DESIGN OF AN INTEGRATED HARDWARE PLATFORM FOR FOUR DIFFERENT LIGHTWEIGHT BLOCK CIPHERS

By

© Haohao Liao

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Engineering

FACULTY of ENGINEERING AND APPLIED SCIENCE

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

June 2015

St. John’s Newfoundland

Abstract

In recent years, there are more and more embedded devices with limited hardware resources, such as RFID tags and smart cards. In these devices, since the resources are limited, we need some specially designed cryptographic ciphers to ensure the required security level. Many lightweight block ciphers, such as PRESENT, PRINTcipher, LED and Piccolo, were designed to meet these requirements. In these resource-constrained environments, we need specific hardware implementations for these ciphers to minimize resources.

In this thesis, we investigate the hardware implementation of four different, but similar, lightweight block ciphers: PRESENT, Piccolo, PRINTcipher and LED. The purpose of this thesis is to present a common platform which integrates these four ciphers into one system using a shared datapath, with the objective of reducing the area below the total sum of area consumed by the individual ciphers. First, we implement these four ciphers separately, and then design a platform which integrates these four ciphers together into a basic iterative design. Then, we compare the resource consumption results of the platform and the four individual ciphers. In addition to the normal iterative design, we also present a serialized design of the platform, which is more compact than the iterative design. The structure and implementation of the platform is clearly stated in the thesis with the target technology being the Altera Cyclone IV FPGA. The final synthesis result shows the whole design has successfully

achieved the desired objective of flexibility, low resource consumption and compatibility to many applications. We save a lot of hardware resources by significantly reducing the number of dedicated logic registers and combinational functions used in the FPGA.

Acknowledgement

I would like to show my best respect to my supervisor Dr. Howard Heys for his supervision during the past two years. His patience gives me a lot of time to achieve the targeted goal in my research. His experience gives me the best path for my research. I would not be able to successfully finish my graduate studies without his guidance, encouragement, and inspiration. I would also thanks the financial support given to me by him and the School of Graduate Studies.

I also would like to show my best thanks to Dr. Cheng Li and Dr. Lihong Zhang. From their courses, I learnt a lot of expertise which helped me finish my research. In particular the course Advanced Digital System, from the course project, the principle for hardware design, which is always thinking physically, is deeply engraved in my brain. It helps me a lot in my research career.

Thanks to my colleagues Yuanchi Tian, Fan Jiang and Jiming Xu for their friendship. They gave me a lot of help not only in my research, but also in my daily life. We always talk together for the problems encountered during the research, the daily life. Besides, as an international student, we also celebrate some traditional Chinese festival together, which makes me no longer feel alone in Canada.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

3DES Triple DES

ASIC Application Specified Integrated Circuit

AES Advanced Encryption Standard

CMOS Complementary Metal-Oxide Semiconductor

DES Data Encryption Standard

FPGA Field Programmable Gate Array

FSM Finite State Machine

GE Gate Equivalent

Mbps Megabits per second

MHz Mega Hertz

MUX Multiplexer

NIST United States National Institute of Standards and Techonology

TSMC Taiwan Semiconductor Manufacturing Company

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Chapter 1

Introduction

In this chapter, we give a brief introduction about cryptography and some typical applications. Two different design principles are also introduced.

1.1 A Model for Security in Communication Networks

Figure 1.1 shows a typical model for security in a communication network. First, the sender will send the message into the cipher or encryption algorithm, where the message is encrypted into a secret message by using the key. The key is unknown and the secret message is unreadable to the opponent. Then, after the recipient receives the secure message, the decryption algorithm in the recipient's side will perform the decryption where the secure message will be decrypted and readable to the recipient.

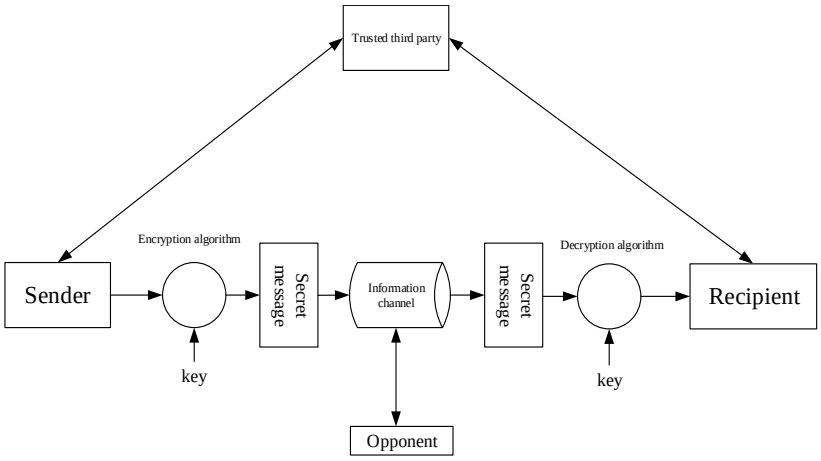


Figure 1.1 Model for Network Security [1]

Here the encryption algorithm and decryption algorithm may be a symmetric key

cipher such as the Advanced Encryption Standard (AES) [2]. Also, it can be public a key cipher such as RSA [2]. A trusted third party may act as the distributer of the secret information. For example, if the encryption algorithm and decryption algorithm is AES, the trusted third party will be used to distribute the cipher key for both the sender and recipient. This key must not be obtained by the opponent, so it must be ensured that the transmission between the sender/recipient and the trusted third party is secure.

There are four tasks in this general model [1]:

1. Design a secure algorithm which will act as the security-related transformation and which should be secure enough to defeat the attempt of the opponent to get the messages sent by the sender. For example, AES is the most applied encryption algorithm today and is believed to have a very high security level.

2. Generate the security information for the algorithm. For example, AES needs to get the encryption/decryption key to finish the encryption/decryption.

3. A secure scheme should be developed to distribute the security information. This is also known as the key distribution scheme.

4. Both the sender and recipient need a protocol to make sure the encryption algorithm, decryption algorithm and key are correctly working to receive the security goal of this model.

1.2 Symmetric Key Ciphers and Public Key Ciphers

A cipher is a set of functions which are combined together to generate an algorithm which transfers the original set of data into another set of data which has no obvious features and is unreadable to an opponent.

Typically, ciphers are divided into two categories: symmetric key ciphers and public key ciphers. Furthermore, symmetric key ciphers consist of block ciphers and stream ciphers.

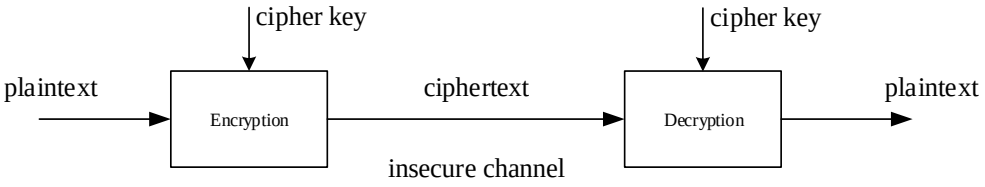


Figure 1.2 Typical Model of Symmetric Key Cipher

Figure 1.2 shows a typical model of how a symmetric cipher works. Normally, the plaintext will be encrypted by using the cipher to transfer the plaintext into ciphertext, which is sent through an insecure channel. At the recipient's end, the ciphertext will be decrypted into plaintext again by using the same cipher. In encryption or decryption, the same key is required to correctly execute the operation. In order to keep the plaintext from any potential attackers, the key must be secret to any other untrusted entities. In this way, if the cipher is carefully designed, the attackers can not recover the ciphertext without the correct key.

1.3 Block Ciphers and Stream Ciphers

Symmetric key ciphers can be divided into two different categories: block ciphers and stream ciphers. A stream cipher encrypts one bit or one byte of data at a specific time [2], while a block cipher divides the data into different blocks, and deals with one block of data at a specific time [2]. Typical block sizes range from 48 to 128 bits. Normally the stream cipher is similar to a one-time pad by using a keystream generator. The keystream generator generates a pseudorandom sequence of bits which have no obvious characteristics. For block ciphers, the architecture of the algorithm usually follows a Feistel cipher structure or Substitution-Permutation Network (SPN) structure.

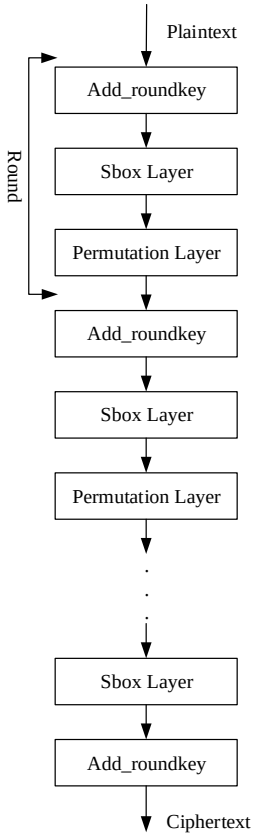


Figure 1.3 A Basic SPN Structure

The SPN structure is widely used in many ciphers. Figure 1.3 shows a typical

structure of an SPN. Basically, an SPN consists of several rounds of three components: Add_roundkey, substitution or sbox layer and permutation layer. The Add_roundkey component is typically a bit-by-bit XOR of the data block and the round key. The round key is achieved through the key scheduling algorithm from the original cipher key. The substitution layer is a non-linear function applied to the data block executed by mapping sub-blocks using a fixed nonlinear function. For example, in many lightweight block ciphers, a 4x4 sbox maps 4 inputs to 4 outputs using nonlinear Boolean functions. The permutation layer is a bit position change of the data block, usually, at the cost of no hardware resources. Sometimes the permutation is replaced by a more complex linear transformation as is done, for example, in AES. A typical lightweight block cipher usually consists of several identical rounds applying a different round key in each round to finally get the ciphertext.

1.4 Public Key Ciphers

In symmetric key ciphers, at the transmitter, plaintext is encrypted into ciphertext which is transferred through an insecure channel and then decrypted at the receiver by using a symmetric key. If the opponent knows the key, it would be very easy for the opponent to recover the ciphertext. In this way, the distribution of the symmetric key is a serious problem. Under this circumstance, public key ciphers are employed. RSA is one of the best known public key ciphers and is based on number theory.

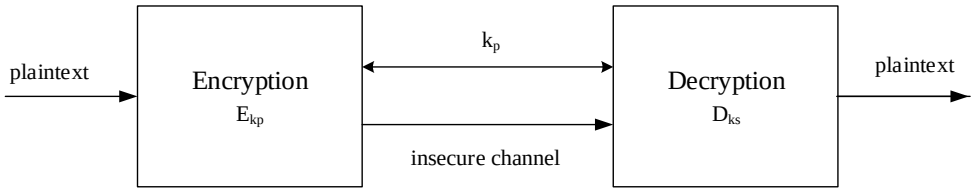


Figure 1.4 Public Key Cipher Model

Figure 1.4 shows the model of a public key cipher. In this model, the encryption key and decryption key are different. The encryption key k_p is a public key which is known to all the senders, whereas the decryption key k_s is a secret key which is only available to the receiver. In order to make sure that after the decryption function, the receiver can recover the plaintext, the following equation must be satisfied: $D_{ks}(E_{kp}(\text{plaintext})) = \text{plaintext}$. Under this model, the public key k_p and secret key k_s are related to each other. However, since the public key is known to everyone, the secret key must not be easily determined by the known public key.

1.5 Lightweight cryptography

Modern cryptographic algorithms are widely used in many applications such as RFIDs, smart cards, Internet of Things (IoT), etc. The requirement for security level ranges from lightweight applications to conventional applications. In some applications, resource consumption is critical. For example, an RFID tag is a lightweight application where hardware cost is the major factor considered and the required security level is to achieve enough security but not a very high security. However, in conventional applications such as web-based banking, AES is the most suitable choice since the

security requirement is usually very high and there are no significant implementation constraints.

1.6 Iterative Design and Serialized Design

For symmetric key block ciphers, different hardware design approaches are available. One approach is the iterative design. In this approach, one clock cycle will deal with all the functions inside a round. For example, the PRESENT cipher has three operations in one round: Add_roundkey, sbox and permutation. If an iterative design is applied, all these three operations will be completely finished in one clock cycle. If the block size is 64 bits, we need a 64 bit "XOR" and 16 4x4 sboxes. The iterative design is sometimes also called the round-based design.

Compared to the iterative design, a more compact design approach is the serialized design. In a serialized design, only one sbox will be applied for one clock cycle. Besides, for the Add_roundkey layer, the size of the "XOR" is based on the size of datapath we want to use, and usually, we choose the same size of the "XOR" as the size of the sbox. For example, the serialized design of PRESENT, a 4 bit "XOR" will be applied to the Add_roundkey layer and only one 4x4 sbox will be applied in one clock cycle. Usually, a shift register will be used for the state register and key register inside a serialized design.

1.7 Motivations and Contributions

Lightweight cryptography is a hot and promising research area. However, typically

specific lightweight block ciphers are found to be most suitable for particular applications. For example, PRINTcipher is designed for IC-Printing. However, in some applications, such as RFID tags and other embedded systems, there is a potential need to integrate several different lightweight block ciphers into a single device to supply higher flexibility for multiple application environments. However, there is always a trade-off between flexibility and hardware resource consumption. Under this situation, we develop a digital hardware platform which integrates four well regarded recently proposed lightweight block ciphers: PRESENT, Piccolo, PRINTcipher, and LED. This platform is designed to only modestly increase the hardware resource consumption beyond the requirements of one cipher. It is found that the platform is efficient and resource-friendly, important considerations since the resource usage is critical for most embedded systems.

The purpose and motivation of my research is to design the platform which can meet these requirements: high flexibility, low resource consumption and compatibility to various applications.

On one hand, it is easier for the users to apply the platform in different application environments. On the other hand, they do not need to worry about the large increase of the hardware resource consumption since the platform is an efficient implementation of four lightweight block ciphers. In this way, the users can have a very compatible embedded devices with high flexibility with only a very modest cost.

The organization of this research thesis is presented in subsequent chapters as listed below:

- Chapter 2 provides the detailed background for the design in this thesis. The design specification of PRESENT, Piccolo, PRINTcipher and LED are presented in this chapter. Besides, it also introduces the differences between the conventional block ciphers and lightweight block ciphers. A list of the implementation results of several lightweight block ciphers are also provided in this chapter.

- Chapter 3 introduces the iterative hardware design and implementation of both the four individual ciphers and the multi-cipher platform. Block diagrams and the state transition diagrams for the finite state machines (FSMs) are provided. It is particularly focused on the structure of the platform and the philosophy of how the hardware resources are saved. The synthesis results are also provided.

- Chapter 4 presents the serialized hardware design and implementation of both the four individual ciphers and the multi-cipher platform. Block diagrams, state transition diagrams for the FSMs and some simulations results are provided. Also, it will particularly pay attention to the structure of the platform. Some different ways of implementation are also presented with analysis in relation to the synthesis results.

- Chapter 5 draws a conclusion for all the designs and implementations.

Additionally, it also provides the future research direction.

Chapter 2

Background

In this chapter, we provide the background for our research. Four lightweight block ciphers which are integrated into the platform are discussed in detail while some other lightweight block ciphers are also briefly introduced. Besides, we also analyze the reason why we chose these four ciphers and not others.

2.1 Conventional Cryptography and AES

Cryptography has a very long history dating back to the Caesar cipher [2] and beyond. Nowadays, in order to provide higher security, more and more ciphers with complex algorithms are being invented. For most conventional block ciphers, the hardware implementation typically has a very high security level at the sacrifice of costing a large amount of hardware resource such as area on a chip.

The Data Encryption Standard (DES) [3], which was the old standard of the National Institute of Standards and Technology (NIST), is still in service in some applications. Also, after some modifications to the algorithms used in DES, 3-DES, which is also called triple-DES was developed. However, in order to provide a better encryption algorithm which is capable ensuring the security of sensitive government information in the 21st century, Rijndael was selected as the Advanced Encryption Standard (AES) [4].

AES is the most widely used conventional block cipher all over the world. It is a symmetric key block cipher which is designed to replace DES in many commercial and government applications. The block size for AES is fixed as 128 bits, while the key size is flexible and can be 128, 192 or 256 bits. As a conventional block cipher, AES is very complex in its data processing structure. In particular, the implementation of the sbox layer inside AES is basically not very hardware-friendly. As a result, the hardware implementation of AES costs a lot of resources, but generates a very high security level which is more than needed in some embedded applications.

2.2 Lightweight Cryptography

In most cryptographic applications, AES is the best choice. However, AES is not suitable for some resource-constrained environments. Under this circumstance, lightweight cryptography is designed to meet the requirements. Lightweight cryptography has 4 important design considerations [5] :

1. The targeted environment is a resource-constrained environment.

2. It is not a good choice for all applications. In most resource-sufficient environments, AES may be a better choice.

3. Lightweight is not equal to weak. Lightweight cryptography also has a enough security for most attacks but likely not for an extremely strong opponent.

4. Lightweight cryptography is designed for a specific platform, while conventional cryptography (eg. AES, DES and so on…) considers a broad range of target

platforms.

Table 2.1 Difference Between Lightweight Block Ciphers and Conventional Block Ciphers

Features / **Cipher**	**Key Scheduling Algorithm**	**Key size**	**Block Size**	**Number of Rounds**
Lightweight block cipher	Simple	Small (80 bits)	Small (64, 80 bits)	More (25,32,48)
Conventional block cipher	Complex	Large (128 bits)	Large (128 bits or larger size)	Fewer (typically 10)

Many lightweight block ciphers are designed according to these criteria. PRESENT, Piccolo, PRINTcipher and LED are four typical lightweight block ciphers [6]. Compared to traditional block ciphers, lightweight block ciphers always have a smaller block size, a simpler key scheduling algorithm and a smaller key size. In order to achieve enough security for lightweight applications while the key size and block size are reduced, the key scheduling algorithm is simplified, and the number of rounds for block ciphers is increased. Table 2.1 shows the differences between lightweight block ciphers and conventional block ciphers.

2.2.1 PRESENT Cipher

The PRESENT cipher is an ultra-lightweight block cipher presented in 2007 [7]. It uses a typical SPN structure.The label PRESENT-80 refers to the cipher structure with an 80 bit length key, while PRESENT-128 uses a 128 bit length key. PRESENT-80 consists of 32 rounds by using a structure of substitution permutation network and it works based on a block size of 64 bits.

Figure 2.1 shows the structure of the PRESENT cipher. The "Add_roundkey" component performs a simple 64 bit XOR between the round key and the state. For each round, the 64 bit block is divided into 16 4-bit nibbles for input to the sbox layer. The sbox in PRESENT is 4 bits, half of the size of the sbox in AES. Although an 8 bit sbox can achieve higher security, it is not hardware friendly [8]. However, since the sbox is the only non-linear component in PRESENT, it is also carefully designed against differential and linear attacks. The permutation layer is a bit position change performed on the 64 bit data block according to the following pseudocode:

```
for i in 0 to 15 generate
    output(i) <= input (4* i);
    output(i+16) <= input(4*i+1);
    output(i+32) <= input(4*i+2);
    output(i+48) <= input(4*i+3);
```

Note that the last round of the PRESENT cipher does not include an sbox layer or permutation layer. The last round key will XOR with the last state and finally generate the ciphertext.

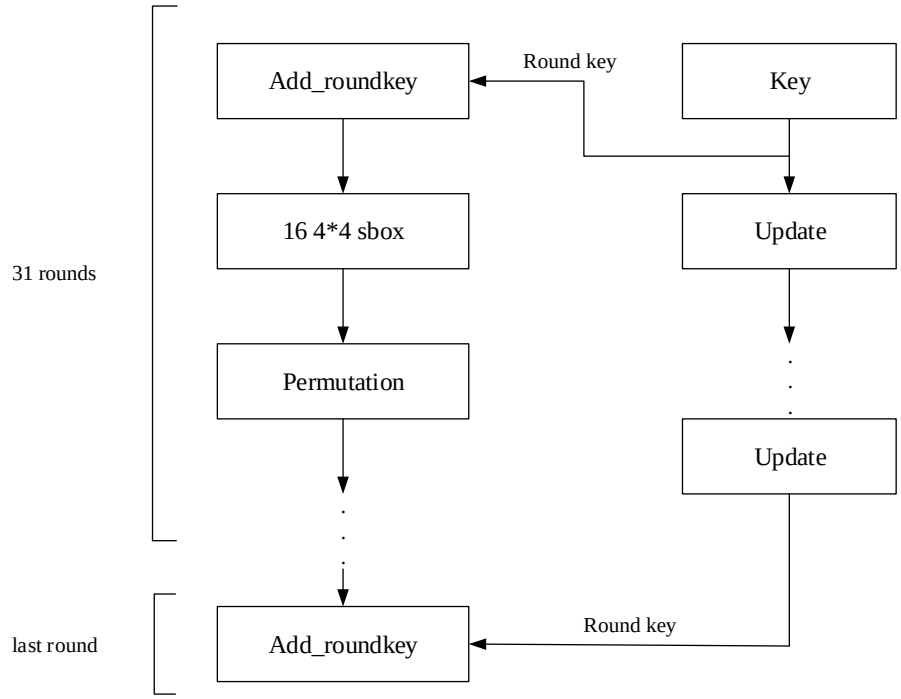


Figure 2.1 Structure of PRESENT. [7]

The key scheduling part of this cipher is comprised of a 61 bit left shift, an sbox and an XOR with the round counter. For PRESENT-80, the key scheduling algorithm is performed according to the following steps [7]:

1. $[k_{79}k_{78} \ldots k_1k_0] <= [k_{18}k_{17} \ldots k_{20}k_{19}]$.

2. $[k_{79}k_{78}k_{77}k_{76}] <= S[k_{79}k_{78}k_{77}k_{76}]$.

3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] <= [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus$ [round counter].

where S represents the sbox operation.

2.2.2 Piccolo Cipher

The Piccolo cipher is a lightweight block cipher recently published in 2011 [9]. It uses a Generalized Feistel Network (GFN). Figure 2.2 shows the structure of the Piccolo cipher. The cipher is similar to the PRESENT cipher. It also has two different key lengths: 80 bits and 128 bits. The 80-bit key version, Piccolo-80, consists of 25 rounds.

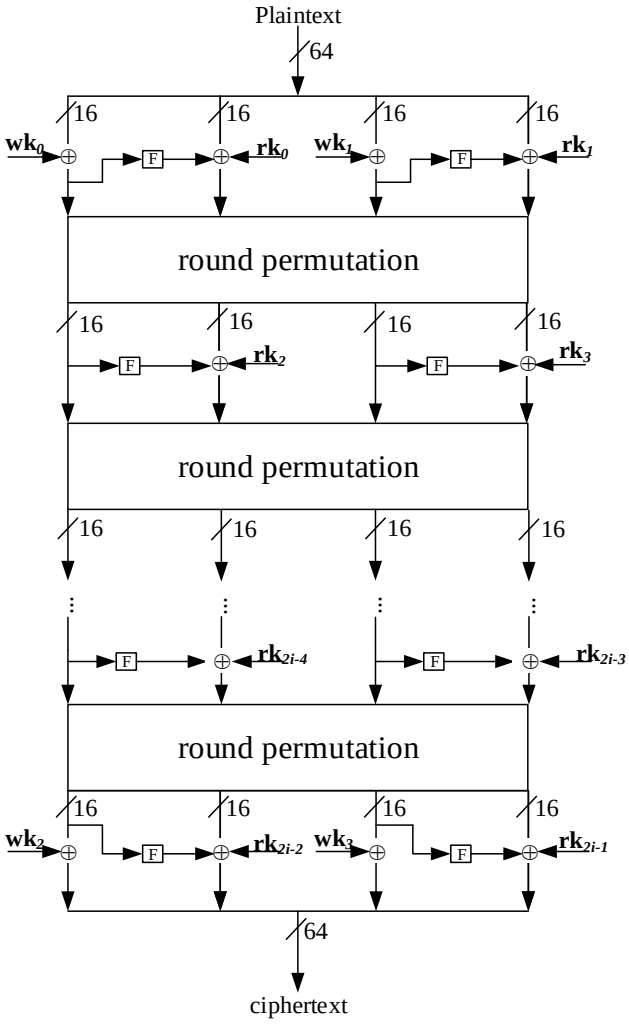
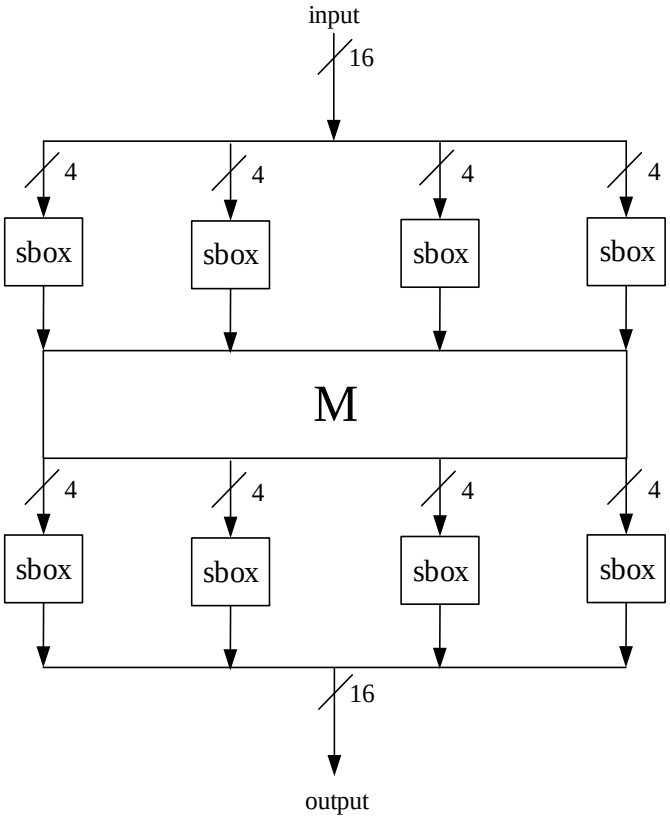


Figure 2.2 Structure of Piccolo Cipher [9]

The combinational datapath of the Piccolo-80 includes a round function, F, which consists of a diffusion matrix which is taken from AES and has eight sboxes. Figure 2.3 shows the structure of the F function and the matrix used in the F function.



$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

Figure 2.3 F Function and the Diffusion Matrix

The calculation of the matrix is based on the Galois Field $GF(2^4)$ [1] by using an irreducible polynomial $x^4 + x + 1$. For a 16 bit input of the F function, the result of this calculation is included in the VHDL code which can be found in Appendix A.1.

The key scheduling part for Piccolo consists of two different keys. One is whitening

key wk_i which is used in the first and last round of the encryption. The other one is round key rk_i which has two different values for each round. For round *i*, the two different round keys are labelled as rk_{2i-2} and rk_{2i-1}. In each round, two round constants rc_{2i-2} and rc_{2i-1} are used to generate the round key rk_{2i-2} and rk_{2i-1}. The details of this key scheduling algorithm is too complicated to demonstrate here. Refer to [9] for more details.

2.2.3 PRINTcipher

PRINTcipher is a lightweight block cipher published in 2010 [10]. Unlike the PRESENT cipher and Piccolo cipher, the cipher operates on a 48 or 96 bit data block. However, PRINTcipher still uses the conventional substitution permutation network except the permutation is selected by the key. The label PRINTcipher-48 refers to the cipher structure with a 48 bit length data block, while PRINTcipher-96 uses a 96 bit length data block. Figure 2.4 shows one round of PRINTcipher-48. SK_1 is the 48 most-significant bits of the original key, while SK_2 is the 32 least-significant bits of the original key.

Each round of PRINTcipher-48 consists of 4 different layers. First, the state value will XOR with the 48 bit round key SK_1, which is called the “Add_roundkey” layer. Second, in the permutation layer, the state will perform a simple linear diffusion. Third, the last 5 bits of the state performs a bitwise XOR with the round constant “rc”. Finally, after the combination of sbox and keyed-permutation, the last layer is also called the

virtual sbox layer [10]. The input and output of the virtual sbox is listed in Table 2.2. The 3-bit data input refers to the data input of the virtual sbox and the 2 bit key input refers to the key input which is taken from SK_2. In Table 2.2, output(x|key=k) refers to the output of the virtual sbox when the 2 bit key is k and 3 bit data input is x.

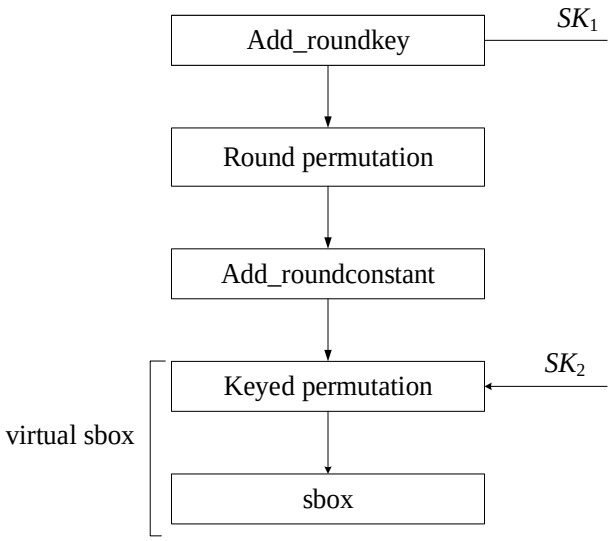


Figure 2.4 One Round of PRINTcipher

Table 2.2 Virtual sbox of PRINTcipher [10]

3 bit data input	0	1	2	3	4	5	6	7
Output(x|key=0)	0	1	3	6	7	4	5	2
Output(x|key=1)	0	1	7	4	3	6	5	2
Output(x|key=2)	0	3	1	6	7	5	4	2
Output(x|key=3)	0	7	3	5	1	4	6	2

The PRINTcipher consists of 48 rounds. The key scheduling part for PRINTcipher is less complex than the previous two ciphers. The key is identical for each round. In each round, 48 bits of key, SK_1, are used to XOR with the state and 32 bits of key, SK_2, are used for the keyed-permutation. PRINTcipher uses a 3-bit sbox, which can form a virtual sbox when combined with the keyed-permutation.

2.2.4 LED Cipher

The LED cipher is a lightweight block cipher published in 2011 [11]. Similar to the above mentioned three ciphers, LED cipher also uses an SPN structure. The cipher operates on a 64 bit data block. It has four different key sizes: 64 bit, 80 bit, 96 bit and 128 bit and they are labelled as LED-64, LED-80, LED-96 and LED-128, respectively. LED-64 needs 32 rounds to finish encryption while the others need 48 rounds to finish encryption. In the LED cipher, a step is defined as four identical rounds without an "Add_roundkey" process. The major difference between the LED cipher and other three ciphers is the key scheduling component. LED cipher does not need a round key for each round. Instead, it needs a step key for each step.

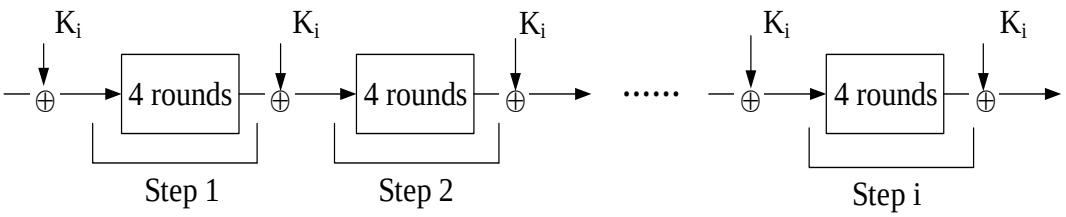
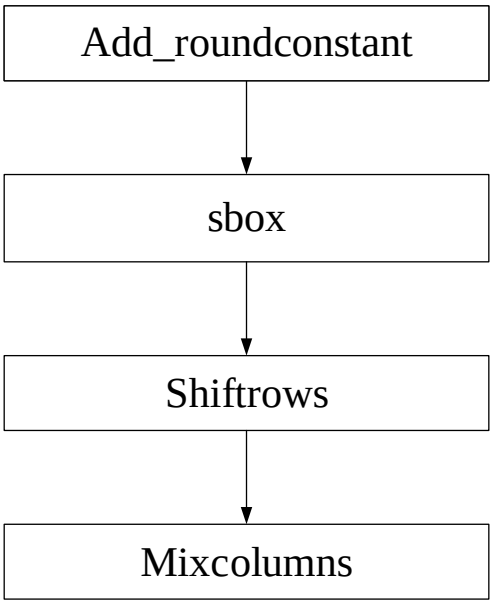


Figure 2.5 Structure of LED Cipher [11]

Figure 2.5 shows the data flow of the LED cipher. For LED-64, 8 steps are needed

to finish the whole encryption process, while for LED-80, LED-96 and LED-128, 12 steps are needed to finish the encryption process. As described above, each step has four identical rounds. Each round consists of four layers listed as: “Add_roundconstant”, “sbox”, “Shiftrows” and “Mixcolumns”. The structure of these four layers is shown in Figure 2.6.



$$A^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}^4 = \begin{pmatrix} 4 & 2 & 1 & 1 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix} = M$$

Figure 2.6 One Round and The Mixcolumns Matrix of LED Cipher.

The LED cipher also has a Mixcolumns layer which is similar to AES. Usually, for a round-based implementation, matrix *M* is used in the Mixcolumns, while in compact serialized implementation, matrix *A* is adopted for four times in the Mixcolumns. The computation of the diffusion matrix is based on $GF(2^4)$ by using an irreducible polynomial $x^4 + x + 1$.

2.2.5 Some Other Lightweight Block Ciphers

In addition to the mentioned-above four ciphers, there are many more other lightweight block ciphers. For example, KATAN32 [12], which is one of the variants of KATAN cipher family, has a 32 bit block size. In KATAN32, the state register is divided into two parts L1 and L2 and two different functions $f_a(\ )$ and $f_b(\ )$ to perform as the nonlinear part inside this cipher. Another example is KLEIN [13] cipher, in which the same Mixcolumn operation is used as AES. The round permutation, which is called the RotateNibbles step, for KLEIN cipher is pretty simple, just an 8 bit left shift. Both these lightweight block ciphers have much less complex encryption or decryption algorithms than AES.

The reason why we choose PRESENT, Piccolo, PRINTcipher and LED to be implemented and integrated into our platform rather than other lightweight block ciphers is these four ciphers were recently presented at the well-regarded Cryptographic Hardware and Embedded Systems (CHES) conference and are considered by the cryptographic community to be serious proposals. Furthermore, these four ciphers share a similar structure compared to other ciphers, such as the KATAN cipher which has no sbox layer. Instead, it uses two other different nonlinear functions for each round. Since we need to find similar components for the ciphers to integrate into the multi-cipher platform, we chose to use these ciphers in which similar structure are shared.

2.3 Implementation Results and Comparison of Lightweight Block Ciphers

In recent years, lightweight cryptography has been an extraordinarily hot research field. Not only the above-mentioned four typical lightweight block ciphers, but also many lightweight block ciphers such as MIBS [14], KlEIN [13], KATAN [12] and TWINE [15] have also been developed. All these lightweight block ciphers are aimed at efficiency and low-cost. However, it is extremely hard to find suitable evaluation metrics for these different lightweight block ciphers since different ciphers are designed based on different perspectives. In [16], the author introduced several different metrics which could be applied to evaluate the hardware implementation lightweight block ciphers.

Table 2.3 shows several metrics and their relationships. The column of "Relative to" means the change of a specific application constraint is related to other constraints [16]. For example, if we want to save a lot of area or reduce the instantaneous power of a hardware implementation, we can simply share as many resources as possible or just reduce the datapath to 1 bit. However, in this way, the throughput will be significantly reduced, which results in the increase of the time constraints. Also, to finish one encryption cycle, the energy cost may be increased. In this way, the possible design principles of block ciphers can be divided into two categories. The first one is "design for low area and power" and the second category is "design for high throughput and low energy". In [17], similar opinions are provided. Actually, the evaluation metric for

a lightweight block cipher should also include the security level. Usually, a high throughput implementation of a block cipher with a high security level always results in a high cost or area [17].

Table 2.3 Summary of Evaluation Metrics for Hardware Implementations [16]

<table>
<tr><th>Application constraints</th><th>Relative to</th><th>Hardware design goals</th><th>Algorithmic design goals</th></tr>
<tr><td>area (um^2 or Gate Equivalent (GE))</td><td rowspan="2">Time or energy constraints</td><td>Share resources</td><td rowspan="2">Reduce components cost & versatility</td></tr>
<tr><td>instantaneous power (J/sec)</td><td>Reduce datapath</td></tr>
<tr><td>throughput (bit/sec)</td><td rowspan="2">Area or power constraints</td><td>Pipeline, parallelize</td><td rowspan="2">Minimize the total combinational cost</td></tr>
<tr><td>energy (J/bit)</td><td>unroll</td></tr>
</table>

The hardware design goals and the algorithmic design goals are the methodology which could be applied to reduce the area, power and/or energy or increase the throughput. In our thesis, all these four ciphers we choose to be integrated on the platforms have basically simple algorithms both in the datapath and the key scheduling. As a result, these lightweight block ciphers achieve the goal of area efficiency.

In this thesis, we study an iterative design and a serialized design focused on sharing the hardware components to reduce the area cost which focuses on both reducing the width of the datapath and sharing similar hardware components.

2.4 FPGA Design and Implementation Methodology

Field-Programmable Gate Array (FPGA) and Application Specified Integrated Circuits (ASIC) are both potential choices for our targeted environment. However, FPGA is our first choice since it is easier to quickly prototype in FPGA and the development tools are mature and reliable.

Hardware Description Languages (HDLs) [18] are now widely used in industry and academic fields. HDLs are used to describe the hardware circuits in programmable languages. Three different design levels of HDLs are: behavior level, Register-to-Transfer (RTL) level and structural level. The most popular two HDLs are VHSIC HDL (VHDL) and Verilog HDL. In our research, we use VHDL as the programmable language.

2.4.1 Logic Elements (LEs)

In our thesis, the number of Logic Elements (LEs) is used to indicate the hardware resources consumed our design since they are the smallest unit [19] in the Altera Cyclone IV FPGA devices. Figure 2.7 shows the structure of the LEs used in Altera Cyclone IV device. The basic components of one LE are a 4 input Look-Up Table (LUT) and a single programmable register. A 4 bit input LUT is actually a 16x1 Random

Access Memory (RAM). The 4 bit input acts as the address line of the RAM and after synthesis, the synthesis tool will calculate the possible outputs of a certain combinational circuit and store the results in the RAM. The LUT and the programmable register can separately drive the three different outputs. This feature allows the synthesis tool to generate the resource utilization result in two different categories: combinational functions and dedicated logic registers. The LEs have two different working modes. The first one is normal mode which is suitable for normal combinational functions. The second one is arithmetic mode which is suitable for comparators or counters. In our design, except for the counters, most LEs are working in normal mode.

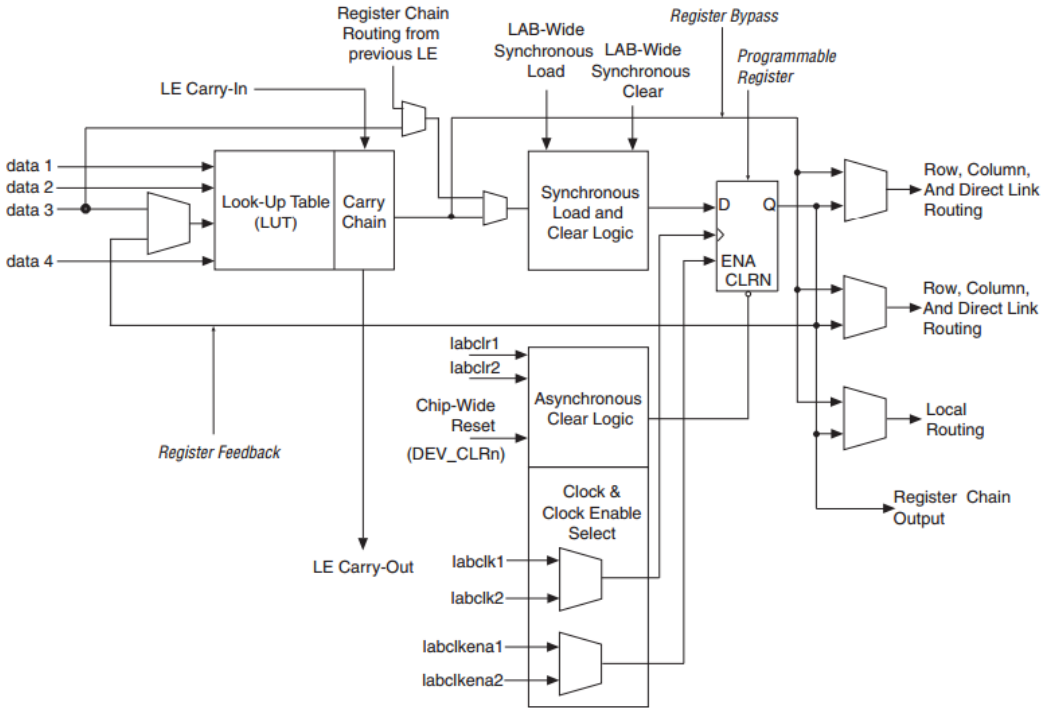


Figure 2.7 Logic Elements for Cyclone IV device [19]

In this thesis, we often enumerate resources based on synthesis results which list

the number of combinational functions and dedicated logic registers. Note that one logic element is needed for every combinational function and/or register. Hence, the number of LEs needed can be assumed to be the larger of the number of combinational functions and dedicated logic registers.

2.4.2 FPGA Design Flow

Figure 2.8 provides the FPGA design flow for Altera devices. The "Design" includes analyzing the specification of a design and VHDL coding. Then the whole project will be compiled before we apply the functional simulation. In the "Compile" process, we need to deal with any syntax error in our code, while in the "Simulate" process, we need to make sure that the test vectors can run through the project correctly. After finishing the simulation, the synthesis tool Quartus II [20] will be used to program the project to our targeted device.

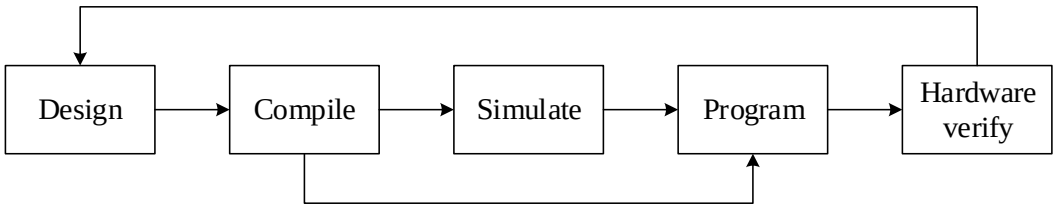


Figure 2.8 FPGA design flow [21]

In our research, we do not include the "Hardware verify" process in the design flow. We only complete the simulation and synthesis and leave the real hardware verification for the future work. Moreover, "Hardware verify" is not necessary to obtain the desired results of the analysis.

2.5 Summary

In this chapter, we introduced the necessary background required for our work in this thesis. First, we discussed the differences between lightweight cryptography and conventional cryptography. Then, we introduce the details of four lightweight block ciphers that are designed and implemented in our thesis. Besides, we discussed the reasons why we chose these four ciphers instead of other lightweight block ciphers. Finally, a basic background on FPGAs and their design flow was introduced in the last section.

In the next chapter, we will discuss the details of the iterative design for each individual cipher and the multi-cipher platform. Comparison of the synthesis results between these designs will be provided.

Chapter 3

Iterative Design of Individual Ciphers and the Multi-cipher Platform

In this chapter, an iterative design and implementation of the four individual ciphers and the multi-cipher platform are presented. An iterative design is also a round-based design. In this design, one clock cycle deals with one round of a single state. Thus, the structure of each design is basically straightforward. Since in many applications, only the forward or encryption process of the block cipher is required (eg. counter mode of operation), this chapter does not discuss the decryption process, although due to the similar structure of encryption and decryption processes, similar results of resource consumption are expected. Some of the results of this chapter were presented in [22].

3.1 Iterative Design of Four Individual Ciphers

In this section, we discuss the iterative design of the individual ciphers. We first discuss the common interface used for all designs and then describe the details of the structure of each cipher.

3.1.1 Block Diagram

For all the four individual ciphers, only PRINTcipher uses a 48 bit data block while 64 bit data block is required for PRESENT, LED and Piccolo. In order to keep

consistency for the size of data block with a view towards integrating all these ciphers together into one platform, in our work, we use a 64 bit input data for PRINTcipher with the 16 most-significant bits set to all ‘0’. The intended datapath also uses 64 bits. For all these 4 ciphers, an 80 bit key version is used in the hardware implementation.

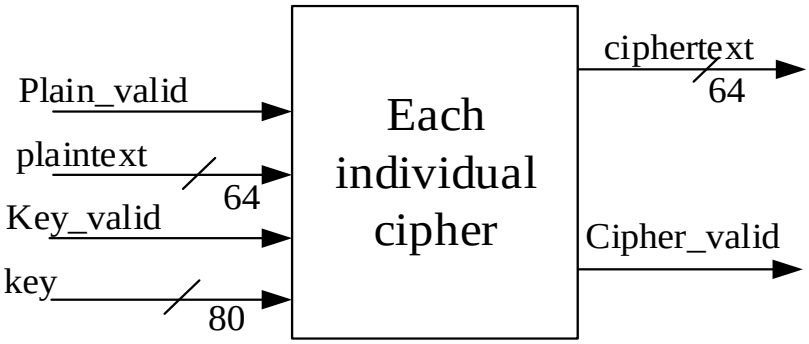


Figure 3.1 Block Diagram for Each Individual Ciphers

Except for the data signal, two different valid signals, “Key_valid” and “Plain_valid” are used to indicate when the input “key” or “plaintext” is ready, respectively. After an appropriate number of rounds of encryption, a “Cipher_valid” signal is asserted and the “ciphertext” is ready at the output data line. For PRINTcipher, the 16 most-significant bits of output “ciphertext” will be set to default value of ‘0’.

3.1.2 Iterative Design of PRESENT Cipher

Figure 3.2 shows the hardware structure of the PRESENT cipher. According to the specification of the PRESENT cipher, the combinational datapath which is included in the dashed line of Figure 3.2, includes three layers. The “Add_roundkey” layer performs a 64 bit-wise XOR with the round key. Since the iterative design uses a 64 bit datapath, 16 4x4 sboxes are needed for each round. The permutation layer is a simple

linear layer which changes the bit positions of the state.

The state register is used to store the current state for the current clock cycle. The "MUX" is used to select data from the current state or the plaintext. For the whole encryption process except the first round of the encryption process, the "MUX" should choose the input from the current state register. If the current round is the last round, the "Cipher_valid" signal will be asserted and we can get the ciphertext from the output of the "Add_roundkey" component.

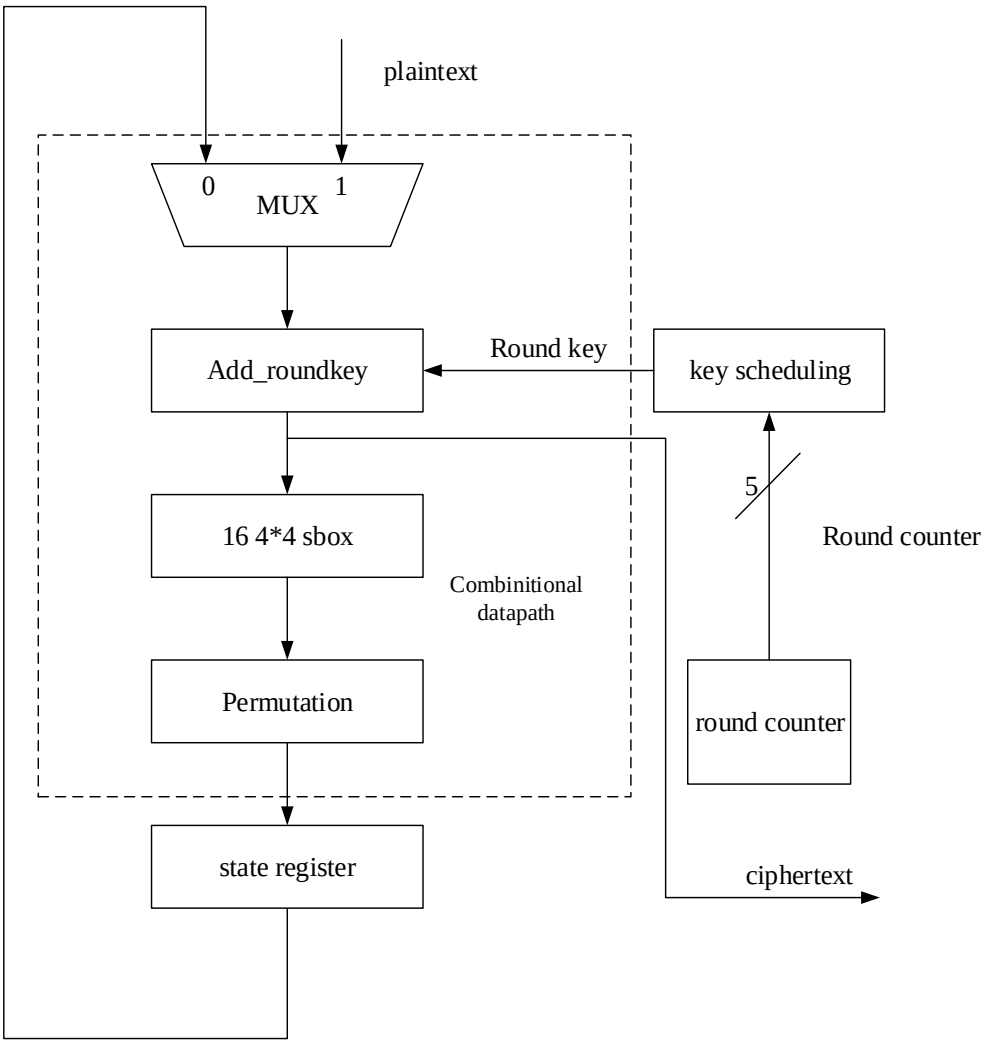


Figure 3.2 Hardware Structure of Iterative Design of PRESENT Cipher

The key scheduling component consists of an 80 bit key register and the key scheduling algorithm consists of a 61 bit left shift, an sbox layer for the 4 most-significant bits and an XOR with the round counter for bit 19 to bit 15.

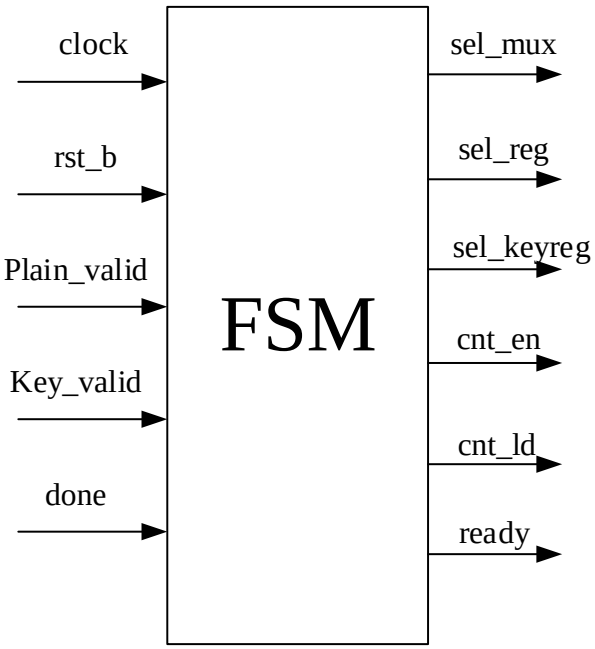


Figure 3.3 Block diagram of PRESENT FSM

The FSM is introduced in Figure 3.3. Basically, for the iterative design of PRESENT, the FSM is not very complex. A Mealy machine is used for the FSM since the outputs of the FSM are based on both the current state and input. Only three states - “IDLE”, “wait_plain”, “ENCRYPTION” - are needed. The whole system uses an active-low asynchronous reset signal, “rst_b”, and assumes that the upstream sends the key to the system first. The system is in “IDLE” state after the asynchronous “rst_b” is de-asserted, and starts to work after it is asserted. The FSM steps into “wait_plain” state after the “Key_valid” signal is asserted which means the key for this encryption process is already loaded into the key register. In this state, the system is waiting for the plaintext

and ready to start the encryption. After the “Plain_valid” signal goes to high, the FSM transfers to “ENCRYPTION” state and the “cnt_en” signal is asserted to allow the round counter to start counting. After 31 clock cycles, the round counter asserts the “done” signal to indicate the end of the encryption process and at the same time, the “ready” signal is asserted by the FSM to tell the top level entity that the ciphertext is ready and the “ready” signal is asserted. At last, the FSM transfers to “IDLE” state and waits for the next encryption.

The whole design is coded by using VHDL and ModelSim is used for the simulation. The simulation results are shown in Figure 3.4. The input plaintext and key are all ‘0’ with the corresponding ciphertext being 5579C1387B228445 which is exactly the same as the test vectors supplied in [7].

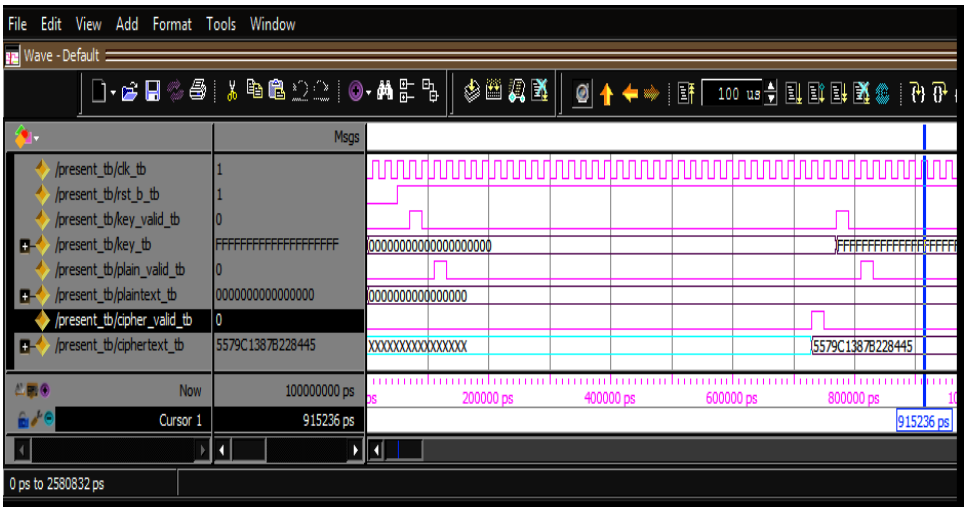


Figure 3.4 Simulation Results of PRESENT Cipher

After successfully simulating the design on ModelSim [23], Quartus II [20] is used for the synthesis based on Altera’s Cyclone IV FPGA [24]. Table 3.1 shows the result of the resource consumption. From Table 3.1, the number of the total combinational

functions and dedicated logic registers consumed by the whole cipher is 608 and 153, respectively. Moreover, the key scheduling component consumes 328 combinational functions which is more than half of the total combinational functions of the whole cipher. This is a very important point that will be further discussed in later sections.

Table 3.1 Synthesis Result of the Iterative Design of PRESENT Cipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
PRESENT	608	153	100%
key scheduling	328	80	53.5%

3.1.3 Iterative Design of Piccolo Cipher

Figure 3.5 shows the structure of the iterative design of Piccolo. Compared to the PRESENT cipher, it is a little bit more complex since Piccolo uses four different whitening keys before the first round and after the last round. Multiplexers “MUX0”, “MUX1”, “MUX2”, “MUX3” are 16 bit 2-to-1 multiplexers while “MUX4” and “MUX5” are 64 bit 2-to-1 multiplexers. The 16 bit multiplexers are used to choose the data from each round or the result of adding the whitening key. “MUX4” is used to choose the correct input to the combinational datapath from the current state or the plaintext. Since the last round does not apply a round permutation, “MUX5” is needed.

The original key is loaded into the key register which is included in the key scheduling component. To generate round key "rk_{odd}" and "rk_{even}", two different round constants, "rc_{odd}" and "rc_{even}" which are generated by the round counter, are required.

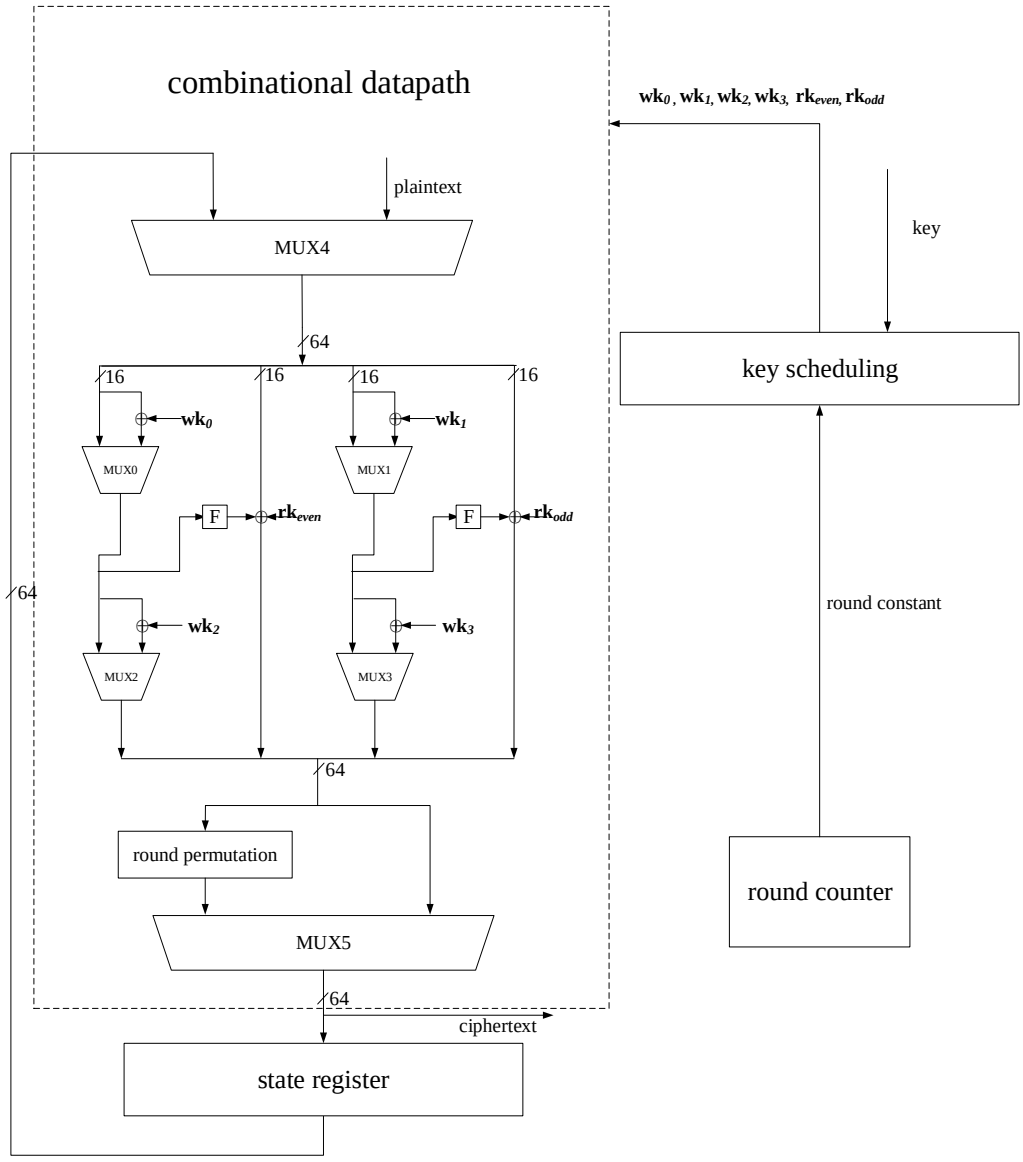


Figure 3.5 Hardware Structure of Iterative Design of Piccolo Cipher

Since the block diagram for the FSM of Piccolo is almost exactly the same as PRESENT cipher except that the FSM of Piccolo cipher has five more outputs since it has five more multiplexers, the block diagram for the FSM of Piccolo cipher is not

provided here. However, Figure 3.6 is provided to introduce the state transition diagram of Piccolo cipher. Three states listed as “IDLE”, “wait_plain”, “ENCRYPTION”, which are exactly the same as the states in PRESENT cipher are required. The whole system starts from the “IDLE” state. After the “rst_b” is asserted which means the asynchronous reset is not active and the “key_valid” goes high, the FSM transfers to “wait_plain” state. When the “plain_valid” has a one clock cycle positive pulse, the FSM steps into the “ENCRYPTION” state and begins the encryption process and at the same time, the “cnt_en” signal will be asserted to enable the round counter to start to count. If the “done” signal is asserted, the FSM goes back to “IDLE” state and waits for the next encryption cycle.

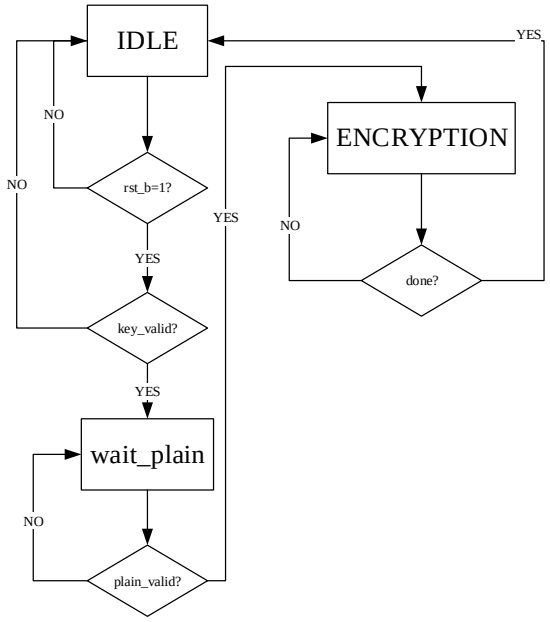


Figure 3.6 State Transition Diagram of the FSM of Piccolo

The whole design is coded in VHDL and simulated by ModelSim. Figure 3.7 shows the simulation result of Piccolo cipher. The test vector used in this simulation is exactly

the same as what it is in [6]. The corresponding ciphertext shows that the design is correct.

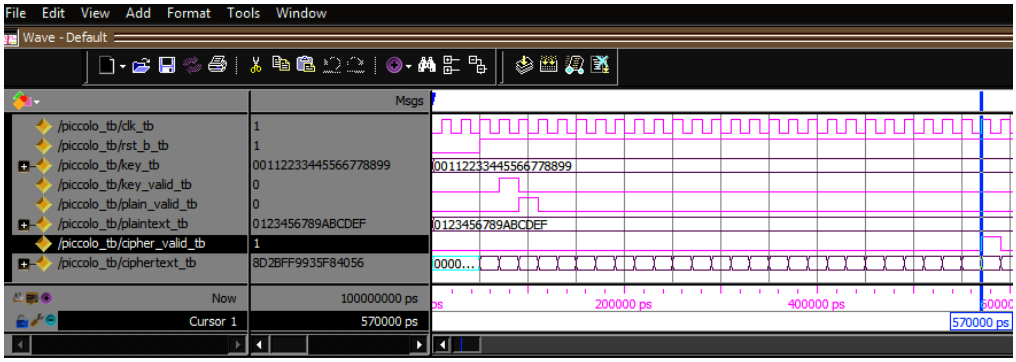


Figure 3.7 Simulation Results of Piccolo Cipher

Table 3.2 Synthesis Result of the Iterative Design of Piccolo Cipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
Piccolo	809	153	100%
key scheduling	431	80	53.2%

Similar to the PRESENT cipher, this design is also synthesized based on the same device by using Quartus II. Table 3.2 shows the synthesis results of the Piccolo cipher. Compared to the PRESENT cipher, the iterative design of Piccolo cipher consumes 809 combinational functions which is slightly more than the PRESENT cipher and 153 dedicated logic registers. The reason is because of the structure of the iterative design of the Piccolo cipher: it has four more 16 bit 2-to-1 multiplexers and one more 64 bit

2-to-1 multiplexers. Similar to the PRESENT cipher, the key scheduling component also consumes over 50 percent of the total number of combinational functions.

3.1.4 Iterative Design of PRINTcipher

Figure 3.8 presents the iterative design of PRINTcipher. Although PRINTcipher is a 48 bit cipher, a 16 bit all '0' default value is used to ensure the datapath of the design is also 64 bits which is the same as the datapath of the other three ciphers. The input to the "MUX" is a 64 bit signal with the 16 most-significant bits set as '0' and the 48 least-significant bits being the original plaintext. All the components inside the dashed box form the combinational datapath of PRINTcipher. The datapath within the dashed box is divided into two routes. One is a 16 bit all '0' datapath which is just a bypass of all the components inside the combinational datapath, another one is 48 bit datapath which contains the valid information of each state and passes through all the components inside the combinational datapath. These two different routes are combined together at ***A***.

In PRINTcipher, as described in Section 2.2.3, the key scheduling algorithm is quite simple. Actually, the round key "SK_1" and permutation key "SK_2" used for each round is identical. The 48 most-significant bits of the original 80 bit key is used as the round key "SK_1", while the 32 least-significant bits act as the permutation key "SK_2".

The FSM of this design is almost the same of the PRESENT cipher. The only difference is that PRINTcipher needs 48 rounds to finish one encryption cycle while

PRESENT needs 32 rounds. In this way, the “done” signal which indicates the end of the encryption process will be asserted later than that in the PRESENT cipher.

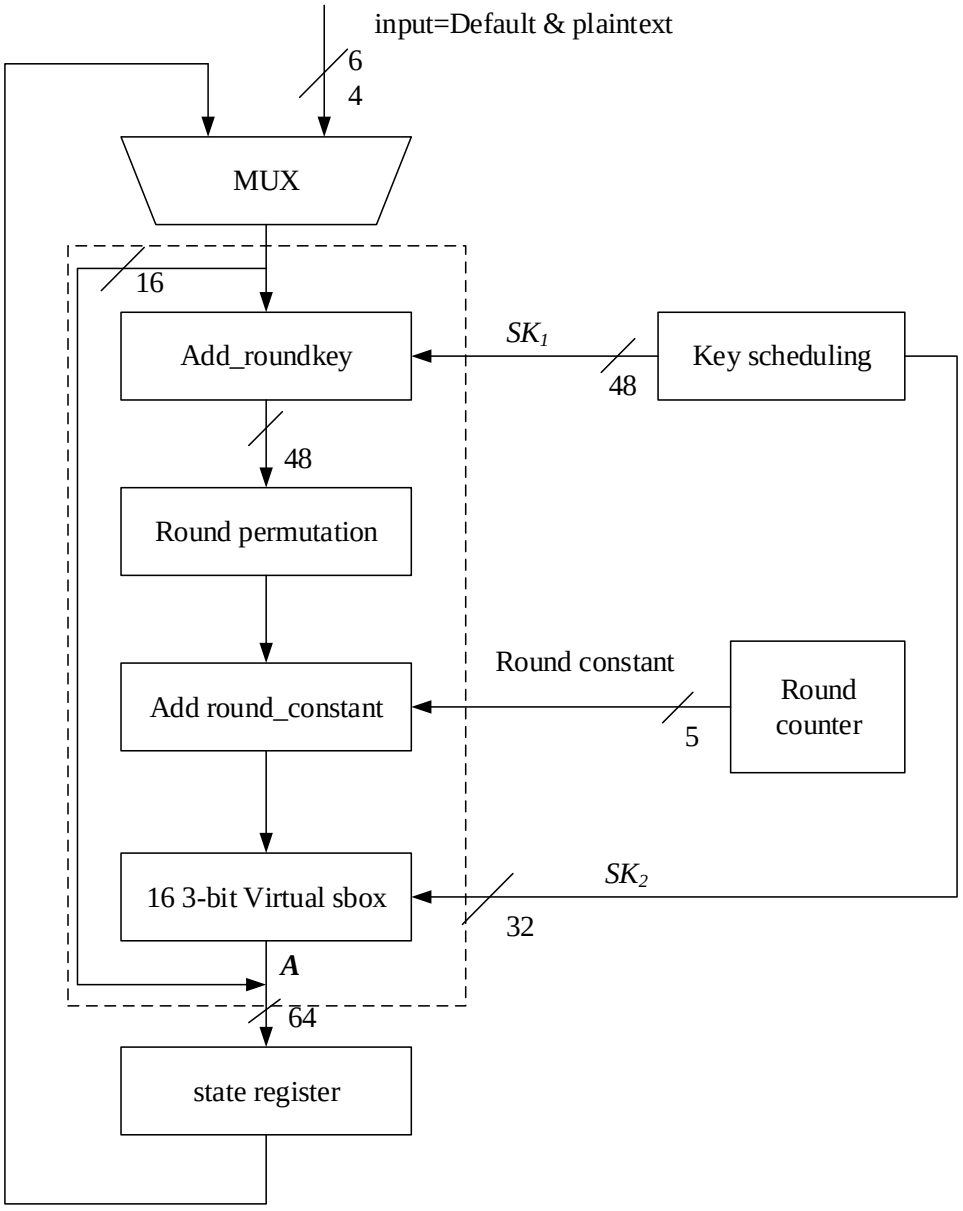


Figure 3.8 Hardware Structure of the Iterative Design of PRINTcipher

Figure 3.9 shows the simulation result of the iterative design of PRINTcipher, the simulation is also based on ModelSim and the test vector is obtained from [7]. From the waveforms, the corresponding ciphertext is same as the ciphertext provided by [7],

which means the design is successful.

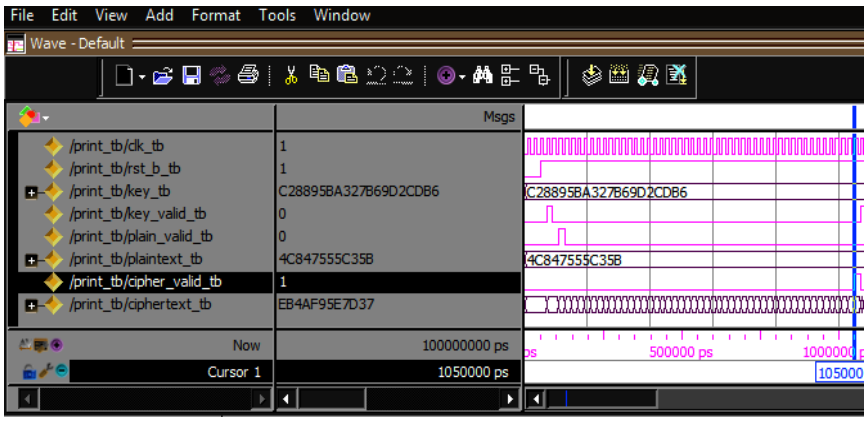


Figure 3.9 Simulation Result of Iterative Design of PRINTcipher

Table 3.3 Synthesis Result of the Iterative Design of PRINTcipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
PRINTcipher	516	143	100%
key scheduling	242	80	46.9%

Table 3.3 shows the synthesis result of PRINTcipher. The result is based on Altera Cyclone IV FPGA and uses Quartus II as the synthesis tool. Compared to the PRESENT cipher and the Piccolo cipher, PRINTcipher not only consumes a smaller number of combinational functions, but also consumes fewer dedicated logic registers. In fact, after investigating the reports in Quartus II and viewing the resource utilization by entity, we find that the state register in PRINTcipher only consumes 48 dedicated logic

registers while in VHDL code, a 64 bit state register is applied. Actually, since the 16 most-significant bits in the state register are never used and all set to ‘0’, the synthesis tool optimizes the design by using a 48 bit register.

3.1.5 Iterative Design of LED Cipher

Figure 3.10 provides the hardware structure of the iterative design of the LED cipher. Since the “Add_roundkey” is not applied to every round of the LED cipher, “MUX2” is required to choose the input to the “Add_constant” component between the output from “MUX1” or “Add_roundkey” component. The rest of the combinational datapath is basically straightforward. An 80 bit key size is chosen, the round key used for each “Add_roundkey” component is decided by the number of current step. Actually, a 64 bit left shift for the key register each time after the “Add_roundkey” component is applied, since the new 64 most-significant bit inside the key register is the correct choice for the next “Add_roundkey” process.

The FSM of this design is slightly different from the previous three design. Since the “Add_roundkey” component should be applied to the datapath every four rounds, the round counter not only outputs a “done” signal but also outputs the “cnt” signal which indicates the number of rounds to the FSM. In this way, the FSM knows the correct round to determine when the select signal for “MUX2” should be inverted to apply the “Add_roundkey” component. As a Mealy state machine is applied, the state transition diagram is similar to the previous three designs. The difference is during the

"ENCRYPTION" state, the selection signal for "MUX2" is inverted when the "Add_roundkey" component is applied.

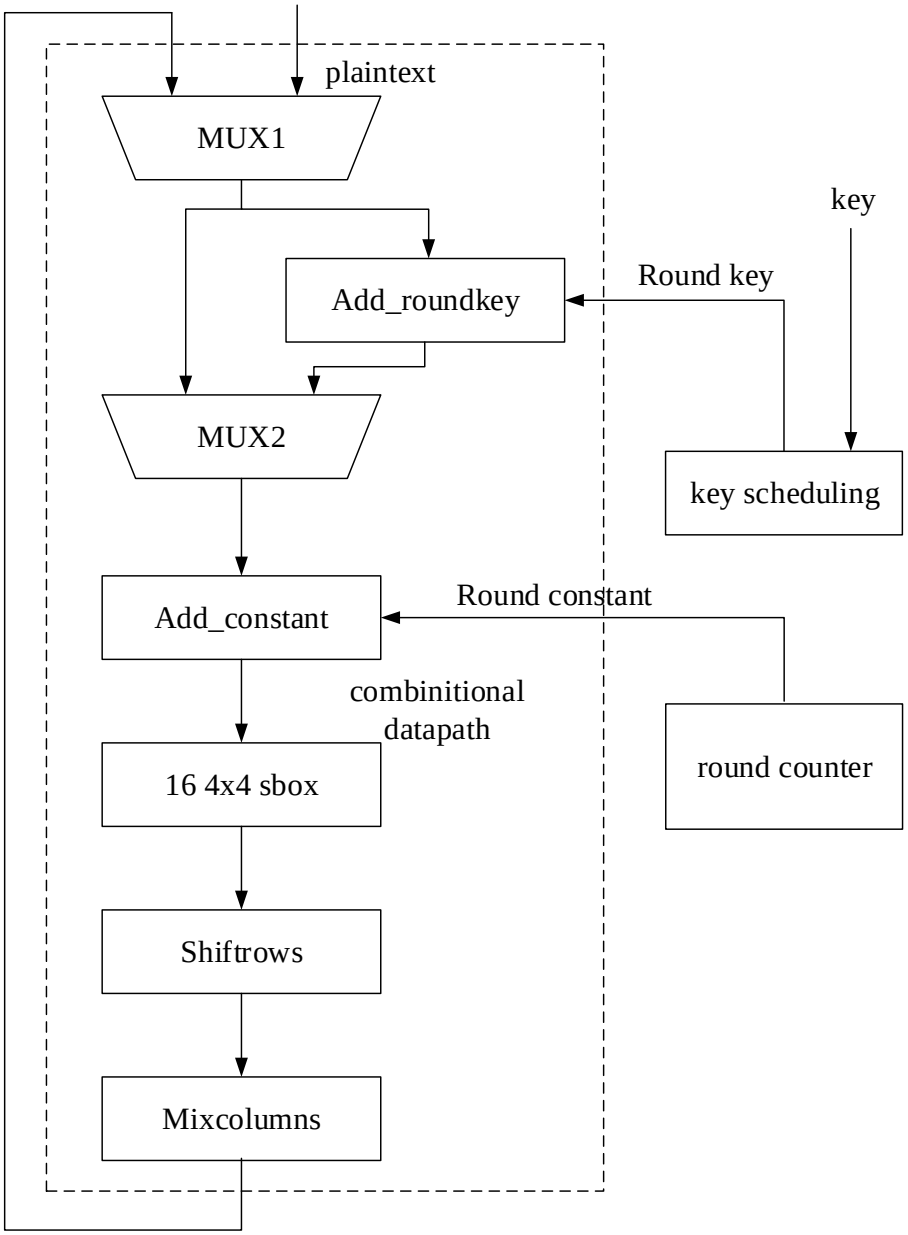


Figure 3.10 Hardware Structure of the Iterative Design of LED Cipher

The "Mixcolumn" component here uses the matrix *M* since an iterative design is used here. The calculation for the output of the "Mixcolumn" is complex and the VHDL code is found in Appendix A.2.

Figure 3.11 provides the simulation results of LED cipher. This waveform is also based on ModelSim. The test vectors are obtained from [25]. The simulation result indicates that the output ciphertext is correct with the corresponding input key and ciphertext.

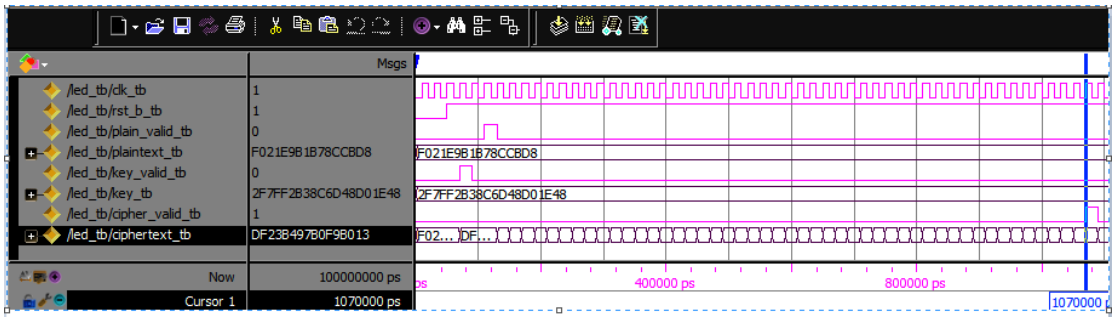


Figure 3.11 Simulation Result of the Iterative Design of LED Cipher

Table 3.4 Synthesis Result of the Iterative Design of LED Cipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
LED	821	160	100%
key scheduling	404	80	49.2%

Table 3.4 shows the synthesis result of the iterative design of the LED cipher. Compared to [22], the total number of combinational functions is reduced. Actually, in our design, the structure of the key scheduling component is optimized. In [22], the round counter is used to choose different bits inside the key register to form the round

key, while in this design, a simple 64 bit left shift is applied once after each "Add_roundkey" component is used in the datapath.

3.2 Iterative Design of the Multi-cipher Platform

Figure 3.12 provides the block diagram of the multi-cipher platform. As different ciphers have different output lines, we use four different signals "PRESENT_valid", "LED_valid", "Piccolo_valid" and "PRINT_valid" to be the valid signal of the four different output ciphertexts, respectively. For PRINTcipher, the output ciphertext "PRINT" is 64 bits with the 16 most-significant bits as the default value of all '0'.

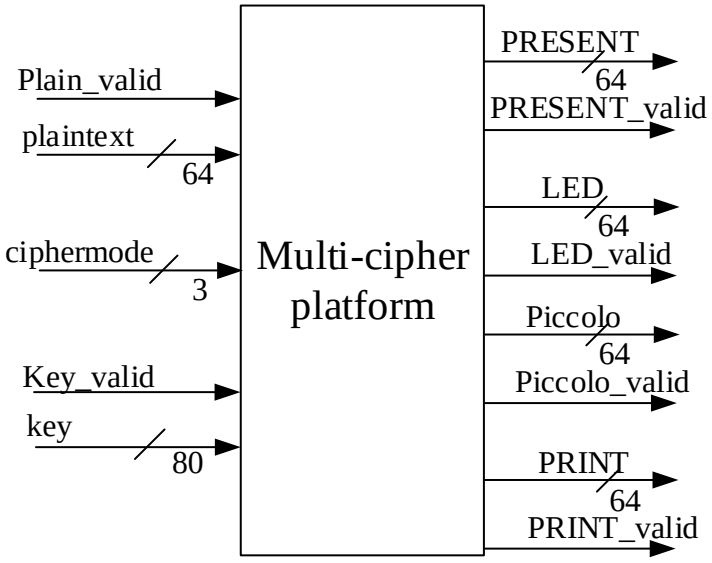


Figure 3.12 Block Diagram of the Multi-cipher Platform

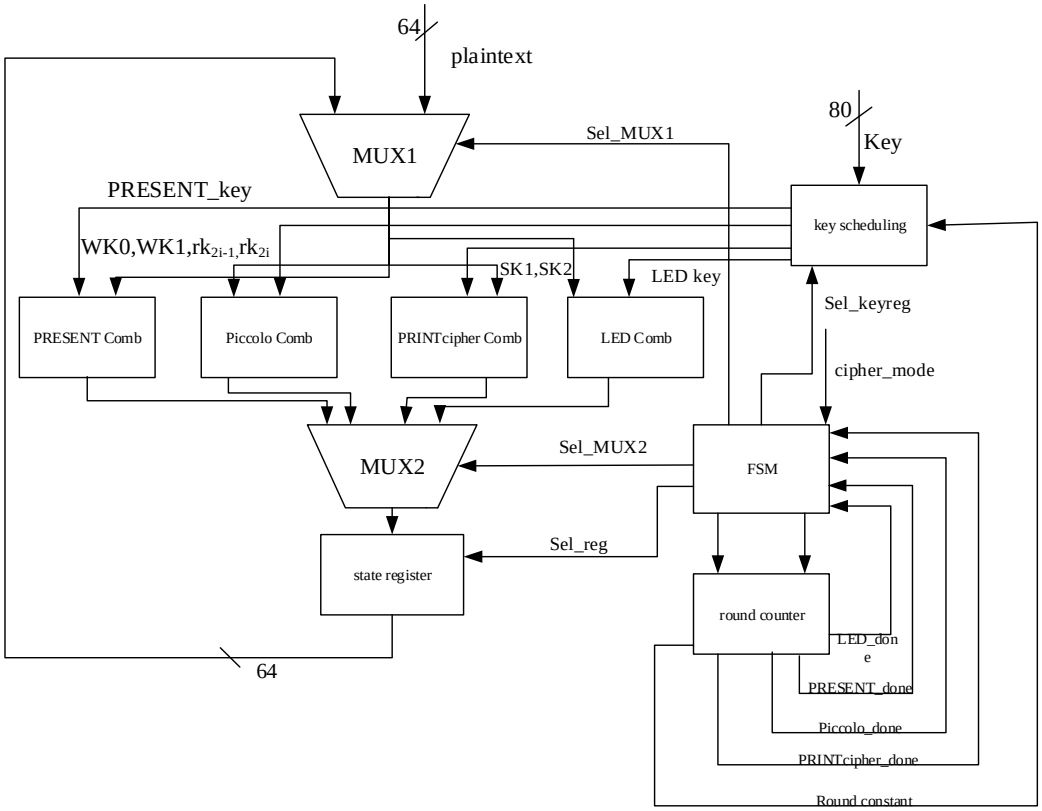


Figure 3.13 Hardware Structure of the Iterative Design of the Multi-cipher Platform

The purpose of our design is to integrate PRESENT, Piccolo, PRINTcipher and LED into one platform. Figure 3.13 provides the hardware structure of the iterative design of the platform. Compared to the previous design of each individual cipher, another 3-bit “cipher_mode” signal is used to indicate which cipher should be chosen in the next encryption process. The coding of the “cipher_mode” signal is shown in Table 3.5. Actually, since a 3-bit signal can represent 8 different values, we can use this signal to indicate 7 different ciphers with the last value representing NULL. In this way, we can add 3 more ciphers inside our platform. The components “PRESENT comb”, “Piccolo comb”, “PRINTcipher comb” and “LED comb” are the combinational datapaths of PRESENT, Piccolo, PRINTcipher and LED, respectively. In Figure 3.2,

Error! Reference source not found., Figure 3.8 and Figure 3.10, the datapath included in the dashed rectangle is the structure of the combinational datapath for each cipher. Two different multiplexers are used in this design. Multiplexer “MUX1” is used to select the input from the plaintext or the state register, while “MUX2” is used to load the correct output of the combinational datapaths into the state register. The round counter is a counter which is used to indicate the end of the encryption process. It also generates the round constant which is used in the key scheduling algorithm.

Table 3.5 The Coding of "cipher_mode" Signal

cipher_mode	**chosen cipher**
000	NULL
001	PRESENT
010	Piccolo
011	PRINTcipher
100	LED
101-111	extensions

To decrease the area and resources consumed by this platform, we have tried to share some similar components with different ciphers. For example, the

“Add_roundkey” process for PRESENT and Piccolo are almost the same, so we only use a 64 bit XOR for these two ciphers. However, in this way, we need to use an additional 64 bit 2-to-1 multiplexer and after examining the synthesis result, we found that this approach is not worthwhile. In our final design, we have decided that our approach to save the area is based on sharing the state register within the cipher datapath and the key register inside the key schedule block. The penalty of this structure is that we need to use one more 64 bit 4-to-1 multiplexer to choose the correct data to be loaded into the state register.

Before the encryption process, the “cipher_mode” signal must be loaded into the FSM to choose the cipher that would be used in the encryption process. Since the four ciphers need a different number of rounds to finish the encryption process, we need four different “done” signals from round counter component to indicate the end of the encryption process. The block size of PRINTcipher is 48 bits while PRESENT, Piccolo and LED have a 64 bit block. If we choose to use the PRINTcipher, the 16 most-significant bits of output will be set to default value of 0.

Figure 3.14 provides the state transition diagram of the platform. The following steps introduce how the platform works:

1. Before the “cipher_mode” signal is changed to any valid value for one of the four ciphers, the FSM will stay in “IDLE” state.

2. After the “cipher_mode” signal is loaded into the platform, the FSM will step into

“PRESENT_IDLE”, “Piccolo_IDLE”, “PRINT_IDLE” or “LED_IDLE” based on the value of “cipher_mode” signal. For example, if “cipher_mode” = “001”, then the FSM will step into “PRESENT_IDLE” state.

3. In “PRESENT_IDLE”, “Piccolo_IDLE”, “PRINT_IDLE” or “LED_IDLE” state, the FSM will wait for the key that will be used in the following encryption process. After the key is loaded into the platform, The FSM will step into “PRESENT_waitplain”, “Piccolo_waitplain”, “PRINT_waitplian” or “LED_waitplain” state and wait for the plaintext that needs to be encrypted.

4. After the plaintext is loaded into the platform, the FSM will step into the “PRESENT_encryption”, “Piccolo_encryption”, “PRINT_encryption” or “LED_encryption” state and start the encryption process.

5. In “PRESENT_encryption”, “Piccolo_encryption”, “PRINT_encryption” or “LED_encryption”, when the “done” signal indicates that the encryption process is over, the FSM will step into “IDLE” or one of the four “cipher_IDLE” states based on the value of “cipher_mode” signal. The four ciphers have separate control paths, which is used to ensure that each cipher is not influenced by another.

Moreover, to ensure that the whole system works smoothly, we also have a more robust design for the following cases:

1. The FSM will stay in “IDLE” state if no “cipher_mode” signals have been loaded into it. Any plaintexts and keys will be ignored.

2. During the encryption process, any “cipher_mode” signals, keys, and plaintexts will be ignored.

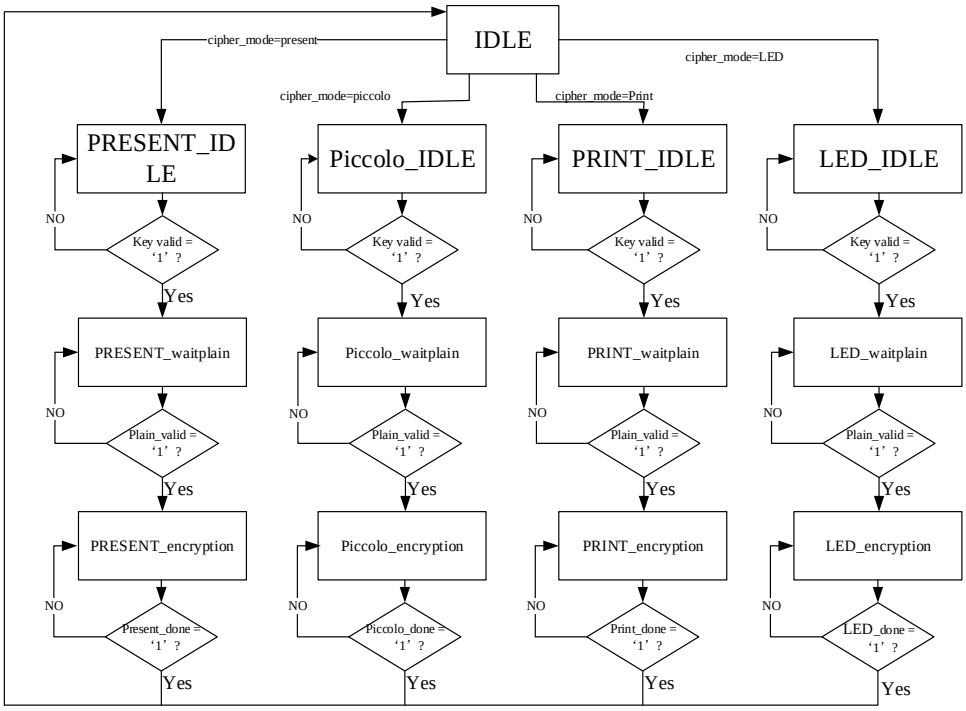


Figure 3.14 State Transition Diagram of the Iterative Design of Multi-cipher Platform

Figure 3.15 shows the simulation results of the platform. The whole simulation is based on ModelSim [23]. According to the design of this platform, only one cipher could be executed at a specific time inside the platform. The waveforms in Figure 3.15 are based on this principle. First, the testbench sets “cipher_mode” signal to “001” which means the next cipher should be the PRESENT cipher and loads the input key and plaintext to the platform. Then, “cipher_mode” signal is set to “010” which indicates Piccolo cipher is chosen for the next encryption process after we get the ciphertext of PRESENT cipher. For PRINTcipher and LED cipher, the same philosophy is used. Comparing Figure 3.15 to the simulation results of each individual cipher, we

can conclude that the design is successful.

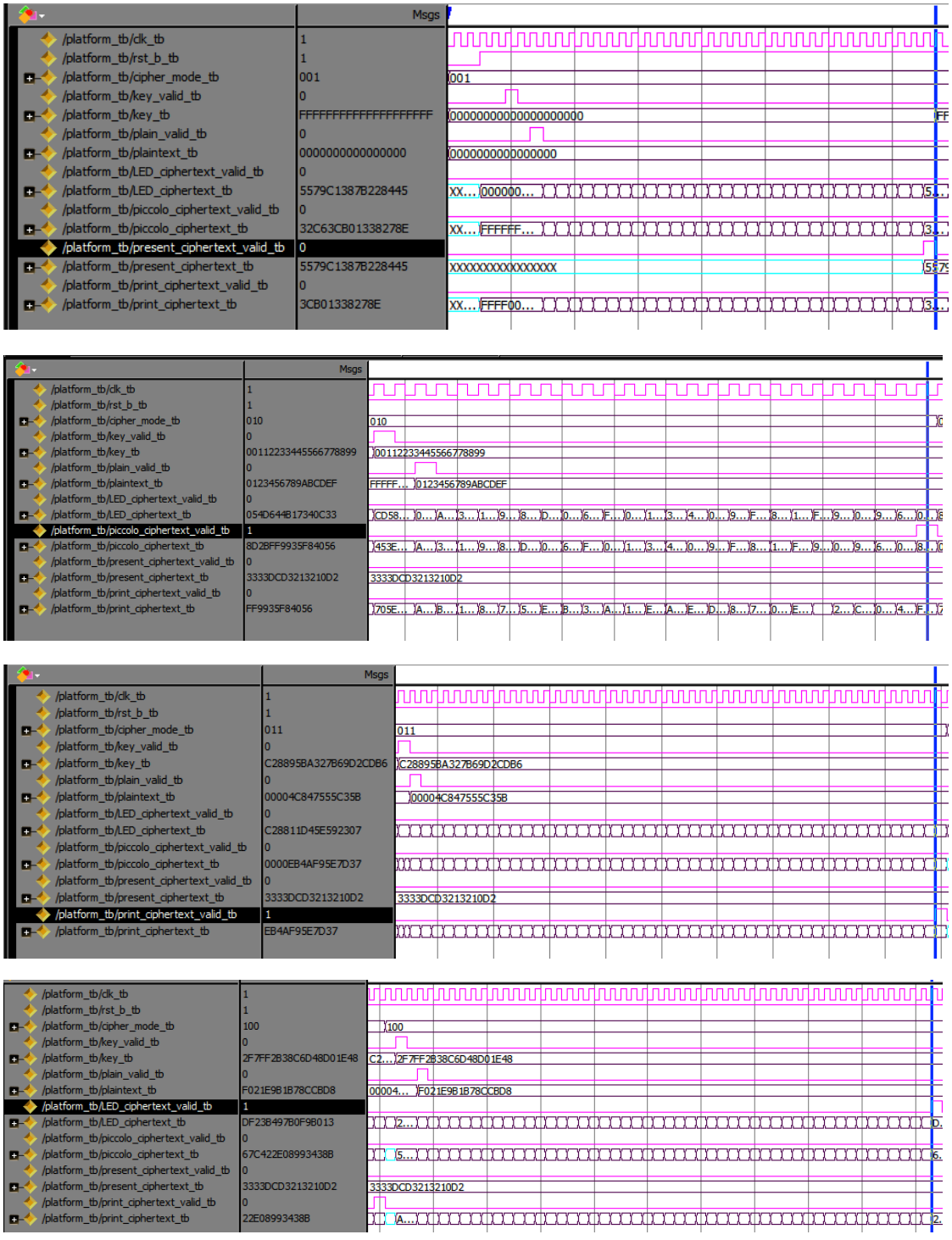


Figure 3.15 Simulation Result of the Iterative Design of Platform

Table 3.6 Resources Usage Comparison for Different Ciphers and Platform

Cipher	Combination functions	Dedicated logic registers
PRESENT	613	153
Piccolo	809	153
PRINTcipher	516	143
LED	821	160
Total	**2759**	**609**
Platform	**1864**	**172**

Table 3.6 presents the usage summary of the multi-cipher platform compared to the individual cipher implementation. From Table 3.6 it is clear that the platform saves a lot of resources, not only in combinational functions, but also in dedicated logic registers. The total combinational functions consumed by these four ciphers should be 2759, while the platform only consumes 1864 combinational functions. We save 32.4% combinational functions for the platform. This results from simplification that occurs in the key scheduling component when the four ciphers are combined. In fact, we find that the key scheduling algorithms consume a large amount of combinational functions since the key scheduling algorithms of these four ciphers are complex. Table 3.7 shows the number of combinational functions for the key scheduling consumed by the four

ciphers and our platform. From the table, it is clear why the total combinational functions of our platform is smaller even though we add an extra 64 bit 4-to-1 multiplexer in the platform. Moreover, the number of dedicated logic registers is significantly reduced by sharing the state register and key register. The total number of dedicated logic registers consumed by these four ciphers should be 609, while the platform only consumes 173 dedicated logic registers. Compared to [22], the total combinational functions for the platform and LED is also reduced due to the optimization in the key scheduling design of the LED cipher.

However, the drawbacks for this design are also obvious. First, the performance of this design is not very high compared to each individual cipher and other high speed implementations. In Table 3.8, we examine the performance of our system by presenting the resulting throughput of the 4 ciphers using our platform, as determined by the synthesis tools. The data of throughput is based on the maximum frequency of our platform being 224.7 MHz. The throughput of the four ciphers are different from each other since they need different numbers of rounds to finish the encryption process. The number of clock cycles that are needed to finish the encryption process are 32, 25, 48 and 48 for PRESENT, Piccolo, PRINTcipher and LED, respectively, after the plaintext is loaded into our platform.

Furthermore, for this iterative design, a large number of I/O pins are needed. For each individual cipher, 213 I/O pins (64 for the input plaintext, 64 for the output ciphertext, 80 for the input key, 3 for data valid signal, 1 for clock and 1 for reset) are

needed. In the platform, 395 pins are required, since in the platform, different ciphers generate different ciphertexts at different positions. Should we want to use same pins for the output ciphertext of different ciphers, some combinational functions are required in the top level entity to choose the output from different ciphers, and this will increase the amount of hardware resources.

Table 3.7 Number of Combinational Functions of Key Scheduling

Cipher	**Comb functions of Key Scheduling**	**Percentage of Comb functions**
PRESENT	328	53.5%
Piccolo	431	53.2%
PRINTcipher	242	46.9%
LED	404	49.2%
Total	**1405**	**50.9%**
Platform	**515**	**27.6%**

If should be noted that high throughput is usually not a requirement in lightweight applications since area and resources consumption are the only thing that matters. In this way, in order to save more area and use fewer pins, a more compact design is

required. A serialized design is considered to be a better way to save the hardware resource even though the throughput may be decreased since fewer bits are processed during one clock cycle.

Table 3.8 Performance of the Iterative Implementation

Cipher	**Throughput**
PRESENT	448 Mbps
Piccolo	573 Mbps
PRINTcipher	224 Mbps
LED	299 Mbps

3.3 Summary

In this chapter, we present the details of the iterative design of each individual cipher and the multi-cipher platform. In addition, we also compare the synthesis results of each individual cipher and the platform. The synthesis result shows that our design successfully reduces the total number of combinational functions and dedicated logic registers and hence, reduces the number of logic elements required for the design.

In the next chapter, we will present a more compact serialized design. Similar to this chapter, we will first present the serialized design of each cipher and then provide the details of the serialized design of the multi-cipher platform. In addition, the

comparison and analysis of the resource consumption result will be provided.

Chapter 4
Serialized Design of Individual Ciphers and the Platform

In this chapter, a serialized design for each individual cipher and the multi-cipher platform is provided. Compared to the iterative design, the serialized design is more compact. For PRESENT, Piccolo and LED cipher, a 4 bit datapath is used, while for PRINTcipher, a 3 bit datapath is used for the virtual sbox and 4 bit datapath is applied for the “Add_roundkey” layer. Since we use a smaller datapath, only 3 or 4 bits can be dealt with in one clock cycle which certainly leads to the reduction of throughput. However, also due to the smaller datapath, more hardware resources can be saved.

Before introducing the structure of the ciphers, here is some of the notation that will be used in this chapter:

Table 4.1 Notation in Chapter 4

Notation	**Meaning**
MUX(X,Y)	X inputs Y bit multiplexer
&	bit concatenation
cc counter	clock cycle counter

4.1 Serialized Design of Each Individual Ciphers

In this section, we provide the details of the serialized design of each individual cipher. The hardware structure, state transition diagram, simulation result and synthesis result will be provided and discussed.

4.1.1 Block Diagram

Figure 4.1 shows the block diagram for the serialized design of each individual cipher. Notice that the data length for plaintext, key and ciphertext is 4 bits. In this way, far fewer I/O pins are required compared to the iterative design. Besides, since the actual data length of ciphertext for PRESENT, Picclo and LED is 64 bits, while for PRINTcipher, a 48 bit output of ciphertext is used, in order to ensure that the design is consistent for each cipher, the most-significant bit of the output ciphertext of PRINTcipher is always set to '0', while the three least-significant bits are the valid bits of the ciphertext.

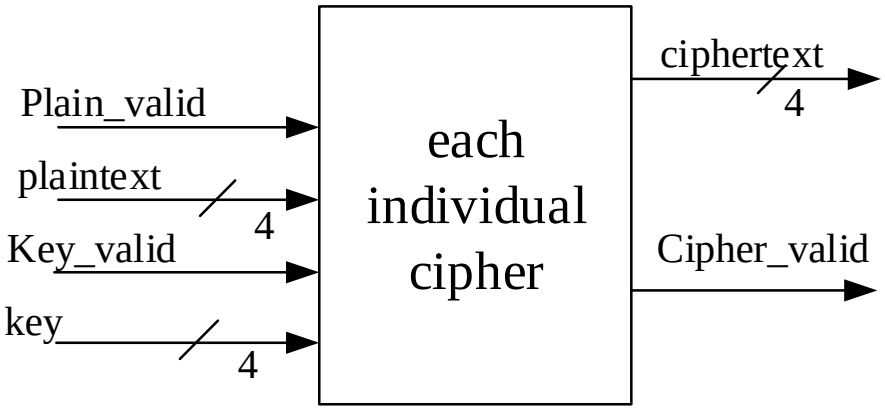


Figure 4.1 Block Diagram for Each Individual Cipher

In this design, we assume that the "Key_valid" signal will be asserted for a

consecutive 20 clock cycles to load the “key” and the “Plain_valid” signal will be asserted for a consecutive 16 clock cycles inside the 20 clock cycles pulse of “Key_valid” signal. All the designs discussed below are based on this assumption.

4.1.2 Serialized Design of PRESENT Cipher

Figure 4.2 provides the hardware structure of the serialized design of the PRESENT cipher which is taken from [26]. In this design all the three multiplexers used in this figure are MUX(2,4). Two counters are needed in this design even though they are not included in Figure 4.2. One of the counters is called the “cc counter”, while the other one is referred to as the round counter. The “cc counter” is a 4 bit counter which counts from 0 to 15 for each clock cycle and it is used to indicate if the “Add_roundkey” and sbox layer are finished. Actually, since we use a 4 bit datapath for these two layers, 16 clock cycles are needed to finish these two layers. The round counter is used to indicate the end of one entire encryption process.

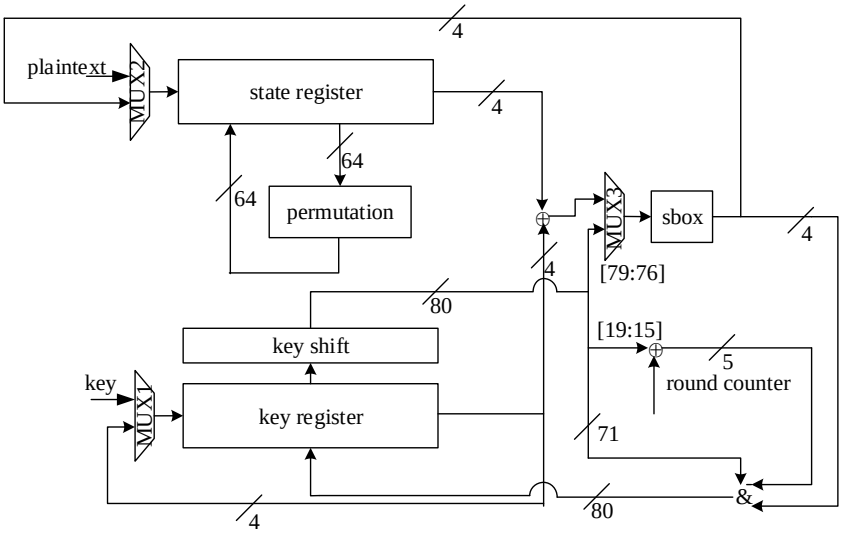


Figure 4.2 Hardware Structure of the Serialized Design of PRESENT Cipher [26]

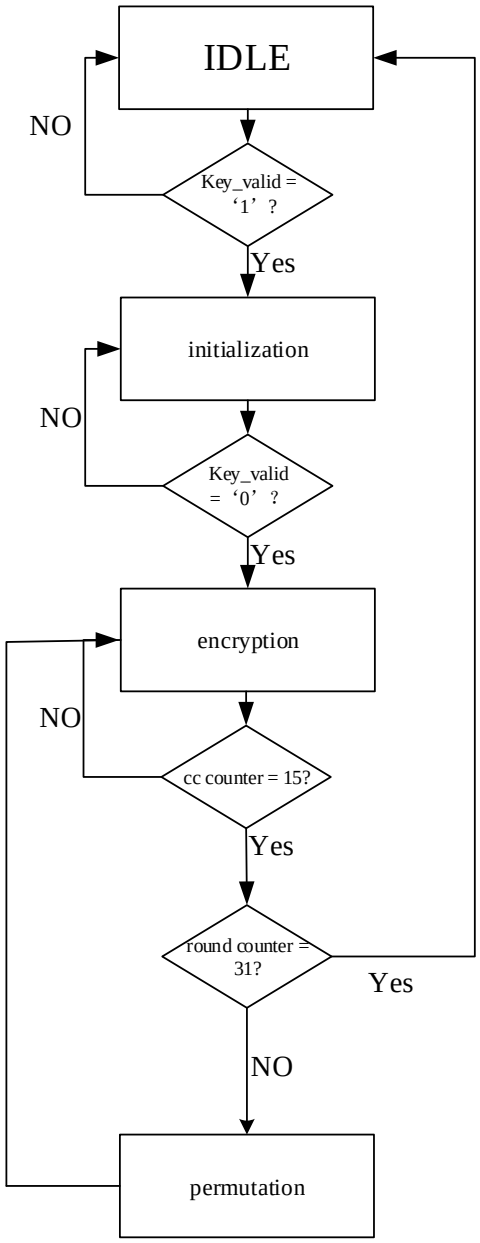


Figure 4.3 State Transition Diagram of the Serialized Design of PRESENT Cipher

Figure 4.3 provides the state transition diagram of this design. The details of this state transition diagram are introduced below:

1. The whole system begins with the "IDLE" state. After the "Key_valid" signal is asserted, the FSM steps into the "initialization" state, and according to the assumption at the beginning of this chapter, during the "initialization" state, the

"key" and "plaintext" will be loaded into the key register and state register respectively.

2. After the "Key_valid" signal is de-asserted, which means the "key" has been fully loaded and the "plaintext" must also be fully loaded according to the assumption at this same time, the FSM steps into the "encryption" state.

3. In the "encryption" state, the state register and the key register will left shift 4 bits every clock cycle. Also, the "cc counter" also starts to count. When the "cc counter" reaches 15 and the round counter has not reached 31, the FSM transfers to "permutation" state. If the "cc counter" reaches 15 and the round counter has reached 31, the FSM transfers to "IDLE" state.

4. In the "permutation" state, the round counter will increment in this state. The key register and the state register will parallel load the 80 bit input and 64 bit input, respectively. In fact, in this state, the key scheduling algorithm is performed. After one clock cycle, the FSM will perform an unconditional transfer to "encryption" state again.

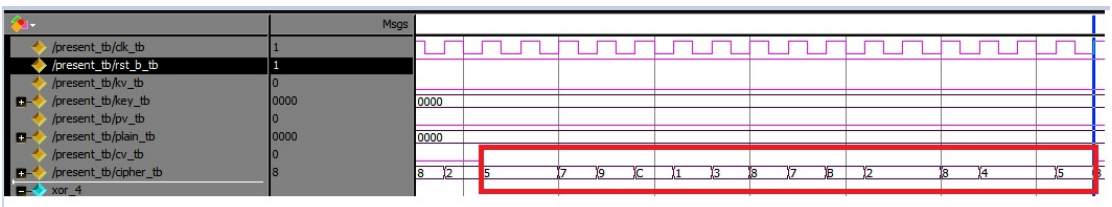


Figure 4.4 Simulation Result of the Serialized Design of PRESENT Cipher

Figure 4.4 provides the simulation result of this design. The signal "cv_tb" is the valid signal for the ciphertext. Comparing the results with **Error! Reference source**

not found., we can conclude that the design is successful.

Table 4.2 Summary of the Resource Consumption of the Serialized Design of PRESENT Cipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
PRESENT	206	158	100%
FSM	28	4	13.6%
key register and state register	155	144	75.2%

Table 4.2 provides the total resource consumption of this design. Compared to Table 3.1, the number of total combination functions consumed by the serialized implementation is only 206, about one-third of the number of total combinational functions consumed by the iterative design. However, the dedicated logic registers consumed by the serialized design is slightly increased due to the fact that we use another 4 bit “cc counter” inside the design.

Figure 4.5 provides a typical structure of the registers used in all of our designs. Usually, if we need more modes of operations, we need to increase the inputs to the “MUX” which will result in an increase in the number of combinational functions

consumed by the registers.

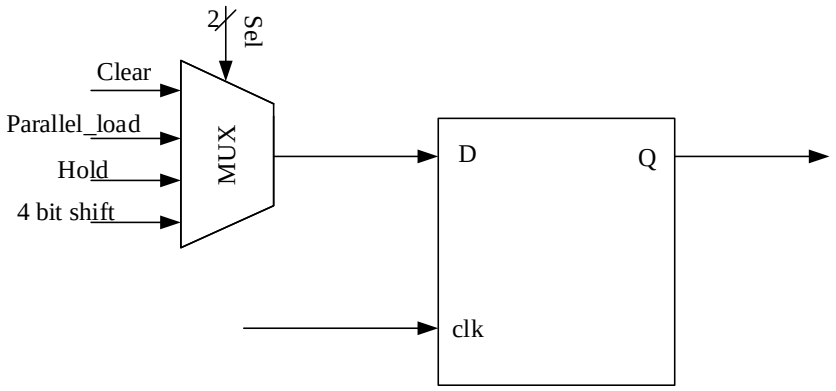


Figure 4.5 1-bit of the Structure of the Registers with 4 Modes of Operation

Since the key register and state register in PRESENT need another operation which is 4 bit left shift in the serialized design and the size of the datapath is reduced to 4 bits, the number of combinational functions consumed by the registers inside this design is increased while the number of combinational functions consumed by the datapath is reduced, which leads to the result that the percentage of the combinational functions consumed by the key register and state register is as high as 75%.

4.1.3 Serialized Design of Piccolo Cipher

Figure 4.6 introduces the serialized design of the Piccolo cipher. Multiplexer "MUX1" is a MUX(2,4) which is used to select the two different inputs to the sbox since the F function has two layers of sbox. Multiplexer "MUX3" is a MUX(4,16) with the purpose to select different whitening keys and a MUX(2,16), MUX6, is used to select different round keys. Multiplexers "MUX2", "MUX4" and "MUX5" are all MUX(4,4). Counter "cnt_2" and "cnt_4" are two "cc counters" and "cnt_6" is the round

counter. The state register has more different modes of operations as listed in Table 4.3.

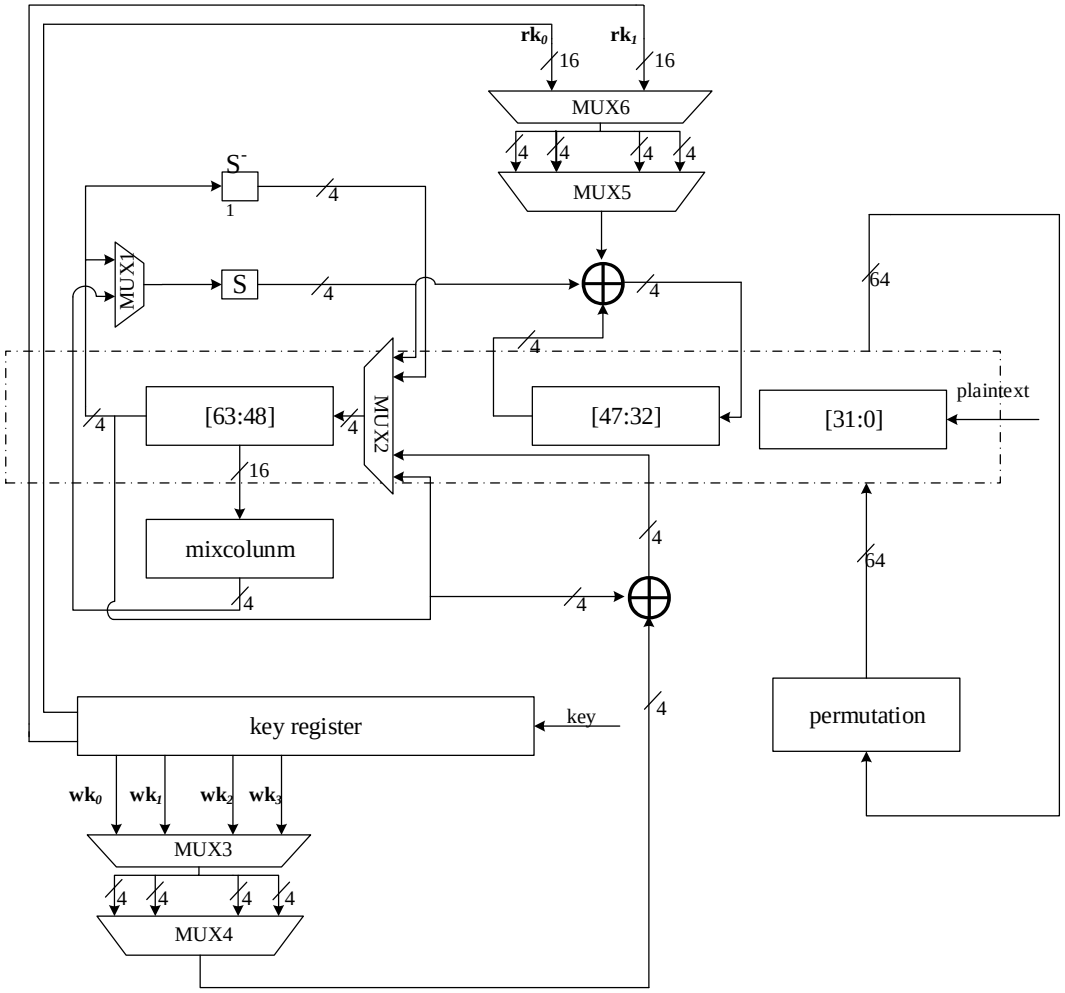


Figure 4.6 Hardware Structure of the Serialized Design of Piccolo Cipher

The reason that we design the state register in this way is we need to deal with the F function in the Piccolo cipher. The main challenge for this design is how to deal with the F function in Piccolo cipher. Since the F function deals with a 16 bit data block, we require extra registers to store the intermediate value of the F function if we use a 4 bit datapath. In [9], a brief introduction of a smart approach has been provided about how to deal with this issue.

Table 4.3 Mode of Operations in Piccolo State Register

Mode	**Operation**
1	Hold
2	32 most-significant bits swapped with the 32 least-significant bits
3	4 bit left shift to all the 64 bits with the 4 bits input shift to the 4 least-significant bits.
4	4 bit left shift to the 16 most-significant bits with the 4 bits input shift to 51^{st} bit to 48^{th} bit, while the rest is hold.
5	4 bit left shift to the 16 most significant bits and 47^{th} bit to 32^{nd} bit with two different inputs shift to 51^{st} bit to 48^{th} bit and 35^{th} to 32^{nd} respectively, while the rest is hold.
6	Parallel load.

First, let us take a look at Figure 2.3. We assume that the output of the first sbox layer inside the F function is a vector $S_1 = [A\ B\ C\ D]^T$, and the output of the diffusion matrix is $S_2 = [a\ b\ c\ d]^T$. Then we perform the calculation of the diffusion matrix and we can get the result that :

$$S_2 = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = M * S_1 = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} * \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 2A + 3B + C + D \\ A + 2B + 3C + D \\ A + B + 2C + 3D \\ 3A + B + C + 2D \end{pmatrix}$$

This operation can be easily achieved by a shift register as shown in Figure 4.6. If we can left shift 4 bits for the 16 most-significant bits every clock cycle, then we can use a "Mixcolumn" component which only deals with 16 bits each time. The details of this operation will be discussed in the state transition diagram.

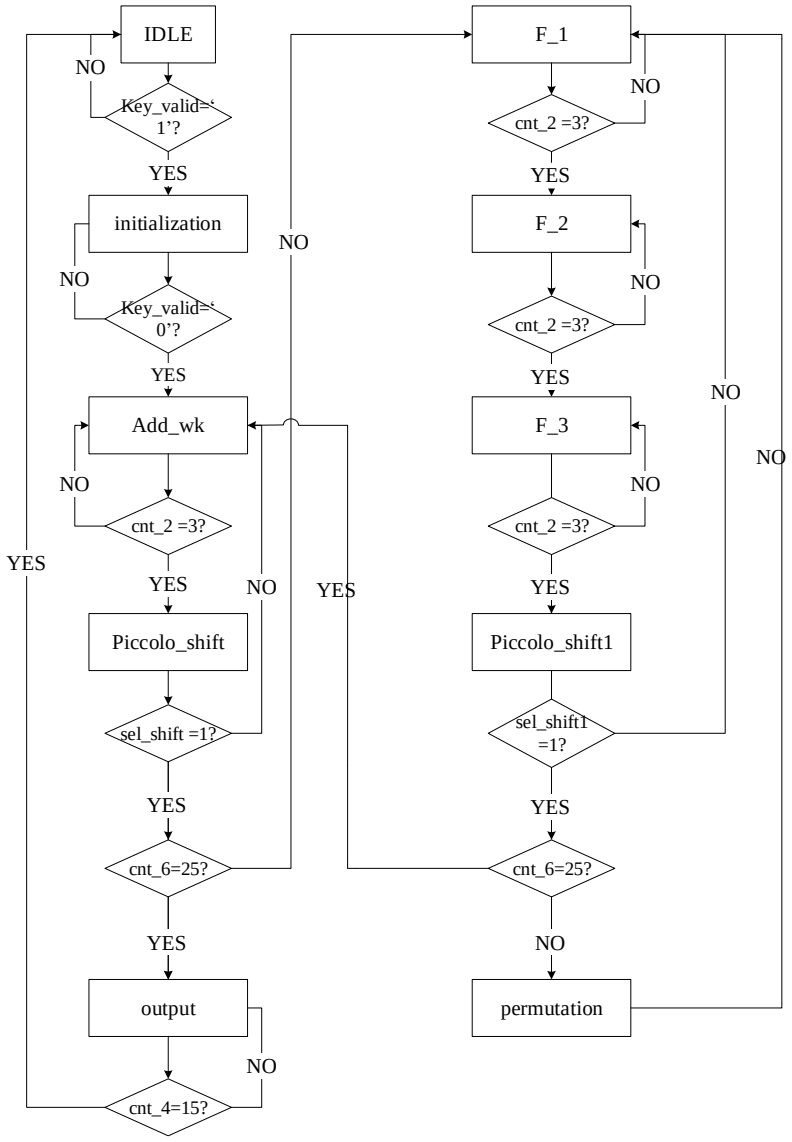


Figure 4.7 State Transition Diagram of the Serialized Design of Piccolo Cipher

Figure 4.7 provides the state transition diagram of the serialized design of Piccolo cipher. The whole system starts with the “IDLE” state and since the “initialization” state is exactly the same as the serialized design of PRESENT cipher, we describe the details of this design from the “Add_wk” state.

1. In the “Add_wk” state, the selection signal for “MUX4” will be incremented by one for every clock cycle. In this way, the 16 most-significant bits of the state register can “XOR” with the correct bits of the whitening key. “MUX2” will select the input from the output of the 2 input “XOR” and the state register will choose mode 4. Since we need to step into the “Add_wk” state twice for the first and last round, at the fourth clock cycle of “Add_wk” state, a control signal “sel_shift” with a initialized value of ‘0’ will be inverted to indicate that if we have already stepped into the “Add_wk” state twice. After 4 clock cycles, the “cc counter” “cnt_2” will reach a value of 3, and then the FSM transfers to “Piccolo_shift” state.

2. In the “Piccolo_shift” state, the state register will choose mode 2 to switch the bit position of the 32 most-significant bits and the 32 least-significant bits. This state will only last for 1 clock cycle and the system will step into the next state according to different conditions

- If “sel_shift” signal is ‘1’, which means the “Add_wk” has only been done once, then next state should be “Add_wk”.

- If “sel_shift” signal is’0’, and the round counter “cnt_6” is 25, which means the

post-whitening has been finished, then the next state should be “output” state.

- If “sel_shift” signal is ‘0’ and the round counter does not reach 25, which means “Add_wk” state has been finished twice and we should start a round operation, then the next state is “F_1” state.

3. In the “F_1” state, this is the first step to deal with the “F” function inside the Piccolo cipher. In this state, the main purpose is to add the first sbox layer to the 16 most-significant bit of the state register. “MUX4” will choose the output of the sbox and the state register is in mode 4. After this state, the 16 most-significant bits are the vector we mentioned $S_1 = [A\ B\ C\ D]^T$. This state will last for 4 clock cycles and the “cc counter” “cnt_2” will indicate the end of this state, resulting in a transfer to “F_2” state.

4. In the “F_2” state, “MUX4” will choose the left shift output of the state register and the state register will be in mode 5. This state will also last for 4 clock cycles. After the first clock cycle inside this state, the 16 most-significant bits in the state register can be represented as $S_1 = [B\ C\ D\ A]^T$, and then the input of the sbox in the second clock cycle during the “F_2” state is $[2B+3C+D+A]$ which is just the correct input of the sbox for the second 4 bits of the output of the F function. At the same time, the output of the sbox will “XOR” with the round key and the 47^{th} to 32^{nd} bits of state register. After 4 clock cycles, the FSM transfers to “F_3” state and at this time the 16 most-significant bits in the state register are still $[B\ C\ D\ A]^T$.

5. In the “F_3” state, the state register will be in mode 4. This state also lasts for 4

clock cycles. The purpose of this state is to recover the 16 most-significant bits before applying them to the first sbox layer of the state register. “MUX2” will choose the input from the output of the “S^{-1}” component. The “S^{-1}” component is an inverse sbox. In the fourth clock cycle of “F_3” state, another control signal “sel_shift1” with a initialized value of ‘0’ will be inverted. After 4 clock cycles, the FSM transfers to “Piccolo_shift1” state.

6. In the “Piccolo_shift1” state, the state register will be in mode 2. Since the next state decode logic is different from “Piccolo_shift” state, we need two different states. This state will also only last for 1 clock cycle and steps into the next state according to different conditions:

- If “sel_shift1” signal is ‘1’, which means the F function has only been done once, then next state should be “F_1”.
- If “sel_shift1” signal is ‘0’, and the round counter “cnt_6” is 25, which means all the rounds have been finished, then we need to step into “Add_wk” state to finish the post-whitening.
- If “sel_shift1” signal is ‘0’ and the round counter has not reached 25, which means the F function has been completed twice and we should start the permutation of the round, then the next state is “permutation” state.

7. In the “permutation” state, the state register will be mode 6 and parallel load the output of the “permutation” component. This state lasts for only one clock cycle and

the system will subsequently step into the “F_1” state.

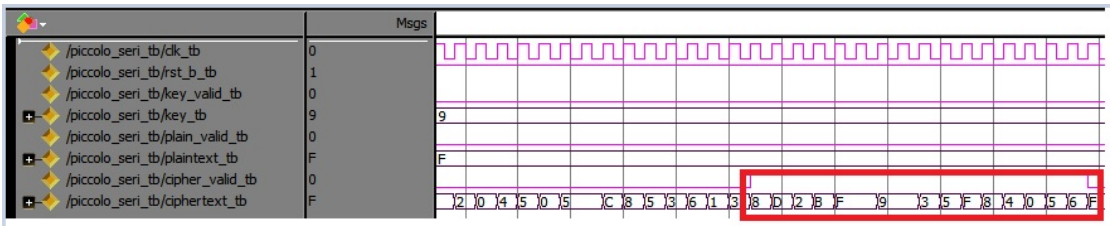


Figure 4.8 Simulation Result of the Serialized Design of Piccolo

Figure 4.8 provides the simulation result of the serialized design of Piccolo cipher. When the “cipher_valid_tb” signal is asserted, we can get the ciphertext as 8D2BFF9935F84056. We can conclude that the design is successful after comparing the result with the test vectors provided in [9].

Table 4.4 Summary of the Resource Consumption of the Serialized Design of Piccolo

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
Piccolo	372	164	100%
FSM	80	11	21.5%
key register and state register	197	144	53.0%

Table 4.4 provides the result of resource consumption of the serialized design of Piccolo. Compared to Table 3.2, the total number of combinational functions is

significantly reduced by 434. However, the percentage of the reduction of the combinational functions is not as large as the result of the serialized design of PRESENT. Actually, by investigating the resource utilization by entity in Quartus II [20], we find that the FSM consumes 80 combinational functions, which is far more than the FSM in the iterative design of Piccolo cipher which consumes only 22 combinational functions.

Although the complexity of the FSM increased, the percentage of the combinational functions consumed by the key register and state register is slightly reduced to 53% compared to the serialized design of PRESENT cipher.

4.1.4 Serialized Design of PRINTcipher

Figure 4.9 shows the structure of the serialized design of PRINTcipher. Since PRINTcipher uses a 3 bit sbox and the keyed permutation needs a 2 bit key, the state register needs to left shift 3 bits out and the key register needs to left shift 2 bits out. Both the multiplexers used in this design are MUX(2,4). Compared to the Piccolo cipher, this structure is basically straightforward. The only issue inside this design is to deal with the 16 bits default value of all '0'. In order to keep consistent to other ciphers, the ciphertext of PRINTcipher will also be 4 bits per clock cycle, and the "cipher_valid" signal will also be asserted for 16 clock cycles. As the last component in one round of PRINTcipher is the "Virtual sbox" and it deals with 3 bits per clock cycle for 16 clock cycles, we add a default '0' to the most-significant bit of the output ciphertext every

clock cycle. The details of this structure will be discussed in the following state transition diagram.

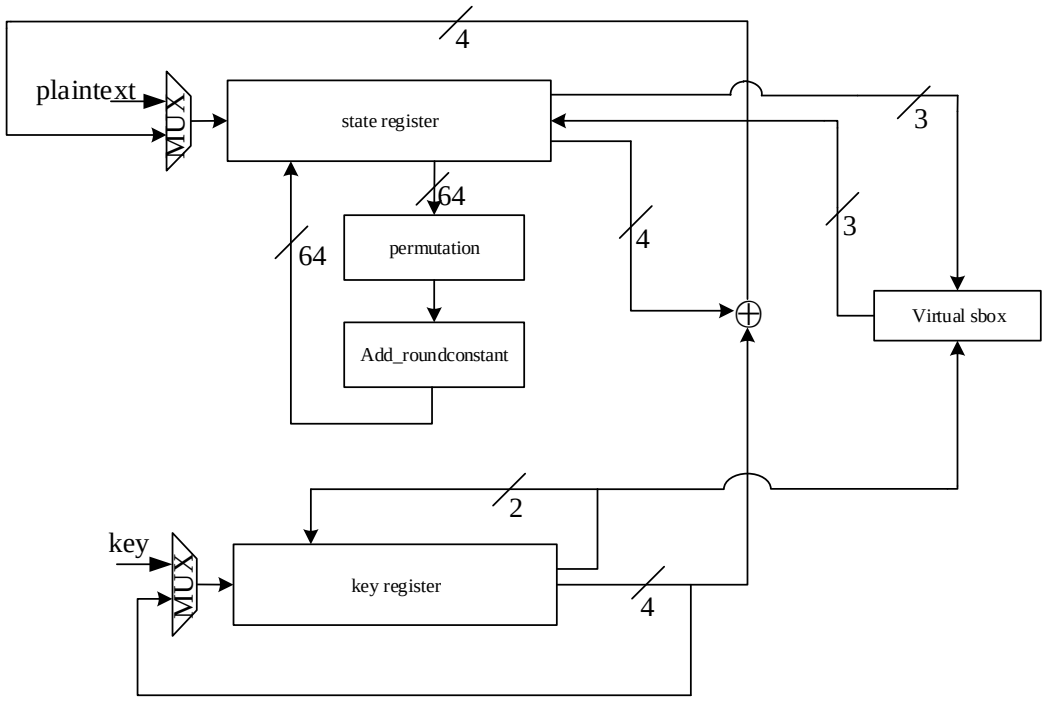


Figure 4.9 Hardware Structure of the Serialized Design of PRINTcipher

Figure 4.10 presents the state transition diagram of our design. We need five different states to finish an encryption cycle for the serialized implementation of PRINTcipher. The FSM also starts from “IDLE” state, and transfers to the “initialization” state when the “Key_valid” signal is asserted. The operations inside the “initialization” state are exactly the same as the previous two designs. The only difference is that for PRINTcipher, since the block size is 48 bits, only data in the first 12 clock cycles in the initialization state contains the valid block values and the last 4 clock cycles are all default ‘0’s.

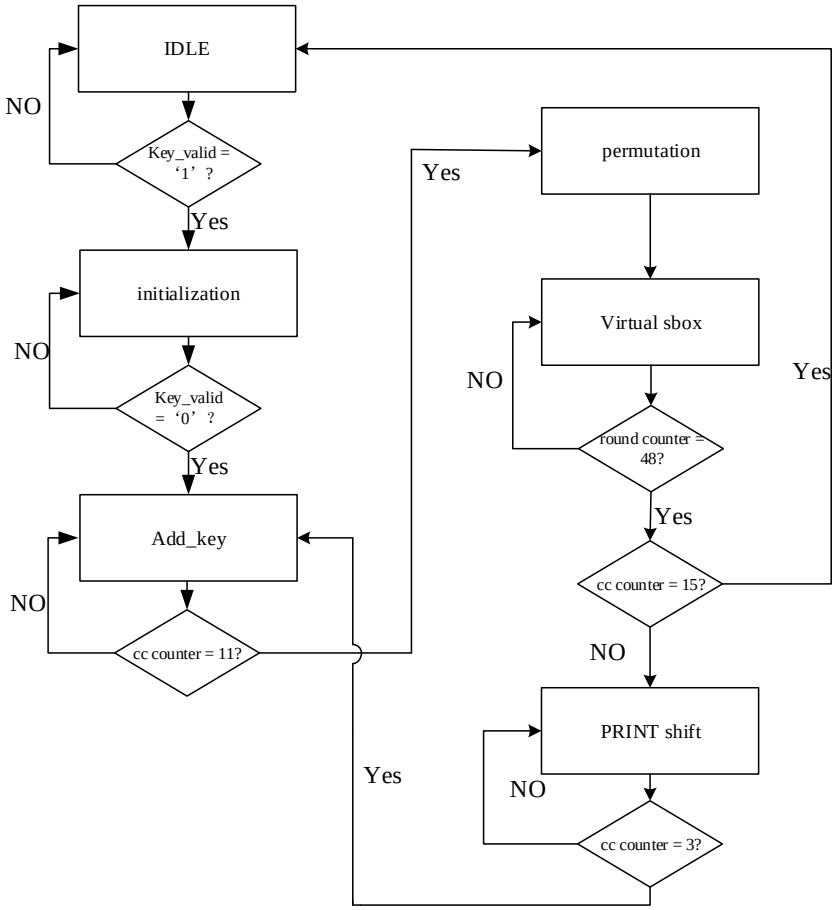


Figure 4.10 State Transition Diagram of the Serialized Design of PRINTcipher

The details of this FSM is described below starting with the "Add_key" state:

1. In the "Add_key" state, the key register and the state register will both left shift 4 bits every clock cycle. The "cc counter" here will increment by 1 every clock cycle. After 12 clock cycles, when the "cc counter" reaches 11, then the "Add_key" state is over. The FSM will transfer to the "permutation" state.

2. In the "permutation" state, the key register will hold for 1 clock cycle while the state register will parallel load the 64 bit input. Actually, after the "Add_key" state, the valid bits inside the state register are the 48 least-significant bits. The "permutation" component executes the permutation layer according to Figure 4.11 and after applying

the "permutation" layer, the 48 most significant bits will be the valid bits in the state register. Then the state register can be ready for the next “Virtual sbox” state. The “permutation” state will only last for one clock cycle and do an unconditional transfer to the “Virtual sbox” state.

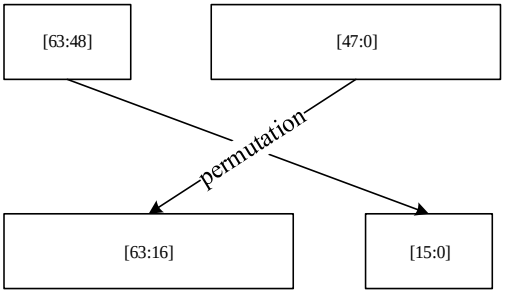


Figure 4.11 Structure of the "permutation" Component

3. In the “Virtual sbox” state, the state register will left shift 3 bits out and the key register will shift 2 bits out every clock cycle to perform the “Virtual sbox” component. The FSM will first check if the round counter reaches 48. If it reaches 48 and the “cc counter” reaches 15, the FSM will transfer to “IDLE” state. If not, it will step into the “PRINT shift” state when the “cc counter” reaches 15.

4. In the “PRINT shift” state, the state register will circular left shift 4 bits every clock cycle while the key register will hold in this state. The purpose for this state is to shift the 48 valid least-significant bits to the 48 most-significant bits and prepare for the next “Add_key” state.

Figure 4.12 provides the simulation result of our design. Notice that the output ciphertext is a 4 bit signal which is only valid for the last 3 bits. Converting the output ciphertext 7264537127476467 which is in octal to hexadecimal gives a result of

EB4AF95E7D37, which is exactly the same with the test vectors provided in [10]. We can conclude that our design is successful.

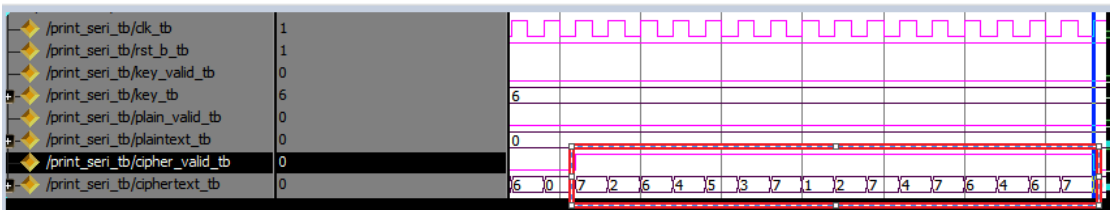


Figure 4.12 Simulation Result of the Serialized Design of PRINTcipher

Table 4.5 Summary of the Resource Consumption of the Serialized Design of PRINTcipher

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
PRINTcipher	225	161	100%
FSM	43	7	19.1%
key register and state register	147	144	65.3%

Table 4.5 provides the synthesis results of the serialized design of PRINTcipher. Compared to the iterative design, a lot of combination functions are saved resulting in a much smaller datapath. The total number of combinational functions consumed by the serialized design is only 225 while the iterative design consumes 516 combinational functions. However, there is a slight increase for the dedicated logic registers since we

add a 4 bit “cc counter” and we need more states in the FSM. The key register and state register consumes 65 percent combinational functions of the whole design due to the fact that we use a 4 bit datapath for the “Add_roundkey” layer and a 3 bit datapath for the “Virtual sbox” layer, which results in the combinational datapath only consuming a small number of combinational functions. Moreover, the number of combinational functions consumed by the FSM can be viewed as a metric for the complexity of the algorithm for a specific cipher. This design only consumes 43 combinational functions for the FSM, which is only half of the number for the FSM in the serialized design of the Piccolo cipher.

4.1.5 Serialized Design of LED Cipher

Unlike the previous three ciphers, we have two different designs for the serialized design of the LED cipher. The first one is a design without serializing the “Mixcolumns” component. The other one has serialized the “Mixcolumns” component.

4.1.5.1 Design Without Serializing the “Mixcolumns” Component

Figure 4.13 provides the structure of our design. Multiplexers “MUX1” here is a MUX(2,4) which is used to load the correct output of the initial key or the 4 bit shift output of the key register. Since in the LED cipher, the sbox is not connected directly with the “Add_roundkey”, two different datapaths are required. We use a “MUX2” which is a MUX(3,4) here to choose different inputs from the initial plaintext and these two different datapaths. More details of the operation in this structure will be illustrated

in the state transition diagram.

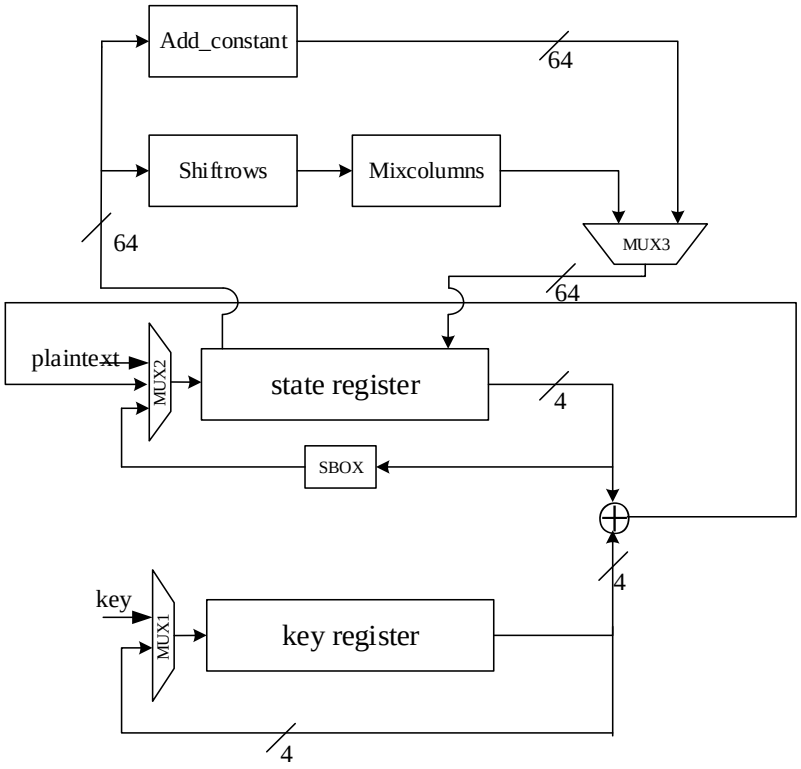


Figure 4.13 Hardware Structure of the Serialized Design of LED Cipher Without Serializing the "Mixcolumns" Component

The state transition diagram of the design is given in Figure 4.14. As the “initialization” state is same as the previous designs, we introduce the state transition diagram from the “Add_key” state.

1. In the “Add_key” state, the state register and the key register will both left shift 4 bits out every clock cycle. A total of16 clock cycles are required to finish this state. We use a 4 bit “cc counter” “cnt_4” to count the number of clock cycles in this state. When it reaches 15 and the round counter, “cnt_6”, does not reach 48 which means the encryption process is not over, the FSM will transfer to the “Add_constant” state. If the round counter reaches 48, the FSM will transfer back to the “IDLE” state.

2. In the “Add_constant” state, the key register will hold while the state register will parallel load the 64 bit input from the output of “MUX3” which should select the input from the “Add_constant” component. The FSM will do an unconditional transfer to “sbox” state.

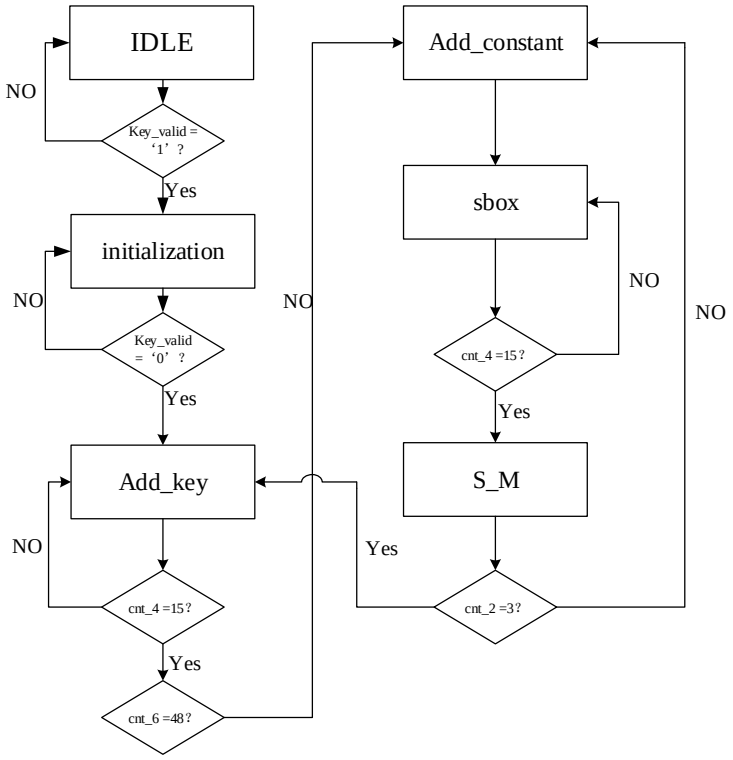


Figure 4.14 State Transition Diagram of the Serialized Design of LED Cipher Without Serializing the "Mixcolumns" Component

3. In the “sbox” state, the key register will also hold in this state and the state register will left shift 4 bits every clock cycle. Multiplexer “MUX2” will choose the output of the “sbox” component in this state and after 16 clock cycles when the “cc counter” reaches 15, the FSM will step into “S_M” state which refers to “Shiftrows” and “Mixcolumns”.

4. In the “S_M” state, the key register will still hold and the state register will parallel

load the 64 bit input from “MUX3” which will choose the correct output from the “Mixcolumns” component. Besides, another counter “cnt_2” will increment by one in this state. The FSM will also check if “cnt_2” reaches 3. If it reaches 3, then one step has been finished and the FSM will transfer to “Add_key” state. If not, the FSM will transfer to “Add_constant” state to do the next round operations.

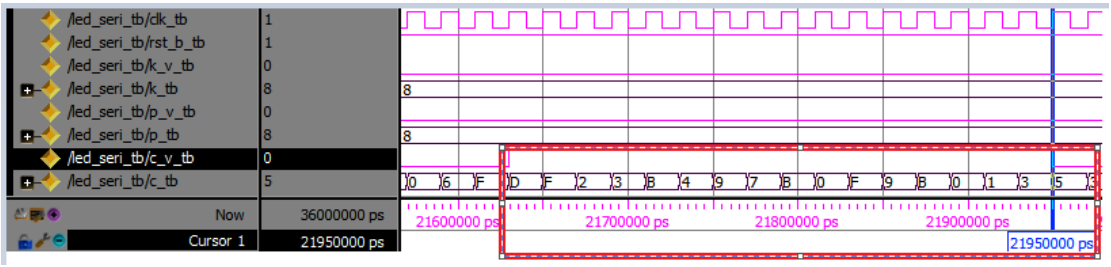


Figure 4.15 Simulation Result of Serialized Design of LED Cipher Without Serializing the "Mixcolumns" Component

Figure 4.15 shows the simulation result of this design. Comparing the output ciphertext with the test vectors provided in [25] proves that the design is successful. Notice that the time for the cursor inserted in the Modelsim [23] is 21950 ns. This is the time when all the bits of ciphertext have all been generated. Since another figure which is almost exactly the same as this one will be provided in the next section to show the simulation result of the serialized LED with the serialized “Mixcolumns” component, we point out this time to show the difference in the next section.

Table 4.6 provides the resource consumption result of our design. Compared to the iterative design, the serialized design saves a lot of combinational functions. The total combinational functions consumed by this design is only 293. Since we do not use a serialized “Mixcolumns” component in this design, it consumed 33.4 percent of the

total combinational functions. Besides, the combinational functions consumed by the state register and key register are not as high as the previous three ciphers because we only need 2 modes - 4 bit left shift and hold - for the key register and three modes for the state register - hold, 4 bit left shift and parallel load. In the Quartus II [20] synthesis results, “MUX3” is integrated into the state register, which increases the number of combinational functions consumed by the state register.

Table 4.6 Summary of the Resource Consumption of the Serialized Design of LED Cipher Without Serializing the "Mixcolumns" Component

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
LED	293	160	100%
FSM	51	6	19.1%
Mixcolumns	98	0	33.4%
key register and state register	91	144	31.1%

4.1.5.2 Design With Serialized “Mixcolumns” Component.

Recall the matrix used in the “Mixcloumns” in LED cipher. The matrix *M* can be viewed as the fourth power of matrix *A*.

$$A^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}^4 = \begin{pmatrix} 4 & 2 & 1 & 1 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix} = M$$

Matrix *A* is more hardware-friendly than *M*. Actually, in the LED cipher, we usually consider the 64 bit block as another matrix *C* with sixteen 4 bit nibbles:

$$C = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 & c_7 \\ c_8 & c_9 & c_{10} & c_{11} \\ c_6 & c_{13} & c_{14} & c_{15} \end{pmatrix} = (C_0 \quad C_1 \quad C_2 \quad C_3)$$

We use the first column vector C_0 as an example for the calculation. We can get the following equations

$$A * C_0 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix} * \begin{pmatrix} c_0 \\ c_4 \\ c_8 \\ c_6 \end{pmatrix} = \begin{pmatrix} c_4 \\ c_8 \\ c_{12} \\ 4c_0 + c_4 + 2c_8 + 2c_{12} \end{pmatrix}$$

According to this equation, we can use a less complicated “Mixcolumn” component which only deals with the calculation of $4c_0 + c_4 + 2c_8 + 2c_{12}$ and deals with only one column each clock cycle. In this way, 4 clock cycles are needed to finish one complete matrix. Since matrix *M* is matrix *A* to the power of four, we need to apply *A* four times. In total, 16 clock cycles are required to finish the “Mixcolumn”. The VHDL code can be found in Appendix A.3.

Figure 4.16 provides the structure of the serialized design of LED cipher with the serialized "Mixcolumns" component. Compared to Figure 4.13, the only difference here is that we change “MUX3” from MUX(2,64) to MUX(3,64) and the contents inside the “Mixcolumns” component is different. Thus, we only discuss how the “Mixcolumns”

component works in the following steps in relation to Figure 4.17:

1. The 64 bit block data will be treated as a matrix C inside the "Mixcolumns" component with the most-significant bits located in c_0 and least-significant bits in c_{15}.

2. The "Mixcolumns" component will only deal with the first column of the matrix C. Then it will shift the result of the first column to the last column and output the results to the register.

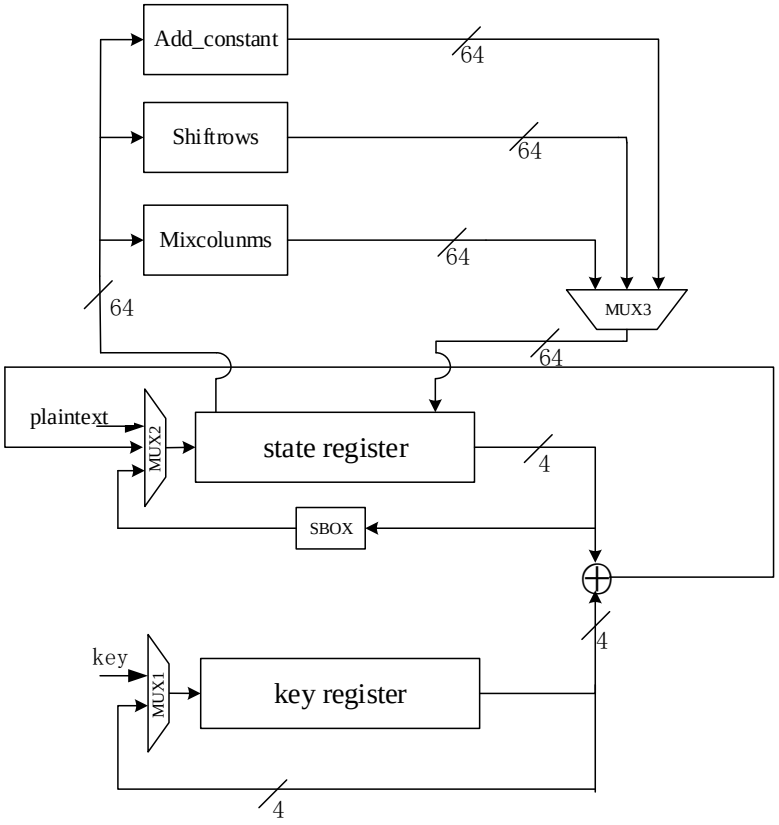


Figure 4.16 Hardware Structure of the Serialized Design of LED Cipher With the Serialized "Mixcolumns" Component

The state transition diagram for this design is almost identical to Figure 4.14 with only two differences. The first one is that we need to divide the "S_M" state into two different states, one is "Shiftrows" and the other one is "Mixcolumns". The second

difference is that the “Mixcolumns” and “Add_key” states both need the “cnt_4” to count from 0 to 15. Hence, we add another “remain” state between these two states to clear the counter. Thus, this design needs 780 more clock cycles to finish one entire encryption process than the design without the serialized “Mixcolumns”. The “Shiftrows” state will only last for one clock cycle while the “Mixcolumns” will last for 16 clock cycles.

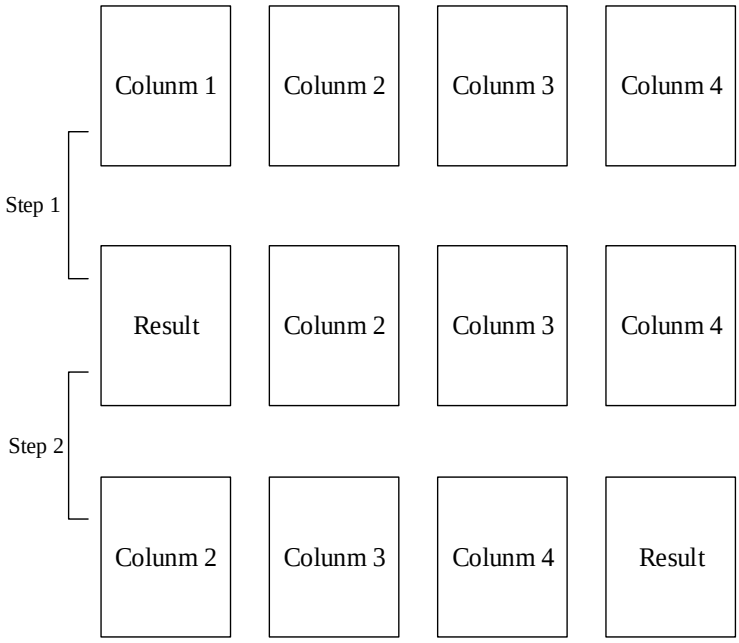


Figure 4.17 Steps of the the "Mixcolumn" component

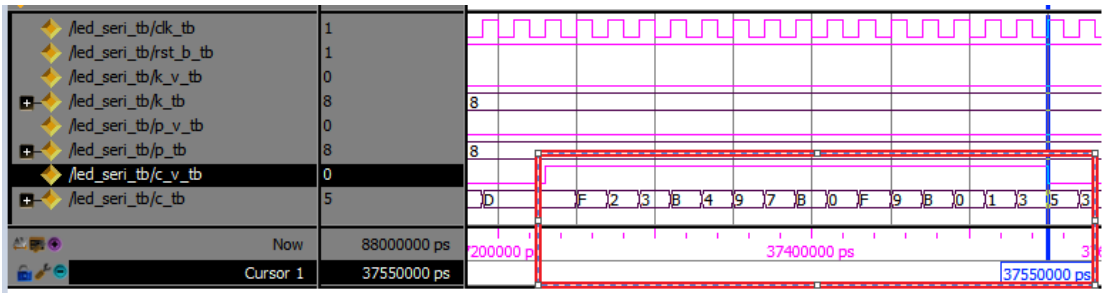


Figure 4.18 Simulation Result of Serialized Design of LED Cipher With the Serialized the "Mixcolumns" Component

Figure 4.18 provides the simulation result of our design. Since the “Mixcolumns”

needs 16 clock cycles each round, the time at which the last 4 bits of the ciphertext is generated is 37550 ns. Since the period for the clock cycle used in the simulation is 20 ns, this is consistent with the need for 780 more clock cycles than the previous design where the ciphertext is complete at 21950 ns. The simulation results prove that the design is successful.

Table 4.7 provides the synthesis result of the design with the serialized "Mixcolumns" component. Compared to Table 4.6, we find that the total number of combinational functions is slightly increased. Actually, to use the serialized "Mixcolumns" component, we need to add one more input to a 64 bit multiplexer. Although we do save 93 combinational functions for the "Mixcolumns" component, the one more input for the multiplexer consumes more resources than we save. Besides, we also find that the number of combinational functions consumed by the state register and key register is reduced. However, that is because the synthesis tool does not integrate "MUX3" into the state register. The actual number of combinational functions consumed by these registers should be 69+169= 238.

As the design with serialized "Mixcolumns" consumes more resources than the design without serialized "Mixcolumns" and has a lower throughput, we use our first design of LED when we integrate all these four ciphers into the final platform.

Table 4.7 Summary of the Resource Consumption of Serialized Design of LED Cipher with the Serialized the "Mixcolumns" Component

Entity	**Combinational functions**	**Dedicated logic registers**	**Percentage of the combinational functions**
LED	342	162	100%
Mixcolumns	5	0	1.5%
MUX3	169	0	49.4%
key register and state register	69	144	20.1%

4.2 Serialized Design of the Multi-cipher Platform

Figure 4.19 provides a block diagram of the serialized design of the platform. Similar to the iterative design of the platform, we also need another 3 bit input signal “cipher_mode” to indicate which cipher should be applied. The coding of the “cipher_mode” signal is the same as the previous iterative design and shown in Table 3.5. The input key and plaintext is 4 bits and the output ciphertext is also 4 bits. Unlike the iterative design of the platform, only one “Cipher_valid” and one ciphertext output are used to help reduce the number of I/O pins required.

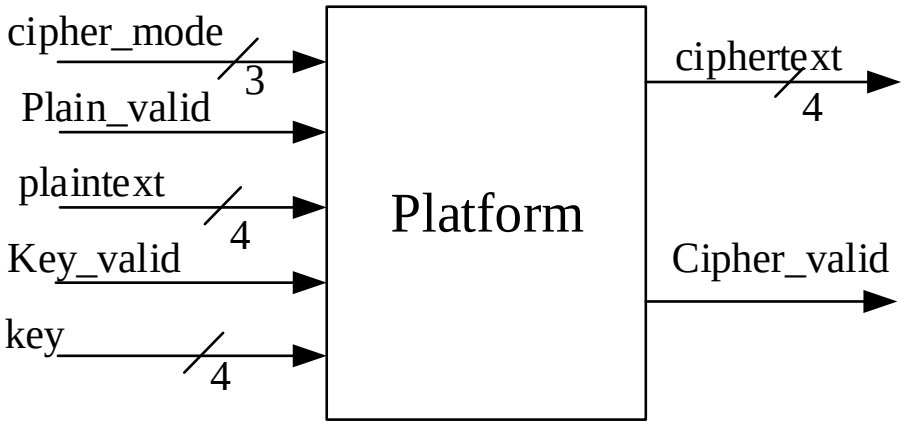


Figure 4.19 Block Diagram of the Serialized Design of the Platform

Figure 4.20 provides the structure of the serialized design of the platform. As this platform is compatible for all four ciphers, the register structure includes all the modes of operations required by all these four ciphers. The modes of operations for the state register are presented in Table 4.8 and illustrated by VHDL code in Appendix B.

We use a total number of eleven multiplexers inside the platform. Multiplexers “MUX1”, “MUX2”, “MUX3”, “MUX4”, “MUX9” and “MUX10” are exactly the same as the multiplexers used in the serialized design of the Piccolo cipher in Figure *4.6*. Also, “MUX11” is the same as the “MUX3” in the serialized design of the PRESENT cipher in Figure *4.2*. Multiplexers “MUX7” is a MUX(2,4) which is used to load the correct input from the output of the “PRESENT sbox” or “4 bit XOR A” component since the “4 bit XOR A” component is used to execute the “Add_roundkey” layer for PRESENT, LED and PRINTcipher. Multiplexers “MUX4” and “MUX8” are used to load the correct 4 bit left shift input to the state register and key register. Multiplexer “MUX5” is an extra multiplexer added in the platform which is used to load different parallel input to the state register.

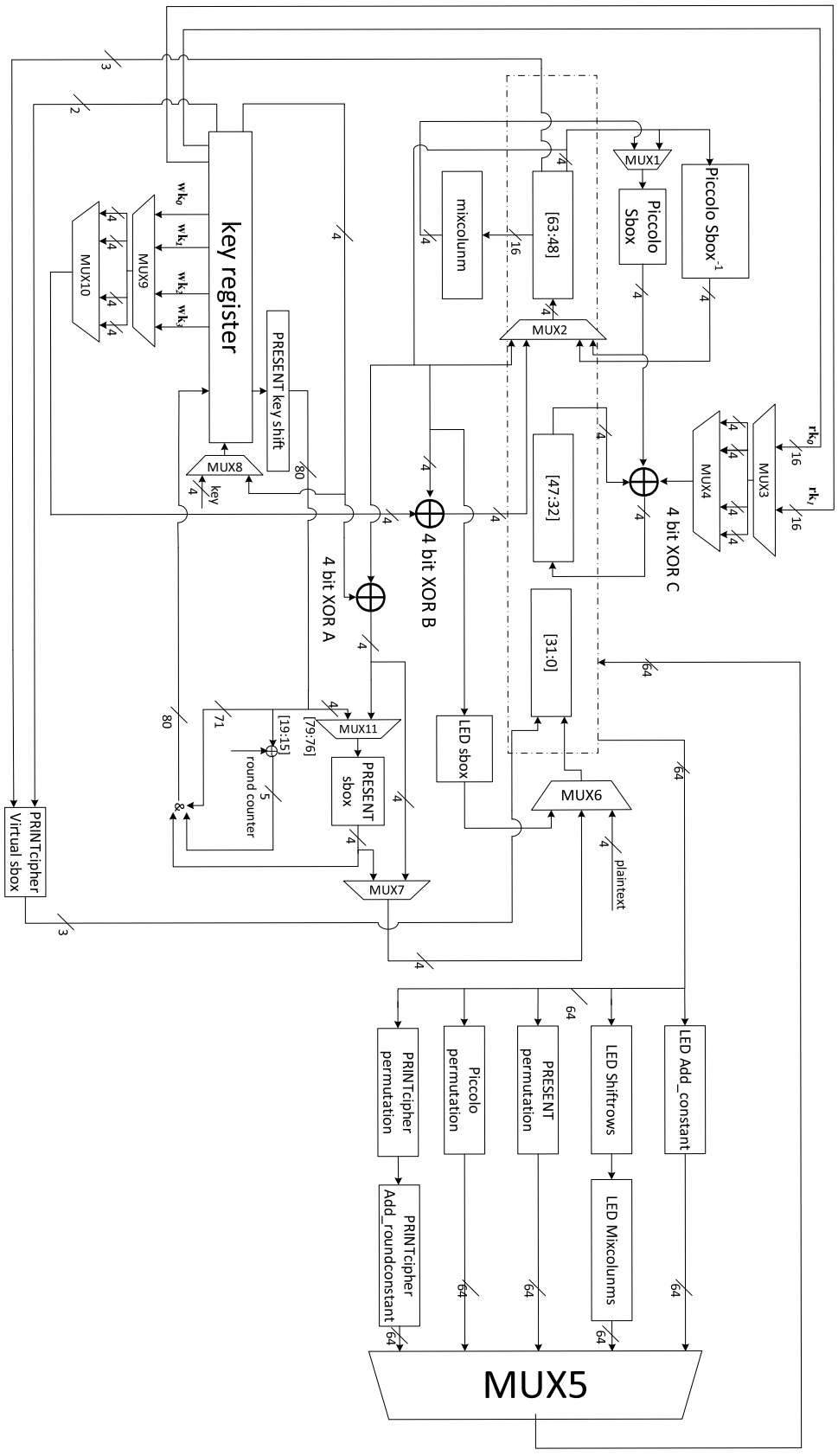


Figure 4.20 Hardware Structure of the Serialized Design of the Platform

The major factor we considered for this design is to reduce the total resources consumed by the platform after integrating PRESENT, Piccolo, PRINTcipher and LED into it. To achieve this objective, we share the key register and the state register for all four ciphers since they share several of the same modes of operations for the registers.

Table 4.8 provides the nine different modes of operations for the state register and key register in this platform. In the table, the ciphers that use these different operations are listed. Except that Piccolo and PRINTcipher have 5 different operations which are not used by PRESENT and LED, the remaining four operations are shared by all four ciphers. Recall as we discussed in relation to Figure 4.5, one operation is one more input to a 64 bit multiplexer and 80 bit multiplexer for the state register and key register, respectively. If we could share the same operations for different ciphers, since at one certain time only one cipher will be running in the platform, a lot of resources could be saved.

Table 4.8 Modes of Operations Shared by Different Cipher Registers

Modes of operations	**State register**	**Key register**	**Ciphers that needs this operation**
Hold.	Yes	Yes	All four ciphers
4 bit left shift to all the 64/80 bits with the 4	Yes	Yes	All four ciphers

bits input shift to the 4 least-significant bits.			
4 bit left shift to the 16 most-significant bits with the 4 bits input shift to 51^{st} bit to 48^{th} bit, while the rest is held.	Yes	No	Piccolo
32 most-significant bits swapped with the 32 least-significant bits	Yes	No	Piccolo
4 bit left shift to the 16 most significant bits and 47^{th} bit to 32^{nd} bit with two different inputs shift to 51^{st} bit to 48^{th} bit and 35^{th} to 32^{nd} respectively, while the rest is held.	Yes	No	Piccolo
3 bit left shift to all the 64 bits with the 3 bits input shift to the 3 least-significant bits.	Yes	No	PRINTcipher
2 bit left shift to all the 80 bits with the 2 bits input shift to the 3 least-significant bits.	No	Yes	PRINTcipher
Parallel load	Yes	Yes	All four ciphers
Synchronous clear	Yes	Yes	All four ciphers

For each different cipher, their combinational datapath is relatively independent by

using the multiplexers inside the design to choose the correct input or different input ports to the registers. For example, PRINTcipher needs a 3 bit left shift for the state register and a 2 bit left shift for the key register to execute the "Virtual sbox" component. For the other three ciphers, they need a 4 bit left shift for the state register to execute the sbox layer. Thus, PRINTcipher uses different input ports for the registers. By using this philosophy, different ciphers have comparatively separated combinational datapaths which contributes to the simplification of the simulation.

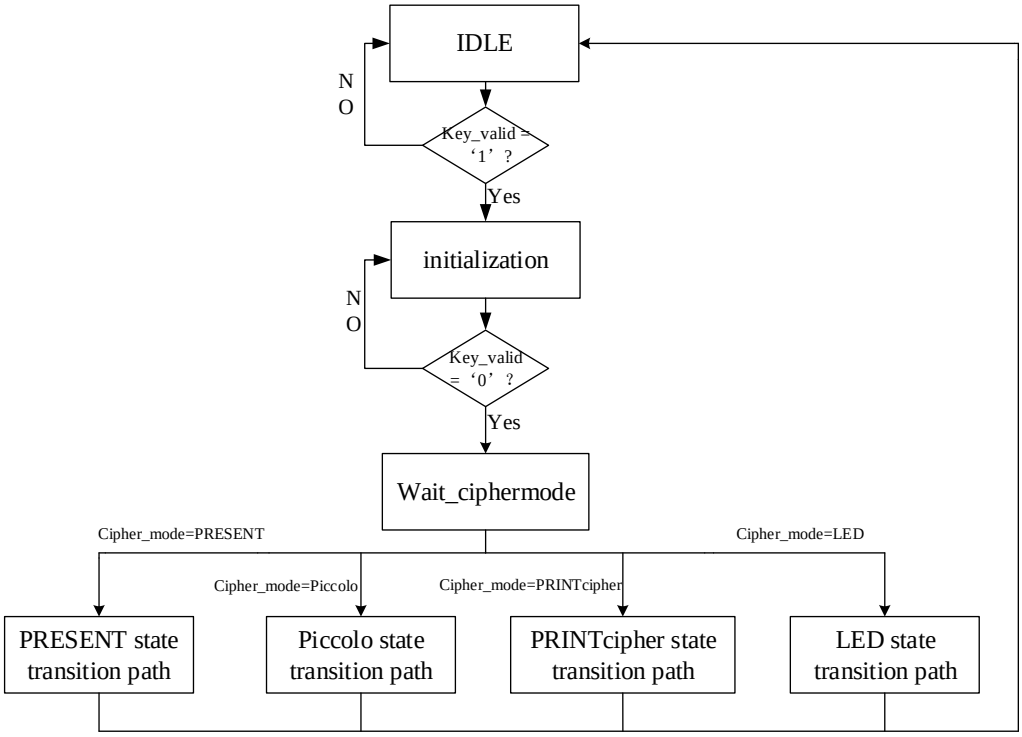


Figure 4.21 State Transition Diagram of the Serialized Design of the Platform

Figure 4.21 provides the state transition diagram of our design. The serialized design of each different cipher we discussed before and the serialized design of the platform, they all have the same "IDLE" state and "initialization" state. Following these two states, we define the rest of the state transition paths as "PRESENT state transition

path", "Piccolo state transition path", "PRINTcipher state transition path" and "LED state transition path". Since the "IDLE" state and "initialization" state are exactly the same as for the four individual ciphers, the details of this FSM is described below starting with the "Wait_ciphermode" state:

1. In "Wait_ciphermode" state, the FSM will wait for a valid value of the 3 bit "Cipher_mode" signal and step into one of the four state transition paths when receiving a valid "Cipher_mode" signal. For example, if the "Cipher_mode" signal is "001", the FSM will step into the "PRESENT state transition path".

2. In one of the state transition path states, the FSM will work according to each individual state transition path and transfer back to "IDLE" state when the ciphertext has been generated.

Figure 4.22 shows the simulation result of our design. Notice that the "c_m_tb" signal is the "Cipher_mode" signal. The platform will choose PRESENT, Piccolo, PRINTcipher or LED when the "Cipher_mode" signal is "001", "100", "011" or "010", respectively. Comparing to the waveforms for the previous four individual ciphers, we can draw the conclusion that our design is successful.

Figure 4.22 Simulation Results of the Serialized Design of the Platform

From Table 4.9, we find that we save a lot of dedicated logic registers since all the registers are shared by the four ciphers in the platform. Indeed, the number of dedicated logic registers are reduced to about 25% of the sum of the individually synthesized ciphers. Also, there is a slight decrease for the combinational functions. The total number of combinational functions consumed by the four individual ciphers is 1145, while the platform consumes only 1095 combinational functions. Compared to the iterative design of the platform, we have reduced the synthesized design by 769

combinational functions. We can conclude that the serialized design of the platform saves a large amount of hardware resources.

Table 4.9 Resource Usage Summary for Different Ciphers

Cipher	Combinational functions	Dedicated logic registers
PRESENT	206	158
Piccolo	372	164
PRINTcipher	225	161
LED	342	162
Total	**1145**	**645**
Platform	**1095**	**177**

However, another interesting result could be found from Table 4.9. Compared to the iterative design of the platform which saves 895 combinational functions over the individual cipher design, only 50 combinational functions are saved by the serialized design of the platform over the sum of the individual design resources. Actually, in the serialized design of the platform, the combinational functions consumed by the combinational datapath for each individual cipher is quite small, which results in the fact that “MUX5” in the serialized design of the platform is a much more significant

factor for the combinational functions compared to the “MUX2” in Figure 3.13. Notably, “MUX5” has one more 64 bit input than “MUX2” in Figure 3.13.

From Figure 4.19 we find that LED cipher uses two out of the five inputs of “MUX5”. If we remove the LED cipher from the platform, we can reduce the number of inputs of “MUX5” from 5 to 3 and it is reasonable to expect we can have a significantly better result for the combinational functions in the platform. Using the synthesis tool, we investigated that possibility and found that the new serialized platform consumes 852 combinational functions, while the remaining three individual ciphers consume a total number of 803 combinational functions. Hence, removing LED only reduced the number of combinational functions by 22.2% from 1095 to 852 while we expect the reduction to be perhaps more than 25%. In fact, the number of combinational functions for the platform with the remaining 3 ciphers is larger than the sum of the 3 individual ciphers. Hence, the result shows that the optimization by removing LED cipher does not improve the consumption of combinational functions.

Table *4.10* illustrates the reason why we can not reduce more combinational functions even though we remove the LED cipher from the platform. The key register and the state register consumed 37.5 percent of the total combinational functions. Even though the LED cipher is removed, the modes of operations for the key register and state register are still required by other ciphers. For this reason, we can not reduce more

than 25% of the combinational functions by removing the LED cipher.

Table 4.10 Number of Combinational functions of Key Register and State Register

Cipher	**Combinational functions of key register and state register**	**Percentage of Combinational functions**
PRESENT	155	75.2%
Piccolo	197	53.0%
PRINTcipher	147	67.3%
LED	238	69.6%
Total	**737**	**64.4%**
Platform	**411**	**37.5%**

Table 4.11 shows the performance of the serialized implementation. The throughtput is based on the maximum frequency of the platform being 183 MHz as determined by the critical path in the design. Different ciphers need different numbers of clock cycles to finish the encryption process. From this table, it is obvious that the throughput is significantly low. However, in most lightweight applications, such as RFID tags, the requirement for the throughput is not the major factor that will be taken

into consideration. Since the serialized platform saves a lot of hardware resources, we can conclude that our design is suitable for the lightweight applications or embedded systems where resource consumption is often critical. Moreover, we only need 20 I/O pins for the platform.

Table 4.11 Performance of the Serialized Implementation

Cipher	**Number of clock cycles**	**Throughput**
PRESENT	544	21.5 Mbps
Piccolo	711	16.8 Mbps
PRINTcipher	1581	5.6 Mbps
LED	1073	10.9 Mbps

4.3 Summary

In this chapter, we present the serialized design of the four individual ciphers and the multi-cipher platform. In addition, we provide two different designs of the serialized LED cipher. Comparison of these different implementations are also provided and we carefully investigated the differences of the resource consumption result.

In the next chapter, we will make a summary and give a conclusion for all the designs in this thesis. Several future work approaches are also provided.

Chapter 5

Conclusions and Future Work

In this thesis, we provide two different designs of four individual lightweight block ciphers and the multi-cipher platform. The iterative design of the four individual ciphers and the platform is comparatively straightforward. By sharing the similar components of these four ciphers, we significantly reduced both the combinational functions and dedicated logic registers in comparison to the total resources required to implement the ciphers individually. Hence, the number of LEs is significantly reduced. However, by using the serialized design, the number of combinational functions is reduced even further even though we have a slight increase in the dedicated logic registers over the iterative platform due to the complexity in the FSM. The whole design has been simulated by ModelSim [23] and test vectors are used to verify the correct operation of the systems designed. Our work has built a flexible platform which provides the possibility to add more lightweight block ciphers on the basis of our design.

5.1 Conclusions

The research objective of this thesis is the design of a platform which has small hardware resource consumption and high flexibility. First we investigated several papers which introduce several lightweight block ciphers and then we selected four

lightweight block ciphers: PRESENT, Piccolo, PRINTcipher and LED to be integrated into our platform. Two different multi-cipher platform designs have been proposed and simulated.

The first design is the iterative design which is also called the round-based design. We carefully designed the functional component of each individual cipher to save as many resources as possible. In the design of each individual cipher, we also considered the common features so that the integration of these four ciphers is not a difficult task. After the integration of these four ciphers into the iterative multi-cipher platform, we also considered the flexibility of the platform and included fault-tolerant design aspects to ensure that our platform could work smoothly. The simulation results have been provided for both the four individual ciphers and the platform, which proves that the design is successful. Synthesis results show the hardware resource savings resulting from sharing the similar components. A total of 32.4% combinational functions and 71.8% dedicated logic registers are saved for an iterative design. By simply changing the "Cipher_mode" signal, different ciphers could be chosen, which ensures the flexibility of the platform. However, since we realized that serializing the combinational datapath, more combinational functions could be saved, we were motivated to design a more compact platform.

The second design is the serialized design. Compared to the iterative design, it is much more complex. We have re-investigated these four ciphers and found methods to deal with the difficulties of the serialization of the datapath. One challenge was to

serialize the “Mixcolumns” both in LED cipher and Piccolo cipher and this was solved by using a specially designed shift register. Another problem was to deal with the F function inside the Piccolo cipher and this was solved by using an inverse sbox. After successfully finishing the serialized design of each individual cipher, the combinational datapath for each cipher was seriously investigated to check if more components can be shared other than the registers. Finally, we found that only the “4 bit XOR A” component could be shared by PRESENT, PRINTcipher and LED cipher to finish the “Add_roundkey” layer. Hence, we chose, when we integrated these four ciphers into the platform, to keep their combinational datapath comparatively separated. The simulation results and synthesis results are provided. The results prove that the design is successful. Compared to the iterative design, we save 41.3% more combinational functions. The penalty is that the throughput is really low. However, there is always a trade-off between hardware resource consumption and the performance. Since the resource consumption is the biggest factor we need to take into consideration for lightweight applications and embedded systems, we conclude that our design is worthwhile and successful.

5.2 Future Work

In this section, we conclude the thesis by identifying some directions for future work.

5.2.1 Transferring the Platform from FPGA to ASIC

All designs in this thesis are synthesized based on Altera Cyclone IV FPGA [24].

However in most lightweight applications, an FPGA is not a good option since usually it costs more than ASIC implementation. A lot of extra resources are provided in an FPGA chip that may not be used in lightweight applications. For example, in the synthesis results given by Quartus II [20], all these designs in this thesis use less than 1% of the total combinational functions and dedicated logic registers provided by the device. In the future, a synthesis based on CMOS technology should be applied. For example, we can consider transferring the platform to the Taiwan Semiconductor Manufacturing Company (TSMC) 90 nm CMOS technology.

5.2.2 Verification in Real Test Environment

Further verification should be applied for the platform on a real test board. For example, since we use Altera Cyclone IV FPGA in our design, we could download the project to the DE-II board [27]and run more functional tests in the real test environment. This verification could examine any possible issues inside the dataflow of our platform.

5.2.3 Adding More Lightweight Block Ciphers on the Platform

Since our multi-cipher platform is designed with the consideration of flexibility, more lightweight block ciphers could be added to our platform easily. However, the potential ciphers need to be carefully investigated so that the selected cipher could also share the similarities with the integrated four ciphers. If the selected cipher has a big difference with the four integrated ciphers, it may increase the number of multiplexers

needed in the implementation.

Even though throughput is not the major factor we consider for the multi-cipher platform, adding more ciphers into the platform may result in a larger delay in the critical datapath, which could lead to a decrease of the throughput. As a result, the trade-off between the resource consumption and throughput needs to be seriously investigated.

References

[1] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, Handbook of Applied Cryptography, New York: CRC Press, 2001.

[2] W. Stallings, Cryptography and Network Security Principles and Practices., New Jersey: Pearson Prentice Hall, 2006.

[3] "DATA ENCRYPTION STANDARD (DES)," National Institute of Standards and Technology (NIST), 25 Oct 1991. [Online]. Available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

[4] "Advanced Encryption Standard (AES) Development Effort," National Institute of Standards and Technology, [Online]. Available: http://csrc.nist.gov/archive/aes/index2.html.

[5] G. Leander, "Lightweight Block Cipher Design," 2014. [Online]. Available: http://summerschool-croatia14.cs.ru.nl/slides/Lightweight%20Block%20Cipher%20Design.pdf.

[6] "Lightweight block ciphers website," 2015 May 15. [Online]. Available: www.cryptolux.org/index.php/Lightweight_Block_Ciphers..

[7] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," vol. 4727, pp. 450-466, Lecture Notes in Computer Science, in Cryptographic Hardware and Embedded Systems, 2007.

[8] G. Leander and A. Poschmann, "On the Classfication of 4 Bit S-boxes," vol. 4547, pp. 159-176, Lecture Notes in Computer Science, in Proceedings of Arithmetic of Finite Fields, Heidelberg, 2007.

[9] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita and T. Shirai, "Piccolo: An Ultra-Lightweight Blockcipher," vol. 6917, pp. 342-357, Lecture Notes in Computer Science , in Cryptographic Hardware and Embedded Systems, 2011.

[10] L. Knudsen, G. Leander, A. Poschmann and M. J. Robshaw, "PRINTcipher: A Block Cipher for IC-Printing," vol. 6225, pp. 16-32, Lecture Notes in Computer Science, in Cryptographic Hardware and Embedded Systems, 2010.

[11] J. Guo, T. Peyrin, A. Poschmann and M. Robshaw, "The LED Block Cipher," vol. 6917, pp. 326-341, Lecture Notes in Computer Science, in Cryptographic Hardware and Embedded Systems, 2011.

[12] C. D. Cannière, O. Dunkelman and M. Knežević, "KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers," vol. 5747, pp. 272-288, Lecture Notes in Computer Science, in Cryptographic Hardware and Embedded Systems, 2009.

[13] Z. Gong, S. Nikova and Y. W. Law, "KLEIN: A New Family of Lightweight Block Ciphers," vol. 7055, pp. 1-18, Lecture Notes in Computer Science in RFID. Security and Privacy, 2012.

[14] M. Izadi, B. Sadeghiyan, S. S. Sadeghian and H. A. Khanooki, "MIBS: A New Lightweight Block Cipher," vol. 5888, pp. 334-348, Lecture Notes in Computer Science, in Cryptology and Network Security, 2009.

[15] T. Suzaki, K. Minematsu, S. Morioka and E. Kobayashi, "TWINE: A Lightweight Block Cipher," vol. 7707, pp. 339-354, Lecture Notes in Computer Science, in Selected Areas in Cryptography, 2013.

[16] S. Kerckhof, F. Durvaux, C. Hocquet, D. Bol and . F.-X. Standaert, "Towards Green Cryptography: A Comparison of Lightweight Ciphers from the Energy Viewpoint," vol. 7428, pp. 390-407, Lecture Notes in Computer Science, in Cryptographic Hardware and Embedded Systems, 2012.

[17] T. Eisenbarth, C. Paar , A. Poschmann, S. Kumar and L. Uhsadel, "A Survey of LightweightCryptography," IEEE Design & Test of Computers, no. Special Issue on Secure ICs for Secure Embedded Computing, pp. 522-533, 2007.

[18] Z. Navabi, VHDL: Analysis and Modeling of Digital Systems, New York: McGraw-Hill, 1997.

[19] "Logic Elements and Logic Array Blocks in Cyclone IV Deivice," [Online]. Available: https://www.altera.com/en_US/pdfs/literature/hb/cyclone-iv/cyiv-

51002.pdf.

[20] "Quartus II Handbook," Altera, [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/quartusii_handbook.pdf.

[21] "My First FPGA Design Tutorial," [Online]. Available: https://www.altera.com/en_US/pdfs/literature/tt/tt_my_first_fpga.pdf.

[22] H. Liao and H. M. Heys, "An Integrated Hardware Platform For Four Lightweight Block Ciphers," in CCECE, Halifax, 2015.

[23] "ModelSim® User's Manual," [Online]. Available: http://www.microsemi.com/document-portal/doc_view/131619-modelsim-user..

[24] "Altera Cyclone IV Device Handbook," [Online]. Available: http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf..

[25] J. Guo, T. Peyrin, A. Poschmann and M. Robshaw, "The LED Block Cipher," [Online]. Available: https://sites.google.com/site/ledblockcipher/.

[26] C. Rolfes, A. Poschmann, G. Leander and C. Paar, "Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents," in Smart Card Research and Advanced Applications, 2008.

[27] "DE2 Development and Education Board User Manual," [Online]. Available: ftp://ftp.altera.com/up/pub/Webdocs/DE2_UserManual.pdf.

Appendix A

VHDL Code for “Mixcolumns” Component in Piccolo Cipher and LED Cipher

A.1 “Mixcolumns” in Piccolo Cipher

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.all;
entity piccolo_mix is
 port ( x : in std_logic_vector( 15 downto 0);
   output : out std_logic_vector( 15 downto 0));
end entity;

architecture rtl of piccolo_mix is
begin
   output(15) <=x(14) xor x(11) xor x(10) xor x(7) xor x(3);
   output(14) <=x(13) xor x(10) xor x(9) xor x(6) xor x(2);
   output(13) <=x(15) xor x(12) xor x(11) xor x(9) xor x(8) xor x(5) xor x(1);
   output(12) <=x(15) xor x(11) xor x(8) xor x(4) xor x(0);
   output(11) <=x(15) xor x(10) xor x(7) xor x(6) xor x(3);
   output(10) <= x(14) xor x(9) xor x(6) xor x(5) xor x(2);
   output(9)   <= x(13) xor x(11) xor x(8) xor x(7) xor x(5) xor x(4) xor x(1);
   output(8)   <= x(12) xor x(11) xor x(7) xor x(4) xor x(0);
   output(7)   <= x(15) xor x(11) xor x(6) xor x(3) xor x(2);
   output(6)   <= x(14) xor x(10) xor x(5) xor x(2) xor x(1);
```

```
   output(5)   <= x(13) xor x(9) xor x(7) xor x(4) xor x(3) xor x(1) xor x(0);
   output(4)   <= x(12) xor x(8) xor x(7) xor x(3) xor x(0) ;
   output(3)   <= x(15) xor x(14) xor x(11) xor x(7) xor x(2);
   output(2)   <= x(14) xor x(13) xor x(10) xor x(6) xor x(1);
   output(1)   <= x(15) xor x(13) xor x(12) xor x(9) xor x(5) xor x(3) xor x(0);
   output(0)   <= x(15) xor x(12) xor x(8) xor x(4) xor x(3);
end architecture;
```

A.2 “Mixcolumns” in LED Cipher Using Matrix *M*

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
entity LED_mixcolumn is
      port( a : in std_logic_vector(63 downto 0);
      o : out std_logic_vector(63 downto 0));
end entity;
architecture rtl of LED_mixcolumn is
begin
      label1: for i in 0 to 3 generate
      -- 63-60
      o(63-4*i)<= a(61-4*i) xor a(47-4*i) xor a(30-4*i) xor a(14-4*i);
      o(62-4*i)<= a(63-4*i) xor a(60-4*i) xor a(46-4*i) xor a(29-4*i) xor a(13-4*i);
      o(61-4*i)<= a(63-4*i) xor a(62-4*i) xor a(45-4*i) xor a(31-4*i) xor a(28-4*i) xor
a(15-4*i) xor a(12-4*i);
      o(60-4*i)<= a(62-4*i) xor a(44-4*i) xor a(31-4*i) xor a(15-4*i);
      --47-44
      o(47-4*i)<= a(63-4*i) xor a(60-4*i) xor a(46-4*i) xor a(45-4*i) xor a(31-4*i) xor
a(29-4*i) xor a(14-4*i) xor a(13-4*i);
```

```
o(46-4*i)<= a(63-4*i) xor a(62-4*i) xor a(47-4*i) xor a(45-4*i) xor a(44-4*i) xor a(31-4*i) xor a(30-4*i) xor a(28-4*i) xor a(15-4*i) xor a(13-4*i) xor a(12-4*i);

o(45-4*i)<= a(62-4*i) xor a(61-4*i) xor a(46-4*i) xor a(44-4*i) xor a(31-4*i) xor a(30-4*i) xor a(29-4*i) xor a(14-4*i) xor a(12-4*i);

o(44-4*i)<= a(61-4*i) xor a(47-4*i) xor a(46-4*i) xor a(30-4*i) xor a(28-4*i) xor a(15-4*i) xor a(14-4*i);

--31-28

o(31-4*i)<= a(62-4*i) xor a(60-4*i) xor a(47-4*i) xor a(46-4*i) xor a(45-4*i) xor a(44-4*i) xor a(31-4*i) xor a(30-4*i) xor a(28-4*i) xor a(12-4*i);

o(30-4*i)<= a(63-4*i) xor a(61-4*i) xor a(46-4*i) xor a(45-4*i) xor a(44-4*i) xor a(31-4*i) xor a(30-4*i) xor a(29-4*i) xor a(15-4*i);

o(29-4*i)<= a(63-4*i) xor a(62-4*i) xor a(60-4*i) xor a(45-4*i) xor a(44-4*i) xor a(31-4*i) xor a(30-4*i) xor a(29-4*i) xor a(28-4*i) xor a(14-4*i);

o(28-4*i)<= a(63-4*i) xor a(61-4*i) xor a(60-4*i) xor a(47-4*i) xor a(46-4*i) xor a(45-4*i) xor a(31-4*i) xor a(29-4*i) xor a(13-4*i) xor a(12-4*i);

--15-12

o(15-4*i)<= a(62-4*i) xor a(46-4*i) xor a(30-4*i) xor a(29-4*i) xor a(28-4*i) xor a(14-4*i) xor a(12-4*i);

o(14-4*i)<= a(61-4*i) xor a(45-4*i) xor a(29-4*i) xor a(28-4*i) xor a(15-4*i) xor a(13-4*i);

o(13-4*i)<= a(63-4*i) xor a(60-4*i) xor a(47-4*i) xor a(44-4*i) xor a(28-4*i) xor a(15-4*i) xor a(14-4*i) xor a(12-4*i);

o(12-4*i)<= a(63-4*i) xor a(47-4*i) xor a(31-4*i) xor a(30-4*i) xor a(29-4*i) xor a(28-4*i) xor a(15-4*i) xor a(13-4*i) xor a(12-4*i);

end generate;

end architecture;
```

A.3 Serialized “Mixcolumns” in LED Cipher Using Matrix *A*

```
LIBRARY IEEE;

USE IEEE.std_logic_1164.ALL;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_arith.ALL;
```

```
entity LED_mixcolumn_seri is
    port(input3, input2, input1, input0: std_logic_vector(3 downto 0);
    output: out std_logic_vector(15 downto 0));
end entity;
architecture rtl of LED_mixcolumn_seri is
begin
output(15 downto 12) <= input2;
output(11 downto 8) <= input1;
output(7 downto 4) <= input0;
output(3)<= input3(1) xor input2(3) xor input1(2) xor input0(2);
output(2)<= input3(3) xor input3(0) xor input2(2) xor input1(1) xor input0(1);
output(1) <= input3(3) xor input3(2) xor input2(1) xor input1(3) xor input1(0) xor input0(3) xor input0(0);
output(0) <= input3(2) xor input2(0) xor input1(3) xor input0(3);
end rtl;
```

Appendix B

VHDL Code for the State Register Used in the Serialized Design of the Platform

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.all;
entity platform_reg is
   port(clk, rst_b: in std_logic;
      sel_reg: in std_logic_vector(2 downto 0);
      parallel_input: in std_logic_vector(63 downto 0);
   shift_input1 : in std_logic_vector(3 downto 0);
   shift_input2 : in std_logic_vector(3 downto 0);
   shift_input3 : in std_logic_vector(3 downto 0);
   shift_input_3bit : in std_logic_vector(2 downto 0);
   shift_out_1,shift_out_2: out std_logic_vector(3 downto 0);
   shift_out_3bit : out std_logic_vector(2 downto 0);
      parallel_out: out std_logic_vector(63 downto 0));
end entity;
architecture rtl of platform_reg is
   signal reg: std_logic_vector(63 downto 0);
begin
   p0:process(rst_b, clk)
   begin
      if (rst_b='0') then
```

```
        reg<=(others=>'0');
      elsif (clk='1' and clk'event) then
        case sel_reg is
        when "000"=> reg<=reg;
        when "001"=> reg<=reg(59 downto 0) & shift_input1;
        when "010"=> reg<=reg(59 downto 48) & shift_input2 &reg(47 downto 0);
        when "100"=> reg<=reg(31 downto 0) & reg(63 downto 32);
        when "011" => reg<= reg(59 downto 48) &shift_input2 &reg(43 downto 32) &
shift_input3 &reg(31 downto 0);
        when "101"=> reg<= reg(60 downto 0) & shift_input_3bit;
        when "110"=> reg<= parallel_input;
        when "111"=> reg<=   (others=> '0');
        when others=>null;
        end case;
      else
        null;
      end if;
   end process;
   parallel_out<=reg(63 downto 0);
   shift_out_1 <= reg(63 downto 60);
   shift_out_2 <= reg(47 downto 44);
   shift_out_3bit<= reg(63 downto 61);
end rtl;
```