LIGHTWEIGHT IMPLEMENTATIONS OF THE SHA-3
FINALIST KECCAK ON FPGAS

by

Smriti Gurung
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____	Dr. Jens-Peter Kaps, Thesis Director
_____	Dr. Kris Gaj, Committee Member
_____	Dr. Bernd-Peter Paris, Committee Member
_____	Dr. Andre Manitius, Department Chair of Electrical and Computer Engineering
_____	Dr. Lloyd J.Griffiths, Dean, Volgenau School of Engineering
Date: May 4, 2012	Spring Semester 2012 George Mason University Fairfax, VA

Lightweight Implementations of the SHA-3 Finalist Keccak on FPGAs

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Smriti Gurung
Bachelor of Science
Delhi College of Engineering, 2008

Director: Jens-Peter Kaps, Professor
Department of Electrical and Computer Engineering

Spring Semester 2012
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents Gen. Umesh Kumar Gurung and Mrs. Uma Gurung for their undying love and support towards my work.

Acknowledgments

I would like to thank my advisor Dr Jens-Peter Kaps for his utmost faith towards my academic abilities. Without his friendly advice and encouragement this work would not have been complete. Especially his ability to decrypt my somewhat incomprehensible queries at times which helped a lot in our smooth communication. I am also thankful to my CERG team members for their suggestions and ideas during and after our group meetings. In particular I would like to thank Rajesh, Panasaya, Kishore and Susheel for their unconditional moral support and presence, even during ungodly hours of the day.

Table of Contents

List of Tables

List of Figures

Abstract

LIGHTWEIGHT IMPLEMENTATIONS OF THE SHA-3
FINALIST KECCAK ON FPGAS

Smriti Gurung, MS

George Mason University, 2012

Thesis Director: Dr. Jens-Peter Kaps

The Secure Hash Algorithm (SHA) is a cryptographic hash function published by the National Institute of Standard and Technology (NIST) as a U.S Federal Information and Processing Standard (FIPS). In the past few years, a flaw discovered in the SHA-1 shows its vulnerability to attacks. The current hashing standard SHA-2 which shares similarities to SHA-1 is therefore under scrutiny for a possible attack. In 2007, NIST announced the SHA-3 competition in hopes of finding a new algorithm with higher margin of security and which is also more efficient in terms of software and hardware performance. Out of the 51 candidates selected in round one, only five remain in the third and final round namely BLAKE, Grostl, JH, Keccak and Skein.

So far, several high speed implementations of the SHA-3 algorithms on FPGAs have been published. However, these implementations become impractical for resource constrained environments where area is a limitation for e.g small battery powered hand held devices. Our goal was to design different lightweight architectures for the sponge construction based algorithm Keccak. We tried to evaluate its performance with respect to its scalability. In this study all the implementations were designed with an area constraint of 800 slices or 400-600 slices and one block RAM, targeting the low cost Spartan-3 devices. Designs were also synthesized on different Xilinx and Altera devices for comparison with other published results. Although our implementation of Keccak is one of the smallest reported so far, this reduction came at the cost of lower throughput to area ratio.

Chapter 1: Introduction

1.1 Hash

The introduction of computers has brought about the need for automated tools to protect the files and information stored on it. Such a need is even more acute for systems that can be accessed over telephone network, data network, or the Internet. Secure communication for transmitting data free from attacks by unauthorizes users is of utmost concern to us. Thus in order to prevent and detect cheating or other malicious activities we employ cryptography, a set of means or techniques for providing information security. The framework of cryptography is based on four important goals

- Confidentiality : Secrecy from unauthorized individuals.
- Data integrity : Protection from unauthorized alteration of data.
- Authentication: Related to the identification of both the parties involved in the communication.
- Non-Repudiation : A service which prevents an entity from denying previous commitments or actions during a dispute.

There are three types of cryptographic algorithms: Message Encryption, Message Authentication Code (MAC) and Hash functions.

To provide data integrity, any message could be simply encrypted. But the heavy mathematical functions involved utilize bulk of the CPU resources thus incurring large overheads. To reduce this load, certain applications use a lightweight procedure called a one-way hash or simply hash functions. Unlike Message Encryption and MAC, hash functions have no key. A hash function accepts a variable-size message M as input and produces a

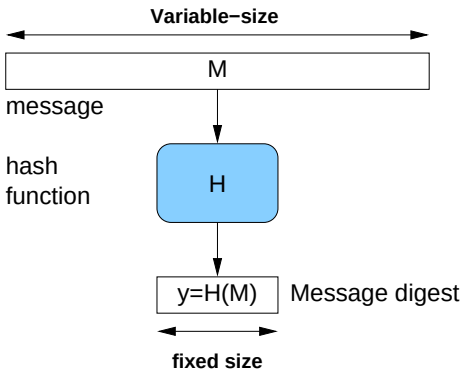


Figure 1.1: Hash Digest

fixed-size output, referred to as a hash code H(M). The hash code is also referred to as a *Message Digest* or *Hash Value*.

Typical application of Hash function includes Universal Unique Identifiers (UUID/GUID), Password tables, Random Number Generators and many more. But they finds their major application in Digital Signatures. As shown in Fig. 1.2 the messages being sent from Alice typically include both the plain text (unencrypted) and a digest of the message(encrypted). The hash algorithm is applied to the received plain text at Bob‘s end and if the result matches the message digest received, this means the received data was not altered. The message digest is, in some senses, similar in concept to a checksum but has significantly different mathematical properties.

To sum it up the basic requirements for a cryptographic hash function H are as follows

- H should accept data of any size as input.
- H should produce a fixed-length output no matter what the length of the input data is.
- Given a message digest h, it is computationally difficult to find a message M such that $H(M) = y$. This is called the one-way or preimage resistance property.

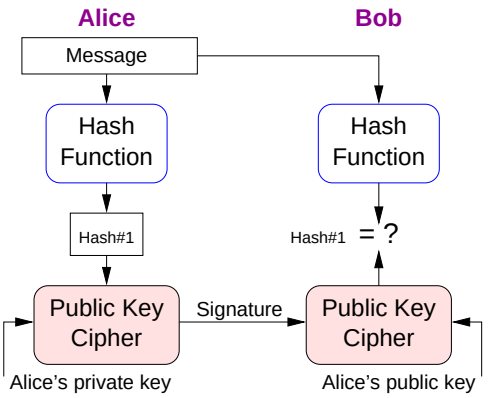


Figure 1.2: Hash Application in Digital Signature

- Given a message M1, it is computationally infeasible to find another message M2 with $H(M1) = H(M2)$. This is called the second preimage resistance property.
- These can be compared with a collision attack, which involves finding two arbitrarily different messages M1 and M2 such that $H(M1) = H(M2)$.

A good cryptographic function is always collision resistant such that a change as small as a single digit in the input message should produce a large change in the hash value of the message. The hash digest then produced is like a digital fingerprint of a message M that is unique. These properties are therefore required in order to prevent or withstand certain types of attacks which may render a cryptographic hash function useless and insecure.

1.2 Secure Hash Algorithms

The Secure Hash Standard (SHS) is a set of cryptographically secure hash algorithms specified by the National Institute of Standards and Technology (NIST)[1]. This Standard specifies five secure hash algorithms (SHA) namely SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The SHA is a cryptographic one-way hash functions that can process a message to produce a condensed representation called a Message Digest. This is used in

Table 1.1: Collision Resistance of SHA

HASH	COLISSIONS	SECOND PREIMAGE	PREIMAGE
SHA−0 (80 rounds)	2^{39}	upto 52 rounds	upto 52 rounds
SHA−1 (80 rounds)	$2^{60.3}$	upto 48 rounds	upto 48 rounds
SHA−256 (64 rounds)	upto 24 rounds	upto 43 rounds	upto 43 rounds
SHA−512 (80 rounds)	upto 24 rounds	upto 46 rounds	upto 46 rounds

the generation and verification of Digital Signatures, Message Authentication Codes and in the generation of Random Numbers or bits. The Table 1.1 gives an overview of the security, namely the collision and the preimage resistance provided by the current Federal Information Processing Standard (FIPS).

In 2005 the NIST approved SHA-1 standard showed potential vulnerability towards serious attacks. One such proposed attack was published in [2]. This casted a shadow of doubt towards the current FIPS standard SHA-2, due to its similarities to SHA-1. In November 2007, after receiving public feedback and comments, the NIST officially announced the SHA-3 competition in hopes of finding a new hashing standard with better security margin and hardware-software performance than the current SHA-2. Out of the 64 candidates submitted on October 2008 only 51 entered the first round. By the second round this was downsized to 14 candidates. During this period NIST allowed the authors to make small tweaks in their respective algorithms. These tweaks were such that they did not invalidate previous results. The candidates for the third and final round were announced in December 2012. They are BLAKE, Grostl, JH, Keccak and Skein.

The Third SHA-3 conference was held in Washington DC on Mar 22-23, 2012. This was the last opportunity for many cryptographers and cryptanalyst to submit their findings on the studies of these SHA-3 algorithms. Based on the valuable feedback from the public and the judgment by the NIST panel, it is expected that the NIST would announce the winner of the new SHA-3 standard before the Fall of 2012.

1.3 Previous Work and Motivation

The main selection criteria for a good cryptographic hash function after security is performance in software and hardware. For hardware implementations the two major platforms are Application Specific Integrated Circui t(ASICs) and Field Programmable Gate Arrays (FPGAs).

When considering hardware implementations, apart from speed the scalability of the algorithm also plays an important role. Reconfigurability at run time, design re-usability, low non recurring engineering costs and faster time to market make the FPGA preferable at certain times. With the development of low power and low cost FPGAs, implementing applications that use cryptographic hash functions (along with some other component) especially for resource constrained environments like battery powered hand held devices, on FPGAs thus seems a logical choice.

Most of the implementations that were reported so far [3], [4], [5] were on based on high speed implementations on FPGAs. To be an all round candidate, it is thus important that the SHA-3 algorithms perform well on resource constrained environments. The goal is to implement them as small as possible, but it is also important to take the throughput into consideration while designing. With respect to the basic architectures of the algorithms that can be built in a straightforward fashion, the reduction in the datapath for compact implementations may sometimes lead to unrealistic clock cycles or complex controller logic. This is due to interdependencies among intermediate values in forthcoming functions. This leads to unrealistic processing time and low throughput. Thus the best compact implementation would be the one that offers optimum trade off between speed and area.

One of the finalists for SHA-3 is the sponge based construction, Keccak, that is claimed to have provable security against all generic attacks. It is one of the best algorithms while considering high speed implementations on FPGAs [3],[4]. Only two low area implementations [6], [7] have been reported for the so far. Still rating the finalists is an issue as they are implemented on different target devices as well as vary in wide range for area and throughput. In order to have a fair comparison among finalists we thus decided to design

Keccak in a given area budget with maximum throughput to area ratio. We also decided to implement it on other devices of Xilinx and Altera for comparison with other reported results.

1.4 Assumptions and Goals

Only 256 digest version of the algorithm was implemented. Since implementing for low area alone may lead to unrealistic run times therefore our primary optimization goal at the end was to get the best throughput to area architecture within the given area target. Compact implementations requires the storage of initial and intermediate states as well as constants. For this purpose we decided to design architectures with one Block RAM and 400-600 slices or with only 800 slices targeting the low cost Spartan-3 device. We firstly compared the implementations of the other SHA-3 round 2 candidates by the Cryptographic Engineering Research Group (CERG) with similar constraints. The area budget was then determined by the range of area over which they varied as well as the results reported on the SHA3 Zoo [8] at that time.

1.5 Thesis Organization

Chapter 2 acts as a guide to the SHA-3 competition so far. Chapter 3 focuses on the protocols and the design methodology adopted for our implementations. In the subsequent chapters we give a short introduction to the algorithm Keccak, and then move on to the different architectures implemented on various Xilinx and Altera devices. In the final chapter we look at the results of our implementations and conclude.

Chapter 2: Secure Hash Algorithms

2.1 Background

In 2005, a paper on collision search attack on SHA-1 [2] was published. This paper claimed that collisions of SHA-1 could be found with complexity $< 2^{69}$ hash operations. This was a joint effort by a team of two researchers Xiao Yunwang and Hong Boyu of Shandong University, China, along with Yiqun Lisa Yin an independent security consultant in the United States. Although the current standard is SHA-2, yet its similarity to SHA-1 poses a possible threat to its security also.

Due to these serious attacks, NIST held workshops to assess the status of its approved hash function . As a result of the response they received, the NIST has decided to develop a new HASH function through public competition similar to Advanced Encryption Standard (AES). In January 2007, NIST publicly announced the SHA-3 competition and asked for public comments. By October 2008, 64 submissions were received. Adhering to the minimum acceptability requirements only 51 made it to the round 1.

The first SHA-3 conference was held in Feburary 2009. At that point in time NIST discarded 27 out of the 51 submissions that had been considered broken by then. MD6 designed by Ron Rivest was also withdrawn during this time. The reason they cited was the inability to provide a substantial proof for the security of the reduced round MD6 against differential attacks. At the point when the candidates were down to 23, the NIST allowed them to make some suitable tweaks to the algorithms that would not substantially affect their previous analysis. Finally in July 2009 only 14 candidates entered the round 2.

Table 2.1: SHA-3 Round 1 Candidate

Abacus	ARIRANG	AURORA	Blake	Blender
BMW	Boole	Cheetah	CHI	CRUNCH
CubeHash	DCH	Dynamic SHA	Dynamic SHA2	ECHO
ECOH	EDON−R	Enrupt	ESSENCE	FSB
FUGUE	Grostl	HAMSI	JH	Keccak
Khichdi 1	Lane	Luffa	LUX	MCSSHA−3
MD6	MeshHash	NaSHA	NKS2D	SANDstorm
Sarmal	Sgail	Shabal	SHAMATA	SIMD
SKein	SHAvite−3	Spectral Hash	StreamHash	Swifftx
TANGLE	TIB3	Twister	Vortex	WaMM
Waterfall				

Table 2.2: SHA-3 Round 2 Candidate

Blake	BMW	CubeHash	ECHO	FUGUE
Grostl	HAMSI	JH	Keccak	Luffa
Shabal	SHAvite−3	SIMD	Skein	

2.2 Round 2

The Second SHA-3 conference was held in August 2010. The Table 2.2 shows the list of the 14 round 2 candidates.

Many papers and presentations focused on different criteria like cryptanalysis, design diversity and hardware and software performances. Despite the input from various sources, it was still difficult to pin point a clear winner in the overall categories. Up till now many implementations in hardware have also been done by multiple research groups and individuals on ASICs and FPGAs. A Comprehensive list and links of these implementations can be found at [8]. At the end of 2010, the five finalists were announced as Blake, Grostl, JH, Keccak and Skein. From then until the period leading to Third SHA-3 conference in

March 2012 the remaining finalist were attacked, analyzed and implemented on various hardware and software platforms. The analysis and findings by many groups were thus presented at the conference in hopes of contributing to NISTs criteria in picking out a winner. Currently the NIST is deliberating over the finalists based on their evaluation criteria and analyzing results from various public sources available. In the next few months the new SHA-3 will finally be chosen. The only possible best guess about the winner is that the new SHA-3 will be the one who offers optimal security, lesser performance requirements and faster than the current SHA standard.

Chapter 3: Methodology

3.1 Tool selection and Benchmarking

All the initial designs targeted the Spartan-3 FPGA. This meant that we had to explore the architecture of Spartan-3 device, in order to get the best possible our low area design of our implementation of the algorithm. Looking at the implementations by other groups [6], [7] on various Xilinx devices, we also implemented our designs further on Virtex-5, Virtex-6, Spartan-6 and Cyclone II devices. This was based on our interest to see how our designs would perform on other families of FPGAs. All designs were implemented using the vendor tools: Xilinx ISE 13.3 Web Pack and Altera Quartus II v. 10.0Web Edition. All results were generated using the open source benchmarking tool ATHENa (Automated Tool for Hardware EvaluatioN) [9].

3.2 Interface and Protocol

3.2.1 Interface

The I/O interface and the protocol for our implementation is based on the one suggested in [10]. In a typical scenario we assume that our SHA-3 core is surrounded by two FIFOs at the input and the out put. The SHA core is the active component that acts as a wrapper around our implementation while the FIFO is a w bit wide register that is used to hold intermediate data between two cores during processing. With the help of some handshaking protocols the FIFO acts as a source of synchronization between two interfaces working at different frequency or rates. For our version of the lightweight implementation we will be considering the $w = 16$ bit data bus width.

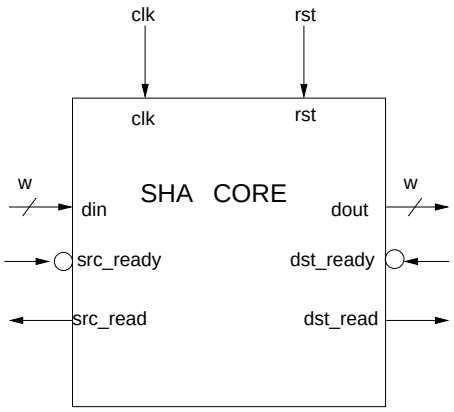


Figure 3.1: SHA Core

3.2.2 Protocol

The protocol or the format of the input data is based on two scenarios

1. When the message length is known in advance

   The message is transmitted as a single chunk or segment, having the message length after padding "msg_len_ap" as the first 32-bit word, concatenated with a '1', followed by message length before padding "msg_len_bp" in bits followed by the message. For some algorithms "msg_len_bp" is required for computation even after padding the message.

2. When the message length is not known

   In this case the message is processed in segments $seg_0, \cdots, seg_{n-1}$. The segments $seg_0, \cdots, seg_{n-2}$ are headed by "seg_len_ap" concatenated with a '0', this means that there are more segments of the message coming in. The final segment seg_{n-1} has "seg_len_ap" concatenated with '1' and followed by "seg_len_bp" followed by the message and all the padding bits. The padding for the time being is considered to be done in software for all purposes.

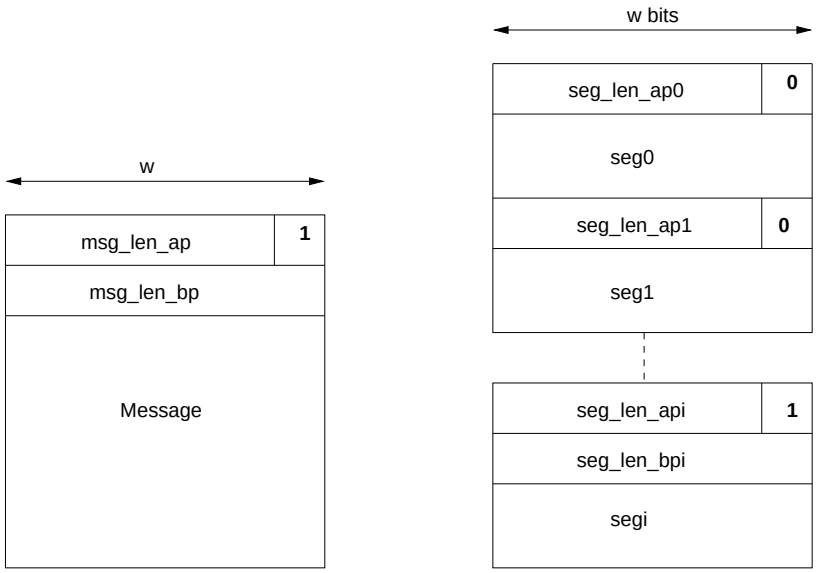


Figure 3.2: a) Message Length Known b) Message Length Unkown

$$msg\_len\_ap = \sum_{i=0}^{n-1} seg\_len\_ap_i \cdot 32$$

$$msg\_len\_bp = \sum_{i=0}^{n-2} seg\_len\_ap_i \cdot 32 + seg\_len\_bp_{n-1}$$

The maximum amount of data that can be processed in a single block is therefore limited to 2^{32} bits or 4 Giga bytes which we assume is sufficient for such small implementations.

3.3 Performance Metrics

When considering the performance metrics it is very difficult to determine what would be the best possible means to judge the performance by. When talking about hardware performance it could be cycles per block, cycles per byte, Latency (cycles), Latency (ns),

Throughput for long messages, Throughput for short messages, Throughput at 100 KHz, Clock Frequency, Clock Period, Critical Path Delay etc. The favoured parameter in general however is Throughput (Mbits per sec) and Latency (ns). This is because they allow easy cross-comparison among implementations in software (microprocessors), FPGAs (various vendors) and ASICs (various libraries).

One of the key factors on which the performance depends on is the number of clock cycles it takes to hash N blocks of message. The number of clock cycles needed to hash N-blocks of message is given by the formula :

$$No.of clk\ cycles = i + h + l1 + l \cdot (N - 1) + p \cdot N + z + o$$

i	Initialization (if not precomputed)	p	Processing one block
h	Loading protocol header of message	z	Finalization
$l1$	Loading first block	o	Output of the hash value
l	Loading each subsequent block		

Each of these specify the number of clock cycles required to perform that particular function.

This formula can be simplified as

- Initial steps before processing : $st = i + h + (l1 - l)$
- Loading and processing of one message block : $(l + p)$T
- The hashing out and finalization : $end = z + o$

Thus the number of clock cycles to hash N-blocks of data:

$$No.of clk\ cycles = st + (l + p) \cdot N + end$$

Another performance metrics that is of concern to us is the throughput. It is defined as the number of input bits processed per unit time. The throughput of the hash function is

dependent on the number of message blocks N that is to be hashed, the number of clock cycles to process one message block, block size b of the algorithm and the delay per clock period T. Therefore throughput of a hash function can be given as:

$$throughput(N) = \frac{b \cdot N}{clk \cdot T} = \frac{b \cdot N}{(st + (l + p) \cdot N + end) \cdot T}$$

For short messages we assume that there is only one block of message after padding. This makes the value of $N = 1$ in the above equation for computing the throughput. For long messages, the *st* and *end* are only considered once at the begining and the finalization of the message. So it makes a negligible effect on throughput. This leads us to the simplified equation for throughput of long messages, which is

$$throughput_{long} = \frac{b}{(l + p) \cdot T}$$

3.4 Spartan-3 Architecture overview

We targeted the low cost Spartan-3 FPGA. This family of FPGAs are low cost integrated circuits made of 100,000 to 1.6 million system gates. In order to implement the design an overview of the resources available to us is also very important. Referring to [11] the Spartan-3 architecture has five basic components :

- Configurable Logic Blocks (CLBs)

  Consists of RAM based look up table to implement logic and storage elements that can be used as flip-flops or latches. CLBs can be programmed to perform a wide variety of logic functions as well as store data.

- Input Output Blocks (IOBs)

  Controls the flow of data between IO pins and internal logic. Each IOB supports bidirectional data flow, 3-state operation, and numerous different signal standards.

- Block RAM (BRAM)

  Provides data storage in the form of 18-kbit dual-port/single-port blocks.

- 18 bit Multiplier Blocks

  Accepts two 18 bit binary numbers as input to calculate the product.

- Digital Clock Manager (DCM)

  Provides means for distributing, delaying, multiplying, dividing and phase shifting clock signals.

These elements are organized as shown in the Fig. 3.3

3.5 Design Process

Given the set of specifications and interface that we are going to use, designing any algorithm now requires two phases

- Phase I

  Identifying the resources available on that particular implementation device and exploring the ones we require for our implementation.

- Phase II

  Identifying the area minimization techniques applicable on the algorithm based on its functionality and the resources available on the device.

3.5.1 Phase I: Architectural Overview

The initial step requires us to study the algorithm in detail and then design a pseudo code based on our understanding. This means we need to identify and list out the important functions it performs. For Keccak the only major functions it has is Cyclic Shift Offset, XOR and other logic functions. It has a huge internal state of1600 bits, divided into 25 words of 64 bits each. Intermediate values generated during operations needs to be stored for future processing. Using registers to store such huge data consumes a lot of area. Instead

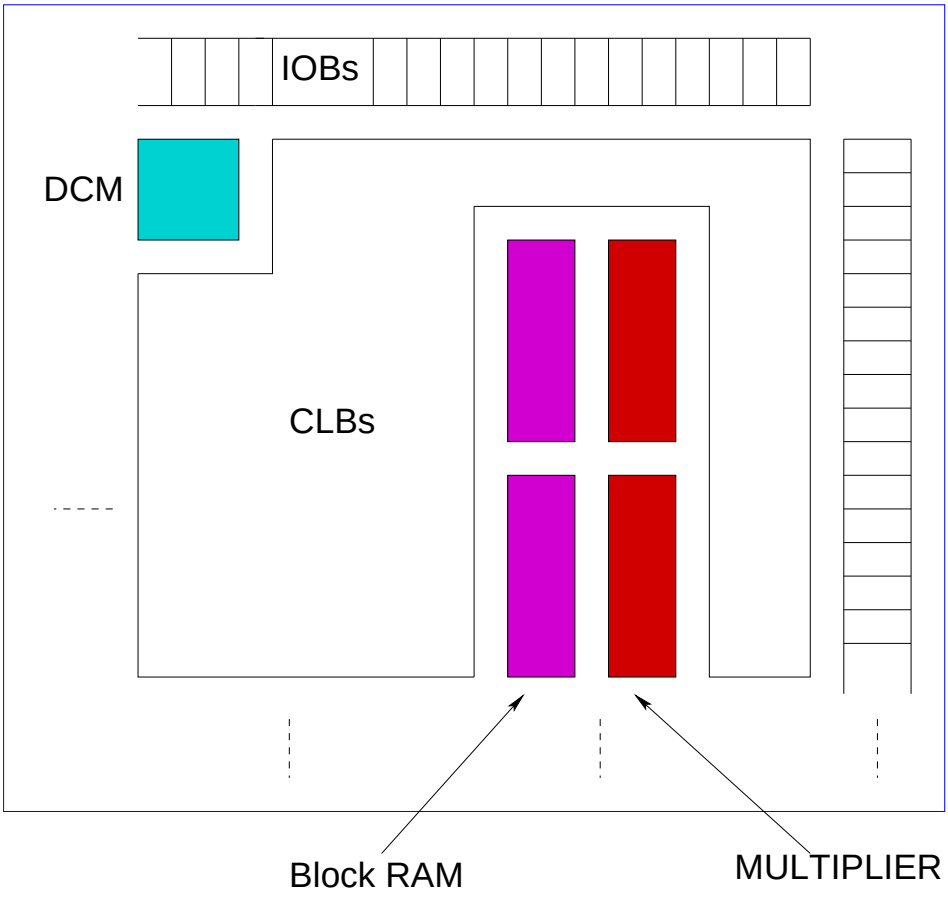


Figure 3.3: Spartan-3 Architecture

the embedded memory resource available looks like a better compact option. Out of the five major components we identified on the Spartan-3 device it is observed that we require 3 of them namely The CLBs, IOBs and the Block RAM. The next step is then to explore these components in a bit detail.

1. **Configurable Logic Blocks (CLBs)**

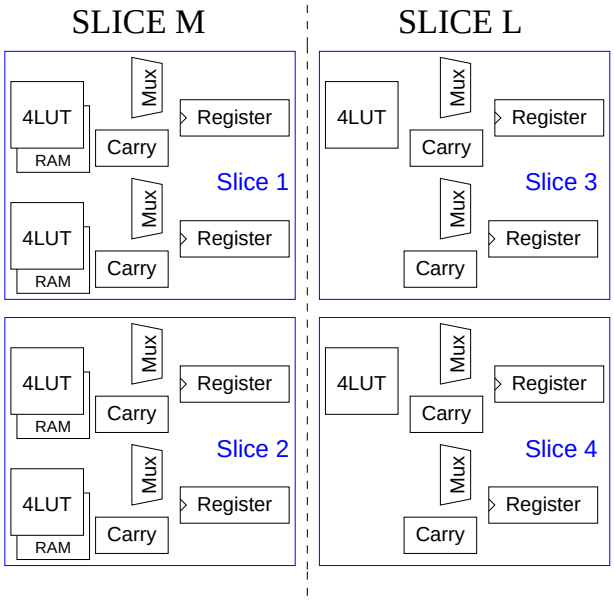


Figure 3.4: Basic Components in CLB

The Configurable Logic Blocks (CLBs) is the main logic resource for implementing synchronous as well as combinatorial circuits. A CLB is composed of four slices, each slice further contains two sets of four input Look-Up Tables (LUTs) to implement logic and two dedicated storage elements that can be used as flip-flops or latches. The two 4-input LUTs may be used as a 16X1 memory (RAM16) or as a 16-bit shift register (SRL16), while additional multiplexers and carry logic simplify wide logic and arithmetic functions. Since Each LUT has four inputs this implies that any logical function with four inputs can be easily implemented in a Look Up Table. Since a slice

is composed of such two LUTs, all logical functions are automatically mapped to the slice resources in the CLBs. Therefore this knowledge aids in estimating the number of slices required. This is particularly useful when trying to optimize a design.

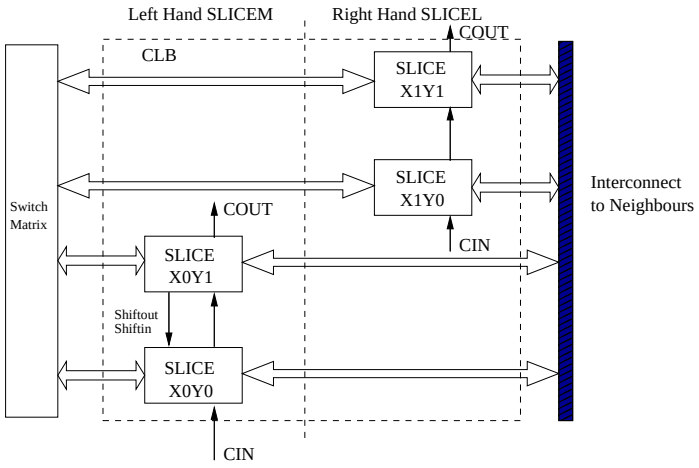


Figure 3.5: Configuration Logic Block Architecture

2. **Distributed RAM**

In a CLB, the LUT within each SLICEM function generator resource optionally implements a 16-deep x 1-bit synchronous RAM. The LUTs within a SLICEL slice do not have distributed RAM. Distributed RAM writes synchronously and reads asynchronously. However, if required the register associated with each LUT could implement a synchronous read function. Distributed RAM are sometimes preferred over Block RAM in many high-performance applications that require relatively small embedded RAM blocks, such as FIFOs or small register files. The Distributed RAM can be implemented in two ways :

- Single-port RAM with synchronous write and asynchronous read. Synchronous reads are possible using the flip-flop associated with distributed RAM.
- Dual-port RAM with one synchronous write and two asynchronous read ports.

Like single-port RAMs synchronous reads are possible.

The four LUTs in each CLB allows us to configure a 64-bits single port RAM or a 32-bit dual port RAM shown in Fig. 3.6.

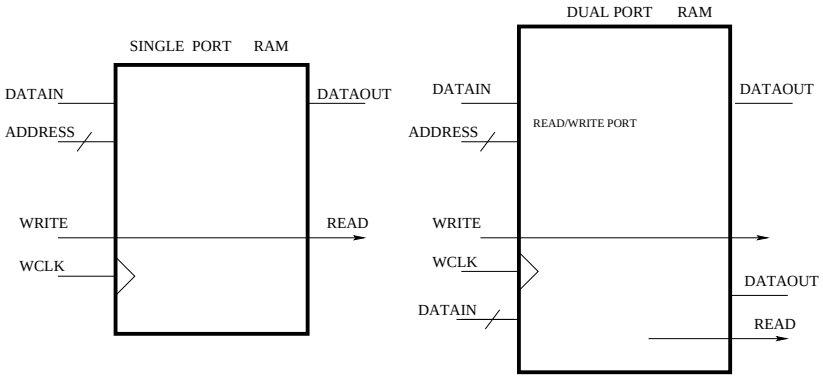


Figure 3.6: Single and Dual Port Distributed RAMs

3. **Shift Registers**

The Look-Up Table (LUT) in a SLICEM can configured as a 16 bit shift register,

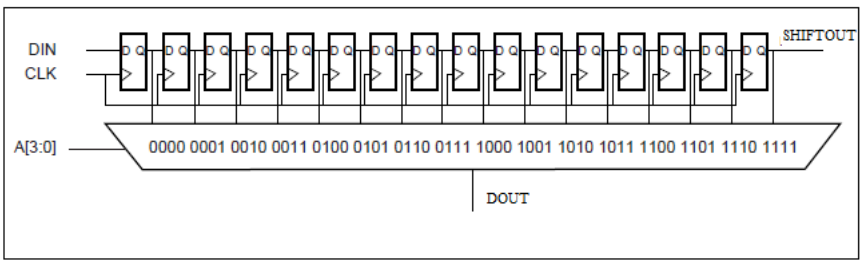


Figure 3.7: LUT Configured as an Addressable Shift Register

called SRL16. The Shift-in operations are synchronous with the clock. A separate dedicated output allows the possibility of cascading multiple 16-bit shift registers in

order to create a shift register of any needed size. Thus each CLB resource can be configured using four of the eight LUTs as a 64-bit shift register. The Look-Up Table can be described as a 16:1 multiplexer. The four inputs serve as select lines for the data programmed into the Look-Up Table. With the SRL16 configuration, the fixed LUT values are configured instead as an addressable shift register.

4. **Block RAM**

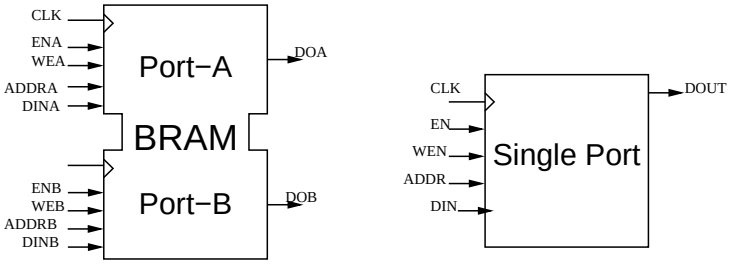


Figure 3.8: Single and Dual Port Block RAMs

Storing large amount of data in the logic resources is a waste of space as they consume a lot of area. Instead, a dedicated resource on the device called Block RAMs are used for this purpose. The [12] is a good source to understand BRAMs present in Spartan-3 FPGAs. BRAMs have large memory space of upto 18kbits to store data but are limited by the number of I/O ports available. Like DRAMs they can be configured in two modes i.e either a single port of 64-bits or dual port with 32 bits on each port. The dual port BRAM has two completely independent access ports. The addressing associated with the ports is independent of each other and so are the read and the write operations. Both the ports are governed by their own individual clocks which can either be synchronized to a single clock or both can run at different frequencies. Generally in practice we prefer to source them by the same clock. Because of the limited number of ports available, the amount of data that can be read and written during a clock cycle is also limited. There are two modes of operation that you can

set your Block RAM to work in

- Read First Write Next mode :

  It reads the old data from the memory location being addressed in that particular clock cycle before updating that memory location with the new data available at.

- Write First Read Next mode:

  In this mode of operation the memory location being addressed gets updated with the new data at its input, and this new data is then read out. In this mode the previous stored value thus gets lost.

We prefer to use the read first write next mode. An important thing to remember is that,for BRAM the data is written to the address applied in the current clock cycle, but data read out is from the address given in the previous clock cycle. In Fig. 3.9 "00" is the address location given during the first clock cycle, while the data is read out from that location during the second clock cycle.The data available at the input port during the second clock cycle is M[01] which is read in and stored at location "01" addressed during that clock cycle.

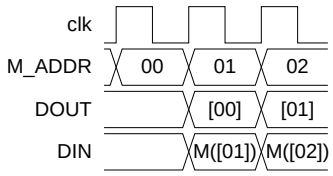


Figure 3.9: Block RAM Timing Diagram

3.5.2 Phase II: Area Minimization Tecchniques

1. **Datapath**

   Thinking about the area minimization techniques, the most obivious approach that

one can think of is folding. This can either be *horizontal folding* or *vertical folding* as shown in Fig. 3.10. The folding depends on the kind of parallelism the algorithm has along its vertical or horizontal axis. In horizontal folding the critical path gets reduced by the factor by which it was folded. While in vertical folding folding the number of bits processed in clock cycle gets reduced. In both the cases the area gets reduced but the number of clock cycles increases comparably by the folding factor. Again the extent of folding depends on parallelism certain sub functions have in the algorithm.

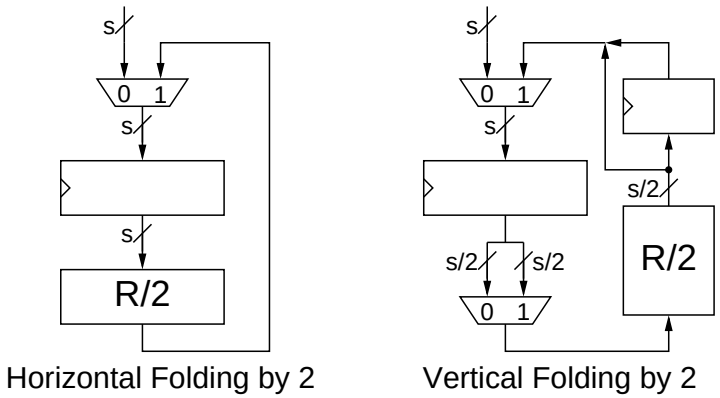


Figure 3.10: Folding

Another method which is generally used to increase the overall throughput of any system is *Pipelining* as shown in Fig. 3.11. This involves performing multiple tasks simultaneously using different resources. By using registers an instruction can be divided into stages. The potential speed up depends on the number of pipeline stages. The output of each stage may serve as an input to the next stage which allows smooth data forwarding. However there are limitation to pipelining such as if an instruction depends on the result of a prior instruction still in the pipeline. In such cases the data needs to be stored and forwarded at the right time. This involves interleaving computations between two pipeline stages and is known as *quasi-pipelining*. This may

require introduction of stalls at certain stages in order to wait for data needed by certain functional units which may require certain cycles before proceeding forward. This is to ensure the correct functionality of the algorithm during the execution of multiple cycles simultaneously. We may also perform certain independent computations to refrain from wasting clock cycles.

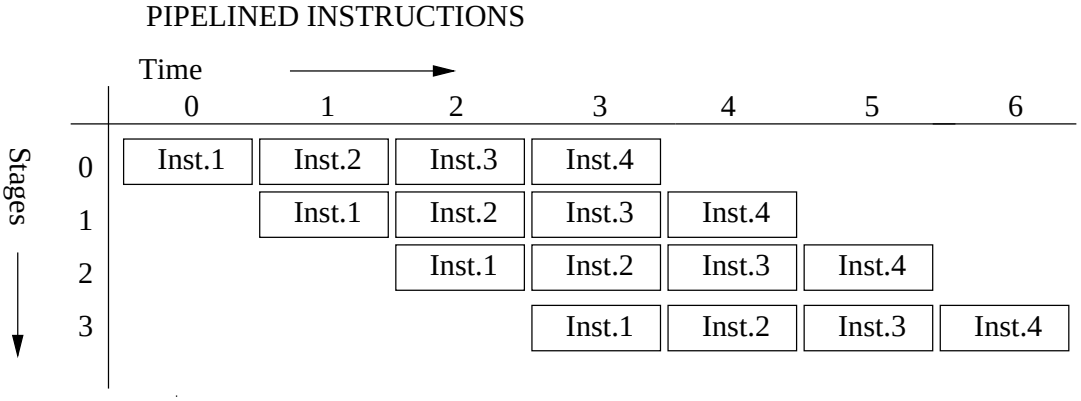


Figure 3.11: Pipelining

The last technique is the *Rescheduling*. At certain points even during pipelining it is observed that data interdependencies may lead to introduction of stalls in the pipeline. This increases the depth of the pipeline stages which could potentially lead to slow down in our over all system. By rescheduling of data in certain manner during pipelining we want the subsequent data generated to be independent so that the pipelined stall are eliminated or at least reduced. At the same time the overall functionality of the algorithm is not affected.

2. **Controller**

Designing of the controller for any low area implementation is a challenge. The controller typically consists of Finite State Machines (FSM), ROMs to store the control signals, Counter to calculate the clock cycles during each state and few registers to

hold in certain data or states. The area of a controller is mostly dependent on the size of the ROM. The control signals define the width of the ROM while the clock cycles define its depth. As the area of the datapath reduces, the number of clock cycles incresase as well as the switching pattern between control signals. This increases the size of the ROM and thus overall area in such a way that in certain cases the size of the controller may become greater than the overall area of the datapath. In order to reduce the size of the controller we use the concept of sequence or bit matching between control signals. Firstly all the signal transitions during each clock cycle are observed and noted down. We try to look for signals that have the same bit pattern, so that the two different destinations can be sourced from a single sourcel. This reduces the overall number of states and thus the controller size. The addressing scheme in the BRAM is of main concern to us. In case of Keccak because of data interdependencies and potential dedicated write cycles, the addressing needs to be stored as control bits in ROM which leads to consumption of significant area.

Chapter 4: Keccak

4.1 Introduction

Keccak [13] is a cryptographic hash function designed by Guido Bertoni, Joan Daemen, Michal Peeters and Gilles Van Assche. Keccak is one of five finalists in the NIST hash function competition to select a SHA-3 algorithm.

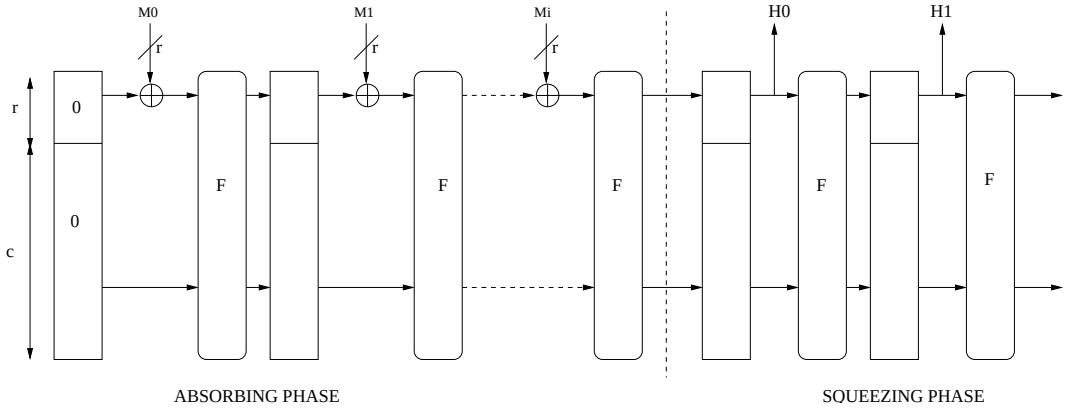


Figure 4.1: Sponge Construction

Keccak belongs to the family of hash functions that is based on the sponge construction. The underlying theme behind the sponge construction is a fixed length iterated permutation (or transformation) operating on the intial state b, called the width, for building a function F that maps a variable-length input to a variable length output. Keccak has two phases, called *absorbing* and *squeezing*. The message M is padded with a specified padding scheme such that it is always a multiple of the block size r. During the absorbing stage, the initial state b is initialized to all zeros and the r bits of the input blocks are XORed with the first

r bits of the state and processed through the function F. When all the input blocks are processed in a similar manner, it switches to the squeezing mode where the first r bits of the state are returned as output blocks.

Some of the key features behind choosing the sponge construction for Keccak are

- The Sponge construction that allows for modes that are provably secure against generic attacks
- It is a kind of a block cipher without a key schedule
- The choice of operations it consists are simple XOR, AND, NOT and cyclic shift offsets. There is no look uptables, arithmetic operations or data dependant operations.
- It is a function capable of generating variable length output.
- It is flexible in terms of security level, by trading in bit rate for capacity, without changing the permutation .

Due to all of the properties mentioned above it can be used for multiple functions like MAC function, stream cipher, mask generating function, reesedable pseudo random bit generator as well as efficient authenticated encryption.

The permutation for Keccak[b] is chosen from a set of seven values such that b $\in$ 25, 50, 100, 200, 400, 800, 1600, where b is the width of the permutation. Here b= r+c, where r is the bit rate and c is the capacity. The number of rounds Nr depends on the permutation width, and is given by Nr= 12 + 2l, where 2^l= b=25. This gives 24 rounds for Keccak-f[1600] state.

NIST requires the candidate algorithms to support at least four different output lengths n $\in$ 224, 256, 384, 512 with associated security levels. Depending on these output variants the input parameters for Keccak varies as shown in the Table 4.1. For the 256 bit digest thus the default b state is 1600. The initial round 1 consisted of 12+ l rounds which were increased to 12+2l in round two. The other changes included were in the values of capacity and bit rate. The capacity was made twice the value of the output digest. The tweaks in

Table 4.1: Parameters for Keccak

Algorithm	Round	bit rate (b)	cpapacity (c)	Identifier
Keccak-224	1	1024	576	28
Keccak–256	1	1024	576	32
Keccak–384	1	512	1088	48
Keccak–512	1	512	1088	64
Keccak-224	2	1152	448	28
Keccak–256	2	1088	512	32
Keccak–384	2	832	768	48
Keccak–512	2	576	1024	64

the third round is the removal of the diversifier d and the simplification of padding scheme to pad rule 10*1.

4.1.1 Notations

The psudocode for the sponge mode of operation is described as follows

KECCAK[r,c,d](M)

INITIALIZATION AND PADDING

$S[x,y] = 0, \qquad \forall\ (x,y)\ in \quad (0\cdots4, 0\cdots4)$

P $= M \parallel$ 0x01 $\parallel$ byte (d) $\parallel$ byte(r/8) $\parallel$ 0x01 $\parallel$ 0x00 $\parallel \cdots \parallel$ 0x00

ABSORBING PHASE

for every block P_i in P

$S[x,y] = S[x,y] \oplus$ P$_i$[x+5y] $\qquad \forall\ (x,y)$ such that x+5y < r/w

S $= KECCAK - f[r+c](S)$

$SQUEZZINGPHASE$

Z $= emptystring$

$while output is requested$

Z $=$ $Z \parallel S[x+y],$ $\forall\ (x,y)$ such that x+5y < r/w

S $=$ $KECCAK - f[r+c](S)$

return Z

Here

- M : Is the original message.
- P : Message after padding. The padded message P is organised as an array of blocks Pi that are further organized as arrays of lanes.
- S : Denotes the state as an array of lanes.A lane is a set of w bits with constant x and y coordinates.

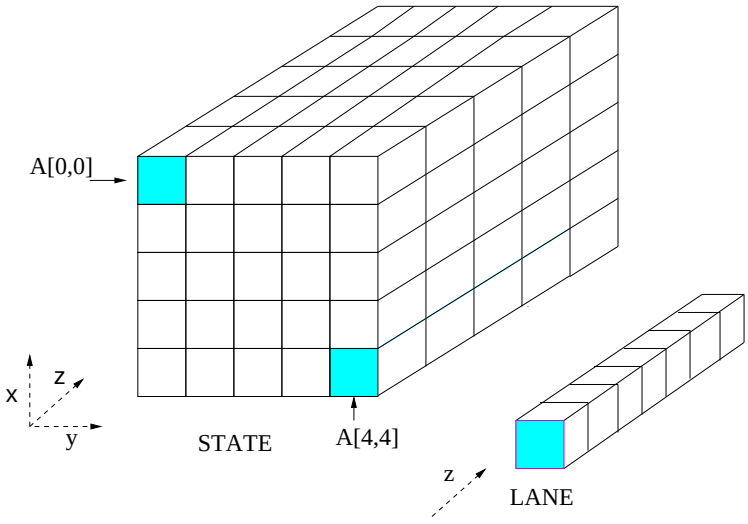


Figure 4.2: State and Lane

4.1.2 Constants

The Keccak 256 requires a set of cyclic offsets. Each of the values of these offsets depends on the lane on which it operates and is given by r[x,y]. The values are shown in the Table 4.2

Table 4.2: r[x,y] Cyclic Shift Offsets

Axis	x=0	x=1	x=2	x=3	x=4
y=0	0	1	62	28	27
y=1	36	44	6	55	20
y=2	3	10	43	25	39
y=3	41	45	15	21	8
y=4	18	2	61	56	14

Each block processing requires 24 rounds of operation. Each round has a certain round constant needed at the end of the round and is shown in Table 4.3

Table 4.3: Round Constants [RC]

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

4.2 Keccak-256

The Keccak-256 requires an input message size of 1088 bits. In the initial state the message M is concatenated with a set of 512 bits of 0's to get the input state of 1600 bits before processing. Each set of message block is processed for 24 rounds before hashing out.

4.2.1 Round Mode

KECCAK-$f[b]$(A)

for i in $0 \cdots \mathrm{n}_r - 1$

$A = \mathrm{Round}[b](\mathrm{A},\mathrm{RC}[i])$

return A

The general mode of operation consists of iterating over 24 rounds for a block of message. The message is stored as a state array A of 1600 bit size in total . Each round [b](A,RC[i]) basically consists of invertible steps that operate on states.

Round$[b]$(A,RC)

θ *STEP*

C[x] $= A[x,0] \oplus A[x,1] \oplus A[x,3] \oplus A[x,4], \quad \forall\ x\ in \quad 0 \cdots 4$

D[x] $= C[x-1] \oplus ROT(C[x+1],1), \quad \forall\ x\ in \quad 0 \cdots 4$

A[x,y] $= A[x,y] \oplus D[x] \quad \forall\ (x,y)\ in \quad (0 \cdots 4, 0 \cdots 4)$

ρ *AND* π *STEPS*

B[y,2x+3y]$= ROT(A[x,y], r[x,y]) \quad \forall\ (x,y)\ in \quad (0 \cdots 4, 0 \cdots 4)$

χ *STEP*

A[x,y]$= B[x,y] \oplus ((NOTB[x+1,y])AND\ B[x+2,y]), \forall\ (x,y)\ in \quad (0 \cdots 4, 0 \cdots 4)$

ι $STEP$

A[0,0]= $A[0,0] \oplus RC$

$returnA$

- A : Permutation state array at the beginning of the round
- A[x,y]: Particular lane in the state array A
- B[x,y],C[x],D[x]: Intermediate variables
- XOR : Exclusive OR
- AND : Bitwise logic AND
- NOT : Bitwise logic NOT
- OR : Bitwise logic OR
- ROT(W,r) : Bitwise cyclic shift modulo the lane size i.e cyclic left rotator

The 1600 bits is divided into 5x5 array of 64 bits each. This initialized state array is called A. The sequence of operations are performed in the order shown above. The θ operation is simple XORing and rotation between internal states. It is important to remember that each of these operations being performed across the lane is always a modulo of the lane size. The ρ stage applies cyclic shift offsets to the intermediate state array A resulting from the operations in θ. The amount of offset depends on the value specified in the Table 4.2 and is dependant upon the location of the lane. The π invloves the rearrangements of the permutation state array into a new intermediate matrix B. This new matrix B serves as an input to our χ stage. This stage performs pure logical operations of XOR, AND and NOT to give the final resultant State array A. The ι stage simply consits of a XOR operation between the lane A[0,0] of the finalized state array A with a Round Constant. The value of the Round Constant is again dependant on the round and is defined in the

Table 4.3. The finalized State array A thus now serves as the new input state array A for the next round.

4.3 Block Diagram Of Keccak-256

The following is the block diagram for the datapath of Keccak-256 :

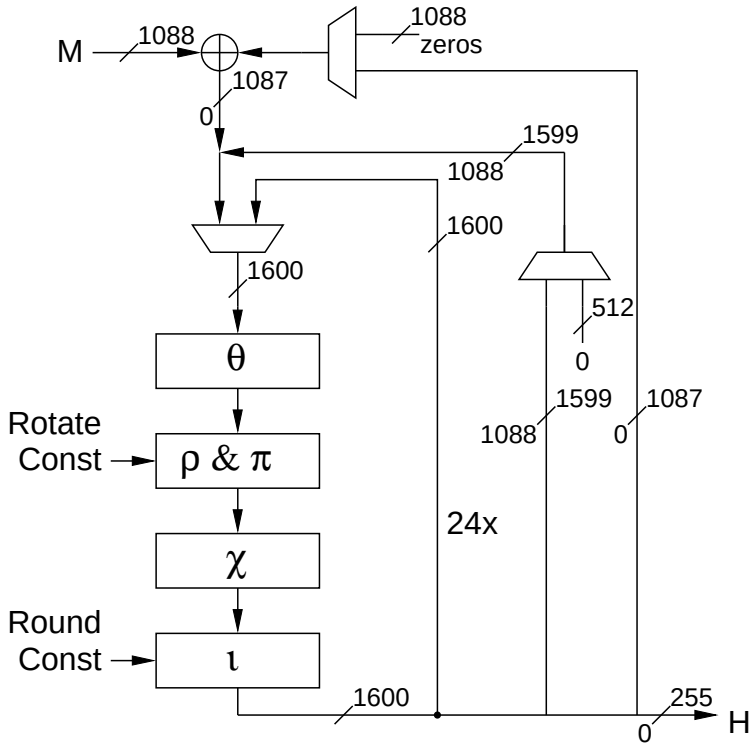


Figure 4.3: Block Diagram For Keccak

Chapter 5: Lightweight Implementations of Keccak

5.1 Scheduling Data Flow in Keccak

The flow of operations for Keccak have been discussed in chapter 4. Before designing the datapath it is important to understand how we can schedule the execution of operations with as few clock cycles as possible. As per the scheme the 1600 bit initial state is stored as 25 states in a 5X5 state array A of 64 bits each. This initial state goes through a series of five major function during a round. In a straightforward manner the flow of data through the functions is in the order of θ, ρ, π, χ and ι. We have divided processing of the round in two functions. Function I consists of the θ , ρ and π function while Function II consists of χ and ι. For our first implementation we decided to schedule the execution of operations as shown in the Table 5.1. Here a level is of 5 clock cycles each, assuming intermediate XOR and shift operation takes one clock cycle. The value of y axis in the Table 4.1 varies from 0 to 4. Stage 1,2,3 and 4 comprise of the θ function while the stage 6 performs the ρ and π function. The stages at each level are being executed in parallel. The output from each preceding stage serves as an input for succeeding stages. An estimate is that these functions should be executed on an average of 40 clock cycles. The χ involve simple logical operations of XOR, NOT and AND between three 64 bit lanes of the intermediate state array B at a time to generate new state array A. This should take an average of another 35 clock cycles. The final operation ι involves XORing of the lane A[0,0] with a constant pre defined for that round. This should take another 3 clock cycles.

Table 5.1: Function I : Pipelining of θ , ρ and π Function

Level(=5 clock cycles)	Stage 1	Stage 2	stage 3	Stage 4	Stage 5
I	A[0,y]	C[0]			
II	A[2,y]	C[2]	D[1]		
III	A[1,y]	C[1]		A'[1,y]	B[y,2+3y]
IV	A[4,y]	C[4]	D[3]		
V	A[3,y]	C[3]	D[2]	A'[3,y]	B[y,6+3y]
VI	A[2,y]	C[2]	D[0]	A'[2,y]	B[y,4+3y]
VII	A[0,y]	C[0]	D[4]	A'[0,y]	B[y,0+3y]
VIII	A[4,y]			A'[4,y]	B[y,8+3y]

5.2 Design I

Taking the basic scheduling scheme assumed in the previous section we came up with the following datapath shown in Fig. 5.1.

The various components serve the following purpose

- Block RAM(BRAM) : This is responsible for storing the initial state array A as well as the 24 round constants .
- Reg A,B,C,D,X : These are used to store the intermediate states C[x] generated at each level (every 5 clock cycles) .
- ρ and π : Consists of a 64 bit barrel shifter as shown in Fig. 5.2. π simply implies the rearrangement of data.
- Single port DRAM(SDRAM) : stores the intermediate state array B. Because of data interdependency and pipelining among among the stages, the presence of the DRAM eliminates the possibility of Address contention in BRAM. This is specially important while storing intermediate data like state array B, at locations that contain data which might be reused for some other operation in future.
- Reg1,Reg2,Reg3 : reads in the values from SDRAM to serve as an input to the logic function χ.

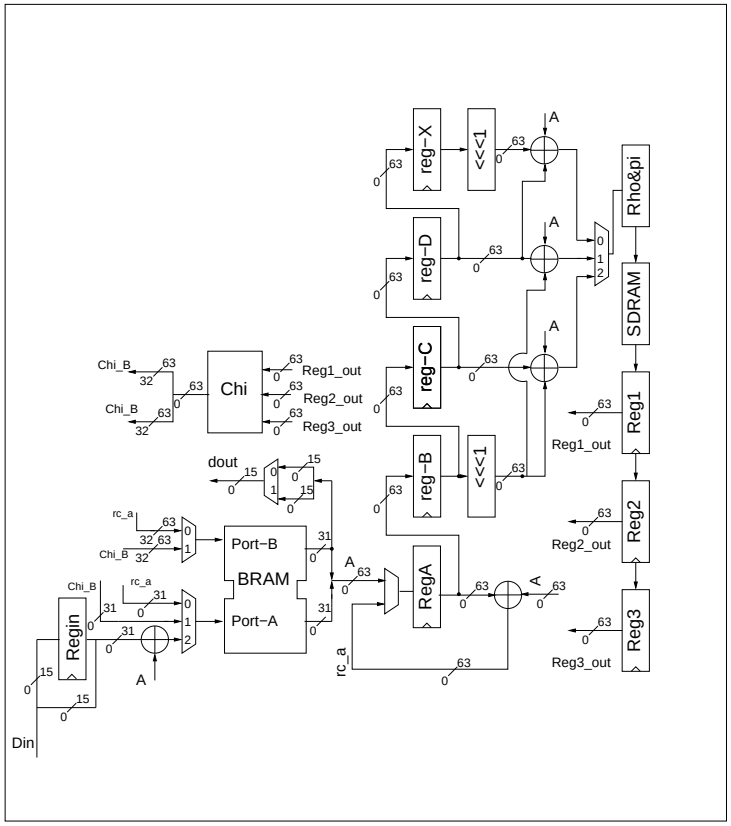


Figure 5.1: Datapath of Basic Keccak Design I

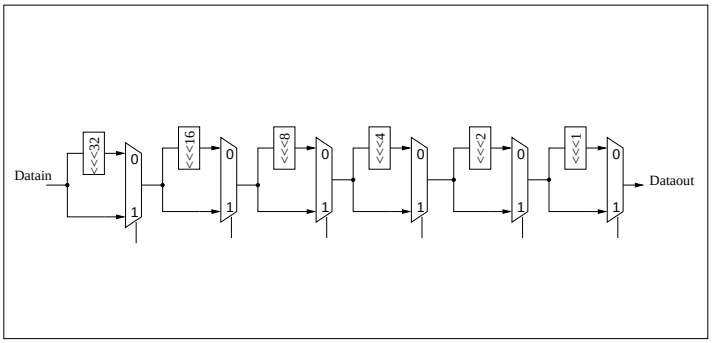


Figure 5.2: 64 bit Barrel Shifter

As discussed already that in a BRAM data is written to the address applied in the current clock cycle, while data is read from the address of the previous clock cycle. Following the scheduling scheme in Table 5.1 we execute the Function I in 42 clock cycles. While performing the χ operation, in order to ensure efficient utilization of data read during each clock cycle from the DRAM, the new state matrix A is generated columnwise from intermediate values B stored in the DRAMs. One column of such A[x,y] is generated in 9 clock cycles, while the whole operation takes 45 clock cycles. The ι operation reads in the value of the round constant from the BRAM and XORs it with A[0,0] before storing it back into the BRAM at the end of the round. This step requires another 4 clock cycles. Thus a whole round in this datapath takes a total of 91 clock cycles for processing.

The controller is based on Finite State Machines(FSM) along with ROMs to hold the states of the control signal states during processing. The finite state machine switches between the states depending on the control signals and the execution cycle. These control signals are stored as control words in the ROMs. The major area consumption of controller is thus defined by the size of the ROMs .

To optimize the controller we first identify all the required control signals and note down the bit switching during each clock cycle. Then we clubbed those signals together that had similar switching patterns. This process is somewhat similar to horizontal or vertical folding of a state into sub-states based on the parallelism offered by the bit patterns in control signals. Avery simple example in this case would be sourcing the enables for Register A,B,C,D,X and Register 1,2,3 from the same output of ROM in the controller. The addressing scheme for the BRAM need 8 bit to address the memory locations for the data. Another 5 bits is required to address the data in the DRAM. The select signals for the required offset in the variable rotator require another 6 bits. In some cases it is observed that the signals may follow a certain bit pattern similar to one or a combination of the bits offered by the counter simultaneously. In such cases those control words are sourced directly by the counter which saves area at no extra cost.

Despite using various minimization techniques the overall consumption in area was over

800 slices which was 200 more than our original target. Further this design could not fit into the smallest Spartan-3 devices which has only 768 slices. This motivated us to focus on minimizing the area in the datapath itself and led to the Design II architecture.

5.3 Design II

Although Design I was unable to meet our area requirements, it was good base to build our second architecture upon. The main focus of this design architecture was to cut down on the area from the previous version. The new datapath is as shown in the Fig. 5.3. The

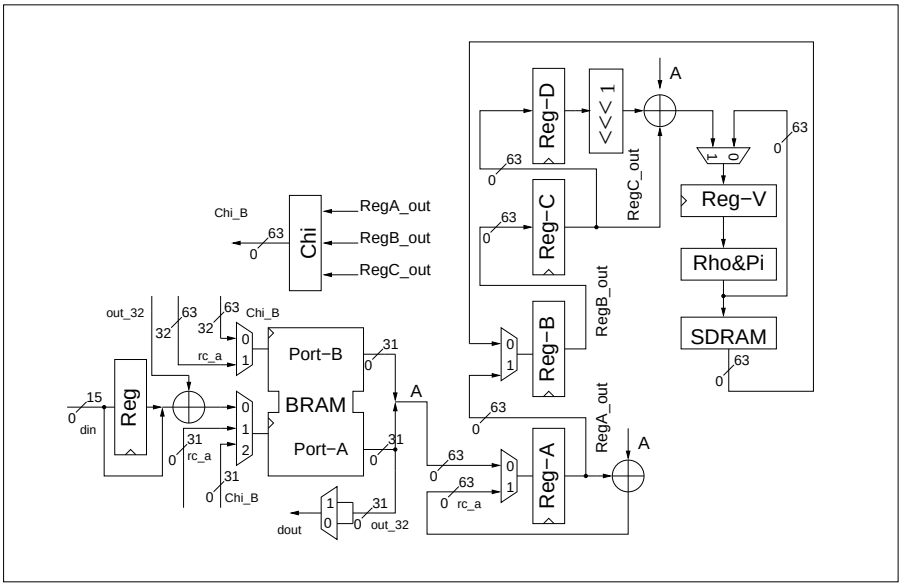


Figure 5.3: Datapath of Keccak Design II

notable difference in the architecture is

1. Removal of four sets of registers i.e Reg1, Reg2, Reg3 and Reg X. And addition of a register before the variable rotator to hold the intermediate rotation offset.

2. Downsizing the 64 bit barrel shifter(192 slices) to 16 bit (128 slices). This is as shown in the Fig. 5.4.

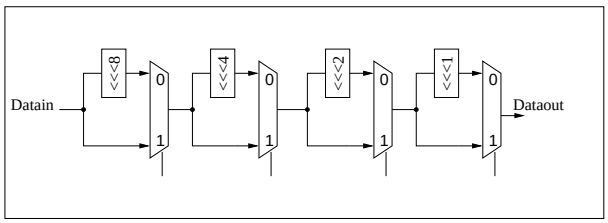


Figure 5.4: 16 bit Barrel Shifter

The order of scheduling of data read out from the BRAM needed some revisions in order to accommodate the structural change . But surprisingly we were still able to compute Function I in 42 clock cycles by downsizing the rotator as the 25 sets of cyclic shift offsets now required 58 clock cycles which is 33 more than the one in design I. Since our design was still pipelined at different levels along the various stages this change in offset cycles meant we needed to introduce stalls at the preceding stages to prevent data misalignment that may affect the overall functionality. Introducing stalls in pipeline stages thus quasi-pipelined our overall datapath. The DRAM still serves the same function as before and stores the intermediate array state B. We now employ the re-usability of the registers A, B and C in the datapath to serve as a source of input for the values read from DRAM, into the logic function χ. The controller employed followed a similar design structure to the one in design one with a few extra clock cycles. The overall area consumed is around 620 slices which is almost 200 slices cut down from the previous version. However this is at the cost of 33 clock cycles per round or 792 clock cycles more per block of message. Which is a good trade off considering our area requirements. However we were still over budget with our area requirements, and this led to the designing of the third datapath where our goal was to make the design even more smaller than before to reach our area target.

5.4 Design III

With the Design II architecture as our basis again the structural changes now included

1. Removal of Register D
2. Removal of DRAM
3. Modified Rotator that can only shift the 25 sets Keccak needs. On an average it takes 1.5 clock cycles per offset and consumes only 128 slices. The values chosen as offsets were obtained after playing around with minimum possible combination of offsets for all the listed sets of rotations. The advantage was that for the same number of slices you get lesser clock cycles for as well as shorter critical path.

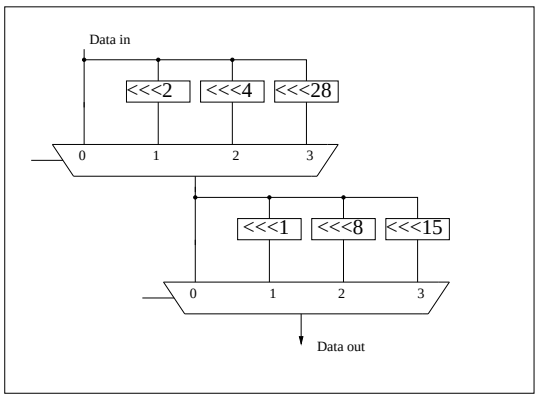


Figure 5.5: Modified Offset Rotator

The datapath is as shown in the Fig. 5.6.

With the removal of register D, the data read from the BRAM during the θ, stage had to be rescheduled in a different order. The ρ and π had to be quasi pipelined as the intermediate values of state array B,now needed dedicated clock cycles to be stored back into the BRAM. Therefore these dedicated clock cycles enable us to avoid address contention for locations that reused data for processing in future. For remedying this situation we also

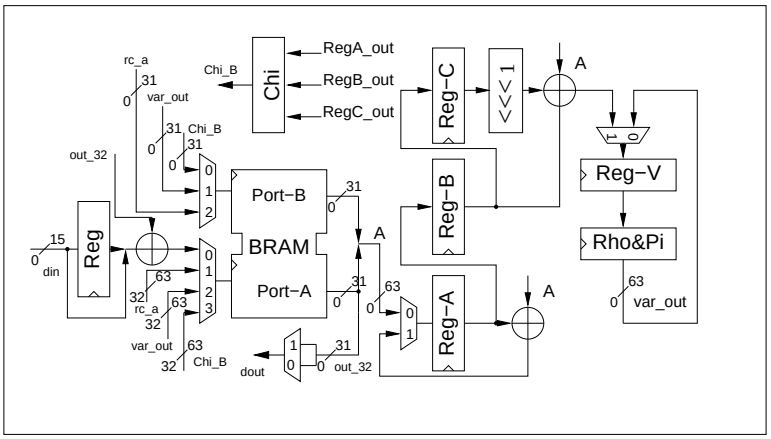


Figure 5.6: Datapath of Keccak Design III

needed to introduce suitable stalls which had no fixed patterns and varied depending upon the offset cycles and the scheduling scheme. These three operations of Function I now took total of 91 clock cycles. Since the data for the χ had to be read from the BRAM processed and stored back into it at the same time we again had to use another sets of dedicated write cycles to store back the values at the correct locations. There is no difference for the ι operation. One round is now computed in 153 clock cycles. Although we were able to get downsize the datapath, the greatest set back however came in the form of overall area of the design because of the huge size of the controller. The final design exploded to 627 slices with almost 700 clock cycles more. This is because of the quasi-pipelining and random address generation during each clock cycle, made the minimization of the ROM size difficult in the controller. For the cut down in the area of the datapath the corresponding increase in the size of the controller was almost more than double which defeated the purpose of our earlier goal. This made us realize that cutting down on the datapath and increasing the number of clock cycles may not be the solution to our goal.

5.5 Design IV

The paper [14] is a good source of reference for various scheduling techniques and algorithms. It specially has a section that described the principle behind resource constrained scheduling. Since the controller was a big issue in our previous design, we decided to give a little bit of leeway in the datapath size and re-structure our design based on some of the scheduling schemes described in this paper. We then decided to do the following changes :

1. Removal of register C such that we now only have two sets of registers A and B. This increases the number of clock cycles at the θ stage.

2. Full width 64 bit barrel shifter to remove any stalls in the system for regularized addressing pattern.

3. Reorganizing the data locations in a new manner inside the Block RAM so as to generate an addressing pattern during loading or processing or finalization.

4. Adding the Single port DRAM back into the datapath to remove address contention and removes the dedicated clock cycles.

Based on these targets our datapath is designed as shown in the Fig. 5.7 We again rescheduled the order in which data was read from the BRAM into the register for initial processing to suit the execution order in the new datapath. The mode of operation here is similar to the pipelining architecture we had for our Design II. The only difference is that the output of the Sinle port DRAM now serve as the third input to the χ along with the output of Register A and B in the absence of a register C. The θ, ρ and π function now took 57 clock cycles for execution. While the χ and ι took another 49 clock cycles. One round was thus executed in 106 clock cycles. With these new modifications we were finally able to get a datapath for Keccak under 600 slices on Spartan-3 device. From this observation we realized that the regularized pattern had helped in the optimization of the controller.

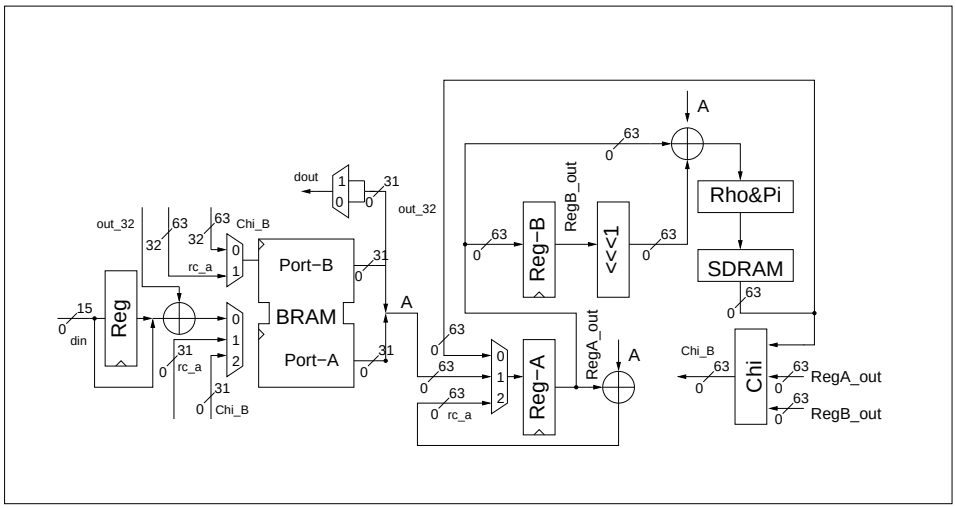


Figure 5.7: Datapath of Keccak Design IV

5.6 Design V: Distributed RAM Architecture

With all the different possible architectures with Block RAM. It piqued our interest to see how the performance of our design would vary if implemented using pure logic resources on FPGAs. In order to facilitate the replacement of Block RAMs we gave ourselves a new goal of implementing Keccak in under 800 slices on the Spartan-3 device using only logic resources. For our logic version we can splits the BRAM into four single port Distributed RAMs of 16-bits each to store the initial state.The message being read is in chunks of 16 bits at a time. The round constants can be easily stored in a ROM. This datapath is based on the Design II version of our Block RAM with minor tweaks. This tweak includes using the modified rotator instead of the full 64 bit shifter. The four registers(A through D) at the output of the four DRAM's compute the θ stage. The output for the ρ and π is now stored in an additional 64-bit single port Distributed RAM. The DRAM aids in de-coupling the read from and write into the Block RAM which eliminates address contention. This saves us from using dedicated write clock cycle. The ρ , π and θ stages are now computed in 58 clock cycles, while the χ and ι stages take another 39 clock cycles. Overall one round is

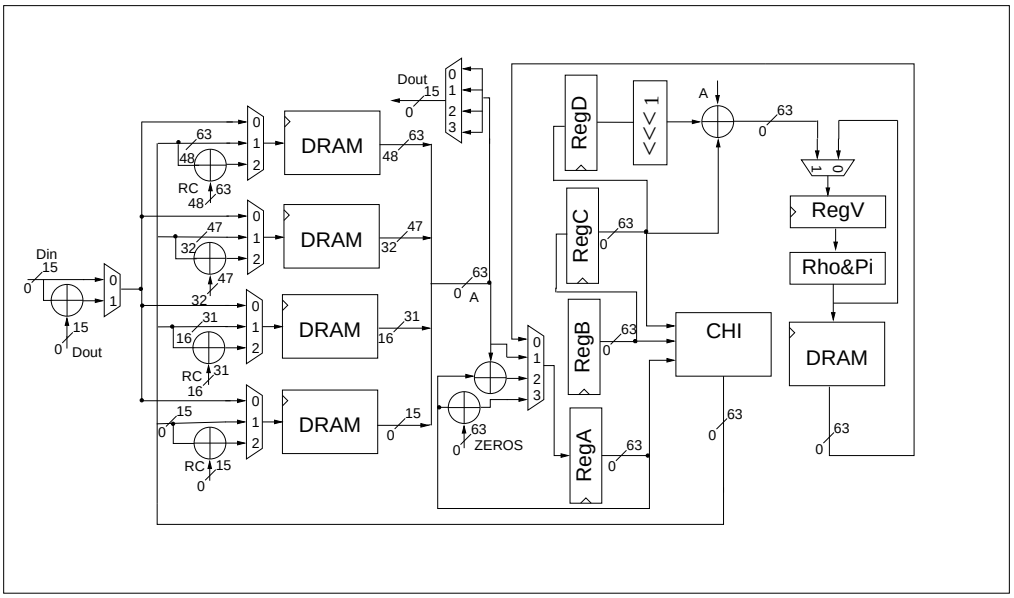


Figure 5.8: Datapath of Distributed RAM Based Architecture for Keccak

processed in 97 clock cycles while one block is computed in 2328 clock cycles. It consumes 768 slices and just manages to fit into the smallest Spartan 3 device.

Table 5.2: Modified Rotator Offset Combinations

r[x,y]	ρ	0	2	4	28	0	1	8	15
r[0,0]	0	✓				✓			
r[0,1]	36				✓			✓	
r[0,2]	3		✓				✓		
r[0,3]	41			✓	✓		✓	✓	
r[0,4]	18	✓	✓				✓		✓
r[1,0]	1	✓					✓		
r[1,1]	44	✓			✓			✓✓	
r[1,2]	10		✓					✓	
r[1,3]	45		✓		✓	✓			✓
r[1,4]	2		✓			✓			
r[2,0]	62			✓	✓				✓✓
r[2,1]	6	✓		✓			✓✓		
r[2,2]	43	✓			✓	✓			✓
r[2,3]	15	✓							✓
r[2,4]	61	✓	✓		✓		✓		✓✓
r[3,0]	28				✓	✓			
r[3,1]	55			✓	✓			✓	✓
r[3,2]	25	✓	✓					✓	✓
r[3,3]	21		✓	✓		✓			✓
r[3,4]	56				✓✓	✓✓			
r[4,0]	27	✓		✓				✓	✓
r[4,1]	20	✓		✓			✓		✓
r[4,2]	39		✓		✓		✓	✓	
r[4,3]	8	✓						✓	
r[4,4]	14		✓	✓		✓		✓	

Chapter 6: Results and Conclusions

6.1 Results and Analysis of Lightweight Implementations

We implemented different lightweight architectures for our 256 bit hash version of Keccak. These different architectures were designed using various optimization techniques discussed in Chapter 3. Keccak requires simple logic functions and the intermediated datas show inter-dependencies. This eliminated athe option of employing folding in the structure. Therefore the only options left were to quasi- pipeline the structure and schedule the data efficiently during each clock cycle. A brief description of the structural differences in the various architectures of Keccak thus implemented is shown in the Table 6.1.

Table 6.1: Datapath Summary for Different Schemes of Keccak-256 Implemented

Desdin No.	Version	Register A	Register B	Register C	Register D	Register X	Register 1	Register 2	Register 3	SDRAM	Rotator Version
Design I	BRAM	✓	✓	✓	✓	✓	✓	✓	✓	✓	64-bit
Design II	BRAM	✓	✓	✓	✓					✓	16-bit
Design III	BRAM	✓	✓	✓							modified
Design IV	BRAM	✓	✓							✓	64-bit
Design V	Logic	✓	✓	✓	✓					✓	64- bit

Apart from the low area constraint the performance of our implementations is judged on the basis of Throughput to Area ratio. The number of clock cycles needed to execute N blocks of message determines the throughput of the system. The number of clock cycles needed for executing the same algorithm may vary depending on how it is implemented. Comparing the two tables Table 6.1 and Table 6.2 we observe that, the introduction of a

Table 6.2: Throughput Formulae for Different Implementations of Keccak

Version	Block Size (bits) b	Rounds	Function I $(\theta\ \rho\ \pi)$ clock cycles	S Function II $(\chi\ \iota)$ clock cycles	Clock Cycles per Round	Clock Cycles to hash N blocks $clk =$ $st + (\ l + \ p) \cdot N + \ end$	Throughput (long Messages) $\frac{b}{(l+p) \cdot T}$
Design I	1088	24	42	49	91	$2 + (\ 68 + \ 2184) \cdot N + \ 17$	$1088/(\ 2252 \cdot T)$
Design II	1088	24	42	81	123	$2 + (\ 68 + \ 2952) \cdot N + \ 17$	$1088/(\ 3020 \cdot T)$
Design III	1088	24	91	63	154	$2 + (\ 68 + \ 3696) \cdot N + \ 17$	$1088/(\ 3764 \cdot T)$
Design IV	1088	24	57	49	106	$2 + (\ 68 + \ 2544) \cdot N + \ 17$	$1088/(\ 2612 \cdot T)$
Design V	1088	24	59	38	97	$2 + (\ 68 + \ 2328) \cdot N + \ 17$	$1088/(\ 2396 \cdot T)$

single port DRAM to store in the intermediate states in the architecture reduces the number of clock cycles by almost one third. Since the barrel shifter was one of the biggest components in our design, cutting down on the size did not ensure that the final implementation met our area constraints. On the other hand using a modified version of the rotator saved clock cycles and reduced the critical path. Storing of data into particular memory locations of BRAM is also an important factor in our designs. Based on the order of execution, the data can be stored into continuous or interleaved memory locations. This also contributes to generating a definite order of control bit sequence during processing. The address signals can then be generated with simple counters or logic and save space on ROMs in the controller.

The graphs in Fig. 6.1 and Fig. 6.2 give an idea about the area consumption and hardware performance of all our different lightweight implementations of Keccak on Xilinx devices. In Fig. 6.1 we observe that Design version V was second biggest implementation in terms of slices on Spartan-3 but on Virtex-5 it is one of the smallest ones. While comparing only BRAM versions of our designs if Design IV was the smallest possible implementation on Spartan-3 our target device, but on all the other devices Design III is the winner in terms of low area implementation. A similar unpredictable pattern can be observed in the Throughput versus Area results.

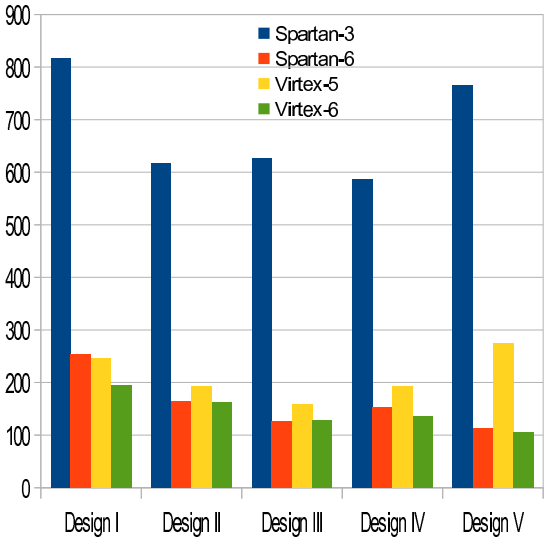


Figure 6.1: Area Consumption on Xilinx Devices

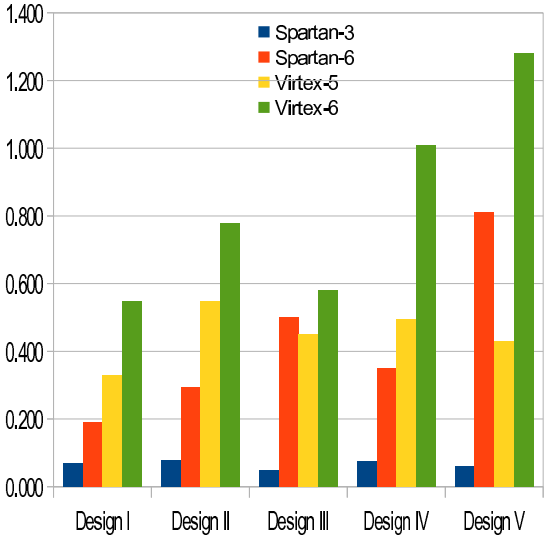


Figure 6.2: Throughput Versus Area Performance on Xilinx Devices

Table 6.3: Results on Xilinx Spartan-3

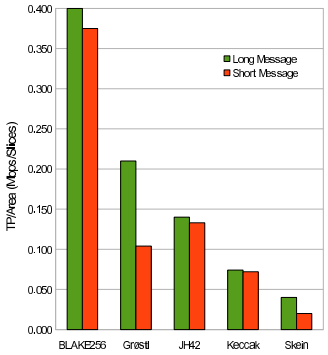
Algorithm / Keccak	Area (slices)	Block RAMs	Maximum Delay (ns) T	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)
Design I	817	1	8.68	55.65	0.068	53.72	0.065
Design II	618	1	7.33	49.11	0.079	48.84	0.079
Design III	627	1	8.90	32.5	0.05	32.20	0.05
Design IV	586	1	9.41	43.12	0.074	42.98	0.073
Design V	766	0	9.83	46.2	0.06	45.8	0.060

From the results obtained in Table 6.3 we can clearly see that the best implementation according to our area budget and performance is the Design version IV of our BRAM implementation.

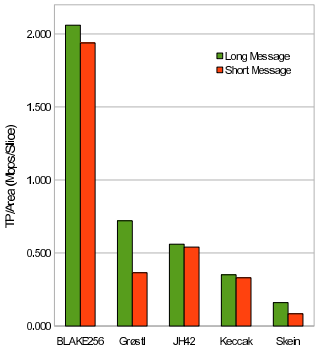
6.2 Comparison Of SHA 3 Finalists

For low area implementations, after meeting the area considerations it is also important that they perform be efficient in terms of throughput. For a fair comparison of performance in hardware among the five finalists, the performance metrics we thus use is throughput versus area. The Cryptographic Engineering Research Group at GMU,Virginia implemented the other four finalists with the same design criteria of area constraints and targeting the Spartan-3 device [15]. We propose Design IV of Keccak-256 for comparison with these reported results. We have implemented them on various Xilinx devices also for relative comparison of their performance. The Result of our implementations is as shown in the Table 6.5. These results were generated using AtheNA [16].

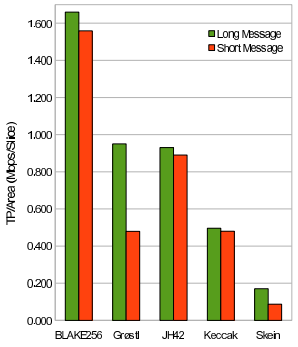
The comparison of the lightweight implementations of the finalist on Xilinx and Altera devices gives some interesting observations. The ranking of the candidates based on throughput over area in Fig. 6.3 shows that Keccak is not one of the best candidates for



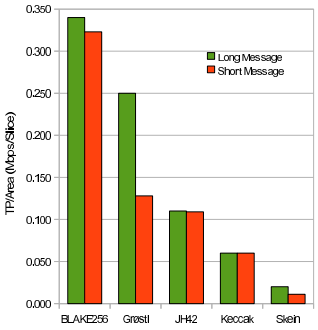
(a) on Spartan-3 (BRAM)



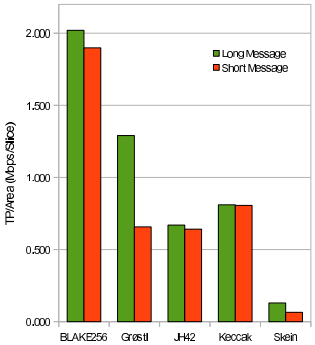
(b) on Spartan-6 (BRAM)



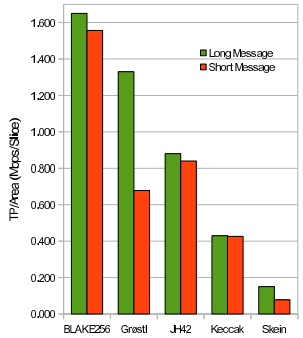
(c) on Virtex-V (BRAM)



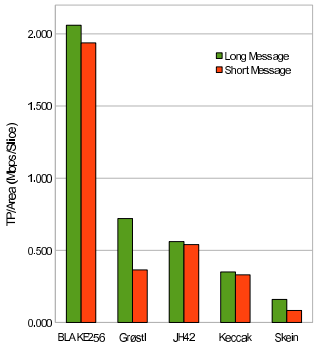
(d) on Spartan-3 (Logic)



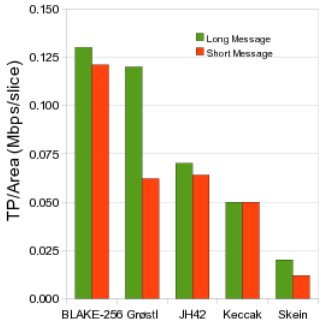
(e) on Spartan-6 (Logic)



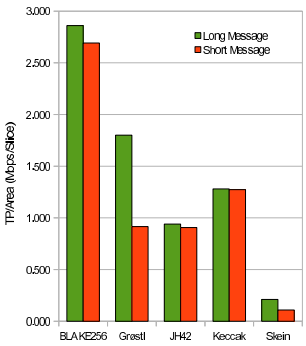
(f) on Virtex-V (Logic)



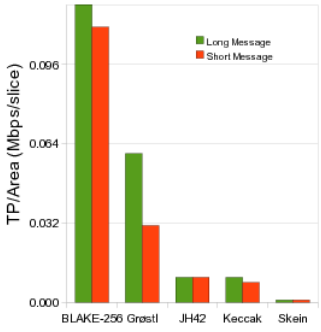
(g) on Virtex-6 (BRAM)



(h) on Cyclone-II (BRAM)



(i) on Virtex-6 (Logic)



(j) on Cyclone-II (Logic)

Figure 6.3: Throughput Over Area Ratio on Xilinx and Altera Devices

Table 6.4: Comparison of Lightweight Implementations of SHA-3 finalists on Xilinx FPGAs

Algorithm	Reference	Device	I/O Width	Datapath Width	Clock Cycles per block (p)	Area (slices)	Maximum Frequency (MHz)	Throughput (Mbps)	TP/Area (Mbps/slice)
Keccak	[Kerckhof]	xc6vlx75t-1	64	64	2137	144	250	128	0.897
Keccak	[Jungk]	xc6vl	32	200		397	197	1071	2.69
Keccak	[TW]	xc6vlx75t-1	16	64	2612	135	335	136	1.010
Keccak	[Jungk]	xc5v	32	200		393	159	864	2.19
Keccak	[TW]	xc5vlx20-2	16	64	2612	198	231	98	0.496
Keccak	[Jungk]	xc3s	32	200		1665	71.2	387	0.23
Keccak	[TW]	xc3s50pq-2	16	64	2612	586	106.2	43.12	0.074

low area implementations. This could be attributed to the fact that the performance of the algorithm is affected by its scalability. In case of Keccak the interdependency of data and irregular diffusion patterns in the state matrix leads to over thousands of clock cycles per block of message and hence affects the overall throughput. At the same time its inability to scale down makes it huge in size compared to others.

6.3 Comparison with Other Published Results

One of the first reported results was by Bertoni et al.[13] with a proposed low co-processor design on Virtex-5 device. This design used external system memory instead of having all the storage capabilities internally. There is no information about the interface or the protocols. But one round is processed in 215 clock cycles while it consumes an area of 448 slices which is almost twice the size of our current implementation. There are also other reported results by Kerckhof et al. [7] and Jungk et al. [6] but on different Xilinx devices. Although we targeted the spartan-3 device in our implementations we still synthesize them on other devices to make comparisons with these reported results .

The comparison in Table 6.4 shows that we have the smallest reported implementation so far. While considering the throughput over area ratio the comparison is somewhat difficult. Jungk et al. [6] implements Keccak with a 200 bit datapath width. and uses 25 sets of 8∗

8 DRAMs. The huge datapath width and the independent read functions for each operand during an operation aid in reducing the execution cycles to almost one tenth in practice. But all of this is at the cost of almost double the area.

Similarly while looking at the architectures of Keccak in chapter 5 we observe that the two independent set of Single port DRAMs eliminate the address contention issue we always have while using BRAM implementations. The results for the comparison have been shown in the Table 6.5

6.4 Conclusion

In this paper we presented the different low area implementations of SHA-3 finalist Keccak. Only 256 bit digests were implemented and the designs targeted the low cost Spartan-3 device. We were able to implement it in the given area constraint. Compared to the other reported results it is one of the smallest implementations. The comparison with other BRAM versions of the SHA-3 finalist implemented at GMU [15] help us in ranking Keccak with respect to other finalists in terms of its performance in the resource constrained environment. Apart from the basic functions that affect the number of clock cycles, the scalability of an algorithm also plays an important role in its performance. The graph in Fig.6.1 shows that the scalability also varies with the architecture of the device upon which it is implemented. Through our different versions of Keccak we observed that the datapath and the controller do not share a direct or even a inverse relationship in terms of area. We thus realized that the key to scalability and performance of Keccak is efficient scheduling without data dependency especially in memory based implementations.

Table 6.5: Implementation Results of Keccak on Xilinx

Device	Algorithm	Version	Area (slices)	Block RAMs	Long		Short		
					Maximum Delay (ns) T	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)
Spartan 3	Keccak-256	I	817	1	11.23	55.653	0.068	55.6	0.065
	Keccak-256	II	608	1	8.869	51.393	0.079	51.4	0.77
	Keccak-256	III	627	1	8.97	32.2	0.5	32.3	0.52
	Keccak-256	IV	570	1	12.805	43.606	0.074	42.8	0.077
	Keccak-256	V	766	0	12.83	16.6	0.02	8.5	0.04
Spartan 6	Keccak-256	I	240	1	10.327	46.7	0.19	46.2	0.2
	Keccak-256	II	163	1	8.459	42.5	0.26	42.7	0.25
	Keccak-256	III	133	1	5.07	57.0	0.43	56.7	0.427
	Keccak-256	IV	143	1	7.809	53.3	0.37	54.2	0.3
	Keccak-256	V	161	0	5.77	78.7	0.49	78.1	0.42
Virtex 5	Keccak-256	I	246	1	5.95	83.12	50.33	82.5	0.29
	Keccak-256	II	193	1	3.402	108.9	0.549	107.6	0.5
	Keccak-256	III	159	1	4.04	71.6	0.45	71.3	0.42
	Keccak-256	IV	193	1	5.019	99.6	0.49	98.9	0.48
	Keccak-256	V	275	0	3.85	118.1	0.43	117.2	0.42
Virtex 6	Keccak-256	I	192	1	4.637	110.02	0.548	109.02	0.53
	Keccak-256	II	157	1	3.10	129.73	0.779	128.2	0.7
	Keccak-256	III	129	1	3.84	75.2	0.58	74.9	0.58
	Keccak-256	IV	135	1	2.977	141.43	1.010	140.4	1.09
	Keccak-256	V	106	0	3.34	136.0	1.28	135	1.27

Table 6.6: Implementation Results of our implementations of SHA-3 Candidates

Device	Version	Message Algorithm	Area (slices)	Block RAMs	Maximum Delay (ns) T	Long Throughput (Mbps)	Long TP/Area (Mbps/slice)	Short Throughput (Mbps)	Short TP/Area (Mbps/slice)
Xilinx Spartan-3 (xc3s50-5)	BRAM	BLAKE-256	549	1	8.05	219.3	0.40	205.9	0.375
		Grøstl	594	1	7.65	122.4	0.21	61.9	0.104
		JH42	502	1	9.19	69.6	0.14	34.0	0.068
		Keccak	586	1	9.414	43.12	0.074	43.10	0.072
		Skein	498	1	10.65	19.7	0.04	10.0	0.020
		SHA-2	547	1	8.48	101.5	0.19	98.4	0.180
	Logic only	BLAKE-256	631	0	8.16	216.3	0.34	203.0	0.322
		Grøstl	766	0	6.83	192.6	0.25	97.9	0.128
		JH42	558	0	10.05	63.7	0.11	31.2	0.056
		Keccak	766	0	9.83	46.2	0.06	45.8	0.060
		Skein	766	0	12.83	16.6	0.02	8.5	0.011
		SHA-2	745	0	8.52	137.8	0.19	132.1	0.177
Xilinx Spartan-6 (xc6slx4csg-3)	BRAM	BLAKE-256	152	1	5.63	313.8	2.06	294.5	1.938
		Grøstl	271	1	4.80	195.0	0.72	98.7	0.364
		JH42	182	1	6.23	102.6	0.56	50.2	0.276
		Keccak	153	1	7.77	53.8	0.35	53.4	0.0.34
		Skein	182	1	7.19	29.2	0.16	14.8	0.081
		SHA-2	140	1	5.93	145.2	1.04	140.7	1.005
	Logic only	BLAKE-256	164	0	5.34	330.6	2.02	310.2	1.882
		Grøstl	230	0	4.43	297.3	1.29	151.2	0.657
		JH42	156	0	6.14	104.2	0.67	51.0	0.327
		Keccak	113	0	4.95	91.8	0.81	91.1	0.806
		Skein	190	0	8.77	24.3	0.13	12.4	0.065
		SHA-2	227	0	5.74	204.6	0.90	196.0	0.864
Xilinx Virtex-5 (xc5vlx20-2)	BRAM	BLAKE-256	248	1	4.29	411.9	1.66	386.6	1.559
		Grøstl	271	1	3.65	256.5	0.95	129.8	0.479
		JH42	176	1	3.91	163.5	0.93	80.0	0.454
		Keccak	198	1	4.314	98.2	0.496	96.2	0.85
		Skein	218	1	5.69	36.9	0.17	18.7	0.086
		SHA-2	234	1	3.98	216.2	0.92	209.5	0.895
	Logic only	BLAKE-256	271	0	3.94	448.2	1.65	420.7	1.552
		Grøstl	313	0	3.15	417.4	1.33	212.3	0.678
		JH42	183	0	3.99	160.3	0.88	78.5	0.429
		Keccak	275	0	3.85	118.1	0.43	117.2	0.426
		Skein	246	0	5.66	37.7	0.15	19.2	0.078
		SHA-2	312	0	4.24	277.0	0.89	265.4	0.851
Xilinx Virtex-6 (xc6vlx75T-1)	BRAM	BLAKE-256	163	1	5.06	348.7	2.14	327.3	2.008
		Grøstl	241	1	4.09	229.1	0.95	115.9	0.481
		JH42	196	1	4.11	155.4	0.79	148.9	0.760
		Keccak	135	1	2.977	136.9	1.010	136.3	1.009
		Skein	207	1	6.00	35.0	0.17	17.8	0.086
		SHA-2	155	1	4.84	177.8	1.15	172.3	1.111
	Logic only	BLAKE-256	166	0	3.72	474.6	2.86	445.4	2.693
		Grøstl	263	0	2.78	473.3	1.80	240.7	0.915
		JH42	171	0	3.96	161.5	0.94	154.9	0.906
		Keccak	106	0	3.34	136.0	1.28	135.0	1.273
		Skein	193	0	5.17	41.3	0.21	21.0	0.109
		SHA-2	238	0	3.86	304.2	1.28	291.5	1.225

Table 6.7: Implementation Results of SHA-3 Finalists on Altera

Device	Version	Message / Algorithm	Area (LEs)	Memory Bits	Maximum Delay (ns) T	Long: Throughput (Mbps)	Long: TP/Area (Mbps/slice)	Short: Throughput (Mbps)	Short: TP/Area (Mbps/slice)
Altera CycloneII(ep2c8f256c6)	BRAM	BLAKE-256	1,367	2,048	9.98	176.9	0.13	166.0	0.121
		Grøstl	1,221	3,072	6.26	149.6	0.12	75.7	0.062
		JH42	1,045	3,840	9.15	69.9	0.07	66.9	0.064
		Keccak	996	8,192	5.48	52.7	0.05	52.5	0.053
		Skein	930	4,096	9.89	21.6	0.02	11.0	0.012
	Logic only	BLAKE-256	2,019	0	7.39	238.8	0.12	224.9	0.111
		Grøstl	3,937	0	5.52	238.4	0.06	121.2	0.031
		JH42	5,527	0	10.05	63.7	0.01	61.1	0.011
		Keccak	6,247	0	8.49	53.5	0.01	53.1	0.008
		Skein	6,141	0	15.83	13.5	0.001	6.9	0.001

Bibliography

Bibliography

[1] *Secure Hash Standard (SHS)*, National Institute of Standards and Technology (NIST), FIPS Publication 180-2, Aug 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[2] X. Wang, Y. Yin, and H. Yu, "Finding collisions in the full sha-1," in *Advances in Cryptology - CRYPTO*, 2005.

[3] 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," Second SHA-3 Candidate Conference, Tech. Rep., 2010.

[4] E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs," Cryptology ePrint Archive, Report 2010/445, 2010, http://eprint.iacr.org/.

[5] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGA," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. LNCS, S. Mangard and F.-X. Standaert, Eds., vol. 6225. Springer Berlin / Heidelberg, 2010, pp. 264–278.

[6] B. Jungk and J. Apfelbeck, "Area-efficeint FPGA implementations of the SHA-3 finalists," in *International Conference on ReConfigurable Computing and FPGAs*. IEEE: ReConfig'11, DEC 2011.

[7] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale, and F.-X. Standaert, "Compact fpga implementations of the five sha-3 finalists," in *10th Smart Card Research and Advanced Application Conference, CARDIS 2011*, Leuven, Belgium, Sep 2011.

[8] "The SHA-3 Zoo, Information Societies Technology (IST) Programme of the European Commission, note = http://ehash.iaik.tugraz.at/wiki/the_sha-3_zoo."

[9] K. Gaj, J.-P. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Y. Brewster, "ATHENa – Automated Tool for Hardware EvaluatioN: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs," in *20th International Conference on Field Programmable Logic and Applications - FPL 2010*. IEEE, 2010, pp. 414–421, winner of the FPL Community Award.

[10] *Hardware Interface of a Secure Hash Algorithm (SHA)*, v. 1.4 ed., Cryptographic Engineering Research Group, George Mason University, Jan 2010.

[11] *Spartan-3 Generation, FPGA User Guide*, Ug331 (v1.2) ed., Xilinx, Inc., Apr 2007.

[12] *Using Block RAM in Spartan-3 Generation FPGAs*, Xapp463 (v2.0) ed., Xilinx, Inc., Mar 2005.

[13] G. Bertoni, J. Daemen, M. Peeters, and G. Van Asche, "Keccak function version 2.0," Sep 2009.

[14] S. Govindrajam, "Scheduling algorithms for high level synthesis," University of Cincinnati, Dept of ECECS, University of Cincinnati, Term Paper, March 1995.

[15] J.-P. Kaps, P. Yalla, K. K. Surapathi, B. Habib, S. Vadlamudi, S. Gurung, and J. Pham, "Lightweight implementations of SHA-3 candidates on FPGAs," in *Progress in Cryptology – INDOCRYPT 2011*, ser. Lecture Notes in Computer Science (LNCS), D. J. Bernstein and S. Chatterjee, Eds., vol. 7107. Springer, Dec 2011, accepted, to be published.

[16] "ATHENa results database," http://cryptography.gmu.edu/athenadb/, Automated Tool for Hardware EvaluatioN project.

Curriculum Vitae

Smriti Gurung received her Bachelor degree from Delhi College of Enigneering under Delhi University, India in 2008. She completed her Masters Degree in Computer Engineering from George Mason University, Farifax in May 2012. She has worked as a Teaching Assistant to ECE 332 course from Spring 2010 to Fall 2011. She is also a part of the Cryptographic Engineering Research Group (CERG)group at GMU since Spring 2010.