

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



使用巨集塊映射方法於數位電路之逆向工程

Digital Circuit Reverse Engineering
Using Macro block mapping Methods

張景翔

Ching-Hsiang Chang

指導教授：郭斯彥 博士

Advisor: Sy-Yen Kuo, Ph.D.

中華民國 109 年 7 月

July 2020

誌謝



能完成這篇論文，我要特別感謝我的指導教授郭斯彥老師，他提供我相當完善的研究環境。也感謝袁世一教授，他給予我許多研究態度以及報告口條的指導。在我的研究過程中，幫助我最多的莫過於亞睿資訊股份有限公司同事給我的支持及資源。藉由他們的幫助，我大大縮短了實驗實作的工作時數。最後要感謝實驗室的每一位成員對我的支持與鼓勵，謝謝大家。

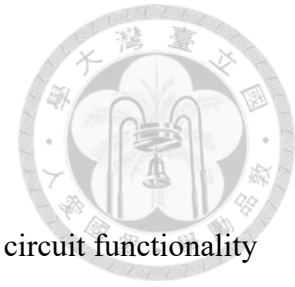
摘要



數位電路的反向工程一直以來都是用於重建電路功能性相當有力的工具。而重建電路功能性可以有以下幾種應用：其一是可以幫助我們找出惡意電路（亦稱硬體木馬），其二是針對某些規格書已經佚失的舊有設計，我們可以利用反向工程的工具以便釐清其功能。據我們所知，反向工程大概是這些問題唯一的解決方案。在本研究中我們提出一個可以讓使用者從平坦化的閘級網路連線表擷取出功能模組的硬體反向工程演算法，而且不需要人工介入。提出方法使用了切割枚舉方法以及布林匹配技術以辨識我們感興趣的功能塊。更明確的說，我們推廣了現有的切割枚舉方法，讓它變成一個子電路枚舉方法，然後確認該子電路是否正好是預先定義好的巨集庫的一員。實驗結果顯示我們的方法無法擴展至含有數千個邏輯單元的電路，肇因於過大的計算複雜度。

關鍵字：數位電路、反向工程、自動化設計、子電路枚舉、形式驗證、硬體安全

Abstract



Digital circuit reverse engineering has been a powerful tool for circuit functionality reconstruction, which can have several applications. On the one hand, understanding the circuit's functionality helps us to find out malicious circuitry (a.k.a. hardware Trojan) inside the device under test (DUT). On the other hand, for some legacy designs whose specification is lost, we can use reverse engineering tool to clarify its functionality. To the best of our knowledge, reverse engineering (RE) is arguably the only solution to these problems. In this work we propose a hardware reverse engineering algorithm which enables a user to extract functional modules from a flattened gate-level netlist with no manual intervention. The proposed method utilizes a cut enumeration method together with Boolean matching technique to recognize functional blocks in which we are interested. More specifically, we extend the existing cut enumeration method to a subcircuit enumeration method, and then check whether the subcircuit happen to be a functional macro block of the predefined macro library. The experimental result shows that our method cannot scale up to circuits containing thousands of logic cells because the computational complexity is just quite high.

Keywords: digital circuits, reverse engineering, design automation, subcircuit enumeration, formal verification, hardware security



Contents

List of Figures	vi
List of Tables	vii
Chapter 1 Introduction	1
Chapter 2 Related works	3
Chapter 3 Preliminaries	5
3.1 Cofactor	5
3.2 Boolean network	5
3.3 And-invert graph (AIG)	6
3.4 Tool ABC	6
3.5 Boolean matching	6
3.6 Cut enumeration	8
3.6.1 Cut	8
3.6.2 Cut set	8
3.6.3 Enumerative procedure	9
3.7 Subcircuit validation	9
3.7.1 Swallowed output	10
3.7.2 Unexpected output	10

3.7.3	Valid subcircuit	11
Chapter 4	AIG hash function	12
4.1	Simple AIG hash function	12
4.2	Complex AIG hash function	14
Chapter 5	Proposed method	16
5.1	Macro library	16
5.2	Macro mapping algorithm	17
5.3	Improvement	20
Chapter 6	Experimental results	23
Chapter 7	Conclusion and future works	25
Bibliography		26





List of Figures

3.1	Gate-level schematic for illustrating concept of cut set	8
3.2	A 2-input XOR gate implementation.	10
3.3	A 2-input XOR gate implementation with extra output(red line). . . .	11
5.1	Transforming a netlist to a DAG	18
5.2	Illustration for collecting candidate PO nodes	21



List of Tables

6.1	Experiment result of the proposed reverse engineering algorithm. . . .	23
-----	--	----



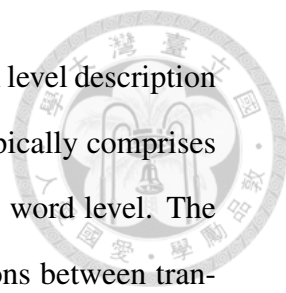
Chapter 1

Introduction

As technology advances, the complexity of modern integrated circuit keeps on growing continuously and drastically. In order to cut down on the design effort, IC designers tend to use macro blocks to build their designs. This design methodology provides a higher level abstraction of integrated circuits and macro blocks can be reused in separate projects or be replaced with another faster, smaller or lower-power macro if newer IC process has been released. In this case, the integrity of macro blocks must be unquestioned, otherwise the integrated circuit is exposed to the risk of malicious attack, a.k.a. **hardware trojan** [1, 2]. Hardware Trojan horse is a component hidden inside a designed hardware which appears to perform a certain action but in fact performs a hidden task.

Another scenario would target at circuit understanding. For some legacy designs whose specification is lost, since their functionality is unclear, engineers cannot hastily reuse them or make modifications to it. What's even worse, if the design were a building component of another project, we will have difficulty analyzing the high-level description of that project. Hence, if an EDA tool can parse a flattened gate level netlist and thus helps user clarify its functionality, that would be very useful.

For the two scenarios stated above, engineers can benefit from the result of reverse



engineering in that the kernel concept of which is to reconstruct a high level description of the given netlist. In context of hardware reverse engineering, it typically comprises two phases: from transistor level to gate level and from gate level to word level. The former is achieved by image processing and recognize the connections between transistors. The latter start from synthesized gate level circuit, some formal, sweeping or ATPG methods are applied to the netlist to clarify its functionality. Reverse engineering of digital hardware has been a tough task in EDA industry [3] because the RT-level module boundary may be destroyed due to boolean optimization or technology mapping during logic synthesis process.

Given this situation, this thesis presents a macro block recognition algorithm consists of a subcircuit enumerative procedure and Boolean matching solver. Roughly speaking, the proposed method enumerates single output subcircuits of the netlist and forms a complex macro by combining them. Employing this method a user is capable of reducing the complexity of his or her design and thus it should become easier to understand.

The rest of this thesis is organized as follows. Related works are introduced in Chapter 2. Chapter 3 presents background knowledge including terminologies and framework we used in this work. Chapter 4 elaborates on two AIG hash functions which can effectively avoid unpromising problem solvings. Chapter 5 describes what does the macro library contain and how are the macros stored. Afterwards, the core algorithm of this work is revealed. We show the performance of our approach using the experiment results in Chapter 6. Lastly chapter 7 concludes this thesis and depict our future research direction.

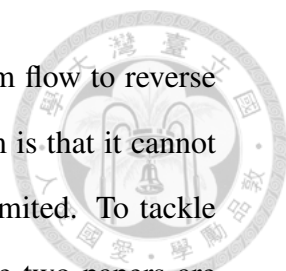


Chapter 2

Related works

There has been a comprehensive reverse engineering research developed in the last century [4]. Hansen *et al.* investigated the well-known ISCAS'85 benchmark circuits. They revealed functional blocks reside in these circuits. It was found that ISCAS'85 benchmarks share common basic combinational blocks and some of them has replicated structures inside. In addition, they shared their experience and gave several useful reverse engineering techniques as well. Nonetheless, the strategies they presented are mostly manual. In particular, the authors stated that reverse engineering is inevitably a trial-and-error process. Even so, to reverse engineer designs contain millions of gates, a fully automated approach is still needed.

P. Subramanyan *et al.* presents an automated and scalable method in [5, 6]. Their method consists of two parts, identifying combinational modules and identifying sequential modules. In the former case, first extract word-level information from the netlist [7] and then examine the combinational part delimited by the words. This is done by solving quantified Boolean formulas (QBFs). In the latter case, it was further divided into several fields, namely counter, shift register, RAM arrays and multibit registers. For each field the authors developed a customized strategy to recognize the functional block, mostly are based on structural analyses.



Although the authors of [6] presented a fairly thorough algorithm flow to reverse engineer netlist schematic, it still has a few limitations. One of them is that it cannot identify control logic. That is one of the reasons the coverage is limited. To tackle this problem, a few research papers has been published [8, 9]. The two papers are complementary in that one focuses on reconstructing a high-level description of the finite state machine while the other focuses on recognizing logic cells that belong to the control logic.

Some other researchers use a macro block mapping scheme to achieve reverse engineering [10]. It focused on algorithmic circuits so it's less powerful. By building the XOR trees, the researchers claimed that their method is able to identify any combinations of arbitrary adders and multipliers. The notion is based on the fact that multiplication is essentially additions of operands shifted left by different number of bits. XOR gates implements summation of half adders and one can analyze the data flow of carry out signals to determine the bit significance.



Chapter 3

Preliminaries

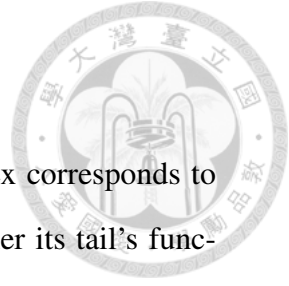
In this chapter, we would like to review some necessary background knowledge. The concepts introduced in this chapter are essential building blocks of this work.

3.1 Cofactor

The cofactor of a Boolean function $f(x_1, \dots, x_n)$ is defined by assigning constant value to a input variable, i.e., $f(x_1, \dots, x_i = 0, \dots, x_n)$ and $f(x_1, \dots, x_i = 1, \dots, x_n)$. The former one is called **negative** cofactor and the latter **positive** cofactor with respect to x_i . Cofactoring is primarily used to examine a Boolean function under the condition that some Boolean variables are fixed values.

3.2 Boolean network

A Boolean network is a directed acyclic graph (DAG) $G(V, A)$ where vertices correspond to logic functions and arcs specify the interconnection relation between functions. For an arc $(f_u, f_v) \in A$, it indicates that f_v takes f_u as its input, i.e. $f_v(\dots, f_u, \dots)$. The vertices which have no incoming arc are called *primary inputs (PIs)*. The vertices which have no outgoing arc are called *primary outputs (POs)*.



3.3 And-invert graph (AIG)

An AIG is a specific type of Boolean network where each vertex corresponds to logical conjunction and each arc can carry an information of whether its tail's function output is negated or not. In this thesis, we use the words Boolean function and AIG interchangeably. Compared to BDDs, AIGs are way more memory efficient as BDDs consume exponential memory space. Therefore, AIG enables an engineer to tackle the increasingly complex designs. Nonetheless, AIGs are not canonical. That is, a Boolean function may have multiple different AIG representations and it requires an algorithm to tell the equivalency. To mitigate this drawback, a concept called *functionally reduced* AIG (FRAIG) which makes it semi-canonical has been proposed [11].

3.4 Tool ABC

ABC is a system for synthesis and formal verification of logic circuits developed by UC Berkeley [12]. It provides a highly optimized framework for academic researchers and companies to develop their own work. ABC adopts AIG as the primary network representation and provides comprehensive helper APIs. Most of the synthesis and technology mapping commands utilize AIG to accomplish the task, as well as Boolean matching command we adopt.

3.5 Boolean matching

Given two functions $f(A)$ and $g(B)$, Boolean matching tries to determine whether they are functionally equivalent under permutation(P) or negation(N) of inputs and outputs. Boolean matching problem can be broken down into two steps: (1) determining permutation and polarity of IO ports and (2) checking equivalence. In this sense, equivalence checking is a particular case of Boolean matching and we can control the

problem complexity by relaxing or tightening constraint at step 1. Sorted by restriction level, the equivalence between $f(A)$ and $g(B)$ can have several forms:



- **NPNP equivalent**

negation and permutation of inputs and outputs

- **NPN equivalent**

negation and permutation of inputs and negation of outputs

- **NP equivalent**

negation and permutation of inputs

- **PP equivalent**

permutation of inputs and outputs

- **P equivalent**

permutation of inputs

For example, suppose $f = a_1a_3 + a_2a_4$ and $g = b_2b_4 + b_1b_3$, a reliable Boolean matcher should give a result like $(a_1 \rightarrow b_2), (a_2 \rightarrow b_1), (a_3 \rightarrow b_4), (a_4 \rightarrow b_3)$ and f is P- equivalent to g . For another example, suppose $f = a_1a_3 + a_2$ and $g = b_3(b'_2 + b_1)$, a reliable Boolean matcher should give a result like $(f \rightarrow g'), (a_1 \rightarrow b'_1), (a_2 \rightarrow b'_3), (a_3 \rightarrow b_2)$ and f is NPN- equivalent to g .

Specifically, PP-equivalence can be used to recognize sub-circuit in the netlist. Given a collection of library cells in the netlist, a PP-equivalence Boolean matcher enables us to validate the existence of a macro block. Hadi Katebi *et al.* contributed two Boolean matching commands to ABC, namely `bm` and `bm2` [13, 14]. Both two commands check the PP-equivalence between two AIGs and we use them to integrate our research.



3.6 Cut enumeration

3.6.1 Cut

A cut C of a circuit node n is defined as a set of circuit nodes such that every path from a PI node to n contains at least one element in C . In other words, a consistent assignment of truth values to each node in C completely determines the value of n . In a network of size m , the number of cuts of size k is $O(m^k)$ [15, 16].

Definition 3.6.1. k -feasible

A cut is said to be k -feasible if $|C|$ doesn't exceed k .

3.6.2 Cut set

For a node in the netlist, it is very likely to have multiple cuts. All these cuts form a cut set. Take circuit in Figure 3.1 as an example, the cut set for each node would be:

$$\left\{ \begin{array}{l} A, B, C, D : \emptyset \\ n1 : \{ \{A, B\} \} \\ n2 : \{ \{C, D\} \} \\ n3 : \{ \{n2\}, \{C, D\} \} \\ Q : \{ \{n1, n3\}, \{n1, n2\}, \{n1, C, D\}, \{A, B, n3\}, \{A, B, n2\}, \{A, B, C, D\} \} \end{array} \right.$$

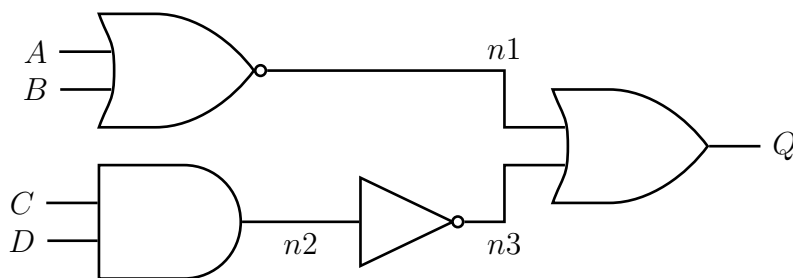
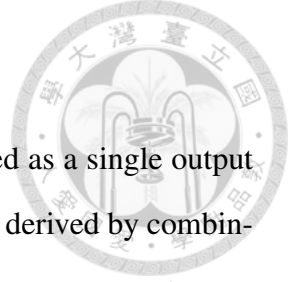


Figure 3.1: Gate-level schematic for illustrating concept of cut set



3.6.3 Enumerative procedure

Obviously the logic gates enclosed by n and C can be regarded as a single output subcircuit. Based on this concept, a multi-output subcircuit can be derived by combining subcircuits enclosed by $(n_1, C_1), \dots, (n_l, C_l)$. As a result, we can enumerate possible subcircuits by utilizing an enumerative procedure which computes all k -feasible cuts.

For convenience let's define the operation $A \bowtie B$ as follows:

$$A \bowtie B = \{u \cup v \mid u \in A, v \in B\}$$

Let $\Delta_1, \dots, \Delta_m$ be cut sets for each input I_x of a library cell, the cut set at output pin can be computed as follows:

$$\left(\Delta_1 \cup \{\{I_1\}\}\right) \bowtie \left(\Delta_2 \cup \{\{I_2\}\}\right) \bowtie \dots \bowtie \left(\Delta_m \cup \{\{I_m\}\}\right)$$

Given the expression above, the cut set Φ for node n can be defined as:

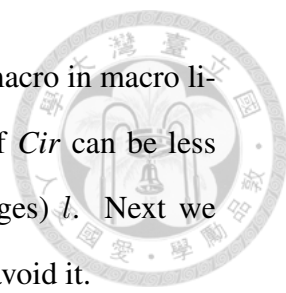
$$\Phi(n) = \begin{cases} \emptyset, & n \in \text{PI} \\ \left(\Phi(I_1) \cup \{\{I_1\}\}\right) \bowtie \dots \bowtie \left(\Phi(I_m) \cup \{\{I_m\}\}\right), & \text{otherwise} \end{cases}$$

One can observe that a node's cut set depends on its drivers' cut sets. Typically there are two approaches for implementing above equation, namely recursion and dynamic programming. In this work we adopt latter approach which traverses the netlist and computes cut sets one by one from PI to PO in topological order. During the traversal, the newly computed key-value pair $(n, \Phi(n))$ is put into a hash table for further lookup. Lastly, it's worth mentioning that we set a length limit to cuts in $\Phi(n)$ simply because the complexity is incredibly high.

3.7 Subcircuit validation

In previous section, we claim that library cells enclosed by $(n_1, C_1), \dots, (n_l, C_l)$ constitute a subcircuit Cir . In this case, one may suppose that Cir has l outputs, and

regard it as a functional block if it happens to be PP equivalent to a macro in macro library. However there is a pitfall. In fact, the number of outputs of Cir can be less than (being swallowed), equal to or greater than (extra output emerges) l . Next we demonstrate the cause of this inconsistency and describe the way to avoid it.



3.7.1 Swallowed output

Refer to Figure 3.2, there are two node-cut set pairs, (n_1, C) and (n_2, C) , while obviously the circuit has only one output. A careless individual may say that there is a half adder with carry out lies at n_2 and summation lies at n_1 . To prevent output from being swallowed, each output should at least feed to one cell which doesn't belong to the subcircuit.

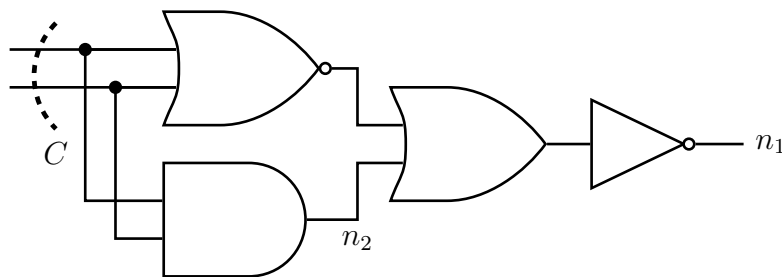


Figure 3.2: A 2-input XOR gate implementation.

3.7.2 Unexpected output

Refer to Figure 3.3, there is only one node-cut set pair (n_1, C) , and the logic function of n_1 with respect to C is exclusive-or. A careless individual may say that there is an XOR gate in between n_1 and C while obviously the subcircuit has two outputs. To prevent unexpected output, each cell in subcircuit shall totally feed to other cells in same subcircuit. In other words, the subcircuit should be *fanout-free* with respect to its roots $\{n_1, \dots, n_l\}$.

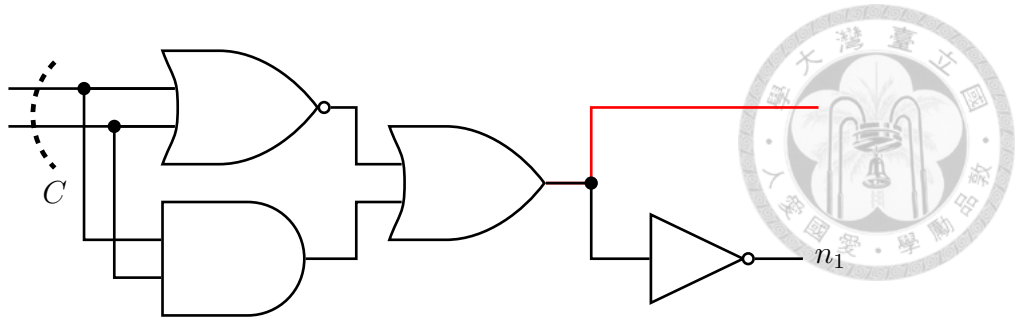


Figure 3.3: A 2-input XOR gate implementation with extra output(red line).

3.7.3 Valid subcircuit

Definition 3.7.1. A subcircuit is said to be **valid** if it satisfies following two conditions:

1. For each po node, it drives at least one cell which isn't part of the subcircuit.
2. For each internal node, it drives only macro cells which are part of the subcircuit.

In this work, all recognized macro blocks are guaranteed to be valid.



Chapter 4

AIG hash function

Since Boolean matching is quite a computation intensive task, one should avoid invoking Boolean matcher frequently. In other words, the number of times of unmatched calls should be reduced. To this end, hash functions are pretty suitable for the situation. In this work we implemented two hash functions, a simple one and a complex one, which compute hash values for AIGs. A valid hash function should give identical hash value for identical input. Here we employ PP equivalence to design our hash function. That is, for two PP equivalent AIGs, they should have identical hash values. Here we introduce a notion of **AIG parameter**. An AIG parameter is an integral value which can be used to characterize an AIG. Generally AIG parameters are derived by injecting test patterns to PI nodes and retrieve response at PO nodes. The hash functions proposed in the following sections make use of AIG parameters to build their return values, i.e., hash values.

4.1 Simple AIG hash function

The simple AIG hash function returns an integer as hash value. The idea is that a hash value can be derived by combining several plain hash values. Here are six parameters we adopt:



1. Number of 1s at output when all inputs are 0
2. Number of 1s at output when all inputs are 1
3. Number of 1s at output when only one input is 0
4. Number of 1s at output when only one input is 1
5. Number of 1s at output when two inputs are 0
6. Number of 1s at output when two inputs are 1

Note that all parameters are themselves mini-hash function. We assert that the resulting hash function which is a composite of above parameters must be valid as well.

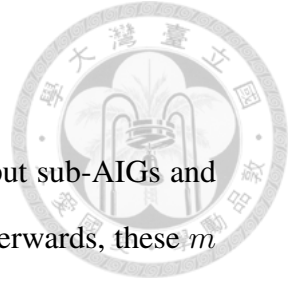
Algorithm 1 illustrates how we compound these parameters into a 32-bit wide hash value. Returned hash value is just a concatenation of $Sum1$, $Sum2$ and $Sum3$. The design principle behind Algorithm 1 is that the input AIG has small size and consequently the concatenation does not exceed 32-bit. It goes without saying that the distinguishing power decreases as the size of AIG enlarges.

Algorithm 1 AIG simple hash function

Input: An AIG network N

Output: Simple hash value (an integral value)

- 1: $Parameters \leftarrow$ Compute the parameters of N
 - 2: $Sum1 \leftarrow$ 1st-of-Parameters + 2nd-of-Parameters
 - 3: $Sum2 \leftarrow$ 3rd-of-Parameters + 4th-of-Parameters
 - 4: $Sum3 \leftarrow$ 5th-of-Parameters + 6th-of-Parameters
 - 5: $Len1 \leftarrow$ bit-width of $Sum1$
 - 6: $Len2 \leftarrow$ bit-width of $Sum2$
 - 7: Rotate $Sum2$ left by $Len1$
 - 8: Rotate $Sum3$ left by $Len1 + Len2$
 - 9: $HashValue \leftarrow Sum1 \oplus Sum2 \oplus Sum3$
 - 10: **return** $HashValue$
-



4.2 Complex AIG hash function

For an AIG with m outputs, we decompose it into m single output sub-AIGs and derive an array of input and output signatures for each sub-AIG. Afterwards, these m arrays will be merged into one large array and sorted in ascending order to obtain a consistent result. The sorted array is then the complex hash value.

Similar to simple hash function, complex hash function also exploits AIG parameters. These parameters are used as constituents of PI signature. In this section we only need four parameters because the performance is just good enough.

1. Output value when all inputs are 0
2. Output value when all inputs are 1
3. Number of 1s at output when only one input is 0
4. Number of 1s at output when only one input is 1

Note that in context of complex hash function, the input AIG is decomposed so the computation process can be customized to single output AIG.

Algorithm 2 presents the procedure to compute complex hash value. The algorithm starts by decomposing input AIG into multiple single output sub- AIGs. Afterwards, it computes an integer array *SignatureArray* for each sub-AIG (line 3 to 17). As mentioned at the beginning of this section, the complex hash is essentially a collection of signatures of primary IOs. Signature of primary output is just simple hash value of the sub-AIG (line 5) while signature of primary input has two portions (line 7 to 15), positive (cofactor) and negative (cofactor). Each portion is obtained by XORing all parameters together (line 11 to 12). For two PP equivalent AIGs, they would have same set of numbers in *HashValue* but with different sequence. As a result, *HashValue* should be sorted and thus transformed into a “canonical” representation for convenience of comparing equivalence (line 18).



Algorithm 2 AIG complex hash function

Input: An AIG network N

Output: Complex hash value (an integer array)

- 1: $HashValue \leftarrow \emptyset$
 - 2: $SubAIGs \leftarrow$ Decompose N into sub-AIGs
 - 3: **for each** sub AIG $N' \in SubAIGs$ **do**
 - 4: $SignatureArray \leftarrow \emptyset$
 - 5: $POsignature \leftarrow$ Simple hash value of N'
 - 6: Push $POsignature$ to $SignatureArray$
 - 7: **for each** PI node $\in N'$ **do**
 - 8: $(N'_{PosCofactor}, N'_{NegCofactor}) \leftarrow$ Compute cofactor of N'
 - 9: $Parameters_{Pos} \leftarrow$ Compute parameters of $N'_{PosCofactor}$
 - 10: $Parameters_{Neg} \leftarrow$ Compute parameters of $N'_{NegCofactor}$
 - 11: $PIsignature_{Pos} \leftarrow$ Exclusive-OR($Parameters_{Pos}$)
 - 12: $PIsignature_{Neg} \leftarrow$ Exclusive-OR($Parameters_{Neg}$)
 - 13: $PIsignature \leftarrow$ Join $PIsignature_{Pos}$ and $PIsignature_{Neg}$ together
 - 14: Push $PIsignature$ to $SignatureArray$
 - 15: **end for**
 - 16: Merge $SignatureArray$ into $HashValue$
 - 17: **end for**
 - 18: Sort $HashValue$ in ascending order
 - 19: **return** $HashValue$
-



Chapter 5

Proposed method

In this chapter we are going to elaborate the core concept of this work. First let's start with the macro library. We are going to describe what's inside and how is it represented in data structure. Second we describe our macro mapping algorithm in detail with pseudocode. Generally speaking, the mapping algorithm essentially enumerates subcircuits with respect to each node in the circuit, and check if there exist a macro who has one output at that node. If multiple macros are found, we then pick the biggest one, i.e. the one with the most gate count.

5.1 Macro library

The macro library is a verilog source file which contains some common combinational circuits and is created by the user. Our macro library currently contains the following types of circuits:

- Multiple input logic gates, including AND/NAND/OR/NOR/XOR/XNOR
- Multiplexer and demultiplexer
- Adder and subtractor

- Encoder and decoder
- Comparator



For a macro with m outputs, we perform symbolic simulation to get m symbolic traces and build an AIG for each symbolic trace. Note that we build m AIGs for every PO instead of one AIG for whole macro. As mentioned in section 3.6, the enumerative procedure of cuts is quite time consuming so we set a limit to number of inputs of macros. In practical the information of macro library is stored into a hash table with key-value pair (*simple hash value*, [*list of tuples*]) where each tuple comprises a sub-circuit belonging to a macro and the subcircuit's complex hash value.

5.2 Macro mapping algorithm

Algorithm 3 describes our procedure for macro block mapping. The algorithm starts by building a DAG (directed acyclic graph) from input netlist D (line 1). We convert circuit wires into vertices and library cells into edges. There is a one-to-one correspondence between netlist wires and graph vertices. An edge from vertex v_x to vertex v_y exists if there exists a cell who has v_x as its input and v_y as its output. With a graph representation, it is more convenient to apply existing graph algorithms on it and perform modification. The transforming process is illustrated in figure 5.1.

This graph is considered as an *alias* of the netlist. That is, all modification on it will reflect on the netlist. In the following context we shall use netlist wire and graph vertex interchangeably. Afterwards, it performs topological sort on the graph (line 2) so as to traverse the netlist in topological order (line 3 to 18). As a high level overview, we want to traverse the netlist from in topological order and try to determine whether there exists a macro who has one of output at current position. Initially no macros are recognized (line 4). We then compute the cut set of current node (line 5). For each cut set, infer macros with algorithm 4 (line 7) and record the result (line 8). If no

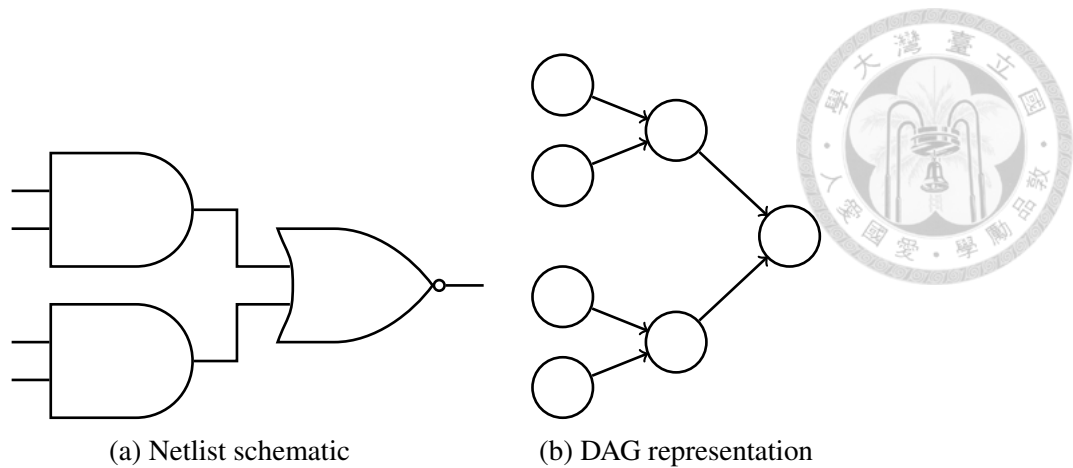


Figure 5.1: Transforming a netlist to a DAG

macros found, we then save the cut set computed previously for future use (line 11). Otherwise the macro with the most number of cells is selected (line 13) and the DAG is rewritten by deleting *macro*'s internal wires and creating edges from *macro*'s input pins to output pins (line 14). In addition, *TopoOrder* is updated as well (line 15). Since rewriting process changes the structure of G , the cut set of *macro*'s output pins should be recomputed (line 16).

Algorithm 3 Macro mapping algorithm

Input: A fully flattened combinational netlist D and a library of macro blocks L

Output: A macro-mapped netlist D'

- 1: $G \leftarrow$ Build DAG G from the netlist D
 - 2: $TopoOrder \leftarrow$ Topological sort on G
 - 3: **for each** vertex $v \in TopoOrder$ **do**
 - 4: $MappedMacros \leftarrow \emptyset$
 - 5: $CutSet \leftarrow$ Compute cut set with respect to v
 - 6: **for each** cut $c \in CutSet$ **do**
 - 7: $Candidates \leftarrow$ Infer macros starting from c
 - 8: $MappedMacros \leftarrow MappedMacros \cup Candidates$
 - 9: **end for**
 - 10: **if** $MappedMacros$ is empty **then**
 - 11: Cut set of $v \leftarrow CutSet$
 - 12: **else**
 - 13: $macro \leftarrow$ Biggest among $MappedMacros$
 - 14: Rewrite G by viewing $macro$ as a basic block
 - 15: Delete vertices in $TopoOrder$ which reside in $macro$
 - 16: Compute cut sets of $macro$'s primary outputs
 - 17: **end if**
 - 18: **end for**
-

Algorithm 3 can be repeated several times to recognize bigger macros and increase coverage.

Algorithm 4 shows a macro inference algorithm invoked by algorithm 3 at line 7. Given a circuit node, one of its cut and a macro library, algorithm 3 tries to discover macros who have a sub-circuit enclosed by the cut. First we build AIG of the given cut for subsequent operation (line 2). This can be easily done by converting each cell in the cut to AIG node one-by- one. Then we use hash function to accelerate the process of finding possible equivalent macro block sub-circuits (line 3 to 4). The main loop (line 5 to 23) iteratively check sub-circuit N' in *ListOfAIG*. At each iteration, the first step is to prove equivalence of N and N' with a formal tool (line 6), i.e. Boolean matcher. An iteration will be discarded once the prove fails (line 7). If N' belongs to a multi-output macro block, the algorithm tries to search for other outputs (line 10 to 19). Since they must be located at topologically subsequent nodes, we adopt an enumerative approach. If all outputs can be found and the selected sub-circuit is valid (line 17), push it to set *InferredMacros* and finish a iteration of main loop. On the other hand, if N' is itself a macro block, just collect it without doing anything special (line 21).



Algorithm 4 Macro inference algorithm

Input: A circuit node n , a cut c and a library of macro blocks L

Output: A set of macro blocks

```

1:  $InferredMacros \leftarrow \emptyset$ 
2:  $N \leftarrow$  Build AIG from  $c$ 
3:  $HashVal \leftarrow$  Compute hash values of  $N$ 
4:  $ListOfAIG \leftarrow$  Use  $HashVal$  to lookup subcircuits in  $L$ 
5: for each AIG  $N' \in ListOfAIG$  do
6:   if  $N$  and  $N'$  is not PP-equivalent then
7:     continue
8:   end if
9:   if  $N'$  is part of a multi-output macro  $M$  then
10:     $CandidateSet \leftarrow$  Topologically subsequent nodes of  $n$ 
11:    Compute cut sets for each node in  $CandidateSet$ 
12:    for each subcircuit  $Ckt$  of  $M$  do
13:      for each node  $\in CandidateSet$  do
14:        Check whether the node matches  $Ckt$ 
15:      end for
16:    end for
17:    if all  $Ckts$  are matched and resulting subcircuit is valid then
18:       $InferredMacros \leftarrow InferredMacros \cup M$ 
19:    end if
20:    else if subcircuit formed by  $c$  is valid then  $\triangleright N'$  is itself a macro
21:       $InferredMacros \leftarrow InferredMacros \cup N'$ 
22:    end if
23:  end for
24: return  $InferredMacros$ 

```

5.3 Improvement

In this work we apply some heuristics and techniques to accelerate the proposed method. They can be described in these aspects:

1. Data structure of cut set

Although is a set in view of mathematics, it don't have to support operations such as union, intersection, and difference. The only thing to take into account in practice is the operation depicted in section 3.6. To this end, cuts with same length can be grouped together and stored in a linked list for constant time inser-



tion. A cut set is implemented with an array where each element is a pointer to a list of cuts with same length. This strategy can reduce the cut look-up time if the desired length of cut is given.

2. Graph traversal in algorithm 3

In a netlist there might be chains of buffers or inverters. These chains can be merged into the macro without changing its functionality so we shall skip them when traversing the circuit nodes.

3. *CandidateSet* in algorithm 4

When initializing *CandidateSet*, if inputs of N' covers inputs of M , we shall just include nodes which are solely driven by inputs of N' (directly or indirectly). This process can be done using a breadth first search based strategy. The idea is simple: a wire is included if and only if its driving nets are included previously or source nodes. Take figure 5.2 as an example. Let the blue wires be source nodes of the algorithm. Both G4's output and G5's output are included because their driving nets are source nodes. In addition, G6's output is included as well because its driving net, G4's output has been included. However, G7's output will not be included because one of its PI, G3's output, wasn't included previously.

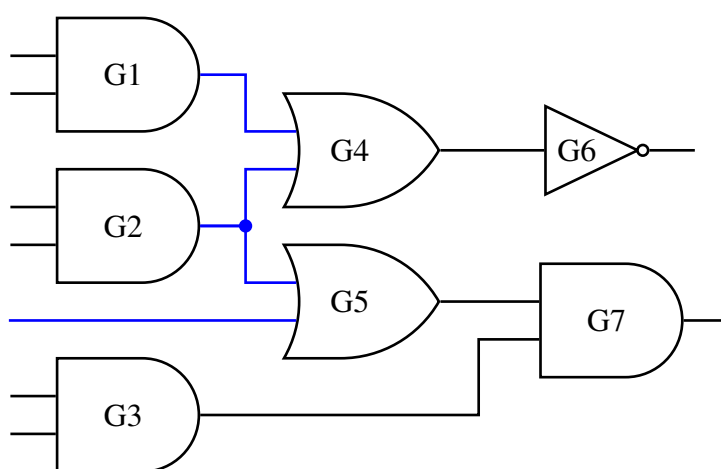
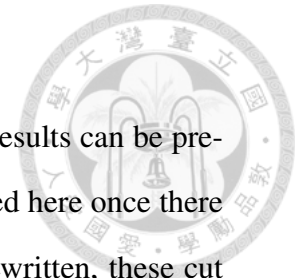


Figure 5.2: Illustration for collecting candidate PO nodes

4. Cut sets of candidate nodes in algorithm 4

When computing cut sets for candidate nodes, the computed results can be preserved for further use. We can directly use the cut set computed here once there is a need. If the state of the graph had changed, i.e. graph rewritten, these cut sets shall be discarded.





Chapter 6

Experimental results

The proposed algorithm was implemented in C and exploited the AIG data structure and Boolean mathing command in Berkeley ABC logic synthesis framework [12]. Experiments were performed on an Intel® Core™ i5-7400 CPU clocked at 3.50GHz with 8 GB of RAM and the circuits are synthesized using the 180 nm library named GSCLib which is provided by IWLS2005 benchmark. We evaluated the proposed method on the ISCAS benchmark circuits. Table 6.1 provides the experiment result of this work. There are total seven columns, from left to right: benchmark circuit name, number of inputs of the circuit, number of outputs of the circuit, number of gates in the circuit, number of macros recognized by our algorithm, the proportion of gates mapped by our approach, and lastly running time.

Benchmarks	Inputs	Outputs	Cells	Mapped macros	Coverage	Runtime(sec)
c880	60	26	225	22	23.1%	13
c2670	233	140	327	16	13.5%	16.4
c3540	50	22	559	15	6.4%	481
s1238	14	14	508	1	0.6%	123.7
s1423	19	5	657	105	59.8%	10.9
s5378	37	49	1132	-	-	>1200

† here we only enumerate 6-feasible cuts

Table 6.1: Experiment result of the proposed reverse engineering algorithm.

The experiment result shows that the algorithm tends to recognize numerous small macros and distributed evenly in the netlist. To be a handy EDA tool, it should have produced result of large macros with small quantity. For benchmark c1238, there's only one recognized macro but takes quite a while. The reason is that the macro mapping algorithm relies on constantly recognizing macros to shrink the circuit size and thereby reduce the computational complexity. As section 3.6 shows, the number of cuts grows exponentially as cut size increases. We observed that a node's cut set can have 500+ cuts when there's only few macros are recognized. The running time would explode with so many cuts because Boolean matching is a computation intensive task. Even though we impose time limit on the Boolean matcher, the running time is still enormous due to too many calls to the Boolean matcher.



Chapter 7

Conclusion and future works

Both industry and academia have been dealing with hardware reverse engineering for several decades. In this thesis, we have presented reverse engineering techniques on gate-level netlists. The proposed method is based on a macro block mapping scheme and the objective was to maximize the size of macro blocks and minimize the amount of macro blocks. The method involves subcircuit enumeration and Boolean matching problem. Because they both have quite high time complexity, our experiment result shows that the running time would explode as scale of netlist grows and there was still a lot of room for improvement. One major problem of the macro library is the generality. It's hard to include all types of macros into the macro library, so we need a more generic approach. Another possible solution would be artificial intelligence (AI). In AI there is a set of problems called *classification* that classifies an object to a category. The idea is to extract characteristics from the circuit construct and let the AI algorithm learn them. Hoping we can find an AI model that is able to classify hardware circuits according to their characteristics.




Bibliography

- [1] M. Fyrbiak, S. Wallat, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, R. Tessier, and C. Paar, “Hal — the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion,” *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 3, pp. 498–510, 2019.
- [2] X. Zhang and M. Tehranipoor, “Case study: Detecting hardware trojans in third-party digital ip cores,” in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 67–70, 2011.
- [3] M. Fyrbiak, S. Strauss, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, “Hardware reverse engineering: Overview and open challenges,” *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pp. 88–94, 2017.
- [4] M. C. Hansen, H. Yalcin, and J. P. Hayes, “Unveiling the iscas-85 benchmarks: a case study in reverse engineering,” *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.
- [5] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, “Reverse engineering digital circuits using functional analysis,” in *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13*, (San Jose, CA, USA), pp. 1277–1280, EDA Consortium, 2013.
- [6] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, “Reverse engineering digital circuits us-



- ing structural and functional analyses,” *IEEE Transactions on Emerging Topics in Computing*, vol. 2, pp. 63–80, jan 2014.
- [7] W. Li, A. Gascon, P. Subramanyan, W. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, “Wordrev: Finding word-level structures in a sea of bit-level gates,” in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, (Los Alamitos, CA, USA), pp. 67–74, IEEE Computer Society, jun 2013.
- [8] T. Meade, S. Zhang, and Y. Jin, “Netlist reverse engineering for high-level functionality reconstruction,” in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 655–660, 2016.
- [9] Y. Shi, C. W. Ting, B. Gwee, and Y. Ren, “A highly efficient method for extracting fsm’s from flattened gate-level netlist,” in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 2610–2613, 2010.
- [10] X. Wei, Y. Diao, T. Lam, and Y. Wu, “A universal macro block mapping scheme for arithmetic circuits,” in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1629–1634, 2015.
- [11] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, “Fraigs: A unifying representation for logic synthesis and verification,” tech. rep., 2005.
- [12] R. Brayton and A. Mishchenko, “Abc: An academic industrial-strength verification tool,” in *Proceedings of the 22Nd International Conference on Computer Aided Verification, CAV’10*, (Berlin, Heidelberg), pp. 24–40, Springer-Verlag, 2010.
- [13] H. Katebi and I. L. Markov, “Large-scale boolean matching,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, (Los Alamitos, CA, USA), pp. 771–776, IEEE Computer Society, mar 2010.

- 
- [14] H. Katebi, K. A. Sakallah, and I. L. Markov, “Generalized boolean symmetries through nested partition refinement,” in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD ’13, (Piscataway, NJ, USA), pp. 763–770, IEEE Press, 2013.
- [15] J. Cong, C. Wu, and Y. Ding, “Cut ranking and pruning: Enabling a general and efficient fpga mapping solution,” in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA ’99, (New York, NY, USA), pp. 29–35, ACM, 1999.
- [16] S. Chatterjee, A. Mishchenko, and R. Brayton, “Factor cuts,” in *2006 IEEE/ACM International Conference on Computer Aided Design*, pp. 143–150, Nov 2006.