# COMPACT IMPLEMENTATIONS AND BENCHMARKING OF TWO SHA-3 FINALISTS BLAKE AND JH ON FPGAS

by

Susheel Choudary Vadlamudi
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:

| | |
|---|---|
| [illegible] | Dr. Jens-Peter Kaps, Thesis Director |
| KGaj | Dr. Kris Gaj, Committee Member |
| [illegible] | Dr. Peter Pachowicz, Committee Member |
| [illegible] | Dr. Andre Manitius, Department Chair of Electrical and Computer Engineering |
| [illegible] | Dr. Lloyd J.Griffiths, Dean, Volgenau School of Engineering |
| Date: 01/20/12 | Spring Semester 2012<br>George Mason University<br>Fairfax, VA |

# Compact Implementations and Benchmarking of Two SHA-3 Finalists BLAKE and JH on FPGAs

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

By

Susheel Choudary Vadlamudi
Bachelor of Science
Vignan Engineering College, 2009

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

Spring Semester 2012
George Mason University
Fairfax, VA

## Dedication

I dedicate this work to my father Sri. Vadlamudi Lakshmi Narayana Rao and my mother Smt. Vadlamudi Vinoda.

## Acknowledgments

This thesis would not have been possible without the support of my thesis advisor Dr. Kaps, who not only guided but also encouraged me throughout my academics. Thank you for bearing with me all this time and for the help you extended.

Secondly, I would like to thank Dr. Gaj for his valuable suggestions throughout my thesis and course work. I would also like to thank each and every CERG group member who gave their ideas and suggestions with friendly chat. I specially thank Rajesh, Panasayya, Kishore, Smrithi and Mahidhar who helped me to finish my work according to the plan and shared their experiences. Finally, I would like to express my deep gratitude to my family members who gave moral support when I needed the most.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

COMPACT IMPLEMENTATIONS AND BENCHMARKING OF TWO SHA-3 FINALISTS BLAKE AND JH ON FPGAS

Susheel Choudary Vadlamudi, M.S.

George Mason University, 2012

Thesis Director: Dr. Jens-Peter Kaps

Security and message authentication play a crucial role in communications. The current hashing standards for message authentication are SHA-1 and SHA-2. Attacks on SHA-1 which show potential vulnerabilities of this algorithm were published in 2005. As SHA-2 is based on SHA-1 it might be vulnerable to the same attacks. National Institute of Standards and Technology (NIST) initiated a contest to determine a new American hash standard called SHA-3. After evaluating the initial 64 algorithms that were submitted, five algorithms were selected for the final round. The five finalists are: BLAKE, Grøestl, JH, Keccak and Skein. Security and performance in hardware as well as software were the key factors in choosing the five finalists. Evaluating the performance of these algorithms in resource constrained environments, like PDAs and smart phones is of major interest for mobile ubiquitous computing. New low-power Field Programmable Gate Arrays (FPGA), which are suitable for battery powered devices, have low non-recurring engineering cost and faster time to market than Application Specific Integrated Circuits (ASIC).

We designed compact architectures for BLAKE and JH, targeting Xilinx low-cost Spartan-3 FPGAs. To achieve a good throughput to area ratio we developed different architectures, maintaining the design criteria of 800 slices, or 400-600 slices with one Block RAM, respectively, on Xilinx Spartan-3 devices. We compared the performance of our implementations by synthesizing them on several devices from Xilinx and Altera. Our compact implementations of BLAKE and JH outperform other published results in terms of throughput to area ratio. Considering the lightweight implementations of all the five finalists, BLAKE has the best performance and JH has an average performance.

# Chapter 1: Introduction

In recent years, new methods are employed for improving ways of communication. The purpose of communication is to transmit information to others clearly and unambiguously. This information has to be secured from attacks by unauthorized users. To make the information and communication to be secure we use *cryptography*, which uses the two security goals. They are:

- *Confidentiality*: Information must be protected from unauthorized personals.
- *Integrity*: Data can be changed only by authorized entities, using authorized methods.

Cryptography protects information by transforming it into unreadable format called *cipher text.* Only the authorized personal can decipher the information. In this way the confidentiality of the data can be maintained. In addition to confidentiality, integrity of the data has to be preserved. To preserve the integrity, the data is passed through an algorithm called as a *cryptographic hash function.*

A Hash function is an algorithm which takes arbitrary data as a block and returns a fixed length of bits called as *hash value.* Any changes to the data produces a new hash value thus maintaining the integrity of the information. The arbitrary data is referred as *message* and the hash value is termed as *message digest* or simply *digest.*

For a cryptographic hash function to be secure it should satisfy three main properties. They are:

- Preimage resistance

  The hash algorithm should be a one way function. Given a hash value $h$ it should be difficult to find the message $m$ such that $hash^{-1}(h) = (m)$.

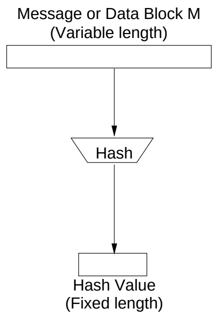


Figure 1.1: Hash Function

- Second preimage resistance

  For two different messages $m_1$, $m_2$, where $m_1 \neq m_2$. It should be difficult to find a message $m_2$ by using message $m_1$ such that hash($m_1$) = hash($m_2$).

- Collision resistance

  It should be difficult to find two messages $m_1$ and $m_2$ that produces the same hash value. It can be represented as hash($m_1$) $\neq$ hash($m_2$).

## 1.1 Secure Hash Standard

The *Secure Hash Standard*, also referred to as *Secure Hash Algorithm* is a standard developed by the National Institute of Standards and Technology (NIST) and published as Federal Information Processing Standard (FIPS 180). The FIPS 180-3[1] specifies five hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 for federal use and these algorithms were widely adopted by information industry.

With the advancement of technology, attacks on several cryptographic hash algorithms were reported[2]. Though the algorithms were not approved by NIST, serious attacks on NIST approved SHA-1 were also published in 2005[3], which show potential vulnerabilities of this algorithm. SHA-2 architecture is based on SHA-1 making it vulnerable to same attacks. NIST conducted two workshops to analyze the security of the Secure-Hash-Algorithms(SHA). As a result of these workshops they concluded to develop a new American hash standard through a public competition similar to AES[4]. NIST announced the competition for the third SHA candidate on November 2007. The winning algorithm will be called as SHA-3. As a part of the competition sixty four entries were submitted to NIST by October 2008. Among the sixty four candidates only fourteen qualified for the second round on July 2009. One year time line was alloted to evaluate the security and performance of these algorithms in hardware and software. On December 2010, five algorithms progressed to the third and final round. The five algorithms are:

- BLAKE
- Grøstl
- JH
- Keccack
- Skein

A year was allocated to deeply evaluate the performance of these algorithms. NIST will be announcing the final winner in 2012.

## 1.2 Motivation

Performance in hardware has been a key factor in selecting the five finalists. Many hardware implementations of the SHA-3 finalists have been published and most of them are targeted to a design criteria of good throughput. Implementing the hash algorithms in resource constrained environment is also a key factor as they should be implemented on wide variety of platforms like PDAs and smart phones. Cryptographic algorithms may not be the main application but a part of the chip, many other applications have to be included in addition to security for System on Chip devices (SOC). In such cases area is a major constraint, for implementing the hash algorithms. Generally System on chip devices are developed using

Application Specific Integrated Circuits(ASICs). ASICs have low non-recurring engineering costs and also takes long time to market. With the development of low-cost FPGAs which are suitable for battery powered devices and also have low non-recurring engineering costs than ASICs, compact implementation of hash algorithms on FPGAs have become more important.

Basic architecture of any algorithm is designed by straight forward implementation of the function using the specifications described. Compact implementation is quite different from basic implementations, as you need to minimize the size in proportion to speed. Decreasing the size of datapath may increase the clock cycles to process, which results in a low throughput. So consistency has to be maintained between area and also speed for a good throughput over area ratio. Reducing the datapath results in a complex controller which again is a hurdle to optimize. One such hurdle is permutation in hash functions. Hash functions uses permutation, sometimes the permutation may not be in a sequence which makes the controller complex. BLAKE and JH of SHA-3 finalists has complex permutation functions. Optimizing these algorithms for a good throughput to area ratio has been really interesting. Several compact implementations of BLAKE and JH on FPGAs have been reported but they are implemented on different vendors and with no specific area limitation. In order to have a fair comparison among the finalists, it is necessary to have an area constraint and design the algorithms so that they achieve a good throughput to area ratio and still falling under the area budget.

## 1.3 Assumptions and Goals

Only 256-bit variants of BLAKE and JH were implemented as these are the most likely variants to be used in area constrained designs. Designing low-area architectures for hash functions requires memory to store the state, message and also other constants needed for hash computation. So memory/storage is also important in designing compact architectures. Two ways of accumulating the data is by using the embedded resources or the logic resources in FPGAs. Considering all the factors we fixed a design criteria of 400-600 slices with one

Block RAM or 800 slices with no Block RAM, respectively on Xilinx Spartan-3 device. This design criteria was based on the results published on SHA3-Zoo[5] website and also our analysis of the five finalists. Keeping this limitations, we designed our BLAKE and JH to achieve a maximum throughput to area ratio and still falling under the postulated area constraint.

## 1.4 Thesis Organization

Chapter 2 describes the protocol we used for our compact implementations and optimization techniques for designing compact architectures. In subsequent chapters we discuss specifications of BLAKE and different compact architectures of BLAKE, followed by JH specification and compact architecture of JH. The implementation results and comparisons are discussed in chapter 7 along with summary and future work.

# Chapter 2: Methodology

## 2.1 Interface and Protocol

Our hardware interface and I/O protocol (Fig.2.1) is based on the one presented in [6] and revised in [7]. The SHA core contains our implementation and FIFOs are used to feed and collect the data from it. A FIFO is a $w$ bit wide register, which is used for synchronization purpose in hardware. It has two pointers for having a hand shake protocol and this can be easily interfaced to other micro controllers and circuits. The FIFO interface discussed in [6] also suits for compact implementations. Compact implementations have lower databus width than other implementations. Considering this we will be using a databus width $w$ of 16 bits instead of 32 or 64 bits proposed in [6]. The SHA protocol supports two scenarios

1. when the message length is known
2. when the message length is not known

*When the message length is known*, the message is transmitted as a single 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.

*When the message length is not known*, 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’, which states that more message segments are in queue. 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 formula to compute the total number of bits before padding and after padding is given below.

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

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

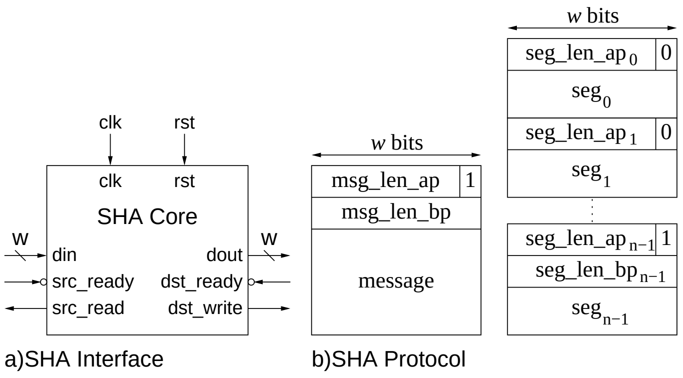


Figure 2.1: Interface and protocol for our SHA cores

## 2.2 Architectural Overview Of Spartan-3 FPGA

Utilization of architectural features provided in FPGAs is crucial for designing compact architectures. Our target device is Spartan-3 FPGA, which is a low-cost and low-power device. In order to have a great compact design we need to interpret the structural features of Spartan-3 FPGAs. Compact architectures of BLAKE and JH are developed making the use of these architectural features.

The spartan-3 FPGA, architecture consists of five fundamental programmable functional elements

1. Configurable Logic Blocks (CLBs)

2. Input/Output Blocks (IOBs)

3. Block RAM

4. Multiplier Blocks

5. Digital Clock Manager (DCM) Blocks

The organization of these elements in a Spartan-3 device is given in Fig 2.2. Among the listed functional elements we will utilize CLBs for implementing synchronous as well as combinational circuits and Block RAM to store the bits required.

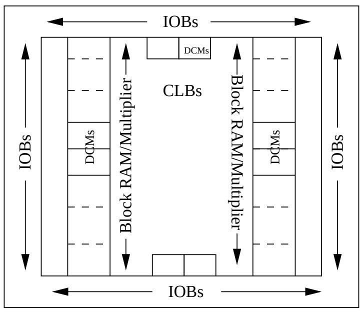


Figure 2.2: Architecture of Spartan-3 FPGA

### 2.2.1 Configurable Logic Blocks (CLBs)

CLBs are used for implementing combinational and synchronous logic circuits. Each CLB in Xilinx Spartan-3 FPGA is comprised of four slices, and each slice has two Look-Up Tables(LUTs) for implementing logic. The LUTs are followed by dedicated storage elements that can be used as registers or latches. The organization of slices in a CLB is given in Fig 2.3. Each CLB is divided into two half's and each half is referred as SLICEL and SLICEM, respectively. The two slices of SLICEM supports logic, arithmetic and ROM functions

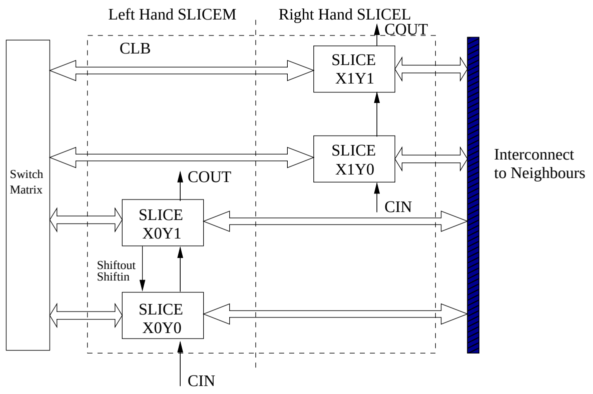


Figure 2.3: Organization of slices in CLB

where SLICEL supports only logic. The resources in SLICEM and SLICEL are show in Fig 2.4 and Fig 2.5 respectively. Each slice in Xilinx Spartan-3 FPGA has the following elements to support the above listed functions.

- Two 4-input LUT function generators F and G
- Two storage elements
- Two wide-function multiplexers
- Carry and arithmetic logic

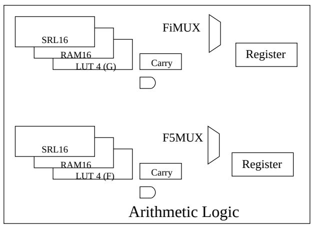


Figure 2.4: SLICEM

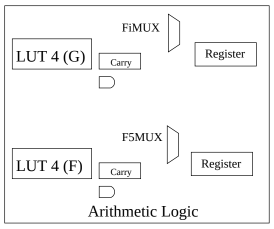


Figure 2.5: SLICEL

### 2.2.2 Look-Up Tables

Look-Up Tables (LUTs) are mainly used for implementing logic functions. In addition, LUTs in each SLICEM can be configured as Distributed RAMs or 16-bit shift registers called SRL16. Each LUT have four logic inputs (A1-A4) and a single output D. Any function with four inputs can be implemented in a single LUT. Functions with more inputs are implemented by cascading LUTs and are connected by using carry and arithmetic logic along with muxes. The LUTs are also directly connected to CLB output or to the storage elements of CLB.

### 2.2.3 Distributed RAM

A Look-Up Table in SLICEM can be configured to store 16-bits of data, and each bit can be addressed by the four inputs of LUT. This configuration of LUT helps us to build Distributed RAMs. The LUTs can be cascaded to realize deeper memories with minimal penalty on timing. Distributed RAM supports the following memory types:

- Single-port RAM with synchronous write and asynchronous read. Synchronous reads are possible using the storage associated with the LUT
- Dual-port RAM which one synchronous write and two asynchronous read ports. As above, synchronous reads are possible. The second read port is an independent read port.

Fig. 2.6 shows the Single-port and Dual-port Distributed RAMs. Each CLB contains four SLICEM LUTs, which can be configured as 64-bits of single port RAM or 32-bits of dual-port RAM. Large amount of data can be stored using RAM16s, which helps us to access the required data independently. RAMs can be initialized with data depending on the design criteria.

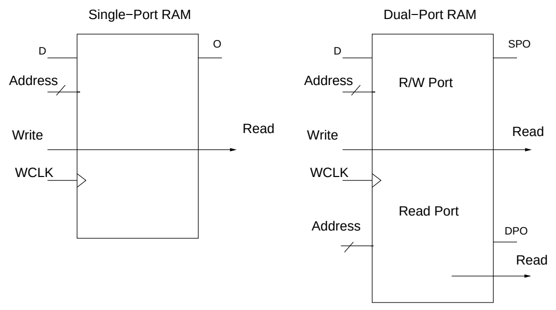


Figure 2.6: Single-port and Dual-port Distributed RAMs

### 2.2.4 Shift Registers

The SLICEM LUT can be configured as a 16-bit shift register, without using the flipflop available in each slice. The F-LUT and G-LUT of SLICEM shown in Fig 2.4, are the basic components for constructing SRL16. The basic structure of SRL16 is given in Fig 2.7. LUTs within SLICEM can be cascaded by connecting MC15, the output of G-LUT to DI, input of F-LUT through $DIFMUX$. SLICEMs can be cascaded by SHIFTIN and SHIFTOUT lines which are routed through $DIGMUX$ to form large shift registers. Each SRL16 provides a shift output MC15 for the last bit in each LUT in addition to addressable access to any bit in the shift register through normal D output. D output is available in SRL primitives, where as MC15/Q15 signal that drives SHIFTOUT are available in SRLC16 primitive. SRLC16 refers to cascade-able shift registers. Each CLB can be configured as a 64-bit shift registers using the four LUTs of SLICEM.

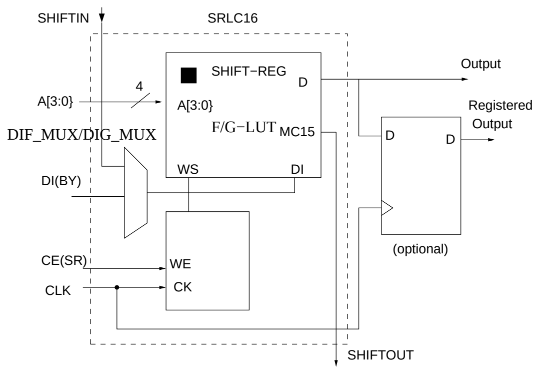


Figure 2.7: Shift Register

### 2.2.5 Block RAM

Block RAMs (BRAMs) are the embedded memory elements on FPGAs. They can be used for storing large amount of data, in place of logic resources which consume area on FPGAs. BRAMs have large memory space but limited number of I/O ports. Each BRAM can be configured as a single-port or dual-port memory with a maximum data width of 64-bits or 32-bits per port, respectively. Each port is associated with an address input. A dual port BRAM has two completely independent access ports. Both ports are interchangeable and supports data read and write operations. The read and write operations of a port are synchronous with its own clock, enable and write enable signals. BRAM limits the number of independent values that can be accessed in one clock cycle because of the limited read ports. Configuration of BRAM as a single-port and dual-port memory is given in Fig 2.8.

The spartan-3 data sheet specifies that, for a BRAM data is written to the address applied in the current clock cycle, but reads out the data from the address of the previous clock cycle.

Hence if you want to write the data to the same location i.e. $Mem[i] = Mem[i] + k$, where each element is 64-bit word, it takes two clock cycles. So, if an address shift is not acceptable you need to have dedicated write clock cycles.

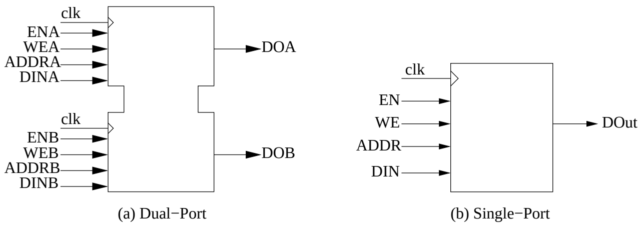


Figure 2.8: Single and Dual Port Block RAMs

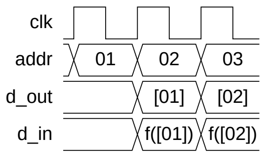


Figure 2.9: Timing diagram of BRAM

When address shift is acceptable i.e. $Mem[i+i] = Mem[i] + k$, you don't need to have dedicated write clock cycles.

## 2.3 Optimization Techniques

### 2.3.1 Datapath Optimization

Efficient compact architectures can be designed by optimizing the datapath of the algorithm. Datapath optimization can be attained by following three main steps. They are

- Folding
- Pipelining
- Rescheduling

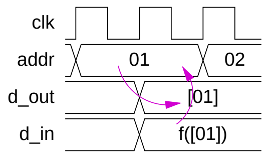


Figure 2.10: Dedicated write clock cycles for BRAM

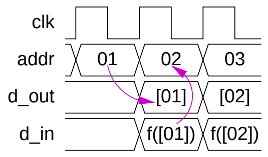


Figure 2.11: Address shift in BRAM

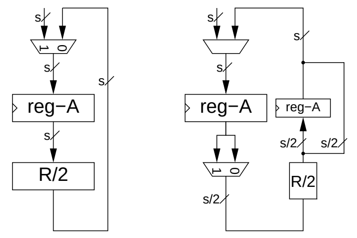
**Folding**

Scalability of an algorithm is important for developing compact architectures. Depending on scalability, an algorithm can be folded either horizontally or vertically. Folding an algorithm reduces the datapath width or the number of functions that are processed in one clock cycle. The method of horizontal folding and vertical folding is given in fig 2.12(a) and fig 2.12(b) respectively.

◇ **Horizontal Folding**

Depending on parallelism a function can be folded across the horizontal axis. Folding a function horizontally either reduces the critical path or the number of functions computed in one clock cycle. This increases the clock cycle count depending on the number of times the function is folded.

For example, in fig 2.12(a) a function $R$ is folded by half. So in one clock cycle only half of $R$, which is $R/2$ is computed and stored back in the register. The remaining half is executed in the next clock cycle, making two clock cycles to execute a single function. By



a)Horizontal Folding b)Vertical Folding

Figure 2.12: Folding of Algorithms

this we reduce the area by almost half and also the critical path is cut down resulting in doubling the throughput.

◇ **Vertical Folding**

Some functions have analogy across the vertical axis. Folding such functions across the vertical axis reduces the number of bits that are processed in one clock cycle. The reduce in the datapath width increases the clock cycles which is dependent on the number of times the datapath is folded.

For example, in fig 2.12(b) the function $R$ is folded across vertical axis. $R/2$ function processes half of the data and stores in a register, the remaining half is processed in the next clock cycle. The processed two halves are concatenated to be stored back in the state register. As you have free registers after each LUT, storing the data after one clock cycle will not increase the area.

### Pipelining

Throughput of a design can be increased by implementing pipeline technique. Pipelining is a method in which multiple instructions are overlapped and executed. It consists of breaking long combinational path by introducing/using registers, this divides the instructions into *stages*. Each stage completes or processes a set of instructions in parallel. All the stages are connected to form a pipe - instructions enter at one end, progress through the stages and exit at the other end. Pipelining reduces the critical path by breaking the combinational path, thus allowing to attain high clock frequencies.

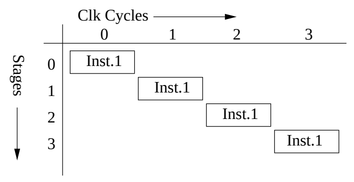


Figure 2.13: Unpipelined Machine

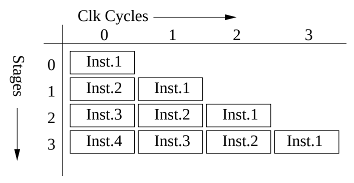


Figure 2.14: Pipelined Machine

In some functions, the output of one pipeline stage becomes the input of the other pipeline stage. This rises the concept of data forwarding, which allows flow of information from one pipeline stage to the other. The flow of information may be to the next pipeline stage or to a stage which is not subsequent. In such cases data has to be stored and forwarded for the required pipeline stage. This technique is called as *quasi-pipelining*. In this technique two or more pipeline stages allow interleaving computations belonging to any two functions.

### Rescheduling

Pipelining gives the concept of data forwarding and also data dependency. The data which is needed by other functions to be computed may be in the pipeline and this results in pipeline stalls. By rescheduling the functions we can avoid data dependency and also avoid pipeline stalls. The rescheduling of functions should be designed so that the overall functionality of the algorithm is not altered.

In addition to the steps mentioned, choosing the appropriate feature for implementing a specific component in the design and re-use of hardware results in an effective compact design. For choosing appropriate feature we need to utilize the structural features of the targeted FPGA. Storage is an important concern for compact architectures. Use of BRAM for storing will reduce the area, but this restricts the data access and further increases clock cycles. By using DRAMs for data which needs to be accessed frequently will reduce the clock cycles with a slight increase in area.

### 2.3.2 Controller Optimization

Finite State Machines (FSM) are used along with ROMs for realizing the control logic of complex systems. This led to ROM-based FSM implementation which proved to be efficient [8], [9], [10]. Our control logic hash one main FSM, up to 8-states and additional counters to count the clock cycle per state. The control signals are minimized by by eliminating some signals which has the same pattern as others. Some signals can be generated using logic and this reduces the area for storing the bits. The other important thing in designing controllers is the addressing of BRAM. BLAKE has permutation which can't be minimized because of the irregularities. Using ROMs to generate the addressing for both ports consumes significant area, in this case we need to optimize other signals to obtain minimum area.

## 2.4 Tools and Benchmarking

All our designs are targeted to Spartan-3 FPGA, but it is interesting to see the performance of designs on other low-cost devices developed by Altera. So we implemented our designs on Altera Cyclone-II, new low-cost spartan-6 devices and on high speed devices like Virtex-5 and Virtex-6. All designs were implemented using Xilinx ISE 13.1 Web Pack and Altera Quartus II v10.0 Web Edition. All results were generated using benchmarking tool ATHENa

(Automated Tool for Hardware EvaluatioN) [11].

## 2.5 Performance Metrics

The performance of the implementation depends on the number of clock cycles it takes to hash $N$ message blocks. This can be computed from the number of clock cycles that each function requires to perform. The functions being:

| | | | |
|---|---|---|---|
| $i$ | 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 | | |

As a result, the formula for the number of clock cycles for hashing $N$-blocks of message is:

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

This formula can be simplified, by dividing the initial steps before the processing can begin $st = i + h + (l1 - l)$, loading and processing of one message block $l + p$ and finally the hashing out and finalization $end = z + o$. This results, the number of clock cycles to hash $N$-blocks of data:

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

Throughput is the key factor for comparing the performance of algorithms. 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 $N$ to be hashed, the number of clock cycles to process one message block, block size $b$ of the algorithm and the delay/clock period $T$. So, throughput of a hash function can be given as:

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

In embedded applications, messages can be short, so it is important to calculate the throughput of short messages. For this we use and empty message which after padding has one block of message. This makes the value of $N = 1$ in the above equation for computing the throughput. For long messages, the *st* and *end* takes place one time and so it makes a negligible effect on throughput. This leads us to the simplified equation for throughput, which is

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

# Chapter 3: BLAKE

## 3.1 Introduction

BLAKE [12] is a hash function developed by Jean-Phillippe Aumasson and his group from Switzerland. It was submitted to Secure Hash Algorithm-3 (SHA-3) contest that was organized by National Institute of Standards and Technology (NIST). The heritage of BLAKE is threefold

1. BLAKE's Iteration mode is HAIFA
2. BLAKE's internal structure is local wide-pipe
3. BLAKE's compression algorithm is a modification of Bernstein's stream cipher ChaCha.

BLAKE is a family of four hash functions, which produce message digests of 224,256,384 and 512 bits with the same parameter sizes of SHA-2[1]. The second round submission of these four hash functions were referred to as BLAKE-28, BLAKE-32, BLAKE-48 and BLAKE-64. For the third round they were renamed as BLAKE-224, BLAKE-256, BLAKE-384 and BLAKE-512. The difference between Round-2 and Round-3 is number of rounds. BLAKE's 32-bit version is named as BLAKE-256 and the 64-bit version is reffered as BLAKE-512 which produces 256-bit and 512-bit hash outputs, respectively. Table.1 gives the properties of the BLAKE hash functions.

BLAKE hash functions follows HAIFA(HAsh Iterative FrAmework) iteration mode[13]. The HAIFA iteration mode and the compression function depends on the Salt and the number of message bits hashed with the help of a counter. The compression function is developed from LAKE[14], which hash a large internal state which is initialized using the initial hash values, salt and the counter. The inner state is updated by rounds which is

Table 3.1: Properties of BLAKE hash functions

| Algorithm | Word | Message | Block | Digest | Salt | Round-2 | Round-3 |
|---|---|---|---|---|---|---|---|
| BLAKE-224 | 32 | $< 2^{64}$ | 512 | 224 | 128 | 10 | 14 |
| BLAKE-256 | 32 | $< 2^{64}$ | 512 | 256 | 128 | 10 | 14 |
| BLAKE-384 | 64 | $< 2^{128}$ | 1024 | 384 | 256 | 14 | 16 |
| BLAKE-512 | 64 | $< 2^{128}$ | 1024 | 512 | 256 | 14 | 16 |

message dependent and it is finally compressed to produce the next chaining hash value. This strategy is called local wide-pipe[15].

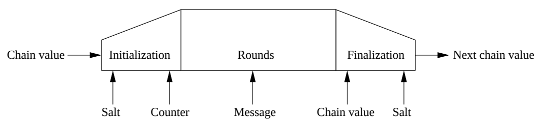


Figure 3.1: Widepipe of BLAKE

The inner state is represented as 4x4 matrix and the compression function, first updates the all the four columns independent and four disjoint diagonals after that. For each update two message blocks are used depending on the round. After the rounds, the state is reduced using the initial hash value and salt.

### 3.1.1 Notations

The following notations are used in BLAKE description. If P is a bit string, it is viewed as a string of words and $P_i$ denotes $i^{th}$ word component.For a message M, $M^i$ denotes its $i^{th}$ 16-word block. So $M^i_j$ is the $j^{th}$ word of the $i^{th}$ block of message M. For N-block message, M is decomposed as M $= M_0, \cdots, M_{N-1}$ and each block $M_0$ is composed of words.

The initial hash value is represented as $IV$ and is composed of eight 32-bit words $IV_0, \cdots, IV_7$. The intermediate hash values in the iterated hash are called chain values and the hash output of the final message block of message is the final *hash value* or *hash.*

$$\begin{aligned} P &= P_0 \parallel P_1 \parallel ......P_{i-1} \parallel P_i \\ M &= M_0 \parallel M_1 \parallel ......M_{i-1} \parallel M_i \\ M_0 &= m_0 \parallel m_1 \parallel ......m_14 \parallel m_15 \\ IV &= IV_0 \parallel IV_1 \parallel ......IV_6 \parallel IV_7 \end{aligned} \tag{3.1}$$

The mathematical operations used in BLAKE are denoted as follows

Table 3.2: Operation & symbols used in BLAKE

| Symbol | Meaning |
|---:|---:|
| $\leftarrow$ | Variable assignment |
| $\boxplus$ | Addition modulo $2^{32}$ |
| $\oplus$ | Boolean Exclusive OR(XOR) |
| <<< k | rotating of k bits towards more significant bits |
| >>> k | rotating of k bits towards less significant bits |

## 3.2 BLAKE-256

BLAKE-256 operates on 32-bit words of message and returns a 32-byte hash value. This section describes BLAKE-256, starting from the constants, compression function and finally to the iteration mode.

### 3.2.1 Constants

BLAKE-256 uses the same initial hash value as SHA-256.

$$\begin{aligned} IV_0 &= \text{6A09E667} & IV_1 &= \text{BB67AE85} \\ IV_2 &= \text{3C6EF372} & IV_3 &= \text{A54FF53A} \\ IV_4 &= \text{510E527F} & IV_5 &= \text{9B05688C} \\ IV_6 &= \text{1F83D9AB} & IV_7 &= \text{5BE0CD19} \end{aligned}$$

BLAKE-256 requires 16 constant words.

$$\begin{aligned} c_0 &= \text{243F6A88} & c_1 &= \text{85A308D3} \\ c_2 &= \text{13198A2E} & c_3 &= \text{03707344} \\ c_4 &= \text{A4093822} & c_5 &= \text{299F31D0} \\ c_6 &= \text{082EFA98} & c_7 &= \text{EC4E6C89} \\ c_8 &= \text{452821E6} & c_9 &= \text{38D01377} \\ c_{10} &= \text{BE5466CF} & c_{11} &= \text{34E90C6C} \\ c_{12} &= \text{C0AC29B7} & c_{13} &= \text{C97C50DD} \\ c_{14} &= \text{3F84D5B5} & c_{15} &= \text{B5470917} \end{aligned}$$

### 3.2.2 Compression function

The compression function is comprised of three functions namely:

- Initialization
- Round function
- Finalization

The compression function takes 30 words consisting of message, initial hash value, salt and counter to produce a 32 byte chaining hash value. Compression function is represented as h' = compress(h,m,s,t)

- chain hash h = $h_0, \cdots, h_7$
- message block m = $m_0, \cdots, m_{15}$
- salt s = $s_0$,$s_1$,$s_2$,$s_3$
- counter t = $t_0$,$t_1$

**Initialization**

The initialization step is used to initiate an inner state of 16 words represented as $v_0, v_1, \cdots, v_{15}$. The state is represented as a 4x4-matrix and is generated using the initial hash, salt, constants and counter.

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} \leftarrow \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{bmatrix}$$

**Round function**

The round function operates on the inner state $V$ and loops for 14 times to produce a new $V$ after each round. It consists of eight G-functions which operates on four words of inner state. The first four G-function updates the four columns of the matrix independently and the next four G-function updates the independent diagonals of the matrix. The G-functions are represented as:

$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$

For a round r, G-function is denoted as $G_i(a,b,c,d)$ and it is defined as follows:

$$\begin{aligned}
a &\leftarrow a \boxplus b \\
a &\leftarrow a \boxplus (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}) \\
d &\leftarrow (d \oplus a) >>> 16 \\
c &\leftarrow c \boxplus d \\
b &\leftarrow (b \oplus c) >>> 12 \\
a &\leftarrow a \boxplus b \\
a &\leftarrow a \boxplus (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}) \\
d &\leftarrow (d \oplus a) >>> 8 \\
c &\leftarrow c \boxplus d \\
b &\leftarrow (b \oplus c) >>> 7
\end{aligned}$$

The first four G-functions $G_0, \cdots, G_3$ can be computed in parallel as they operate on independent columns of the matrix. Similarly $G_4, \cdots, G_7$ operate on independent diagonals of matrix and can be computed in parallel. The G-functions $G_4, \cdots, G_7$ depends on the output of $G_0, \cdots, G_3$.Each G-function uses two permuted words of message and constants. The permutation used for each round is determined by using $\sigma_{r\ mod\ 10}$ operation. Where r denotes the round number. The permutation values for BLAKE-256 are given in Table 3.3.

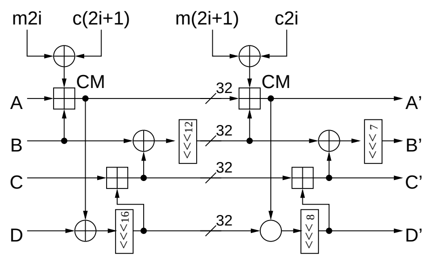


Figure 3.2: $G_i$ Function

Table 3.3: Permutations of Message and Constants used by BLAKE

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\sigma_0$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| $\sigma_1$ | 14 | 10 | 4 | 8 | 9 | 15 | 13 | 6 | 1 | 12 | 0 | 2 | 11 | 7 | 5 | 3 |
| $\sigma_2$ | 11 | 8 | 12 | 0 | 5 | 2 | 15 | 13 | 10 | 14 | 3 | 6 | 7 | 1 | 9 | 4 |
| $\sigma_3$ | 7 | 9 | 3 | 1 | 13 | 12 | 11 | 14 | 2 | 6 | 5 | 10 | 4 | 0 | 15 | 8 |
| $\sigma_4$ | 9 | 0 | 5 | 7 | 2 | 4 | 10 | 15 | 14 | 1 | 11 | 12 | 6 | 8 | 3 | 13 |
| $\sigma_5$ | 2 | 12 | 6 | 10 | 0 | 11 | 8 | 3 | 4 | 13 | 7 | 5 | 15 | 14 | 1 | 9 |
| $\sigma_6$ | 12 | 5 | 1 | 15 | 14 | 13 | 4 | 10 | 0 | 7 | 6 | 3 | 9 | 2 | 8 | 11 |
| $\sigma_7$ | 13 | 11 | 7 | 14 | 12 | 1 | 3 | 9 | 5 | 0 | 15 | 4 | 8 | 6 | 2 | 10 |
| $\sigma_8$ | 6 | 15 | 14 | 9 | 11 | 3 | 0 | 8 | 12 | 2 | 13 | 7 | 1 | 4 | 10 | 5 |
| $\sigma_9$ | 10 | 2 | 8 | 4 | 7 | 6 | 1 | 5 | 15 | 11 | 9 | 14 | 3 | 12 | 13 | 0 |

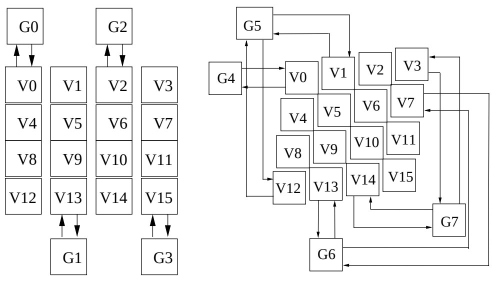


Figure 3.3: Column and diagonals step of Round function

## Finalization

After the round function, a new chain value $h'_0, \cdots, h'_7$ is generated using $v_0, v_1, \cdots, v_{15}$, input chain value $h_0, \cdots, h_7$ and salt $s_0, s_1, s_2, s_3$.

$$\begin{aligned}
h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{aligned}$$

### 3.2.3 Iterated hash

Each message is split into 16 word message block $m_0, \cdots, m_{N-1}$. The iterated hash of BLAKE-256 is given by

```
$h^0 \leftarrow IV$
$for\ i = 0, \cdots, N-1$
    $h^{i+1} \leftarrow compress(h^i, m^i, s, l^i)$
$return\ h^N$
```

### 3.2.4 Block diagram

The block diagram of BLAKE-256 is shown in Fig.3.4.

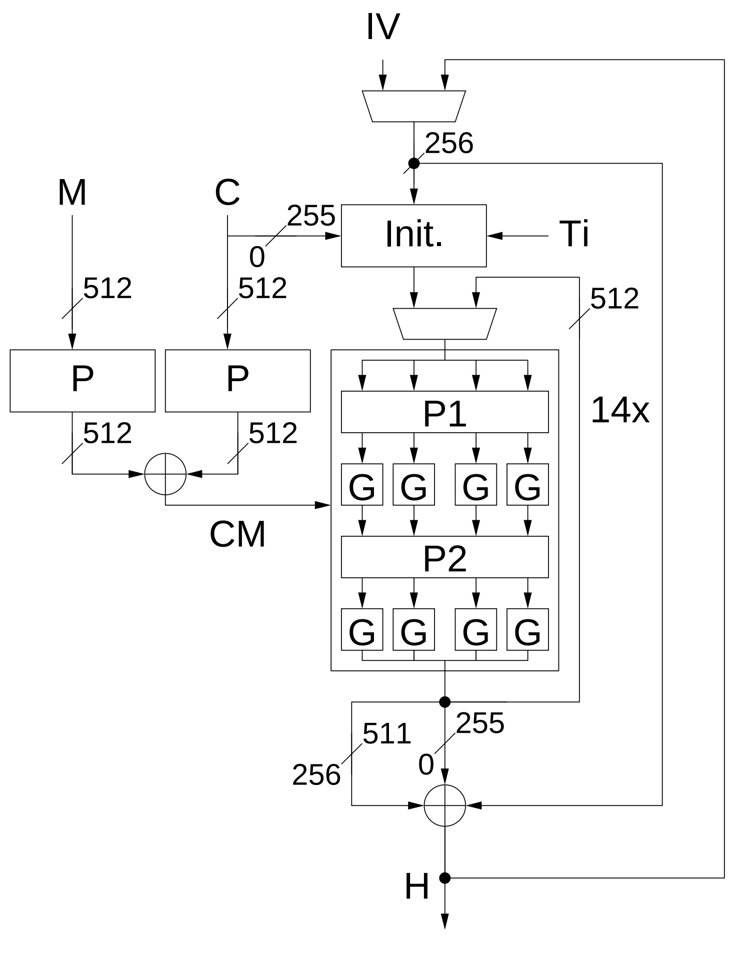


Figure 3.4: Block Diagram of BLAKE-256

# Chapter 4: Compact Implementations of BLAKE

## 4.1 Dedicated Write Design-I

### 4.1.1 Datapath

Compact architecture of BLAKE-32 is designed by utilizing the optimization techniques discussed in chapter 2. The storage element Block RAM is used to store message, constants, initial & chaining hash values, salt, internal state $V$ and counter. The internal state $V$, which is stored in Block RAM should be accessed in each round, in fact for each G-Function. So, in order to have the state in the same address location we have dedicated write clock cycles. This helps us to optimize the controller as we can achieve congruence across the rounds.

The main function of BLAKE is the round function which consists of eight G-Functions. This compact implementation has a 32-bit datapath and processes two G-Functions in pipeline. Pipelining is implemented by using the free registers after the combinational logic. As we implement a 32-bit datapath, the block RAM is configured as dual-port which can read and write 32-bits of data for each port. The initialization and finalization functions are simple $XOR$ operations on 32-bit words. This $XOR$ operations of 32-bit words, are performed using a 32-bit $XOR$ gate out side the pipelined round function. The initialization and finalization values are stored back into the block RAM and this takes 16 and 48 clock cycles, respectively.

For processing two G-Functions, we need to read eight words of internal state $V$ and four words each of message and constants. The four words of message and constants for each round are permuted. In order to provide permuted messages and constants for the respective rounds we generate addresses from the controller. The controller addresses the correct message and constant word that is required for the G-Function in each round. The processed eight words of state has to be written back to the same locations from which we read them. For writing back to the same locations of block RAM we use dedicated write clock cycles.

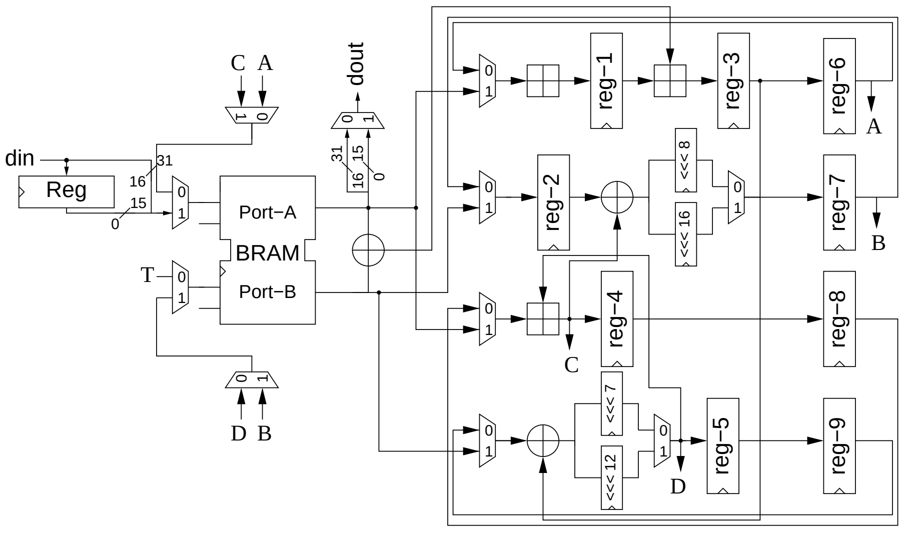


Figure 4.1: Datapath of Dedicated Write Design-I

Let's assume $G_i$ and $G_{i+1}$ are two G-functions that are to be processed in pipeline. As we need to read 16 words of data for both the G-Functions, each G-Function takes two clock cycles for reading the state and two clock cycles for constants and message. The reads are scheduled so that half of $G_i$ can be processed and stored in the reserve registers, followed by $G_{i+1}$. The second half of $G_i$ is computed by pipelining it with the $G_{i+1}$. It takes eight clock cycles to read out the data from dual-port block RAM and four clock cycles to write back eight words of the internal state $V$. The scheduling of two pipelined G-functions is given in Fig.4.2. Ports A and B denotes the reading and writing of the words for the two G-functions. Level represents the instructions performed in each stage.

It takes 12 clock cycles to compute two G-Functions and write them back into the block RAM. So, one round of BLAKE takes 48 clock cycles to process. The final or the chaining hash value is stored back into the block RAM after finalization.

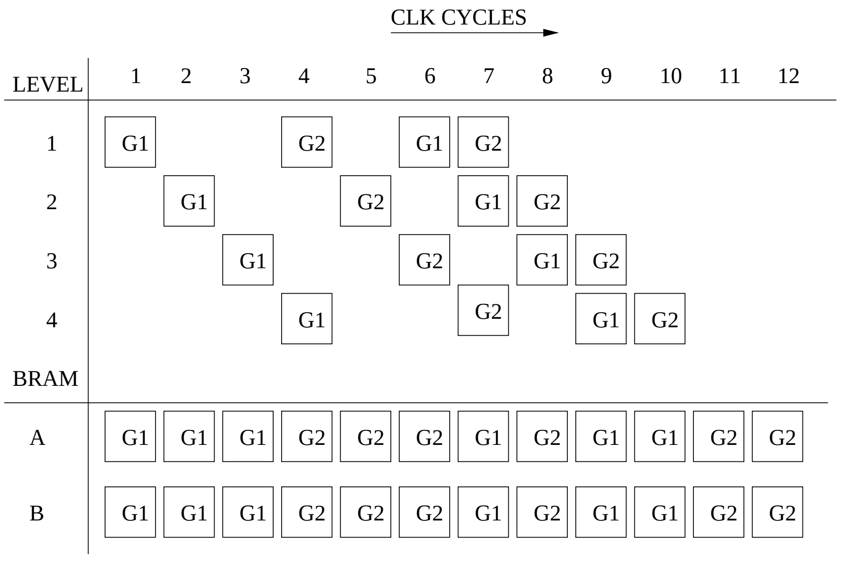


Figure 4.2: Pipeline of Two G-Functions

### 4.1.2 Control Unit

The control unit is developed using counters and Distributed ROMs which store the control words and addresses for block RAM. The block RAM is divided into segments, each consisting of 16 words. It uses an eight bit address to read/write data from/to block RAM. Four MSB bits are used for handling the segments and the four LSB bits are used for accessing the words. Loading of message into desired segment of block RAM is addressed with the help of a four-bit counter. For the initialization and finalization the control words and the addresses are stored in separate ROMs. The round function hash congruence among other rounds and we develop control signals and address bits for one round which can be used over for other rounds. Addressing permuted message and constants for the round function are stored in a $160x4-bit$ ROM. The selection between the message,constants and the internal state segments of block RAM is is controlled with selection bits stored in a ROM. The permutation of message words and constants have no pattern, no method can be employed to optimize this ROM.

The design utilizes the optimization techniques but still it has a low throughput, and also has large controller area. This is because of the long critical path though we use pipelining and the permutation function which can't be optimized. In addition to above mentioned, accessing of the internal state and the message from block RAM is a contention as we can either read/write only two words per clock cycle. It increases the clock cycle count for processing one message block which pulls down the overall throughput to area ratio.

## 4.2 Dedicated Write Design-II

### 4.2.1 Datapath

Previous design implements pipelined architecture but still has a long critical path which reduces the throughput. We can reduce the critical path by rescheduling the stages of pipeline. This increases the clock cycles but may increase the overall throughput because of the reduced critical path.

In order to cut down the critical path, the design is modified as shown in Fig.4.3. The modification in design results in increase of one clock cycle for computing the $G_{i+1}$ function of the pipelined two G-Functions. This is because of the bubble inserted after computing $G_i$ function which helps to process the values required for computing $G_{i+1}$. The initialization and finalization functions are still the same as the previous design and take the same amount of clock cycles.

### 4.2.2 Control Unit

The control unit is same as the previous design, which uses counters and distributed ROMs. The control unit is built by making slight changes to the controller designed above. In the earlier design it takes 48 clock cycles for one round function. We use a six bit counter to generate the signals from the ROM which stores control words for round function. This implementation takes 52 clock cycles, and by adding additional signals to the ROM does

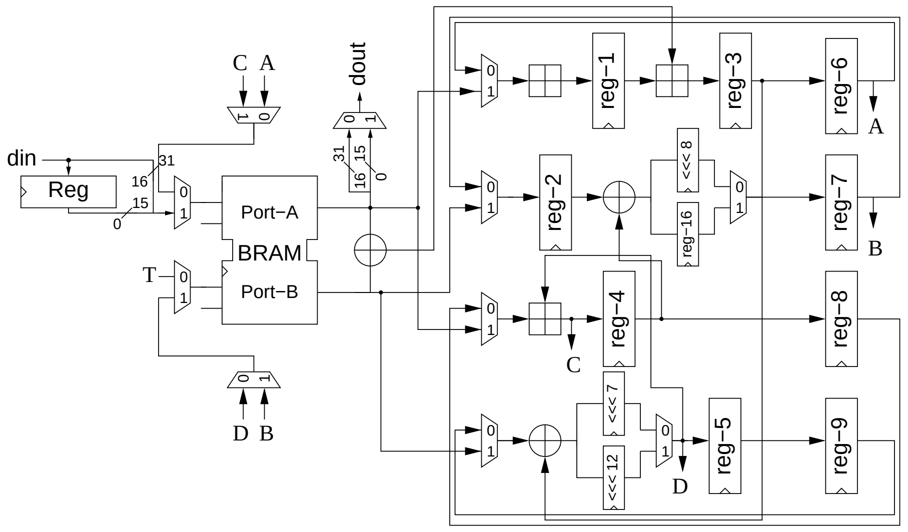


Figure 4.3: Datapath of Dedicated Write Design-II

n't increase the area as a six bit addressable ROM can hold up to 64 control words.

We employed rescheduling to further increase the throughput of the design. When increasing the clock cycles to decrease the critical path, consistency has to be maintained so that overall throughput can raise. When this is not maintained it further reduces the throughput. The reason for more number of clock cycles is because of the block RAM contention which was previously discussed. Avoiding the block RAM contention will give us a better throughput design and also reduces the area as the clock cycles count goes down.

## 4.3 Unpipelined Half G-Function

### 4.3.1 Datapath

#### Block RAM Architecture

Folding an algorithm is the main methodology of designing compact architectures. For one round, BLAKE processes eight identical G-Functions, which operates on a 512-bit state. These eight identical G-Functions can be folded and a single G-Function can be implemented for compact architecture. But, each G-Function is symmetrical along the vertical axis, so a G-Function can be folded vertically and half G-Function is implemented.

Our former design's, store the internal state in the block RAM which arises the ease of accessing in other words, contention to access the internal state. In order to avoid block RAM contention, we move the state out of the block RAM. The other storage elements in an FPGA are registers and Distributed RAM's. Storing a 512-bit state in 16 word registers consumes a large area, so we use distributed RAMs. But preserving all the 16 words in a single distributed RAM still gives a contention problem. To over come this we use four $4x32$-bit Distributed RAM's to store the 512 bit internal V-state. This gives the scope to access independent columns of the internal state matrix.

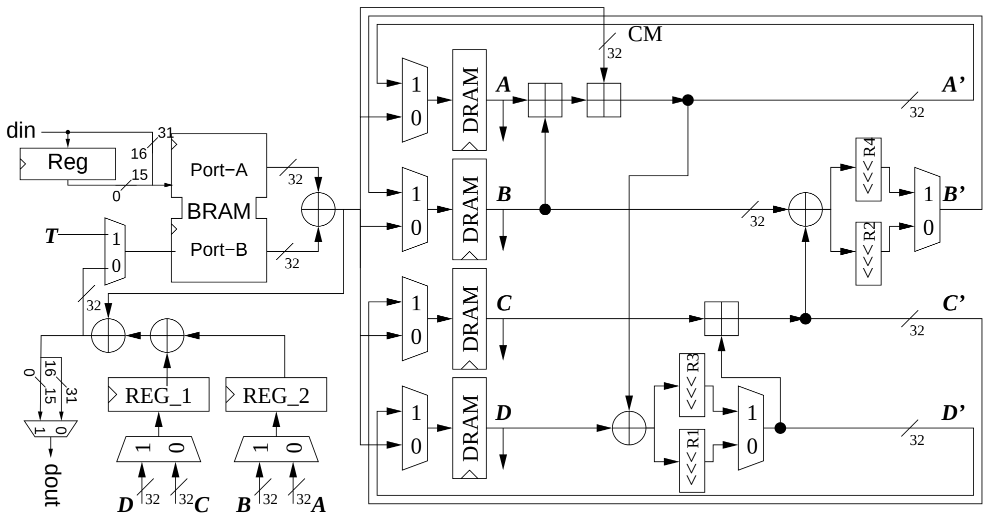


Figure 4.4: Datapath of Unpipelined Half G-Function

The datapath, implements a half G-Function for processing the round function of BLAKE. Block RAM stores the constants, message, salt and counter value. The initialization and finalization functions are implemented using two 32-bit $XOR$ gates. For the initialization, the initial/chaining hash values, constants and counter values are $XORED$ and stored in the four Distributed RAMs. The finalization step produces the chaining hash values and performs $XOR$ operation on the internal state and chaining hash value. The computed chaining/final hash value is stored into block RAM, which helps to process the subsequent message blocks.

The half G-Function implemented for processing the round function, takes an input of four 32-bit words from the four distributed RAM's and one word of message and constant from the block RAM. Each round performs eight G-Functions, as we are implementing half a G-Function it takes two clock cycles to process one G-Function. Every half G-Function for each round takes one word of permuted message and constant. The permuted message and constants for the round function are read consecutively from the block RAM. As we are using a single port asynchronous distributed RAM's, the internal state words are stored back into the same locations and this simplifies the control logic.

### Distributed RAM Architecture

Replacing the block RAM with logic resources helps us to compare the performances between logic versus embedded resources. In some cases converting the block RAM into logic resources, may help in improving the datapath design. Block RAM has data contention problems, i.e., only two words of data can be accessed which we can over come by using distributed RAM's. For BLAKE, the internal state is moved into four independent distributed RAM's for the block RAM design. For the Distributed RAM design, block RAM has to be replaced with distributed RAM. We configured the block RAM as dual-port read/write, but in spartan-3 we can't design a dual-port distributed RAM. So we replace the dual-port block RAM with two single-port distributed RAM's. As a block RAM is synchronous to clock, we can either design synchronous distributed RAM or store the output of the RAM into a register to make it synchronous.

We use two $32x32-bit$ single-port distributed RAM's to store the message, constants, chaining hash values, salt and counter. The message and constants are to be stored in different distributed RAM's so that message and constant words can be $XORED$ for the round function. Apart from changing the block RAM into distributed RAM, we implemented the same half G-Function design as the block RAM architecture.

### 4.3.2 Control Unit

ROM based FSM controller design is employed to develop the control unit. It has a main FSM, which selects the states and controls the state transfers. We use distributed ROMs to store the control words and the ROM outputs are selected depending on the state. Similar to our former design, block RAM is divided into sections and as we move the state out of block RAM we use only two bits to access the sections. The message and constants for the round function are permuted and no optimization technique can be employed to reduce the addressing. As the total number of clock cycles to process one message block is less than 256, a 256 state ROM suits the purpose. So we use two $256x4 - bit$ ROM's to store the addresses of each port. Distributed RAMs has to be addressed for initialization, round and finalization states. Addresses for the RAMs are stored in different ROMs depending on the state and each ROM is selected according to the state. For the round function, the internal state is stored back into the same address location of distributed RAMs. This helps us to use a single ROM for storing the addresses and this can be used for all the rounds.

Shifting the internal state out of the block RAM, increases the ease of access and this reduces the clock cycles count. As we cut down the clock cycle count by more than half the throughput increases and also the area decreases because of the minimum control logic. The ROMs for addressing the block RAM consumes almost 70% of the controller and this is the main set back in the control unit. In addition to the huge ROMs, critical path is large which reduces the frequency of the design. The critcal path is all along the half G-Function and this reduces the throughput. Cutting the critical path with the use of free registers may increase the clock cycle count and further increases the throughput.

## 4.4 Pipelined Half G-Function

### 4.4.1 Datapath

#### Block RAM Architecture

Our former design implements a half G-Function, which is achieved by folding the round function. The design results in good throughput instead of the large critical path. The throughput of the design can be further increased by reducing the critical path. Pipelining is employed to cut down the critical path and this is achieved by using free registers in the half G-Function. One register usage delays the data transfer to the next stage by one clock cycle. So using registers increases the clock cycles count.

This pipelined compact implementation of BLAKE implements the initialization and finalization functions similar to our previous designs. The pipeline architecture we implement in a half G-Function is a quasi pipelined architecture which is described in chapter.2. Using pipeline registers in half G-Function delays the data transfer to the next stage, for this we need to add stalls and we need to schedule the stages of pipeline. In this architecture we use stalls, so that data can be written back into distributed RAM's and schedule the stages to complete one round function with a minimal increase in clock cycles.

The scheduling of one round or eight G-Functions is given in fig 4.6. The quasi pipelined architecture takes four clock cycles to process four half G-functions and one extra clock cycle (stall) to write back the values into DRAM. So eight G-Functions takes 20 clock cycles to process and one extra clock cycle for each round to write the extra word into the DRAM. This makes 21 clock cycles for each round. The message and constants for the round are read out from the BRAM at the required clock cycles.

#### Distributed RAM Architecture

Similar to our former distributed RAM design, this architecture is developed by replacing block RAM with two single-port distributed RAM's. We use registers after each single-port RAM, in order to have synchronous read/write operation with the clock like the block RAM

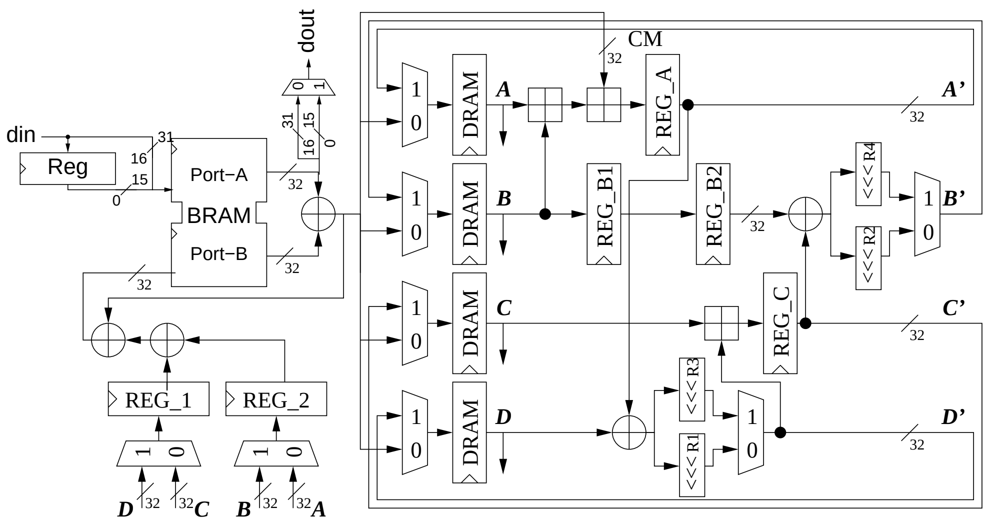
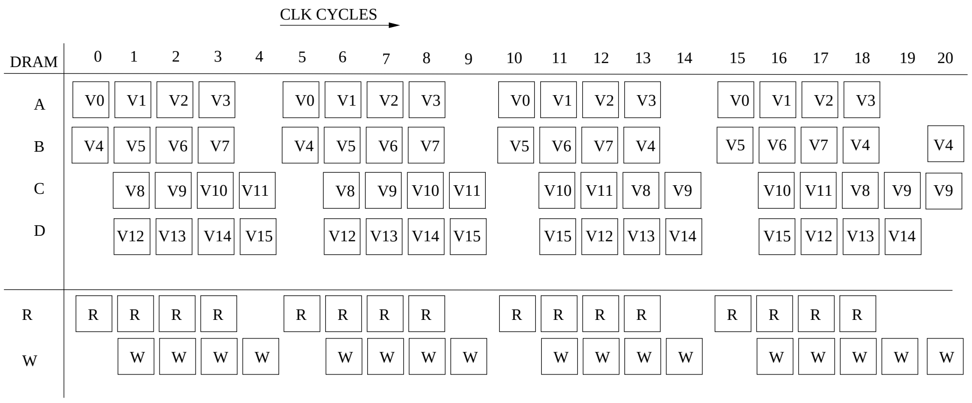


Figure 4.5: Datapath of Pipelined Half G-Function



| DRAM | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | V0 | V1 | V2 | V3 | | V0 | V1 | V2 | V3 | | V0 | V1 | V2 | V3 | | V0 | V1 | V2 | V3 | | |
| B | V4 | V5 | V6 | V7 | | V4 | V5 | V6 | V7 | | V5 | V6 | V7 | V4 | | V5 | V6 | V7 | V4 | | V4 |
| C | | V8 | V9 | V10 | V11 | | V8 | V9 | V10 | V11 | | V10 | V11 | V8 | V9 | | V10 | V11 | V8 | V9 | V9 |
| D | | V12 | V13 | V14 | V15 | | V12 | V13 | V14 | V15 | | V15 | V12 | V13 | V14 | | V15 | V12 | V13 | V14 | |
| R | R | R | R | R | | R | R | R | R | | R | R | R | R | | R | R | R | R | | |
| W | | W | W | W | W | | W | W | W | W | | W | W | W | W | | W | W | W | W | W |

Figure 4.6: Pipeline of Round Function

architecture. As we have same architecture, controller designed for the block RAM version serves the purpose for the distributed RAM with a simple tweak at the addressing of block RAM.

### 4.4.2 Control Unit

The control unit is build using ROM based FSM methodology. We use a main FSM to control the flow of the state and ROM's to store the control words. The controller designed for this architecture is used for both BLAKE-32 and BLAKE-256. The difference between the two algorithms is the number of rounds. BLAKE-32 has ten rounds which process eight G-Functions and BLAKE-256 consists of fourteen rounds. For the round function, we use a distributed ROM to store the control words for the total 21 clock cycles. The control word consists of the distributed RAM addresses and also write enable signals for the registers in half G-Function. The addresses for the dual-port block RAM are stored in distributed ROMs. The extra four rounds of BLAKE-256 uses the same permuted values of message and constants as that of the initial four rounds. So a simple $if$ condition helps to loop back and perform the additional four rounds before stepping to the finalization step. It takes more than 256 clock cycles to process one message block for BLAKE-256. So we accommodate enable signals to disable the address ROM during pipeline stalls. This helps us to use the same $256x4 - bit$ ROM for addressing the block RAM.

Pipelined architecture reduced the critical path and with scheduling we are able to implement a single round function in 21 clock cycles, which is just five clock cycles more than our previous design. Though we increase our throughput, the clock cycles increases by 50 if we consider the whole BLAKE-32 algorithm. As we increase the number of rounds to fourteen in BLAKE-256 it still reduces the throughput when compared to our BLAKE-32 design. We can employ rescheduling to reduce the clock cycles and still sticking to the pipelined architecture.

## 4.5 Rescheduled Pipelined Half G-Function

### 4.5.1 Datapath

#### Block RAM Architecture

Compact and efficient design of BLAKE is achieved by a implementing a complete folded architecture of the G-function. In addition to the folded architecture we used pipeline registers to reduce the time delay. With the use of pipeline registers we increased the clock cycles count but we still maintain a good throughput. Rescheduling of G-Functions can reduce the clock cycles needed for each round and still provide the functionality of the algorithm.

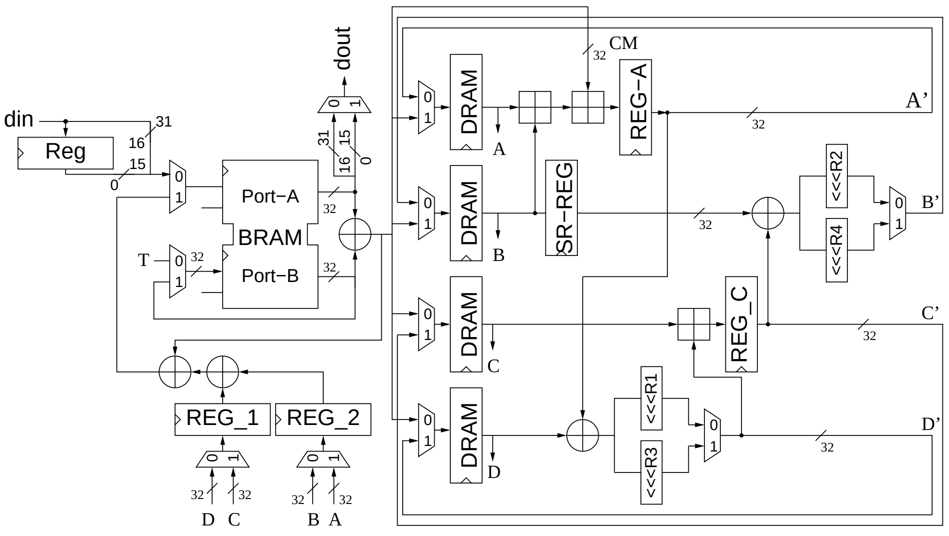


Figure 4.7: Datapath of Rescheduled Pipelined Half G-Function

The main function of BLAKE is the round function which implements eight G-Functions. The four G-Functions $G_4, \cdots, G_8$ depend on the output of $G_0, \cdots, G_3$. Using pipeline registers will raise the use of stall, for getting the data available. To reduce the pipeline stalls and limit the number of clock cycles we reschedule G-functions. A half G-Function with quasi pipeline registers is implemented and we replace two consecutive pipeline registers with a two depth 32-bit shift register(SRL16) which reduces the area. Executing the eight G-Functions in the order of $G_0, G_1, G_3, G_2, G_7, G_6, G_4, G_5$ will eliminate pipeline stalls. By rescheduling and eliminating the pipeline stalls we implement one round in sixteen clock cycles which is same amount of clock cycles as that of the unpipelined half G-Function implementation. The scheduling of G-Functions across one round is given in Fig 4.8. After the fourteen rounds of BLAKE-256, it the design requires two additional clock cycles before proceeding to finalization. The two additional clock cycles are required to write back the internal state words into distributed RAMs in other words to empty the pipeline stage.

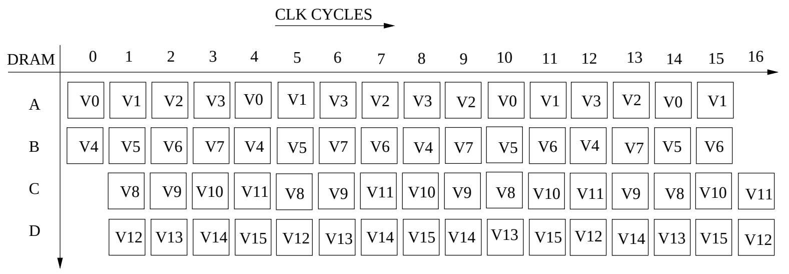

CLK CYCLES
DRAM
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
A
V0 V1 V2 V3 V0 V1 V3 V2 V3 V2 V0 V1 V3 V2 V0 V1
B
V4 V5 V6 V7 V4 V5 V7 V6 V4 V7 V5 V6 V4 V7 V5 V6
C
V8 V9 V10 V11 V8 V9 V11 V10 V9 V8 V10 V11 V9 V8 V10 V11
D
V12 V13 V14 V15 V12 V13 V14 V15 V14 V13 V15 V12 V14 V13 V15 V12

Figure 4.8: Pipeline of Round Function

### Distributed RAM Architecture

Distributed RAM version of this design replaces the block RAM with two distributed RAM's which is similar to our previous design. We a register the value after the XOR operation as the LUT based design has free register and it won't increase the area. As we have same architecture, controller designed for the block RAM version serves the purpose for the distributed RAM with a simple tweak at the addressing of block RAM.

## 4.5.2 Control Unit

We develop a ROM based FSM controller for this architecture. The main state is controlled by an FSM and control words are stored in distributed ROMs which are addressed by a counter. The addressing of the distributed RAM's repeats for each round, with this we can reduce the area of the controller. We develop a ROM which stores the addresses of the distributed ROMs and this can be utilized across all the rounds. We employ the same methodology as our previous design to address the additional four rounds of BLAKE-256. As we reduce the clock cycles count, we can reschedule the addresses in $256x4 - bit$ ROM for addressing the permuted message and constants in the block RAM. Though we employ all the optimization techniques, the addressing takes 70% of the controller and still occupies most of the area.

# Chapter 5: JH

## 5.1 Introduction

JH[16] is a hash function developed by Hongjun Wu for the Secure Hash Algorithm-3(SHA-3) contest initiated by NIST for selecting a new hashing standard. JH has four hash algorithms JH-224, JH-256, JH-384 and JH-512. The algorithms are built on simple components which makes efficient to implement in hardware and software.

JH structure is constructed using compression function from a large block cipher with a constant key. It is based on generalized AES design methodology, so that a large block cipher can be constructed from small components easily. Each message block in JH is 64 bytes and it passes through 42-round compression function. The compression function structure is given in Fig 5.1. The block size of the cipher is $2m$ bits. In the compression function, the $2m$-bit hash value $H^{(i-1)}$ and $m$-bit message block $M^{(i)}$ are compressed to produce a $2m$-bit $H^{(i)}$. The compression function structure uses a key which is set to constant (permutation). By this no extra variables are introduced into the middle of the function.

## 5.2 Generalized AES Methodology

AES uses substitution-permutation network (SPN) with the input as a two-dimensional array. A Maximum Distance Separable (MDS) code is applied to the column in the even rounds and to the odd rounds. Row rotations in AES makes the round functions identical. This AES design methodology is generalized so that a large block cipher can be constructed from small components. In the generalized methodology, the bits are divided into elements and they form a $d$-dimensional array.

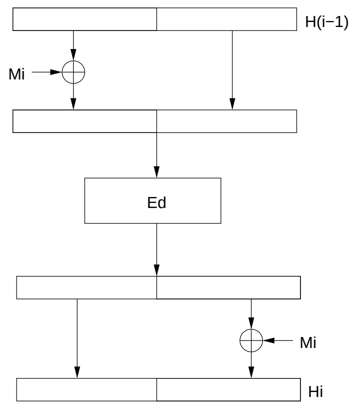


Figure 5.1: Compression Function

For JH, eight-dimensional generalized AES design is used to construct the block cipher. The input of the block cipher, which is 64 bytes is divided into 256 4-bit elements, and they form an eight-dimensional array. The constant used as round keys are generated using a six-dimensional block cipher.

## 5.3 Specifications

### 5.3.1 Notations

The following Notations and Parameters are used in JH.

| | |
|---|---|
| $Word$ | A group of bits |
| $A^i$ | $i^{th}$ bit in word A |
| $C_r^{(d)}$ | $2^d$-bit constant word used in round. |
| $d$ | dimension of a block of bits consisting of $2^d$ 4-bit elements. |
| $h$ | Number of bits in a hash value. |
| $H^{(i)}$ | $i^{th}$ hash value of $h$-bits. |
| $H^{(i),j}$ | $j^{th}$ bit of the $i^{th}$ hash value. |
| $l$ | Length of message in bits. |
| $m$ | Number of bits in a message block $M$. |
| $M$ | Message to be hashed. |
| $M^{(i)}$ | Message block $i$. |
| $N$ | Number of blocks in the padded message. |

The following operations are used in JH specifications.

| | |
|---|---|
| $\&$ | Bitwise AND operation. |
| $\mid$ | Bitwise OR operation. |
| $\oplus$ | Bitwise XOR operation. |
| $\neg$ | Bitwise complement operation. |
| $\mid$ | Concatenation operation. |

### 5.3.2 Functions

The following functions are used in JH specifications.

#### S-boxes

JH uses two S-boxes, $S0$ and $S1$ which are 4x4-bit S-boxes. The xored input passes through the Sboxes and every round constant bit selects which Sboxes to be used. This increases the overall algebraic complexity.

Table 5.1: Sboxes S0 and S1

| $x$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S_0(x)$ | 9 | 0 | 4 | 11 | 13 | 12 | 3 | 15 | 1 | 10 | 2 | 6 | 7 | 5 | 8 | 14 |
| $S_1(x)$ | 3 | 12 | 6 | 13 | 5 | 7 | 1 | 9 | 15 | 2 | 0 | 4 | 11 | 10 | 14 | 8 |

**Linear transformation L**

Linear transformation operates on two 4-bit words and implements a (4,2,3) Maximum Distance Separable (MDS) code over $GF(2^4)$. The multiplication $GF(2^4)$ is defined as the multiplication of binary polynomials modulo $x^4 + x^2 + 1$. If $A, B$ are the input and $C, D$ are the outputs, linear transform can be represented as

$$(C, D) = L(A, B) = (5 \cdot A + 2 \cdot B + 2 \cdot A + B)$$

Bit wise computation of linear transformation can be represented as shown in Fig 5.2.

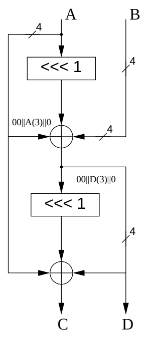


Figure 5.2: Linear Transformation $L$

**Permutation $P_d$**

$P_d$ is a simple permutation of $2^d$ 4-bit elements which is similar to row rotations in AES. It is constructed from three different permutations $\pi_d$, $P'_d$ and $\phi_d$. Each takes $2^d$ input

$A = (a_0, \cdots, a_{2^d-1})$ and produces $2^d$ output $B = (b_0, \cdots, b_{2^d-1})$.

*Permutation $\pi_d$*

The computation of B $= \pi_d(A)$ is follows and is illustrated in Fig 5.3.

$$\begin{aligned} b_{4i+0} &= a_{4i+0}\ for\ i = 0\ to\ 2^{d-2}-1 \\ b_{4i+1} &= a_{4i+1}\ for\ i = 0\ to\ 2^{d-2}-1 \\ b_{4i+2} &= a_{4i+3}\ for\ i = 0\ to\ 2^{d-2}-1 \\ b_{4i+3} &= a_{4i+2}\ for\ i = 0\ to\ 2^{d-2}-1 \end{aligned}$$

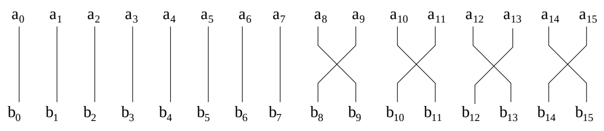

$a_0$ $a_1$ $a_2$ $a_3$ $a_4$ $a_5$ $a_6$ $a_7$ $a_8$ $a_9$ $a_{10}$ $a_{11}$ $a_{12}$ $a_{13}$ $a_{14}$ $a_{15}$
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$ $b_{15}$

Figure 5.3: Permutation $\pi_d$

*Permutation $P'_d$*

The computation of B $= P'_d(A)$ is follows and is illustrated in Fig 5.4.

$$\begin{aligned} b_i &= a_{2i}\ for\ i = 0\ to\ 2^{d-1}-1 \\ b_{i+2^{d-1}} &= a_{2i+1}\ for\ i = 0\ to\ 2^{d-1}-1 \end{aligned}$$

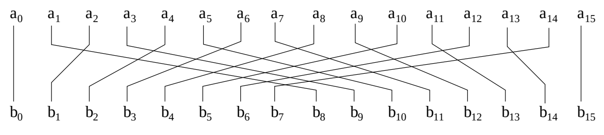

$a_0$ $a_1$ $a_2$ $a_3$ $a_4$ $a_5$ $a_6$ $a_7$ $a_8$ $a_9$ $a_{10}$ $a_{11}$ $a_{12}$ $a_{13}$ $a_{14}$ $a_{15}$
$b_0$ $b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$ $b_{15}$

Figure 5.4: Permutation $P'_d$

*Permutation $\phi_d$*

The computation of B = $\phi_d$(A) is follows and is illustrated in Fig 5.5.

$$\begin{aligned} b_i &= a_i \ fori = 0 \ to \ 2^{d-1} - 1 \\ b_{2i+0} &= a_{2i+1} \ for \ i = 2^{d-2} \ to \ 2^{d-1} - 1 \\ b_{2i+1} &= a_{2i+0} \ for \ i = 2^{d-2} \ to \ 2^{d-1} - 1 \end{aligned}$$

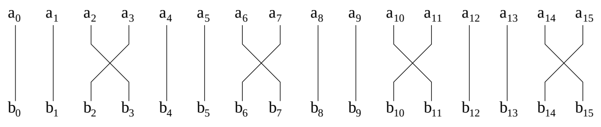

a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15

Figure 5.5: Permutation $\phi_d$

$P_d$ is a composition of $\pi_d$, $P_d'$ and $\phi_d$ and is illustrated in Fig 5.6.

$$P_d = \pi_d \ o \ P_d' \ o \ \phi_d$$

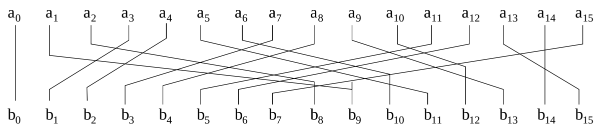

a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15

Figure 5.6: Permutation $P_d$

## Round Function $R_d$

Round function implements generalized AES design methodology and consists of three layers. They are

1. S-boxes

2. Linear Transformation
3. Permutation Layer

The input for the round function is $2^d$ 4-bit word A denoted as $A = (a_0, \cdots, a_{2^d-1})$ and produces $2^d$ output $B = (b_0, \cdots, b_{2^d-1})$. The computation of $B = R_d(A, C_r^{(d)})$ is given as follows:

- for i = 0 to $2^d - 1$,

  ```
  {
  if $C_r^{(d),i} = 0$ then $v_i = S_0(a_i)$;
  if $C_r^{(d),i} = 1$ then $v_i = S_1(a_i)$;
  }
  ```

- $(w_{2i}, w_{2i+1}) = L(v_{2i}, w_{2i+1})$ for $0 \leq$ i $\leq 2^{d-1} - 1$;
- $(b_0, \cdots, b_{2^d-1}) = P_d(w_0, \cdots, w_{2^d-1})$

**Bijective Function $E_d$**

Bijective function $E_d$ is based on $d$-dimensional generalized AES design methodology. It applies SPN and MDS to the d-dimensional array and is constructed from $6(d-1)$ rounds of $R_d$. Let $A, B$ be the input and output of the $E_d$ and $Q_r, Q_{r+1}$ be the input and output for the round function. The computation of $B = E_d(A)$ and is as follows

1. grouping of A into $2^d$ 4-bit elements $Q_r$
2. for r = 0 to 6(d-1)-1, $Q_{r+1} = R_d(Q_r, C_r^{(d)})$
3. Ungrouping of $Q_{r+1}$ to obtain B

*Grouping*

```
for i = 0 to $2^{d-1} - 1$,
{
```

```
$q_{0,2i} = A^i \parallel A^{i+2.2^d} \parallel A^{i+3.2^d}$;
$q_{0,2i+1} = A^{i+2^{d-1}} \parallel A^{i+2^{d-1}+2.2^d} \parallel A^{i+2^{d-1}+3.2^d}$;
}
```

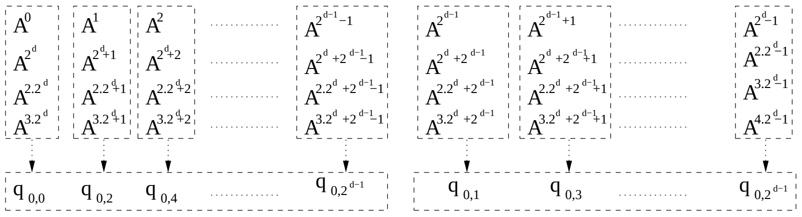


Figure 5.7: Grouping of $A$

*Degrouping*

```
for i = 0 to $2^{d-1} - 1$,
{
$B^i \parallel B^{i+2.2^d} \parallel B^{i+3.2^d} = q_{6(d-1),2i}$;
$B^{i+2^{d-1}} \parallel B^{i+2^{d-1}+2.2^d} \parallel B^{i+2^{d-1}+3.2^d} = q_{6(d-1),2i+1}$;
}
```

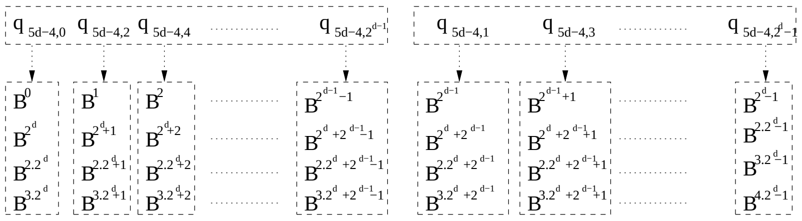


Figure 5.8: Ungrouping of $q$

**Round Constants**

The round constants are a $2^d$-bit word. The round constants $C_r^{(d)}$ for each round are generated using the round function $R_{d-2}$.

$$C_r^{(d)} = R_{d-2}(C_{r-1}^{(d)}, 0)$$

## 5.4 Compression Function

The compression function $F_d$ is given in Fig 5.1. $F_d$ compresses the $2^{d+1}$-bit message block and $2^{d+2}$-bit $H^{(i-1)}$ into $2^{d+2}$-bit $H^{(i)}$.

$$H^{(i)} = F_d(H^{(i-1)}, M^{(i)})$$

### 5.4.1 $F_8$

$F_8$ compresses 512-bit message block $M^{(i)}$ and 1024-bit $H^{(i-1)}$ into 1024-bit $H^{(i)}$. The computation of $H^{(i)} = F_8(H^{(i-1)}, M^{(i)})$ is as follows

1. $A^j = H^{(i-1),j} \oplus M^{(i),j} \quad for\ 0 \leq j \leq 511$

   $A^j = H^{(i-1),j} \quad for\ 512 \leq j \leq 1023$

2. $B = E_8(A)$

3. $H^{(i),j} = H^j \quad for\ 0 \leq j \leq 511$

   $H^{(i),j} = B^j \oplus M^{(i),j-512} \quad for\ 512 \leq j \leq 1023$

### 5.4.2 Block diagram

The block diagram of JH42 is shown in Fig.5.11.

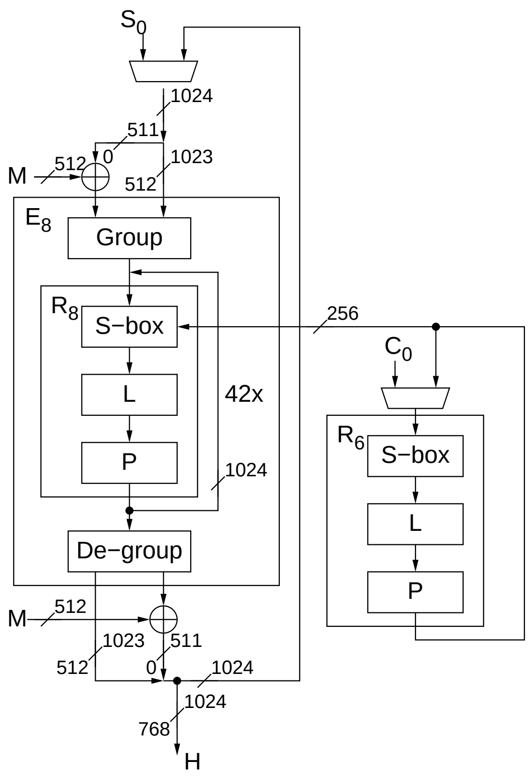


Figure 5.9: Block Diagram of JH42

# Chapter 6: Compact Implementations of JH

## 6.1 Compact Implementation of JH

### 6.1.1 Datapath

#### Distributed RAM Architecture

Optimization techniques discussed earlier and developing resource constrained implementations of BLAKE help us to design efficient compact architecture of JH. JH is built on simple functions, but requires more storage as the internal state and chaining hash value are 1024-bits each. In addition to this, we need to store the message and round constants for the round function $R_d$. Using a single memory space to store all the required bits will limit the access which increases the clock cycles and this effects throughput of the design.

The compression function $F_d$ of JH has one main function $E_d$ which consists of grouping, round function $R_d$ and ungrouping. Grouping and ungrouping reorders 1024-bits of chaining hash value to produce a 1024-bit internal state and vice versa. For each message block, the ungrouped chaining hash value is grouped to form an internal state and after 42 iterations of the round function $R_d$ the internal state is ungrouped. Though it does n't change the value of the bits, the two functions consume significant amount of clock cycles. Storing the initial/chaining hash values in grouped state helps to reduce the clock cycles count. The grouped internal state is produced by $XOR$ing the initial/chaining hash value with the message. Accumulating the hash values and the internal state in same storage element requires dedicated write clock cycles, for synchronous RAM and this doubles the clock cycles count. So we use different distributed RAM's to store the hash values and the internal state. The initial and chaining hash values are 1024-bit each, we use two $32x32 - bit$ single port distributed RAMs for storage. The hash values are split into half, and stored in different

Table 6.1: Permutation of Sixteen 4-bit words of Internal state

| Initial | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi_d$ | 0 | 1 | 3 | 2 | 4 | 5 | 7 | 6 | 8 | 9 | 11 | 10 | 12 | 13 | 15 | 14 |
| $P'_d$ | 0 | 128 | 129 | 1 | 2 | 130 | 131 | 3 | 4 | 132 | 133 | 5 | 6 | 134 | 135 | 7 |
| $\phi_d$ | 0 | 129 | 128 | 1 | 2 | 131 | 130 | 3 | 4 | 133 | 132 | 5 | 6 | 135 | 134 | 7 |
| $P_d$ | 0 | 129 | 128 | 1 | 2 | 131 | 130 | 3 | 4 | 133 | 132 | 5 | 6 | 135 | 134 | 7 |
| Final | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 |

distributed RAMs. This gives independent access to two 32-bits of the same hash value which reduces clock cycles during $XOR$ operation with the message.

The round function $R_d$ iterates for 42 times and operates on 1024-bit internal state. As $R_d$ consists of simple s-box, $XOR$ and rotate operations, processing two 32-bit values in one clock cycle can be done with minimal increase in area and it also cuts down the clock cycles count by half. For this, we store the 1024-bit internal state in two $16x32 - bit$ distributed RAMs. Our implementation uses 16 s-boxes and eight linear transform functions to process the two 32-bit values. The half processed values are permuted and this produces sixteen 4-bit words which are out of sequence. In order to process the internal state for the following rounds, we rearrange the 4-bit words to produce a sequenced 4-bit words. These words are stored back in the distributed RAMs. The distributed RAMs are configured with different read and write addresses. This helps to avoid address shift because of the register and also reduces the control logic because of congruency among the rounds. The permutation and rearranging of sixteen 4-bit words is given in table 6.1. The round constants required by $R_d$ are processes through round function $R_6$ which is similar to round function $R_d$. The $R_6$ processes two 4-bit words in one clock cycle and the initial round constants for the first message block are stored in distributed ROM. For each new message the two $16x4 - bit$ distributed RAMs are loaded with the initial round constants. As we process 64-bits from both the distributed RAMs in each clock cycle, it takes 16 clock cycles to complete one round.

The internal state after processing for 42 rounds through round function $R_d$, should be

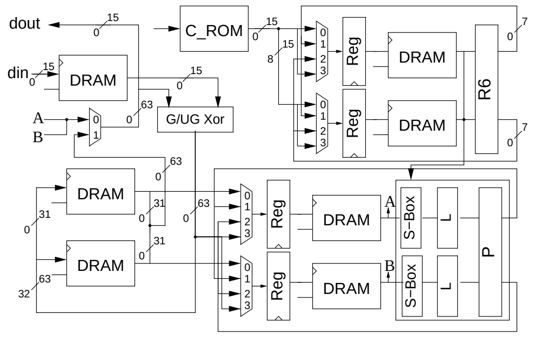


Figure 6.1: Datapath of JH

$XOR$ed with the message for generating the chaining or the final hash value. This requires the storage of message in a distributed RAM. As we have a 16-bit interface and considering the grouping and ungrouping function of JH, we use a $32x16-bit$ distributed RAM to store the message. For each clock cycle, only four bits of message is $XOR$ed with the internal state/hash value. So, both the grouping and ungrouping required 64 clock cycles each. After producing the chaining hash value, it is written back into $32x32-bit$ distributed RAMs which store the initial and chaining hash values.

### Block RAM Architecture

The distributed RAM architecture uses both single port and read/write port distributed RAMs. In spartan-3 architecture, a block RAM cannot be configured to have a both read/write address for a single port. So the distributed RAMs which store the internal state cannot be converted into block RAM. Two single port synchronous distributed RAMs which store the initial and chaining hash values are joined and they can be replaced with a dual-port block RAM. As we design synchronous distributed RAMs to store the initial and chaining hash values, converting them to block RAM will not effect the clock cycles count.

### 6.1.2 Control Unit

We have been employing ROM based FSM methodology for designing control units of compact implementations. Compact implementation of JH also implements ROM based FSM controller, which consists of a main FSM for controlling the transfer of states. The total control signals are divided into states and ROMs are used to store the select signals for each state. Each ROM is selected depending on the state and a counter is used to address the ROM for generating the specified/required control bits for each clock cycle. Apart from the select signals the other control signals are the addressing of distributed RAMs of both internal state and the round constants. As the round constants and the internal state are processed using same round function architecture they have same address bits in a single clock cycle. So, a single ROM can hold the read and write addresses required by both distributed RAM pairs, this minimizes the control logic area. The message is stored in a $32x16 - bit$ distributed RAM and only four bits is processed in a single clock cycle. As a five bit counter takes less area than a 32-depth ROM, we use a counter which steps up the value after two clock cycles for addressing distributed RAM which stores the message.

# Chapter 7: Results and Conclusion

## 7.1 Tools Used

All the implementations described in chapter 4 and 6 are targeted and implemented on the smallest Spartan 3 device XC3S50-5 using Xilinx ISE v13.1 Webpack. Along with the smallest Spartan 3 device, we implemented the designs on low-cost Spartan-6 and high speed devices like Virtex-5 and Virtex-6 for a fair comparison with other group implementations. The designs are also synthesized and implemented on Altera Cyclone-II devices using Altera Quartus II v10.0 Web edition. All results were generated using benchmarking tool ATHENa (Automated Tool for Hardware EvaluatioN) [11].

## 7.2 Results and Analysis of Compact Implementations

Compact architectures of BLAKE and JH are implemented using the optimization techniques discussed in chapter 2. Different architectures are developed considering various design configurations. Pipelining, quasi-pipelining and configuration of internal state in block RAM/distributed RAM are the design configurations, which results in diverse architectures. Table 7.1 summarizes the implementation details of BLAKE and JH algorithm.

The performance of an implementation depends on the number of clock cycles it takes to hash $N$ message blocks. Clock cycles required to process one message block varies with the implementations. The initial designs, which store the internal state in block RAM requires more number of clock cycles to process one message block. By moving the state into distributed RAM's, the clock cycles count reduces by more than half. Introducing pipeline registers and rescheduling of the G-Functions reduces the critical path by keeping the same clock cycles per round as the un-pipelined design.

Table 7.1: Implementation details of BLAKE and JH

| Algorithm | Version | Design | Datapath Size (bits) | Rounds | Clock cycles per Round | Additional Clock cycles | Clock cycles per block $p$ |
|---|---|---|---|---|---|---|---|
| BLAKE-32 | BRAM | 4.1 | 32 | 10 | 48 | 64 | 544 |
| BLAKE-32 | BRAM | 4.2 | 32 | 10 | 52 | 64 | 584 |
| BLAKE-32 | BRAM | 4.3 | 32 | 10 | 16 | 32 | 192 |
| BLAKE-32 | BRAM | 4.4 | 32 | 10 | 21 | 24 | 234 |
| BLAKE-256 | BRAM | 4.4 | 32 | 14 | 21 | 24 | 318 |
| BLAKE-256 | BRAM | 4.5 | 32 | 14 | 16 | 34 | 258 |
| JH42 | BRAM | 6.1 | 32 | 42 | 16 | 64 | 736 |
| BLAKE-32 | Logic | 4.3 | 32 | 10 | 16 | 32 | 192 |
| BLAKE-256 | Logic | 4.4 | 32 | 14 | 21 | 24 | 318 |
| BLAKE-256 | Logic | 4.5 | 32 | 14 | 16 | 34 | 258 |
| JH42 | Logic | 6.1 | 32 | 42 | 16 | 64 | 736 |

Throughput is one of the key factor for comparing the performance of implementations. 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 $N$ to be hashed, the number of clock cycles to process one message block, block size $b$ of the algorithm and the delay/clock period $T$. Table 7.2 gives the formulae for calculating the throughput for different compact implementations of BLAKE and JH.

The implementation results are summarized in Table 7.3. It can be seen that all implementations fall within our target range of 400 to 600 slices with one block RAM and 800 slices using logic cells on Spartan-3 device. We also compare the throughput for long and short messages in Table 7.3.

The dedicated write clock cycle-I design stores the internal state in block RAM and requires 480 clock cycles to process one message block. The critical path of the design is 15.05 ns which results in a low throughput. Rescheduling the pipeline stage to reduce the time delay to 14.46 ns increases the clock cycles count but the decrease in time delay is not consistent with the increase in clock cycles. This reduces the throughput of dedicated write

Table 7.2: Throughput formulae for BLAKE and JH

| Version | Algorithm | Design | Block Size (bits) $b$ | # Rounds | Clock Cycles per Round | Additional Clock Cycles | Clock Cycles to hash $N$ blocks $clk = st + (\ l + \ p) \cdot N + \ end$ | Throughput (long Messages) $\frac{b}{(l+p)\cdot T}$ |
|---|---|---|---|---|---|---|---|---|
| BRAM | BLAKE-32 | 4.1 | 512 | 10 | 48 | 64 | $2 + (\ 32 + \ 544) \cdot N + \ 17$ | $512/(\ 576 \cdot T)$ |
| | BLAKE-32 | 4.2 | 512 | 10 | 52 | 64 | $2 + (\ 32 + \ 584) \cdot N + \ 17$ | $512/(\ 616 \cdot T)$ |
| | BLAKE-32 | 4.3 | 512 | 10 | 16 | 32 | $2 + (\ 32 + \ 192) \cdot N + \ 17$ | $512/(\ 224 \cdot T)$ |
| | BLAKE-32 | 4.4 | 512 | 14 | 21 | 24 | $2 + (\ 32 + \ 234) \cdot N + \ 17$ | $512/(\ 266 \cdot T)$ |
| | BLAKE-256 | 4.4 | 512 | 14 | 21 | 24 | $2 + (\ 32 + \ 318) \cdot N + \ 17$ | $512/(\ 350 \cdot T)$ |
| | BLAKE-256 | 4.5 | 512 | 14 | 16 | 34 | $2 + (\ 32 + \ 258) \cdot N + \ 17$ | $512/(\ 290 \cdot T)$ |
| | JH42 | 6.1 | 512 | 42 | 16 | 64 | $2 + (\ 32 + \ 736) \cdot N + \ 17$ | $512/(\ 800 \cdot T)$ |
| Logic only | BLAKE-32 | 4.3 | 512 | 10 | 16 | 32 | $2 + (\ 32 + \ 192) \cdot N + \ 17$ | $512/(\ 224 \cdot T)$ |
| | BLAKE-256 | 4.4 | 512 | 14 | 21 | 24 | $2 + (\ 32 + \ 318) \cdot N + \ 17$ | $512/(\ 350 \cdot T)$ |
| | BLAKE-256 | 4.5 | 512 | 14 | 16 | 34 | $2 + (\ 32 + \ 258) \cdot N + \ 17$ | $512/(\ 290 \cdot T)$ |
| | JH42 | 6.1 | 512 | 42 | 16 | 64 | $2 + (\ 32 + \ 736) \cdot N + \ 17$ | $512/(\ 800 \cdot T)$ |

clock cycle-II resulting a low throughput/area ratio. Moving the internal state out of the block RAM results in relatively less clock cycles for processing one message block but still the time delay is the lagging factor. So, we employ both quasi-pipelining and rescheduling to develop an effective design which requires relatively less clock cycles and also a decent time delay which results in good throughput/area ratio of all the designs. All the compact architectures of BLAKE fall within the area of 400 to 500 slices with one block RAM on Xilinx Spartan-3 device, which is our primary design target. The permutation function of BLAKE is different across the rounds and this results in a complex and large controller. For JH, both block RAM and distributed RAM architectures are smaller than BLAKE due to constant permutation across the rounds. The constant permutation results in a simple and small controller which occupies just 20% of the total design area.

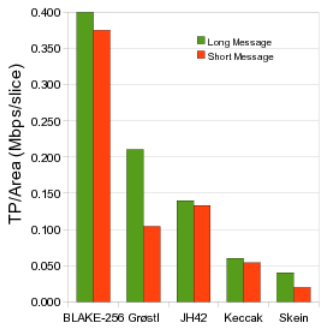
## 7.3 Comparison of SHA-3 Finalists

Performance of an algorithm in software and hardware are key factors in selecting the five finalists. Compact architectures evaluate the efficiency of these algorithms in area constrained environment which is also an essential criteria. Our Cryptographic Engineering Research Group (CERG) implemented the remaining three finalists with the same design criteria, targeted for the same device which helps for a fair comparison. The results are reported in the following publication [17] and also available on ATHENa website [18]. Though all the implementations are designed and targeted for Xilinx Spartan-3, they are implemented on other devices for evaluation. We implemented different architectures for BLAKE, but we consider our best design for comparison with the five finalists. Table 7.4 and 7.5 gives the implementation results of all the five finalists on Xilinx and Altera devices.

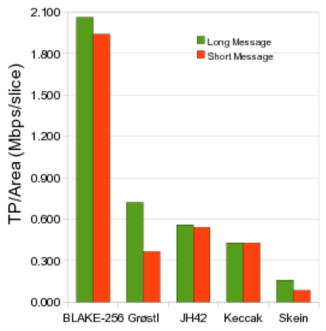
Compact implementation results of the five finalists on Xilinx and Altera show that BLAKE has a good throughput to area ratio of all the algorithms on Xilinx and Altera devices. Compact implementations depend on the scalability of the algorithm. BLAKE consists of eight identical G-Functions which are really scalable. This scalability option of BLAKE results in a high throughput to area ratio over the other algorithms. JH performs decently among all the algorithms considering its scalability and number of rounds. Though BLAKE and JH requires same amount of clock cycles for processing one round, 42 rounds of JH results in more number of clock cycles for processing one message block which reduces the overall throughput of the design. Apart from the number of rounds, JH is scalable and also has constant permutation function which results in consistent low area of all the algorithms.

## 7.4 Comparison With Other Published Implementation Results

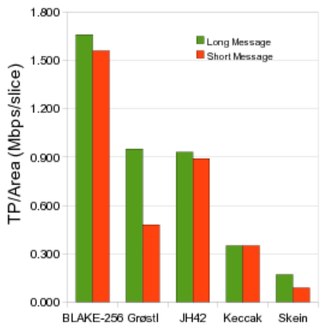
The first compact implementation of BLAKE was reported by Jean-Luc Beuchat and Teppei Yamazaki [19]. This implementation uses two block RAMs, one for storage and one for



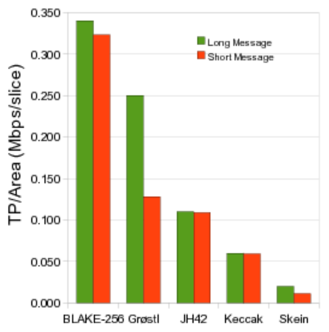
(a) on Spartan-3 (BRAM)



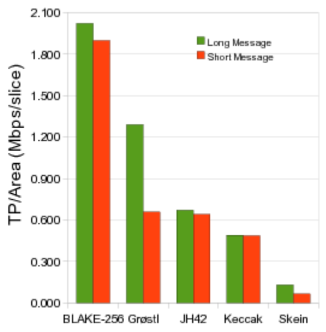
(b) on Spartan-6 (BRAM)



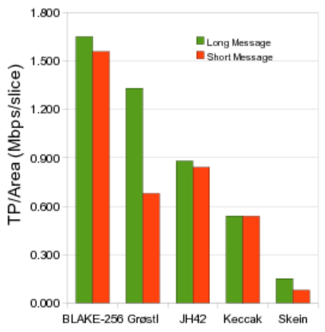
(c) on Virtex-V (BRAM)



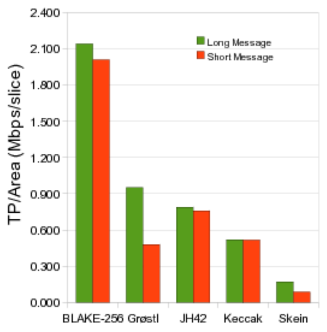
(d) on Spartan-3 (Logic)



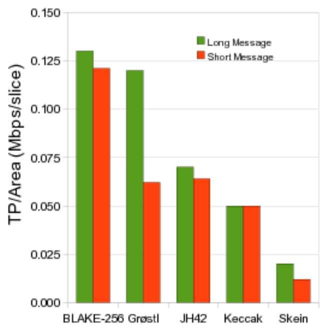
(e) on Spartan-6 (Logic)



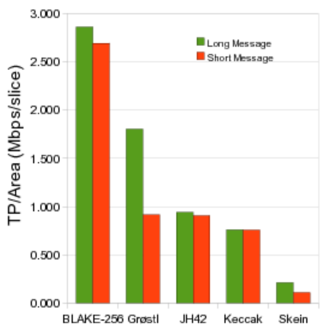
(f) on Virtex-V (Logic)



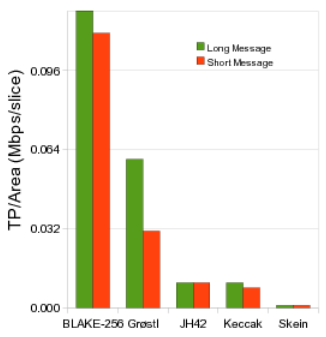
(g) on Virtex-6 (BRAM)



(h) on Cyclone-II (BRAM)



(i) on Virtex-6 (Logic)



(j) on Cyclone-II (Logic)

Figure 7.1: Throughput over area ratio on Xilinx and Altera devices

generating the control signals. It has four pipeline stages and also has very low area on Spartan-3 device but it takes more number of clock cycles for processing one message block. The implemented design uses two block RAM and has low throughput to area ratio and no specific design criteria. We developed a design criteria for a fair comparison of all the five finalists. BLAKE and JH are implemented considering this design criteria are implemented on the same devices for an equitable comparison with other published results. Table 7.6 gives the comparison of our implementation results with other reported results.

The comparison shows that our compact implementation of BLAKE and JH has a good throughput to area ratio than other group implementations. Compact implementations of BLAKE and JH by [20] and [21] doesn't consider the loading clock cycles for calculating the throughput. Our GMU-Light Weight Protocol considers the clock cycles for loading the message as it takes place for each message block and also effects the throughput of the design. Since our implementations are targeted to Spartan-3 devices, implementing the same designs on Spartan-6 and Virtex-5 do not utilize the architectural features of those devices, effecting the throughput of the designs. Though the designs don't use the same I/O protocol and datapath size, effort was made for a fair comparison by using the same target device as of other implementations.

## 7.5 Conclusion

In this work, we presented compact FPGA implementations of two SHA-3 finalists BLAKE and JH. Both the algorithms were implemented using the same assumptions, goals, tools, interface, and the same area optimization techniques. All the implementations of BLAKE and JH are evaluated with respect to throughput over area ratio. The results are compared with other SHA-3 finalists which use the same assumptions, goals, tools and interface. The comparison helps to rank the algorithm based on performance in area constrained environment. Ranking of compact implementations shows that BLAKE ranks first among all the implementations followed by Grøstl and JH. The ranking is different from implementations for best throughput to area ratio designs reported in [7], [22], [23]. Increasing the area criteria of compact implementations for a better throughput over area ratio will help us to evaluate the scalability of these algorithms. Our compact implementation results of BLAKE and JH are compared with other published results and the variations shows that our results outperform all the reported results.

## 7.6 Future Work

In this thesis compact implementations of BLAKE and JH are implemented with a design criteria of 400-600 slices using one block RAM and 800 slices with no block RAM on Xilinx Spartan-3 device. The implementation results show that both compact implementations of BLAKE and JH fall far below the area criteria of 800 slices with no block RAM. Evaluating the throughput of these designs by unrolling will increase the area and also helps for better understanding the scalability of the algorithms. In addition to unrolling the compact architectures on Spartan-3, designing compact architectures by considering architectural features of low power devices like Spartan-6 and high throughput devices like Virtex-5 and Virtex-6 will help use to develop better throughput over area ratio designs.

Table 7.3: Implementation results of BLAKE and JH

| | | Message | | | | Long | | Short | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Device | Version | Algorithm | Design | Area (slices) | Block RAMs | Maximum Delay (ns) $T$ | Throughput (Mbps) | TP/Area (Mbps/slice) | Throughput (Mbps) | TP/Area (Mbps/slice) |
| Xilinx Spartan-3 (xc3s50-5) | BRAM | BLAKE-32 | 4.1 | 537 | 1 | 15.05 | 59.1 | 0.11 | 57.2 | 0.106 |
| | | BLAKE-32 | 4.2 | 540 | 1 | 14.46 | 57.4 | 0.10 | 55.8 | 0.103 |
| | | BLAKE-32 | 4.3 | 450 | 1 | 14.93 | 153.1 | 0.34 | 141.1 | 0.313 |
| | | BLAKE-32 | 4.4 | 527 | 1 | 7.38 | 261.0 | 0.50 | 243.6 | 0.462 |
| | | BLAKE-256 | 4.4 | 545 | 1 | 8.42 | 173.8 | 0.32 | 164.8 | 0.302 |
| | | BLAKE-256 | 4.5 | 549 | 1 | 8.05 | 219.3 | 0.40 | 205.9 | 0.375 |
| | | JH42 | 6.1 | 502 | 1 | 9.19 | 69.6 | 0.14 | 66.6 | 0.133 |
| | Logic only | BLAKE-32 | 4.3 | 527 | 0 | 15.65 | 146.0 | 0.27 | 135.7 | 0.257 |
| | | BLAKE-256 | 4.4 | 612 | 0 | 9.12 | 160.4 | 0.26 | 152.9 | 0.249 |
| | | BLAKE-256 | 4.5 | 631 | 0 | 8.17 | 216.3 | 0.34 | 203.6 | 0.323 |
| | | JH42 | 6.1 | 558 | 0 | 10.05 | 63.7 | 0.11 | 61.1 | 0.110 |
| Xilinx Spartan-6 (xc6slx4-3) | BRAM | BLAKE-32 | 4.1 | 136 | 1 | 7.26 | 122.4 | 0.90 | 118.5 | 0.871 |
| | | BLAKE-32 | 4.2 | 142 | 1 | 7.09 | 117.2 | 0.83 | 113.7 | 0.800 |
| | | BLAKE-32 | 4.3 | 141 | 1 | 10.28 | 222.1 | 1.57 | 204.7 | 1.452 |
| | | BLAKE-256 | 4.4 | 139 | 1 | 6.47 | 226.0 | 1.62 | 214.3 | 1.542 |
| | | BLAKE-256 | 4.5 | 152 | 1 | 5.62 | 313.8 | 2.06 | 294.5 | 1.937 |
| | | JH42 | 6.1 | 182 | 1 | 5.34 | 102.7 | 0.56 | 98.4 | 0.540 |
| | Logic only | BLAKE-32 | 4.3 | 158 | 0 | 11.45 | 199.6 | 1.26 | 185.5 | 1.174 |
| | | BLAKE-256 | 4.4 | 152 | 0 | 7.21 | 202.8 | 1.33 | 193.2 | 1.272 |
| | | BLAKE-256 | 4.5 | 164 | 0 | 5.34 | 330.6 | 2.02 | 311.2 | 1.898 |
| | | JH42 | 6.1 | 156 | 0 | 6.14 | 104.2 | 0.67 | 100.0 | 0.641 |
| Xilinx Virtex-V (xc5vlx20-2) | BRAM | BLAKE-32 | 4.1 | 147 | 1 | 5.89 | 150.9 | 1.02 | 146.1 | 0.995 |
| | | BLAKE-32 | 4.2 | 159 | 1 | 5.66 | 146.6 | 0.92 | 142.2 | 0.894 |
| | | BLAKE-32 | 4.3 | 184 | 1 | 7.47 | 305.6 | 1.66 | 281.7 | 1.533 |
| | | BLAKE-32 | 4.4 | 238 | 1 | 4.37 | 440.8 | 1.85 | 411.4 | 1.724 |
| | | BLAKE-256 | 4.4 | 212 | 1 | 4.30 | 340.3 | 1.61 | 322.8 | 1.520 |
| | | BLAKE-256 | 4.5 | 248 | 1 | 4.29 | 411.9 | 1.66 | 386.5 | 1.550 |
| | | JH42 | 6.1 | 176 | 1 | 3.91 | 163.6 | 0.92 | 156.8 | 0.890 |
| | Logic only | BLAKE-32 | 4.3 | 214 | 0 | 7.85 | 291.1 | 1.36 | 270.6 | 1.264 |
| | | BLAKE-256 | 4.4 | 231 | 0 | 4.12 | 355.0 | 1.53 | 338.6 | 1.465 |
| | | BLAKE-256 | 4.5 | 271 | 0 | 3.94 | 448.2 | 1.65 | 422.0 | 1.557 |
| | | JH42 | 6.1 | 183 | 0 | 3.99 | 160.3 | 0.88 | 153.8 | 0.840 |
| Xilinx Virtex-6 (xc6vlx75T-1) | BRAM | BLAKE-32 | 4.1 | 138 | 1 | 6.03 | 147.4 | 1.06 | 142.7 | 1.035 |
| | | BLAKE-32 | 4.2 | 135 | 1 | 5.87 | 141.4 | 1.04 | 137.2 | 1.014 |
| | | BLAKE-32 | 4.3 | 139 | 1 | 8.10 | 282.1 | 2.03 | 260.0 | 1.871 |
| | | BLAKE-256 | 4.4 | 146 | 1 | 5.26 | 277.7 | 1.90 | 263.4 | 1.804 |
| | | BLAKE-256 | 4.5 | 163 | 1 | 5.06 | 348.7 | 2.13 | 327.2 | 2.007 |
| | | JH42 | 6.1 | 196 | 1 | 4.11 | 155.5 | 0.79 | 149.0 | 0.760 |
| | Logic Only | BLAKE-32 | 4.3 | 165 | 0 | 8.39 | 272.4 | 1.65 | 253.2 | 1.534 |
| | | BLAKE-256 | 4.4 | 142 | 0 | 4.01 | 364.8 | 2.56 | 347.9 | 2.450 |
| | | BLAKE-256 | 4.5 | 166 | 0 | 3.72 | 474.6 | 2.86 | 446.9 | 2.692 |
| | | JH42 | 6.1 | 171 | 0 | 3.96 | 161.5 | 0.94 | 154.9 | 0.906 |

Table 7.4: Implementation results of SHA-3 Finalists on Xilinx

| 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 | 66.6 | 0.133 |
| | | Keccak | 590 | 1 | 8.97 | 32.2 | 0.06 | 32.1 | 0.054 |
| | | Skein | 498 | 1 | 10.65 | 20.0 | 0.04 | 10.2 | 0.020 |
| | Logic only | BLAKE-256 | 631 | 0 | 8.17 | 216.3 | 0.34 | 203.6 | 0.323 |
| | | Grøstl | 766 | 0 | 6.83 | 192.6 | 0.25 | 97.9 | 0.128 |
| | | JH42 | 558 | 0 | 10.05 | 63.7 | 0.11 | 61.1 | 0.109 |
| | | Keccak | 766 | 0 | 9.91 | 45.8 | 0.06 | 45.5 | 0.059 |
| | | Skein | 766 | 0 | 12.83 | 16.6 | 0.02 | 8.5 | 0.011 |
| Xilinx Spartan-6 (xc6slx4-3) | 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 | 98.3 | 0.540 |
| | | Keccak | 133 | 1 | 5.07 | 57.0 | 0.43 | 56.7 | 0.427 |
| | | Skein | 182 | 1 | 7.19 | 29.7 | 0.16 | 15.1 | 0.083 |
| | Logic only | BLAKE-256 | 164 | 0 | 5.34 | 330.6 | 2.02 | 311.2 | 1.898 |
| | | Grøstl | 230 | 0 | 4.43 | 297.3 | 1.29 | 151.2 | 0.657 |
| | | JH42 | 156 | 0 | 6.14 | 104.2 | 0.67 | 100.0 | 0.641 |
| | | Keccak | 161 | 0 | 5.77 | 78.7 | 0.49 | 78.1 | 0.485 |
| | | Skein | 190 | 0 | 8.77 | 24.3 | 0.13 | 12.4 | 0.065 |
| Xilinx Virtex-V (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 | 156.6 | 0.890 |
| | | Keccak | 207 | 1 | 3.99 | 72.4 | 0.35 | 72.0 | 0.348 |
| | | Skein | 218 | 1 | 5.69 | 37.5 | 0.17 | 19.1 | 0.087 |
| | Logic only | BLAKE-256 | 271 | 0 | 3.94 | 448.2 | 1.65 | 422.0 | 1.557 |
| | | Grøstl | 313 | 0 | 3.15 | 417.4 | 1.33 | 212.3 | 0.678 |
| | | JH42 | 183 | 0 | 3.99 | 160.3 | 0.88 | 153.8 | 0.840 |
| | | Keccak | 203 | 0 | 4.14 | 109.8 | 0.54 | 109.0 | 0.537 |
| | | Skein | 246 | 0 | 5.66 | 37.7 | 0.15 | 19.2 | 0.078 |
| 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 | 4.12 | 155.4 | 0.79 | 148.9 | 0.760 |
| | | Keccak | 145 | 1 | 3.84 | 75.24 | 0.52 | 75.0 | 0.517 |
| | | Skein | 207 | 1 | 6.00 | 35.6 | 0.17 | 18.1 | 0.087 |
| | Logic only | BLAKE-256 | 166 | 0 | 3.72 | 474.6 | 2.86 | 446.9 | 2.692 |
| | | 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 | 164 | 0 | 3.64 | 124.9 | 0.76 | 123.9 | 0.756 |
| | | Skein | 193 | 0 | 5.17 | 41.3 | 0.21 | 21.0 | 0.109 |

Table 7.5: 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 |

Table 7.6: Comparison of lightweight implementations of BLAKE and JH on Xilinx FPGAs, Logic only
([TW] – this work)

| Algorithm | Reference | Device | I/O Width | Datapath Width | Clock Cycles per block (p) | Area (slices) | Maximum Frequency (MHz) | Throughput (Mbps) | TP/Area (Mbps/slice) |
|---|---|---|---|---|---|---|---|---|---|
| BLAKE-256 | [20] | xc6vlx75t-1 | 64 | 64 | 1,336 | 117 | 3.65 | 105.0 | 0.897 |
| BLAKE-256 | [TW] | xc6vlx75t-1 | 16 | 32 | 290 | 166 | 3.72 | 474.6 | 2.860 |
| BLAKE-256 | [21] | xc5v | 32 | 32 | 228 | 251 | 4.73 | 115.0 | 0.927 |
| BLAKE-256 | [TW] | xc5vlx20-2 | 16 | 32 | 290 | 271 | 3.93 | 448.2 | 1.650 |
| JH42 | [20] | xc6vlx75t-1 | 64 | 64 | 689 | 240 | 3.47 | 214.0 | 0.892 |
| JH42 | [TW] | xc6vlx75t-1 | 16 | 32 | 800 | 171 | 3.96 | 161.5 | 0.940 |
| JH42 | [21] | xc5v | 32 | 8 | 6,466 | 205 | 2.93 | 27.0 | 0.132 |
| JH42 | [TW] | xc5vlx20-2 | 16 | 32 | 800 | 183 | 3.99 | 160.3 | 0.880 |

# 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. Lu and H. Howard M, “A simple power analysis attack against the key schedule of the camellia block cipher,” in *Information Processing Letters-IPL 2005*, 2005, pp. 409–412.

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

[4] *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology (NIST), FIPS Publication 197, Nov 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[5] “The SHA)-3 Zoo, month = , year = , howpublished = ECRYPT, Information Societies Technology (IST) Programme of the European Commission, note = http://ehash.iaik.tugraz.at/wiki/the_sha-3_zoo.”

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

[7] 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.

[8] M. Rawski, H. Selvaraj, and T. Luba, “An application of functional decomposition in ROM-based FSM implementation in FPGA devices,” *J. Syst. Archit.*, vol. 51, no. 6-7, pp. 424–434, 2005.

[9] I. García-Vargas, R. Senhadji-Navarro, G. Jiménez-Moreno, A. Civit-Balcells, and P. Guerra-Gutiérrez, “ROM-based finite state machine implementation in low cost FPGAs,” in *International Symposium on Industrial Electronics, ISIE 2007*. IEEE, June 2007, pp. 2342–2347.

[10] V. Skylarov, “Synthesis and implementation of RAM-based finite state machines in FPGAs,” in *Field-Programmable Logic and Applications – FPL’00*, ser. LNCS, R. W. Hartenstein and H. Grünbacher, Eds., vol. 1896. Springer-Verlag, 2000, pp. 718–728.

[11] 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.

[12] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE," Submission to NIST (Round 3), 2010, http://131002.net/blake/blake.pdf.

[13] E. Biham and O. Dunkelman, "A framework for iterative hash functions - HAIFA," Cryptology ePrint Archive, Report 2007/020, June 2007, http://eprint.iacr.org/2007/278.pdf.

[14] J. P. Aumasson and M. Willi, "The hash function family LAKE," Cryptology ePrint Archive, Report 2008, 2008, http://eprint.iacr.org/.

[15] S. Lucks, "A failure-friendly design principle for hash functions," Cryptology ePrint Archive, Report 2005, 2005, http://eprint.iacr.org/.

[16] H. Wu, "The hash function JH," Submission to NIST (updated), Sep 2009, http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/.

[17] 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.

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

[19] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "Compact implementations of BLAKE-32 and BLAKE-64 on FPGA," Cryptology ePrint Archive, Report 2010/173, 2010.

[20] 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.

[21] 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.

[22] 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/.

[23] 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.

# Curriculum Vitae

Susheel C Vadlamudi received his Bachelor degree from Vignan Engineering College under Jawaharlal Nehru Technological University (JNTU), India in 2009. He joined GMU during Fall 2009 and is aiming for a Masters Degree in Computer Engineering. He worked as a Teaching Assistant to several undergrad courses and was a grader to ECE 511 under Dr. Kaps. He was part of CERG group from Spring 2010.