B.TECH. PROJECT REPORT

On

OPTIMIZATION OF MAJORITY-INVERTER GATE LOGIC OVER AND-OR-INVERTER GATE LOGIC

BY

AMAN SINGH   140001007

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, INDORE

2017

OPTIMIZATION OF MAJORITY-INVERTER GATE LOGIC OVER AND-OR-INVERTER GATE LOGIC

A PROJECT REPORT

*Submitted in partial fulfillment of the
requirements for the award of the degree*

of
BACHELOR OF TECHNOLOGY
in

COMPUTER SCIENCE AND ENGINEERING

Submitted by

AMAN SINGH   140001007

Guided by
Dr. Bodhisatva Mazumdar

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, INDORE

2017

Candidate's Declaration

I hereby declare that the project entitled **"Optimization of Majority-Inverter Gate Logic over And-Or-Inverter Gate Logic"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'Computer Science and Engineering' completed under the supervision of **Dr. Bodhisatwa Mazudar, Assistant Professor, Computer Science and Engineering, IIT Indore** is an authentic work.

Further, I declare that I have not submitted this work for the award of any other degree elsewhere.

<div align="right">Aman Singh</div>

Certificate

It is certified that the above statement made by the student is correct to the best of my knowledge.

<div align="right">

Dr. Bodhisatwa Mazumdar
Assistant Professor
Department of Computer Science and Engineering
IIT Indore

</div>

Preface

This report on **Optimization of Majority-Inverter Gate Logic over And-Or-Inverter Gate Logic** is prepared under the guidance of **Dr. Bodhisatwa Mazumdar**.

Through this report, I have tried to analyze the benefits of Majority-Inverter Gate Logic Representation over the And-Or-Inverter Gate Logic Representation. I have implemented the axioms included in the boolean algebra, and the optimization algorithms utilizing the given axioms. I compared the two representations on the basis of their size and depth.

I have tried to the best of my abilities and knowledge to explain the content in a lucid manner. I have also added figures to make it more illustrative.

Aman Singh
B.Tech. IV Year
Discipline of Computer Science and Engineering
IIT Indore

Acknowledgements

I am profoundly grateful to **Dr. BODHISATWA MAZUMDAR** for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

I would like to express deepest appreciation towards **Dr. SURYA PRAKASH**, Head of Department of Computer Engineering whose invaluable guidance supported me in completing this project.

At last, I must express my sincere heartfelt gratitude to my partner, **Mr. AKASH KUMAR**, for supporting and assisting me throughout the journey, as well as all the staff members of Computer Engineering Department who helped me directly or indirectly during this course of work.

Aman Singh

Abstract

The performance of today's digital integrated circuits largely depends on the capabilities of logic synthesis tools. In this context, efficient representation and optimization of Boolean functions are key features. Some data structures and algorithms have been proposed for these tasks. Most of them consider, as basis operations, inversion (INV), conjunction (AND), disjunction (OR) and if-then-else (MUX). Other Boolean operations are derived by composition.

Even though existing design automation tools, based on original optimization techniques produce good results and handle large circuits, the possibility to push further the efficacy of logic synthesis continues to be of paramount interest to the Electronic Design Automation (EDA) community. With this aim in mind this present the implemenation using Majority-Inverter Graph(MIG), a novel logic representation structure for efficient optimization of Boolean functions.

A MIG is a directed acyclic graph (DAG) consisting of three-input majority nodes and regular/complemented edges. We show that MIGs include any AND/OR/Inverter Graphs (AOIGs), containing also the wellknown AIGs. In order to support the natural manipulation of MIGs, we introduce a new Boolean algebra, based exclusively on majority and inverter operations, with a complete axiomatic system. Theoretical results show that it is possible to explore the entire MIG representation space by using only five primitive transformation rules. Such feature opens up a great opportunity for logic optimization and synthesis. We showcase the MIG potential by proposing a delay-oriented optimization technique.

The study of majority-inverter logic synthesis is also motivated by the design of circuits in emerging technologies. In the quest for increasing computational performance per unit area, majority/ minority gates are natively implemented in different nanotechnologies and also extend the functionality of traditional NAND/ NOR gates. In this scenario, MIGs and their algebra represent the natural methodology to synthesize majority logic circuits in emerging technologies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Predominance of AOIG Synthesis

Efficient representation and optimization of Boolean functions are key features in the performance of today's digital integrated circuits. Some data structures and algorithms have been proposed for these tasks. Most of them consider AOIG logic presently. Even though existing design automation tools, based on aforementioned optimization techniques produce good results and handle large circuits, the possibility to push further the efficiency of logic synthesis continues to be of paramount interest to the Electronic Design Automation (EDA) community.

1.2 Why Majority (MIG) Logic?
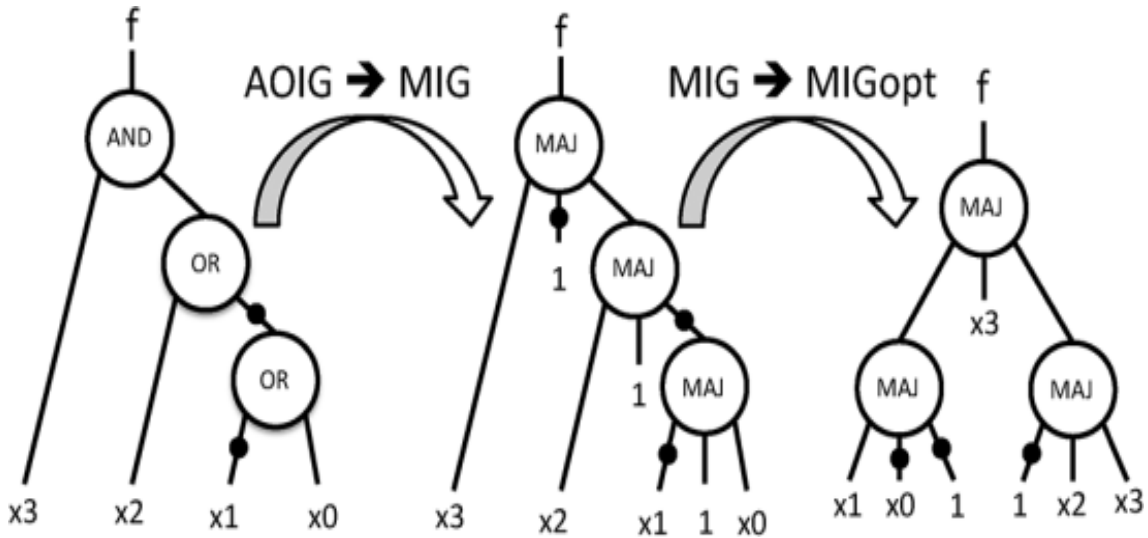
Figure 1.1: MIG representation for Formula $f = x3(x2 + (x1' + x0)')$**. Complementation is represented by bubbles on the edges.**

In this project, we propose a paradigm shift in implementing and optimizing logic by using only majority (MAJ) and inversion (INV) functions as basic operations. We represent logic functions by majority-inverter graph (MIG): a directed acyclic graph

consisting of three-input majority nodes and regular/complemented edges. We optimize MIGs via a new Boolean algebra, based exclusively on majority and inversion operations. As a complement to MIG algebraic optimization, we are making an attempt to develop powerful Boolean methods exploiting global properties of MIGs, such as bit-error masking. MIG algebraic and Boolean methods together attain very high optimization quality.

Chapter 2

Implementation

MIG is a homogeneous logic network with an indegree equal to 3 and each node representing the majority function. In an MIG, edges are marked by a regular or complemented attribute.The majority operator $M(a,b,c)$ behaves as the conjunction operator $AND(a,b)$ when $c = 0$ and as the disjunction operator $OR(a,b)$ when $c = 1$. Therefore, majority can be seen as a generalization of conjunction and disjunction.

2.1 MIG Boolean Algebra

For implementation purposes we follow a set of rules of MIG Boolean algebra. The following five primitive transformation rules form an axiomatic system ,referred to as Ω for manipulating the given circuit , so as to ultimately attain a better efficiency.

$$
\Omega
\begin{cases}
\textbf{Commutativity} - \Omega.C \\
M(x,y,z) = M(y,x,z) = M(z,y,x) \\
\textbf{Majority} - \Omega.M \\
\begin{cases}
\text{if}(x = y)\text{: } M(x,y,z) = x = y \\
\text{if}(x = y')\text{: } M(x,y,z) = z
\end{cases} \\
\textbf{Associativity} - \Omega.A \\
M(x,u,M(y,u,z)) = M(z,u,M(y,u,x)) \\
\textbf{Distributivity} - \Omega.D \\
M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z) \\
\textbf{Inverter Propagation} - \Omega.I \\
M'(x,y,z) = M(x',y',z')
\end{cases}
$$

Figure 2.1: MIG Boolean Algebra: Primitive transformation rules

It is possible to traverse the entire MIG representation space just by using Ω. From a logic optimization perspective, it means that we can always reach a desired MIG starting from any other equivalent MIG. However, the length of the exact transformation sequence might be impractical for modern computers. To alleviate this problem, we derive from Ω three powerful transformations, referred to as Ψ, shown below.

$$\Psi \begin{cases} \textbf{Relevance} - \boldsymbol{\Psi}.R \\ M(x,y,z) = M(x,y,z_{x/y'}) \\ \textbf{Complementary Associativity} - \boldsymbol{\Psi}.C \\ M(x,u,M(y,u',z)) = M(x,u,M(y,x,z)) \\ \textbf{Substitution} - \boldsymbol{\Psi}.S \\ M(x,y,z) = \\ M(v,M(v',M_{v/u}(x,y,z),u),M(v',M_{v/u'}(x,y,z),u')) \end{cases}$$

Figure 2.2: MIG Boolean Algebra: Powerful transformation rules

2.2 Representation of logic circuits in Python

We have implemented these above mentioned axioms (both Ω and Ψ) successfully. We used Python for programming along with *pyverilog* for parsing the input verilog files.

2.2.1 Pyverilog

Pyverilog is an open-source hardware design processing toolkit for Verilog HDL. All source codes are written in Python. Pyverilog includes code parser,dataflow analyzer,control-flow analyzer and code generator.

For the representation of logic networks in form of a tree graph, we use two classes, namely *Gate* class and *Terminal* class. The network can be imagined as the alternating sequence of Gate and Terminal levels.

2.2.2 Class : Gate

The Gate class comprises of three attributes : operator, destTerm and children. The operator attribute is a string describing the logic of the represented Gate ,for instance, *and* represents an AND logic gate. The destTerm attribute contains the index for output terminal in the terminals list. Finally , the children attribute is the list of indices for the input terminals in the 'terminals' list. The size of the list varies w.r.t. the represented logic gate.

2.2.3   Class : Terminal

The Terminal class comprises of four attributes : name, typ, pg list and cg list. The name attribute contains the defined name of the terminal in the input verilog file. The typ attribute defines the type of the terminal, namely input , signal or wire. The pg and cg lists contain the indices of the parent(output) gates and child (input) gates respectively in the gates list .

2.2.4   Graphical Representation of Circuits

To represent the circuits in suitable data structure in Python we basically used two lists, namely *gates* and *terminals*. The connections in the circuits are stored using the indices of the various elements. Moreover, the gates are only connected to terminals and the terminals are only connected to gates, ensuring alternating levels of gates and terminals in the circuit graph.

Various types of terminals are used for different purposes mentioned as follows : *wire* to connect two gates , *input* to connect a particular input terminal to the gate and *signal* to send a particular signal (high/low) to a particular terminal of a gate.

Traversal :   For traversing the circuit graph , we first search for an output terminal from the terminals list and trace its child gate from gate list using index present in the cg attribute of the terminal class. Now, from the children attribute of the gate class of the current gate we trace its children terminals. Similarly , we find the children gates of the corresponding children terminals and continually process the circuit graph in the same manner.

2.3   Implementation of Axioms

2.3.1   Majority - $\Omega.M$

$$if(x = y) : M(x,y,z) = x = y$$
$$if(x = y') : M(x,y,z) = z$$

For a particular gate we define a function named *checkMaj* which returns a list of all possible combinations of input terminals to the specified gate where two of the terminals are equivalent i.e. give same output value for same inputs. The result list will also contain the third terminal when the other two terminals are complementary.

We also define another function called *opMaj* which implements the $\Omega.M$ axiom by replacing the specified gate with any one of the terminals obtained from *checkMaj* function. Thus, implementing the $\Omega.M$ axiom.

2.3.2 Associativity - $\Omega.A$

$$M(x, u, M(y, u, z)) = M(z, u, M(y, u, x))$$

For a particular gate we define a function named *checkAssoc* which looks for a majority gate from input terminals of the given gate having at least one input terminal equivalent to the any one of the remaining input terminals of the given gate. This function returns all such possible combinations in form of a list.

Now, we define another function called *opAssoc* which implements the $\Omega.A$ axiom by swapping the *x* and *z* terminals present in the two majority gates defined by the resolve obtained from the *checkAssoc* function. Thus, implementing the $\Omega.A$ axiom.

2.3.3 Distributivity - $\Omega.D$

$$M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$$

The current implementation for $\Omega.D$ axiom operates in a right to left manner making it suitable for size optimization. For a particular gate we define a function named *checkDistr* which looks for a pair of majority gates among the input terminals of the given gate having at least two terminals equivalent. This function returns all such possible combinations in form of a list.

The result from the above mentioned function is utilized in another function named *opDistr*. This function takes the pair of equivalent terminals from the selected two majority gates and connects them to the give majority gate input terminals. Also , a new gate is created comprising of the remaining input terminals as its input terminal, which is then connectted as the third input terminal of the given majority gate.

2.3.4   Relevance - $\Psi.R$

$$M(x,y,z) = M(x,y,z_{x/y'})$$

We define a function named *opRel* which takes the indices of x and y as input. The function traverses through the third input terminal z and replaces all occurrences of terminals equivalent to x with the compliment of y. Thus implementing the $\Psi.R$ axiom.

2.3.5   Complementary Associativity - $\Psi.C$

$$M(x,u,M(y,u',z)) = M(x,u,M(y,x,z))$$

We define a function named *checkCompAssoc* which looks for a majority gate in the input terminal list having a terminal complementary to any one of the remaining terminals in given gate. It returns all such combinations of the selected majority gate x terminal and u terminal in the form of a list.

We also define a function *opCompAssoc* which takes the result list from the aforementioned *checkCompAssoc* function and replaces the u' int the selected gate with the x terminal from the initial gate. Thus implementing the $\Psi.C$ axiom.

2.3.6   Substitution - $\Psi.S$

$$M(x,y,z) = M(v,M(v',M_{v/u}(x,y,z),u),M(v',M_{v/u'}(x,y,z),u'))$$

We define a function named *opSubs* which takes an input of two terminal indices of v and u respectively as well as the gate index of the given gate. Then , this function creates two duplicate subtrees of the given gate and replaces the terminals equivalent to v in the first duplicate subtree with given u terminal and in the second duplicate subtree with u'. We connect the first duplicate subtree to a new majority gate with its other input terminals as v' and u. Similarly, we connect the other duplicate subtree to a new majority gate with its other terminals as v' and u'. Finally, we modify the initial defined gate's input terminals to be v terminal and two majority gates created before. Thus implementing the $\Psi.S$ axiom.

Hence, we have implemented the required axioms for size optimization using pyverilog. Morevover, we have tried to remove the redundant gates by deleting them after each tranformation in accordance with the axioms.

Chapter 3

Optimization

Logic optimization consists of manipulating a logic representation structure in order to minimize some target metric. Usual optimization targets are size (number of nodes/elements), depth (maximum number of levels), inter- connections (number of edges/nets), etc.

In this project, we present a new representation form, based on majority and inversion, with its native Boolean algebra. We show algebraic and Boolean optimization techniques for this data structure unlocking new points in the design space.

Under the multilevel logic representation, optimization aims at reducing graph size and depth or other accepted complexity metrics. There, multilevel logic optimization methods are divided into two groups: 1) Algebraic methods, which are fast and 2) Boolean methods, which are slower but may achieve better results. Advanced multilevel logic optimization methodologies, and associated tools, use both algebraic and Boolean methods.

Note that early attempts to majority logic have already been reported in the 60s, but, due to their inherent complexity, failed to gain momentum later on in automated synthesis. We address, in this project, the unique opportunity led by majority logic in a contemporary synthesis flow.

3.1 Algebraic Size Optimization

To optimize the size of an MIG, we aim at reducing the number of its nodes. The optimization algorithm basically consists of two phases; elimination phase and reshape phase.

Algorithm: MIG Algebraic Size-Optimization Psuedocode

INPUT: MIG α **OUTPUT:** Optimized MIG α

    for (cycles=0; cycles<*effort*; cycles++) **do**

        $\Omega.M_{L \to R}(\alpha);$ $\Omega.D_{R \to L}(\alpha);$

        $\Omega.A(\alpha);$ $\Psi.C(\alpha);$

        $\Psi.R(\alpha);$ $\Psi.S(\alpha);$

        $\Omega.M_{L \to R}(\alpha);$ $\Omega.D_{R \to L}(\alpha);$

    end for

3.1.1 Elimination Phase

The elimination phase basically consists of two boolean algebraic axioms: Majority and Distributivity.

1. Majority - $\Omega.M$:

$$if(x = y) : M(x,y,z) = x = y$$

$$if(x = y') : M(x,y,z) = z$$

In the MIG Boolean algebra domain, the evaluation of the majority axiom ($\Omega.M$) from left to right (L $\to$ R) reduces the size of the MIG, as M(x, x, z) = x, which is evident in the figure below.
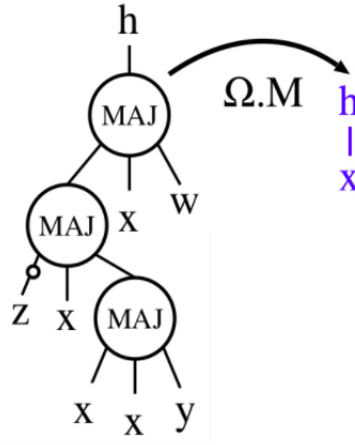
Figure 3.1: Majority Transformation in elimination phase

2. Distributivity - $\Omega.D$:

$$M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z)$$

In the MIG Boolean algebra domain, the evaluation of the distributivity axiom ($\Omega.D$) from right to left (R $\rightarrow$ L) reduces the size of the MIG, which is evident in the figure below.

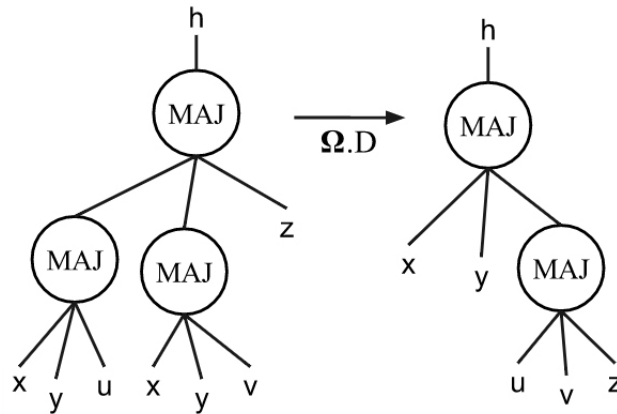**Figure 3.2: Distributivity Transformation in elimination phase**

3.1.2 Reshape Phase

The reshape phase is responsible for generating new elimination opportunities in a local minimum MIG. This phase consists of four boolean algebraic axioms: Associativity, Complementary Associativity, Relevance and Substitution.

1. Associativity - $\Omega.A$:

$$M(x,u,M(y,u,z)) = M(z,u,M(y,u,x))$$

In the MIG Boolean algebra domain, the evaluation of the associativity axiom ($\Omega.M$) from left to right (L $\rightarrow$ R) generates new elimination opportunities, which is evident in the figure below.
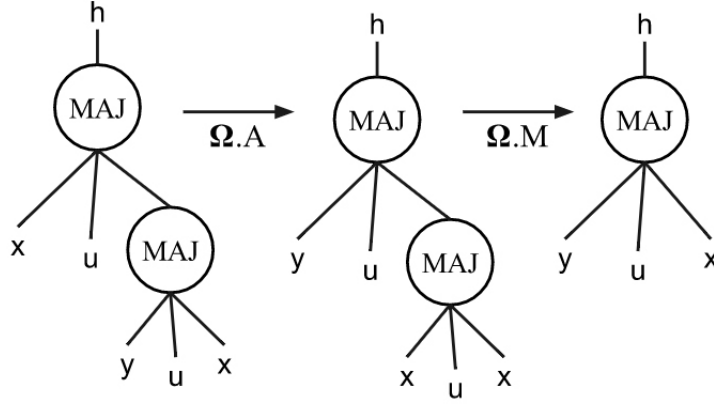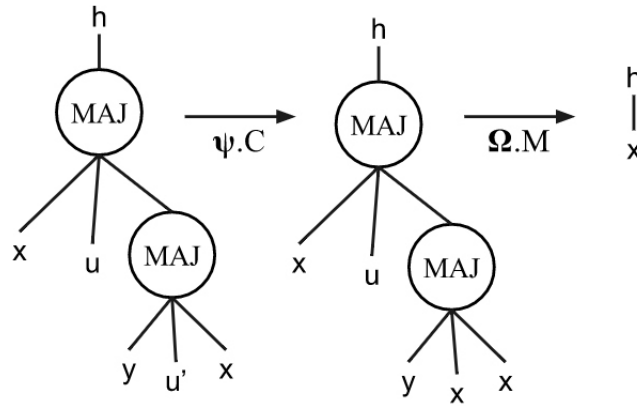

Figure 3.3: Associativity Transformation in reshape phase

2. Complementary Associativity - $\Psi.C$:

$$M(x, u, M(y, u', z)) = M(x, u, M(y, x, z))$$

In the MIG Boolean algebra domain, the evaluation of the complementary associativity axiom ($\Psi.C$) from left to right (L $\rightarrow$ R) generates new elimination opportunities, which is evident in the figure below.


Figure 3.4: Complementary Associativity Transformation in reshape phase

3. Relevance - $\Psi.R$:

$$M(x, y, z) = M(x, y, z_{x/y'})$$

In the MIG Boolean algebra domain, the evaluation of the relevance axiom ($\Psi.R$) from left to right (L $\rightarrow$ R) generates new elimination opportunities, which is evident in the figure below.
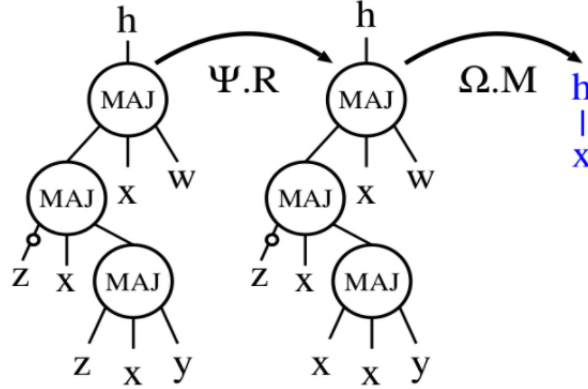
Figure 3.5: Relevance Transformation in reshape phase

4. Substitution - $\Psi.S$:

$$M(x,y,z) = M(v, M(v', M_{v/u}(x,y,z), u), M(v', M_{v/u'}(x,y,z), u'))$$

In the MIG Boolean algebra domain, the evaluation of the substitution axiom ($\Psi.S$) from left to right (L $\rightarrow$ R) generates new elimination opportunities, which is evident in the figure below.
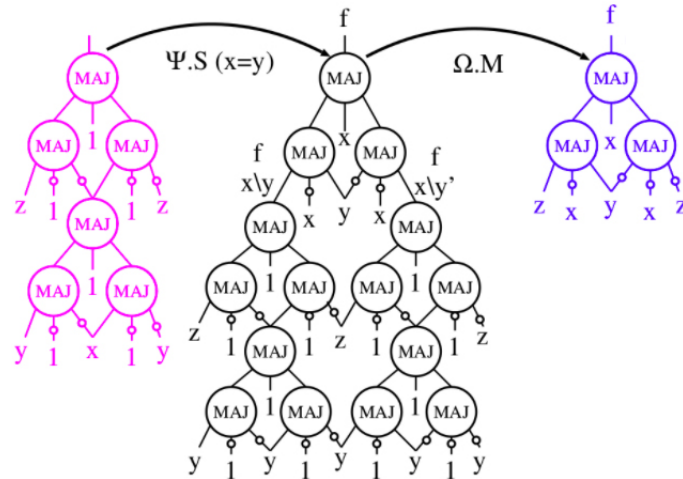
Figure 3.6: Substitution Transformation in reshape phase

3.1.3 Order of execution

An iteration of the size optimization algorithm includes the elimination phase, followed by the reshape phase, and then another elimination phase. These iteration are repeated "effort" times, which is set to 10 in the project.

3.2 Algebraic Depth Optimization

To optimize the depth of an MIG, we aim at reducing the length of its critical path. A valid strategy for this purpose is to move late arrival (critical) variables close to the outputs. This optimization algorithm consists of two phases; push-up phase and reshape phase.

Algorithm: MIG Algebraic Depth-Optimization Psuedocode

INPUT: MIG α **OUTPUT:** Optimized MIG α

    for (cycles=0; cycles $<$ *effort*; cycles++) **do**

        $\Omega.M_{L \to R}(\alpha); \quad \Omega.D_{L \to R}(\alpha); \quad \Omega.A(\alpha);$

        $\Omega.A(\alpha); \quad \Psi.C(\alpha);$

        $\Psi.R(\alpha); \quad \Psi.S(\alpha);$

        $\Omega.M_{L \to R}(\alpha); \quad \Omega.D_{L \to R}(\alpha); \quad \Omega.A(\alpha);$

    endfor

3.2.1 Push-Up Phase

The push-up phase basically consists of three boolean algebraic axioms: Majority, Distributivity and Associativity.

1. Majority - $\Omega.M$:

$$if(x = y) : M(x, y, z) = x = y$$

$$if(x = y') : M(x, y, z) = z$$

In the MIG Boolean algebra domain, the evaluation of the majority axiom ($\Omega.M$) from left to right (L $\to$ R) reduces the depth of the MIG, as M(x, x, z) = x, which is evident in the figure below.
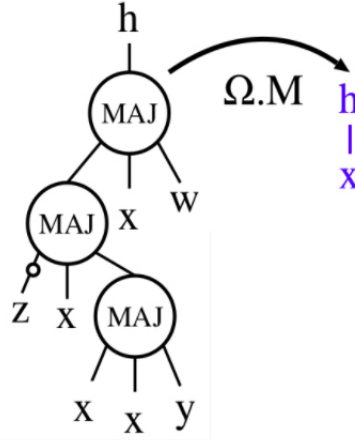
Figure 3.7: Majority Transformation in push-up phase

2. Distributivity - Ω.*D*:

$$M(x,y,M(u,v,z)) = M(M(x,y,u),M(x,y,v),z)$$

In the MIG Boolean algebra domain, the evaluation of the distributivity axiom (Ω.*D*) from left to right (L → R) reduces the depth of the MIG in case the z terminal is the critical terminal for the node, which is evident in the figure below.
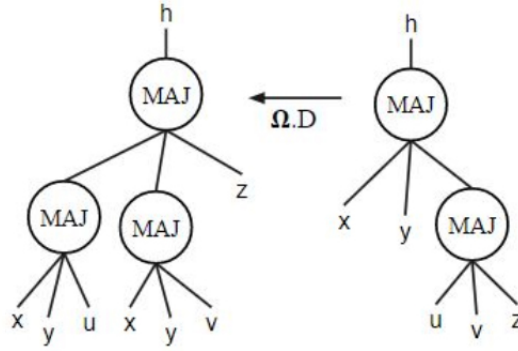

Figure 3.8: Distributivity Transformation in push-up phase

3. Associativity - Ω.*A*:

$$M(x,u,M(y,u,z)) = M(z,u,M(y,u,x))$$

In the MIG Boolean algebra domain, the evaluation of the associativity axiom (Ω.*M*) from left to right (L → R) can be used to reduce depth, which is evident in the figure below.
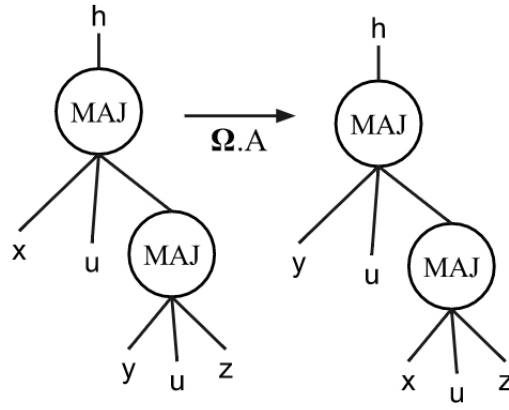
Figure 3.9: Associativity Transformation in push-up phase

3.2.2   Reshape Phase

The reshape phase in the algebraic depth optimization is the same as the one in the algebraic size optimization [3.1.2].

3.2.3   Order of execution

Similar to the algebraic size optimization, an iteration of the depth optimization algorithm includes the push-up phase, followed by the reshape phase, and then another push-up phase. These iteration are repeated "effort" times, which is set to 10 in the project.

3.3 Boolean Depth Optimization

Algorithm: MIG Boolean Depth-Optimization Pseudocode

INPUT: MIG α **OUTPUT:** Optimized MIG α

    for (cycles=0; cycles $<$ *effort*; cycles++) **do**

        {a,b} = search_critical_voters(α);

        c = size_bounded_root(α, a, b);

        x_1^n = common_parents(α, a, b);

        $c^A = c^{b/a'}$;

        $c^B = c^{x_1^n/a}$;

        $c^C = c^{x_1^n/b}$;

        MIG$-$depth_Alg_Opt(c^A);

        MIG$-$depth_Alg_Opt(c^B);

        MIG$-$depth_Alg_Opt(c^C);

        c = M(c^A, c^B, c^C);

        MIG$-$depth_Alg_Opt(c);

        if depth(c) is not reduced **then**

            reduce to previous MIG state

        endif

    endfor

Inserting Safe Errors in MIG: MIGs are hierarchical majority voting systems. One notable property of majority voting is the ability to correct different types of bit-errors. This feature is inherited by MIGs, where the error masking property can be exploited for logic optimization. The idea is to purposely introduce logic errors that are successively masked by the voting resilience in MIG nodes. If such errors are advantageous, in terms of logic simplifications, better MIG representations can be generated.

3.3.1 Identifying Advantageous Errors in MIGs

Boolean optimization methods exploit the error insertion schemes presented above. First, we identify advantageous orthogonal errors in MIGs using *Critical Voters Method*.

Critical Voters Method: To discover advantageous triplets of orthogonal errors we analyze an MIG structure. We want to identify critical portions of an MIG to be simplified thanks to these errors. To do so, we focus on nodes that have the highest impact on the final voting decision, i.e., influencing a Boolean function most. We call such nodes critical voters of an MIG. Critical voters can also be primary input themselves. To determine the critical voters, we rank MIG nodes based on a criticality metric.

Criticality Computation: Consider an MIG node m. We label all MIG nodes whose computation depends on m. For all such nodes, we calculate the impact of m by propagating a unit weight value from m outputs up to the root with an attenuation factor of 1/3 each time a majority node is encountered. We finally sum up all the values obtained and call this result criticality of m. Intuitively, MIG nodes with the highest criticality are critical voters.

We first determine two critical voters a and b and a set of MIG nodes fed directly by both a and b, say c1, c2, . . . , cn. In this context, an advantageous triplet of orthogonal errors is: A: a = b', B: c1 = a, c2 = a, . . . , cn = a, and C: c1 = b, c2 = b, . . . , cn = b. C. There, the critical voters are a = m2 and b = x1, while c1 =m3. Thus, the pairwise orthogonal errors are m2 = x1' (A), m3 = x1 (B) and m3 = m2 (C)

3.3.2   Depth Oriented MIG Boolean Optimization

The most intuitive way to exploit safe error insertion in MIGs is to reduce the number of levels. This is because the initial overhead in $w = M(w^A, w^B, w^C)$, where w is the initial MIG and w^A, w^B, w^C are the three erroneous versions, is just one additional level. This extra level is usually amply recovered during simplification and optimization of MIG erroneous branches. For depth-optimization purposes, the critical voters method enables very good results. The reason is the following. Critical voters appear along the critical path more than once. Thus, the possibility to insert simplifying errors on critical voters directly enables a strong reduction in the maximum number of levels. Sometimes, using an actual MIG root for error insertion requires an unpractical size overhead. In these cases, we bound the critical voters search to sub-MIGs partitioned on a depth criticality basis. Once the critical voters and a proper error insertion root have been identified, three erroneous sub-MIG versions are generated as explained above. On these sub-MIGs, we want to reduce the logic height. We do so by running algebraic MIG optimization on them. Note that,

in principle, also MIG Boolean methods can be reused. This would correspond to a recursive Boolean optimization. However, it turned out during experimentation that algebraic optimizations already produce satisfactory results at the local level. Thus, it makes more sense to apply Boolean techniques iteratively on the whole MIG structure rather than recursively on the same logic portion. At the end of the optimization of erroneous branches, the new MIG-roots must be given in input to a top majority voting node. This re-establishes the functional correctness. A last gasp of MIG algebraic optimization is applied at this point, to take advantage of the simplification opportunities arose from the integration of erroneous branches.

Each erroneous branch is handled by fast algebraic optimization to reduce its depth. The most common operation is $\Omega.M$ that directly simplifies the introduced errors. The optimized erroneous branches are then linked together by a top fault-masking majority node. A last gasp of algebraic optimization on the final MIG structure further optimizes its depth. In summary, our MIG Boolean optimization techniques attains a depth reduction of 60 percent.

Chapter 4

Analysis

In this project, we presented a Boolean logic optimization framework based on Majority-Inverter Graph (MIG). The top-order of the optimization algorithms is listed as follows:

- Algebraic Depth Optimization

- Algebraic Size Optimization

- Boolean Depth Optimization

- Algebraic Size Optimization

We processed a few Verilog files and compared the output with the AIG optimization, which are presented in the following table.
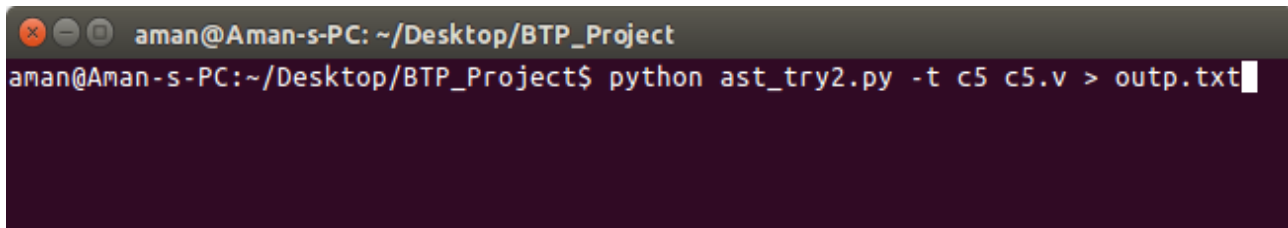
Logic	AIG		MIG	
File	Size	Depth	Size	Depth
Test1	4	3	1	1
Test2	15	6	7	3
C432	152	12	160	11
C1355	392	18	502	18
C1908	363	25	459	23

The current implementation favours reduction of depth at the expense of size, which can be reversed by changing the top-order of the optimization algorithms.

Chapter 5
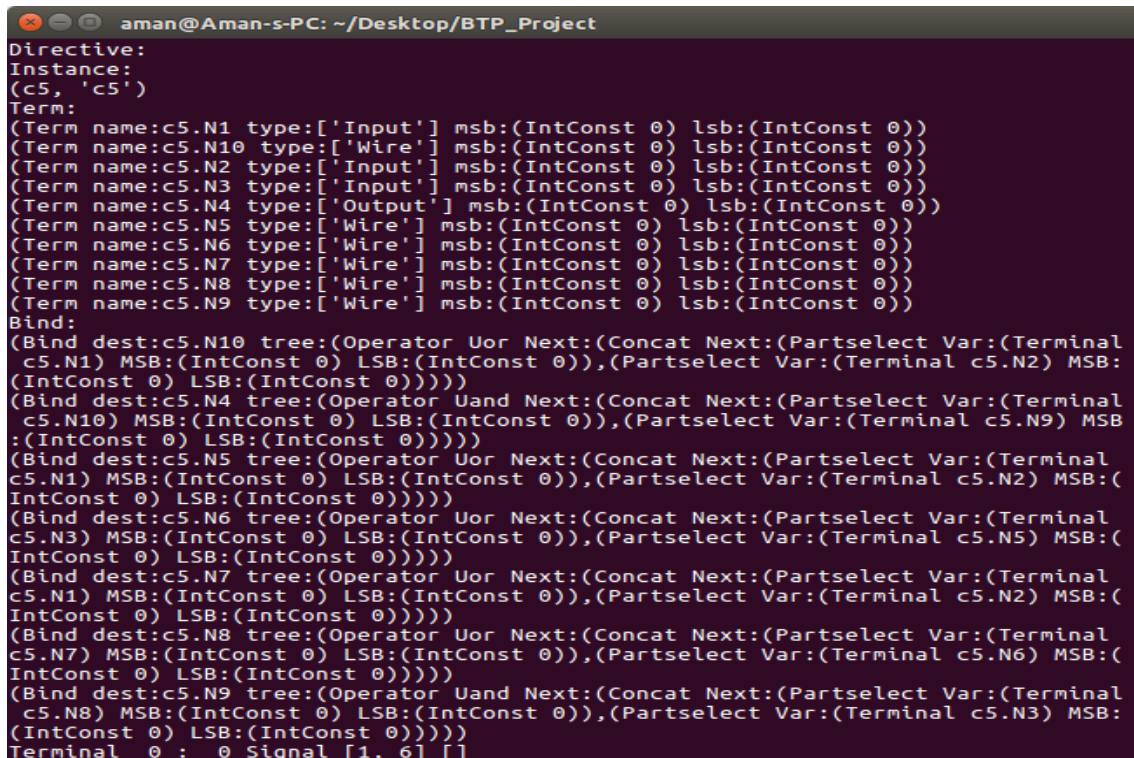
Screenshots of Project

Figure 5.1: Command to execute the program

The first argument after python is the project file name. The one after -t is the module name and next one is the input verilog file.

Figure 5.2: Verilog file parsing using pyverilog

Figure 5.3: Circuit tree representation

The above figure depicts the two lists, namely terminal and gate lists. For each terminal element, we have the name, type, parent gate list and child gate list for the terminal. In case of gates, we have operator, destination terminal and the input terminal list of the respective gate.

Figure 5.4: Expression for the current tree representation

Chapter 6

Contribution

My contributions to the project are listed as follows:

- Implemented and tested the Majority (Ω.M) axiom.

- Implemented and tested the Associativity (Ω.C) axiom.

- Implemented and tested the Complementary Associativity (Ψ.C) axiom.

- Implemented the Algebraic Size Optimization, and analyzed the effect.

- Implemented the Algebraic Depth Optimization, and analyzed the effect.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

In this project, we analyzed a new method of representing and optimizing logic circuits, by using only MAJ and INV as basic operations. We utilized the MIGs: a DAG consisting of three-input majority nodes and regular/complemented edges. We implemented algebraic and boolean optimization techniques for MIGs in order to increase the efficiency of the digital integrated circuits, and we compared the optimized MIG representations with the AIG represensions.

7.2 Future Scope

In this project, we implemented the boolean algebraic transformations and the optimization algorithms, but we look forward to extensively testing the program on increasingly large circuit files. We aim to optimize our code to facilitate faster processing of the verilog files.

References

[1] *Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization*; Luca AmarÂt'u, Pierre-Emmanuel Gaillardon, Giovanni De Micheli

[2] *Synthesis and Optimization of Digital Circuits*; G. De Micheli

[3] *Multiple-valued minimization for PLA optimization*; R.L. Rudell, A. Sangiovanni-Vincentelli

[4] *MIS: A Multiple-Level Logic Optimization System*; R.K. Brayton

[5] *A System for Sequential Circuit Synthesis*; E. Sentovich

[6] *ABC: An Academic Industrial-Strength Verification Tool*; R. Brayton, A. Mishchenko

[7] *Graph-based algorithms for Boolean function manipulation*; R.E. Bryant

[8] *BDS: A BDD-Based Logic Optimization System*; C. Yang and M. Ciesielski

[9] *Device and Architecture Outlook for Beyond CMOS Switches*; K. Bernstein

[10] *InP-based high-performance logic elements using resonant-tunneling devices*; K. J. Chen

[11] *Logical devices implemented using quantum cellular automata*; P. D. Tougaw, C. S. Lent

[12] *Polarity control in Double-Gate, Gate-All-Around Vertically Stacked Silicon Nanowire*; M. De Marchi

[13] *Median algebra*; John R. Isbell

[14] *The Art of Computer Programming*; D. Knuth

[15] *Switching Theory for Logic Synthesis*; T. Sasao

[16] *Lattice Theory*; G. Birkhoff

[17] *A ternary operation in distributive lattices*; G. Birkhoff

[18] *The Theory of Representations of Boolean Algebras*; M. Stone

[19] *Domain theory in logical form*; S. Abramsky