerprinting. *Security and Privacy, 2007. SP '07. IEEE Symposium on* (2007), 296 – 310.

[2] Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. Algebric decision diagrams and their applications. *Formal methods in system design 10*, 2-3 (1997), 171–206.

[3] Bollig, B., and Wegener, I. Improving the variable ordering of obdds is np-complete. *Computers, IEEE Transactions on 45*, 9 (Sep 1996), 993–1002.

[4] Bryant, R. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on C-35*, 8 (Aug 1986), 677–691.

[5] Clarke, E. M., Fujita, M., and Zhao, X. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*. Springer, 1996, pp. 93–108.

[6] Cook, S. A. The complexity of theorem-proving procedures. In *In STOC* (1971), ACM, pp. 151–158.

[7] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (July 1962), 394–397.

[8] De Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., and Segala, R. *Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation*. Springer, 2000.

[9] Dirac, P. A. M. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society 35*, 3 (1939), 416–418.

[10] Fujii, H., Ootomo, G., and Hori, C. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on* (Nov 1993), pp. 38–41.

[11] Fujita, M., Fujisawa, H., and Matsunaga, Y. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 12*, 1 (Jan 1993), 6–12.

[12] Fujita, M., McGeer, P. C., and Yang, J.-Y. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal methods in system design 10*, 2-3 (1997), 149–169.

[13] Gu, J., Purdom, P. W., Franco, J., and Wah, B. W. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1996), American Mathematical Society, pp. 19–152.

[14] Houngninou, D. K., and Thornton, M. A. Implementation of switching circuit models as transfer functions. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)* (May 2016), pp. 2162–2165.

[15] Houngninou, D. K., and Thornton, M. A. Simulation of switching circuits using transfer functions. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)* (Aug 2017), pp. 511–514.

[16] Houngninou, D. K., and Thornton, M. A. (under review) efficient computation of switching function degree and algebraic normal form. *IEEE Transactions on Computers* (2017).

[17] Jin, Y., Kupp, N., and Makris, Y. Experiences in hardware trojan design and implementation. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust* (Washington, DC, USA, 2009), HST '09, IEEE Computer Society, pp. 50–57.

[18] Lu, F., c. Wang, L., ting (tim Cheng, K., Moondanos, J., and Hanna, Z. A signal correlation guided circuit-sat solver. *J. UCS 10* (2004), 1629–1654.

[19] Malik, S., Wang, A., Brayton, R., and Sangiovanni-Vincentelli, A. Logic verification using binary decision diagrams in a logic synthesis environment. In *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on* (Nov 1988), pp. 6–9.

[20] Matsunaga, Y., McGeer, P. C., and Brayton, R. K. On computing the transitive closure of a state transition relation. In *Proceedings of the 30th International Design Automation Conference* (New York, NY, USA, 1993), DAC '93, ACM, pp. 260–265.

[21] Ossowski, J. *Symbolic Representation and Manipulation of Discrete Functions*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2004.

[22] Reda, S. Combinational equivalence checking using boolean satisfiability and binary decision diagrams. *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings* (2001), 122 – 126.

[23] Rudell, R. Dynamic variable ordering for ordered binary decision diagrams. In *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on* (Nov 1993), pp. 42–47.

[24] Sensarma, D., Banerjee, S., Basuli, K., Naskar, S., and Sen-Sarma, S. On an optimization technique using binary decision diagram. *CoRR abs/1203.2505* (2012).

[25] Somenzi, F. Binary decision diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences* (1999), IOS Press, pp. 303–366.

[26] Somenzi, F. *CUDD: CU Decision Diagram Package*, release 3.0.0 ed. University of Colorado at Boulder, December 2015.

[27] Thornton, M. Simulation and implication using a transfer function model for switching logic. *IEEE Transactions on Computers PP* (February 2015).

[28] Thornton, M. A. *Modeling Digital Switching Circuits with Linear Algebra*. Morgan & Claypool Publishers, 2014.

[29] Wolff, F., Papachristou, C., Bhunia, S., and Chakraborty, R. S. Towards trojan-free trusted ics: Problem analysis and detection scheme. In *Proceedings of the Conference on Design, Automation and Test in Europe* (New York, NY, USA, 2008), DATE '08, ACM, pp. 1362–1365.