

# 國立交通大學

資訊工程系

碩士論文

保護 Soft-IP 的資訊隱藏技術



An information hiding technique for  
Soft-IP protection

研究生：陳仕偉

指導教授：陳昌居 博士

中華民國九十四年六月

# An information hiding technique for Soft-IP protection

研 究 生：陳仕偉

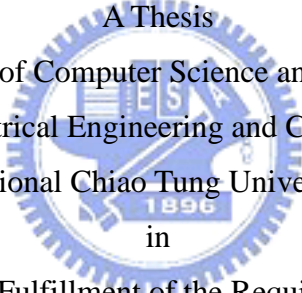
Student : Shi-Wei Chen

指 導 教 授：陳昌居

Advisor : Chang-Jiu Chen

國 立 交 通 大 學  
資 訊 工 程 學 系  
碩 士 論 文

A Thesis  
Submitted to Department of Computer Science and Information Engineering  
College of Electrical Engineering and Computer Science  
National Chiao Tung University  
in  
partial Fulfillment of the Requirements  
for the Degree of Master  
in  
Computer Science and Information Engineering  
June 2005

The logo of National Chiao Tung University is a circular seal. It features a central shield with a book and a torch, surrounded by the university's name in Chinese and English. The year '1896' is inscribed at the bottom of the seal.

Hsinchu, Taiwan, Republic of China

# 保護 Soft-IP 的資訊隱藏技術

研究生：陳仕偉

指導教授：陳昌居

國立交通大學 資訊工程學系

## 摘要

由於近年來電腦輔助設計軟體和超大型積體電路製程的進步，產品從研發到推出市場的時間愈來愈短。為了因應急劇縮短的研發時間，必須重複使用事先設計好的元件。因此在這種情況下，創造了一個新的商機——販售事先經過設計且能夠重複使用的元件(即矽智財、Intellectual Property)。但是這個市場必須要有保護的機制以避免 IP 被惡意的濫用或是轉售。在本論文中，我們提出一個藉由在有限狀態機中隱藏資訊以保護 Soft-IP 的新方法。這個方法將有限狀態機中的 State 重新編碼，使得在特定路徑上相鄰兩個 State 編碼的差值即為所要隱藏的資訊。經過分析發現，我們的方法具有難以被查覺、難以被篡改，以及難以被假造的特性，並且利用 Scan-Chain 即可檢測在特定電路中是否有隱藏資訊。我們用我們的方法將 8 個 8bit 的資訊隱藏在 UART Receiver 電路中，得到的結果顯示，這些電路不僅仍然正常運作，而且只用原來的 State 就能隱藏資訊。

# **An information hiding technique for Soft-IP protection**

**Student : Shi-Wei Chen Advisor : Dr. Chang-Jiu Chen**

**Department of Computer Science and Information Engineering**

National Chiao-Tung University

## **Abstract**

As the progress of EDA tools and manufacturing technologies, we get shorter and shorter time-to-market. In order to catch up with the decreasing time-to-market, reuse pre-defined and pre-verified components is inevitable. This creates a new market of selling reusable pre-defined components ( i.e. Intellectual Property ). But the IP must be protected by some techniques from malicious duplication. In this thesis, we propose a new method to protect Soft-IP by hiding information in FSMs. We re-encode the state coding in the way that the difference of two states of selected edge in a specific path is equal to the information we want to hide. Through analysis, we found our method has the property of hard to be observed, hard to be removed, and hard to be faked. Using scan-chain, we can detect whether a specific FSM hides our information or not. We hide eight 8-bit information data in a UART receiver using our method. The result shows that the UART receiver works correctly and we can hide the eight 8-bit information in it without adding new states.

# Acknowledgment

這兩年的研究生生活中，首先感謝指導老師陳昌居老師提供了穩定的環境和資源讓我能專心研究，同時也感謝黃年時學長和沈銘峰同學的大力協助，沒有他們的話，我不可能完成這項研究。當然也感謝同實驗室的夥伴們，以及家人們給予我的支持和鼓勵促使這篇論文得以順利完成，謝謝大家。



# Contents

摘要 .....	i
ABSTRACT.....	ii
ACKNOWLEDGMENT .....	iii
LIST OF FIGURE .....	vi
LIST OF TABLE.....	viii
Chapter 1 Introduction.....	1
Chapter 2 Related Works.....	5
2.1 OBJECTIVE AND METRICS OF INDIRECT IP PROTECTION.....	5
2.2 INDIRECT IP PROTECTION TECHNIQUES .....	6
2.2.1 Hierarchical Watermarking .....	6
2.2.2 Behavioral Level .....	6
2.2.3 Logic Level.....	13
2.2.4 Field Programmable Gate Arrays.....	14
2.2.5 Standard Cell Place and Route.....	15
Chapter 3 Background of the Proposed Method .....	16
3.1 BASIC DEFINITIONS .....	16
3.2 MOTIVATION AND BASIC CONCEPT.....	16
3.3 COMPARE WITH EVEN/ODD ENCODING.....	18
3.4 THE DETECTION OF THE SIGNATURE.....	20
Chapter 4 State Difference Encoding.....	21
4.1 INPUTS AND OUTPUTS .....	21
4.2 THE ENCODING PROCESS .....	22
4.3 VALID VECTOR.....	23
4.4 FINDING THE PATH .....	25
4.5 FORMULATING THE ENCODING REQUIREMENT INTO INLP .....	29
4.6 GENERATING THE GATE-LEVEL NET-LIST .....	31
4.7 THE DETECTING PROCESS .....	34
4.8 A COMPLETE EXAMPLE.....	34
4.9 EVALUATION OF STATE DIFFERENCE ENCODING .....	39
4.9.1 The Robustness Evaluation .....	40
4.9.2 The Overhead Evaluation .....	41
Chapter 5 Experimental Results.....	43

**Chapter 6 Conclusions and Future Works .....53**  
**References.....55**



# List of Figure

**Figure 2-1 The STG of original design.....7**

**Figure 2-2 Modified STG .....8**

**Figure 2-3 An alternative way to embed the watermark.....9**

**Figure 2-4 The improvement version of Figure 2-3 .....9**

**Figure 2-5 An example of applying the method described in [7] : (a) the original  
FSM, (b) adding transitions to embed watermark, (c) augmenting input and  
adding transitions..... 11**

**Figure 2-6 The explanation of embedding the signature ‘A7’ in the register  
assignment solution.....13**

**Figure 3-1 An example of state difference encoding (a) the original STG of FSM (b)  
the STG after applying state difference encoding to hide 0x10, 0x20, 0x30 .18**

**Figure 3-2 An example of even/odd encoding (a) the original FSM (b) a possible  
solution of even/odd encoding .....19**

**Figure 4-1 The encoding procedure of state difference encoding.....23**

**Figure 4-2 The kiss2 file format.....23**

**Figure 4-3 An example of valid vector (a) the FSM can only hide three  
information without valid vector (b) with valid vector, A->E can be used to  
hide the fourth information.....24**



<b>Figure 4-4 An example of the approach to solve the data item “00” in the information sequence. (a) The information sequence is divided into two parts. (b) the sample FSM. (c) a solution of sub sequence 1. (d) a solution of sub sequence 2. ....</b>	<b>26</b>
<b>Figure 4-5 The FSM of Figure 4-4(b) with shadow states .....</b>	<b>28</b>
<b>Figure 4-6 The pseudo code of path finder .....</b>	<b>29</b>
<b>Figure 4-7 The constraints of Figure4-1(b) for INLP .....</b>	<b>30</b>
<b>Figure 4-8 The pseudo code of INLP solver.....</b>	<b>31</b>
<b>Figure 4-9 A simple FSM.....</b>	<b>32</b>
<b>Figure 4-10 The Verilog description of the FSM in Figure 4-9 .....</b>	<b>33</b>
<b>Figure 4-11 The pseudo code of HDL generator .....</b>	<b>33</b>
<b>Figure 4-12 The architecture of UART receiver .....</b>	<b>35</b>
<b>Figure 4-13 The STG of UART receiver .....</b>	<b>36</b>
<b>Figure 4-14 The encryption process of information sequence .....</b>	<b>36</b>
<b>Figure 4-15 The three constraints of UART receiver .....</b>	<b>37</b>
<b>Figure 4-16 The state encoding, valid vector, hidden information, and inputs used by detecting process .....</b>	<b>38</b>
<b>Figure 4-17 The architecture of UART receiver with verification circuit .....</b>	<b>39</b>

# List of Table

<b>Table 4-1 The state encoding of UART receiver.....</b>	<b>38</b>
<b>Table 5-1 The statistic data of benchmark ISCAS'91 .....</b>	<b>45</b>
<b>Table 5-2 State bits of each circuit when hiding different data items .....</b>	<b>49</b>
<b>Table 5-3 The number of literals before and after embedding 128 bits information</b> <b>.....</b>	<b>52</b>



# Chapter 1 Introduction

As the progress of deep-submicron manufacturing processes recent years, semiconductor density reaches a level that allows all elements of an entire system being merged onto a single chip. Unfortunately, the design productivity has not kept the pace. A study described in [11] shows that semiconductor densities are increasing about 58% per year while design productivity only advances 21% annually. Thus, we need a new design methodology to fill the gap between hardware capacity and design productivity and the system-on-a-chip (SoC) design methodology shows up. In this new design methodology, designers integrate several pre-designed and pre-verified cores, also called intellectual property (IP), onto a single chip. These cores may be obtained from internal sources or a third-party vendor. By reusing these cores, designers save a great amount of time and work that can be spent on developing new products. As a consequence, the design productivity increases dramatically.

Traditionally, IP can be classified into behavioral description (soft IP), structural description (firm IP), or physical description (hard IP) according to the degrees of freedom left to the user to manipulate it. Hard IP is generally in form of routed layout for specified process technology and with very less freedom for further migrations. On the other hand, soft IP, in contrast to hard IP, provides the most freedom for use. It is possible to map soft IP to a variety of final layouts based on different synthesis strategies and process technologies. However, a soft IP user must synthesize, optimize, and validate for the soft IP before integrating it into the system. Firm IP is in the position between hard IP and soft IP. It provides more flexibility and reconfigurability than hard IP and better reuse potential than soft IP. Firm IP is usually in form of technology-independence gate-level net-list or HDL description and can or can not be

mapped to different process technologies depending on the IP designer.

Since an IP represents a digital circuit designed for specific function but do not has a concrete physical manifestation, it is possible for customers to resell the IP as their own even without understanding the details of the IP. This problem has become a great concern of IP vendors and thereby the viability of the SoC design methodology depends on how to protect IPs from malicious duplication or unauthorized use.

In nowadays, there exist two main approaches on IP protection, direct protection and indirect protection. The goal of direct protection is to prevent unauthorized users to use the IP. Hence, direct protection tries to make the medium in which IPs are stored and exchanged more secure. This is usually done by applying encryption algorithms like RSA to IPs. Then the encrypted IPs are placed in a public IP exchange medium, and only authorized users can get private key to decrypt them. The direct protection may potentially not be effective in avoiding infringement due to that once the IP is decrypted, the user can do anything they want to the IP. Thus, indirect protection must be applied to the IP simultaneously to avoid malicious duplication. Most researches related to IP protection including this thesis focus on indirect IP protection.

The purpose of indirect protection is to obtain a unique signature from IPs or embed one into IPs. This allows the IP designer to determine whether a given IP has been infringed upon. Indirect protection can further be divided into two kinds according to whether the original design is modified or not. Active protection embeds a signature which represents the authorship to the IP and in most cases, it will modify the IP. While passive protection tries to obtain a signature from existing feature, hence, it will not alter the IP.

Over the past few years, a considerable number of studies have been made on indirect IP protection using watermarking technique. Watermark is a signature used to

identify the authorship and watermarking means to embed the watermark into something that we want to protect. Originally watermarking is used in video/audio application since the watermark is hardly visible or audible by human eyes and ears. Moreover, the watermark can be extracted and identified even the original picture or music has changed by applying some algorithms to it. Recently, watermarking is also used to protect IPs. There is a similar technique for IP protection, called fingerprinting, which is generating a signature from a design using the existing features and properties at a specified abstraction level.

Although a lot of methods are proposed for indirect protection using watermarking technique, most of them are focus on hard IP protection since it is easier and is efficient for detecting the watermark. Hard IPs can only be used in a specific process technology. Thus, as long as the process technology of silicon foundries progresses, the chip designers must buy these IPs again even though the design of these IPs are the same. This situation makes the designers quite annoying. Hence, selling soft IP or firm IP instead of hard IP will solve the problem since soft IP provides more flexibility for users and the designer can resynthesize them to match new process technology. However, soft IP and firm IP are still not prevalent today because of the lack of soft or firm IP protection. Thus, our study is mainly focus on soft IP protection in order to solve the embarrassing problem.

In this thesis, we present a new method to embed a signature into finite state machine (FSM) by using a special state encoding during synthesis process. We first choose a path in the FSM and encode the states in the way that the difference of every two state along the path equals to a part of the signature we embed. Since the state encoding will not change at all levels under the behavioral level, our approach can protect the IP in the design hierarchy from firm IP to the routed layout. Besides, our method can hide longer signature with less or even no area overhead compared to

other proposed method.

The remainder of this paper is organized as the following. Chapter 2 discusses some researches related to IP protection. Chapter 3 introduces the concept of our method. Chapter 4 details the proposed approach itself. Chapter 5 evaluates the approach through an experiment. Chapter 6 gives our conclusions and the directions for future development.



## Chapter 2 Related Works

In this chapter, we will introduce the objective and metrics of an indirect IP protection method, and some IP protection methods that have already been proposed.

### 2.1 Objective and Metrics of indirect IP protection

For a successful indirect IP protection technique, it should have the following properties:

1. **Proof of Authorship and Authenticity** : the signature embedded in IP must be unambiguous.
2. **Correctness of functionality** : the added functionality should not affect the correctness of original functionality.
3. **Low overhead** : the hardware overhead should be low.
4. **Ease of detection** : the signature should be detectable with a low cost technique.

The following metrics are defined to evaluate a specific indirect IP protection technique.

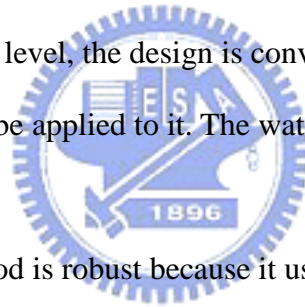
1.  $P_u$  : the probability of uniqueness. It means the odds that another design carries an identical signature.
2.  $P_m$  : the probability of a miss. It means the odds that the signature is not detected after tampering although the signature still exists.
3.  $P_f$  : the probability of false alarm. It means the odds that a signature is detected unintentionally. Generally,  $P_u = P_f$ .

## 2.2 Indirect IP protection techniques

In this section, we introduce some indirect IP protection techniques according to the design abstraction level they applied to.

### 2.2.1 Hierarchical Watermarking

In [5], the author proposes a method to protect IP by marking each step of the synthesis and layout processes with a specific watermark. At the layout level, typical mixed signal constraints, such as symmetry, grouping, clustering, fixed objects, and alignment are used to form the watermark. At the gate level, the watermark is used to generate a pseudo-random sequence of symbols, and each symbol represents a given standard cell. At the structural level, the design is converted to a directed graph and some specific constraints can be applied to it. The watermark is coded in these constraints.



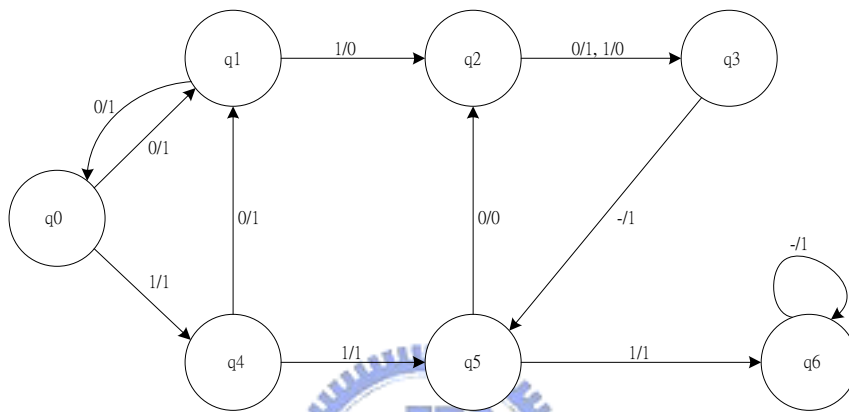
As we can see, this method is robust because it uses multiple watermarks. If one of them is removed, other watermarks can still be detected. On the other hand, the method is the most costly and complex compared to other methods.

### 2.2.2 Behavioral Level

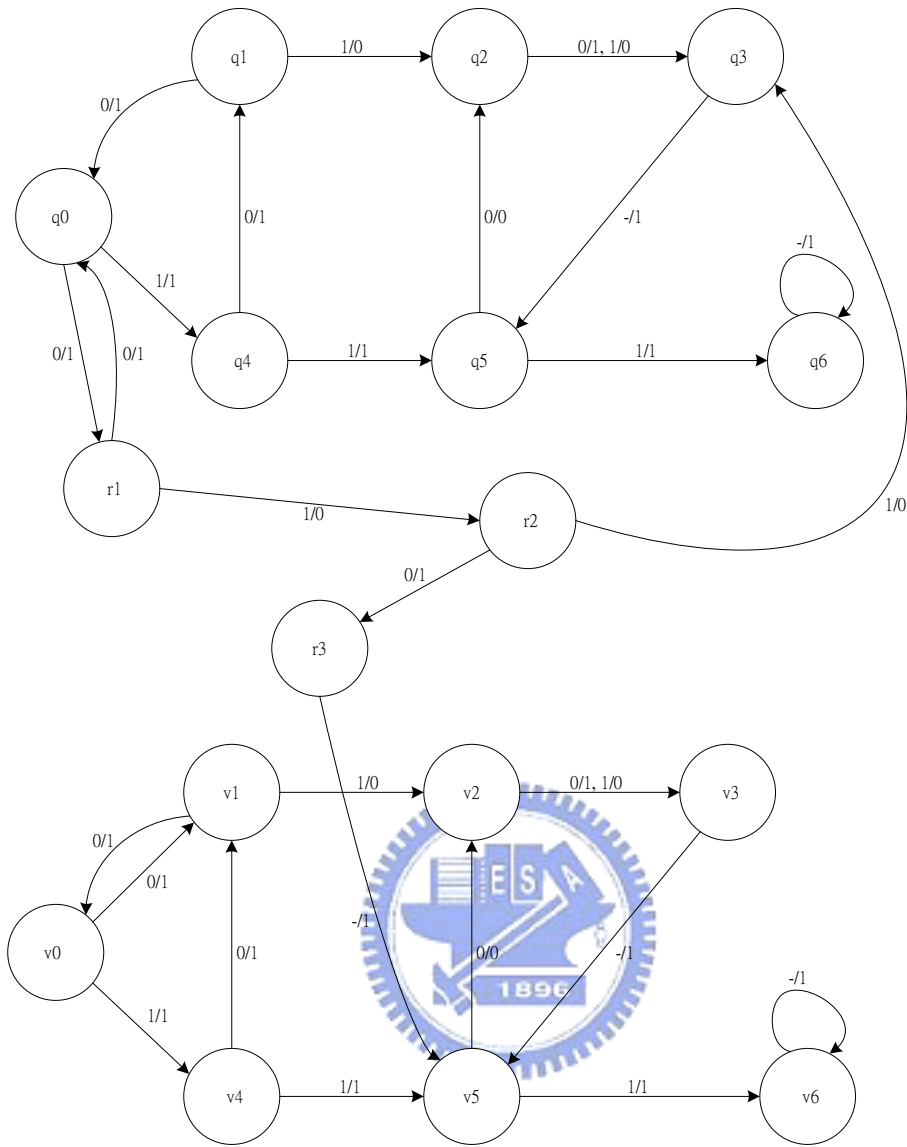
In [2], A. L. Oliveria proposes a method to watermark FSMs by adding a set of new states in such a way that when the special input sequence is applied to the modified FSM, the added states are traversed in a predefined order. For example, consider the STG shown in Figure 2-1. Assume we want to embed a three bit signature 010, we then create three new states  $r_1, r_2, r_3$  together with states  $v_i$  and change the source and destination of the involved edges in Figure 2-1. Then, we will obtain the STG shown in Figure 2-2. Note that the only way to traverse states



$r_1, r_2, r_3$  in this order is to apply the sequence 010 from the reset state. Once states  $r_1, r_2, r_3$  are traversed, the FSM enters states  $v_i$ . Then we can claim that the sequence of traversed states  $r_1, r_2, r_3$  exhibits a specific property that can be used to identify the design. We shall note that the states  $r_1, r_2, r_3$  must maintain the original outputs and state transitions of sequence 010 since there are some input sequences that contain the same prefix as 010 but are not equal to 010.

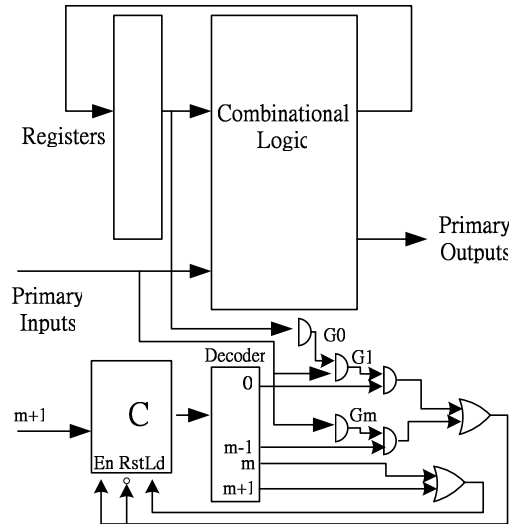


**Figure 2-1 The STG of original design**



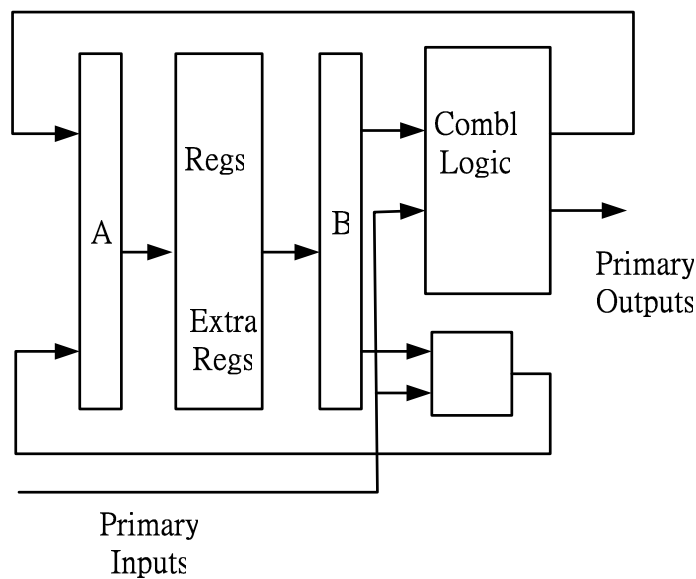
**Figure 2-2 Modified STG**

It seems that the area overhead of this method is very high because the number of states of the watermarked STG is more than twice as much as the original STG. The authors use a simple technique to avoid this problem. Consider the FSM in Figure 2-3, the authors attach a small FSM with the original FSM. The function of this small FSM is to monitor the primary input and keeps track of the presence of the author signature. If the author signature has been inputted, the small FSM asserts a signal to notify the presence of the author signature.



**Figure 2-3 An alternative way to embed the watermark**

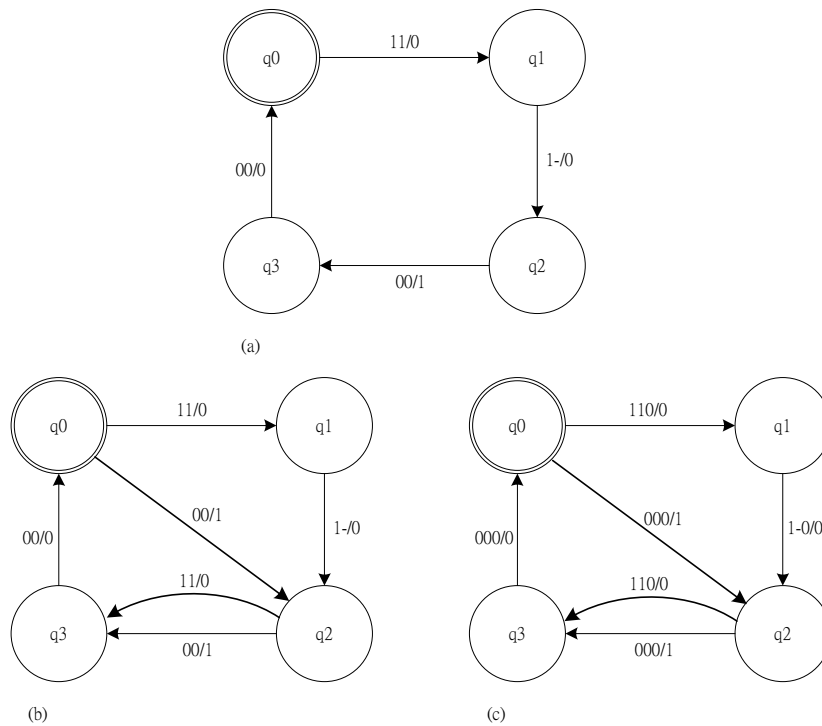
However, there still is a problem of this new method. Because the small FSM is attached to the original FSM, the small FSM has nothing to do but the primary input with the original FSM. Hence, the small FSM is easy to be detected and removed. Again, the authors use another simple technique to solve it. The authors try to make a relationship between the registers used by the original FSM and the registers used by the small FSM by a mapping. Thus, the attackers can not easily distinguish which register is used by the attached FSM. This is illustrated in Figure 2-4.



**Figure 2-4 The improvement version of Figure 2-3**

This method has three main advantages : (1) simple computation. (2) the signature can be detected in lower-level derived design. (3) the modification can be directly added to a synthesized design instead of resynthesis. But the area overhead may be unacceptable for embedding large signature. The authors claim that they can use MD5 to shorten the signature to 128 bits without loss of strength of the authorship proof.

In [7], the author proposes a simple technique to watermark sequential functions which may be Completely or Incompletely specified FSMs. The main idea of this method is to represent the watermark as the primary output sequence that shall be produced only when a special input sequence which is unspecified in the original FSM is applied. By using unused path to encode watermark, the functional correctness of the FSM is guaranteed. For example, Figure 2-5(a) is the STG of a simple FSM which has two input bits and one output bit. Assume a watermark **01** is enough for identify this design, Figure 2-5(b) is a possible result of applying this method to the FSM of Figure 2-5(a). It uses two unspecified transitions,  $q_0$  to  $q_2$  when input sequence is 00 and  $q_2$  to  $q_3$  when input sequence is 11, to encode the watermark.



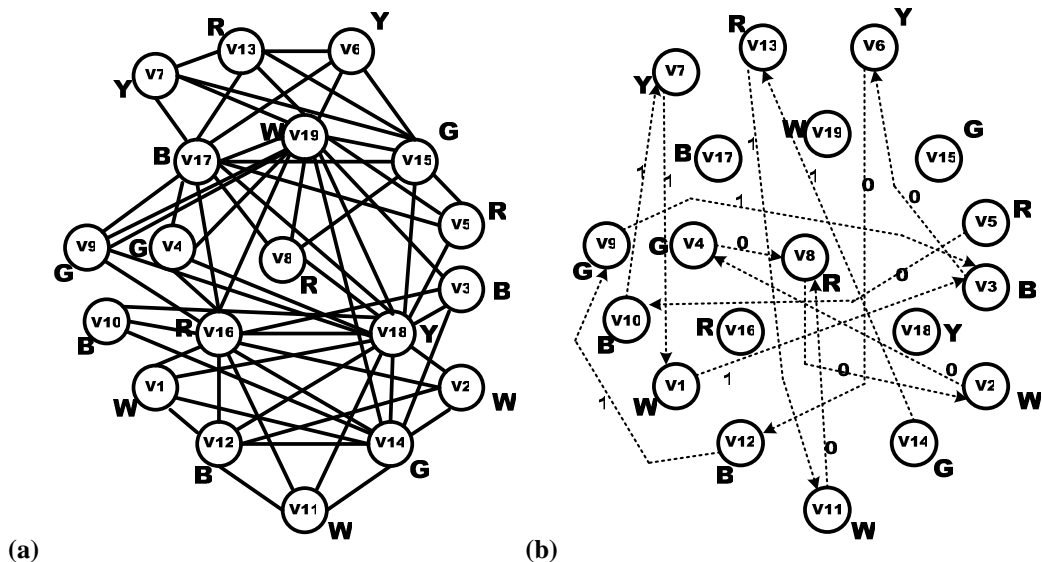
**Figure 2-5 An example of applying the method described in [7] : (a) the original FSM, (b) adding transitions to embed watermark, (c) augmenting input and adding transitions**

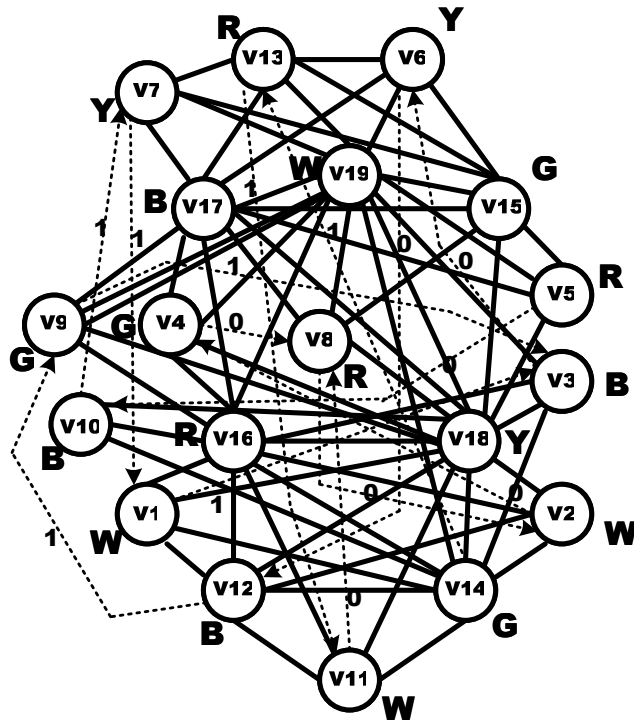
There is a special case that all transitions of the given FSM are specified. In this case, we have no choice but to increase the number of primary inputs by one. Figure 2-5(c) shows augmenting a new input and adding transitions into Figure 2-5(a). The main advantage of this method is simple computation and the ease of detection of the watermark. And the signature is detectable in lower-level derived design.

In [6], Inki Hing and Miodrag Potkonjak propose a method for soft-IP protection by inserting the author signature into the process of register assignment during behavior synthesis. This method takes advantage of the fact that there are usually more than one solutions of register assignment. An interval graph is an undirected graph that each node in this graph represents one register in behavioral description. In the interval graph, an edge between two nodes means the life time of the registers represented by these two nodes is overlapped. Thus, these registers can not be used at one time and edges in an interval graph are also called timing constraints. Register

assignment means to find a way to color an interval graph such that for every two nodes with an edge between them can not be the same color. The solution of coloring problem is not unique, so legal register assignment is not unique, too.

There are three steps to apply Inki Hing and Miodrag Potkonjak's method to a FSM. First, we must obtain an interval graph from the behavioral description of the FSM. Then, we use a special way to add the author signature into the interval graph. Starting from node v1, if the first bit we want to hide is '1', we add a new edge between v1 and v3. Otherwise, we add a new edge between v1 and v2. Repeat this step until all signature bits are added into the interval graph. Last, use any method for solving coloring problem to obtain a register assignment of the modified interval graph. Besides the second step, other two steps are the same as normal process of register assignment. Figure 2-6 shows the process of embedding a signature of 14 bits ASCII code 'A7' into an interval graph. Figure 2-6(a) is the original interval graph, Figure 2-6(b) contains all the edges with encoded values we want to embed, and Figure 2-6(c) is the result interval graph.





Start node	v1	v2	v3	v4	v5	v6	v7	
End node	v3	v4	v6	v8	v10	v12	v1	
Embedded	1	0	0	0	0	0	1	= A
Start node	v8	v9	v10	v11	v12	v13	v14	
End node	v2	v3	v7	v8	v9	v11	v13	
Embedded	0	1	1	0	1	1	1	= 7

Figure 2-6 The explanation of embedding the signature 'A7' in the register assignment solution

The advantages of this method are easy computation and the embedded author signature is difficult to be observed. Also, the unoptimal register assignment due to the extra constraints provides a robust proof that there is a signature embedded. On the other hand, there are some disadvantages of this method. This method is not suitable for embedding long signature since one register can only embed one bit. The other problem is that given a lower-level design, it is difficult to detect the signature.

### 2.2.3 Logic Level

In [3], D. Kirovski proposes a less resilient but simpler watermarking technique in gate level. In this technique, the watermark is represented as a set of primary

outputs that does not appear in the original logic network. These gates are selected according to the pseudo-random bits generated from the author's signature. There are several steps to apply this method. First, in order to ensure that each watermark of different author signature is unique, all gates in the circuit must save in a standard way. Then, every gate is assigned an unique identifier and some gates will be picked up according the pseudo-random. Last, these gates will become primary output and these gates form the watermark.

In assigning identifiers to each gate, we must ensure that every gate has different identifier in order to prevent the watermark from misinterpretation. Since to check two gates are functionally identical is difficult, the authors propose a heuristic way to assign identifiers according to eight criteria. In general, it is merely impossible that all the eight criteria of two gates are identical. Unfortunately, this case still exists. If two gates are not distinguishable using these eight criteria, we assign random unique identifiers to them and memorize the assignment for future proof of authorship.

#### **2.2.4 Field Programmable Gate Arrays**

In [8], the watermark is in form of a bit stream and directly placed in the design of FPGA. This is done by using the lookup table of unused configurable control logic blocks. Each unused lookup table hides one bit of the bit stream. Then all the control logic blocks which hides data are routed with all the original control blocks. This method is refined in [9]. The signature is modified before being embedded in the control blocks so as to mimic the property of the existing design.

In [10], the author tries to partition a large signature into small sections and uniformly embeds these small pieces of signature in the design. The method ensures higher levels of robustness because it is difficulty to remove all small watermarks using pattern matching and removal techniques.



## 2.2.5 Standard Cell Place and Route

In [1], two methods are proposed. The first method places constraints on the physical location of standard cells. These constraints can be easily coded into a signature. Thus, the signature is embedded in the layout by these constraints. The second method put specific constraints on the realization of detailed routing. The constraints usually involve wire width, spacing and the choices of topological routing.



# Chapter 3 Background of the Proposed Method

In this chapter, we introduce our method and make some explanation of it. We also discuss about the detection of the signature in gate level and physical level in this chapter.

## 3.1 Basic Definitions

This section introduces some general definitions of finite state machines.

**Definition 1 :** A Mealy-type FSM is defined as  $M = ( \Sigma, \Delta, Q, q_0, \delta, \lambda )$ , where  $\Sigma$  is a finite set of input symbols,  $\Delta$  is a finite set of output symbols,  $Q$  is a finite set of states,  $q_0 \in Q$  is the initial (reset) state,  $\delta(q, a) : Q \times \Sigma \rightarrow Q$  is the transition function, and  $\lambda(q, a) : Q \times \Sigma \rightarrow \Delta$  is the output function.

**Definition 2 :** The output of a sequence  $\alpha = (a_1, \dots, a_k)$  applied to states  $q$  denoted by  $\lambda(q, \alpha)$  represents the output of the FSM after a sequence of inputs  $(a_1, \dots, a_k)$ , is applied in state  $q$ . The output of such a sequence is defined to be  $\lambda(q, \alpha) = \lambda( \delta( \delta(\dots \delta(q, a_1)\dots), a_{k-1} ), a_k )$ .

**Definition 3 :** The destination state of a sequence  $\alpha = (a_1, \dots, a_k)$  denoted by  $\delta(q, \alpha)$  represents the final state reached by an FSM after a sequence of inputs  $(a_1, \dots, a_k)$  and is applied in state  $q$ . This state is defined as  $\delta(q, \alpha) = \delta( \delta(\dots \delta( \delta(q, a_1), a_2), \dots ), a_k )$ .

**Definition 4 :** State  $q_i$  and  $q_j$  are equivalent iff  $\lambda(q_i, \alpha) = \lambda(q_j, \alpha)$  for every sequence  $\alpha$ . Two FSMs  $M$  and  $M'$  are equivalent iff their reset states are equivalent.

**Definition 5 :** FSMs  $M = ( \Sigma, \Delta, Q, q_0, \delta, \lambda )$  and  $M' = ( \Sigma, \Delta, Q', q_0', \delta', \lambda' )$  are equivalent iff  $\lambda(q_0, \alpha) = \lambda'(q_0', \alpha)$  for every input sequence  $\alpha$ .

## 3.2 Motivation and Basic Concept

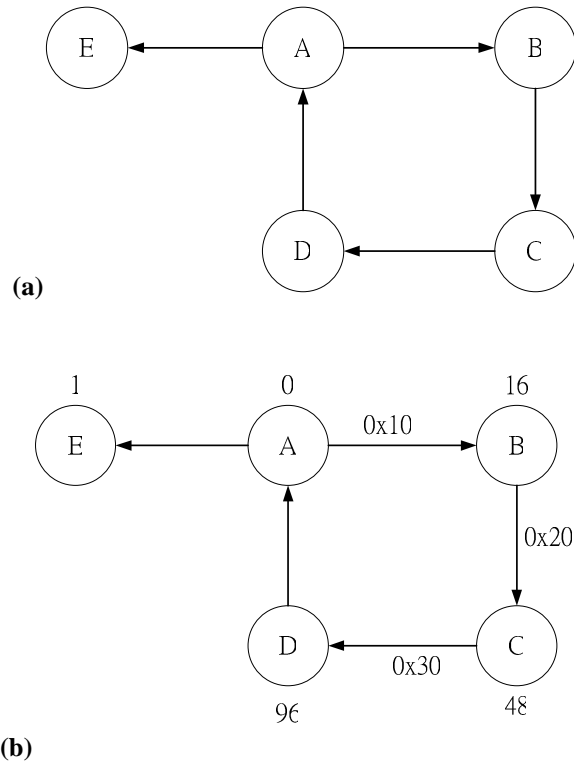
Based on the methods discussed in chapter 2, we can find that all IP protection methods have some disadvantages. In HDL level, there is no way to hide information.

Although some methods can protect IP in gate level or physical level, they can only protect the IP in the specified level. Some other methods can extend the protection from behavioral level to gate level by using register assignment or other approaches, but the protection will be destroyed when the IP is mapped to lower level.

Thus, we want to find a new IP protection method that has two main properties : (1) the method is used in behavioral level since behavioral level provides better reusability (2) the method can provide thorough protection from behavioral level to physical level. Besides, the method must satisfy the requirements mentioned in Section 3.1.

The main idea of our method is to embed the author signature in the difference of state number in a FSM. In this way, the first property is satisfied because the FSM description is in behavioral level. The state number will not change as long as the CAD tool we use does not do any optimization about state numbers. Thus, the author signature we embedded can be extracted in behavioral level, gate level, and physical level.

Figure 3-1 uses a simple example to explain the idea of our method. Figure 3-1(a) is the original FSM. The signature we want to embed is 0x10, 0x20, 0x30. First, we choose an arbitrary path from the reset state a and the path must contains three edges for embedding three number of the signature. Here, we choose A->B->C->D. Then we must find a set of state numbers such that state number of A and state number of B are differenced by 0x10, state number of B and state number of C are differenced by 0x20, state number of C and state number of D are differenced by 0x30. Figure 3-1(b) shows a possible state encoding of the FSM using our method.



**Figure 3-1 An example of state difference encoding (a) the original STG of FSM (b) the STG after applying state difference encoding to hide 0x10, 0x20, 0x30**

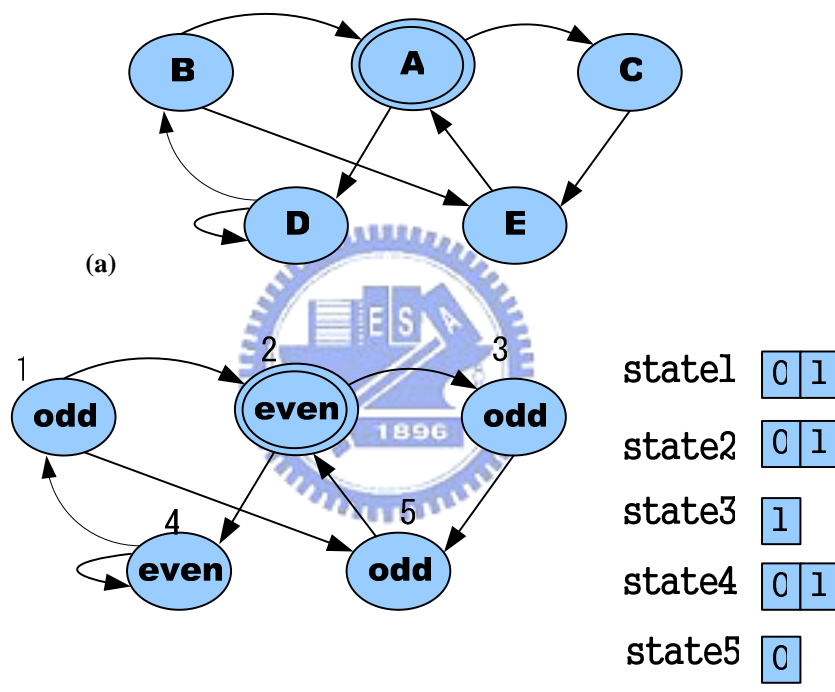
With very low probability, it is possible that we can find another meaningful signature from the FSM which a signature has been embedded in. To decrease the probability of this situation, we can first encrypt the author signature with some encryption algorithms like RSA, and then we embed the encrypted signature into the FSM. In this way, it is almost impossible to find another signature that is meaningful when decrypted from the modified FSM.

### 3.3 Compare with Even/Odd Encoding

There is an easier way to embed the author signature into a FSM by using even/odd encoding which is similar to the method used in [6]. The states will first be assigned an order from 1 to the number of states. Then, we hide the first bit in the 1<sup>st</sup> state using the way that if the bit is 0, we choose a transition that the start state is the 1<sup>st</sup> state and encode the destination state as even. If the bit is 1, encode the destination

state as odd. Repeat this step until all bits have been hidden in the FSM.

There is a serious problem of the even/odd encoding. Consider the example shown in Figure 3-2. This is a result of applying even/odd encoding. As we can see, state1, state2, and state4 can hide arbitrary data because they have both transitions that the destination state are odd and even. For 3 bit data, only (000), (010), (011) can not be hidden in the FSM. 5 out of 8 combinations are possible, and thus even/odd encoding can not be used in IP protection.



**Figure 3-2 An example of even/odd encoding (a) the original FSM (b) a possible solution of even/odd encoding**

Using state difference encoding, a path “A->C->E->A->D->D” will only present the sequence of code |A-C|, |C-E|, |E-A|, |A-D|, |D-D|. Besides, for each state, the code can be hidden is restricted to some specific values. Hence, fake signature is difficult to be created.

### 3.4 The Detection of the Signature

Our state difference encoding uses two techniques to detect the signature we hide. In behavioral and gate level, we only need to simulate the given FSM and observe the value of the state registers. In physical level, we take advantages of the scan-chain. Using scan-chain, we can set the FSM to a specific state and run one step. Then we dump the state value to observe if there is any signature hidden in the given FSM.



# Chapter 4 State Difference Encoding

In this chapter, we use a more formal way to introduce our state difference encoding method.

## 4.1 Inputs and Outputs

The inputs of our method are listed as follows :

1. The FSM that need protection. The definition of FSM is already discussed in section 3.1.

2. The predefined path,  $path = \{ state_0, state_1, \dots, state_n \}$ . A predefined path is a path that the information will hide in. This path can be generated by a path finder or provided by the user. If the path is given by the user, the associated valid vector should be given, too.

3. The information sequence,  $info = \{ info_1, info_2, \dots, info_m \}$ . Information sequence means the ASCII code sequence that we want to embed to the FSM. This is equal to the author signature mentioned before. The variable  $m$  is determined by the path length and the total length of the information sequence. If the length of information sequence is  $\alpha$ , then the variable  $m$  is  $\alpha / m$ . The information sequence can firstly be encrypted by an arbitrary encryption algorithm or just plain text of ASCII codes. The encryption process is recommended since this process decreases the probability of fake meaningful signature found in the FSM which we have already hidden a signature in.

The outputs of our method are :

1. The modified FSM. The function of the modified FSM will be the same as the original FSM. Only the state encoding will be different since we use state

difference encoding to hide information in the FSM.

2. The valid vector,  $\text{valid\_vector} = \{ \text{valid\_vector}_1, \text{valid\_vector}_2, \dots, \text{valid\_vector}_{n-1} \}$ . We will encounter a situation which usually results in no feasible state encoding when the original FSM has a feedback loop. Thus, we introduce the valid vector to prevent this situation. The detail of valid vector will be discussed in section 4.3. We shall note that the variable  $n$  is the same as the variable used in predefined path. This means the length of valid vector will be one less than the predefined path.

## 4.2 The Encoding Process

The encoding process of our method is divided into three steps, as shown in Figure 4-1.

- **Step1 , Path Finder** : the goal of path finder is to find a path to hide information from the FSM ( in kiss2 format, which is shown in Figure4-2 ) given by user and pass the path, valid vector, and the information to INLP solver.
- **Step2 , INLP Solver** : INLP solver tries to find a legal state encoding that embeds the information. Sometimes, INLP solver can not find a legal state encoding of the path passed by path finder. In this case, we go back to step 1 and find another path. Repeat this step until a legal state encoding is found.
- **Step3 , HDL Generator** : This is the easiest part of the state difference encoding process. The HDL generator takes the FSM given by user and the state code found by INLP solver to generate a new FSM in gate level description that embeds the information. The user can choose that the modified FSM is in Verilog form or VHDL form.



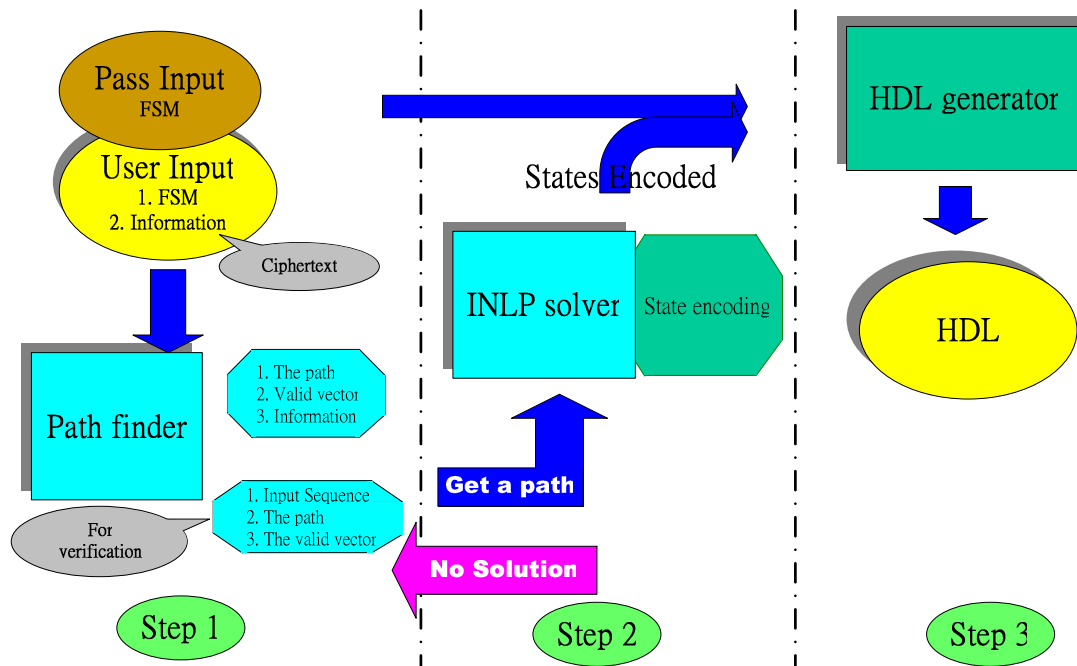


Figure 4-1 The encoding procedure of state difference encoding

```

.i 3 => the number of inputs
.o 3 => the number of outputs
.p 10 => the number of edges
.s 5 => the number of states
.r st0 => the reset state, it must one of the states

-10 st0 st1 001 the description of a single transition
: it means when the FSM is in state st0
: and input is -10 the FSM will move to
: state st1 and put 001 on output

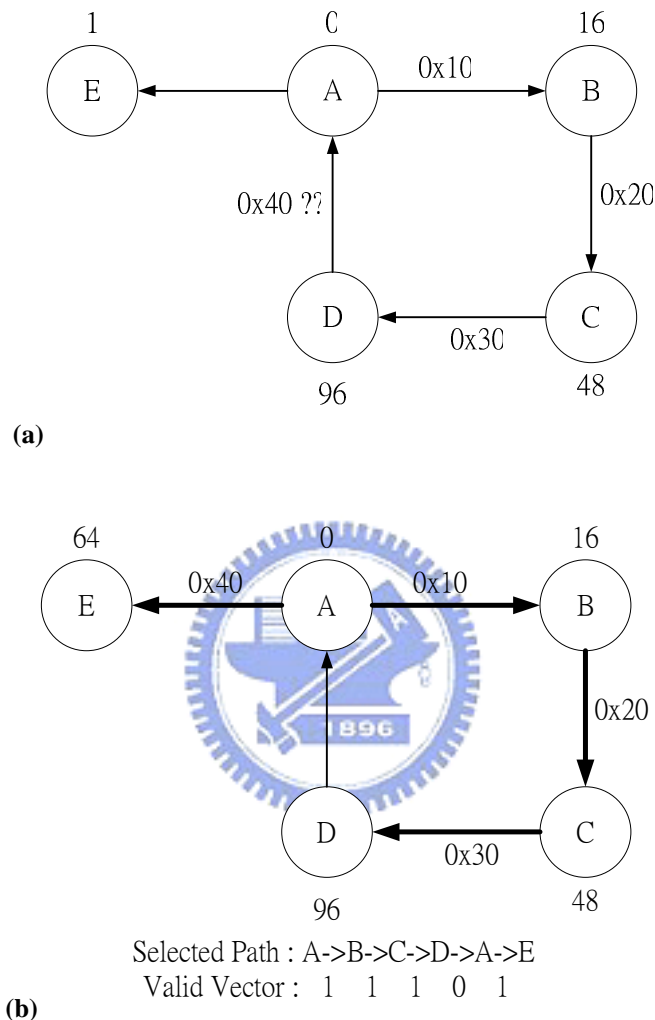
```

Figure 4-2 The kiss2 file format

### 4.3 Valid Vector

In ideal case, a path with  $n$  edges can hide  $n$  information in it using our method. But in fact, most FSMs have loops or self-loops in order to reset the FSM or wait for the arriving of some signals. This restricts us in finding a long path to hide the information. Consider the case illustrated in Figure 4-3(a). We want to hide the information sequence '0x10, 0x20, 0x30, 0x40' in the path "A->B->C->D->A". It can be found that once three information have been hidden in edge "A->B, B->C, C->D",

the state encoding of state A, B, C, and D are determined and there is almost no room for adjusting the encoding to match the fourth piece of information, “0x40”. As a result, the edge “D->A” can not be used to hide information.



**Figure 4-3 An example of valid vector (a) the FSM can only hide three information without valid vector (b) with valid vector, A->E can be used to hide the fourth information**

If we can make some of the edges hiding information and others not, we can hide one more information in the FSM of Figure4-3(a) by hiding the fourth information in the edge A->E and disregarding the edge “D->A” in the path “A->B->C->D->A->E”. The choice of edges is what we called “valid vector”. A valid vector is a bit sequence of 0 and 1 incorporated with the predefined path. If a bit of valid vector is 1, the edge is used for hiding information. Otherwise, if the bit of valid vector is 0, we will omit

the edge.

The valid vector is not only useful for hiding longer information and increase the utility rate of the edges. It is helpful for distribute the hidden information to the whole path. Thus, it is harder for the attackers to find out what information we hide and the protection of our method will be stronger.

#### 4.4 Finding the path

Although the path which is used to hide information can be provided by the user, the path finder is still required for the case that the user does not provide the path.

There are some problems we encountered in implementing the path finder. The first problem is the restriction in path selecting due to a data item “00” in information sequence. It is possible that the information sequence contains a data item “00”.

Because we use the state difference encoding, the only choice of hiding “00” is to find a state with a self loop transition and use this transition to hide the “00”. Without the “00” condition, we can simplify the path finding process as finding a path in FSM such that the number of “1” in valid vector equals to the number of data item in the information sequence.

In order to solve the problem, we use the data item “00” as delimiter to divide the information sequence into several sub sequence. Then, each sub sequence is a simple path finding problem. There are two requirements of the simple path finding problem :

1. The end node of the path must contain a self-loop in order to hide the delimiter “00”.
2. The efficiency length has to be larger than the number of items in the sub sequence.

As shown in Figure 4-4, the information sequence 13, 00, 14, 55, 00 is divided into

two sub sequences 13, 00 and 14, 55, 00. Then we find two paths to hide these sub sequences respectively.

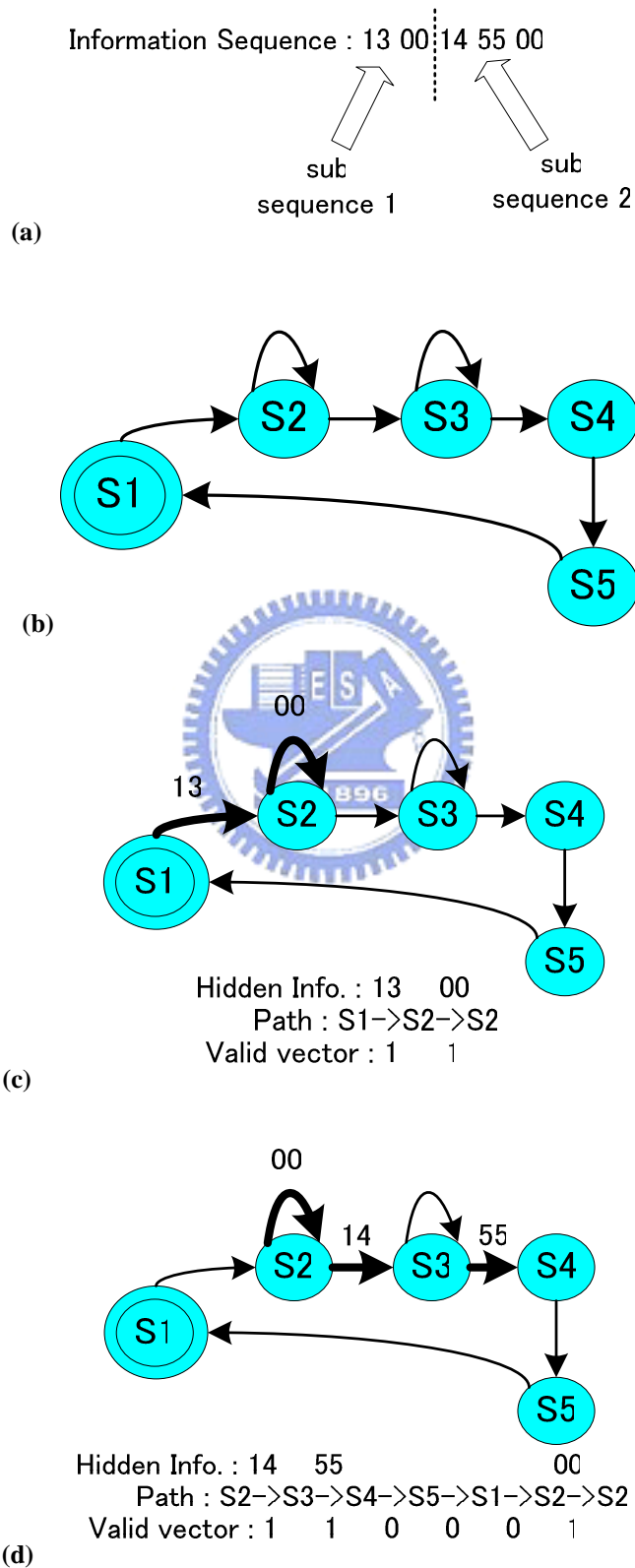


Figure 4-4 An example of the approach to solve the data item “00” in the information sequence.

**(a) The information sequence is divided into two parts. (b) the sample FSM. (c) a solution of sub sequence 1. (d) a solution of sub sequence 2.**

The second problem we encountered is that we want to find all possible paths in a given FSM but the original model can not satisfy our requirement due to valid vector. Consider the FSM in Figure 4-4(b) and the path “S1->S2->S3->S4->S5->S1->S2->S2->S3->S4->S5->S1->S2->S2” with valid vector ( 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1 ). The path goes through the whole loop twice. The edge “S2->S3” in first loop hides information, but the same edge in the second loop does not. However, using traditional path finding algorithm can not see the difference between the first loop and second loop. Thus, we need to extend the original FSM with shadow states. For each state S, we create a associated shadow state S'. The state S' is the same as the original state including incoming edges, outgoing edges, and self-loop edges. For every edge with state S as destination state, we create a new edge from the start state to state S'. In the same way, for every edge with state S as start state, we create a new edge from S' to the end state. The modified FSM of Figure 4-4(b) is shown in Figure 4-5. The problem is solved by using the new FSM to find a path. If one edge in the new path has a shadow state as destination state, we define the corresponding valid vector bit is '0'. In other words, there will be no data hidden in this edge. Otherwise, if the edge has a normal state as destination state, we will hide a data in this edge. By using this modified FSM, we can use normal path finding algorithm to find all paths. For example, the path discussed above can be represented by “S1->S2'->S3->S4'->S5'->S1'->S2'->S2->S3'->S4->S5->S1'->S2'->S2”.

Although now we can enumerate every path in the STG, it is very inefficient that finding a path from the beginning every time. Besides, this approach may lead to the degradation of protection. Thus, we use a random approach to find a path. If M is the FSM that we want to protect and M has n edges. Firstly, we generate n / 8 ( this

number will increase 1 when n is not divided by 8 ) random numbers such that every bit of random number is mapped to one edge. The edges with a '1' in its mapping random number bit are selected to hide information. If the number of one's is not equal to the number of information items we want to hide, just re-generate the random number again. In order to make the edge sequence more randomly, we randomly pick one edge up to form another sequence in order. Then, we connect these edges together to form a path by applying the all pair shortest algorithm to the edges to find the path from one edge's destination state to another edge's start state. If all edges can be connected, we get a path for hiding information and we pass it to next step. Sometimes, there is no path between two edges and all pair shortest path algorithm will fail. In this case, we go back to the random number generation step and do the whole procedure again. The pseudo code of path finder is shown in Figure 4-6.

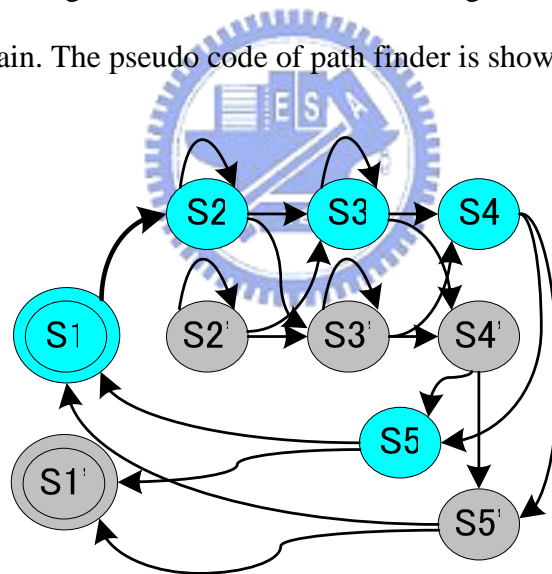


Figure 4-5 The FSM of Figure 4-4(b) with shadow states

```

Path Finder( number of edges n , number of data items d )
{
Repeat:
do
{
if( n divided by 8 )
generate n / 8 random numbers
else
generate n / 8 + 1 random numbers

count the number of '1' in random numbers
}
while( the number of '1' != d )

for( i = 1 to d )
{
pick a random edge j from 1 to d such that the jth selected edge
will be in ith position of new order
}

for( i = 1 to d )
{
use all pair shortest path algorithm to find the path from the
destination state of ith edge to the start state of i+1th edge

if( there is no path between ith edge and i+1th edge )
goto Repeat
}
}

```

**Figure 4-6 The pseudo code of path finder**

## 4.5 Formulating the encoding requirement into INLP

In order to acquire a state encoding, we formulate the problem as an integer non-linear problem (INLP), and then we use a tool called LINGO to solve the INLP.

We use two constraint and a cost function to create the INLP :

1. **Unique Constraint.** Every state must have different state encoding.
2. **Feasible Constraint.** The absolute state difference of start state and end state of the  $i^{\text{th}}$  not excluded edge should be equal to the  $i^{\text{th}}$  hidden data for

every  $i$ .

3. **Cost Function.** We can perform variant optimization for encoding. In general, the cost function is set in order to acquire a set of minimum state encoding.

Here we use Figure 4-1(b) as an example. The path we want to hide information is  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow E$  excluded the edge  $D \rightarrow A$ . The hidden information sequence is  $0x10, 0x20, 0x30, \text{ and } 0x40$ . Figure 4-7 shows the unique constraint, feasible constraint and the cost function of Figure 4-1(b).

## Unique Constraint

$$A \neq B \quad B \neq C \quad C \neq D$$

$$A \neq C \quad B \neq D \quad C \neq E$$

$$A \neq D \quad B \neq E$$

$$A \neq E$$

## Feasible Constraint

$$|A - B| = 0x10$$

$$|B - C| = 0x20$$

$$|C - D| = 0x30$$

$$|A - E| = 0x40$$

## Cost Function

$$\text{Cost} = A + B + C + D + E$$

**Figure 4-7 The constraints of Figure4-1(b) for INLP**

In implementing the INLP solver, we simply list all constraints in a command, create the environment of LINGO, and call LINGO to solve the INLP. For a path given by path finder, we generate the feasible constraints such that the start state and destination state of selected edges are equal to the  $i$ th data item we want to hide. We also generate the unique constraints such that the encoding of every state must not be equal to the encoding of another state since every state must have its own



unique encoding in FSM. Then, we generate the cost function to minimize the state encoding. There are two kinds of cost function we can use. One is to minimize the sum of every state encoding and the other is to minimize the maxima state encoding of a single state. These two cost functions results in a similar state encoding in LINGO, so we do not need to bother which one should use. The pseudo code of INLP solver is shown in Figure 4-8.

```

INLP solver( a path p given by path finder , FSM M, information sequence i )
{
    for( j = 1 to number of information data items )
    {
        generate the constraint
        | the start state of jth selected edge in p -
        the destination state of jth selected edge in p | = jth data item in I
    }

    for( every two different states I, J in M )
    {
        generate the constraint
        state code of I must not equal to state code of J
    }

    generate the cost function
    minimize the sum of every state encoding of M
}

```

**Figure 4-8 The pseudo code of INLP solver**

## 4.6 Generating the gate-level net-list

With the state code generated by LINGO and the FSM description given by the user, we shall create a new FSM in Verilog form or VHDL form with the information sequence hidden between states.

Because some EDA tools will perform optimization like state minimization on the circuit when it found its input is a FSM, the information sequence we hide in the FSM will be destroyed. In order to prevent the FSM from optimization, we tear every

bit of state variable apart and store them in separate latches. For every bit of next function and output function, consider every situation which makes this bit become 1, and combine all situations by a big OR gate. Consider the case of Figure 4-9. The FSM has three states, one input, and one output. For every transition, we produce a product term to describe the happening of this situation. The happening condition of the left transition in Figure 4-9 is when the FSM is in state '101' and input equals 1.

The two product terms of the transition in Figure 4-9 should be

$$p0 = \text{state [2] and not state [1] and state [0] and input}$$

$$p1 = \text{not state [2] and state [1] and state [0] and not input}$$

With these product terms, we can generate every bit of next state by adding the corresponding product term to the bit of state variable. For the left transition of Figure 4-6, the destination state is "011". Then we add the corresponding product term  $p0$  to bit 1 and bit 0 of next state. The destination state of right transition of Figure 4-9 is "101", thus we add  $p1$  to bit 0 and bit 2 of next state. The next state of Figure 4-9 should be

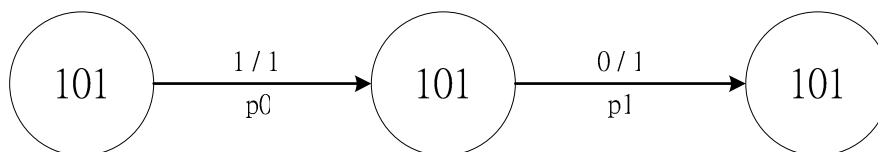
$$\text{next\_state[2]} = p1$$

$$\text{next\_state[1]} = p0 \text{ or } p1$$

$$\text{next\_state[0]} = p0$$

Finally, we generate the output of this FSM. We simply combine all product terms which make the output become 1 by an OR gate. The output of Figure 4-9 should be

$$\text{output} = p0 \text{ or } p1$$



**Figure 4-9 A simple FSM**

The final FSM of Figure 4-10 in Verilog form is shown in Figure 4-9.

```

module FSM( clk, clr, scaninput, scanmode, I, O );
    input clk, clr, scaninput, scanmode, I;
    output O;
    wire [2:0] ST;
    wire [2:0] N_ST;
    wire [1:0] p;

    assign p[0] = ST[2] & ~ST[1] & ST[0] & I;
    assign p[1] = ~ST[2] & ST[1] & ST[0] & ~I;

    assign N_ST[2] = p1;
    assign N_ST[1] = p0 | p1;
    assign N_ST[0] = p0;

    assign O = p0 | p1;

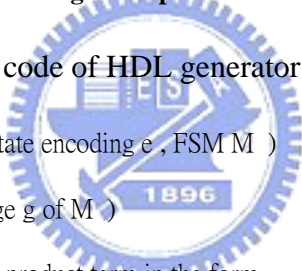
    ScanDFF d0( clk, clr, N_ST[0], scaninput, scanmode, ST[0], 1'b1 );
    ScanDFF d1( clk, clr, N_ST[1], ST[0], scanmode, ST[1], 1'b0 );
    ScanDFF d2( clk, clr, N_ST[2], ST[1], scanmode, ST[2], 1'b1 );

endmodule

```

**Figure 4-10 The Verilog description of the FSM in Figure 4-9**

Figure 4-11 is the pseudo code of HDL generator.



```

HDL generator( state encoding e, FSM M )
{
    for( every edge g of M )
    {
        generate a product term in the form
        p( g ) = AND of every bit of the start state and input
    }

    for( every state s of M )
    {
        for( every edge g of M )
        {
            if( the destination state of g == s )
            {
                for( every bit b of next function )
                {
                    add the product term p( g ) to b
                }
            }
        }
    }

    generate the cost function
    sum of every state encoding in M
}

```

**Figure 4-11 The pseudo code of HDL generator**

## 4.7 The Detecting Process

How do we verify whether a given FSM hides our information? First we shall prepare the modified FSM we want to detect and the path, valid vector, the information sequence of our signature. Then, for each pair of input state and input, we fed them into the suspicious FSM and observe the destination state of each step. If all destination states are equal to the destination state of corresponding edge in our path, the designer of this FSM is suspicious for copy our design.

In gate level, this procedure can be done by simply simulating the suspicious FSM using any EDA tool with simulator and observe the state transition sequence. But how do we detect the design that already been made in a chip ? In physical level, it is harder to find out the current state of a FSM. Fortunately, most design of circuit contains scan-chain or JTAG for debugging the circuit. We can use the scan-chain to scan in the start state of the path and let the FSM run one step. Then we scan out the current state of the FSM and compare it with the path. What if the target circuit does not have scan-chain ? In this case, we use a more difficult method to find the current state. In a traditional FSM, the state variable is stored in a latch or a register and the latches can easily be identified. We can make a copy of the combination part of the FSM. Then we can feed the state and input to the copy circuit to observe the next state that it sends to the latches. But how do we know the order of latches ? We can firstly use an arbitrary order to feed the start state of a specific edge and observe the destination state. If the destination state does not match, try another order until we found a match or all possible orders have been tried.

## 4.8 A Complete Example

In this section, we use the control circuit of an UART receiver to show the

complete procedure of our IP protection method. We divide the whole UART receiver into control circuit, shift register, and latch for convenience and we implement the control circuit by FSM in order to hide an information sequence in it. The architecture of UART receiver is shown in Figure 4-12. In this architecture, we use eight times faster clock frequency to ensure the correctness of data sampling. The most significant bit of three shift registers of this architecture is '1' in normal situation and other bits are all '0'. The shift register 1 triggers the sampling action in the fourth frame. The shift register 2 counts 8 clocks to determine the end of data receiving. The shift register 3 controls the eight latches to hold input data. When eight data is arrived, the UART receiver will send a signal RCV\_REQ to notify the environment the arriving of the input data.

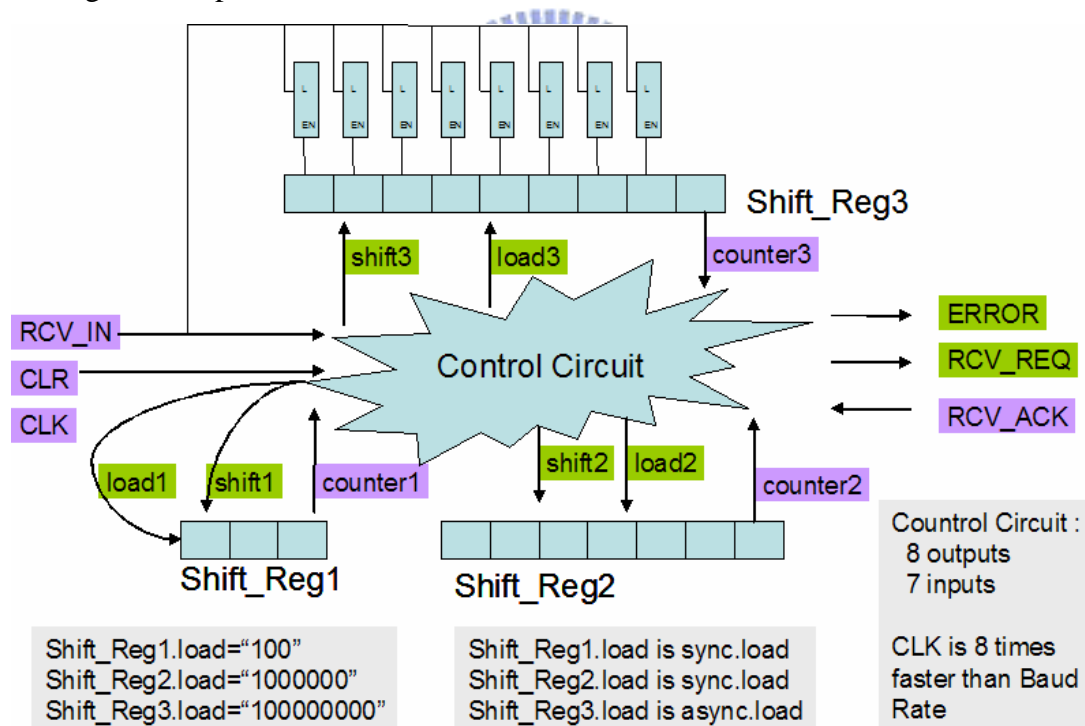


Figure 4-12 The architecture of UART receiver

Figure 4-13 is the STG of the control part of UART receiver. The five inputs in Figure 4-13 are represented for RCV\_IN, RCV\_ACK, counter1, counter2, counter3 respectively. The eight outputs are shift1, load1, shift2, load2, shift3, load3,

RCV\_REQ, ERROR respectively. The reset state is state 8 and the error state is state

7. The information sequence we use here is the ASCII code of “NCTU IPP”.

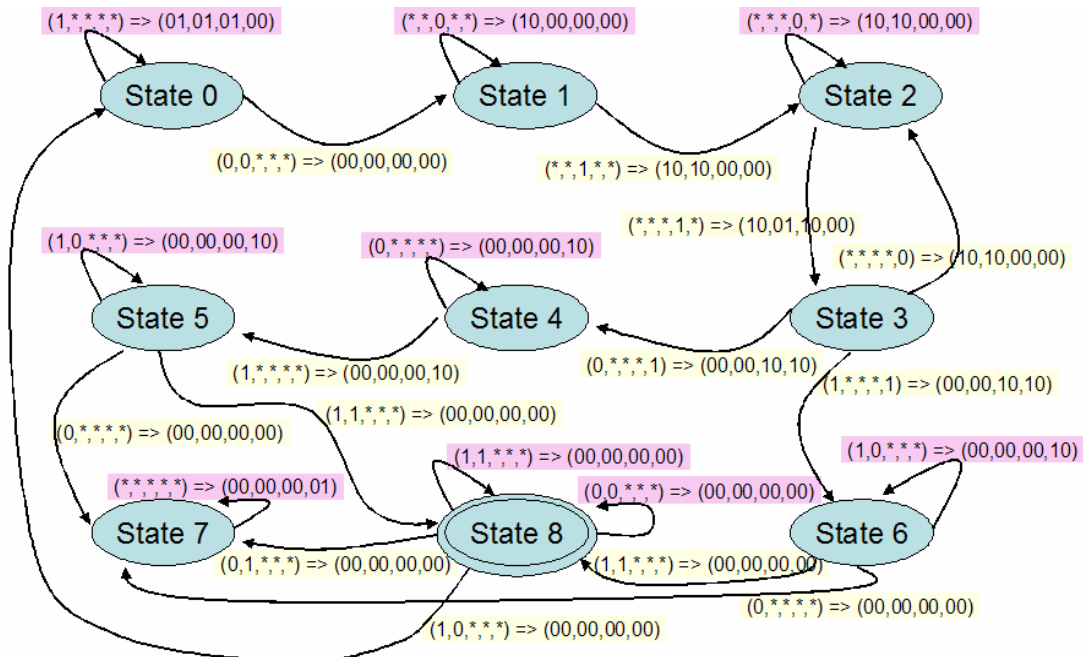


Figure 4-13 The STG of UART receiver

In the first place, we use a tool “crypt” in SUN Solaris operating system to encrypt the information sequence with the key “1234”. The whole process is shown in Figure4-14. The encrypted data of “NCTU IPP” is 0x40C8DD855F3BD908. Though we can use more powerful encryption tool like 3DES, RSA, AES to encrypt the information sequence, we use crypt here due to simplicity.

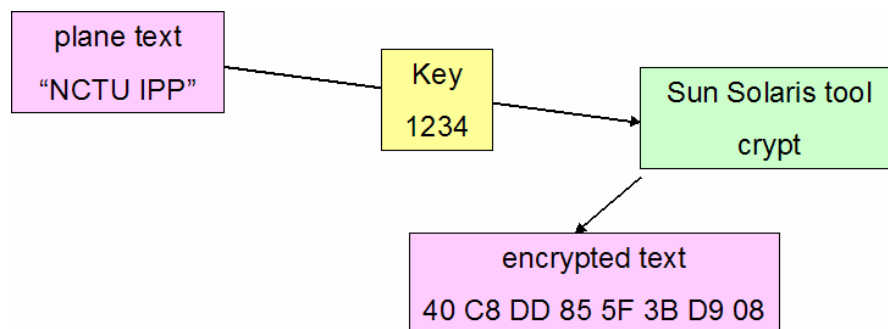


Figure 4-14 The encryption process of information sequence

In order to add the encrypted text into the UART receiver controller, we divide the encrypted data into eight 8-bits data and we find a path which is long enough to

hide data. The path we found is state8 -> state0 -> state1 -> state2 -> state3 -> state4 -> state5 -> state8 -> state0 -> state1 -> state2 -> state3 -> state6 -> state8 -> state7 and the corresponding valid vector is 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1. This means the first state8 -> state0 edge, the first state0 -> state1 edge, the first state1 -> state2 edge, the first state2 -> state3 edge, the state3 -> state4 edge, the state4 -> state5 edge, the state3 -> state6 edge, and the state8 -> state7 edge are used to hide information. Then, generate the three constraints of UART receiver and call LINGO to solve it. Figure 4-15 shows the three constraints of UART receiver used to acquire the state encoding .

### Unique Constraint

```

State0 != State1 State1 != State2 State2 != State3 State3 != State4 State4 != State5 State5 != State6 State6 != State7 State7 != State8
State0 != State2 State1 != State3 State2 != State4 State3 != State5 State4 != State6 State5 != State7 State6 != State8
State0 != State3 State1 != State4 State2 != State5 State3 != State6 State4 != State7 State5 != State8
State0 != State4 State1 != State5 State2 != State6 State3 != State7 State4 != State8
State0 != State5 State1 != State6 State2 != State7 State3 != State8
State0 != State6 State1 != State7 State2 != State8
State0 != State7 State1 != State8
State0 != State8

```



### Feasible Constraint

```

| State8 - State0 | = 0x4C      | State3 - State4 | = 0x5F
| State0 - State1 | = 0xC8      | State4 - State5 | = 0x3E
| State1 - State2 | = 0xDE      | State3 - State6 | = 0xD9
| State2 - State3 | = 085       | State7 - State8 | = 0x08

```

### Cost Function

Cost = State0 + State1 + State2 + State3 + State4 + State5 + State6 + State7 + State8

**Figure 4-15 The three constraints of UART receiver**

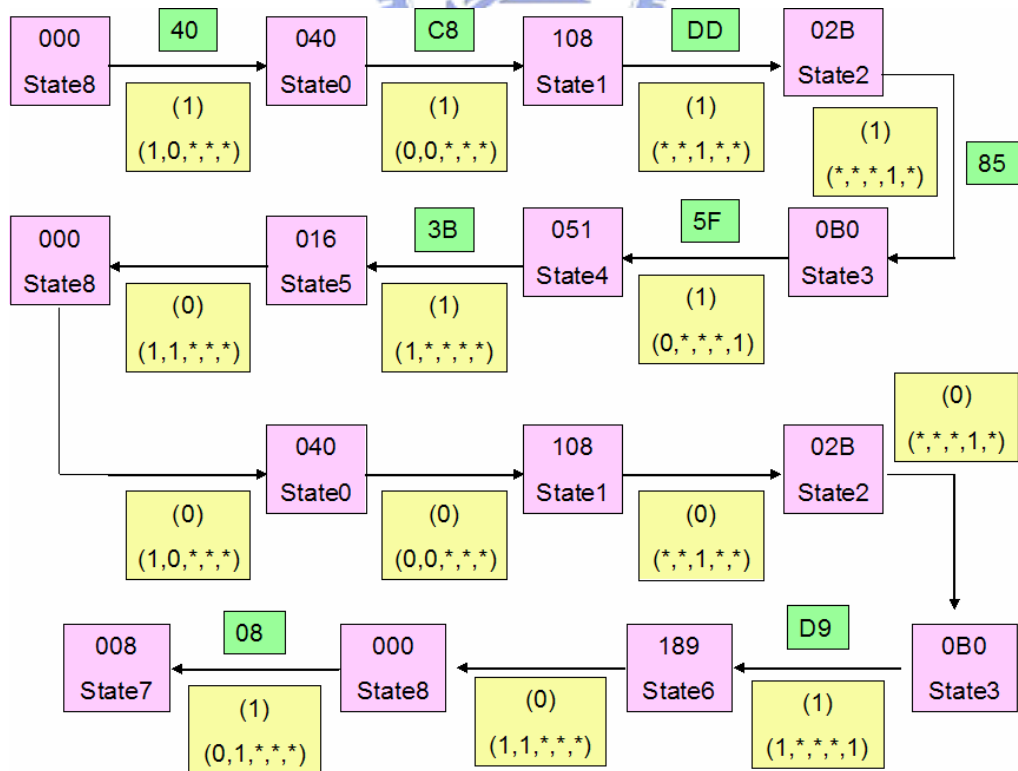
Then use LINGO to solve the INLP and we get a solution shown in Table 4-1.

State 0	040
State 1	108
State 2	02B
State 3	0B0
State 4	051

State 5	016
State 6	189
State 7	008
State 8	000

**Table 4-1 The state encoding of UART receiver**

In order to detect signature through the path, we need the information sequence we hide and valid vector. We also need the input of each state. Figure 4-16 shows all the data we need to find the hidden data. For example, the edge state8 -> state0 may possibly hide the data 0x40. We set the current state of FSM to state8 ( which is 000 is this case ) and give the input (1,1,\*,\*,\*) to the FSM. Then make the FSM run one step and observe the current state. Repeat this step until the whole path is traversed. If all state differences with valid vector '1' are the same, the FSM may be copied from our design.



**Figure 4-16 The state encoding, valid vector, hidden information, and inputs used by detecting**



process

We implement the UART receiver in Xilinx FPGA in order to make sure the UART receiver work correctly. The whole architecture is shown in Figure 4-17. We connect the FPGA to a PC by serial connection and use communication software ‘TeraTerm’ to send data to the FPGA. After receiving the data, the UART receiver will show the data in seven-segment display. In order to extract the information sequence we hide, we design a simple FSM to feed the current state and inputs which are stored in a separated ROM into the UART receiver. The simple FSM will scan in the start state of a edge in our path, control the UART receiver to run one step and scan out the current state. Finally the simple FSM shows the state number in two seven-segment displays. The difference calculation is left for users.

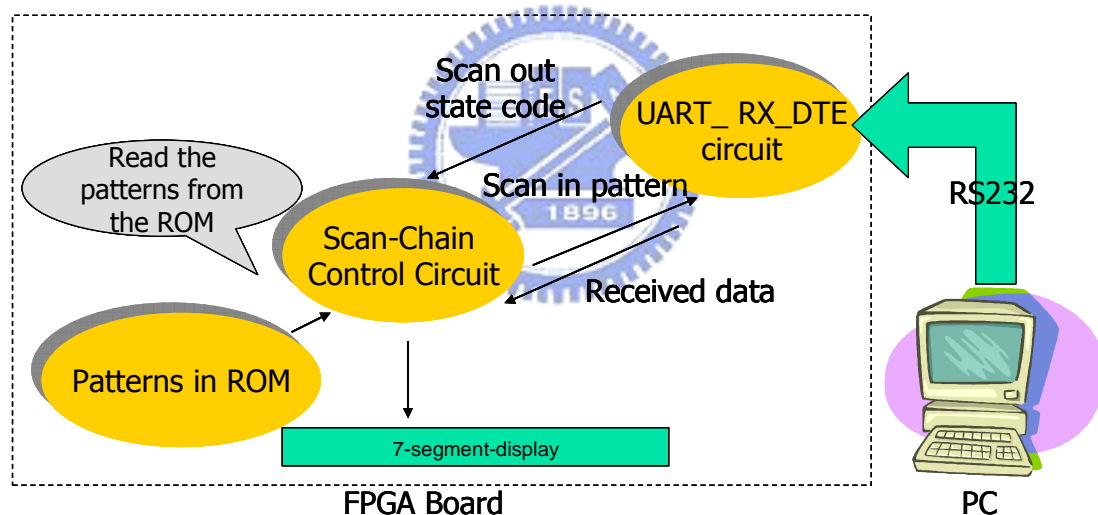


Figure 4-17 The architecture of UART receiver with verification circuit

## 4.9 Evaluation of State Difference Encoding

In this section, we want to discuss about the robustness of our state difference encoding and the overhead we made.

## 4.9.1 The Robustness Evaluation

An attacker may try to attack the circuit protected by our method in three ways. The first way is trying to find out the information sequence we embedded and removing them. The second way is to hide another information sequence in the FSM or find another meaningful information sequence from the FSM and claim that the FSM is designed by him. The third way is to randomly pick up the edge used to hide information and the state difference will exactly the same as ours.

Using our method, it is very hard to attack the FSM by first approach. In order to remove our information sequence, the attacker must encode all states again. But only changing the state encoding will cause the output of the FSM malfunction. Unless the attacker corrects the output which matches its state encoding, the FSM will not work correctly. To remove our information sequence in such a way, the attacker has to change almost the whole FSM which takes a great amount of time.

What if the attacker uses the second approach ? Because we use the state difference to hide information, the data item can be used is already specified and very restricted. The attacker can only use these restricted data to form his information sequence. This will greatly decrease the probability that a fake information sequence is found from our protected FSM. Although the attacker may find a small but meaningful information sequence, we can claim that our information sequence is longer and thus provides more authorship. We can further combine our method with some powerful encryption algorithms. By using encryption, the attacker has to find an encrypted information sequence from our FSM using restricted data and the decrypted information of this information sequence has to be meaningful. The probability of this situation is much lower.

Let we consider the third way. If a FSM has  $m$  states and  $n$  edges and assumes

we want to hide 16 data items in the FSM,  $m$  is greater than 16 and any two states can be used to hide information. The probability that the edges picked up randomly will be equal to the edges we select is  $1/(n)(n-1)(n-2)\dots(n-15)$ . If  $n$  is 100, then the probability will be  $1/2.816 \times 10^{31}$ . Thus, it is almost impossible to happen to pick up the same edges to hide information.

#### 4.9.2 The Overhead Evaluation

Because we use the difference of states to hide information, we do not have any overhead of the FSM ideally. In fact, our method will only make the area of the FSM increase in one case. Take Figure 4-13 and Table 4-1 for example, the FSM has only nine states and can be encoded using four bits. We use nine bits to encode them because each data item of hidden information sequence is too large to hide in the state difference of four bits encoding. If the FSM has more states, we may divide the information sequence into more data items so each data item is smaller and can be hide in the FSM with less state variable bits.

Besides to increase the bits of state variable, we can use an alternative approach to hide large signature in a small FSM. If the target FSM is really small that the states and edges we can use is not enough to maintain the robustness of authorship proof, we add additional redundant states to hide the information sequence. With these redundant states, we can divide the information sequence into more small pieces, thus provide more protection.

On the other hand, the computation overhead of our method is bigger than other methods. Firstly, we must find a path in the STG of original FSM. Then we solve the INLP made from the path and the information sequence. Usually, The INLP solver can not find a feasible solution. Then we call the path finder to find another path and run INLP solver again. This situation usually loops ten or more times before finding a

legal state encoding. Although we spend more time in the computation of state encoding, we can hide an information sequence in the FSM with less area overhead.



## Chapter 5 Experimental Results

In this chapter, we applied our IP protection method to the benchmark ISCAS'91 to evaluate the overhead of our method. The ISCAS'91 benchmark includes 53 different FSMs which all are in kiss2 format. The size of the circuits in ISCAS'91 varies widely. The original information used to be hidden in the FSMs is "DESIGN BY SWCHEN". Firstly, we encrypt the plain text ASCII code to 0xC9A7EC8B9CB77FBDCAA393E2A1E1AE8E by "crypt" with key "1234". As mentioned before, we can divide the information sequence to appropriate data items by situations. In our experiment, the information sequence is divided into 12 cases from 5 items to 16 items and hides them in the FSMs of benchmark respectively. Table 5-1 shows the statistics data of ISCAS'91 benchmark. The columns of Table 5-1 from left to right are the inputs of the circuit, the outputs, the number of edges, the number of states, and the number of minimum state bits required to encode the states.

Circuit	Inputs	Outputs	# of edges	# of states	# of state bits
bbara	4	2	60	10	4
bbsse	7	7	56	16	4
bbtas	2	2	24	6	3
beecount	3	4	28	7	3
cse	7	7	91	16	4
dk14	3	5	56	7	3
dk16	2	3	108	27	5
dk17	2	3	32	8	3
dk27	1	2	14	7	3

dk512	1	3	30	15	4
ex1	9	19	138	20	5
ex2	2	2	72	19	5
ex3	2	2	36	10	4
ex4	6	9	21	14	4
ex5	2	2	32	9	4
ex6	5	8	34	8	3
ex7	2	2	36	10	4
keyb	7	2	170	19	5
kirkman	12	6	370	16	4
lion9	2	1	25	9	4
mark1	5	16	22	15	4
opus	5	6	22	10	4
planet	7	19	115 G	48	6
planet1	7	19	115	48	6
pma	8	8	73	24	5
s1	8	6	107	20	5
s27	4	1	34	6	3
s208	11	2	153	18	5
s298	3	6	1096	218	8
s386	7	7	64	13	4
s420	19	2	137	18	5
s510	19	7	77	47	6
s820	18	19	232	25	5
s832	18	19	245	25	5

s1488	8	19	251	48	6
s1494	8	19	250	48	6
sand	11	9	184	32	5
scf	27	56	166	121	7
shiftreg	1	1	16	8	3
sse	7	7	56	16	4
styr	9	10	166	30	5
tbk	6	3	1569	32	5
tma	7	6	44	20	5
train11	2	1	25	11	4

**Table 5-1 The statistic data of benchmark ISCAS'91**

Table 5-2 shows the required state bits when hiding different number of data items. Note that no matter how many data items are hidden, the total information length is 128 bits. As we can see in Figure 5-2, the required state bits are almost determined by the length of a single data item. In normal case, the more parts 128 bits information is divided, the smaller data length per item and the less required state bits. But sometimes the reverse condition likes 14 items and 15 items of circuit dk16 will happen. This is because we use a random approach to pick up a path for information hiding. Some grids in Table 5-2 are written as N/A because the data items are too many to be hidden in the circuit. This can be solved by adding new states and new edges, but we did not implement the function yet.

Circuit	Ori.	5 items	6 items	7 items	8 items	9 items	10 items
bbara	4	26	23	19	15	14	N/A

bbsse	4	26	22	19	16	15	14
bbtas	3	26	N/A	N/A	N/A	N/A	N/A
beecount	3	26	22	N/A	N/A	N/A	N/A
cse	4	26	22	19	16	15	13
dk14	3	26	22	N/A	N/A	N/A	N/A
dk16	5	26	22	19	16	15	13
dk17	3	26	22	19	N/A	N/A	N/A
dk27	3	26	22	N/A	N/A	N/A	N/A
dk512	4	26	22	19	16	15	14
ex1	5	26	22	19	16	15	13
ex2	5	26	23	19	N/A	N/A	N/A
ex3	4	26	22	19	17	16	N/A
ex4	4	26	22	19	16	15	13
ex5	4	26	22	19	17	N/A	N/A
ex6	3	26	22	19	N/A	N/A	N/A
ex7	4	26	22	N/A	N/A	N/A	N/A
keyb	5	26	22	19	16	15	13
kirkman	4	26	22	19	16	15	14
lion9	4	26	22	19	17	N/A	N/A
mark1	4	26	22	19	16	15	14
opus	4	26	22	19	17	15	14
planet	6	26	22	19	16	15	13
planet1	6	26	22	19	16	15	13
pma	5	26	22	19	16	15	13
s1	5	26	22	19	16	15	13



s27	3	26	N/A	N/A	N/A	N/A	N/A
s208	5	26	22	19	16	15	14
s298	8	26	22	19	16	15	13
s386	4	26	22	19	16	16	14
s420	5	26	22	19	16	15	13
s510	6	26	22	19	16	15	13
s820	5	27	22	19	16	15	13
s832	5	26	22	19	16	15	13
s1488	6	26	22	19	16	15	13
s1494	6	26	22	19	16	15	13
sand	5	26	22	19	16	15	14
scf	7	26	22	19	16	15	13
shiftreg	3	26	22	19	N/A	N/A	N/A
sse	4	26	22	19	16	16	13
styr	5	26	22	19	16	15	13
tbk	5	26	22	19	16	15	13
tma	5	26	22	19	16	15	13
train11	4	26	23	19	16	16	14

Circuit	Ori.	11 items	12 items	13 items	14 items	15 items	16 items
bbara	4	N/A	N/A	N/A	N/A	N/A	N/A
bbsse	4	13	12	N/A	N/A	N/A	N/A
bbtas	3	N/A	N/A	N/A	N/A	N/A	N/A
beecount	3	N/A	N/A	N/A	N/A	N/A	N/A

cse	4	12	11	11	N/A	N/A	N/A
dk14	3	N/A	N/A	N/A	N/A	N/A	N/A
dk16	5	12	11	10	9	10	9
dk17	3	N/A	N/A	N/A	N/A	N/A	N/A
dk27	3	N/A	N/A	N/A	N/A	N/A	N/A
dk512	4	12	11	11	N/A	N/A	N/A
ex1	5	12	11	10	9	10	N/A
ex2	5	N/A	N/A	N/A	N/A	N/A	N/A
ex3	4	N/A	N/A	N/A	N/A	N/A	N/A
ex4	4	13	11	N/A	N/A	N/A	N/A
ex5	4	N/A	N/A	N/A	N/A	N/A	N/A
ex6	3	N/A	N/A	N/A	N/A	N/A	N/A
ex7	4	N/A	N/A	N/A	N/A	N/A	N/A
keyb	5	12	12	10	9	10	9
kirkman	4	12	11	11	9	N/A	N/A
lion9	4	N/A	N/A	N/A	N/A	N/A	N/A
mark1	4	12	11	N/A	N/A	N/A	N/A
opus	4	N/A	N/A	N/A	N/A	N/A	N/A
planet	6	12	11	11	9	9	8
planet1	6	12	11	11	9	10	9
pma	5	12	12	11	9	10	9
s1	5	12	11	10	9	9	9
s27	3	N/A	N/A	N/A	N/A	N/A	N/A
s208	5	12	11	11	9	10	N/A
s298	8	12	11	10	9	9	8

s386	4	12	12	N/A	N/A	N/A	N/A
s420	5	12	12	11	9	9	N/A
s510	6	12	11	10	9	9	9
s820	5	12	11	10	9	9	9
s832	5	12	11	11	9	9	9
s1488	6	12	11	10	9	10	8
s1494	6	12	11	10	10	9	9
sand	5	12	12	10	9	10	8
scf	7	12	11	11	9	10	9
shiftreg	3	N/A	N/A	N/A	N/A	N/A	N/A
sse	4	N/A	N/A	N/A	N/A	N/A	N/A
styr	5	12	11	11	9	10	9
tbk	5	12	12	10	9	9	9
tma	5	12	11	11	9	9	N/A
train11	4	N/A	N/A	N/A	N/A	N/A	N/A

**Table 5-2 State bits of each circuit when hiding different data items**

Although Table 5-2 shows that twice to six times state bits are required, we can claim that this is because we only divide the information sequence into 16 data items. Some circuits in the benchmark are large enough for hiding more items. If the FSM we want to protect has a lot of states, we can divide the information sequence into more parts and thus the required state bits will decrease.

Then we use the tool “SIS” ( Synthesis of both synchronous and asynchronous circuits ) to count the literals of each circuit. Table 5-3 shows the literals of each circuit before and after embedding the information. The first column is the name of each circuit. The second column is the number of literals counted by SIS without

optimization. After applying the optimization script “script.algebraic” which is a build-in script of SIS, the number of literal is shown in the third column of Table 5-3. The fourth column shows the number of literals after embedding the information and applying the optimization script “script.algebraic”.

Circuit	# of ori. lits	# of ori. lits after optimization	With 128 bits embedded
bbara	133	96	244
bbsse	216	153	323
bbtas	53	32	159
beecount	74	59	194
cse	424	255	390
dk14	177	145	287
dk16	502	362	493
dk17	110	99	246
dk27	42	30	181
dk512	104	77	230
ex1	511	303	763
ex2	252	183	344
ex3	111	95	211
ex4	146	105	215
ex5	110	78	207
ex6	211	120	253
ex7	114	94	217
keyb	605	317	394

kirkman	485	305	1070
lion9	47	36	220
mark1	162	112	227
opus	147	96	213
planet	1102	653	780
planet1	1102	653	802
pma	581	230	497
s1	857	468	472
s27	18	14	183
s208	181	86	521
s298	244	138	3151
s386	347	158	337
s420	383	174	438
s510	424	303	504
s820	757	359	854
s832	769	376	920
s1488	1387	751	1249
s1494	1393	762	1438
sand	1126	637	792
scf	1865	953	981
shiftreg	20	8	189
sse	216	153	308
styr	1020	598	758
tbk	1829	953	1814
tma	386	197	367

train11	67	59	212
---------	----	----	-----

**Table 5-3 The number of literals before and after embedding 128 bits information**



## Chapter 6 Conclusions and Future Works

In this research, we propose a new method — state difference encoding — to hide information in a finite state machine. The main idea of our method is to embed the signature which is represented in a sequence of information in the difference of states. The state difference encoding has the following advantages :

1. This method is used in behavioral level and behavioral level provides better reusability.
2. This method can provide thorough protection from behavioral level to physical level.
3. Difficult to observe the signature since it is embedded in the state code.
4. Difficult to fake another signature because the code which can be used is restricted.
5. The signature can be extracted whenever in behavioral level, gate level, or physical level.

On the other hand, our state difference encoding has certain disadvantages :

1. Complex computation. Using our method must find a path for information hiding, formulate the state code requirement into an INLP, and solve the INLP.
2. Hard of detection. The detection procedure includes feed the input state and inputs into the FSM, let the FSM run one step, and extract the current state of the FSM.

There is still something which could be improved in our method. The path finder of our method uses random number to find a path. Although this approach is efficient, the overhead of the selected path varies. We can use smarter algorithm to pick up a

path in order to reduce the overhead instead of random selection. Now, almost the whole computation time is spent on the INLP solver. And The INLP solver takes the same time to solve the INLP no matter the INLP has a solution. Thus, we can tear the whole INLP to many ILPs and solve these ILPs respectively to speed up the computation time. The product term of the modified FSM generated by HDL generator now is directly derived from edges of the original FSM. This will cause the number of literals increases significantly. We can combine some optimization algorithms to the HDL generator to further optimize the generated FSM.





## References

- [1] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, W. Huijuan, and G. Wolfe, "Robust IP watermarking methodologies for physical design," presented at *Design Automation Conference*, 1998.
- [2] A. L. Oliveira, "Techniques for the creation of digital watermarks in sequential circuit designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, pp. 1101, 2001.
- [3] D. Kirovski, H. Yean-Yow, M. Potkonjak, and J. Cong, "Intellectual property protection by watermarking combinational logic synthesis solutions," *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*.
- [4] E. Charbon and I. H. Torunoglu, "On intellectual property protection," presented at *Custom Integrated Circuits Conference*, 2000.
- [5] E. Charbon, "Hierarchical watermarking in IC design," presented at *Custom Integrated Circuits Conference*, 1998.
- [6] I. Hong and M. Potkonjak, "Behavioral synthesis techniques for intellectual property protection," presented at *Design Automation Conference*, 1999.
- [7] I. Torunoglu and E. Charbon, "Watermarking-based copyright protection of sequential functions," *Solid-State Circuits, IEEE Journal of*, vol. 35, pp. 434, 2000.
- [8] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "FPGA fingerprinting techniques for protecting intellectual property," presented at *Custom Integrated Circuits Conference*, 1998.
- [9] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Signature hiding techniques for FPGA intellectual property protection," *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*.
- [10] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Robust FPGA intellectual property protection through multiple small watermarks," presented at *Design Automation Conference*, 1999.
- [11] R. K. Gupta and Y. Zorian, "Introducing core-based system design," *Design & Test of Computers, IEEE*, vol. 14, pp. 15, 1997.