

國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



眾數反向圖改寫技術在邏輯合成與驗證之應用

Application of DAG-Aware MIG Rewriting Technique in  
Logic Synthesis and Verification

王立為

Li-Wei Wang

指導教授：黃鐘揚 教授

Advisor: Chung-Yang (Ric) Huang, Ph.D.

中華民國一百零五年七月

July, 2016

# 誌謝



能完成這份論文，我要感謝我的指導教授黃鐘揚老師，他培養了我對驗證領域的興趣，並在碩士的兩年間打下深厚的基礎。感謝楊明仁從大三時就各種罩我，一直記得大三下在網多實驗室趕EDA導論期末報告，那時候還很納悶為什麼電路一直要畫成三角形；感謝小熊、江弘勳在我碩一修課時提供各種協助，還不時能屁話一番，感謝范寬，每次聽完你的規畫後就發覺自己落後了一大截需要加把勁才行。最後我要感謝我的家人，感謝奶奶平常的照顧，感謝爸爸媽媽，總是放心讓我嘗試各種事情，讓我專心做研究。

# 摘要



眾數反向圖是近年來提出的一種邏輯電路表示法，他將邏輯電路用眾數函數與反向函數組合而成；他的代數與布林特性讓他在邏輯優化的操作上非常有效率，比起目前最先進的方法，眾數反向圖的演算法可以得到更佳的结果。在這篇論文中，我們將無圈有向改寫技術鑲嵌進眾數反向圖，並將其應用在邏輯合成與驗證領域。在邏輯合成方面，實驗結果顯示高度優化的眾數反向圖仍可被我們的演算法再優化；在資料路徑驗證方面，我們的演算法可以提高資料路徑分析的品質，並有效的減少正規驗證所需的時間。

關鍵字: 眾數反向圖、邏輯合成、資料路徑驗證

## Abstract



A Majority-Inverter Graph (MIG) is a recently introduced logic representation form which manipulates logic by using only 3-input majority function (MAJ) and inversion function (INV). Its algebraic and Boolean properties enables efficient logic optimizations. In particular, MIG algorithms obtained significantly superior synthesis results as compared to the state-of-the-art approaches based on AND-inverter graphs and commercial tools. In this thesis, we integrate the DAG-aware rewriting technique, a fast greedy algorithm for circuit compression, into MIG and apply it not only in the logic synthesis but also verification. Experimental results on logic optimization show that heavily-optimized MIGs can be further reduced by 20.4% of network size while depth preserved. Experimental results on datapath verification also show the effectiveness of our algorithm. With our MIG rewriting applied, datapath analysis quality can be improved with the ratio 3.16. Runtime for equivalence checking can also be effectively reduced.

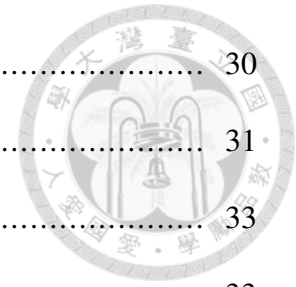
**Index Terms** - Majority-Inverter Graph, Logic Synthesis, Datapath Verification

# Table of Contents



摘要 .....	ii
Abstract .....	iii
<b>Chapter 1 Introduction</b> .....	<b>1</b>
<b>Chapter 2 Preliminaries</b> .....	<b>3</b>
2.1 Function Classification .....	3
2.1.1 The Concept of P Class .....	3
2.1.2 The Concept of NPN Class .....	3
2.2 Formal Verification .....	4
2.2.1 Combinational Equivalence Checking .....	5
2.2.2 Datapath Analysis .....	7
2.3 Previous Work on Majority-Inverter Graph .....	10
2.3.1 MIG Logic Representation .....	11
2.3.2 MIG Boolean Algebra .....	12
2.3.3 Logic Optimization Using Algebraic Transformation .....	14
2.3.4 MIG Boolean Methods and Logic Optimization .....	16
<b>Chapter 3 DAG-Aware MIG Rewriting</b> .....	<b>20</b>
3.1 An Overview of Our Framework .....	20
3.2 Subgraph Preparation Phase .....	21
3.2.1 Exact Synthesis .....	21
3.2.2 Subgraph Generation .....	26
3.2.3 Subgraph Storage .....	28
3.3 Rewriting Phase .....	30

3.3.1	Subgraph Loading .....	30
3.3.2	Subgraph NPN Manipulation .....	31
<b>Chapter 4</b>	<b>Application of DAG-Aware MIG Rewriting .....</b>	<b>33</b>
4.1	From the Perspective of Logic Synthesis .....	33
4.1.1	Rewriting Methodology for Logic Optimization .....	33
4.1.2	A Demonstration Example.....	34
4.2	From the Perspective of Datapath Verification .....	36
4.2.1	Motivation .....	36
4.2.2	Datapath Normalization .....	38
<b>Chapter 5</b>	<b>Experimental Results .....</b>	<b>41</b>
5.1	Logic Synthesis Results.....	41
5.1.1	Methodology .....	41
5.1.2	Optimization Results .....	42
5.2	Datapath Verification Results.....	44
5.2.1	Methodology .....	44
5.2.2	Verification Results.....	47
<b>Chapter 6</b>	<b>Conclusions and Future Work .....</b>	<b>50</b>
	Reference .....	51



# List of Figures



2.1	Combinational Equivalence Checking .....	6
2.2	Typical Verification Flow .....	8
2.3	Verifying Arithmetic Circuit With Datapath Analysis.....	9
2.4	Examples of MIG representations .....	12
2.5	MIG Boolean method .....	19
3.1	Flowchart of Our Framework.....	20
3.2	Optimal MIG for $S_{0,2}(x_1, x_2, x_3, x_4)$ .....	25
3.3	Example of Different Subgraphs for $S_{0,2}(x_1, x_2, x_3, x_4)$ .....	28
3.4	Subgraph Storage Table.....	30
3.5	NPN Relation Mapping .....	31
4.1	MIG Rewriting Example .....	35
4.2	MIG representation for a full adder .....	37
5.1	Synthesis Methodology .....	42
5.2	Arithmetic Design Generation Flow .....	46

# List of Tables



2.1	Number of P Classes and NPN Classes for 1-4 Input Functions.....	4
3.1	Optimal MIGs For All 4-Variables NPN Classes .....	25
5.1	Technology-Independent MIG Optimization Results.....	43
5.2	MIG Optimization Results After Technology Mapping.....	45
5.3	Arithmetic Benchmark Detail Description .....	47
5.4	Datapath Verification Results of Easier Benchmark With MIG Rewriting ...	48
5.5	Datapath Verification Results of Harder Benchmark With MIG Rewriting ..	49



# Chapter 1 Introduction



As semiconductor technology relentlessly pursues the path described by Moore's law, the challenges of SoC design continue to grow dramatically. We are moving from chips with millions of gates to ones with billions of gates. The task of designing such complex systems is daunting. Among different stages of design flow, logic synthesis and verification can be regarded as the most crucial one. In this context, efficient representation of Boolean functions are key features and many effective data structures and algorithms have been proposed for these tasks [1,2].

In the recent years, a novel homogeneous logic representation named Majority-Inverter Graph (MIG) has been proposed [3,4], which operates logic by using only majority (MAJ) and inversion (INV). The major advantage of homogeneous logic representations is that they simplify manipulation algorithms significantly and particularly enable an efficient implementation of them. A complete axiomatic and algebraic transformation system is proposed to support manipulation on MIGs. To extend the capabilities, MIG Boolean methods are also exploited. MIG algebraic and Boolean methods together attain better optimization quality compared to And-Inverter Graph (AIG). Because of these reasons, MIG is believed to be a promising data structure to push the progress of logic synthesis further.

In this thesis, we integrate the DAG-aware rewriting technique [5,6], a fast greedy algorithm for circuit compression proposed for years, into MIG and apply it not only in the logic synthesis but also verification dimension. The main idea of DAG-aware rewriting is to enumerate all 4-input subgraphs rooted at a given node and replace it with a precomputed advantageous subgraph. In order to find 4-input subgraphs, we adopt a SMT-based

algorithm to automate the process of subgraph generation. Then, various MIG rewriting strategies are proposed to achieve different goals.



In summary, the main contributions of this thesis are:

- Strengthen the SMT-based algorithm to find minimum-size subgraphs
- Combine the DAG-aware rewriting algorithm with MIGs
- Propose different rewriting strategies to fit different targets
- Better synthesis results compared to the previous works on MIG
- Pioneer in applying MIGs to the domain of verification

The remainder of this thesis is organized as follows. Chapter 2 provides some significant preliminaries related to this thesis. Chapter 3 describes our framework and DAG-aware MIG rewriting algorithm. Chapter 4 introduces how to apply our rewriting algorithm to logic synthesis and verification domains. Chapter 5 shows the experimental results. Chapter 6 concludes the thesis.

# Chapter 2 Preliminaries



## 2.1 Function Classification

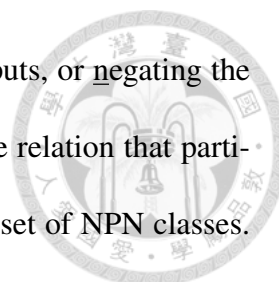
For a given number of input variables, there is a well-defined number of Boolean functions. This number is given by  $2^{2^n}$ , where  $n$  is the number of input variables. In order to reduce the search space of functions, they are classified into different classes set of functions. In this section, we review some concepts for classification of  $n$ -input Boolean functions, including P and NPN equivalence classes [7].

### 2.1.1 The Concept of P Class

The fact that many different 2-input functions may have the same gate-level implementations naturally introduces the concept of P equivalence. Two functions  $f(x_1, \dots, x_n)$  and  $g(x_1, \dots, x_n)$  are P-equivalent if there exists a permutation  $\sigma \in S_n$  such that  $f(x_1, \dots, x_n) = g(x_{\sigma(1)}, \dots, x_{\sigma(n)})$ , i.e.,  $g$  can be made equivalent to  $f$  by permuting inputs. P-equivalence is an equivalence relation that partitions the set of all Boolean functions over  $n$  variables into a smaller set of P classes. For example, there are  $2^{2^2} = 16$  different 2-input functions while there are only 12 different 2-input P classes. Among these 12 classes, four of them are composed by 2 functions and eight of them are composed by only one function.

### 2.1.2 The Concept of NPN Class

Two functions  $f(x_1, \dots, x_n)$  and  $g(x_1, \dots, x_n)$  are NPN-equivalent if there exists a permutation  $\sigma \in S_n$  and polarities  $p, p_1, \dots, p_n$  in  $\mathbb{B}$  such that  $f(x_1, \dots, x_n) = g^p(x_{\sigma(1)}^{p_1}, \dots, x_{\sigma(n)}^{p_n})$ ,



i.e.,  $g$  can be made equivalent to  $f$  by negating inputs, permuting inputs, or negating the output. Similar to P-equivalence, NPN-equivalence is an equivalence relation that partitions the set of all Boolean functions over  $n$  variables into a smaller set of NPN classes. As an example all  $2^{2^n}$  Boolean functions over  $n$  variables can be partitioned into 2, 4, 14, 222, 616126 NPN classes for  $n = 1, 2, 3, 4, 5$ . As the representative of each NPN class, we take the function with the smallest truth table, when truth tables are viewed as a binary number of  $2^n$  bits. Number of P classes and NPN classes is summarized in Table 2.1.

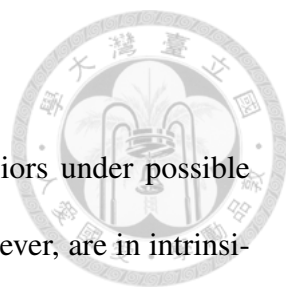
Table 2.1: Number of P Classes and NPN Classes for 1-4 Input Functions

# of inputs	functions	P classes	NPN classes
1	4	4	2
2	16	12	4
3	256	80	14
4	65536	3984	222

## 2.2 Formal Verification

In the design flow, to prevent the situation that bugs escape from our detection and buggy designs are passed to the later flow, *formal verification* [8,9] are developed rapidly in the past twenty years. To be more specific, a technique called equivalence checking [10] plays an important role to check if two circuits at different design stages exhibit the same behaviors. Formal engines such as binary decision diagram (BDD) [11] and boolean satisfiability solver (SAT) [12] are therefore extensively used in the verification domain. In this section, we briefly introduce the combinational equivalence checking (CEC) technique, which is an essential step in the modern design flow. Then, a circuit analysis technique for the arithmetic design, called *datapath analysis* [13], is discussed.

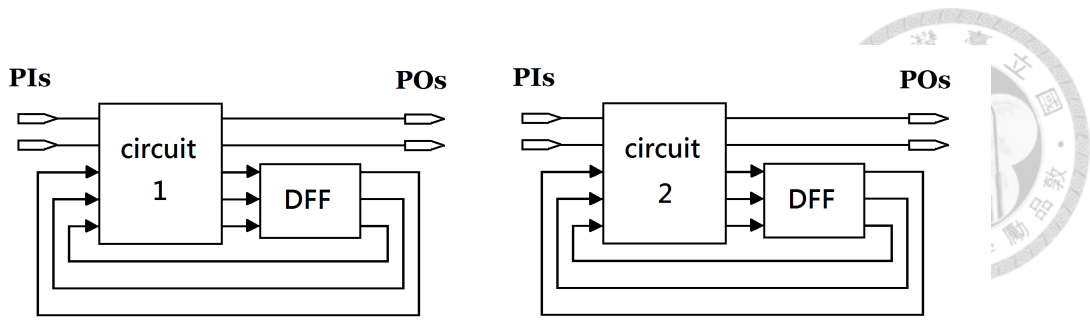
## 2.2.1 Combinational Equivalence Checking



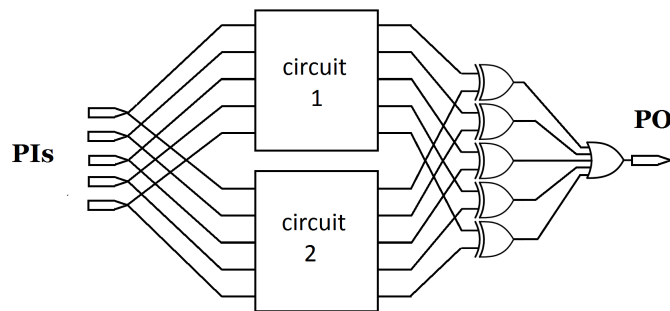
The process to prove that two circuits exhibit the same behaviors under possible condition is called equivalence checking. Most modern designs, however, are intrinsically sequential. This sequential property makes the equivalence checking process almost impossible in the reasonable time due to complexity issue. To avoid the problem, we usually adopt the assumption of *combinational equivalence*, i.e., comparing not only outputs but also registers, which is valid for most logic optimization. Under the assumption, the equivalence checking problem is deeply simplified and the process to prove the property under the assumption is called combinational equivalence checking. Since the scalability of BDD is much poorer than SAT and almost cannot be applied to modern design, only the SAT-based CEC is discussed in the following paragraph.

To embed SAT solver as the formal engine in the CEC process, two circuits are modeled as a combinational miter and formulated as a decision problem asking if there is an input assignment such that the output would be one. The combinational miter is constructed from two circuits to be checked by **sharing corresponding PIs together** and **adding an XOR gate on corresponding POs pairs**. Connecting the outputs of all XOR gates to an OR gate drives the only one output of the combinational miter. As our previous discussion on combinational equivalence assumption, registers' inputs are regarded as pseudo primary outputs (PPOs) and registers' outputs are regarded as pseudo primary inputs and connected to their counterparts. Figure 2.1 shows a combinational miter.

Then, Tseitin transformation [14] is applied to produce CNFs from the combinational miter. By querying the solver whether there is an input assignment such that the



(a) Two circuits to be checked



(b) Combinational miter

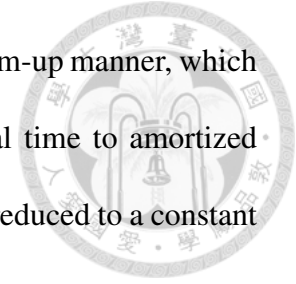
Figure 2.1: Combinational Equivalence Checking

output is true, we can conclude that the two circuits are equivalent or not. If the SAT solver returns a satisfiable result, we know that the two circuits are not equivalent and can fetch the input assignment from the solver to find out the cause of their difference. If the SAT solve return a unsatisfiable result, i.e., there is no input assignment such that the output is true, we can say that two circuits are formally proven to be combinational equivalent.

There are a lot of techniques proposed for speeding up SAT-based CEC process in the past years [10,15]. Among them, *functionally reduced AND-INV graphs* [16], abbreviated as *FRAIG*, is the most effective algorithm to speed up the process. The algorithm is summarized in Alg. 1.

In FRAIG, simulation technique is integrated to detect potential internal equivalence, referred to as functionally equivalent candidates (FECs), in the circuit. With these FECs,

we can guide the SAT solver to learn and prove the property in a bottom-up manner, which strongly alleviates the complexity of SAT solving from exponential time to amortized constant time. Upon the FRAIG completed, the miter circuit will be reduced to a constant zero if the two circuits are combinational equivalent.




---

**Algorithm 1** Functionally Reduced And-Inverter Graph

---

```

1: function FRAIG(Network miter)
2:   solver  $\leftarrow$  initProofModel(miter)
3:   classes  $\leftarrow$  randomSimulate(miter)
4:   for each node in the miter in a topological order do
5:     // collect functionally equivalent candidate
6:     fec  $\leftarrow$  getFEC(classes, node)
7:     if fec = null then continue
8:     for each candidate C in fec do
9:       if satCheckEquivalent(solver, node, C) = UNSAT then
10:        // interanl equivalence
11:        mergeEqualPair(miter, node, C)
12:      else
13:        pattern  $\leftarrow$  getSatPattern(solver)
14:        classes  $\leftarrow$  simulateBySatPattern(pattern)

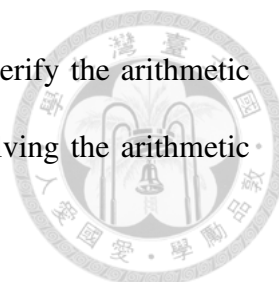
```

---

## 2.2.2 Datapath Analysis

In the previous section, we briefly introduce how to verify two circuits by CEC. In fact, with effective SAT-based CEC techniques, modern designs with more than a million gates can be verified within an hour.

Nevertheless, there is still a situation that CEC can not be finished in a reasonable time. To be more specific, the problem of verifying non-standard, bit-optimized embedded *arithmetic circuits* remains open [17]. Importance of arithmetic verification problem grows with an increased use of arithmetic modules in embedded systems to perform computation intensive tasks in multimedia, signal processing, and cryptography applications.



In this section, we will demonstrate why SAT-based CEC fails to verify the arithmetic design. Then a acceptable method, called *datapath analysis* for solving the arithmetic design verification problem is introduced.

To begin with, we show the typical verification flow in the figure 2.2. The target circuits under verification are RT-Level design and optimized gate-level netlist, labeled with *golden* and *revised* respectively. We want to verify that whether there are mistakes in the process of logic synthesis or not. To convert the golden design into a logic network, quick synthesis is applied to the golden design. Note that no optimization technique is integrated in this step. Also, *proof by construction* technique guarantees the correctness of the process.

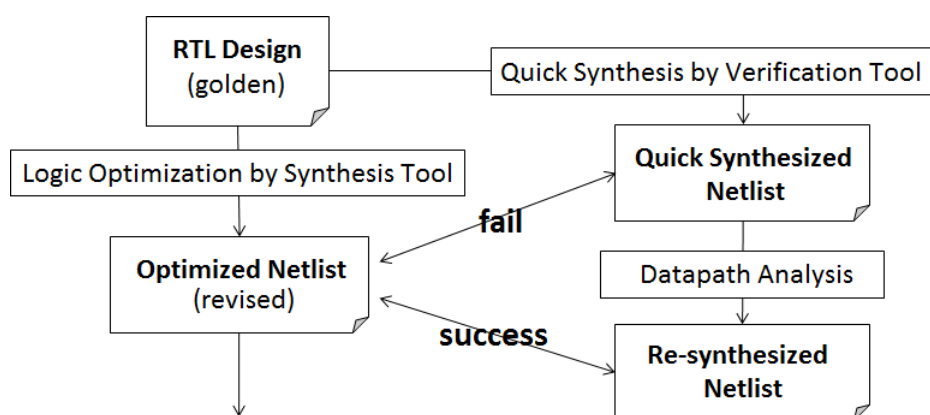


Figure 2.2: Typical Verification Flow

In most cases, we can apply CEC now to know whether the designs are equivalent or not. As mentioned above, however, verification for arithmetic designs often fail in this step. To avoid sticking into the situation, *datapath analysis* method is recommended to be applied before CEC.

To explain why the CEC fails and how datapath analysis resolves the problem, we demonstrate a simple example in figure 2.3.



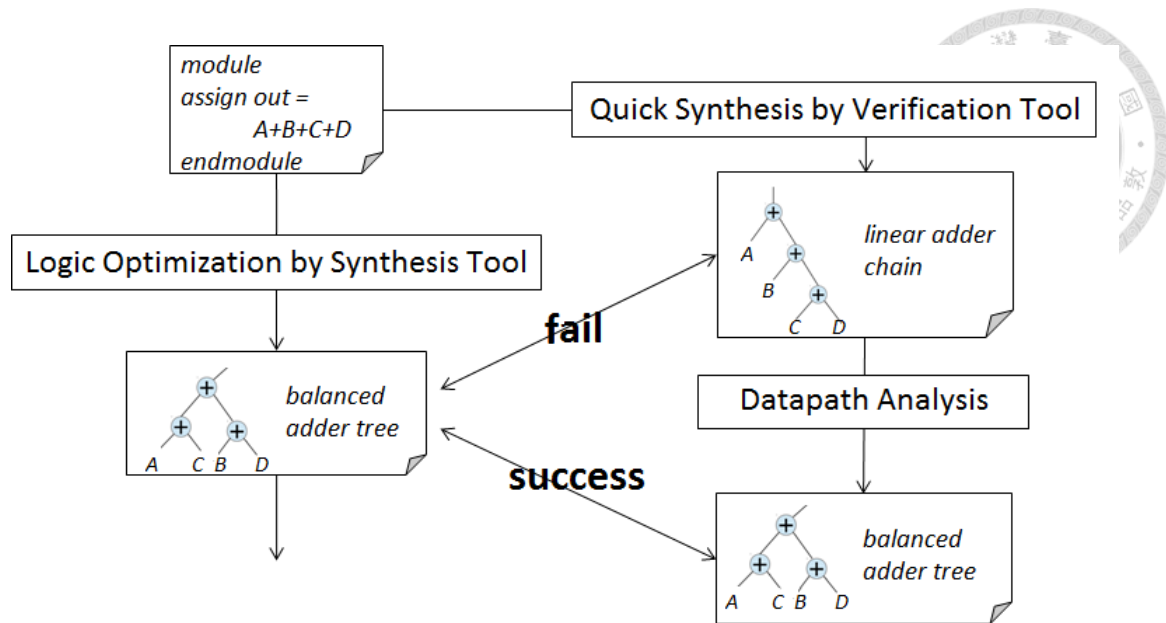
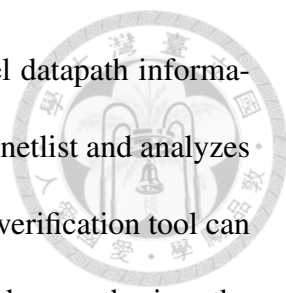


Figure 2.3: Verifying Arithmetic Circuit With Datapath Analysis

We can see that the golden design is a simple adder for 4 input with arbitrary bit width. After logic synthesis, the optimized netlist is in a balanced addition tree structure. Also, input delay may be specified by the designer so the addition ordering is not as straightforward as presented in the golden design. On the other hand, the verification tool makes a quick synthesis according to the golden design and simply construct a linear adder chain.

From the perspective of logic network, the similarity between these two circuits is extremely poor, that is, there isn't any internal equivalence between two designs. The only equivalent points are the outputs, and the SAT solver has no choice but to solve the solid combinational miter from scratch without any implication guidance, leading to an inevitable exponential result.

We know that the main reason is the great difference between two designs and that there is still some useful information neglected in the flow. With those information, the verification tool can synthesize a possibly more similar netlist for CEC. To be more spe-



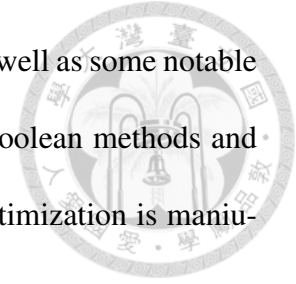
cific, in figure 2.3, *datapath analysis* method collects both world-level datapath information in RTL design and circuit structure information in the optimized netlist and analyzes them with some effective techniques. After analysis, if succeeds, the verification tool can conclude that the addition tree is in the form of  $((A+C)+(B+D))$  and re-synthesizes the golden RTL design according to the analysis result. Finally, the CEC can be executed in a smoother manner.

As the previous example presented, it sounds that with the *datapath analysis* method, arithmetic verification problem can be resolved easily. In some cases, however, the analysis algorithm fails to conclude the exact datapath structure of the optimized netlist due to some difficulties such as excessive optimization or arithmetic points missing. We will discuss more detail in the chapter 4.

## 2.3 Previous Work on Majority-Inverter Graph

A *Majority-Inverter Graph* (MIG) is a recently introduced logic representation manipulating the ternary majority function as logic operation. Due to their simplicity and homogeneity, MIGs simplify manipulation algorithms significantly and particularly enable an efficient implementation of them. A consistent MIG algebraic system is introduced to support logic manipulation. By using these algebra axioms, it is possible to reach all points in the representation space. Also, MIG Boolean methods such as error insertion and don't care condition are proposed to enable logic optimization with a global view. MIG algebraic and Boolean methods together attain better synthesis quality. In particular, when considering logic depth reduction, MIG algorithms obtained superior synthesis results as compared to the state-of-the-art approaches based on And-Inverter Graph (AIG).

This section gives a background on MIG logic representation as well as some notable properties. Then, MIG algebraic methods are presented. Finally, Boolean methods and optimization methodology is introduced to understand how logic optimization is manipulated under a MIG framework.



### 2.3.1 MIG Logic Representation

**Definition** A MIG is homogeneous logic network with indegree equal to 3 and with each node representing the majority function. In a MIG, edges are marked by a regular or complemented attribute.

We explore the properties of MIGs by comparison to the general AND/OR/Inverter Graphs (AOIGs). Note that 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. This property leads to the following theorem.

*Theorem 2.1:* MIGs  $\supset$  AOIGs.

As a corollary of Theorem 2.1, MIGs also include AIGs and are capable to represent any logic function, a important property known as universal representation.

*Corollary 2.2:* MIGs  $\supset$  AIGs.

*Corollary 2.3:* MIG is an universal representation form.

Fig. 2.4 depicts two logic representation examples for MIGs. They are obtained by translating their optimal AOIG representations into MIGs. Note that even if such logic networks are optimal for AOIGs, they can be further optimized with MIGs, as detailed later.

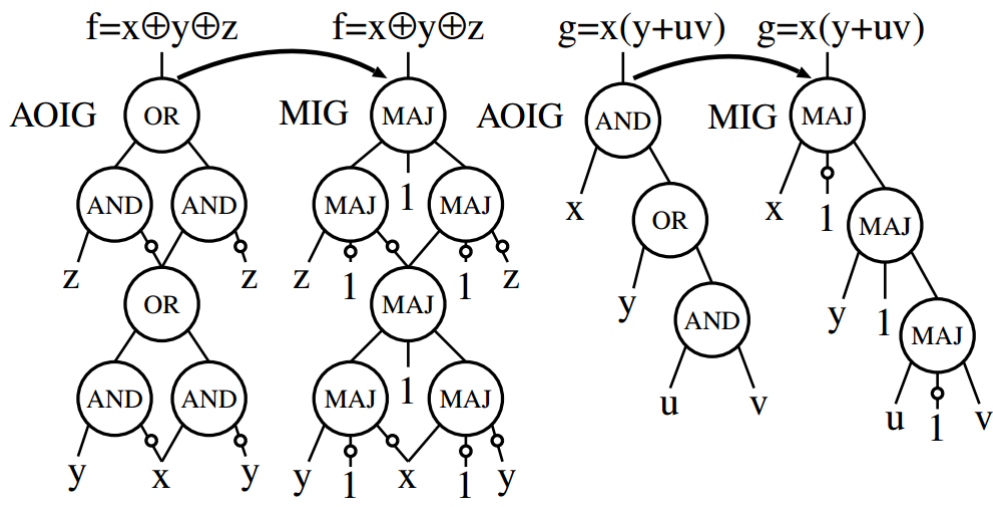


Figure 2.4: Examples of MIG representations

So far, we have shown that MIGs can be configured to behave as AOIGs. Hence, in principle, they can be manipulated using traditional AND/OR techniques. However, the potential of MIGs goes beyond standard AOIGs and, in order to unlock their full expressive power, a new Boolean algebra is introduced, natively supporting the majority/inverter functionality.

### 2.3.2 MIG Boolean Algebra

A novel Boolean algebra is proposed, defined over the set  $(\mathbb{B}, M, ', 0, 1)$ , where  $M$  is the majority operator of three variables and  $'$  is the complementation operator. The following set of five primitive transformation rules, referred to as  $\Omega$ , is an *axiomatic system* for  $(\mathbb{B}, M, ', 0, 1)$ . All the variables considered hereafter belong to  $\mathbb{B}$ .

It can be proved that  $(\mathbb{B}, M, ', 0, 1)$  axiomatized by  $\Omega$  is a Boolean algebra by showing that it induces a complemented distributive lattice. Since being familiar with MIGs in a short time is the goal for this section, we omit some formal proof here.



$$\Omega \left\{ \begin{array}{l}
 \text{Commutativity - } \Omega.C \\
 M(x, y, z) = M(y, x, z) = M(z, y, x) \\
 \text{Majority - } \Omega.M \\
 \left\{ \begin{array}{l}
 \text{if } (x = y) : M(x, y, z) = x = y \\
 \text{if } (x = y') : M(x, y, z) = z
 \end{array} \right. \\
 \text{Associativity - } \Omega.A \\
 M(x, u, M(y, u, z)) = M(z, u, M(y, u, x)) \\
 \text{Distributivity - } \Omega.D \\
 M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z) \\
 \text{Inverter Propagation - } \Omega.I \\
 M'(x, y, z) = M(x', y', z')
 \end{array} \right. \quad (2.1)$$

*Theorem 2.4:* The set  $(\mathbb{B}, M, ', 0, 1)$  subject to axioms in  $\Omega$  is a Boolean algebra.

Note that there are other possible axiomatic systems. For example, it is possible to show that in the presence of  $\Omega.C$ ,  $\Omega.A$  and  $\Omega.M$ , the rule in  $\Omega.D$  is redundant. However,  $\Omega.D$  is still included for the sake of completeness.

*Theorem 2.5:* The Boolean algebra  $(\mathbb{B}, M, ', 0, 1)$  axiomatized by  $\Omega$  is sound and complete.

*Theorem 2.6:* It is possible to transform any MIG  $\alpha$  into any other logically equivalent MIG  $\beta$ , by a sequence of transformations in  $\Omega$ .

Soundness property ensures that if a formula is derivable from the system, then it is valid. Completeness guarantees that each valid formula is derivable from the system. Theorem 2.5 can be proved by linking back to Stone's theorem. To explain Theorem 2.5 intuitively, we say that every  $(M, ', 0, 1)$ -formula can be interpreted as an MIG. Thus, the Boolean algebra induced by  $\Omega$  is naturally applicable in MIG manipulation.

Theorem 2.6 can be proved by Theorem 2.5. As a consequence of Theorem 2.6, any two equivalent MIGs can be transformed one into the other by  $\Omega$ . 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, a more powerful transformation system is introduced, referred to as  $\Psi$ .



$$\Psi \left\{ \begin{array}{l} \textbf{Relevance - } \Psi.R \\ M(x, y, z) = M(x, y, z_{x/y'}) \\ \textbf{Complementary Associativity - } \Psi.C \\ M(x, u, M(y, u', z)) = M(x, u, M(y, x, z)) \\ \textbf{Substitution - } \Psi.S \\ M(x, y, z) = \\ M(v, M(v', M_{v/u}(x, y, z), u), M(v', M_{v/u'}(x, y, z), u')) \end{array} \right. \quad (2.2)$$

The first, relevance ( $\Psi.R$ ), replaces and simplifies reconvergent variables. The second, complementary associativity ( $\Psi.C$ ), deals with variables appearing in both polarities. The third and last, substitution ( $\Psi.S$ ), extends variable replacement also in the non-reconvergent case. We represent a general variable replacement operation, say replace  $x$  with  $y$  in all its appearance in  $z$ , with the symbol  $z_{x/y}$ .

### 2.3.3 Logic Optimization Using Algebraic Transformation

The optimization of a MIG, representing a Boolean function, ultimately consists of its transformation into a difference MIG, with better figures of merit in terms of area(size), delay(depth), and power(switching activity). For the sake of clarity, we only present the heuristic algorithms to optimize the size. Optimization algorithms for depth and power can be achieved by a similar manner.

To optimize the size of a MIG, we aim at reducing its number of nodes, Node reduction can be done, at first instance, by applying the majority rule. In the novel Boolean algebra domain, that is the ground to operate on MIGs, this corresponds to the evaluation of the majority axiom ( $\Omega.M$ ) from *Left to Right* ( $L \rightarrow R$ ), as  $M(x, x, z) = x$ . A different

node elimination opportunity arises from the distributivity axiom ( $\Omega.D$ ), evaluated from *Right to Left* ( $R \rightarrow L$ ), as  $M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$ . By applying repeatedly  $\Omega.M_{L \rightarrow R}$  and  $\Omega.D_{R \rightarrow L}$  over an entire MIG, we can actually eliminate nodes and thus reduce its size. Note that the applicability of majority and distributivity depends on the peculiar MIG structure.

Indeed, there may be MIGs where no direct node elimination is evident. This is because (i) the optimal size is reached or (ii) we are stuck in a local minima. In the latter case, we want to reshape the MIG in order to enforce new reduction opportunities.

---

**Algorithm 2** MIG-size Optimization Pseudocode

---

```

1: function MIGALGEBRAICOPT(MIG  $\alpha$ )
2:   for cycles=0; cycles<effort; cycles++ do
3:      $\Omega.M_{L \rightarrow R}(\alpha)$ ;  $\Omega.D_{R \rightarrow L}(\alpha)$ ;
4:      $\Omega.A(\alpha)$ ;  $\Psi.C(\alpha)$ ;
5:      $\Psi.R(\alpha)$ ;  $\Psi.S(\alpha)$ ;
6:      $\Omega.M_{L \rightarrow R}$ ;  $\Omega.D_{R \rightarrow L}$ ;

```

---

The rationale driving the reshaping process is to locally increase the number of common inputs/variables to MIG nodes. For this purpose, the associativity axioms ( $\Omega.A$ ,  $\Psi.C$ ) allow us to move variables between adjacent levels and the relevance axiom ( $\Psi.R$ ) to exchange reconvergent variables. When a more radical transformation is beneficial, the substitution axiom ( $\Psi.S$ ) replaces pairs of independent variables, temporarily inflating the MIG. Once the reshaping process created new reduction opportunities, majority  $\Omega.M_{L \rightarrow R}$  and distributivity  $\Omega.D_{R \rightarrow L}$  run again over the MIG simplifying it. Reshape and elimination processes can be iterated over a user-defined number of cycles, called *effort*. Such MIG-size optimization strategy is summarized in Alg. 2.

### 2.3.4 MIG Boolean Methods and Logic Optimization



There are some alternatives to algebraic techniques, focusing on global properties of MIGs, such as voting resilience and don't care conditions. Due to their global and general nature, the optimization methods in this section is called "Boolean".

MIGs are hierarchical majority voting systems. One notable property of majority voting is the capability to correct various types of bit-errors. This feature is inherited by MIGs, where error masking can be exploited for optimization purposes. One way for doing so is to purposely introduce logic errors that are successively masked by the voting resilience in MIG nodes. If such logic errors are advantageous, in terms of circuit simplifications, better MIG representations appear. In the immediate following, the theoretical grounds for "safe error insertion" in MIGs is presented, defining what type of errors, and at what overhead cost, can be introduced. Later on, a intelligent procedures for "advantageous errors" insertion is proposed.

#### A. Inserting Safe Errors in MIG

Before entering into the core theory, we briefly review notations and definitions on logic errors.(reference)

**Definition** The logic error between an original function  $f$  and its faulty version  $g$  is the Boolean difference  $f \oplus g$ .

**Notation** A logic circuit  $f$  affected by an error  $A$  is written as  $f^A$ .

To insert safe (permissible) errors in a MIG, we consider a root node  $w$  and we triplicate it. In each version of  $w$  we introduce the three faulty versions of  $w$  to a top majority node exploiting the error masking property. Unfortunately, a majority node cannot mask



all types of errors. This limits our choice of permissible errors. *Orthogonal errors*, defined hereafter, fit with our purposes. Informally, two logic errors are *orthogonal* if for an input pattern they cannot happen simultaneously.

**Definition** Two logic errors  $A$  and  $B$  on a logic circuit  $f$  are said *orthogonal* if the following property holds:  $(f^A \oplus g) \cdot (f^B \oplus g) = 0$ .

Now consider a generic MIG root  $w$ . Say  $A$ ,  $B$ , and  $C$  three pairwise *orthogonal* errors on  $w$ . Being all pairwise *orthogonal*, a top majority node  $M(w^A, w^B, w^C)$  is capable to mask  $A$ ,  $B$ , and  $C$  errors restoring the original functionality of  $w$ . This is formalized in the following theorem.

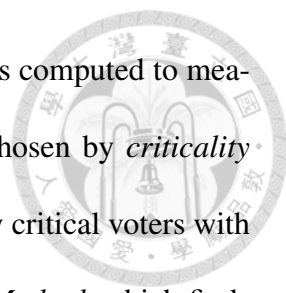
*Theorem 2.7:* Say  $w$  a generic node in a MIG. Say  $A$ ,  $B$  and  $C$  three pairwise *orthogonal* errors on  $w$ . Then the following equation holds:  $w = M(w^A, w^B, w^C)$

The theorem can be proved by showing that  $w \oplus M(w^A, w^B, w^C) = 0$  with the *orthogonal* error condition. Note that a MIG  $w = M(w^A, w^B, w^C)$  can have up to three times the size and one more level of depth as compared to the original  $w$ . This means that simplifications enabled by *orthogonal* errors  $A$ ,  $B$  and  $C$  must be significant enough to compensate for such overhead.

In the following, methods for identifying advantageous triplets of *orthogonal* errors is introduced briefly.

## **B. Identifying advantageous *orthogonal* errors**

A natural way to discover advantageous triplets of *orthogonal* errors is to analyze a MIG structure. We have to focus on nodes that have the highest impact on the final voting decision, i.e., influencing most a function computation. Two methods have been proposed



with this motivation. For the first method, a metric called *criticality* is computed to measure the importance for each node. Then, two critical voters are chosen by *criticality* accordingly. Finally, three pairwise *orthogonal* errors are identified by critical voters with a specific structure. The second method is called *Input Partitioning Method*, which finds *orthogonal* errors by input division. There is also a metric to determine how significant an input is.

For the sake of clarity, we comment on Boolean methods with a simple example using first Boolean method, reported in Fig 2.5. First, the critical voters are searched and identified, being in this example the input  $x_1$  and the node  $m_2$ . The proper error insertion root in this small example is the MIG root itself. So, three different versions of the root  $f$  are generated with errors  $f^{m_2/x_1'}$ ,  $f^{m_3/m_2}$  and  $f^{m_3/x_1}$ . Each faulty branch is handled by fast algebraic optimization to reduce its depth. The detailed algebraic optimization steps involved are shown in Fig 2.5. The most common operation is  $\Omega.M$  that directly simplifies the introduced errors. The optimized faulty 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, the MIG Boolean optimization techniques attain a depth reduction of 60% and, at the same time, a size reduction of 40%. On the other hand, by running just algebraic optimization on this example a depth reduction of 20% is possible at a size overhead cost of 50%.

In this chapter, we briefly introduce the Majority-Inverter Graph and some of its properties for Boolean logic optimization. Also, we review some concepts for classification of  $n$ -input Boolean functions, including P and NPN equivalence classes. With those preliminaries, we can start to introduce the core techniques proposed in this thesis, ref-

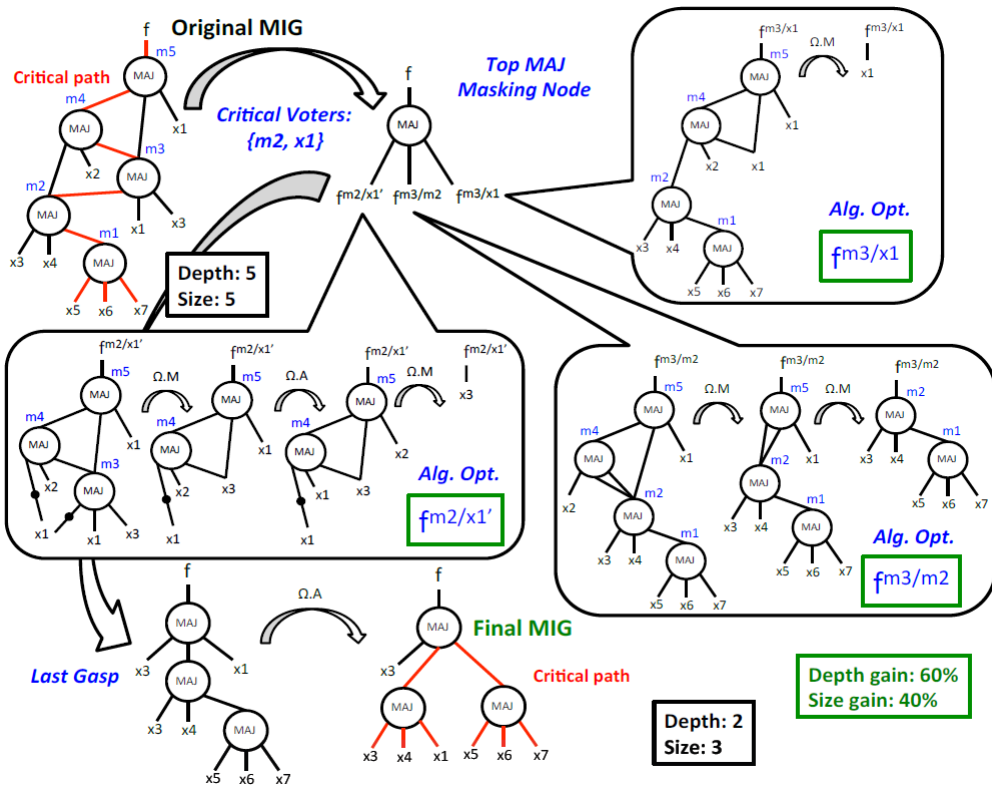


Figure 2.5: MIG Boolean method

ferred to as DAG-aware MIG rewriting, which performs fast and effective MIG rewriting by replacing original network with advantageous 4-input subgraphs. Improved experimental results show that our rewriting technique can be applied to both logic optimization and arithmetic design verification. We'll continue on introducing our core techniques in the next chapter.

# Chapter 3 DAG-Aware MIG Rewriting



## 3.1 An Overview of Our Framework

*DAG-Aware Rewriting* is a fast greedy algorithm for circuit compression first proposed in [5] and extended in [6]. The algorithm minimizes the AIG size by iteratively selecting AIG 4-input subgraphs rooted at a node and replacing them with smaller pre-computed subgraphs, while preserves the functionality of the root node. The same rewriting algorithm can be applied to MIG since it is also a homogeneous logic representation as AIG.

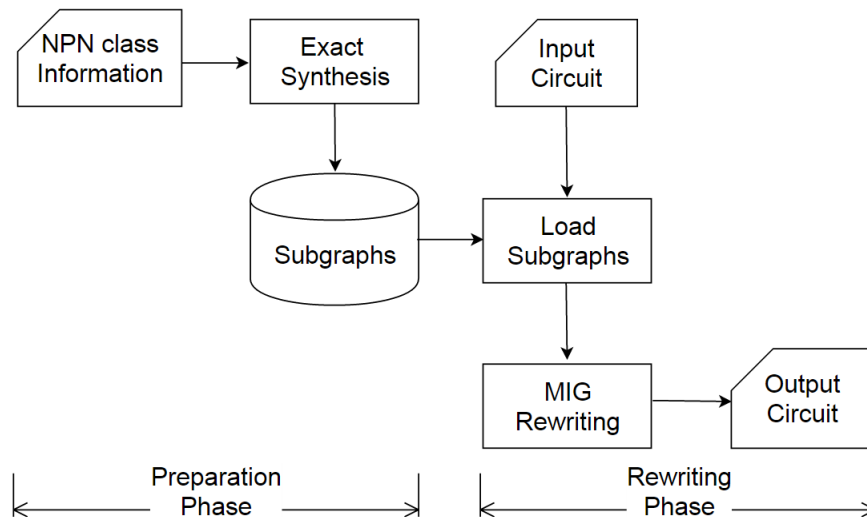
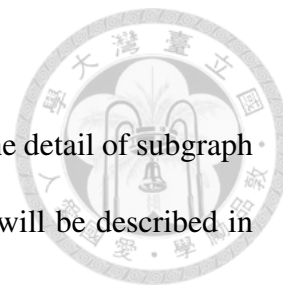


Figure 3.1: Flowchart of Our Framework

Figure 3.1 outlines our MIG rewriting framework. In the preparation phase, the exact synthesis engine reads a file containing NPN class information such as representative truth table for each class and generates subgraphs for all NPN classes. Then all computed subgraphs are compiled as an integer array so that the setup time of the rewriting engine can be noticeably reduced. In the rewriting phase, the engine performs MIG rewriting

according to the specific strategy and writes out the updated circuit.

The rest of this chapter is organized as follows: In section 3.2, the detail of subgraph preparation phase is presented. Then, algorithm in rewriting phase will be described in section 3.3.



## 3.2 Subgraph Preparation Phase

An essential question for DAG-aware rewriting is how to generate non-redundant 4-input subgraphs for 222 NPN equivalence classes. For AIG, pre-computation for subgraphs is achieved by using simple disjoint-support decomposition [18]. For MIG, however, there is no efficient method to compute non-redundant subgraphs because of the non-trivial majority functionality. Fortunately, a method called *exact synthesis* is proposed [19] to find minimal MIG representation based on *Satisfiability Modulo Theories* (SMT). We will describe the detail of *exact synthesis* first, then how MIG subgraphs are generated and stored is discussed.

### 3.2.1 Exact Synthesis

The problem to finding the smallest MIG w.r.t. the size for a given Boolean function is called *exact synthesis*. One way to find such a smallest MIG is to formulate a decision problem that asks whether there exists an MIG with  $k$  nodes that can represent  $f$ . To find a minimum solution, one starts by solving the decision problem for  $k = 0$  and increases  $k$  until a satisfying solution is found.

This section describes the formulation of exact synthesis as a decision problem that can be automatically solved with an SMT solver. The instance is a Boolean function

$f : \mathbb{B}^n \rightarrow \mathbb{B}$  and a non-negative constant  $k$ . In other words, the exact synthesis problem asks whether there exists an MIG

$$M = (\{x_1, \dots, x_n\} \cup \{g_1, \dots, g_k\} \cup \{0\}, E, \{y\})$$

with  $k$  majority operations that represents  $f$ . For the encoding of the SMT formula, we assume that  $k > 0$  and, in our algorithm, we check for the case  $k = 0$ , i.e.,  $f$  is constant functions or single variable functions, explicitly.

Each majority node with index  $l \in \{1, \dots, k\}$  is duplicated for each function value  $0 \leq j \leq 2^n$  and is represented by 10 variables:

- three inputs  $a_{1,l}^j, a_{2,l}^j, a_{3,l}^j \in \mathbb{B}$  of gate  $l$ ,
- one output  $b_l^{(j)} \in \mathbb{B}$  of gate  $l$ ,
- three select variables  $s_{1,l}, s_{2,l}, s_{3,l} \in \mathbb{B}^{\lceil \log_2(n+l) \rceil}$  that encode which are the child nodes of gate  $l$ ,
- polarity variables  $p_{1,l}, p_{2,l}, p_{3,l} \in \mathbb{B}$  that describe whether the edges to the child nodes are complemented.

The node indexes form a topological ordering of the nodes. The following constraints are contained in the SMT formula. Index  $j$  ranges from 0 to  $2^n - 1$  and index  $l$  ranges from 1 to  $k$ .

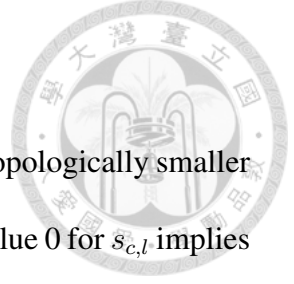
*Majority functionality:* The formula

$$b_l^{(j)} \leftrightarrow \langle a_{1,l}^{(j)} a_{2,l}^{(j)} \vee a_{1,l}^{(j)} a_{3,l}^{(j)} \vee a_{2,l}^{(j)} a_{3,l}^{(j)} \rangle$$

ensures the correct functionality of the node, i.e., the output value of the  $l^{th}$  node  $b_l^{(j)}$  is the majority of the node's three input values  $a_{1,l}^{(j)}$ ,  $a_{2,l}^{(j)}$ , and  $a_{3,l}^{(j)}$  for all assignments  $j$ .

*Input connections:* Constraints on the input connections are given in terms of implications of the select variables  $s_{c,l}$ , where  $c$  ranges from 1 to 3. The formula





$$s_{c,l} < n + l$$

ensures that node inputs can only be the constant, primary inputs, or topologically smaller nodes. In other words, the constraint prohibits cycles in the MIG. A value 0 for  $s_{c,l}$  implies a connection to the constant node, i.e.,

$$(s_{c,l} = 0) \rightarrow (a_{c,l}^{(j)} = \bar{p}_{c,l})$$

Values from 1 to  $n$  imply a connection to a variable node, i.e.,

$$(s_{c,l} = i) \rightarrow (a_{c,l}^{(j)} = bv(j)_{i-1} \oplus \bar{p}_{c,l}) \text{ for } 1 \leq i \leq n,$$

where  $bv(j)_{i-1}$  refers to the  $(i-1)^{th}$  bit in the binary representation of  $j$ . All other values to  $s_{c,l}$  imply a connection to the output of the corresponding majority node, i.e.,

$$(s_{c,l} = n + i) \rightarrow (a_{c,l}^{(j)} = b_i^{(j)} \oplus \bar{p}_{c,l}) \text{ for } 1 \leq i \leq l,$$

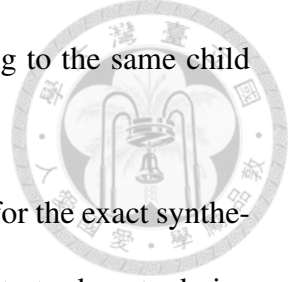
*Function semantics:* Finally, the function semantics is ensured by formula

$$b_k^{(j)} = \bar{p} \oplus f(j)$$

Note that  $k$  is the target node index and therefore refers to the root node. Since the majority operation is self-dual, the variable  $p$  can be omitted and a minimum solution is still found if one exists.

*Symmetry breaking:* All the above constraints ensure a correct result in case of a satisfying assignment. In order to reduce the search space, we exploit associativity of the majority operation and add the following symmetry breaking formula to enforce a unique order of the operands:

$$(s_{1,l} < s_{2,l}) \wedge (s_{2,l} < s_{3,l})$$



Note that there cannot be two edges of a majority node pointing to the same child node in an irreducible MIG.

The connections between the constraints and the resulting MIG for the exact synthesis problem are summarized in the following theorem which also illustrates how to derive the MIG from a satisfying assignment to the variables.

*Theorem 3.1:* Let  $f : \mathbb{B}^n \rightarrow \mathbb{B}$  and  $k$  be an instance to the exact synthesis problem. If there exists a satisfying solution to the corresponding SMT instance as described in this section, then let

$$M = (\{x_1, \dots, x_n\} \cup \{g_1, \dots, g_k\} \cup \{0\}, E, \{y\})$$

be the extracted MIG with:

$$E = \bigcup_{l=1}^k \bigcup_{c=1}^3 (g_l, \text{target}(s_{c,l}), p_{c,l}) \text{ and } y = (g_k, p)$$

and

$$\text{target}(s) = \begin{cases} 0 & \text{if } s = 0, \\ x_s & \text{if } 1 \leq s \leq n, \\ n_{s-n} & \text{if } n < s \leq n + k \end{cases}$$

Then,  $\Phi(y) = f$ .

Table 3.1 is adopted from [19] and summarizes the result of exact synthesis. The representative of the single most difficult NPN class is the symmetric function  $S_{0,2}(x_1, x_2, x_3, x_4)$ :

$$S_{0,2}(x_1, x_2, x_3, x_4) = \overline{x_1 \oplus x_2 \oplus x_3 \oplus x_4} \wedge \overline{x_1 x_2 x_3 x_4}$$

which requires 7 majority operations, depicted in figure 3.2.

In our experiments, the bottleneck of the exact synthesis flow is in the step of proving that there is no solution for a Boolean function when  $k$  is exactly less than the minimum feasible  $k$  by 1. For example, it takes about 16796 seconds to find the minimum number of



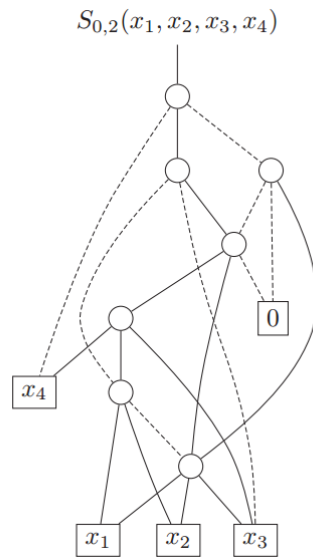
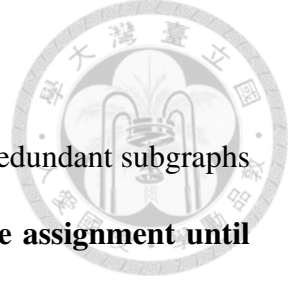


Figure 3.2: Optimal MIG for  $S_{0,2}(x_1, x_2, x_3, x_4)$

nodes for the function  $S_{0,2}(x_1, x_2, x_3, x_4)$ , while over 16000 seconds is used to prove that it is impossible to use 6 majority operations to represent the function. In this sense, we save the corresponding  $k$  for each class so that the bottleneck step can be easily avoided when we want to reproduce the result.

Table 3.1: Optimal MIGs For All 4-Variables NPN Classes

Majority Nodes	Classes	Functions	Time	Avg. time
0	2	10	0	0
1	2	80	0.04	0.02
2	5	640	0.14	0.03
3	18	3300	1.21	0.07
4	42	10352	3.32	0.15
5	117	40064	115.19	0.98
6	35	11058	1458.95	41.68
7	1	32	16796.3	16796.30
$\Sigma$	222	65536	18378.15	16839.23



### 3.2.2 Subgraph Generation

With the *exact synthesis* technique, we can simply find all non-redundant subgraphs for a given Boolean function by **iteratively blocking the satisfiable assignment until the SMT instance returns an unsatisfiable result.**

*Satisfiable assignment blocking:*

$$\neg(p \leftrightarrow model(p) \wedge \bigwedge_{l=1}^k \bigwedge_{i=1}^3 (s_{i,l} \leftrightarrow model(s_{i,l}) \wedge p_{i,l} \leftrightarrow model(p_{i,l}))),$$

where  $model(x)$  refers to the satisfiable assignment of variable of  $x$  stored in the SMT instance.

Note that we have recorded the minimum number of MIG nodes for each class, i.e.,  $k$ , so that we don't need to start from  $k = 1$  for each class.

Even though the bottleneck step is avoided, generating all possible subgraphs is still impractical for some classes. Take a closer look at the formulation of *exact synthesis*, the polarity variables for a node  $l$  can be complemented together to propagate a negation toward fanout region while preserving the functionality since the majority operation is self-dual, i.e.,  $\overline{M(x_1, x_2, x_3)} = M(\bar{x}_1, \bar{x}_2, \bar{x}_3)$ . Consequently, every majority operation can choose to propagate or not to propagate the negation, resulting  $O(2^k)$  subgraphs in the solution space.

To increase understanding of this problem, we take the parity function  $S_{1,3}(x_1, x_2, x_3, x_4) = x_1 \oplus x_2 \oplus x_3 \oplus x_4$  as an example. This function requires 6 majority operations to represent, which means there are  $2^6 = 64$  subgraphs for a specific node connection. Also, since the function is totally symmetric, the input variables can be permuted to generate  $4! = 24$  combinations. In fact, in our experiments, there are 184320 subgraphs in the solution space and it takes about a month for the SMT solver to complete the exhausted

search, which is obviously impractical for both number of subgraphs and runtime.

To alleviate this problem, we have to add more proper constraints to reduce the search space. Two methods are proposed to solve this problem, one is integrated in the formulation phase, the other is integrated in the blocking phase. For the first one, we restrict the first fanin of a majority node to be a regular edge, that is, a canonical condition for complemented edges is added in the formulation phase and can be stated as follow:

*Phase restriction:*

$$\bigwedge_{i=1}^k p_{1,i} \leftrightarrow 0$$

The second method is used to speed up the blocking phase. Originally, a satisfiable assignment consists of three select variables for each node, three polarity variables for each node and one variable for output polarity. Only a subgraph is blocked if this kind of satisfiable assignment is added to SMT instance. To avoid the situation of inverter propagation and block all subgraphs of the same node connection simultaneously, we can generalize our blocking formula as follow:

*Generalized satisfiable assignment blocking:*

$$\neg(p \leftrightarrow model(p)) \wedge \bigwedge_{l=1}^k \bigwedge_{i=1}^3 (s_{i,l} \leftrightarrow model(s_{i,l}))$$

Note that the generalized blocking formula does not impose the constraints on polarity variables, which means once a subgraph has the same node connection as the one found by the solver, no matter how its complemented edges are located, it is blocked by the generalized formula. Therefore, all subgraphs generated for a target NPN class must differ from the structure instead of redundant inverter propagation.

With the methods proposed above, we can efficiently generate desired non-redundant subgraphs with the specific complemented edges distribution (i.e. the first fanin of each

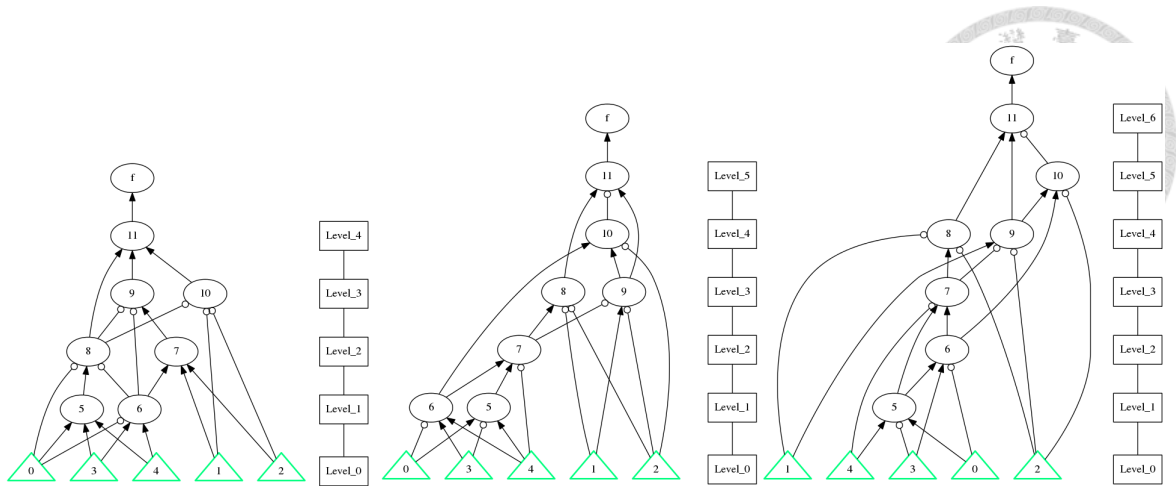


Figure 3.3: Example of Different Subgraphs for  $S_{0,2}(x_1, x_2, x_3, x_4)$

majority node must not be complemented). Figure 3.3 demonstrates three MIGs for the function  $S_{0,2}(x_1, x_2, x_3, x_4)$  as an example. Note that these three subgraphs are consisted of 7 majority nodes while the depths are all different.

There are still few NPN classes having too many subgraphs due to input permutation or structure variation. Under this circumstances, we have to set an upper bound for the number of subgraphs of these NPN classes in order to prevent unacceptable loading time in the rewriting phase. In our experiments, it is practical to set the upper bound number to 500 so that the runtime won't be too long while the variety of subgraphs is preserved.

### 3.2.3 Subgraph Storage

All subgraphs generated from exact synthesis engine will be written into a file for each NPN class containing subgraph structure information in the form of simple integers. Our MIG engine will read these 222 files and load them as a forest-like shared DAG containing approximately twenty eight thousand nodes. The output of each subgraph is marked because we have to load those subgraphs in the rewriting phase.

Note that the loaded DAG is in a shared form, that is, the node of the same fanin

combination (regular/complemented edges is considered) should not be duplicated. This sharing property is achieved by maintaining a hash table using the key computed by fanin condition. Then, each time a newly node is going to be created, a lookup function will check whether the node with this kind of fanin condition was created before or not and return a proper node either newly created or from hash table. The MIG lookup function is illustrated in the Alg. 3.

---

**Algorithm 3** MIG Lookup Function

---

```

1: function MIG_LOOKUP(fanin0, fanin1, fanin2)
2:   if fanin0 = fanin1 then return fanin0           ▷ if(x=y): M(x,y,z)=x
3:   if fanin0 = fanin2 then return fanin0
4:   if fanin1 = fanin2 then return fanin1
5:   if fanin0 = Not(fanin1) then return fanin2     ▷ if(x=y') :M(x,y,z)=z
6:   if fanin0 = Not(fanin2) then return fanin1
7:   if fanin1 = Not(fanin2) then return fanin0
8:   key ← HashKey(fanin0, fanin1, fanin2)
9:   for all node in HashTable[key] do
10:    if node's fanins are equal to fanin0, fanin1, fanin2 then
11:      return node
12:   return createNewNode(fanin0, fanin1, fanin2)

```

---

It is worth mentioning that the MIG lookup function is used not only in the subgraph construction but also any MIG circuit hereafter. We can see that majority axiom ( $\Omega.M$ ) is intrinsically embedded in the line 2 to 7 of the lookup function, which means there is no need to check the trivially redundant node in the MIG structure. Also, for the lookup's natural purpose, the node with the same fanin condition won't be duplicated.

Finally, our MIG engine writes the loaded DAG out as an integer array, which can be compiled into the program.



### 3.3 Rewriting Phase

In the rewriting phase, our MIG engine will read the circuit in aig format and convert it into MIG by direction transformation from AND node. Note that the MIG constructed by direct transformation is not compact since constant zero is connected to each node, therefore a lot of reduction opportunities are presented. On the other hand, the hash table for node lookup is maintained during the MIG construction so that we can guarantee there is no duplicated MIG node.

Before rewriting, we have to load subgraphs from the pre-compiled integer array. Besides, four NPN mapping arrays are maintained to achieve fast subgraph search and replacement. We will describe the details in the following two subsections.

#### 3.3.1 Subgraph Loading

We will initialize a table with 222 entries, i.e., each entry for each NPN class. Then the compiled integer array will be loaded as a forest-like shared DAG.

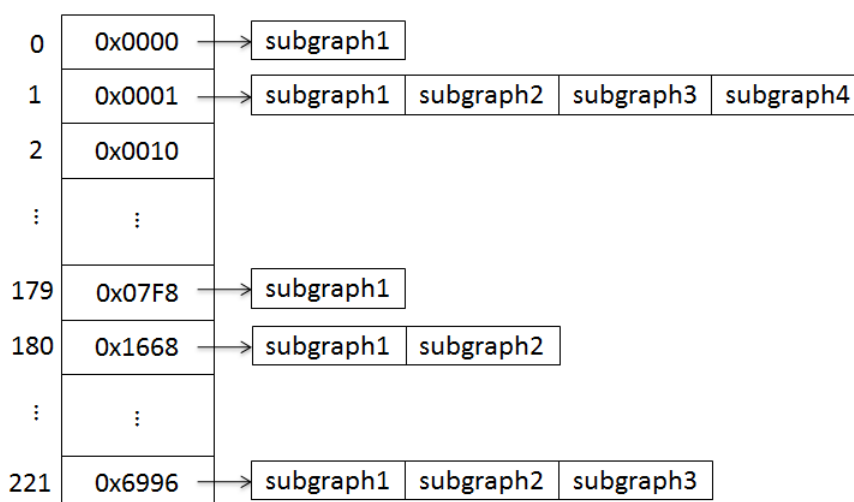
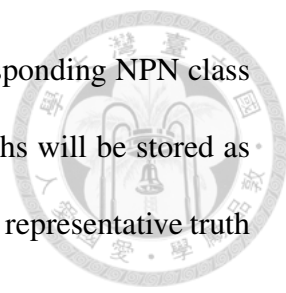


Figure 3.4: Subgraph Storage Table



If the node is marked as output, it will be pushed into the corresponding NPN class in the table. Once the loading completed, the pre-computed subgraphs will be stored as shown in fig 3.4. The hexadecimal numbers labelled in the table is the representative truth table for each class.

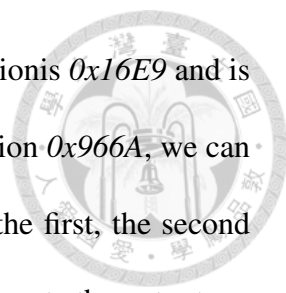
### 3.3.2 Subgraph NPN Manipulation

Given an arbitrary 4-input function  $f$ , we have to build the mapping relation from  $f$  to its NPN representative function and vice versa. To be more specific, we want to know that how to get  $f$  from its NPN representative function by negating inputs, permuting inputs, and negating the output.

	Representative Function/Class		Negation	Permutation
0x0000	0x0000	0	0 0000	{0 1 2 3}
0x0001	0x0001	1	0 0000	{0 1 2 3}
0x0002	0x0001	1	0 0001	{0 1 2 3}
⋮	⋮	⋮	⋮	⋮
0x9669	0x6996	221	1 0000	{0 1 2 3}
0x966A	0x16E9	205	1 0111	{3 1 2 0}
⋮	⋮	⋮	⋮	⋮
0xFFFF	0x0000	0	1 0000	{0 1 2 3}

Figure 3.5: NPN Relation Mapping

To achieve the NPN manipulation, four arrays are stored as illustrated in fig 3.5. The first and second array store the mapping from arbitrary 4-input function to its NPN representative function and class number. The third array stores negation information including one output and four inputs. The last array stores permutation information.



Take function  $0x966A$  for example, its NPN representative function is  $0x16E9$  and is the 205th class among 222 classes. To get the subgraph of the function  $0x966A$ , we can pick any subgraph in the 205th entry of the subgraph table, negate the first, the second and the third input, permute the input with the ordering  $3\ 1\ 2\ 0$ , and negate the output.

Those arrays are computed by brute force enumeration for 0 to 65535. With the mapping relation, we can replace the 4-input subgraphs with our pre-computed subgraphs in constant time while preserving the functionality of the node.

So far, all the preparations for MIG rewriting are ready, which typically can be processed within a second, and we can start to update our network now. As will be introduced in the chapter 4, various rewriting strategies can be selected by different commands according to our purpose and specification. Once the rewriting completed, the updated MIG network will be written out to a Verilog file, which can be easily re-used in other platforms.

In this chapter, we have depicted the overview of our engine and introduced the details for generating MIG subgraphs. The *exact synthesis* method with proper blocking strategies helps us automate the whole preparation phase, resulting a compact integer array containing all non-redundant subgraphs for each NPN class.

With the understanding of our engine, we can start to discuss how to apply our proposed methods in the domain of both logic synthesis and verification by different rewriting methodologies in the next chapter.



# Chapter 4 Application of DAG-Aware MIG

## Rewriting



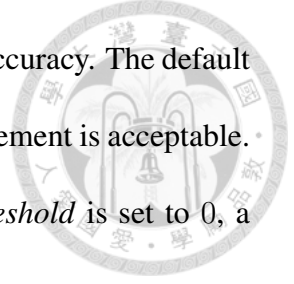
### 4.1 From the Perspective of Logic Synthesis

With the strong synthesis power proven in the previous work [3,4], we expect a better result if the dag-aware rewriting step can be integrated into the MIG synthesis flow. In this section, we introduce the rewriting methodology for logic optimization followed by a demonstration example.

#### 4.1.1 Rewriting Methodology for Logic Optimization

We demonstrate the MIG rewriting procedure by means of Algorithm 3. In the beginning of the rewriting phase, our engine loads the pre-computed subgraphs compiled previously as an integer array. Then the nodes are visited in a topological order. For each 4-input cut of a node, all pre-computed subgraphs of its NPN class are considered (line 8). After trying all available subgraphs for the given node, the one that leads to the largest improvement at a node is used. Finally, if such an advantageous subgraph exists, we update our network by this subgraph.

Note that this function can be used to optimize different kinds of targets such as size, depth by modifying the function *SubgraphsEvaluate* in line 9. For example, if the target of optimization is area (size), function *SubgraphsEvaluate* can be modified to count the number of nodes saved for each possible subgraphs and returns the one with greatest improvement. Also, there is an user-defined integer *threshold* passing to the rewriting



function as a parameter so that subgraphs can be selected with more accuracy. The default value for *threshold* is 1, which means only the subgraphs with improvement is acceptable. Subgraphs that does not improve the target are also accepted if *threshold* is set to 0, a relaxed criterion for rewriting.

---

**Algorithm 4** MIG Rewriting

---

```
1: function MIG_REWRITING(Network ntk, int threshold)
2:   allgraphs  $\leftarrow$  LoadPrecomputedArray()
3:   for each node in the ntk in a topological order do
4:     bestGain  $\leftarrow$  -1
5:     bestGraph  $\leftarrow$  null
6:     for each 4-input cut C of node computed using cut enumeration do
7:       truth  $\leftarrow$  CutTruth(node, C)
8:       subgraphs  $\leftarrow$  HashTableLookup(allgraphs, truth)
9:       gain  $\leftarrow$  SubgraphsEvaluate(ntk, node, subgraphs)
10:      if gain  $\geq$  threshold and gain  $>$  bestGain then
11:        bestGain  $\leftarrow$  gain
12:        bestGraph  $\leftarrow$  G
13:      if bestGraph  $\neq$  null then
14:        Mig_UpdateNework(ntk, node, bestgraph)
```

---

### 4.1.2 A Demonstration Example

To conclude this section, we comment on the MIG rewriting procedure with a simple example reported in Fig 4.1 that optimizes the size. Note that this simple network is adopted from the [4].

In the beginning, all cuts are enumerated in a bottom-up manner and are listed near by the corresponding nodes in the figure. Any cut that is of size greater than 4 is discarded. Then the nodes are visited in a topological order. Boolean function of each 4-input cut is computed to find its corresponding NPN class and hence its corresponding pre-computed subgraphs. For the sake of clarity, only the cuts with improvement are marked in red

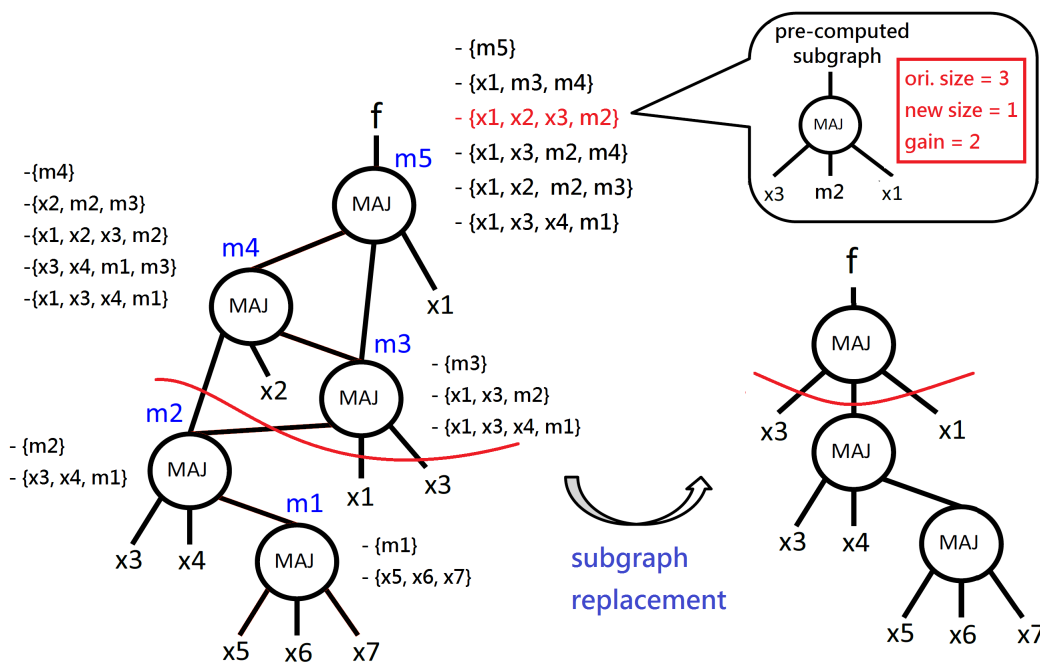
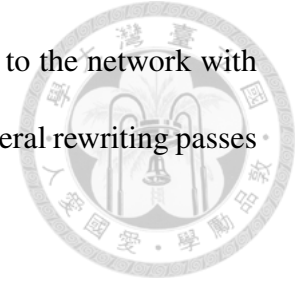


Figure 4.1: MIG Rewriting Example

and discussed. We can see that the pre-computed subgraph for cut  $\{x1, x2, x3, m2\}$  of node  $m5$  can be represented with a single majority node, thanks to the minimality of *exact synthesis* method, leading to a great improvement. The MIG network then is updated with this subgraph to end the rewriting procedure. It is worth noticing that if we perform another rewriting pass with target on depth, the 4-input cut  $\{x1, x3, x4, m1\}$  of output node will be rewritten to a pre-computed subgraph with shorter delay. In fact, after depth-oriented rewriting, the resulting MIG structure is the same as the final MIG generated in [4].

Compared to the method proposed in [4], our rewriting procedure is much simpler in manipulating MIG network and much more powerful in detecting reduction opportunity. To be more specific, instead of searching reduction candidates by some case-dependent metrics, our proposed method can almost guarantee the improvement by replacing advantageous subgraphs in a greedy manner. While still being heuristic and suboptimal, our method does not require much hand-tuning and trial-and-error iterations. The 4-input

rewriting is local, however, rewriting is very fast and can be applied to the network with different targets many times. The result is the cumulative effect of several rewriting passes and the scope of changes is no longer local.



## 4.2 From the Perspective of Datapath Verification

In chapter 2, we have introduced typical verification flow and discussed how to integrate datapath analysis method into the flow to resolve the datapath verification problem. However, algorithms for datapath analysis may fail to find appropriate datapath configuration for some reasons. For example, the declared bit width in RTL design may be different from the revised netlist and hence word structures become unrecongized. Also, logic sharing presented in the revised netlist makes it hard to accurately separate different arithmetic modules. In a word, since the revised netlist underwent a bit-level optimization process, algorithms for detecting word-level structures in a sea of bit-level gates are still hard to be complete.

In this section, we propose a method, named datapath normalization, to cope with bit-optimized netlist based on MIG rewriting. With the method integrated, we expect that datapath can be analyzed in a much smoother way.

### 4.2.1 Motivation

We will discuss what motivates us to adopt MIG rewriting as a strategy for datapath analysis in this section.

In the beginning, we know that arithmetic operations such as addition, multiplication are composed of the basic component, the 1-bit full adder. Those components are con-

nected in various configurations to accomplish the specific operations. It is the various configurations that make datapaths hard to be identified. Note that the backbone of the datapath is the cascaded carryout chains of each full adder. Also, the carryout function of the full adder is exact the 3-input majority function, which is the basic node in MIGs. Therefore, if we can recover the bit-optimized carryout chains by MIG rewriting, we will have a netlist easier to configure.

Besides, the MIG for a full adder, i.e.,  $s = a \oplus b \oplus c_{in}$  and  $c_{out} = ab + ac + bc$  can be represented with only 3 nodes, as depicted in figure 4.2. Notice that the  $c_{out}$  node is shared with the function  $s$ , which makes the structure more compact. In fact, this sharing property could be a strong signal to detect adder-like subgraphs in MIG.

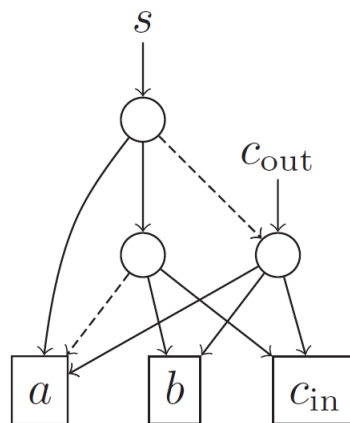


Figure 4.2: MIG representation for a full adder

With the motivations discussed above, we expect that the MIG rewriting-based algorithm can definitely improve the quality of datapath analysis. The proposed algorithm is introduced in the next section.



## 4.2.2 Datapath Normalization

We demonstrate the datapath normalization procedure by means of Algorithm 5. In the beginning, we try to detect full adders in the original AIG. The function *DetectFullAdders* (line 2) computes 3-input *XOR* and 3-input *MAJ* cuts for each node. Then, full adders can be collected by finding common inputs between *XOR* cuts and *MAJ* cuts. The full adders found by this cut enumeration method are in a normalized form, that is, they are not bit-optimized in the network and can be easily identified by datapath analysis algorithm. Then, instead of direct transformation from AIG to MIG, those nodes marked as full adders are constructed into the MIG structures as the representation in Figure 4.2.

---

**Algorithm 5** Datapath Normalization with MIG Rewriting

---

```
1: function DATAPATHNORMALIZATION(Network aig)
2:   vFadds  $\leftarrow$  DetectFullAdders(aig)
3:   mig  $\leftarrow$  AigToMigWithAdderConstruct(aig, vFadds)
4:   vChains  $\leftarrow$  CollectCarryoutChains(mig)
5:   allgraphs  $\leftarrow$  LoadPrecomputedArray()
6:   simulate(mig)
7:   for each node in the vChains do
8:     if isFullAdder(node) then continue
9:     // try to normalize the node
10:    simValue  $\leftarrow$  XorFaninSimValue(node)
11:    vSums  $\leftarrow$  CollectSumCandidate(mig.simTable, simValue)
12:    for each candidate in vSums do
13:      bestGain  $\leftarrow$  -1
14:      bestGraph  $\leftarrow$  null
15:      for each 4-input cut C of candidate do
16:        truth  $\leftarrow$  CutTruth(candidate, C)
17:        subgraphs  $\leftarrow$  HashTableLookup(allgraphs, truth)
18:        gain  $\leftarrow$  DetectFaddStructure(mig, candidate, subgraphs)
19:        if gain  $\geq$  0 then
20:          bestGain  $\leftarrow$  gain
21:          bestGraph  $\leftarrow$  G
22:        if bestGraph  $\neq$  null then
23:          Mig_UpdateNetwork(mig, candidate, bestgraph)
```

---

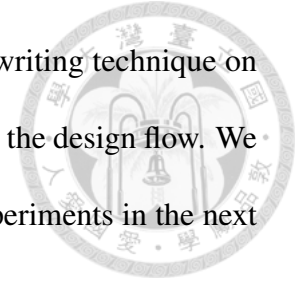
We then collect carryout chains in the MIG structure and sort them into the topological order (line 4). Note that since the network is bit-optimized, there must be some

carryout nodes left in the network without finding corresponding sum nodes. If those nodes can be paired with their corresponding sum nodes, we can normalized the network into a more regular structure. The remaining procedure in the Algorithm 5 tries to accomplish the normalization task.

First, boolean simulation is performed over the MIG and nodes are hashed into a table (line 6). Then, for each unpaired node in the carryout chains, we compute a simulation value by *XORing* its fanins simulation value (line 10). All nodes with this computed simulation value are collected as candidates for the carryout (line 11). Similar to Algorithm 4, we try to perform rewriting on these candidates. The only difference is the criterion for accepted subgraphs. The function *DetectFaddStructure* in line 18 tries to detect subgraphs with node sharing as the structure in Figure 4.2. Also, simulation values in each subgraphs are temporarily computed as a detecting signature so that if the sharing properties can not be found by MIG lookup function, we can rely on those simulation values to decide whether to accept a subgraph or not. Finally, if a subgraph is beneficial to normalize the circuit, we update the MIG with the subgraph.

False positive rewriting may be performed in the procedure since the simulation technique is applied, however, it is not often the case in our experiments since the intrinsic property of arithmetic circuit that it can be easily distinguished by simulation. Compared to AIG, MIG is in a more compact structure, especially for arithmetic circuit, so that it can be manipulated in a more guidable manner. To be more specific, each full adder can be represented with three majority nodes as mentioned before, therefore, we can mark each node as sum, carryout or others. With those marks, we can rewrite the MIG with a more accurate way.

In this chapter, we introduce how to apply DAG-aware MIG rewriting technique on synthesis and datapath verification, which both are essential stages in the design flow. We will continue to verify the power of rewriting technique by some experiments in the next chapter.





# Chapter 5 Experimental Results



In this chapter, we conduct several experiments and show our MIG rewriting technique is efficient and effective in both logic synthesis and datapath verification. The MIG subgraph generation package is implemented in C++ language with the SMT solver Z3 [20]. The MIG rewriting package is implemented in C/C++ language in academic synthesis and verification tool ABC. Our test environment is a Linux 64-bit machine with Intel i7 3.00 GHz CPU and 32 GB memory.

## 5.1 Logic Synthesis Results

### 5.1.1 Methodology

Figure 5.1 depicts the typical synthesis flow in left hand side and details for MIG-based technology independent optimization in the right hand side. Our MIG rewriting technique can be integrated into the last stage of MIG-based synthesis flow so that the circuit can be further compressed.

We adopt the Open Cores IWLS'05 and Arithmetic HDL as benchmarks (differential equation solvers, telecommunication units, sorters, specialized arithmetic units, etc.) for logic optimization. Since the goal is to compare how our algorithm can further improve the synthesis quality, our engine reads the Verilog files released by EPFL, which have been optimized by MIG Algebraic/Boolean methods, and writes back a Verilog description of DAG-aware compressed MIG.

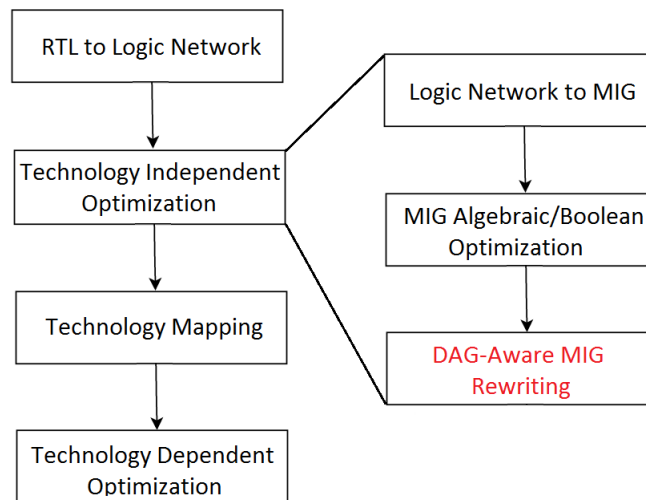


Figure 5.1: Synthesis Methodology

### 5.1.2 Optimization Results

Table 5.1 shows the optimization results. The second column (marked with MLP) is the original statistic from the released files. The third column (marked with Our Strategy) is the statistic after our rewriting technique is applied. Note that in our experiments, best results can be achieved when the parameter *threshold* in algorithm 4 is passed with value 1. Both the *size* and *depth* in the table are in terms of MIG. The last column *ratio* is the size ratio of MLP and our strategy.

We can see a total improvements in the size while preserves the depth for all cases. Considering the IWLS'05 benchmarks that are large but not tall, we see an about 10% reduction on size. Focusing on the arithmetic HDL benchmarks, we see a better size quality. Our MIG rewriting methodology enables further 20.4% reduction compares to the original MIGs, which strongly shows the power of our algorithm. Besides, our rewriting technique is efficient since all cases can be completed within few seconds. All MIG output Verilog underwent formal verification experiments (ABC *cec* command) with success.

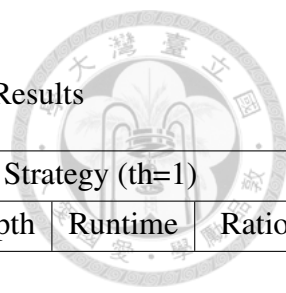



Table 5.1: Technology-Independent MIG Optimization Results

		MLP			Our Strategy (th=1)			
<b>Open Cores IWLS'05</b>		Size	Depth	Runtime	Size	Depth	Runtime	Ratio
Benchmark	I/O							
DSP	4365/4145	44166	35	12.54	39585	35	3.22	0.8963
ac97_ctrl	2267/2262	12996	9	9.76	12065	9	0.64	0.9284
aes_core	789/668	21616	19	10.68	18702	19	1.25	0.8652
des_area	368/72	4259	23	0.63	3890	23	0.26	0.9134
des_perf	9042/9038	76233	16	39.34	71174	16	14.08	0.9336
ethernet	10710/10728	68656	16	20.28	66618	16	15.27	0.9703
i2c	147/142	1114	9	0.21	1009	9	0.09	0.9057
mem_ctrl	1204/1231	8369	20	0.51	7687	20	0.43	0.9185
pci_bridge32	3527/3534	22132	17	3.88	19812	17	1.73	0.8952
pci_spoci_ctrl	89/80	1009	12	0.05	823	12	0.09	0.8157
sasc	133/132	754	7	0.22	688	7	0.1	0.9125
simple_spi	148/147	985	9	0.15	903	9	0.04	0.9168
spi	274/276	3614	20	1.79	3252	20	0.21	0.8998
ss_pcm	106/98	496	7	0.05	432	7	0.07	0.8710
systemcaes	930/819	10367	26	11.21	8605	26	0.55	0.8300
systemcdes	314/258	2712	20	3.62	2449	20	0.19	0.9030
tv80	379/410	7802	31	8.95	6607	31	0.42	0.8468
usb_funct	1894/1879	14842	20	12.62	13386	20	0.96	0.9019
usb_phy	113/111	484	8	0.04	435	8	0.07	0.8988

**AVG 0.8959**

<b>Arithmetic HDL</b>		Size	Depth	Runtime	Size	Depth	Runtime	Ratio
Benchmark	I/O							
MUL32	64/64	9161	37	3.39	7202	37	0.43	0.7862
sqrt32	32/16	2173	165	1.2	1710	165	0.17	0.7869
diffeq1	355/289	18015	220	123.55	14756	220	1.15	0.8191
div16	32/32	4407	103	6.39	3150	103	0.24	0.7148
hamming	200/7	2079	62	18.99	1618	62	0.16	0.7783
MAC32	96/65	9392	42	5.53	8467	42	0.52	0.9015
revx	20/25	7542	144	12.33	5921	144	0.48	0.7851

**AVG 0.7960**



To further verify the effect of our algorithm, we perform LUT mapping on the original MIGs and DAG-aware compressed ones. We adopt the tool *ABC* as synthesizer with mapping command *if -K 6*, which means the MIGs will be mapped into LUTs with 6 inputs. Table 5.2 shows the results of mapped circuits. Both the *size* and *depth* in the table are in terms of LUT. The last column *ratio* is the size ratio of MLP and our strategy.

Results obtained from technology mapping are of higher practical relevance. We can almost obtain better implementations in all cases. From the perspective of size, all cases are compressed except *des perf*, *sasc* and *systemcaes* due to inevitable logic duplications in the mapping phase. On the other hand, depth is preserved for all cases except *diffeq1*.

On average, we see an about 5% reduction on size for IWLS'05 benchmarks while about 12% reduction on Arithmetic HDL benchmarks. The results is better, however, the improvement is not as good as the pure logic optimization results since *ABC* is an AIG-based technology mapper. If a MIG-based mapper has been developed, we can definitely obtain better mapping results.

## 5.2 Datapath Verification Results

### 5.2.1 Methodology

To verify the effect of our algorithm, we adopt commercial verification tool *Cadence Conformal LEC* as our experiment flow. The command *datapath analysis* is built in *LEC*, which performs analysis between RT-level design and synthesized logic netlist before combinational equivalence checking and estimates an analysis quality by metrics such as circuit similarity. We will compare the improvement on the quality with and without

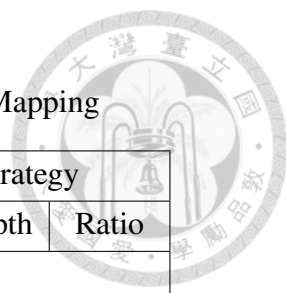


Table 5.2: MIG Optimization Results After Technology Mapping

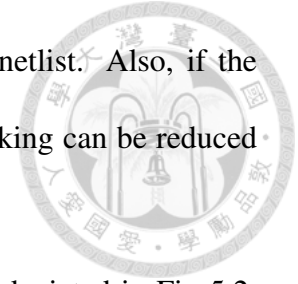
		MLP		Our Strategy		
<b>Open Cores IWLS'05</b>		Size	Depth	Size	Depth	Ratio
Benchmark	I/O					
DSP	4365/4145	11873	12	11148	12	0.9389
ac97_ctrl	2267/2262	2927	3	2911	3	0.9945
aes_core	789/668	5198	6	4703	7	0.9048
des_area	368/72	869	8	794	7	0.9137
des_perf	9042/9038	13062	3	13166	3	1.0080
ethernet	10710/10728	17659	7	17369	7	0.9836
i2c	147/142	275	4	268	4	0.9745
mem_ctrl	1204/1231	2430	7	2339	7	0.9626
pci_bridge32	3527/3534	5452	6	5343	6	0.9800
pci_spoci_ctrl	89/80	312	5	259	5	0.8301
sasc	133/132	153	2	154	2	1.0065
simple_spi	148/147	232	3	228	3	0.9828
spi	274/276	988	7	987	7	0.9990
ss_pcm	106/98	119	2	105	2	0.8824
systemcaes	930/819	1963	8	2086	7	1.0627
systemcdes	314/258	625	7	525	7	0.8400
tv80	379/410	2066	12	1971	12	0.9540
usb_funct	1894/1879	3784	7	3579	7	0.9458
usb_phy	113/111	138	2	138	2	1.0000

**AVG 0.9560**

<b>Arithmetic HDL</b>		Size	Depth	Size	Depth	Ratio
Benchmark	I/O					
MUL32	64/64	1926	11	1777	12	0.9226
sqrt32	32/16	661	66	531	66	0.8033
diffeq1	355/289	4653	49	4146	58	0.8910
div16	32/32	1640	43	1261	43	0.7689
hamming	200/7	545	15	530	16	0.9725
MAC32	96/65	2186	12	2062	12	0.9433
revx	20/25	2258	45	1940	46	0.8592

**AVG 0.8801**

applying our DAG-aware MIG rewriting on the synthesized logic netlist. Also, if the quality is improved, we expect that the runtime of equivalence checking can be reduced effectively.



We generate arithmetic designs as our benchmark by the flow depicted in Fig 5.2. The original RT-level designs are synthesized by commercial synthesis tool *Synopsys Design Compiler*, which optimizes both area and depth. To increase the degree of bit-level optimization, those synthesized netlists are passed to *ABC* with command *dc2*, which performs further bit-level rewriting and refactoring.

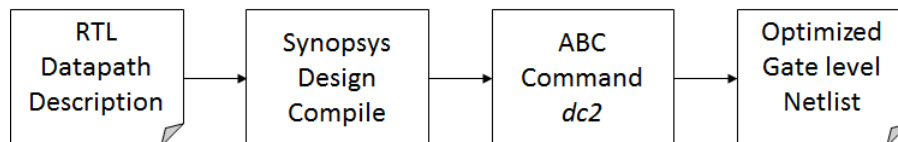
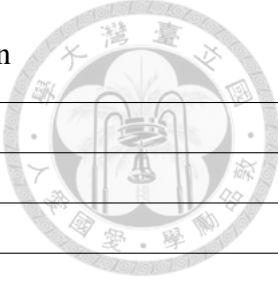


Figure 5.2: Arithmetic Design Generation Flow

The generated RT-level designs include various arithmetic operations such as complex addition trees, plentiful shift operations and multiplications. Also, our benchmark can be divided into two sets according to the complexity of arithmetic operations.

The first set of the benchmark is comparatively simple: only few arithmetic operations and constant multiplications are presented in the designs. The second set is much harder: the variation in arithmetic operations is high and multiplications are embedded in the designs. The detailed RTL descriptions are listed in Table 5.3. Note that those optimized netlists are all poorly analyzed by *LEC*.

Table 5.3: Arithmetic Benchmark Detail Description



Easier	Bitwidth	RTL Description
case1	14	$a-b+c+[(d-e)*10]$
case2	24	$a-b+c+[(d-e)*10]$
case3	30	$a-b+c+[(d-e)*10]$
case4	33	$a-b+c+[(d-e)*10]$
case5	18	$a-b+[(c-d)<<1]+[(c-d)]+e+[(c-d)<<3]$
case6	23	$a-b+[(c-d)<<1]+[(c-d)]+e+[(c-d)<<3]$
case7	26	$a-b+[(c-d)<<1]+[(c-d)]+e+[(c-d)<<3]$
case8	8	$a-b+c+[d<<1]+d+[d<<3]-[e*11]$
case9	10	$a-b+c+[d<<1]+d+[d<<3]-[e*11]$
case10	12	$a-b+c+[d<<1]+d+[d<<3]-[e*11]$
Harder	Bitwidth	RTL Description
case11	14	$a-b+c+[(d-e)<<3]+[(d-e)<<1]+(d-e)+g+[(h-i)<<2]+[(h-i)<<1]$
case12	16	$a-b+[(d*e)<<1]+(d*e)+c+[(d*e)<<3]$
case13	16	$a-b+c+[(d-e)<<3]+[(d-e)<<1]+(d-e)+g+[(h-i)<<2]+[(h-i)<<1]$
case14	22	$a-b+c+[(d-e)<<3]+[(d-e)<<1]+(d-e)+g+[(h-i)<<2]+[(h-i)<<1]+h$
case15	10	$a-b+c+[(d-e)<<2]+[(d-e)<<1]+(d-e)+f+[(h-i)<<3]+[(h-i)<<1]$
case16	10	$a-b+c+[(d*e)<<2]+[(d*e)<<1]+(d*e)-f+[(h-i)<<3]+[(h-i)<<1]$
case17	14	$a-b+(d*e)+[c<<1]+[c<<3]+c+[(d-e)<<1]+[(d-e)<<3]$
case18	20	$a-b+c+[(d-e)<<3]+[(d-e)<<1]+(d-e)+f+[(h-i)<<3]+[(h-i)<<1]$
case19	12	$a-b+c+[(d-e)<<3]+[(d-e)<<1]+(d-e)+f+[(h-i)<<2]+[(h-i)<<1]$
case20	10	$a-b+c+[(d*e)<<2]+(d*e)-h+[(h*i)<<3]+[(h*i)<<1]$

## 5.2.2 Verification Results

Table 5.4 summarizes experimental results for datapath verification of the first benchmark set with MIG rewriting. The column for *size* is the number of *AND* nodes (since they are generated by *ABC* in the last step) in the optimized netlist. *Quality* is the analysis quality estimated by *LEC* and *Runtime* is the runtime of equivalence checking (in seconds) after datapath analysis. There is no non-equivalent output in our cases.

We can see that the average quality after MIG rewriting is more than three times as

Table 5.4: Datapath Verification Results of Easier Benchmark With MIG Rewriting

				Original ( g.v to r.v )			MIG rewrite ( g.v to mig.v )		
	Input	Output	Size	Quality	Runtime	eq/abort	Quality	Runtime	eq/abort
case1	77	21	1441	16%	14.77	21/0	51%	0.88	21/0
case2	127	31	2281	11%	31.05	31/0	49%	1.73	31/0
case3	157	37	2785	9%	45.14	37/0	50%	2.47	37/0
case4	172	40	2911	12%	55	40/0	44%	31.4	40/0
case5	57	17	1121	40%	7.94	17/0	55%	0.4	17/0
case6	97	25	1793	36%	17.9	25/0	54%	1.03	25/0
case7	137	33	2449	10%	36.1	33/0	50%	2.02	33/0
case8	47	15	953	42%	7.24	15/0	56%	1.04	15/0
case9	67	19	1289	39%	10.31	19/0	55%	0.54	19/0
case10	122	30	2197	11%	29.72	30/0	45%	20.41	30/0
						<b>268/0</b>	<b>3.16</b>	<b>0.18</b>	<b>268/0</b>

the one without MIG rewriting, which means our algorithm can effectively assist datapath analysis. Besides, as our expectation, the runtime for equivalence checking is reduced with about 82% since the quality is improved.

Table 5.5 summarizes experimental results for datapath verification of harder benchmark set with MIG rewriting. Note that since those cases are much more complex, some outputs may be aborted by *LEC* because *LEC* finds it hard to complete the equivalence checking in reasonable runtime.

In the table, we see that the average quality result is not we want since it is reduced about 6%. Fortunately, in most cases, aborted outputs can be completely verified after MIG rewriting. The reason for such improvement is twofold. On the one hand, when datapath normalization algorithm is applied, some crucial arithmetic points may be exposed in the netlist so that *LEC* can verify the miter by internal equivalence one after another instead of verifying outputs with brute force. On the other hand, we see the verification



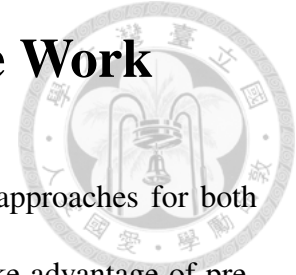
Table 5.5: Datapath Verification Results of Harder Benchmark With MIG Rewriting

				Original ( g.v to r.v )			MIG rewrite ( g.v to mig.v )		
	Input	Output	Size	Quality	Runtime	eq/abort	Quality	Runtime	eq/abort
case11	126	23	2407	39%	29.87	10/13	30%	44.39	23/0
case12	81	21	2833	29%	37.34	14/7	27%	46.81	14/7
case13	140	25	2846	34%	20	7/18	37%	48.43	25/0
case14	188	31	4474	33%	31.47	8/23	42%	64.06	9/22
case15	92	19	2007	32%	20.49	11/8	28%	38.74	19/0
case16	72	19	2387	41%	51.51	10/9	38%	80.18	11/8
case17	61	19	2149	35%	22.61	11/8	37%	31.02	11/8
case18	172	29	3437	34%	23.19	7/22	30%	89.52	29/0
case19	108	21	2145	39%	29.51	10/11	31%	39.42	21/0
case20	80	19	2630	45%	35.96	5/14	38%	55.84	6/13
						<b>93/133</b>	<b>0.94</b>	<b>1.88</b>	<b>168/58</b>

runtime is almost twice as the original one, which means the netlists with MIG rewriting are easier to conquer for *LEC* so that it will not give up solving and abort them in the early stage.

In this chapter, experiment results validate the effect of DAG-aware MIG rewriting technique and prove the potential of MIGs in logic synthesis and verification again. We will conclude this thesis in the next chapter.

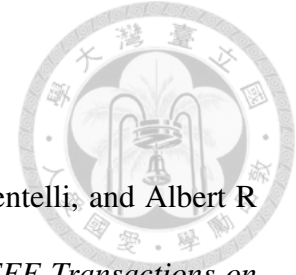
## Chapter 6 Conclusions and Future Work




In this thesis, we've proposed efficient MIG-based rewriting approaches for both logic synthesis and verification. The proposed rewriting method take advantage of pre-computed minimum MIG subgraphs. In the logic synthesis domain, by replacing advantageous subgraphs successively, we can compress the large netlist with remarkable reduction within seconds. On the other hand, we take the first step to apply MIG in the verification domain and our MIG rewriting algorithm targets on recovering the datapaths. Both rewriting strategies are simple, yet powerful. Experimental results have shown that our approach performs better than the previous works on MIG and can definitely improve the quality on datapath analysis.

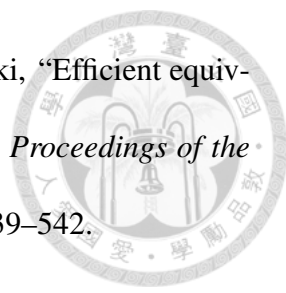
For the future work, there are some points in our flow which can be further improved. First, in the synthesis domain, we can develop a MIG-based technology mapping engine so that the entire logic synthesis process can be manipulated with MIG and better synthesis results are almost guaranteed. For the datapath verification, MIG subgraphs favored by arithmetic circuit can be pre-computed instead of using minimum subgraphs. Also, rewriting strategies can be more sophisticated so that word-level information from the RTL design won't be discarded directly. Finally, since we've successfully pioneered in applying MIG to the verification domain, there must be lots of promising ideas worth trying hereafter.

# Reference



- [1] Robert K Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang, “Mis: A multiple-level logic optimization system,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.
- [2] Robert Brayton and Alan Mishchenko, “Abc: An academic industrial-strength verification tool,” in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.
- [3] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli, “Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization,” in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.
- [4] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli, “Boolean logic optimization in majority-inverter graphs,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.
- [5] Per Bjesse and Arne Borally, “Dag-aware circuit compression for formal verification,” in *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. IEEE Computer Society, 2004, pp. 42–49.
- [6] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton, “Dag-aware aig rewriting a fresh look at combinational logic synthesis,” in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006, pp. 532–535.

- 
- [7] Vinícius P Correia and André I Reis, “Classifying n-input boolean functions,” in *VII Workshop Iberchip*, 2001, p. 58.
- [8] Daniel Brand, “Verification of large synthesized designs,” in *The Best of ICCAD*, pp. 65–72. Springer, 2003.
- [9] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai, “Robust boolean reasoning for equivalence checking and functional property verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.
- [10] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een, “Improvements to combinational equivalence checking,” in *2006 IEEE/ACM International Conference on Computer Aided Design*. IEEE, 2006, pp. 836–843.
- [11] Randal E Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on computers*, vol. 100, no. 8, pp. 677–691, 1986.
- [12] Niklas Sorensson and Niklas Een, “Minisat v1. 13-a sat solver with conflict-clause minimization,” *SAT*, vol. 2005, pp. 53, 2005.
- [13] Jiang Long, Robert K Brayton, and Michael Case, “Lec: Learning driven data-path equivalence checking,” *Prof. of DIFTS@ FMCAD*, pp. 9–18, 2013.
- [14] Grigori S Tseitin, “On the complexity of derivation in propositional calculus,” in *Automation of reasoning*, pp. 466–483. Springer, 1983.

- 
- [15] Demosthenes Anastasakis, Lisa McIlwain, and Slawomir Pilarski, “Efficient equivalence checking with partitions and hierarchical cut-points,” in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 539–542.
- [16] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton, “Fraigs: A unifying representation for logic synthesis and verification,” Tech. Rep., ERL Technical Report, 2005.
- [17] Maciej Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi, “Verification of gate-level arithmetic circuits by function extraction,” in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 52.
- [18] Valeria Bertacco and Maurizio Damiani, “The disjunctive decomposition of logic functions,” in *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 1997, pp. 78–82.
- [19] Mathias Soeken, Luca Gaetano Amar, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli, “Optimizing majority-inverter graphs with functional hashing,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1030–1035.
- [20] Leonardo De Moura and Nikolaj Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.