RTL IMPLEMENTATION OF SECURE HASH ALGORITHM 3 (SHA-3) TOWARDS SMALLER AREA

LIM YEN RUEN

UNIVERSITI SAINS MALAYSIA

2017

RTL IMPLEMENTATION OF SECURE HASH ALGORITHM 3 (SHA-3) TOWARDS SMALLER AREA

by

LIM YEN RUEN

A Dissertation submitted for partial fulfilment of the requirement

for the degree of Master of Science

(Microelectronic Engineering)

August 2017

ACKNOWLEDGEMENT

This dissertation is dedicated to everyone in the field of microelectronic design who embarks the journey of expanding the collection of knowledge and transcendent passion for improving the hash system.

My uttermost gratitude goes to Assc. Prof. Dr. Bakhtiar Affendi Bin Rosdi, my thesis advisor and project supervisor, for his invaluable support and guidance that were crucial for the completion of this project. Dr. Bakhtiar has been supportive during the process of brainstorming. His generous opinion and practical experience has been very helpful in the progress of this project.

Extensive acknowledgement need to be paid to my friends and family, whom have truly support and make the project possible. Thanks for the unlimited love and motivation during my ups and downs throughout the attempts.

Last but not least, I wish to record my sincere appreciation and thanks to all my friends for their invaluable supports and encouragements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
DMUX	Demultiplexer
FIPS	Federal Information Processing Standard
FPGA	Field-Programmable Gate Array
HMAC	Hashed Message Authentication Code
IoT	Internet of Things
LUT	Look Up Table
MAC	Message Authentication Code
MD	Message Digest
MUX	Multiplexer
NIST	National Institute Standards and Technology
PISO	Parallel-In Serial-Out
ROM	Read-Only Memory
RTL	Register Transfer Language
SHA	Secure Hash Algorithm
VCS	Synopsys Verilog Compiler Simulator
XOF	Extendable-Output Function

PELAKSANAAN RTL UNTUK ALGORITMA CINCANGAN SELAMAT KE ARAH SAIZ YANG LEBIH KECIL

ABSTRAK

Pemindahan data dengan selamat merupakan tugas yang paling mencabar bagi Objek Rangkaian Internet (IoT). Keutuhan data perlu dipastikan sebelum dan selepas penghantaran data. Fungsi cincangan kriptografi secara amnya digunakan untuk pengesahan integriti data. Fungsi cincangan kriptografi adalah asas kepada rangkaian internet dan menjalankan proses seperti pengesahan identiti, pemeriksaan integriti fail, penghantaran kod, dan kawalan versi kod sumber. Antara semua cincangan kriptografi, Algoritma Cincangan Selamat (SHA-3) adalah cincangan kriptografi terbaru dan selamat untuk digunakan dalam bidang elektronik. Saiz Intel Microelectronic SHA-3 besar disebabkan banyak logika pertengahan algoritma. Objektif projek ini adalah untuk mereka bentuk SHA-3 dengan 256-bit keluaran dan 1600-bit matriks dengan saiz yang lebih kecil berbanding dengan seni bina Intel Microelectronic SHA-3. Penyelidikan ini melaksanakan SHA-3 dengan mengabungkan semua algoritma SHA-3 secara logik dan hanya menggunakan input lorong matriks bagi menghapuskan logika pertengahan untuk mengurangkan saiz reka bentuk. Kes ujian daripada Institut Piawaian dan Teknologi Kebangsaan (NIST) digunakan untuk pengesahan fungsi. Reka bentuk akhir SHA-3 dalam kajian ini boleh mencapai pengurangan saiz sebanyak 12.57%, pengurangan bilangan get sebanyak 24.35%, pengurangan kritikal panjang sebanyak 18.84%, dan pengurangan kitaran jam yang diperlukan untuk menjana keluaran cincangan sebanyak 75%. Kesimpulannya, SHA-3 yang lebih kecil dan prestasi yang lebih tinggi telah direkabentuk dan berkemungkinan boleh digunakan untuk memenuhi keperluan IoT.

RTL IMPLEMENTATION OF SECURE HASH ALGORITHM 3 (SHA-3) TOWARDS SMALLER AREA

ABSTRACT

Secure data transfer has been the most challenging task for Internet of Things (IoT) devices. Data integrity must be ensured before and after the data transmission. Cryptographic hash functions are generally the basis of a secure network and used for data integrity verification. Cryptographic hash functions carried out processes such as identity verification, file integrity checking, secure key passing, and source code version control. Among all of the cryptography measures, Secure Hash Algorithm 3 (SHA-3) is the newest and secure cryptographic hash algorithm in the current electronic industry. In the previous Intel Microelectronic SHA-3 design, the synthesized area of the design is large due to many intermediate states and logics of the step mapping functions. The objective of this project is to design a synthesizable SHA-3 with 256-bits hash output and 1600-bits state array with lower area compared to Intel Microelectronic SHA-3. This research implements the SHA-3 in ways such that all the step mapping algorithms are logically combined to only use the input lanes of the state array to eliminate the intermediate logics and reduces the area size. Functionality verification is done using the test case provided by National Institute Standards and Technology (NIST). Two squeezing phases are tested to ensure the functionality of design. Final design of SHA-3 in this research can achieve area reduction by 12.57%, the cell count reduction by 24.35%, the critical path length reduction by 18.84%, and reduction of the clock cycles needed to generate the hash output by 75%. In conclusion, the SHA-3 with smaller area and higher performance has been designed and is possible to cater the needs of IoT application.

CHAPTER 1

INTRODUCTION

1.1 Background

Secure data transfer has been the most challenging task for Internet of Things (IoT) devices. Important data must be encrypted before ready to transfer. Although data is encrypted for security, the encrypted data may still be altered on the network. Cryptographic hash functions are generally the basis of a secure network and mainly used for data integrity verification [1]. Cryptographic hash functions carried out processes such as identity verification, file integrity checking, secure key passing, and source code version control [2]. These cryptographic hash functions are widely used as sole cryptographic modules or incorporated in hash-based authentication mechanisms like the Hashed Message Authentication Code [3]. Additionally, applications that employs hash functions include the Secure Electronic Transaction [4], Internet Security Protocol [5] that is a mandatory feature of the Internet Protocol version 6 (IPv6) and the Public Key Infrastructure [6].

There are several Hash functions such as SHA-1, SHA-256, SHA-512, MD4 and MD5. The most-widely used hash function recently was the Secure Hash Algorithm 2 (SHA-2) and Message Digest 5 (MD5) [7, 8]. Recent advances in the cryptanalysis of these commonly used standard hash functions reported some collisions and serious vulnerabilities in these algorithms [9-11]. Although no assaults have yet been accounted for on the SHA-2 variations, but because of hardware likenesses to SHA-1 there are fears

that SHA-2 could likewise be broken sooner rather than later. SHA-3 is introduced as a more secure hash function in a competition organized by NIST [12]. Among all of the cryptography measures, Secure Hash Algorithm 3 (SHA-3) is the newest and secure cryptographic hash algorithm and had not been used in practice.

SHA-3 is an efficient scheme for both hardware and software implementation. Although SHA-3 can be implemented at software level, it is slower and less efficient compared to hardware implementation where parallel operation is possible [13]. For some Internet of Things (IoT) devices, cryptographic hash functions hardware is needed to provide data integrity verification in real time. Due to the demanded high security level, the need for high performance is a significant factor for the choice of security implementation. Thus, for securities issue, hardware implementation is far more suitable compared to the corresponding software implementations. Whereas the hardware implementations of cryptographic hash functions should be low in gate counts to cater the needs of smaller chips in IoT devices. The objective of this project is to design a hardware efficient SHA-3 with lower area compared to previous architecture. The final outcome of this project is a SHA-3 architecture design with small synthesized area.

1.2 Problem Statement

Implementation of cryptographic hash function in IoT devices has to be secured to give extraordinary properties, for example, collision and preimage resistance, which are critical for some applications in data security [14]. Although data is encrypted for security, the encrypted data may still be altered on the network. The modified data can be disastrous for the applications executed at the destination node and can prompt off

undesired responses. Thus, hash function in hardware devices is needed for a more secure of data transmission. However, the commonly used standard hash functions such as SHA-1 and MD5 had been reported for some collisions and serious vulnerabilities [9-11]. To prevent the possibility of SHA-2 to be cracked in the near future due to hardware similarity, SHA-3 is introduced to provide a more secure hash function capability. It was introduced in a competition organized by NIST [14]. Thus, it is importance to move the current security hash function from SHA-2 to SHA-3 in the industry application.

The implementation of hash function hardware has to be area efficient since majority of Internet of Things (IoT) devices are embedded system where power and area is limited. As an emerging technology, promising solutions to transform the operation and role of many existing industrial system can be offered by IoT devices [15]. Since cryptographic hash function is applicable to most of the device in communication and the developed design will be reused for different hardware implementation, the hash function hardware has to be as smallest as possible. This will result in cost saving as well as smaller IoT devices. However, from the previous architecture, the area of the hardware implementations are large due to the implementation of Secure Hash Algorithm 3 without utilizing the resource correctly [16, 17]. By using hardware efficiently, such as reusing resources, or by analysing the algorithm of cryptographic hash function and reducing the unnecessary logic in the function, less hardware will be used and results in smaller area.

1.3 Research Objectives

The objectives of this project are:

1. To reduce the area of Secure Hash Algorithm-3 (SHA-3) in the previous implemented hardware design by logically combined the step mapping algorithms.
2. To compare the performance of the proposed architecture and the previous implemented design in term of area, cell counts, critical path length, and speed.

1.4 Scope of the Research

The focus of this project is to determine the method to develop the Secure Hash Algorithm 3 (SHA-3) cryptographic hash function over the hardware approach by Register-Transfer Level (RTL) implementation. The design architecture is to feed the usage purpose of Intel Microelectronic Sdn. Bhd. Based on the specification, the previous SHA-3 hardware design of Intel Microelectronic Sdn. Bhd. is implemented and synthesized. System Verilog will be used to develop the cryptographic hash algorithm. The hardware section that contributes to the most percentage of total area is determined and analysed for suitable method to reduce the area. Then, the proposed design is developed to reduce the area of SHA-3 hardware implementation. Simulation will be performed on the test bench using Synopsys Verilog Compiler Simulator (VCS). After verified the functionality of the proposed design, the SHA-3 algorithm are then synthesised into logic gates and netlist using Design Compiler. This project will be focusing on area reduction by RTL implementation, the backend design is not covered.

The area report of the proposed design is then analysed for further improvement. The primary focus of this project is to reduce the area used in synthesis of the previous

SHA-3 hardware design. The specification of the SHA-3 are 256-bit hash output with 1600-bits state of SHA-3 consists of 5x5 state matrix of 64-bit words.

1.5 Thesis Outline

This thesis is organized as follows:

Chapter two will summarize the research information which is based on the previous work of cryptographic hash function. It explains different types of methods used in develop the cryptographic hash function over the hardware implementation. The details of Secure Hash Algorithm 3, its previous hardware implementation and their performance analysis especially on area size will be explained here.

Chapter three will be on the discussion of the method to be applied to implement the SHA-3 cryptographic hash function over the hardware approach. The flow charts, block diagrams and operational methods will be explained in this chapter. Method to further reduce the area of the previous architecture design will also be discussed in this chapter.

Chapter four focuses on the simulation and performance analysis of the results of the implementation approach. The discussion will be made based on the area efficiency performance of the system.

Chapter five concludes the project starting from the planning stage until the stage of implementation. The limitations found in this project will also be concluded in this chapter. Future works which might help in enhancing this project will be suggested here.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter is about the study of cryptographic hash function and the research done previously about the hardware implementation methods of the selected cryptographic hash function (i.e., Secure Hash Algorithm 3). The SHA-3 cryptographic hash function comprises of five step mapping functions. The five step mapping functions comprise a round of KECCAK-*p* [*b, nr*] process. These five step mappings are denoted by θ, ρ, π, χ, and ι. These five different step mappings sections will be discussed here. The previous implemented hardware architecture of SHA-3 cryptographic hash function will also be discussed here. Since smaller area is needed to cater the needs in smaller IoT devices, the performance of area in the previous architecture will be explored.

2.2 Cryptographic Hash Function

A Hash function is a function that takes any amount of input and produces an output of fixed size [18]. Hash functions mainly used for data integrity verification. Hash functions can be utilized to build the security of a system network. When hash function is used correctly, they can be utilized to confirm the trustworthiness of the network packet source. Since a message that has been randomized by hash function is unique and cannot be converted back to the original message, it can be used to verify that the message has not been altered if the hash output of the message at the transmitter and the receiver side

are same [19]. Several types of hash functions are currently in used in the industry today, such as SHA-2 and MD5 [7, 8]. By analyse the usefulness and the security accessible in each function, the user can figure out which algorithm is most appropriate for their application.

The most crucial part of any secure system is the data integrity. Cryptographic hash function can detect unauthorized changes in files by using the generated message digests. This is imperative in shielding the essentially sensitive databases [18]. Hash functions are different from other encryption cryptographic method such as Advanced Encryption Standard (AES) [20]. The generated message digest using hash cryptographic function is non-reversible, whereas the output message (i.e., encrypted and decrypted message) generated using AES is reversible. Hash functions are implemented due to their uniqueness and they cannot be reverted [21]. Hash functions can be combined with encryption to produce special message digests (i.e., Message Authentication Codes) that identify the source of the data. The standard algorithm used today is called Hashed Message Authentication Code [3] which gives confirmation of the data source, and furthermore anticipates against security assaults. HMAC uses a secret in the process to generate the hash, thus nobody is able to foresee the exact digest without the secret value and furthermore the hashed data content cannot be resolved from the hash digest.

The input for hash function is known as the message, whereas the output is known as the digest value. The message length can be varied whereas the digest length is fixed based on the desired digest length. A message digest can be illustrated as the “fingerprint” of the data. For example, there will be no more than one data with the same fingerprint. No same output can be derived from multiple set of different input. A cryptographic hash

function is intended to give extraordinary properties, such as collision and preimage resistance, that are imperative for most of the applications data security [14]. For instance, during a fund transfer transaction, if there is a hacker intended to alter irregular bits of information, this may prompt to wrong finance being exchanged, or perhaps exchange to a wrong account. Under this situation, the hash function can provide the collision resistance which assures that the original message could not have been modified to an alternate message with a similar digest value, or a similar signature. This is a one-way deterministic procedure, which makes the hash function a one-way procedure [13].

One of the most important applications of hash functions is to the construction of MACs (*Message Authentication Codes*). MACs are cryptographic schemes designed to prevent an adversary from impersonating a legitimate user and from modifying a message without the legitimate users noticing it. The hash value of the data is figured and then affixed to the data. During the message retrieval at the destination, the hash value is refigured from the retrieved data. The refigured hash value is compared to the hash value that was appended to the original data. In the event that both the hash values do not match, it implies that the data has been changed. Figure 2.1 shows the generation and comparison of HMAC utilizing a shared secret K [13]. The secret K is utilized in the hash generation process of HMAC so that without knowing the exact secret K, nobody can foresee the exact digest value of the data. The H block in Figure 2.1 is simply the message attached with the secret K. The figure shows the hash output generated in the transmitter side is compared to the hash output generated in the receiver side.

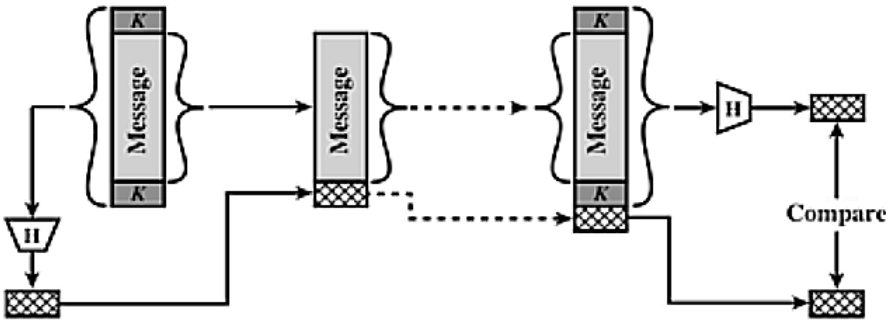

Figure 2.1 Generation and comparison of hash value (HMAC) [13]

Hash functions which are normally used in industry are SHA-1, SHA-256, SHA-512, MD4 and MD5. Recent advances in the cryptanalysis of commonly used standard hash functions reported some collisions and serious vulnerabilities in these algorithms [9-11]. Although no assaults have yet been accounted for on the SHA-2 variations, but because of hardware likenesses to SHA-1 there are fears that SHA-2 could likewise be broken sooner rather than later [22]. To counter the possibility of SHA-2 to be cracked in future, SHA-3 is introduced as a more secure hash function in a competition organized by NIST [14].

2.3 Secure Hash Algorithm 3

This section will explain the details of Secure Hash Algorithm 3. The current Secure Hash Algorithm used in industry is SHA-1 and the SHA-2 family of hash functions specified in Federal Information Processing Standard (FIPS) 180-4 [23]. Secure Hash Algorithm 3 (SHA-3) is a newly formed function that will replace the SHA-1 and SHA-2 family in future. SHA-3 is based on KECCAK [24], which is the algorithm that NIST selected as the winner of the public SHA-3 Cryptographic Hash Algorithm Competition.

The SHA-3 is a hash function, where the input is a binary data or string and the output is a fixed length binary data or string. The SHA-3 input is known as the message, whereas the output is known as the hash or digest value. The SHA-3 family consists of four cryptographic hash functions, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256 [12, 14]. The addition after the dash shows the fixed digest length (e.g., SHA3-512 yields a 512-bit digest). Since SHA-3 provides a similar arrangement of digest lengths with SHA-2, the SHA-3 can be actualized as other options to the SHA-2.

The extendable-output functions XOFs (i.e., SHAKE128 and SHAKE256) are different from hash functions but, as stated in SHA-3 Standard, XOFs are conceivable to be utilized in the same ways, with the adaptability to be adjusted specifically to the prerequisites of individual applications, subject to extra security contemplations. The output of XOFs is adjustable to the desired output length. The suffixes “128” and “256” of the function names indicate the supported security strength [12, 14]. For the hash function to be cryptographically useful it is then a basic requirement that the function be collision resistant, in the sense that it should be infeasible for an adversary to generate a *collision* (i.e., two different messages with the same hash), since hashes have a short fixed length, there will be many messages which have the same hash value, so the hash function is far from being injective but it should look like that to an adversary.

2.3.1 KECCAK-p Permutations

All of the SHA-3 functions utilize the same basic permutation and are methods or modes of operation of the permutation. The permutation is stated in the Federal Information Processing Standards Publication (FIPS PUB 202) as a newly formed family of permutations, the given permutation name is known as *KECCAK-p* [12]. The purpose of this specification is to give the adaptability to alter the security and size parameters in the advancement of any extra operation modes in future documents.

There are two parameters in *KECCAK-p* permutations, which are the number of iterations of step mapping functions, known as a round and the fixed length of strings that undergo permutation, known as the permutation width. The number of rounds is indicated by n_r, whereas the width is indicated by *b*. The *b* range is in (25, 50, 100, 200, 400, 800, 1600) and the n_r is any positive integer. For instance, a *KECCAK-p* permutation which undergoes n_r rounds of step mapping iteration with permutation width *b* is indicated by *KECCAK-p[b, n_r]*. One round of *KECCAK-p* permutation, which indicated by *Rnd,* comprises of a group of five transformations known as step mapping functions. An array of values for *b* bits, known as the state, is repeatedly updated in the permutation. The input data of the permutation is used to initialize the initial value of the state array[12].

2.3.2 State

The *KECCAK-p[b, n_r]* permutation consists of *b* bits, *w* and l, where *w* is *b/25* and l is $log_2(\frac{b}{25})$. The relationship between these variables are given in the Table 2.1.

Table 2.1 KECCAK-p permutation variables' relationship [12]

b	*25*	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

The state array of *KECCAK-p*$[b, n_r]$ can be represented by *A*[*x*, *y*, *z*], a 5-by-5-by- w state matrix, where $0 \leq x < 5, 0 \leq y < 5$, and $0 \leq z < w$. The detailed elements in a state matrix are shown in Figure 2.2, whereas the labeling convention of the coordination for the state matrix is shown in Figure 2.3.

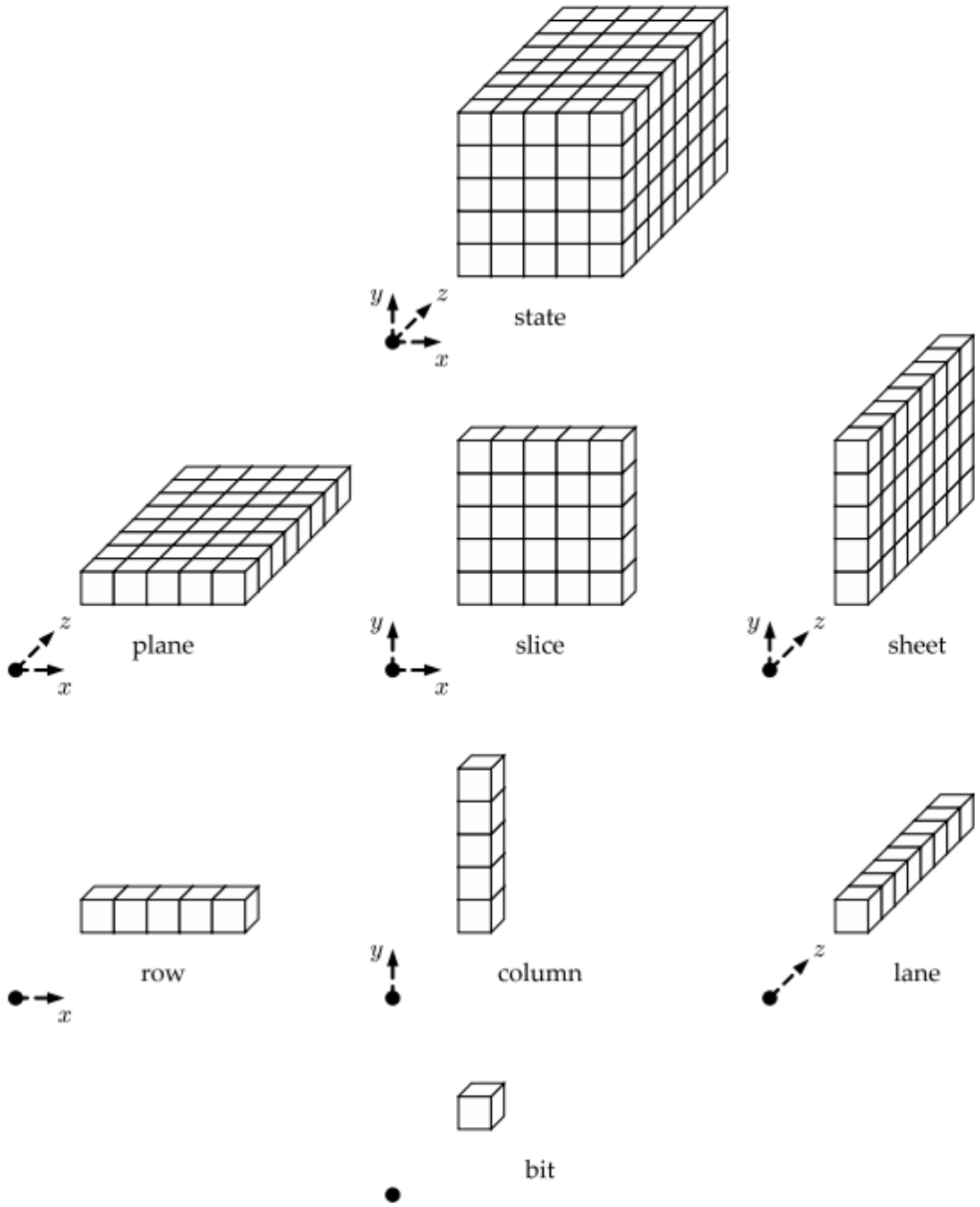

Figure 2.2 The detailed elements in a state matrix [12]

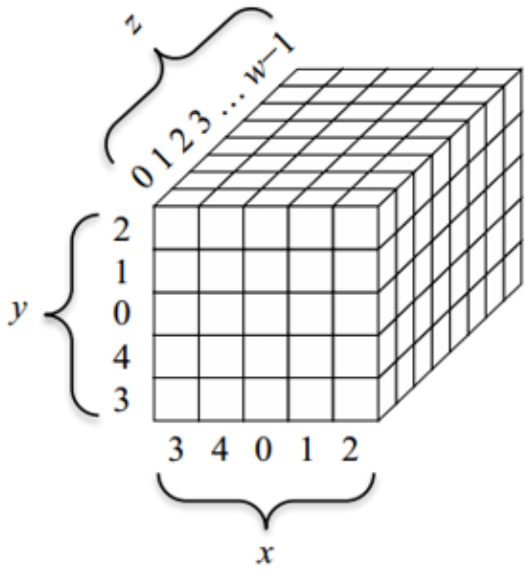

Figure 2.3 The *x*, *y*, and *z* coordination for the elements in a state matrix [12]

2.3.3 KECCAK-p[b, n_r](S)

The final output from *KECCAK-p[b, n_r](S)* permutation given an input string *S* and *b* bits, consists of n_r iterations of Round Function *Rnd*, as shown in the following steps. For example, a string *S* with 1600-bits, from the Table 2.1, *l* will be 6. Thus, n_r will be equal to 24. The string *S* will first be converted to state array *A*. Then the state array *A* will undergo 24 times of Round Function. After that, the state array A' will be converted back to string *S'* with length *b*.

Algorithm to calculate KECCAK-p[*b*, n_r](*S*):

1. Convert *S* into a state array, *A*
2. For i_r from 12 + 2*l* - n_r to 12 + 2*l* – 1, let $A' = Rnd(A, i_r)$
3. Convert A' into a string *S'* of length *b*
4. Return *S*'

2.3.4 Round function

A complete Round Function, *Rnd* consists of five step mapping function, which are Theta (θ), Rho (ρ), Pi (π), Chi (χ), and Iota (ι). The Round Function given a state matrix *A* and a round index i_r is shown in the Equation 2.1. The details of five step mapping function will be explained in the next section.

$$Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A))))), i_r) \quad (2.1)$$

2.3.5 Step Mappings

In this section, the five step mapping functions that result in one round of *KECCAK-p[b, n_r]*, which are Theta θ, Rho ρ, Pi π, Chi χ, and Iota ι, will be explained here based on the SHA-3 specification [12]. Each step mapping function consists of the algorithm where the input is the state matrix *A*, whereas the output is the updated state matrix *A'*. The *b* parameter specifies the size of the state matrix. The operation algorithms of θ, χ and ι consisted of XOR operations with the bits in the state, whereas the operation algorithms of ρ and π consisted of rearrangement of bit position by rotation of the bits in *x*, *y* or *z* coordinates of the lane.

Specification of θ(*A*)

The effect of θ is to perform XOR operation in each bit of the state matrix with the parities of two columns in the state matrix as shown in Figure 2.4. In the illustration in Figure 2.4, the parity or the XOR operation of all the bits in the specified column is indicated by the summation symbol, ∑. For a single bit *A*[*x*, *y*, *z*], the x-coordinate of the first column is (*x*+1) mod 5, and the z-coordinate is *z*, whereas the x-coordinate of the

second columns is (x-1) mod 5, and the z-coordinate is (z-1) mod w. The algorithm for θ is shown in the Equations 2.2 – 2.4.

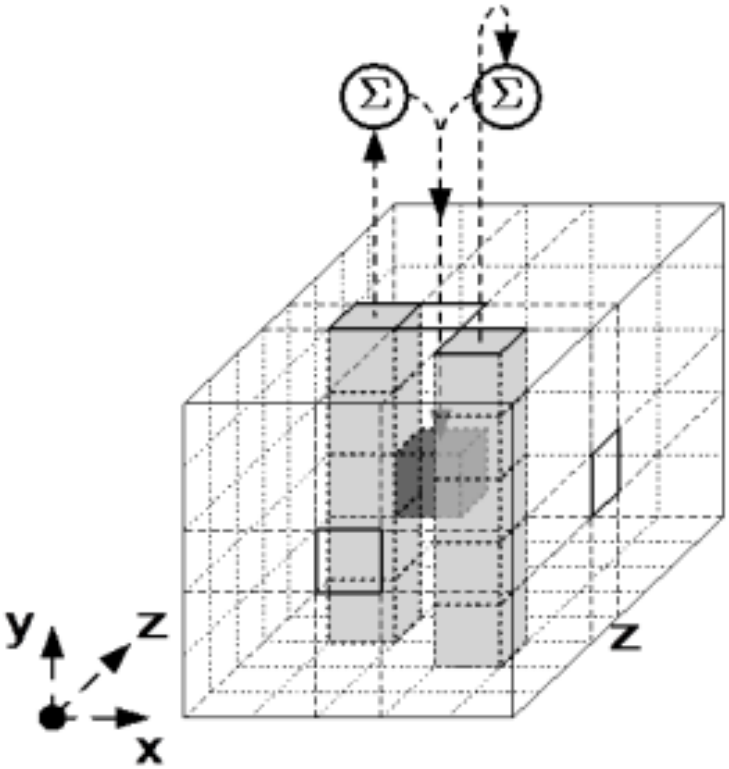

Figure 2.4 Implementation of θ on a single bit [12]

Algorithm to calculate Theta, *θ(A)*:

Input: State Matrix $\boldsymbol{A}$

Output: State Matrix $\boldsymbol{A}'$

1. For 0≤x<5 and 0≤z<w, let

$$C[x,z] = \boldsymbol{A}[x,0,z] \oplus \boldsymbol{A}[x,1,z] \oplus \boldsymbol{A}[x,2,z] \oplus \boldsymbol{A}[x,3,z] \oplus \boldsymbol{A}[x,4,z] \quad (2.2)$$

2. For 0≤x<5 and 0≤z<w let

$$D[x,z] = C[(x-1)\ mod\ 5, z] \oplus C[(x+1)\ mod\ 5, (z-1)\ mod\ w] \quad (2.3)$$

3. For 0≤x<5, 0≤y<5, and 0≤z<w, let

$$\boldsymbol{A}'[x,y,z] = \boldsymbol{A}[x,y,z] \oplus D[x,z] \quad (2.4)$$

Specification of ρ(*A*)

The effect of ρ is to perform rotation of bits in each lane by an offset, depending on the *x* and *y* lane coordination. The z-coordinate of each bit in the lane is moved by a calculated offset. Modulus is used to prevent z-coordinate that larger than the lane size. The algorithm is shown in the Equations 2.5 – 2.6.

Algorithm to calculate Rho, *ρ(A)*:

Input: State Matrix $\boldsymbol{A}$

Output: State Matrix $\boldsymbol{A}'$

1. For $0 \leq z < w$, let $\mathbf{A}'[0, 0, z] = \mathbf{A}[0, 0, z]$
2. Let $x = 1$ and $y = 0$
3. For t from 0 to 23, for $0 \leq z < w$, let

$$\boldsymbol{A}'[x, y, z] = \boldsymbol{A}[x, y, (z - (t+1)(t+2)/2) \bmod w] \quad (2.5)$$

$$(x, y) = (y, (2x + 3y) \bmod 5) \quad (2.6)$$

4. Output $\mathbf{A}'$

Figure 2.5 shows the implementation of ρ when $w = 8$ and $b = 200$. The x and y coordination labelling is stated in Figure 2.2 and 2.3. The black dot shows the bit with 0 z-coordination, whereas the grey cube shows the final position of the corresponding bit after shifted by the calculated offset from the ρ algorithm. All the other bits of the lane will be shifted circularly by the similar offset. If the final z-coordinate is larger than the lane size, the offset is reduced by modulus the lane size. For example, the offset for the lane A[3, 2], which located at the top left corner, has an offset of 1 bit (i.e., 153 mod 8 = 1), therefore in this case, the last bit (i.e., z coordinate of 7) is shifted to the front bit (i.e., z coordinate of 0).

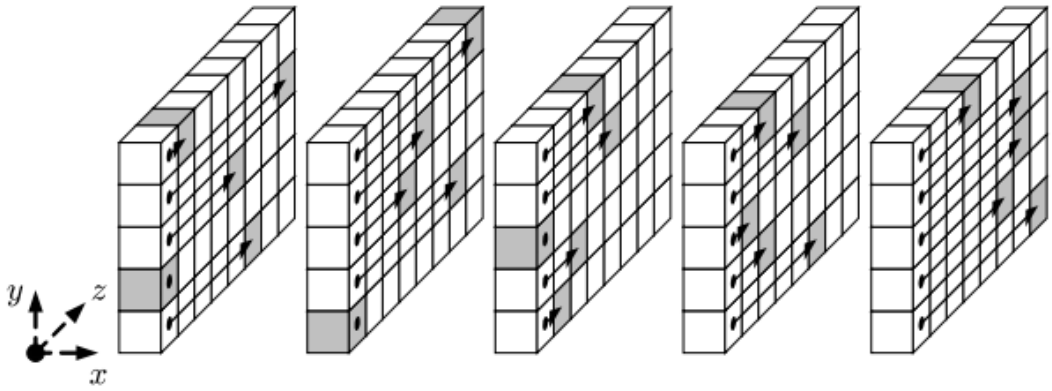

Figure 2.5 Implementation of ρ for the case *b*=200 and *w*=8 [12]

Specification of π(*A*)

The implementation of π is to remap the lane positions as shown in Figure 2.6. The x and y coordination labelling is stated in Figure 2.2 and 2.3. For example, the center of the slice is corresponding to the bit with coordinates *A[0, 0]*. The algorithm is shown in the Equation 2.7.

Algorithm to calculate Pi, *π(A)*:

Input: State Matrix ***A***

Output: State Matrix $\boldsymbol{A'}$

1. For 0⩽x<5, 0⩽y<5, and 0⩽z<w, let

$$A'[x, y, z] = A[(x + 3y) \bmod 5, x, z] \quad (2.7)$$

2. Output A′

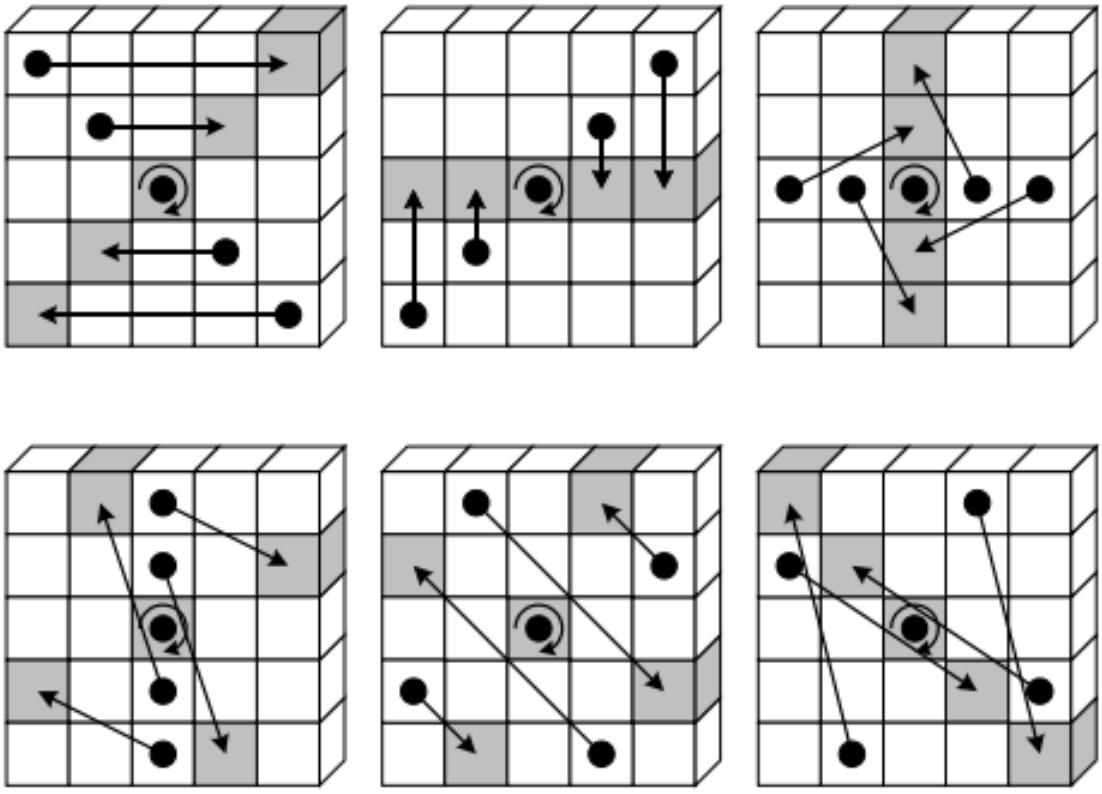

Figure 2.6 Implementation of π on a slice [12]

Specification of χ (*A*)

The implementation of χ is to perform XOR operation on each bit with two other bits in the same row by using non-linear function as shown in Figure 2.7. The algorithm is shown in the Equation 2.8. The dot in the Equation 2.8 is a multiplication, which is also similar to the Boolean "AND" operation in this case.

Algorithm to calculate Chi, *χ (A)*:

Input: State Matrix ***A***

Output: State Matrix $\boldsymbol{A'}$

1. For 0⩽x<5, 0⩽y<5, and 0⩽z<w, let

$$\mathbf{A}'[x,y,z] = \mathbf{A}[x,y,z] \oplus ((\mathbf{A}[(x+1) \bmod 5, y, z] \oplus 1) \cdot \mathbf{A}[(x+2) \bmod 5, y, z]) \quad (2.8)$$

2. Output A′

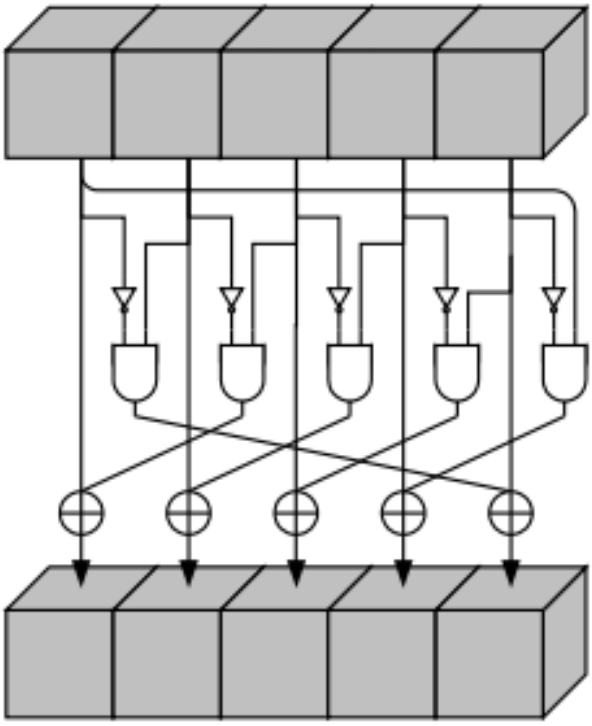

Figure 2.7 Implementation of χ on a row [12]

Specification of ι(A, i_r)

The implementation of ι is to alter part of the bits in Lane (0, 0) depending on the round index i_r. Any other 24 lanes will not be affected by ι. The round index values are specified in the second step of the *KECCAK-p[b, n_r](S)* in section 2.3.3. There is a parameter in specification of ι called round constant *RC*, which is dependent on the algorithm *rc(t)*. The algorithm for *rc(t)* is shown in the Equations 2.9 – 2.14.

Algorithm to calculate Round Constant, *rc(t)*:

Input: Integer ***t***

Output: Round Constant ***rc(t)***

1. If *t* mod 255 = 0, return 1
2. Let *C* = 10000000
3. For *i* from 1 to *t* mod 255, let

$$C = 0||C \quad (2.9)$$

$$C[0] = C[0] \oplus C[8] \quad (2.10)$$

$$C[4] = C[4] \oplus C[8] \quad (2.11)$$

$$C[5] = C[5] \oplus C[8] \quad (2.12)$$

$$C[6] = C[6] \oplus C[8] \quad (2.13)$$

$$C = Trunc_8[C] \quad (2.14)$$

4. Output $C[0]$

The algorithm for the specification ι takes in the input state *A* and the round index i_r. The round constant *RC* is generated from the *rc(t)* function above. The steps for the specification ι is shown in the Equations 2.15 – 2.16.

Algorithm to calculate Iota, $\iota(A, i_r)$:

Input: State Matrix ***A,*** Round Constant ***rc(t)***, Round Index i_r

Output: State Matrix $\boldsymbol{A'}$

1. For 0⩽x<5, 0⩽y<5, and 0⩽z<w, let $\mathbf{A'}[x, y, z] = \mathbf{A}[x, y, z]$

2. Let $RC = 0^w$

3. For $0 \leqslant j < l$, let

$$RC[2^j - 1] = rc(j + 7i_r) \quad (2.15)$$

4. For 0⩽z<w, let

$$\mathbf{A'}[0, 0, z] = \mathbf{A'}[0, 0, z] \oplus RC[z] \quad (2.16)$$

5. Output A'

2.3.6 Sponge Construction

The sponge construction is indicated by *SPONGE[f, pad, r](N, d*). Sponge construction is a framework for specification of *KECCAK-p[b,* n_r *](S)* [12]. Sponge function permits to simplify methods of use where dedicated constructions would be needed for fixed-output-length. Figure 2.8 shows the general sponge construction framework [25]. Given a function *f*, a rate *r*, a padding rule *pad*, and an input string *N* with *d* bits, the sponge construction will give a truncated output *z*. Function *f* is a function operates on fixed length *b*-bits state. Rate *r* is a positive whole number where $r < b$, whereas capacity *c* is a positive whole number where $c = b - r$. The padding rule *pad* is used to produce padding (i.e., a bit string with a suitable length to attach to another bit string). The padded input data are first "absorbed" into the state matrix of the function, then processed state matrix are then "squeezed" out and then truncated to become *z* output. The detailed explanation of sponge construction framework is discussed in [25].

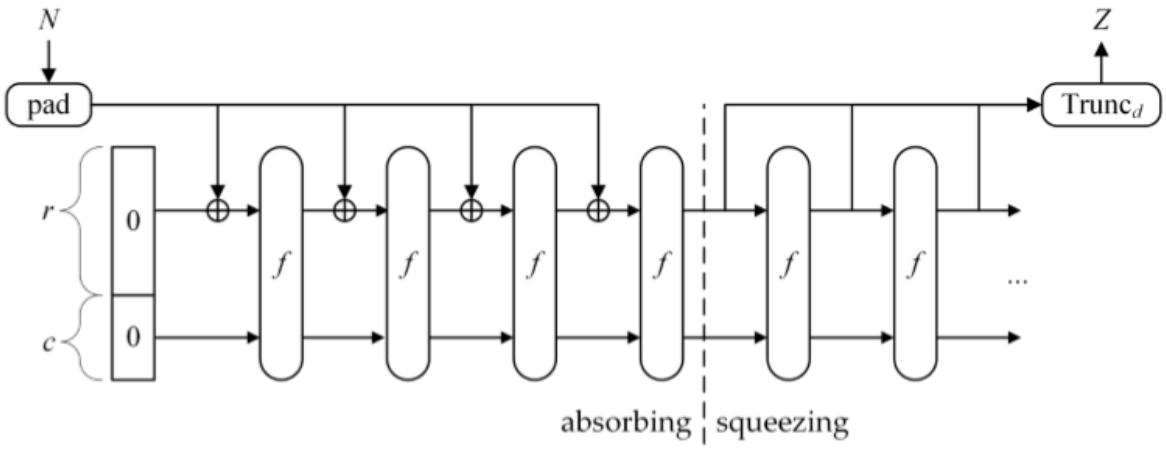

Figure 2.8 The architecture for sponge construction [25]

For instance, the implementation of KECCAK Secure Hash Algorithm 3 in sponge construction with 1600-bits (i.e., *b*-bits) state array, 24 (i.e., n_r) iteration of round function, 256 (i.e., *c*) capacity, 1344 (i.e., *r*) rate, and *S* input string is specified in

Equation 2.17. The choices of capacity *c* can be chosen based on the number of *b*-bits. The padding rule used in sponge construction for KECCAK SHA-3 is *pad10*1*, where the algorithm is explained in the Equations 2.18 – 2.19. The asterisk in "*pad10*1*" shows that the bit "0" is either discarded or repeated to yield an output bit string with desired length that will be appended to the string *S*.

*KECCAK[256] = SPONGE[KECCAK-p[1600, 24](S), pad10*1, 1344]* (2.17)

Algorithm to produce padding *pad10*1(x, y)*:

Input: Positive integer for desired length *y*, positive integer for current length *x*

Output: String *D* such that *x* + length(*D*) is the multiple of *y*

1. $Let\ i = (-x - 2) \bmod y$ (2.18)
2. Output $D = 1 \,||\, 0^i \,||\, 1$ (2.19)

The operation flow of for the equation of *KECCAK[c]* start with appending the input *S* string to 1600-bits by using the padding rule *pad10*1*. The appended string is then "absorbed" into the state array *A* and undergo 24 iterations of round function as explained in section 2.7. After 24 iterations of round function, the output is then "squeezed" out and truncated as a digest output with desired length (e.g., 256-bits digest output).

2.4 Hardware implementation of Secure Hash Algorithm 3

Several previous works had been done on the hardware implementation of SHA-3 algorithm [13, 16, 17, 26-28]. Their works can be summarized with the following procedure. SHA-3 belongs to a sponge function which consists of two main parameters, which are capacity *c* and bit rate *r* as explained in Section 2.3.6. The *b* bits is the sum of bit rate *r* and capacity *c* which determine the SHA-3 width used in the permutation. The

maximum value of *b* bits is 1600-bits. Depending on the required length for digest output, suitable *c* and *r* can be chosen [13, 27].

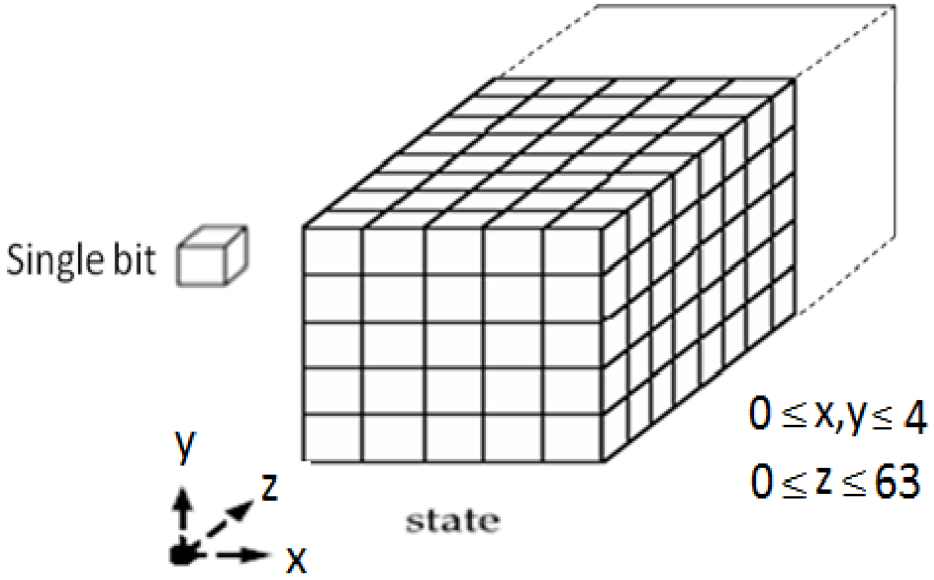

Figure 2.9 State Matrix of Secure Hash Algorithm 3 [13]

The SHA-3 process comprises of three main phases, which are initialization, absorbing and squeezing. Firstly, the state array of SHA-3 is initialized with all zeros. During the absorbing phase, XOR operation is performed between the message with *r*-bits block and the current state array, then 24 iterations of the *round* functions (as explained in Section 2.3.5) are executed. Finally, the state array is truncated as a digest output with desired length in the squeezing phase. A total of 24 iterations of step mapping function of SHA-3 will be carried out. Each iteration of step mapping function comprises of 5 stages, which are Theta θ, Rho ρ, Pi π, Chi χ and Iota i. The general equations used for the step mapping function are shown in Equations 2.20 - 2.25 [13, 27]. The Equations 2.20 – 2.25 are based on the lane of the state array, whereas the equations of SHA-3 in Section 2.3.5 are based on the bit of the state array. The initialization and squeezing phase are similar for the previous works. The main difference is on the hardware architecture implementation for the absorbing phase, which will led to different results in hardware

performance in term of throughput and area. Some of the hardware architecture of previous works will be discussed here.

Theta (θ) Stage, for $0 \leq x < 5$ and $0 \leq y < 5$:

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \quad (2.20)$$

$$D[x] = C[x-1] \oplus ROT(C[x+1],1) \quad (2.21)$$

$$A'[x,y] = A[x,y] \oplus D[x] \quad (2.22)$$

Where *ROT* is the cyclic shift function, *A* is the input state matrix, *C* and *D* are the intermediate state matrixes, and A' is the output state matrix.

Rho (ρ) and Pi (π) Stage, for $0 \leq x < 5$ and $0 \leq y < 5$:

$$B[y, 2x+3y] = ROT(A[x,y], r[x,y]) \quad (2.23)$$

Where *ROT* is the cyclic shift function, *r [x, y]* is the cyclic shift offset, *A* is the input state matrix, and B is the output state matrix.

Chi (χ) Stage, for $0 \leq x < 5$ and $0 \leq y < 5$:

$$A[x,y] = B[x,y] \oplus ((NOT\ B[x+1,y])AND\ B[x+2,y]) \quad (2.24)$$

Where B is the input state matrix, and A is the output state matrix.

Iota (i) Stage:

$$A'[0,0] = A[0,0] \oplus RC \quad (2.25)$$

Where RC is the round constant, *A* is the input state matrix, and A' is the output state matrix.