# An Efficient Implementation of "KECCAK" SHA – 3 Candidate on FPGA

**Submitted by**
**Muhammad Salman Mobin**

**Supervised by**
**Dr. Arshad Aziz**

**Thesis Submitted in Partial Fulfillment of the Requirement for the Degree of Master of Science in Electrical Engineering with Specialization in Communication**

**At The**

**Department of Electronic and Power Engineering**
**Pakistan Navy Engineering College, Karachi**
**National University of Science and Technology**
**H – 12, Islamabad, Pakistan**
**March 2012**

**Dedicated to Parents and Siblings**

## Acknowledgments

All thanks and praises for ALLAH all alone. I am very great full to ALLAH Almighty who bestowed his blessing on me and provides me the resources and energy to complete this thesis without which it was impossible to complete. And without his help I wouldn’t have completed any part of my thesis.

I am very grateful to my supervisor Associate Professor Dr. Arshad Aziz, who helps me and provide his expert opinions on each and every step of my thesis. He encourages me to believe in myself that I can do this thesis it is not as hard as it  looks like. Then I would like to thanks to all my GEC members Dr. Parveez Akhtar, Dr Athar Mehboob and Dr sameer Qazi.

I am very great full to my friends who help me in documentation and provide me the stuff with I wanted to complete my thesis. My special thanks to my parents, they are always there for me when I needed their help. They motivate me in my worst time when I was down I have no idea what to do they were always there for me to find a way out of worst situation.

# Abstract

Cryptographic hash function takes any arbitrary input and produces a fixed length output. It is use for integrity protection, conventional message authentication and digital signatures. The advantage is that if there is change in message bits the receiver will comes to know that message has been altered. A good cryptographic hash function must have strong collision property.

Secure hash algorithm (SHA – 3) is a group of cryptographic hash algorithm publishes by National Institute of Standards and Technology (NIST). The category of SHA – 1 were vulnerable to Hash Collision attack and series of attacks was carried out on SHA – 1.

Implementation of cryptographic hash function on reconfigurable devices such as field programmable gate array (FPGA) provides efficient features of both hardware and software.

The aim of this thesis is to implement one of the SHA – 3 candidates “KECCAK” on FPGA. We will be implementing the algorithm efficiently on FPGA. For optimization of the algorithm we use external memory block with main logic block. The advantage of using external memory block is that we can share this memory block with other algorithms in order to optimize the overall area of Chip on which these algorithms are burned.

# Table of contents

# List of figures

# List of Tables

# Chapter # 01

# 1 Introduction

## 1.1 Background

In recent years the algorithms which were not approved by National Institute of Standards and Technology were successfully attacks and it was found that they were vulnerable to Hash Collision. Series of attack was publishing against secure hash algorithm – 1 (SHA – 1). NIST decided to develop one or more hash algorithm through a public competition. The selected algorithm of the competition will be finalized in spring 2012.

Implementing algorithm in efficient manner means to efficiently utilize the software algorithms and the underneath hardware environment. Our efficient implementation utilizes less hardware area on Field Programmable Gate Array (FPGA).

## 1.2 Secure Hash Algorithm (SHA)

Secure Hash Algorithm (SHA) stands for Secure Hashing Algorithm is a group of hash functions published by the National Institute of Standards and Technology as a US Federal Information Standard. All of the current SHA algorithms are developed by the NSA [1].

SHA-0: A 160-bit hash function published in 1993. It was quickly withdrawn due to an undisclosed flaw. It was replaced by SHA-1[2].

SHA-1: A 160-bit hash function that is similar to the earlier MD5 algorithm but more conservative. It is developed by the National Security Agency to be a part of the Digital Signature Algorithm. It is the most widely used SHA algorithm [2].

SHA-2: It has two similar hash functions. It comes with four different sizes for the output, 224, 256, 384, and 512-bit. The 224-bit and 384-bit versions of SHA-2 are simply the 256-bit and 512-bit versions with truncated outputs [2].

SHA-3: This future hashing function is still under evaluation. The algorithm will be developed by choosing different algorithms to a public competition. The final decision is expected to be announced in 2012 [2].

The evolution of SHA – 3 came into existence when in 2005; Prof. Wang was able to find a hash collision in SHA 1 [3]. Crypt attacks were not as effective on SHA 2. NIST decided to develop one or more hash functions as the transition from SHA-1 to the approved SHA-2 family is made.

NIST chose to develop the new hashing algorithm through a public competition. The competition will accept algorithms from any person or organization [4]. By the end of 2008, 64 algorithms was submitted out of which 51 was selected for first round.

In the second quarter of 2009, candidates that advanced to the second round were announced. Three broad categories of the evaluation criteria were used to evaluate the first round candidates: security, cost and performance, and algorithm and implementation characteristics in software. Five candidates were selected among the 14 competitors to qualify to third round. The winner of the third round will be selected in 2012 [2].

## 1.3 Aim of thesis

The aim of this thesis is to implement one of the SHA – 3 candidates “KECCAK” on FPGA. We will be implementing the algorithm efficiently on FPGA. For optimization of the algorithm we use external memory block with main logic block. The advantage of using external memory block is that we can share this memory block with other algorithms in order to optimize the overall area of IC on which these algorithms are burned.

## 1.4 Thesis outline

In chapter 2 we will be discussing about cryptographic hash functions. In chapter 3 we will present KECCAK algorithm in detail. In chapter 4 an overview of FPGA will be discuss. Chapter 5 we will be discussing about our work. In chapter 6 results of our thesis will be provided. Chapter 8 deals with the conclusion and future work.

# CHAPTER # 02

# 2 Cryptographic Hash Function

## 2.1 Hash Function

### 2.1.1 Description

A cryptographic hash function is a deterministic procedure that receives a variable size data and produces a fix size block of data often called digest. For generating hash value we use the function of the form:

$$h = H(M) \qquad \text{Equation 2.1}$$

Where M is the variable length message and H (M) is fixed length hash value.

If any accidental or intentional change in data occurs during the transmission, the hash computed on receiver side will be completely change and hence the receiver comes to know that any error has occur in data during transmission or data is alter during transmission.

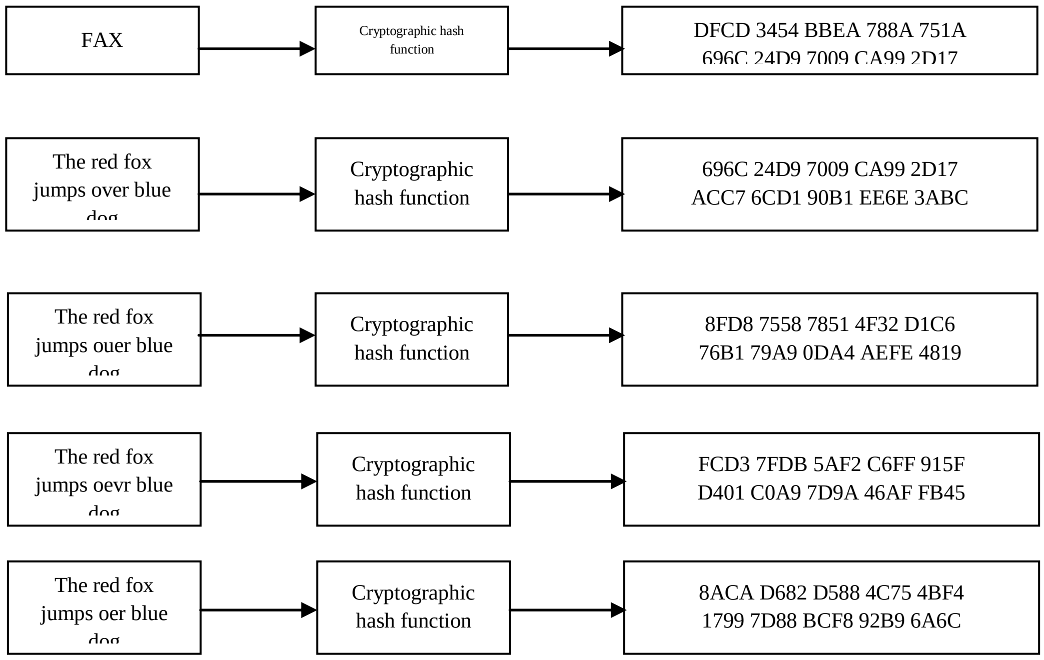
FAX
Cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over blue dog
Cryptographic hash function
696C 24D9 7009 CA99 2D17 ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps ouer blue dog
Cryptographic hash function
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
The red fox jumps oevr blue dog
Cryptographic hash function
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps oer blue dog
Cryptographic hash function
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

Figure 2.1: A cryptographic hash function.

A slight change in data results in drastic change in hash output As we can see in figure 2.1; a slight change in data results in complete change in hash output so if any deliberately or accidently change in data occurs the receiver will come to know.

## 2.2 Requirement for hash Function

The following are the requirements for good hash function.

- H can be applied to a block data of any size.
- H produces a fixed length output.
- H (M) is easy to compute for any M, making both hardware and software implementation practical.
- For any given *h* it is computationally infeasible to find M such that H (M) = *h* sometime also known as one way hash function.
- It is computationally infeasible to find two different messages *M* and *N* such that their hash digests are same. Sometimes as strong collision resistance.

## 2.3 Applications

The following are the application of cryptographic hash functions

### 2.3.1 Password protection

One of the most common uses of secure hash is the password protection. Almost all modern computers requires password for authentication. User has to first setup the password that password is stored on hard drive as a text message for cross referencing the password next time when user log on to the system. The problem is that if someone physically removes the hard drive and extract the password from the hard drive later on then the system is compromise. To prevent this cryptographic hash function must be use. The password is run through hash function first and the hash is stored on hard drive when next time the same person try to access the system the password is first run through the cryptographic hash function and then cross reference to initial hash stored on hard drive. In whole process the actual password is never stored on hard drive actually it's the hash value which is stored on hard drive.

### 2.3.2 Data integrity

Another application of cryptographic hash is in data integrity. This is to ensure that data of interest is not altered during the transmission. If so then the hash value will be completely changed and user will be able to indentify that the data has been altered or any error has occurred in data during the transmission. For this the cryptographic hash must have strong collision resistance property in order to avoid collision that means that two different message does not have same hash value.

# CHAPTER # 03

# 3 KECCAK Algorithm

## 3.1 Overview of KECCAK Algorithm

KECCAK make use of sponge construction, hence belongs to sponge construction family. KECCAK was design by designed by Guido Bertoni, Joan Daemen, Gilles Van Assche (STMicrosystems) and Michaël Peeters (NXP Semiconductors). The construction of KECCAK is based on permutation $f$ from a set of seven permutations that should not have any structural distinguishers. The version we choose of KECCAK operates at 1600 bit state.

## 3.2 The Sponge Construction

The sponge construction (Figure 3.1) is simple iterative procedure for building function $f$ with variable length input and arbitrary length output with fixed length transformation of $f$ operating at fixed number of bits [7]. The sponge construction operates at b = r + c. where r is the number of bits and c is the capacity. The input message is first padded and divided into r number of bits before inputting into the sponge function. The sponge construction proceeds in two stages, the absorbing phase and squeezing phase.

1. In absorbing phase the r bits of input message is XORED with r bits of the state and passes through function $f$ and this process continues until all input message is not entered into absorbing phase, after that it switches to squeezing phase.
2. In squeezing phase the first r bits of state are returned as output interleaved with function $f$. The number of output bits are selected at user will.

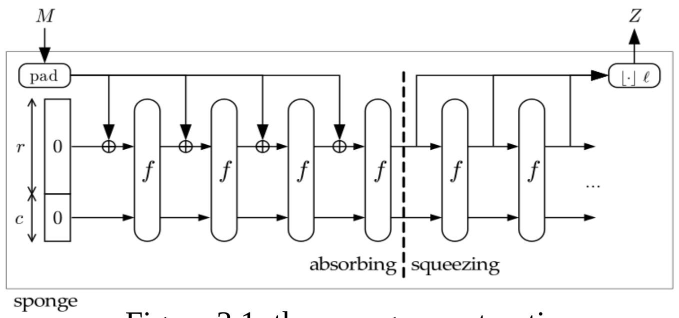

Figure 3.1: the sponge construction

## 3.3 Bit padding

Bit padding is done to make the size of the data equivalent to the input state of the machine for example in case of KECCAK we know that input data must be of 1600 bits well what if the data is less than 1600 bits, the solution is bit padding in this we add a single bit ‘1’ at the end of the message bits and then followed by a series of ‘0’ in order to complete the size of the data or message.

Original message = [10101111 10110110 0111100] (23 bits)

Padded message = [10101111 10110110 0111100**1 00000000**] (32 bits)

As we can see in above example also that our requirement was 32 bit input data but the size of input message was 23 bits so here according to bit padding ‘1’ bit was add and the followed by eight ‘0’ bits in order to complete the block of the message.

For SHA 384 and SHA 512 the padded message is 1024 bits [8]. The message should be padded before entering into KECCAK algorithm. In our case message should be in the multiple of 1024 bits before passing through the KECCAK rounds.

## 3.4 Some terminology of KECCAK

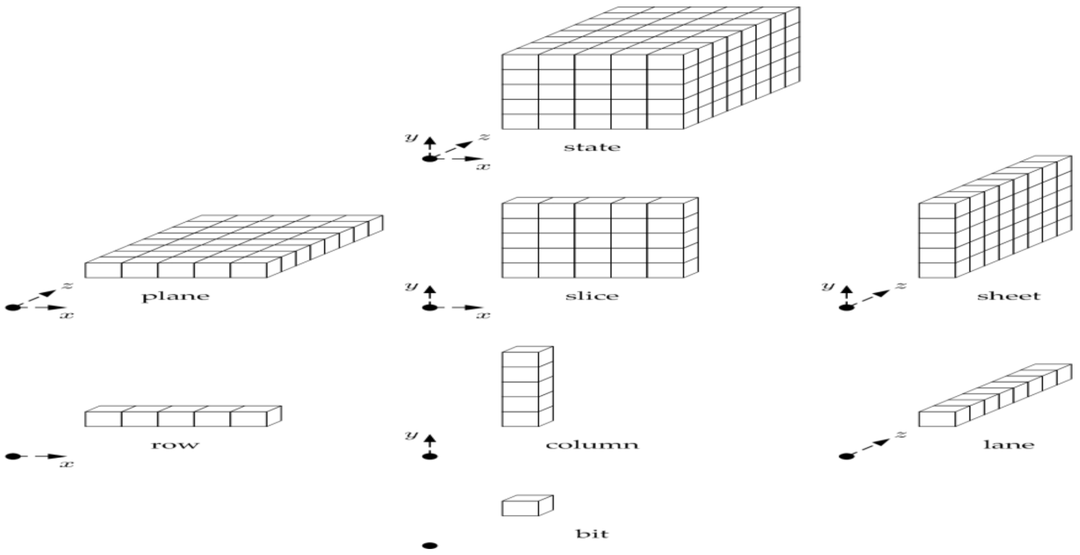

Figure 3.2: terminology of KECCAK

In figure 3.2 some terminology of KECCAK are illustrated. A state consists of 1600 bits. A plane consists of 320 bits in x and z axis. A slice consists of 25 bits in x and y axis. A sheet contains 320 bits in y and z axis. A row contains 5 bits in x axis. A column contains 5 bits in y axis. A lane contains 64 bits in z axis.

## 3.5 KECCAK Algorithm

KECCAK make use of sponge construction, hence belongs to sponge construction family. The version we choose of KECCAK operates at 1600 bit state. Keccak *f* consist of 24 simple rounds. Each round has 5 steps (theta ϴ, Rho ƿ, Pi π, Chi χ, Iotaι). The input and output of keccak round is 5 x 5 matrices whose entries are 64 bit words [6]. The steps of KECCAK are as follows;

1. Theta (ϴ)
2. Chi (χ)
3. Rho (ƿ)
4. Pi (π)
5. Iota (ι)

### 3.5.1 Theta (ϴ) step

The operation perform in theta step is as follows;

$$R[x] = S[x,0] \oplus S[x,1] \oplus S[x,3] \oplus S[x,4] \qquad 0 < x < 4 \qquad \text{Equation 3.1}$$

$$T[x] = R[x-1] \oplus ROT\,(R[x+1],1) \qquad 0 < x < 4 \qquad \text{Equation 3.2}$$

$$S[x,y] = S[x,y] \oplus T[x] \qquad 0 < x < 4 \qquad \text{Equation 3.3}$$

In ϴ step S denotes complete permutation state where R[x] and T[x] are used to store intermediate values, S [x, y] represents the lane. First each lane is XORED with other lane as we can see above in mathematical form; results are stored in intermediate register name R[x]. Then intermediate values are first rotated one bit the XORED with pervious value of array and result is stored in T[x]. Finally theta output is produced by XORING initial value of lane S[x, y] with intermediate value stored in T[x]. Note that all operations in x, y plane are in modulo 5. In theta step 50 bitwise XOR and 5 rotations are performed. Theta step is illustrated in figure 3.3.

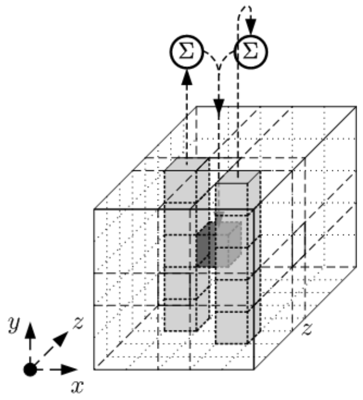

Figure 3.3: The theta step [9]

### 3.5.2 Chi (χ) step

The operation performed in χ step is as follows;

$$S\,[x, y] = S\,[x, y] \oplus ((NOT\ S\,[x + 1, y])\ \&\ S\,[x + 2, y]) \qquad \text{Equation 3.4}$$

The output from Θ step is used as input for chi step. In this a particular column is XORED with result of AND operation performed with next column and negation of next column as you can see in mathematical form above. χ step is illustrated in figure 3.4

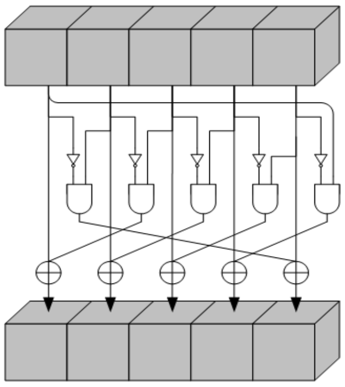

Figure 3.4: The Chi step [9]

### 3.5.3 Rho ($p$) step

The mathematical expression for ρ step is as follows

$$S[x][y][z] = S[x][y][z - (t + 1)(t + 2)/2] \qquad \text{Equation 3.5}$$

Where t satisfying $0 \leq t < 24$ and $t = -1$ if $x = y = 0$

Output from χ step is used as input for ρ step. The ρ step is basically the shifting of the bits in z direction. The shifting of the bits are stated in table 3.1 given below

Table 3.1: transformation of bits in z direction

| **XY** | **X = 3** | **X = 4** | **X = 0** | **X = 1** | **X = 2** |
|---|---|---|---|---|---|
| **Y = 2** | 25 | 39 | 3 | 10 | 43 |
| **Y = 1** | 55 | 20 | 36 | 44 | 6 |
| **Y = 0** | 28 | 27 | 0 | 1 | 62 |
| **Y = 4** | 56 | 14 | 18 | 2 | 61 |
| **Y = 3** | 21 | 8 | 41 | 45 | 15 |

### 3.5.4 The Pi (π) step

Π step can be mathematically expressed as follows;

$$S\,[x, y] = S[y, 2x + 3y] \qquad \text{Equation 3.6}$$

The output from ρ step is used as input for π step. Π step is basically cyclic shifting of lanes according to equation 3.6. Π step is illustrated in figure 3.5 we can see clearly the shifting of the lanes.

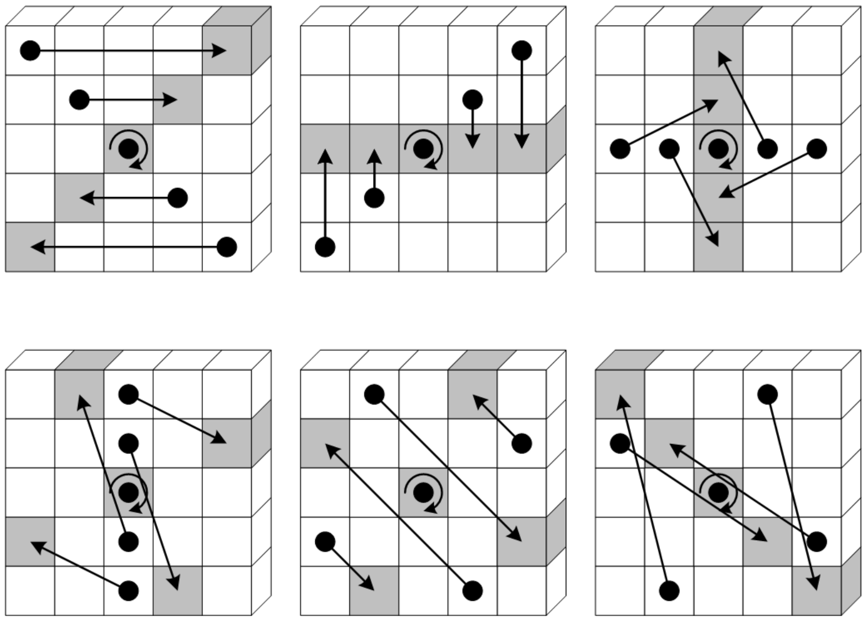

Figure 3.5: The pi step [9]

## 3.5.5 The Iota (ι) step

I step can be mathematically expressed as follows

$$S\,[0,0] = S\,[0,0] \oplus RC \qquad \text{Equation 3.7}$$

The ι step is just basically the XORING a round constant with lane S [0, 0] received from π step. There are 24 round constant, one round constant for each round. Round constant are expressed in table 3.2 below

Note that order of the step is not important you can perform any step first, output of the KECCAK function does not depend on the order of the step.

Table 3.2: Round constants and their values

| S. No | Round constant | S. No | Round constant value |
|---|---|---|---|
| RC [1] | 0000000000000001 | RC [13] | 000000008000808B |
| RC [2] | 0000000000008082 | RC [14] | 800000000000008B |
| RC [3] | 800000000000808A | RC [15] | 8000000000008089 |
| RC [4] | 8000000080008000 | RC [16] | 8000000000008003 |
| RC [5] | 000000000000808B | RC [17] | 8000000000008002 |
| RC [6] | 0000000080000001 | RC [18] | 8000000000000080 |
| RC [7] | 8000000080008081 | RC [19] | 000000000000800A |
| RC [8] | 8000000000008009 | RC [20] | 800000008000000A |
| RC [9] | 000000000000008A | RC [21] | 8000000080008081 |
| RC [10] | 0000000000000088 | RC [22] | 8000000000008080 |
| RC [11] | 0000000080008009 | RC [23] | 0000000080000001 |
| RC [12] | 000000008000000A | RC [24] | 8000000080008008 |

## 3.6 Summary

In this chapter KECCAK algorithm is discuss in detail. KECCAK algorithm uses sponge construction. Input to the KECCAK algorithm must be first padded then enter to the algorithm. The KECCAK *f* consist of 24 rounds. Each round has 5 steps (Theta, Chi, Rho, Pi and Iota). Input data is passed through these 5 steps 24 times. The order of the step is not important you can perform any step first.

# CHAPTER # 04

# 4 Field Programmable Gate Array (FPGA)

## 4.1 Introduction

Field-Programmable Gate Array (FPGA) is an FPD featuring a general structure that allows very high logic capacity [10]. FPGA is a reconfigurable device in which user can reconfigure the program whenever he wants. It's give the flavor of Application specific integrated circuit (ASIC) but with additional feature of upgrading the program stored in it whenever user want.

## 4.2 Advantages of FPGA

FPGA offer a cost effective solution compared to ASIC due to their reconfigurable nature with added benefits of short time to market [11]. Implementing the program on a programmable device gives advantage of upgrading your program whenever you need in order to increase performance and efficiency.

Upgrading cost for FPGA based system is negligible as compared to upgrading cost of ASIC. More over it consumes less power than ASIC.

## 4.3 XILINX FPGA

The platform we choose to implement our program is the Xilinx FPGA. Xilinx holds the largest market share of FPGA. Most researchers preferred Xilinx FPGA for implementing their program as it has more features and it is friendlier to use. XILINX has two main FPGA families; high volume Spartan series and high performance Virtex series.

## 4.4 Virtex Series

Virtex series includes Virtex, Virtex E, Virtex EM, Virtex II, Virtex II Pro, Virtex 4, Virtex 5 and Virtex 6 series. The virtex family is focused on system on a chip (SOC) concept, they includes up to two Embedded IBM PC cores. Some members of virtex 5, virtex 4 and virtex II pro contains power PC cores.

## 4.5 Virtex 5 FPGA

Virtex 5 offers more optimize logic in four domains that provide wide selection of devices with I/Os, hardened IP blocks for logic intensive, embedded processing, serial connectivity applications, optimal mix of logic and digital signal processing (DSP) [12]. It contains 6 input LUT with dual output capability [13], means that you can use one 6 input LUT to apply the logic of two 5 input LUTs.

### 4.5.1 Virtex-5 Family Devices

- Virtex-5 LX for high performance logic.
- Virtex-5 LXT for high performance logic with serial connectivity.
- Virtex-5 SXT for high performance DSP with serial connectivity.
- Virtex-5 FXT for embedded processing with serial connectivity.

### 4.5.2 Architecture over view

The Virtex 5 family consists of six fundamental elements;

- Configurable logic blocks (CLBs)
- Input output blocks (IOBs)
- Block RAM (BRAM)
- Clock resources.
- Digital clock manager (DCM)
- Digital signal processing slice (DSP48E)

#### 4.5.2.1 Configurable logic blocks (CLBs)

In a CLB there are two slices arrange in different column. Slices are not interconnected with each other they are connected to switching matrix to access general routing matrix (GRM). Each slice has independent carry chain logic. One slice contains four 6 inputs LUTs, 4 storage elements, multiplexers and carry chain. The arrangements of Slices are illustrated in figure 4.1.

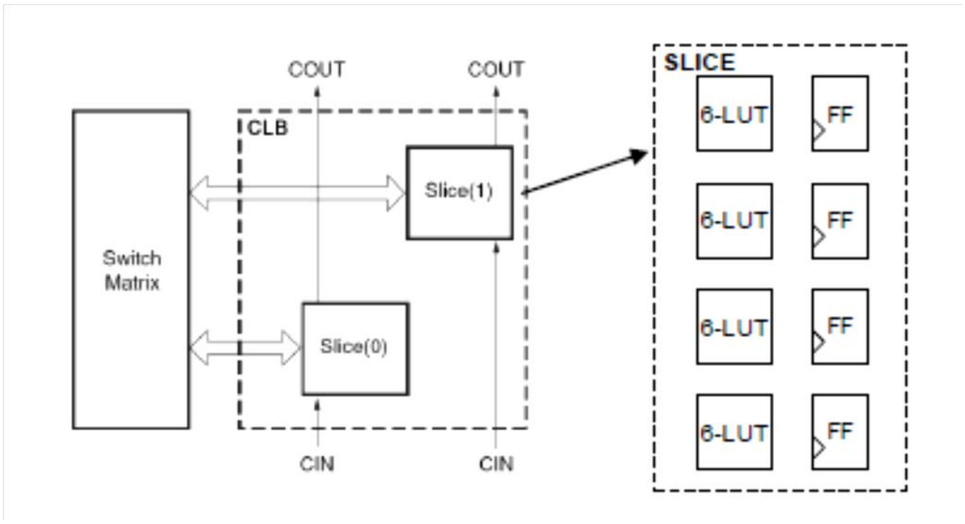

Figure 4.1: Virtex 5 CLB and Slice [14]

Each LUT can implement 6 input Boolean functions. Some slices are called SLICEM used as RAM. SLICEM can be implemented as single port 32 X 1 RAM, Dual port 32 X 1 RAM, quad port 32 X 2 RAM simple dual-port 32 X 6 bit RAM, single-port 64 X 1 bit RAM, dual-port 64 X 1 bit RAM, quad-port 64 X 1 bit RAM, simple dual port 64 X 3 bit RAM, single-port 128 X 1 bit RAM, dual-port 128 X 1 bit RAM and single-port 256 X 1 bit RAM. Some slices are called SLICEL used for implementing logic.

#### 4.5.2.2 Input Output Cell (I/O)

I/O cell contain two IOBs, two ILOGICs, two OLOGICs, and two IODELAYs. ILOGIC include storage element (register). IODELAY introduces delay in incoming signal on individual basis. OLOGIC has two major blocks, one is used to configure output data other is used to configure 3 – state control path. IOB provide interface between package pins and configurable blocks.

#### 4.5.2.3 Block RAM (BRAM)

Block RAM can be configurable as either one 36 kb or two 18 kb RAM. Block RAM can also be configurable as 18 kb or 36 kb FIFO. Each 36 K memory can be configured as 16k X 2, 8k X 4, 4k X 9, 2k X 18 or 1k X 36 memory. The depth of the RAM can be increased by cascading two 36 K RAMs to make one 64 k X 1 RAM.

#### 4.5.2.4 Clock recourses

For clock the Virtex 5 is divided into 8 to 24 regions [14] and also have 32 global clock lines. The clock resources dimension are fixed to 20 CLBs. Clock regions contains 4 clock nets. Clock resources are used to serve localized I/O serialize / de serialize circuits.

#### 4.5.2.5 Digital Clock manager (DCM)

The features of digital clock manager are clock deskew, frequency synthesis and coarse/fine grained phase shifting. Clock deskew feature provide zero propagation delay between source clock and output clock. Frequency synthesis produces frequency in multiple of one frequency. Coarse/fine grained provide output DCM clock to be in phase with input clock.

#### 4.5.2.6 Digital signal processing slice (DSP48E)

DSP slice provides dedicated DSP operation. It is valuable with respect to DSP functionality because it reduces significantly the area occupies on FPGA when it comes to DSP functionality. There are two DSP48E. The DSP slice is based on 25 X 18 bit multiplier. DSP slice provides wider multiplication capability because the slices can be cascaded to form

## 4.6 XILINX ISE

ISE is the design suit provided by XILINX for coding, implementing, simulating and automating the designs for FPGA. It provides many features, synthesizing the design, implementing the design by post and route. It provides ideal platform to implement the design on almost every FPGA of your liking and simulating it in order to see the desire results. In this thesis we use XILINX ISE 12.4 [16].

## 4.7 Summary

In this chapter an overview of field programmable gate array (FPGA) was given and its fundamental elements were discussed. Various advantages of FPGA were highlighted. Virtex 5 FPGA components and architecture were described related to our thesis.

# CHAPTER # 05

# 5 Our Work

## 5.1 Introduction

Cryptographic hash algorithms are widely used for data integrity, password protection and for message authentication. “KECCAK” is one of the SHA – 3 candidates whom we choose to implement over FPGA. KECCAK uses sponge construction for building up the permutation sets. Sponge construction has two phases, absorbing phase and squeezing phase. In absorbing phase the input data is first padded and then XORED with the state bits and passes through permutation and this process repeat until whole block of input data is not entered into the system. Padding means to convert the data into an appropriate block set for instance we know that input bits of KECCAK is 1024 bits of actual data what if the data is smaller than 1024 bits, we have to add additional bits according to bit padding. The next phase of sponge construction is squeezing phase in which the data passed through permutation rounds are output at user will. User will mean that user decides that which output bits are to be selected.

KECCAK consist of 24 rounds, each round consist of 5 steps (theta, chi, rho, pi and iota). Input state of the KECCAK is 1600 bits. In 1600 bits the actual data bits are 1024 and 574 bits are for capacity. These bits are passed through 5 steps 24 times to form a hash output. Order of the steps is not important we can perform any step first. The input bits of KECCAK are arranged in 5 X 5 matrixes whose entries are 64 bits wide [6]. The architecture of KECCAK is shown in figure 5.1

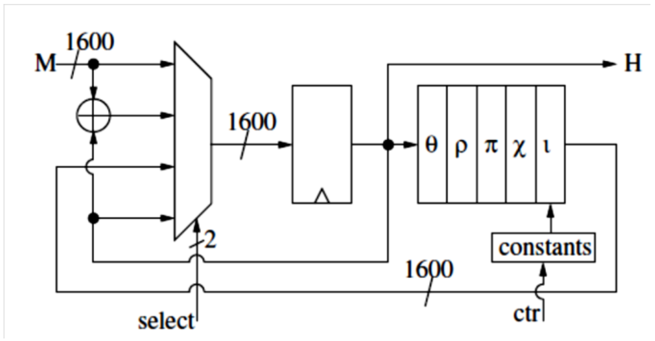

Figure 5.1: KECCAK *f* (1600)

There were many applications of KECCAK on FPGA using different techniques. We got the idea from the paper "Use of embedded FPGA resources in implementations of five round three SHA – 3 candidates" malik umar sharif, rabia shahid, marcin rogawski, kris gaj. In this paper the implementation of KECCAK was done by using embedded FPGA resources such as Digital Signal Processing (DSP) units and Block Memories [17]. For reducing the area further on FPGA we decided to implement the algorithm using separate memory block. The whole idea was that there must be one logic unit which generates address and control signal for retrieving values from memory block.

## 5.2 Initial implementation

At first we implement the KECCAK algorithm by using the raw commands in order to see how much resources it occupies. As we already know that the order of the steps are not important so we start with the theta (Ѳ) step first.

$$R[x] = S[x,0] \oplus S[x,1] \oplus S[x,3] \oplus S[x,4] \qquad 0 < x < 4 \qquad \text{Equation 5.1}$$

$$T[x] = R[x-1] \oplus ROT\,(R[x+1],1) \qquad 0 < x < 4 \qquad \text{Equation 5.2}$$

$$S[x,y] = S[x,y] \oplus T[x] \qquad 0 < x < 4 \qquad \text{Equation 5.3}$$

Theta step is performed by using equation 5.1, 5.2 and 5.3. In equation 5.1, the lanes are XOR with their adjacent lane and the result is stored in register T. We use "for" loop with limit $0 \leq x \leq 4$ and "^' operator for performing XOR operation. In equation 5.2, the results store in array register R is XOR with its array elements. The next array element is first rotated one bit and then XOR with previous array element and the result is stored in array register T. We use bits position for rotating the bits for example R [63:0] was rotated by using (R [62; 0], R [63]) and for XOR operation "^" operator was used. In equation 5.3, the results stored in array register T is XOR with initial input of theta step. We use two "for" loops with limits $0 \leq x, y \leq 4$ and "^" operator for XOR operation. The result of theta step was passed to Rho and Pi step.

We implemented Rho (ƿ) and Pi (π) Step simultaneously by using equation 5.4.

$$U[y, 2x + 3y] = ROT\,(S[x,y], r[x,y]) \qquad \text{Equation 5.4}$$

Rho step is just the rotation of bits in a lane in z direction; the operation is in modulo 64. Pi step is the rotation of the lane according to equation 5.4. The rotation of bits in z direction is given in table 5.1.

Table 5.1: transformation of bits in z direction

| **XY** | **X = 3** | **X = 4** | **X = 0** | **X = 1** | **X = 2** |
|---|---|---|---|---|---|
| **Y = 2** | 25 | 39 | 3 | 10 | 43 |
| **Y = 1** | 55 | 20 | 36 | 44 | 6 |
| **Y = 0** | 28 | 27 | 0 | 1 | 62 |
| **Y = 4** | 56 | 14 | 18 | 2 | 61 |
| **Y = 3** | 21 | 8 | 41 | 45 | 15 |

We implemented by using same bit position technique use in theta step for rotating the bits in z direction and for Pi step we specific lane array for example U[1][3] = ({S[0][1][63-36:0], S[0][1][63:(63-36+1)]}) and so on. The result was passed to Chi step.

The Chi ($\chi$) is stated in equation 5.5

$$\mathrm{S}\,[\mathrm{x},\mathrm{y}] \;=\; \mathrm{S}\,[\mathrm{x},\mathrm{y}] \oplus ((NOT\ \mathrm{S}\,[\mathrm{x}\;+\;1,\mathrm{y}])\ \&\ \mathrm{S}\,[\mathrm{x}\;+\;2,\mathrm{y}]) \qquad \text{Equation 5.5}$$

In Chi step first the next lane is inverted means that “NOT” operation is performed then it is “AND” with its immediate lane and the overall result is XOR with first lane. This process continues for $0 \leq x, y \leq 4$. We use “~” operator for “NOT” operation, “&” operator for “AND” operation and “^” operator for XOR operation. The result from Chi step was passed to Iota step.

The Iota ($\iota$) step is stated in equation 5.6

$$\mathrm{S}\,[0,0] \;=\; \mathrm{S}\,[0,0] \oplus \mathrm{RC} \qquad \text{equation 5.6}$$

In Iota step a 64 bit round constant is XOR with first lane only. There are 24 rounds constant, each for one round. We implement the Iota step by XORING the round constant with the first lane of the output of Chi step.

These five steps were iterated 24 times in order to implement the complete KECCAK $f$ set. We initially don’t use any clock; we just replicate the module 24 times.

## 5.3 Observations

The results obtained from initial implementation were analyzed. The whole KECCAK was occupying 53000 LUTs on FPGA and it was taking 3200 input output blocks (IOBs) which was too much because there is not such IC in FPGA which has 3200 pins. We further note that only one round was taking 2240 LUTs for executing in which only theta step was taking about more than half of the resources.

After looking at initial results we realize that we can perform following improvements in order to make the code perfectly synthesizable.

1. Design a serial interface
2. Clocking the module
3. Use separate memory block for storing intermediate values

### 5.3.1 Serial interface

In order to reduce the IOBs it was needed that input must be taking serially rather than parallel. Parallel input was taking double IOBs, 1600 IOBs for input and 1600 IOBs for output. So the total count was 3200 IOBs. So to reduce the IOBs we design a serial interface which takes data serially on the clock. There was confusion that how much bits the serial interface could take on one clock cycle. We decided that the width of the serial interface should be equal to lane width that is it would take 64 bits of data on one clock cycle. In this way the number of IOBs was minimized to 120 for the data. Figure 5.2 shows the generalize block diagram of serial interface in which we can see that at the clock signal only 64 bits are output.

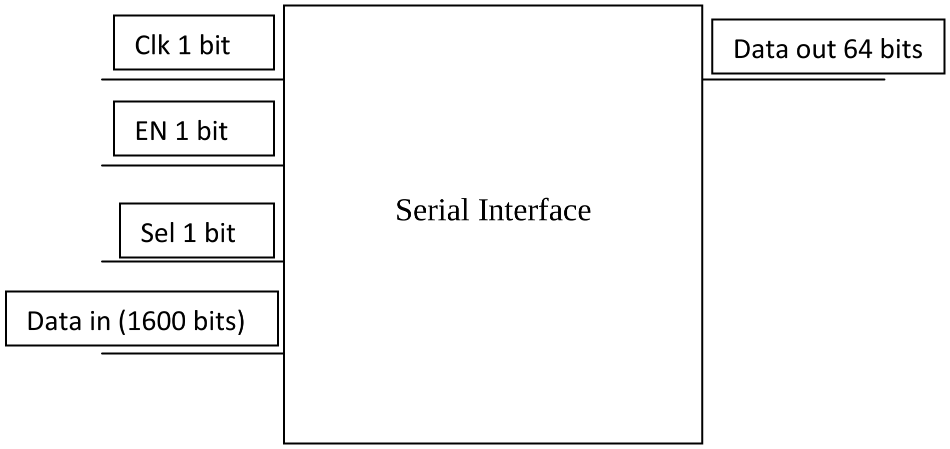

Figure 5.2: Block diagram of serial interface

### 5.3.2 Clocking the module

There was serious need for clock for synchronizing the serial interface with the module. The advantage of clock is that on one clock cycle only one step is performed so it reduces significantly the area occupy by algorithm because we are using only one module space to implement the whole algorithm. We need multiple clock cycles to execute the KECCAK algorithm using the same resources on FPGA before clock when we was just iterating the module so on FPGA actually that module was replicating 24 times it was not good approach when we clock the module, the area on FPGA is reduce because only one module was taken to execute whole KECCAK algorithm only clock cycles increases. Figure 5.3 shows the generalize block diagram of clocking the module.

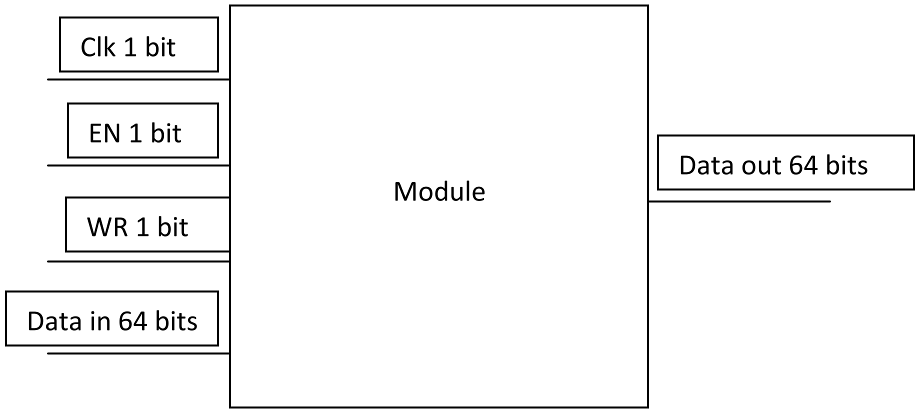

Figure 5.3: Clocking the Module

### 5.3.3 Separate Memory

We come with an idea that we should use separate memory block. In KECCAK algorithm when permutation operation is in progress large space is needed to store the intermediate values for future use. Initially we were storing those intermediate values in internal registers and those registers were occupying too many resources. The idea of implementing separate memory block is illustrated in figure 5.2

## 5.4 Optimized implementation

Our optimized implementation of KECCAK algorithm consists of two modules as illustrated in figure 5.4; one module is purely used for logic implementation and another module is used for storing intermediate value it just acts like a simple memory block. We can characterized our design as follows

1. Logic Module (KECCAK)
    a. Module for generating address and control signal
    b. Module for data.
2. Memory Module (Memory block)

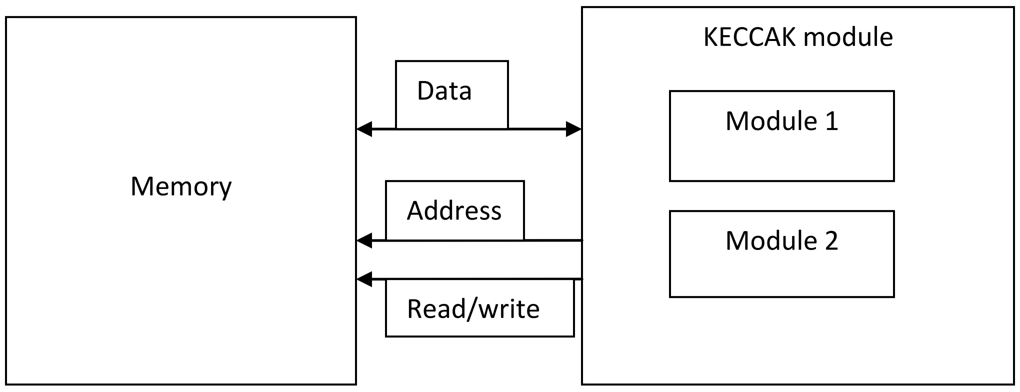

Figure 5.4: Our implementation of KECCAK

### 5.4.1 Logic Module (KECCAK)

The logic module performs the operation for KECCAK algorithm. In this module the equation of KECCAK is implemented. Well from initial implementation results we concluded that the theta step was occupying too many resources alone and the rest of the steps were occupying much less resources than theta step. It is mainly due to the XOR operation present in equation 5.1 occupies too many resources. We implement equation 5.1 using addition operation in modulo 2 fields. We add lanes in modulo 2 field rather XORING with each other and the result was stored in memory block. Than the remaining theta operation are performed. After the theta step the output lanes are rotated according to Rho step. For rotation of lane we use same technique as it was discussed in initial implementation. Then Pi step is performed according to table 5.1. The Pi step is basically the rotation of bits in z direction in modulo 64. We implement Rho and Pi step simultaneously, the lanes which was rotated according to Rho step is used for Pi step. The 64 bits present in lane is rotated in Pi step. So Rho and Pi step was performed simultaneously. Then the Chi step was performed. For Chi step we used three internal registers. The lane is read from the memory and stored in these there registers. As we know that in Chi step first the second lane is “NOT” and then “AND” by third lane and the result is XOR with first lane. The same technique is used and the result is stored in memory block. In last the Iota Step is performed. In Iota step just a 64 bit round constant is XOR with first lane of the state. There are 24 different rounds constant each for a single round.

The whole above implementation is done by using two modules. One module is purely dedicated for generating address and control signal to memory block and other module is used as data path for performing operations.

#### 5.4.1.1 Module for address and control signal

This module is used to store and write the data. First the input data is stored in dedicates part of memory in form of Lanes and address is given to each lane. When start bit is enabled, module begins generating address and control signal in a sequential manner. Generation of address and control signal are implemented using a switch case statement.

Some brief description about the working of module is that it starts with the reset register. If reset register goes to 0 all values store in register goes to 0 and module passes to initial state, in initial state all register are re initialize to their default values which is 0. If start register goes on 1 then the working starts enread register is on 1 which means that module is reading the data from memory at first memory location 0 is read and command 0 is generated than memory location 1 is read and command 1 is generated then memory location 2 is read and command 10 is generated then memory location 3 is read and command 10 is generated then memory address 4 is read and command 10 is generated at this point the module passes through if else statement. Basically what we have done above is that we read a complete column and passes to data module. In if else statement is counter_sheet is not equal to 4 then whole process repeats until its value becomes 4 then module pass to next stage. Then memory address 54 is read and command 1 is generated then memory address 4 is read and command 1001 is generated then only command 1100 is generated then memory address 14 is read and command 100 is generated. Then module progress to if else statement. If counter plane == 4 and counter sheet == 4 the module passes to next stage in which memory address 25 is read and command 0 is passed and counter plane and counter sheet register is re-initialize to 0 value. Then memory address 27 and command 1001 is passed. Then the only command 1001 is passed. Then the value is written to memory on address 0 and module progress to next stage. In next stage number of round register is used in if else statement. If it is less than 23 whole than process is repeated again and if it is equal to 23 then in done register 1 value is passed indicating that the process of hash function is complete.

Basically what we are doing in this module is that we are generating address for memory to pass a single bit in data module and along with a particular command. In this way whole ϴ values and χ values are mapped in the three registered in data module. These combinations of address are chosen according to sheets and plane.

### 5.4.1.2 Module for data

This module performs calculation according to KECCAK equations. This module is used for calculating data. The data extracted from the memory according to control signal generated from memory is used in this module for operation. In this module Chi, Rho, Pi and Iota operations are performed on data.

A briefly working about this module is that the commands generated from first module are used here for performing calculations. For instance if command first bit is 1 then the data from memory is directly pass to register r1 otherwise if command 2nd bit is 1 then data from memory is XORED with r2 and then pass to register r1. If command 3 bit is at 1 then data from memory is XORED with register r3 and the overall result is XORED with one bit cyclic shift of register r2 and result is assigned to r1 if none of the condition is valid the register r1 is assigned the same value. For register r2, command 3rd bit is looked if it is 1 then value of register r1 is assigned to register r2 otherwise same value is retained. Same goes for register r3 also if command's 3rd bit is on high position value from register r2 is assigned to r3 otherwise same value is retained. Then the χ step is performed. Then according to values of counter_plane and counter_sheet the π step is perfomed. According to nr_round value a round constant is pass to iota register. Then commad's 4th bit is observed if it is 1 then register r1 value is written into the memory. If command 5th bit is 1 then value from rho register is written to the memory. Is command 6th bit is 1 then value from register chi_out is written to the memory and if command 7th bit is 1 then register chi_out is XORED with iota and written to the memory.

Basically the data module operated only on command generated from the first module.

### 5.4.2 Memory Module (Memory Block)

The memory module is used for storing intermediate values. Each memory locations is assigned a particular address for storing and retrieving the data. It is basically 64 X 64 RAM. The memory module is not integrated with logic module it is used separately only for storing the intermediate values. Just like our personal computer that it has hard disk for storing data and processor and other things for computing the data. The same approach was taken over hear also.

## 5.5 Advantages

The advantage of placing memory block separate is that other resources can also share the memory when it is not in used by the KECCAK. Suppose in an IC there are many hash functions that requires memory for saving their values so this memory can be share with them and hence over all resources can be optimized because only one memory block is shared by several algorithms. In figure 5.5 the concept of placing memory block separate is illustrated, that many algorithms are sharing the memory block so when we look at over all IC there is significant resources optimize tales place. Instead of having separate memory for every algorithm we just share one memory block to every algorithm.

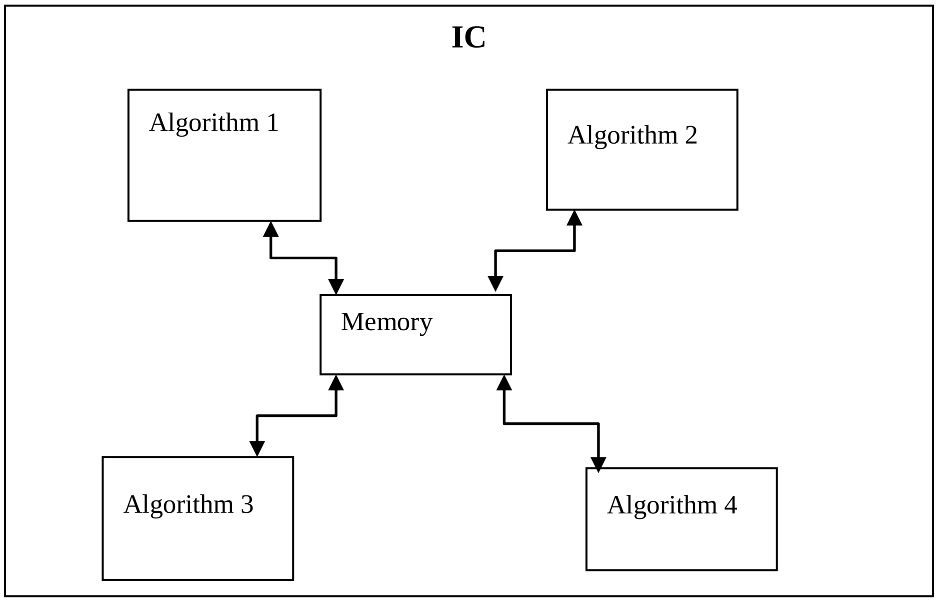

Figure 5.5: Memory sharing by several algorithms

The memory sharing technique is quite efficient is reducing the area occupied by algorithm on FPGA or an IC. Suppose in an IC there are multiple algorithm which requires memory for saving the values, when we look at conventional implementation and compare with our implementation so in conventional implementation every algorithm need its memory for storing their values hence the area occupied on an IC increases. While in our implementation we use same memory as shared memory for all algorithms in an IC which results in area reduction.

# CHAPTER # 06

# 6 Results

## 6.1 Implementation of Results.

The KECCAK algorithm is implemented using Verilog HDL language using XILINX ISE 12.4. We first coded the KECCAK algorithm and then use XILINX XST tool for synthesizing the code. ISE Simulator is use for verifying our design by giving test vectors.

Table 6.1: Implementation results of Logic module

| | **Target Device** | Virtex 5 - 5vlx30ff324-2 |
|---|---|---|
| **S.No** | | |
| a. | Number of slices | 432 |
| b. | Number of LUTs | 1209 |
| c. | Maximum Clock frequency | 264.236 MHz |

Table 6.2: implementation results of memory block

| | **Target Device** | Virtex 5 - 5vlx30ff324-2 |
|---|---|---|
| **S.No** | | |
| a. | Number of slices | 1641 |
| b. | Number of LUTs | 5505 |
| c. | Maximum Clock frequency | 942.774 MHz |

Table 6.1 shows the implementation result of logic block and table 6.2 shows the implementation results of memory block. In these two tables device utilization, input output utilization and timing summary is discussed.

## 6.2 Comparison with papers

We compare out implementation result with the following papers as illustrated in table 6.3. In our implementation only logic block is occupying 432 slices. Basically we implemented KECCAK on a concept of memory sharing. If there are multiple algorithms on an IC which required memory for storing their intermediate value so our memory block can be shared. In our implementation the whole calculation of KECCAK is performed in logic block, memory block is just used for storing intermediate value during the calculation.

Table 6.3: Comparison with papers

| **Comparison with papers** | **Resource utilization (Slices)** | | **Clock frequency (MHz)** | |
|---|---|---|---|---|
| | **Logic** | **Memory** | **Logic** | **Memory** |
| Our results | 432 | 1641 | 264 | 942 |
| [10] Malik Umar 2011 el.al. | 1338 | 1 36k BRAMs | 242 | |
| [20] Brian Baldwin 2010 el.al. | 1971 | | 195 | |
| [24] Kobayashi, K 2010 el.al. | 1433 | | 205 | |

# CHAPTER # 07

# 7 Conclusion and Future Work

## 7.1 Conclusion

We have successfully implemented KECCAK algorithm using different approach in order to minimize the area occupied on FPGA. The optimization was done by separating the memory block because it was occupying too many resources. Our implementation consists of two modules; logic module and memory module. The main module is logic module in which it is further divided into two sub modules, one for generating address and control signals and another used as for calculating data.  Memory module is only used to store intermediate values. The advantage of this approach is that we can share memory module with other algorithms also so in broader sense overall area of IC on which other all algorithms are implemented is optimized.

## 7.2 Future Work

The future work includes sharing the memory block which we separate from our logic module with other algorithms. It can be share with other algorithms in order to save their intermediate values. Basically our memory block is 64 X 64 bit RAM and it can be used as shared memory with any algorithms which requires that much space to store its data or values.

# Bibliography

[1] FIPS PUB 180-4. “Secure Hashing”, available at http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, March, 2012.

[2] FIPS PUB 180-3. “Secure Hash Standards”available at http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf, October, 2008

[3] X. Wang, Y. L. Yin and H. Yu. “Finding collisions in the full SHA-1”, In Proceedings of Crypto 2005-08, pages 17 – 36, Volume 3621. Publication year 2005.

[4] Regenscheid, A., Perlner, R., jen Chang, S., Kelsey, J., Nandi, M., Paul., S.: Status report on the first round of the SHA-3 cryptographic hash algorithm competition. Technical Report 7620, NIST (September 2009), http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/documents/sha3_NISTIR7620.pdf

[5] X. Guo , M. Srivastav , S. Huang , L. Nazhandali and P. Schaumont “Silicon Implementation of SHA-3 Final Round Candidates: BLAKE, Gr_stl, JH, KECCAK and Skein”. In ECRYPT II Hash Workshop 2011, May 2011.

[6] A. Akin, A. Aysu, O. C. Ulusel, and E. Savaş “Efficient Hardware Implementation Of High Throughput SHA 3 Candidates KECCAK, Luffa And Blue Midnight Wish For Single And Multi Hashing”. 3rd International Conference on Security Of Information And Networks, 2010; PP 168 – 177. ISBN: 978-1-4503-0234-0

[7] G. Bertani, J. Daemen, M. Peeters and G. V. Assche. “Cryptographic sponge functions”. January 14, 2011, http://sponge.noekeon.org/CSF-0.1.pdf

[8] Federal Information Processing Standards Publication 180-2. “Secure Hash Standards” available at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf, 2002 August 1.

[9] KECCAK discussion Soham Sadhu. January 9, 2012. Available at http://www.cs.rit.edu/~hpb/Lectures/20112/S_T/Src/32/keccak.pdf 31

[10] S. Brown and J. Rose. “Architecture of FPGAs and CPLDs: A Tutorial”. Published in: IEEE Journal Design & Test archive, pages 42 – 57, Volume 13 Issue 2, June 1996.

[11] K. Parnell, R. Bryner. “Comparing and Contrasting FPGA and Microprocessor System Design and Development” XILINX, WP213 (v1.1) July 21, 2004.

[12] “Virtex-5 Platform FPGA Family” Technical Backgrounder. Xilinx. Available at www.**xilinx**.com/company/press/kits/v5/v5**backgrounder**.pdf

[13] Andrew Percey. “Advantages of the Virtex-5 FPGA 6-Input LUT Architecture” Xilinx WP284 (v1.0) December 19, 2007

[14] Taneem Ahmed, Paul D. Kundarewich, Jason H. Anderson Brad L. Taylor and Rajat Aggarwal. “Architecture-Specific Packing for Virtex-5 FPGAs” *FPGA’08, 16th* international ACM/SIGDA Symposium On Field Programmable Gate Arrays, pages 5 – 13. February 24-26, 2008, ACM 978-1-59593-934-0/08/02

[15] “KECCAK implementation” overview. Version 3.1, September 5th, 2011.

[16] Tutorial on FPGA Design Flow based on Xilinx ISE Webpack and ModelSim. ver. 1.6

[17] M. U. Sharif, R. Shahid, M. Rogawski, K. Gaj “Use Of Embedded FPGA Resources In Implementations Of Five Round Three SHA – 3 Candidates”. International Conference 2011 on Field-Programmable Technology (FPT), PP 1 – 9. Published on 12 – 14, Dec 2011. ISBN: 978-1-4577-1741-3

[18] L. Angeles and Y. Ding. “Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays”. ACM Transactions on Design Automation of Electronic Systems (TODAES), pages 145 – 204, Volume 1 Issue 2, April 1996. ISSN: 1084-4309 EISSN: 1557-7309

[19] H. P. Afshar1, A. Neogy3, P. Brisk2 and P. I. Ecole Polytechnique Fédérale de Lausanne (EPFL), “Improved Synthesis Of Compressor Trees On Fpgas By A Hybrid And Systematic Design Approach” In Proceedings of the International Workshop on Logic and Synthesis (IWLS'10). IEEE, 193--200. 32

[20] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O‟Neill and W. P. Marnane. "FPGA Implementations of the Round Two SHA-3 Candidates". International Conference 2010 on Field Programmable Logic and Applications (FPL), PP 400 – 407, Issue Date: Aug. 31 2010-Sept. 2 2010. ISBN: 978-1-4244-7842-2

[21] Xu Guo, Meeta Srivastav, Sinan Huang, Dinesh Ganta, Michael B. Henry, Leyla Nazhandali and Patrick . "Pre-silicon Characterization of NIST SHA-3 Final Round Candidates". 14th Euromicro Conference 2011 on Digital System Design (DSD), PP 535 – 542, Issue Date: Aug. 31 2011-Sept. 2 2011. ISBN: 978-1-4577-1048-3

[22] Clark N. Taylor ."FPGA Implementation Details". October 22, 2008, available at http://ece320web.groups.et.byu.net/CourseNotes/FPGA.pdf.

[23] K. Latif, A. Aziz, and A. Mehboob. "Efficient Resource utilization of FPGAs". FIT '09 Proceedings of the 7th International Conference on Frontiers of Information Technology, Article No. 26, Published in 2009. ISBN: 978-1-60558-642-7

[24] Kobayashi, K.; Ikegami, J.; Knezevic, M.; Guo, E.X.; Matsuo, S.; Sinan Huang; Nazhandali, L.; Kocabas, U.; Junfeng Fan; Satoh, A.; Verbauwhede, I.; Sakiyama, K.; and Ohta, K.; "Prototyping Platform for Performance Evaluation of SHA-3 Candidates". Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium, pages 60 – 63, Issue Date: 13-14 June 2010. ISBN: 978-1-4244-7811-8

[25] T Ahmed, P D. Kundarewich, and J. H. Anderson Xilinx, Inc. "Packing Techniques for Virtex-5 FPGAs". ACM Transactions on Reconfigurable Technology and Systems, Article No. 18, Volume 2 Issue 3, September 2009. ISSN: 1936-7406 EISSN: 1936-7414

[26] P. Morawiecki and M. Srebrny. "A SAT-Based Pre-Image Analysis Of Reduced KECCAK Hash Functions". Proceedings of IACR Cryptology e-Print Archive. 2010, 285-285. August 6, 2010.

[27] "Cryptography and network security" 4th edition by William stalling. Published in 2005