LIGHTWEIGHT IMPLEMENTATIONS AND POWER MEASUREMENTS OF SHA-3 CANDIDATES ON FPGAS

by

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

Committee:

_____	Dr. Jens Peter Kaps, Thesis Director
KGaj _____	Dr. Kris Gaj, Committee Member
Mulpuri V. Rao _____	Dr. Rao Mulpuri, Committee Member
_____	Dr. Andrew Manitius, Department Chair of Electrical and Computer Engineering
_____	Dr. Lloyd J Griffiths, Dean, Volgenau School of Engineering.

Date: 01/20/2012 _____

Spring Semester 2012
George Mason University
Fairfax, VA

Lightweight Implementations and Power Measurements of SHA-3 Candidates on FPGAs

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

By

Kishore Kumar Surapathi
Bachelor of Science
Gayatri Vidhya Parishad College of Engineering, 2009

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

Spring 2012
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to my parents and friends.

Acknowledgments

I would like to thank Dr. Kaps in guiding me in every aspect of this project. Special thanks to Dr. Gaj in helping out with the design aspects of Digital Systems through his courses and motivating me with his projects. I would also like to thank Rajesh Velagaleti and Panasayya Yalla for their valuable inputs regarding the power measurement setup and in efficient design of controllers. Also thank Susheel Vadlamudi , Smrithi Gurung, John Pham, and Bilal Habib for providing me the designs of four of the five finalists. Finally, would like to thank all the CERG members for their support.

Table of Contents

List of Tables

List of Figures

Abstract

LIGHTWEIGHT IMPLEMENTATIONS AND POWER MEASUREMENTS OF SHA-3 CANDIDATES ON FPGAS

Kishore Kumar Surapathi, MS

George Mason University, 2012

Thesis Director: Dr. Jens Peter Kaps

The National Institute of Standards and Technology (NIST) has opened a public competition for a new Secure Hash Standard, SHA-3 on Nov 2, 2007. Out of the 64 submissions, 51 were selected for the first round in Dec 2008. Among them, 14 algorithms advanced to the second round in July 2009 and 5 to the third and final round in Dec 2010. The final result is expected to be announced in 2012. The selection criteria is primarilly security followed by software, and hardware performance. The hardware performance is evaluated both in FPGAs as well as in ASICs. In FPGAs, most of the research on the SHA-3 candidates is primarily targeted at high throughput. It is very interesting to see how the SHA-3 candidates perform when area is a constraint.

In this work, 4 of the 14 round two candidates (Grøstl, Luffa, SHAvite-3 and BMW) have been implemented. Furthermore, the scalability of the finalist Grøestl has been analyzed in detail. Also, a methodology for measuring power consumption of hash functions on FPGA has been developed and performed the power measurements of all the finalists. Our study shows that Grøstl performs well in resource constraint environment because of its scalability.

Chapter 1: Introduction

1.1 Hash Function

A hash function[21] takes in arbitrary length of message and produces a fixed length output called as the message digest.

A hash function should follow the following properties.

(i) It is impossible to produce the message given the hash value.

(ii)No two messages should produce the same hash output(collision resistant).

(iii)It is impossible to modify the message without modifying the hash.

Applications:

Digital signatures: The digital signature is a function (hash) of the document and can be stored and send separately independent of the document. Since no two documents produce the same signature (property of hash function), the signature is unique.

Finger print of a document: The hash value of the document is computed and when something is changed in the document, it results in a change in the hash value and thus the modification which can be created by a virus or an intruder is detected.

Storing Passwords: Instead of storing the password directly, the hash of the password will be stored.

The most commonly used hash functions are MD5 and SHA-1 until they were broken.

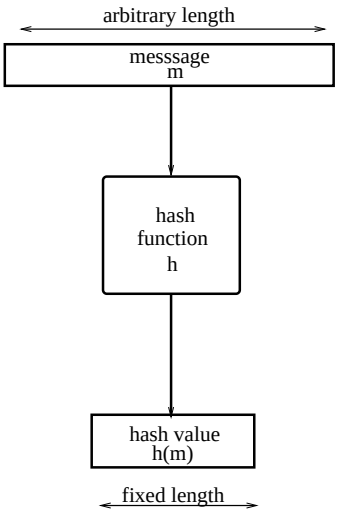


Figure 1.1: Hash Function.

1.2 SHA-3 Competition

SHA-1 was the hash standard followed by the cryptographic world until some serious attacks have been published against it in Feb 2005, following which NIST called for a workshop to assess the status and recommended a transition from SHA-1 to SHA-2 hash function. But even SHA-2 is not secured enough as it is primarily based on SHA-1. So NIST decided to call for a public competition for a new Cryptographic hash standard[2]. The winning algorithm will be named "SHA-3", and will be added to the hash algorithms currently specified in the Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standard.

A total of 64 algorithms were submitted, out of which 51 candidates were selected to advance to the first round on December 10, 2008. After initial study and analysis, 14 algorithms were selected for the second round on July 24, 2009. After extensive research on the hardware and software performance of these algorithms, based on the public and internal reviews, NIST selected five finalists - BLAKE, Grøstl, JH, Keccak, and Skein to advance to the third (and final) round of the competition on December 9, 2010. The submitters were allowed to make some tweaks to their algorithms before they submit them for the

final round of competition. NIST will be announcing the final winner and the new hash standard, SHA-3 in the second quarter of 2012.

1.3 Previous studies and Motivation

The main criteria NIST follows in deciding the final candidate is primarily security, followed by software and hardware performance [25]. Hardware performance is evaluated both in FPGAs as well as in ASICs. Most of the research in FPGAs is targeted at High throughput like the work done by Kobayashi et al. [21] , Matsuo et al [22], Gaj et al. [11],Homsirikamol et al [15] and Baldwin et al [4].

In real time scenario, it is not necessary that hash function is the primary application implemented on the FPGA. In many cases, the hash function may be implemented along with some other components like soft core processers etc. Also there is massive advancements in low powered devices in recent times which resulted in development of low cost and low power FPGAs [26]. So implementations on small FPGAs gained prominence.

This drives us towards the lightweight implementations of the SHA-3 candidates targeting certain fixed area. Four round two algorithms Grøstl , Luffa, SHAvite-3 and BMW were selected for my research based on the reported software and hardware results. At the beginning of my thesis, there are hardly any results reported for lightweight implementations. A result was reported on Grøstl [19], luffa [24] and BMW [8] . No Lightweight implementation of SHAvite-3 was reported till date.

After the final round candidates were announced, there are many results published on Grøstl on different platforms [20], [17]. But there is no result reported on Grøstl targeting the smallest FPGA device of Spartan3 family other than [19].

1.4 Thesis Organization

Chapter 2 explains the assumptions and design methodologies. Chapter 3 deals with the algorithm descriptions and their implementations. The implementation results and their

comparisions with the results posted by other group are explained in chapter 4. The scalability of Grøstl, implementation results and comparisions are discussed in chapter 5. Power measurement setup and results are explained in chapter 6. Chapter 7 deals with conclusions and future work.

Chapter 2: Methodology

2.1 Choice of Language, FPGA devices and tools

Even though all designs were targeted for Spartan-3 devices, it is interesting to see how our implementations perform on low-cost devices from another vendor such as Altera Cyclone-II, newer devices such as Spartan-6 and on high speed devices such as Xilinx Virtex-5. Complete results are published in the ATHENa results database [1]. All designs were implemented using the vendor tools: Xilinx ISE 13.3 Web Pack and Altera Quartus II v. 10.0 Web Edition, and verified after place-and-route against known answer test files provided by the submissions packet of each hash function. All results were generated using the open source benchmarking tool ATHENa (Automated Tool for Hardware EvaluatioN [12]. Other than simplifying the result generation, ATHENa also varies the vendor tool parameters to achieve optimal results.

2.2 Performance Metrics

The number of clock cycles needed to hash N message blocks using our implementations can be computed from the number of clock cycles required to perform the following functions:

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		

This results in the following formula for the number of clock cycles clk for hashing N blocks of data.

$$clk = i + h + l1 + l \cdot (N - 1) + p \cdot N + z + o$$

This formula can now be simplified to reflect the number of clock cycles needed for the initial steps before processing can begin $st = i + h + (l1 - l)$, loading and processing one block of data $l + p$, and finalization and output of the hash value $end = z + o$ resulting in (2.1).

$$clk = st + (l + p) \cdot N + end \tag{2.1}$$

Throughput is defined as the number of input bits processed per unit of time. The precise formula for throughput of a hash function is dependent on the number of message blocks N to be hashed, the block size b of the algorithm, the number of clock cycles needed to hash the message clk and the clock period T. We can derive the formula to compute the throughput from (2.1).

$$throughput(N) = \frac{b \cdot N}{clk \cdot T} = \frac{b \cdot N}{(st + (l + p) \cdot N + end) \cdot T} \tag{2.2}$$

Especially in embedded applications, messages can be very short. It is therefore important to also calculate the throughput for short messages. We use the empty message which after padding is one block long and therefore set $N = 1$ in (2.2) to compute the throughput.

When computing the throughput for very long messages, we can neglect st and end as their influence on the result goes to zero. This leads to the simplified equation (2.3).

$$throughput_{long} = \frac{b}{(l + p) \cdot T} \tag{2.3}$$

Resource Utilization of FPGAs is very difficult to define. All FPGAs contain configurable logic elements which contain flip-flops (Xilinx: slices, Altera: LE), block RAMs,

multipliers and other resources. These resources have different features not only depending on the vendor but even on the FPGA family. Hence, we can compare implementations using the metric of throughput over area ratio only within a specific FPGA family and provided they use the same number of dedicated resources. As area in this formula we use solely *slices* for Xilinx and *LEs* for Altera devices as there is no direct mapping from BRAM (block RAM) utilization to slice or LE.

2.3 Interface and Protocol

The SHA Core assumes that its inputs and outputs are connected to FIFOs. In its simplest form a FIFO is a single w-bit wide register with minimal logic to support the handshake of read/write and ready. This can easily be interfaced to a microcontroller or other circuitry in an embedded system. Lightweight applications usually have smaller databus sizes than 32 or 64 bits. Therefore, we use a databus width w of 16 bits. The protocol supports two scenarios: 1) when the message length is known and 2) when the message length is not known. In case 1) the message is sent as a single segment starting with the message length after padding "msg_len_ap" in 32-bit words concatenated with a '1' followed by the message length before padding "msg_len_bp" in bits followed by the message. The "msg_len_bp" is needed by several algorithms even when the message is already padded. In case 2) the message can be processed in segments $seg_0, seg_1, \cdots, seg_{n-1}$. Each segment $seg_0, \cdots, seg_{n-2}$ is headed by the segment length after padding "seg_len_ap" concatenated with a '0' followed by the segment of the message. The last segment seg_{n-1} follows the format of case 1). It contain a block of the message and must contain all padding. The formulae to compute the total number of bits before padding and after padding are:

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

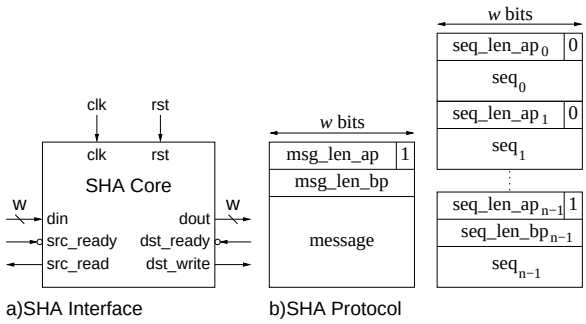


Figure 2.1: Interface and Protocol.

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

Furthermore in order to conserve logic resources needed for message counters we limit the total amount of data in a single message to 2^{32} bits i.e. 4 Gbits which we believe is sufficient for lightweight applications.

2.4 Optimization Target and Design Methodology

The design criteria is to extract maximum throughput for a given area budget. Here the budget is fixed at 500 slices with one BRAM on a Spartan-3 device.

Design Methodology:

Datapath: The key decisions to make are the datapath size, the number of processing elements to be used, number of pipeline stages to be introduced, etc. The datapath size is influenced by the natural size of the processing elements in the algorithm like adders, etc. For compact designs, the two important design rules which results in area reduction are folding and re-using of processing elements. Folding can be either vertical or horizontal. Vertical folding reduces the datapath width while horizontal folding reduces the size of

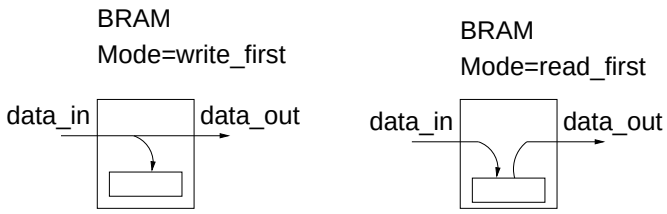


Figure 2.2: Block RAM.

the processing elements. Not all algorithms are flexible enough for folding. For example, BMW is not a right candidate for folding. In such cases the only other option to reduce its area is re-using the processing elements. But by re-using the elements, the clock cycle count rises which in turn affects the throughput. Interleaving operations by introducing pipeline registers helps to some extent in reducing the clock cycle count and boasting the throughput. This feature helps Grøstl to a great extent

Block Ram: In lightweight designs, the datapath sizes vary from 16 to 64 bits. The huge state sizes (>= 512) need to be stored somewhere from which data of smaller sizes are accesed, processed and stored back. Block RAMs (BRAMs) offer a large amount of memory space for storage but have a limited number of ports and I/O lines. Xilinx Spartan 3 BRAMs can be configured as single or dual port memories with a maximum data width of 64 bits or 32 bits per port, respectively. The Spartan-3 BRAM data sheet specifies that data is written to the address applied in the current clock cycle, but read from the address of the previous clock cycle. The new Xilinx Spartan-6 and Virtex-6 devices allow for independent read and write addresses for 64-bit data width. Use of BRAM does not increase the slice count as it is a dedicated resource of the FPGA.

Controller: Designing a Controller is in fact more challenging than the data path especially for Low area designs. A pure Finite State Machine (FSM) type controller occupies

huge area and is most of the times greater than that of the datapath. So FSM based controller is not at all an option for Lightweight designs. To reduce the area of the controller, the number of control signals should be reduced. In order to do that, first the signal transitions of all the control signals are noted and patterns and dependencies should be observed. Sometimes, two or more signals follow the same sequence and hence can be reduced to one. Also, there can be regular patterns in the signal transitions. So a single pattern can be produced and is repeated according to the requirement. The total number of states is reduced and this results in less state transitions. The address signals to the memories and other control signals are generated from memory words and control words which are implemented as ROMs. A combination of FSM with few states (< 10) and memory words and control words of reduced number of signals will significantly reduce the area.

2.5 Power Measurements

2.5.1 Introduction and Previous work

The cryptographic implementations in embedded devices are limited by the power consumed by them. There are many results published so far on the SHA-3 candidates both on software and hardware platforms. The hardware performance is charecterized by speed, area and also power. Power consumptions of all the round 2 candidates on software platform is performed by Benedict [27]. But there are no reported results on the power consumption of the SHA-3 candidates on FPGAs.

Hence, it would be interesting to look at the power consumptions of the SHA-3 finalists. Since we are dealing with lightweight implementations which consume less power compared to the high speed designs, the power consumption of these designs give us an insight about the minimum power requirements of these hash functions when implemented on FPGAs.

Even though there are power estimation tools which provide the power information about the designs, their efficiency is still questionable as reported by Meintanis [23]. To

achiveve more accurate results, physical onboard measurements is the only option. Lately, there is significant amount of work done on the power measurement methodologies on an FPGA board. The work done by Jevtic et all[16] clearly explained a general methodology to measure power on an FPGA board.

In this work, designs of SHA-3 final round functions implemented targeting low area were considered and their power is measured by following a methodology based on [16] described in section 2.5.3.

2.5.2 Design of SHA-3 Finalists

The following are the hash functions selected for the final round by NIST.

BLAKE

Grøstl

JH

Keccak

Skein

All the five SHA functions were designed within the budget of 500 slices and one BRAM targeting the low cost FPGA Spartan 3E. A uniform interface is followed while designing these hash functions.

2.5.3 Measurement methodology

Power measurement on FPGA is not a straight forward task. There are many components on the FPGA board which consume power and have to be separated from the power consumption of the core FPGA. So the core FPGA is powered separately using an external power supply. The next step is to differentiate various power components from the total power consumed by the hash function. The actual logic power of a particular hash function in the core can be obtained by eliminating the static power, clock power and interconnect (routing) power of the core FPGA. Therefore, it is important to find ways to measure the

static power, clock power and interconnect power of the core. The logic or dynamic power is dependent on the switching activity. For different input messages, the overall switching activity would vary which obviously effects the power consumption. The measurement method should be in such a way that this factor is minimal. So measurements are done for multiple input messages including messages with all zeroes and all ones.

All hash functions have the following main states.

Ideal state

Loading state

Processing state

Writing state

In ideal state the hash core is ready and waits for the message. In loading state, the message block is loaded from the FIFO. In the processing state, the message is processed and intermediate hash is generated. If there are multiple message blocks to be processed, the next state would be again loading state. If the block already processed is the last block, then the next state would be writing state.

Static Power: First the design is loaded into the Spartan 3E FPGA and the power is measured without giving the clock and the input vectors.

Clock Power: In this case, the clock signal along with other input vectors is provided and the system is kept in reset state with the clock signal running. Static power subtracted from the power consumed during this state gives us the clock power.

I/O power: When the message is getting loaded into the core FPGA from the control FPGA, the power consumed is the I/O power.

Total Power: When the Core FPGA starts processing the message, the power consumed is termed as the Total Power. This includes both static power, clock power, interconnect and the logic power.

Dynamic power = Total power - Static power - clock power .

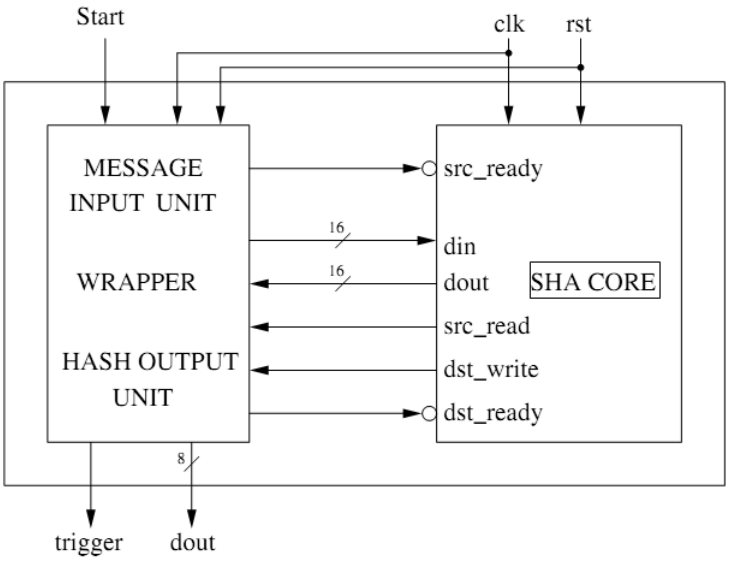


Figure 2.3: Interface

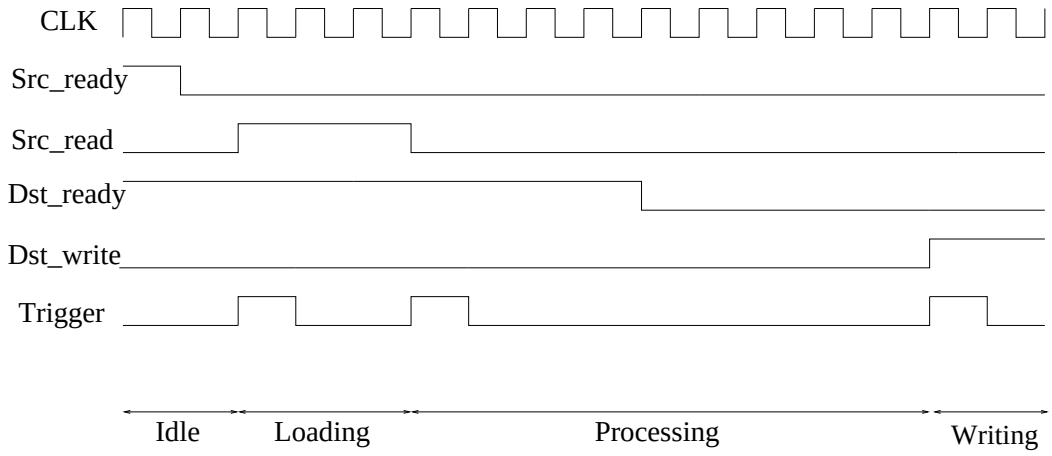


Figure 2.4: Signal Transitions and Trigger output

Interconnect Power: The other factor which effects the power consumption on the FPGA is the interconnect power which is quite complex to extract. One way of extracting this power is by asking the tools to implement the design with very tight area constraints and with no constraints.

The Dynamic power(includes interconnect power) obtained from the implementation with area constraints is subtracted from the one which is implemented with no area constraints. By taking multiple measurements with different area constraints and subtracting from each of them, the dynamic power obtained from the design implemented with tight area constraints, a number of readings will be obtained for the interconnect power and the average of that will give us an approximate value of the interconnect power. A more efficient way of generating the interconnect power is described in [16] but it is too complex to implement here and will be done in future.

Logic Power: This is the power consumed due to the logic elements of the design.
Logic Power = Dynamic power - interconnect power.

In this work, only the dynamic power is measured and the interconnect power is neglected. So logic power is asssumed to be the same as the dynamic power.

Fig 2.3 shows the interface followed for the measurement. The wrapper talks to the SHA core with the help of the control signals and sends input messages to it. It also receives the message digest obtained from the SHA core after processing the message blocks. The wrapper consists of a BRAM and a Finite State Machine. The BRAM has the message initialized in it and stores the 256 bit hash digest sent by the core.

The signals rst, start and trigger are the external I/O signals available for the user. Rst and start are the input signals and trigger is the output signal. The reset (rst) signal is used to reset the entire system and the clock signal is the only one which toggles in this scenario. The start signal is used to control the processing of the SHA core. When the start signal is enabled, the SHA core loads the message from the wrapper and starts processing.

XPOWER: XPOWER of Xilinx allows us to estimate the total dynamic power consumption of the design. The XPOWER tool needs the following files .ncd, .pcf and .vcd. Firstly, post place and route simulation of the design is run and a VCD file is generated using the Modelsim simulator. When the tool (Xilinx ISE) launches the simulator Modelsim for simulation, in the command line prompt of Modelsim, the following instructions are executed to get VCD file.

```
vcd file filename.vcd
vcd add -r /*
run 500 ns
vcd checkpoint
quit -f
```

The run time(here 500 ns) is based on the number of clock cycles the hash function takes to process the message block. The total clock cycles multiplied by the time perid gives the run time. Here clock period is taken as 20 ns as the values are compared to measured values on FPGA run with a clock of frequency 50MHz. The VCD (value change dump) file

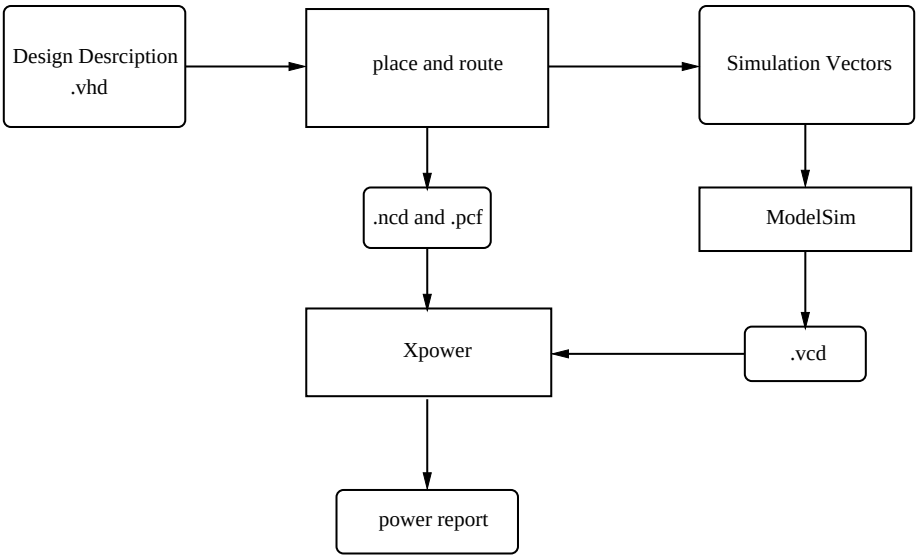


Figure 2.5: Xpower design flow

thus generated has all the switching information of all the signals in the design throughout the run. All the required files are loaded and the output file from the tool is a detailed power report.

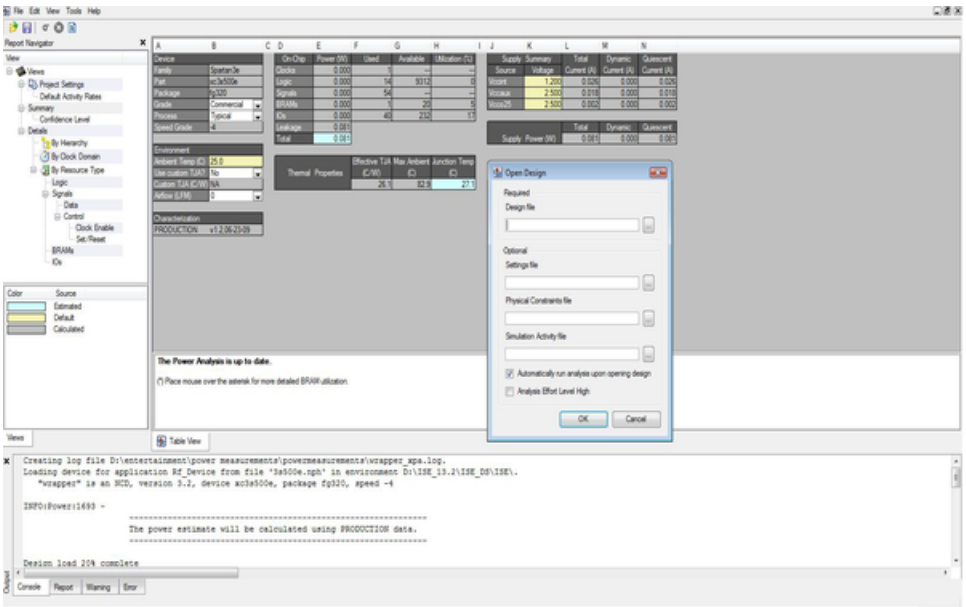


Figure 2.6: Xpower tool from Xilinx

Table 2.1: State of control signals and the accoiciated power

Control signals	Power
No input signals	Static Power
Rst = '1'	Clock Power
Rst = '0', Src_ready = '0' & Src_read = '1'	I/O Power
Rst = '0', Src_ready = '0' & Src_read = '0'	Total Power
Dst_ready = '0' & Dst_Write = '1'	I/O Power

Chapter 3: Lightweight Implementations of 4 round two Candidates

3.1 AES round

A round of AES consist of SubBytes, ShiftBytes, MixBytes and AddConstant operations. Basic round functions of few SHA-3 candidates like Grøstl and SHAvite-3 are based on AES round. The input is arranged in the form of a matrix which is called as the state matrix and the above operations are operated on this matrix.

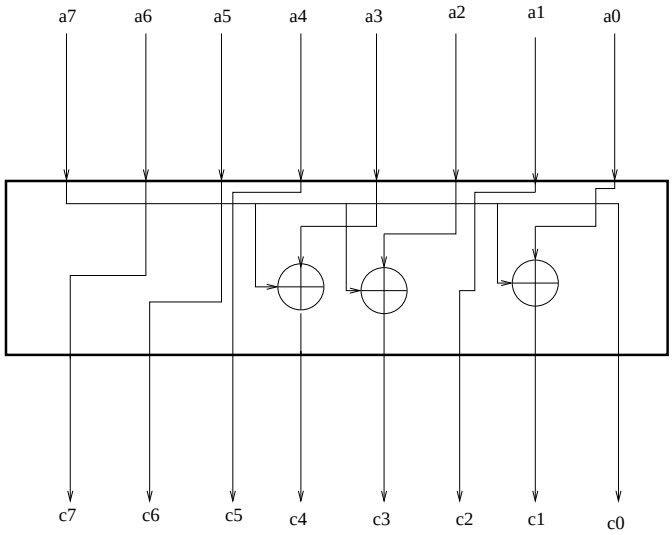
Operations in Galois Field: The coefficients of a polynomial are equal to the respective bits of the binary representation. In AES, multiplication in GF(2^8) is achieved by multiplying the corresponding polynomials modulo a fixed irreducible polynomial. Here the irreducible polynomial is m(x) = $x^8 + x^4 + x^3 + x + 1$.

Galois Field Multipliers:

x02: Polynomial representation of 2 is x. Let us assume an 8 bit number represented by $a_7a_6a_5a_4a_3a_2a_1a_0$ where $a_7, a_6, a_5, a_4, a_3, a_2, a_1$ and a_0 are the corresponding bits of the byte which is being multiplied. The polynomial representation of the byte is $a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x^1 + a_0x^0$. After multiplication with x, the polynomial becomes $a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x^1$ which is reduced to $a_6x^7 + a_5x^6 + a_4x^5 + (a_3 + a_7)x^4 + (a_2 + a_7)x^3 + a_1x^2 + (a_0 + a_7)x^1 + a_7$ using the irreducible polynomial m(x)

x03: Since (03) = (02) + (01)

$a * (03) = a * (02) + a$



*02

Figure 3.1: Galois field multiplier (02)

x04: Polynomial representation of 04 is x^2 . This operation can be achieved by

$a * (04) = (a * (02)) * (02)$

x05: (05) = (04) + (01)

$a * (05) = a * (04) + a$

x07: (07) = (04) + (03)

$a * (07) = a * (04) + a * (03)$

SubBytes: SubBytes is a non-linear substitution in which each byte is replaced with another based on a look up table. SubBytes composed of two basic operations, Multiplication Inversion in the galois field(2^8) with the reduction polynomial m(x) and affine transformation over GF(2).

LUT Implementation: Sbox (SubBytes) can be described as a 256x8 bit look up table which is implemented as ROM. The input is given to the 8bit address of the ROM and the output comes from the databus. The Sbox look up table is of size 256 bytes.

Logic Implementation: The complex function in GF(2^8) can be decomposed into GF(2^4) which can be further decomposed to GF(2^2) and this in turn to GF(2). All the functionalities in GF(2) can be implemented using XOR(addition) and AND(Multiplication).

Spartan-3 Vs Virtex-5 Each Sbox LUT implementation takes 64 slices on a Spartan-3 device and just 8 slices on Virtex-5. This is due to the LUT structure of Virtex-5. The logic implementation takes 32 slices on Spartan-3 device and 16 on Virtex-5. So a logic implementation is advantageous on Spartan family but a LUT based implementation on Virtex-5. But the drawback with logic implementation is that the delay is on the higher side.

ShiftBytes: Each row of the state matrix is shifted towards left . The shift is different for each row. The shiftByte operation is simple routing if the entire state is being operated else is taken care by the way the state, stored in a memory, is addressed.

MixBytes: The state matrix is multiplied (GF(2^8)) with an MDS matrix to produce a new matrix. The first row of the MDS matrix is multiplied with the first column of the state matrix to produce byte0 of the first column in the resultant matrix. Similarly, the second row multiplied with first column produces the byte1 of the first column in the resultant matrix and so on. The MixBytes can be implemented with two layers of Xor gates, the first for GF multiplication of the individual bytes and the second, the Xor of all the resultant bytes of the GF multipliers.

AddConstant: The AddConstant is a byte by byte xor operation of the state matrix with a constant matrix.

3.2 BMW

3.2.1 BMW Algorithm description

Notations

H	Double Pipe
Q	Quadruple Pipe
$H^{(i)}$	i-th double pipe value
$Q^{(i)}$	i-the quadruple value
$H_j^{(i)}$	j-th word of i-th double pipe
$Q_j^{(i)}$	j-th word of i-th quadruple pipe
$Q_a^{(i)}$	The first 16 words from $Q^{(i)}$
$Q_b^{(i)}$	The last 16 words from $Q^{(i)}$
m	number of bits in a Block
$M^{(i)}$	Message block i, of m bits
$M_j^{(i)}$	j-th word of i-th Message block
XL, XH	Temporary words used in computation of double pipe
$CONST^{final}$	16 constant words used in the finalization round

Description

BMW [14] uses there functions f0, f1 and f2 in computing the hash. The first function f0 takes in a message block $M^{(i)}$ and the previous double pipe $H^{(i-1)}$) and computes the first part of quadruple pipe $Q_a^{(i)}$. The second function f1 computes the second part of the quadruple pipe $Q_b^{(i)}$ by taking the first 16 words of the quadruple i.e $Q_a^{(i)}$ along with the message block $M^{(i)}$ and the previous double pipe $H^{(i-1)}$). The final folding function takes in the message block and the complete quadruple $Q_a^{(i)}$ and $Q_b^{(i)}$ to produce the new double pipe $H^{(i)}$. The finalization round of BMW also use the same three functions with different

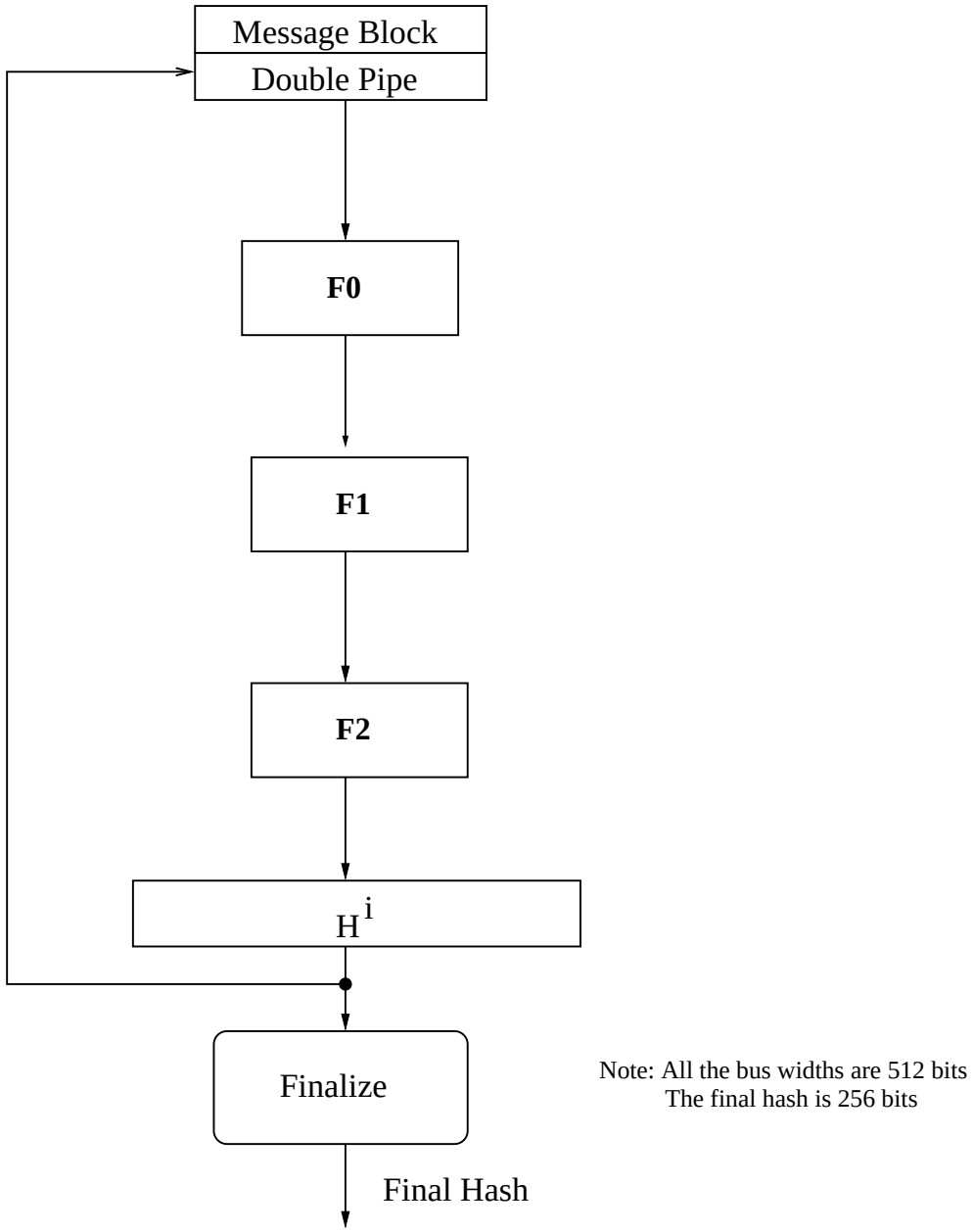


Figure 3.2: BMW compression function.

inputs to the functions. They use the final douple pipe H^N produced after processing the final N^{th} block and $CONST^{final}$. The final hash value H^{final} is produced from the folding function f2 in the finalization round. The least significant 256 bits of H^{final} is message digest of the message M.

For i = 1 to N

{

$Q_a^{(i)} = f_0(M^{(i)}, H^{(i-1)})$

$Q_b^{(i)} = f_1(M^{(i)}, H^{(i-1)}, Q_a^{(i)})$

$H^{(i)} = f_2(M^{(i)}, Q_a^{(i)}, Q_b^{(i)})$

}

The finalization is done according to the following set of functions.

{

$Q_a^{final} = f_0(H^{(N)}, CONST^{final})$

$Q_b^{final} = f_1(H^{(N)}, CONST^{final}, Q_a^{final})$

$H^{final} = f_2(H^{(N)}, Q_a^{final}, Q_b^{final})$

}

3.2.2 BMW lightweight implementation

BMW is a single round hash function. It takes in 32 message words and 32 initialization words and produces 32 quadruples. The first 16 words of the quadruple are obtained from F0 and the remaining 16 words of the quadruple are obtained from F1. The function F2 uses 32 quadruples obtained from F0 and F1 along with the message to produce the intermediate hash. The function F0 consists of multi operand additions/subtractions and xor operations. The function F1 has multioperand xor operations,multioperand additions/subtractions and variable rotations and shifts in both directions. The third function F2 is again based

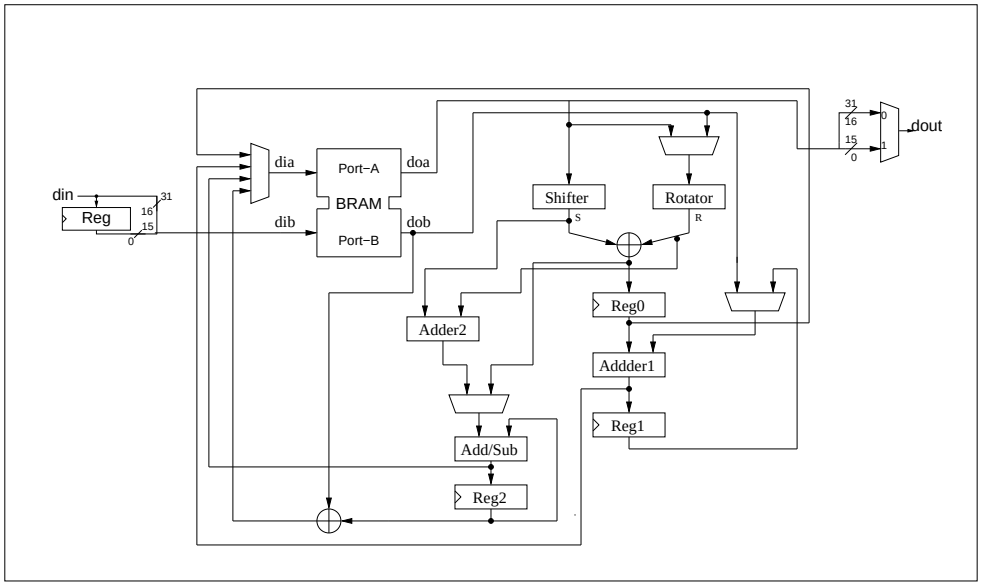


Figure 3.3: Lightweight implementation of BMW

on multioperands additions/subtractions, multioperand xors , variable rotations and shifts. The BRAM stores the entire state and the initialization vector and the input message block. Two words are read from the BRAM in each clock cycle and one of the above operations will be done on the two operands and will be stored in a register if it is a multioperand operation or stored back into the BRAM. The shifter and rotator are implemented separately to generate both the shift and rotate operations at the same time which improves the performance a lot. It takes 636 clock cycles to process a single block of message. The control logic for BMW is relatively huge as there is no proper sequence followed in the address bits of the BRAM.

3.3 Grøstl

3.3.1 Grøstl Algorithm description

Notations

M	Input Message
IV	Initialization vector
P & Q	State matrices formed from message block and intermediate hash
h_i	Intermediate hash after round i
m_i	Input message block in round i
f	Compression function
fn	finalization function

Description

Grøstl [13] is an AES based algorithm which iterate the compression function f as follows. h_i = f(h_{i-1}, m_i) for i = 1,2, . . . ,n.

The input message M is split into 512-bit blocks m_1, m_2, m_3,. . . , m_n . Two 8X8 state matrices P and Q are formed using the message block m_i and the initialization vector IV. The P matrix is formed from the 512 bits obtained from the xor operation of the message block m_i and h_{i-1}(or IV) where as the Q matrix is formed from the message block m_i itself. Ten rounds of AES operations are performed on both P and Q after which the results are xored with h_{i-1} to form the new intermediate hash h_i.

h_i = P(h_{i-1} xor m_i) xor Q(m_i) xor h_{i-1}

The finalization round fn is operated on the h_n . Once again 10 rounds of AES operations are performed on the matrix formed from h_n abd the result is xored with the input h_n to generate the final hash. The final hash is the least significant 256 bits of the final result.

hash = fn(P(h_n) xor h_n)

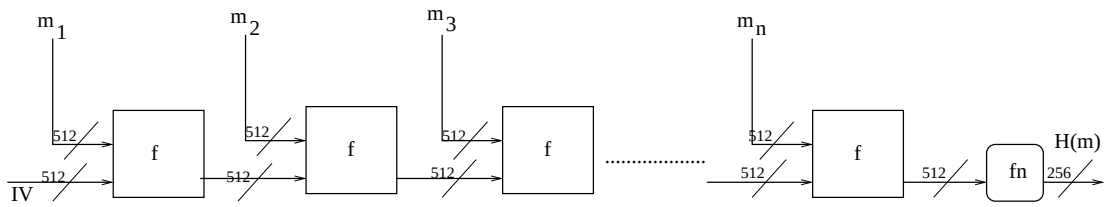


Figure 3.4: Grøstl hash function.

The following round transformations (AES operations) are performed in each round on both P and Q.

AddRoundConstant

SubBytes

ShiftBytes

MixBytes

The AddRoundConstant adds a round dependent constant to the state matrix P (and Q). The round constants are also matrices of the same size as of the state matrix. SubBytes transformation substitutes each byte in the state matrix by another byte from the S-box which is the same as the AES s-box. ShiftBytes shifts the bytes within the row towards left by a certain value. This value is fixed and different for each row. In MixBytes, each column of the state matrix is multiplied with a constant 8X8 matrix in the finite field F_{256} which is defined the same way as the AES.

The AddroundConstant matrix is different for P and Q . Also in the ShiftBytes operation, the number of shifts within the row is different for P and Q.

3.3.2 Grøstl lightweight implementation

The initialization vector is stored in a DRAM and is also used to store the intermediate hash produced after processing each block. Two S-boxes described as ROMs are used and a half GF multiplier is implemented. The message block is xored with IV from the DRAM

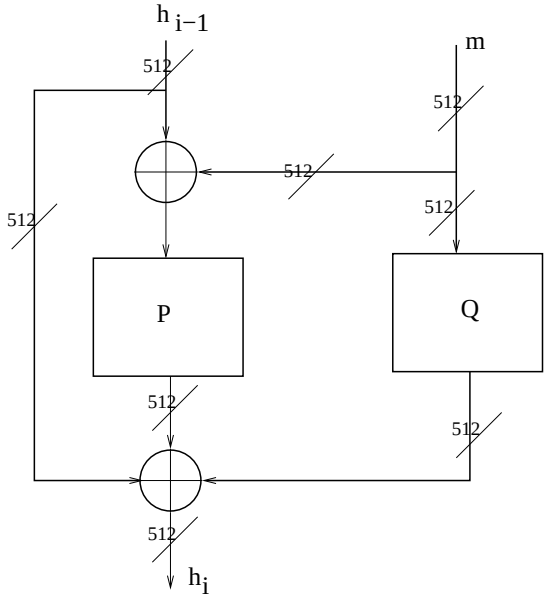


Figure 3.5: Grøstl compression function f.

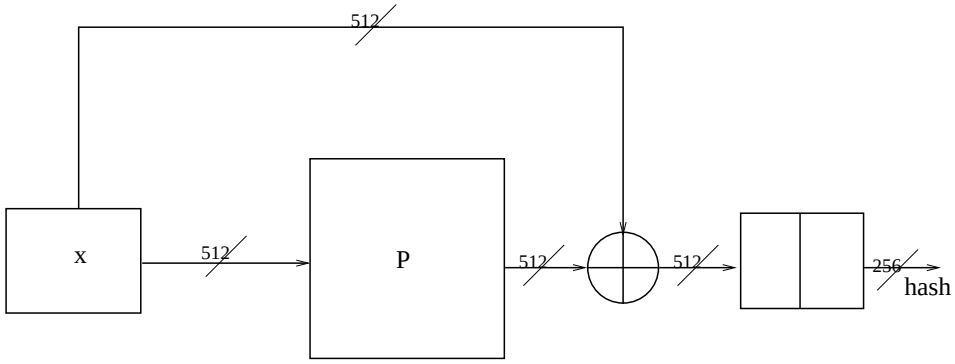


Figure 3.6: Grøstl final round function fn.

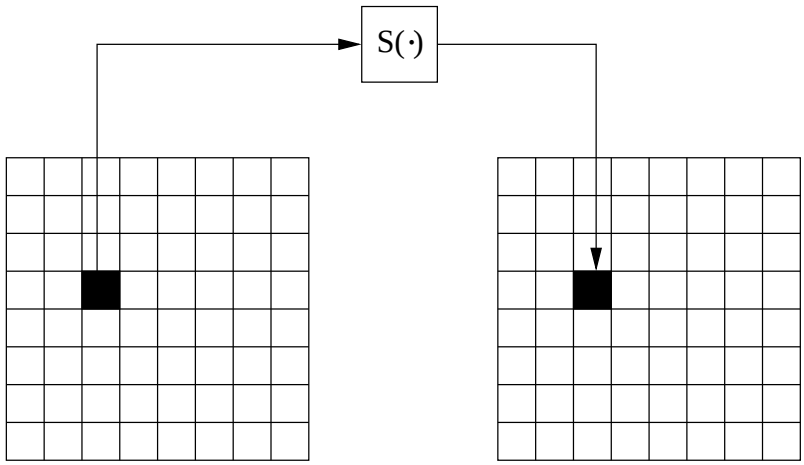


Figure 3.7: Grøstl SubBytes.

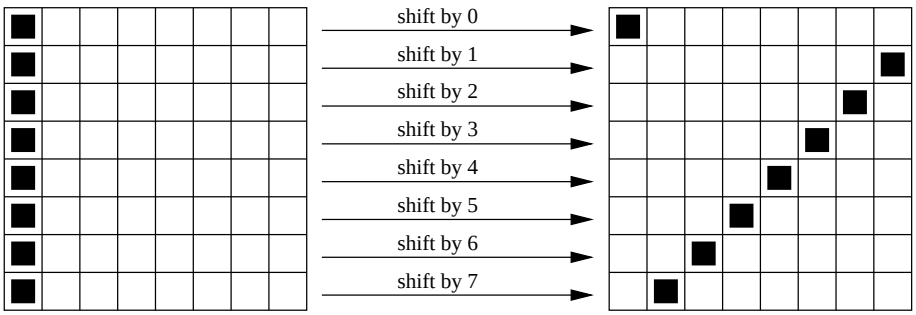


Figure 3.8: Grøstl Shiftbytes.

$$C_P[i] = \begin{bmatrix} i & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \end{bmatrix} \quad \text{and} \quad C_Q[i] = \begin{bmatrix} 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ 00 & 00 & \cdots & 00 \\ i \oplus ff & 00 & \cdots & 00 \end{bmatrix}$$

Figure 3.9: Grøstl Constants.

and is loaded into the BRAM through one port(to form P) and the message directly loaded through the other port of the BRAM(to form Q). First two 32-bit words are accessed from the BRAM, from which only 16-valid bits are available which are stored in a register. This is due to the ShiftBytes operation involved. The AddConstant and the SubBytes operations are performed before loading the 16-valid bits into the register. The constants are generated by the controller depending on the 16-bit data accessed which then pass through two S-boxes before the registers. In the same way, the remaining bits are accessed from the memory and stored into the respective registers which now hold one column. The column goes as input to GF multiplier and produces the first 32-bits of the new column in one clock cycle and the remaining 32-bits in the next clock cycle. The P matrix and Q matrix are executed serially using the same AES resources.

One new column is produced for every 7 clock cycles. Each round of P & Q computes 16 new columns which takes 112 clock cycles. The xor operation after the 10 the round is implemented on the fly and hence takes 0 clock cycles. Thus a block of message is processed in 1120(16 * 7 * 10) clock cycles.

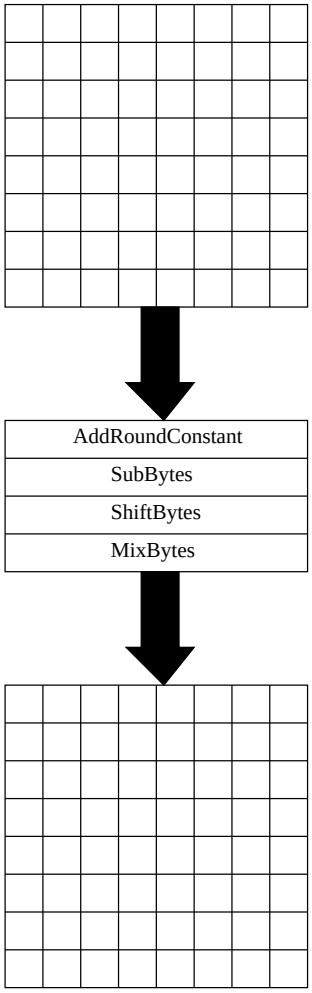


Figure 3.10: Grostl Round Function.

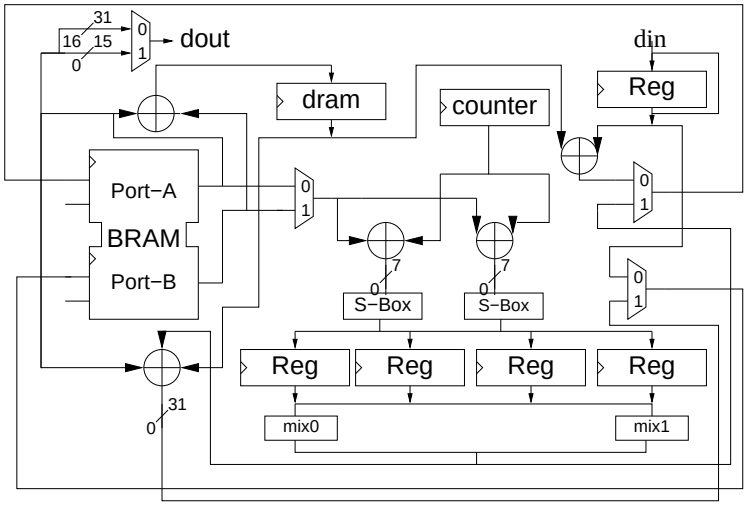


Figure 3.11: Lightweight architecture of Grøstl.

3.4 Luffa

3.4.1 Luffa Algorithm description

Notations

$M^{(i)}$	Message Block in i-th round
Q_j	The permutation dealing with j-th block
$H_j^{(i)}$	j-th intermediate hash vector in i-th round
X_j	j-th output block from Message Injection
$a_j^{(r)}$	j-th word of X in round r

Description

The round function in Luffa[7] consists of Message Injection (MI) and Permutation which is further divided into three sub-permutations Q_0, Q_1 and Q_2. The message Injection Function can be represented using Matrices. The initialization vectors or the intermediate

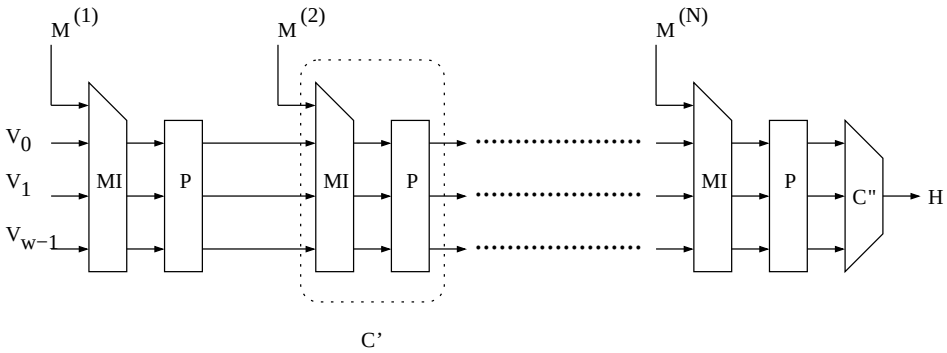


Figure 3.12: Luffa hash function

hash vectors along with the message goes as input to MI and produces three 256 bit vectors X_0, X_1 and X_2. The output from the Message Injection X_j(j = 0,1,2) is divided into 8 32-bit words $b_0, b_1, \ldots b_7$ which go as inputs to the Permutation Q_j. Each permutation Q_j consists of an input tweak which is applied once and a step function, iterated 8 times.

Tweak The last four input words to Q_j are rotated j bits to the left.

The main iterative function in Luffa is the Step Function. The step function consists of the Sub Crumb, Mix Word and Add Constant functions.

SubCrumb The SubCrumb Function takes in 256 bits as 8 32-bit words $a_0, a_1, .a_7$ and produces 8 new words $x_0, x_1, \ldots x_7$. It consists of 64 4-input s-boxes with 32 S-boxes in each SubCrumb slice unit. The first four words a_0, a_1, a_2, a_3 were given to one Subcrumb slice and the remaining four words are given to the other unit. Each S-box takes in 4 bits, one from each word, a_3, a_2, a_1 and a_0 (or a_4, a_7, a_6 and a_5) and produces 4 new bits using the following look up table.In this way 8 new words are produced from SubCrumb which are given to MixWord.

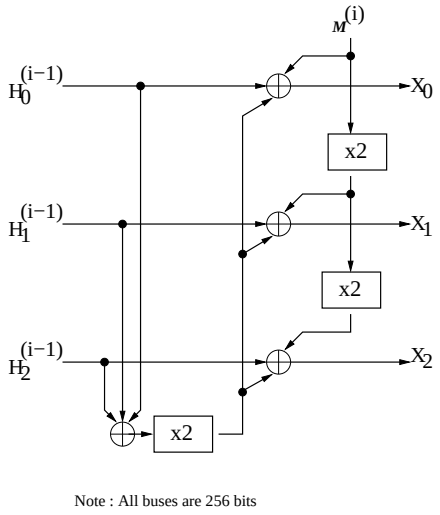


Figure 3.13: Message Injection Function

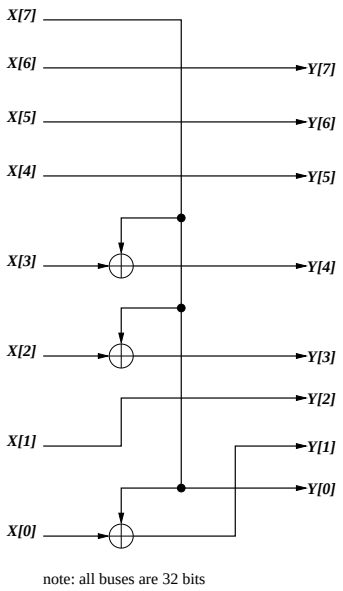


Figure 3.14: x2 Multiplier ($GF(2^8)^{32}$)

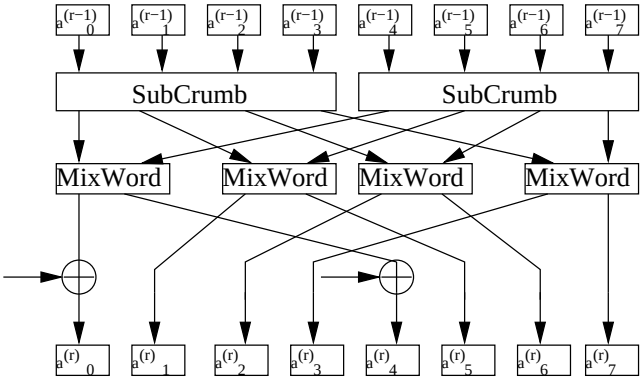


Figure 3.15: Luffa Step Function

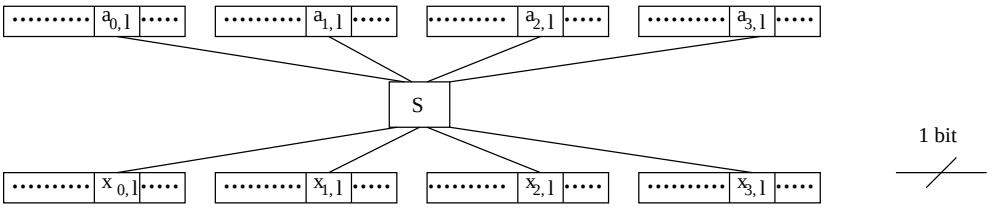


Figure 3.16: Luffa SubCrumb

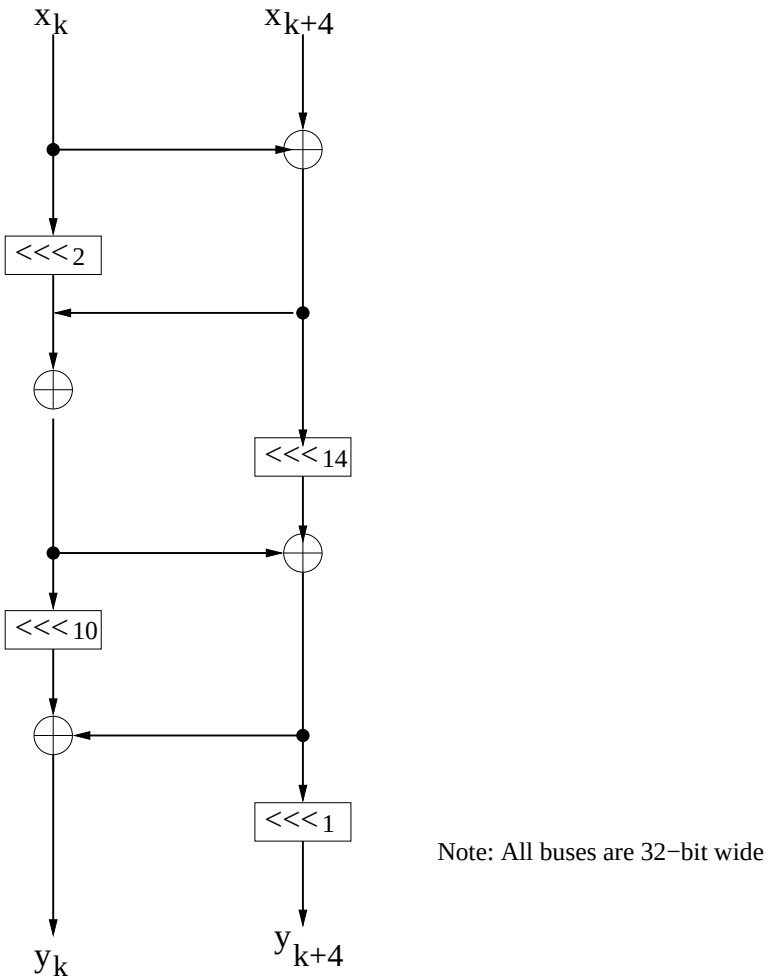


Figure 3.17: Luffa MixWords.

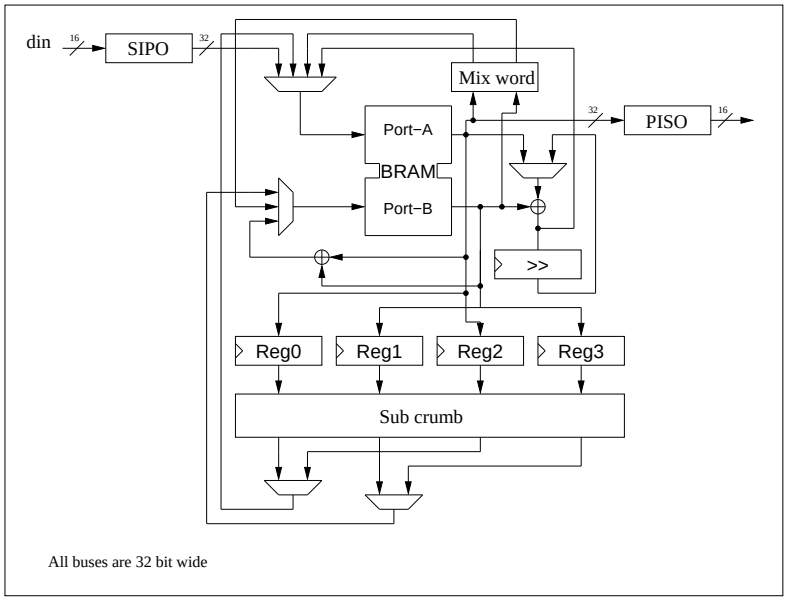


Figure 3.18: Luffa lightweight architecture.

MixWord The MixWord takes in two words x_k and x_{k+4} and produces two new words y_k and y_{k+4} (k = 0,1,2,3) by going through the following transformations. Four MixWords are used to process all the 8 words and produce new words $y_0, y_1, \ldots y_7$.

AddConstants Two words, y_0 and y_4 are xored with the constants generated from the constant generator . The Constant generator generates two constants in each round using a function fl which is an LFSR of Galois configuration.

3.4.2 Luffa lightweight implementation

The message injection function has xor and x2 operations in it. The x2 galois field multiplication is nothing but shifting of words with couple of xor operations. Thus the entire message injection function is basically a series of xor operations which can be implemented with an XOR gate,a MUX and a register. Two words are read from the Block RAM and are xored and the result is stored in a register which is used as one of the inputs for the

next xor operation. This approach is used for multi Operand xor operations. If it is a single xor operation, the words are read and the result after the xor is again stored back into the BRAM at the appropriate location. The Step function is implemented with one SubCrumb and one MixWord. First , 4 words are accessed from the BRAM and stored in the 4 registers which are then send to the Subcrumb and the output from the SubCrumb is stored back into the BRAM. The remaining 4 words are stored into appropriate registers which takes care of the order of the inputs to the Subcrumb. MixWord is operated on two words at a time and the result is stored back into the memory. The addconstant operation is also implemented the same way. Instead of the constant generator, the pre-calculated constants are stored in a ROM and are used in the Addconstant stage in each iteration. The tweak which effects permutation q1and q2 in the first iteration is implemented by a 3x1Mux. The Mux takes in 3 words , in which the second and third word are the rotated versions of the first word, rotated by 1 and rotated by 2 respectively. The finalization round is also implemented the same way using the same resources.

The Message Injection function is executed in 78 clock cycles.Each permutation Qj takes 22 clock cycles which results in 66 clock cycles for one iteration of the step function. The total number of clock cycles to process one block of message are $78+(8*66) = 606$

3.5 SHAvite-3

3.5.1 SHAvite-3 Algorithm description

Notations

M	Input Message
m_c	chaining value or Initialization vector
k_i^j	Subkey j in i-th round (j = 0 ,1 ,2)
rk[i]	i-th 32-bit keyword
cnt0,cnt1	32-bit words formed from the 64-bit counter

Description

SHAvite-3 [6] is another AES based algorithm with compression function c_{256}. The compression function has an underlying block cipher E^{256} which is a 12 round Feistel block cipher. The block cipher E^{256} accepts 256 bits treated as eight 32-bit words P[0,...,7] which are divided into two halves L_0 and R_0. L_0 contains words P[0,1,2,3] and R_0 contains P[4,5,6,7]. Then the following round function is repeated 12 times.

$(L_{i+1}, R_{i+1}) = (R_i , L_i \text{ xor } F^3_{RK_i}(R_i))$

$F^3(.)$ accepts an input of 128 bits, R_i and 384-bit subkey, $RK_i = (k_i^0 , k_i^1 , k_i^2)$ and three rounds of AES are applied on them.

$F^3_{(k_i^0,k_i^1,k_i^2)}(\text{x}) = AESRound_0 128(AESRound_{k_i^2}(AESRound_{k_i^1}(\text{x xor } k_i^0)))$

The output cipher text C = ($L_1 2$, $R_1 2$) is the intermediate hash or the final hash depending on which block it is processing. There is no finalization round in SHAvite-3.

The message expansion algorithm of C_{256}(the key schedule algorithm of E_{256}) takes in the message block, a 256-bit salt and a 64-bit counter to produce the 36 subkeys(or 144 32-bit words rk[i]). The first 4 subkeys are generated directly from the message block and the remaining 32 keys are produced by a series of Non-linear and linear expansion steps.

The non linear process takes four rk[.] words and encrypts them using 4 words of the salt and the result is xored with four other words to produce the next 4 words(a new subkey). The linear process takes in 2 words and produces a new word by xoring them. First, 4 sub keys(or 16 words) are produced using Non-linear step and the next 4 are produced by linear step. This sequence is repeated 4 times to produce 32 subkeys.

Out of the 144 32-bit words produced from the message expansion step, 8 words are xored with counter, either with cnt0 or cnt1.

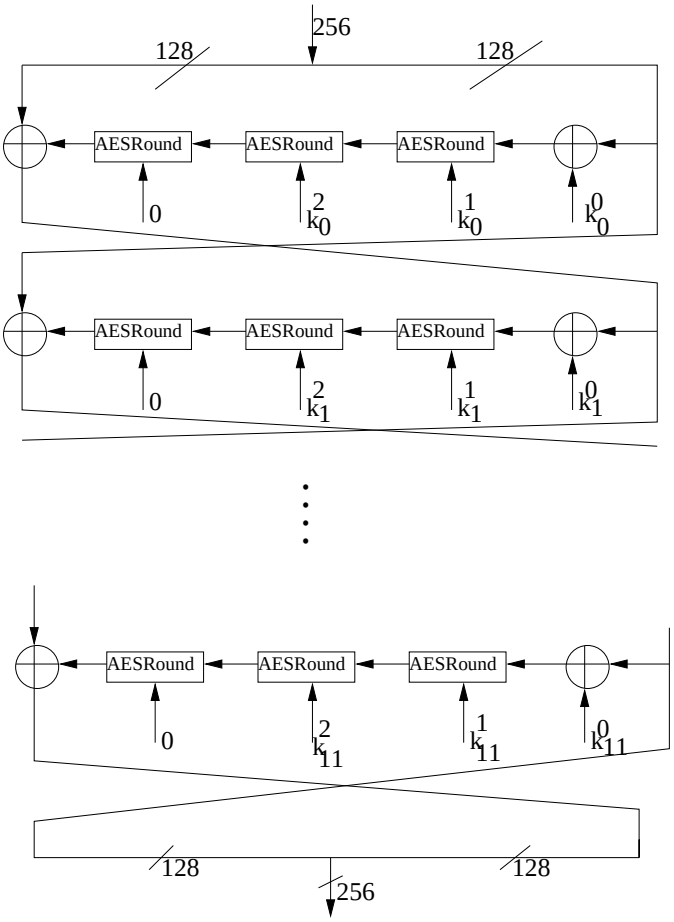


Figure 3.19: SHAvite-3 hash function

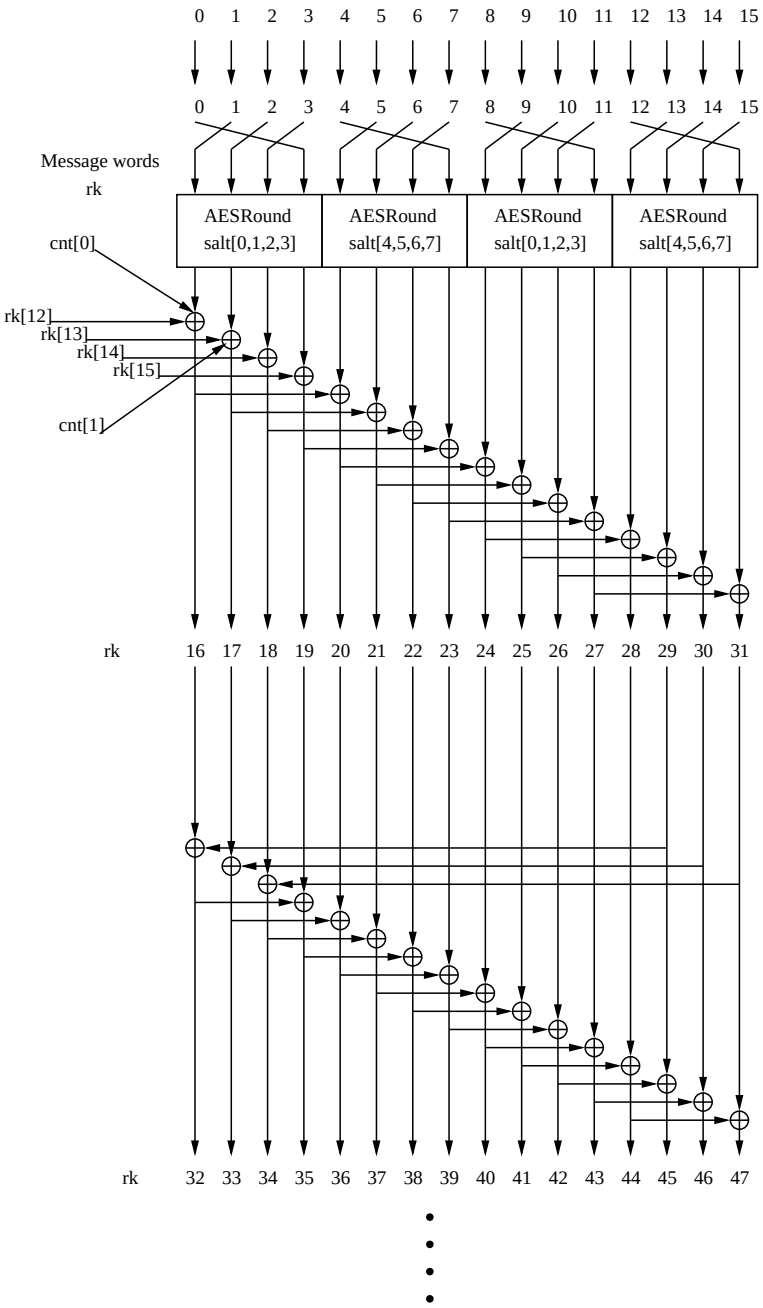


Figure 3.20: SHAvite-3 keygeneration unit

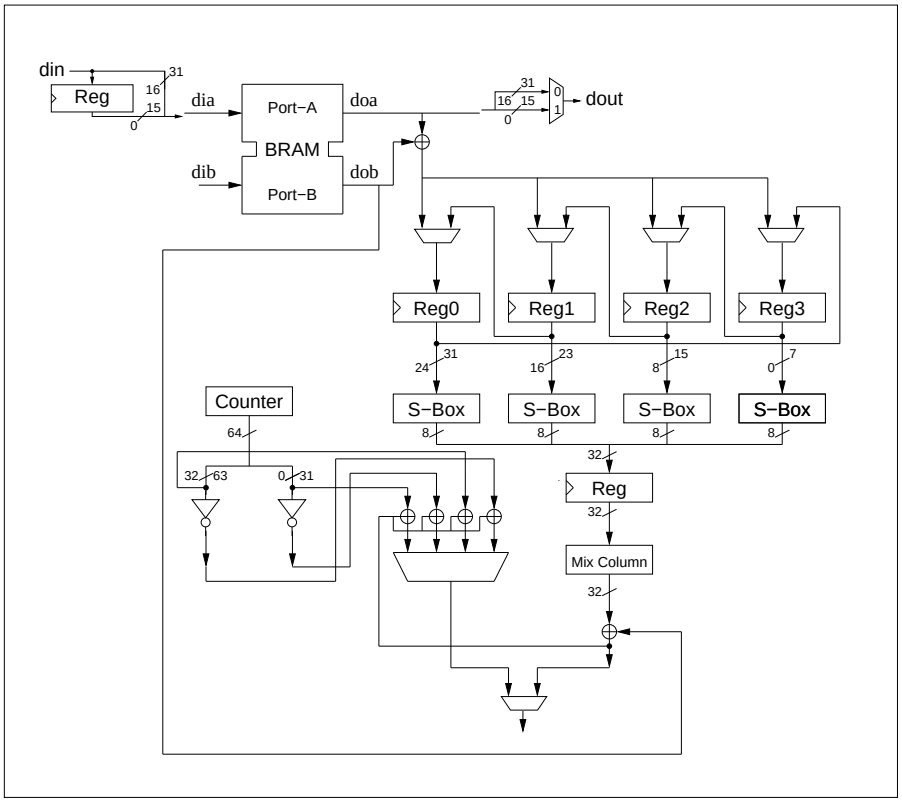


Figure 3.21: SHAvite-3 lightweight architecture.

3.5.2 SHAvite-3 lightweight implementation

SHAvite-3 has two major blocks. One is the key generation unit and the other is the processing unit. Both the blocks are based on AES rounds. A single AES round is implemented and first, the key generation operations are performed using the AES resources and the keys are stored in the BRAM. Then processing operations are performed using the keys stored in the BRAM. A 128 bit input is organized in a 4x4 array and the AES operations are performed on it. The 128 bits are stored in 4 registers from which a byte from each register is given to the four S-boxes for the sub-byte operation. The first column is formed by taking in byte0 from reg-0, byte1 from reg-1, byte2 from reg-2 and byte-3 from reg-3. Then the words are rotated within the registers and the same set of bytes forms second column. In

Table 3.1: Throughput formulae for implementations of the four SHA-3 candidates

Algorithm	Speci-fication	Block Size (bits) b	Clock Cycles to hash N blocks $clk = st + (l + p) \cdot N + end$	Throughput $\frac{b}{(l+p) \cdot T}$
BMW	[14]	512	$2 + (32 + 730) \cdot N + 757$	$512/(762 \cdot T)$
Grøstl	[13]	512	$2 + (32 + 1120) \cdot N + 577$	$512/(1152 \cdot T)$
Luffa	[7]	256	$2 + (16 + 606) \cdot N + 647$	$256/(622 \cdot T)$
SHAvite-3	[6]	512	$2 + (32 + 744) \cdot N + 17$	$512/(776 \cdot T)$

this way the shift row operation is taken care of and the required column is obtained which is sent to sub-bytes and mixbyte operations. The Sbox is implemented as a LUT and four such S-boxes are used. The MixByte has GF multipliers x2 and x3 and a has a total of 8 such multipliers combined so that it takes in a 32-bit column as input and produces new column at a go. The AddConstant is performed by the xoring the output word from the MixByte and the constant stored in the BRAM. The 64 bit counter is split into two halfs cnt0 and cnt1. Few selective words are xored with the counter values either with cnt0 /cnt1 or with the complement of cnt0/cnt1. The salts are assumed to be ZERO.

One 128 bit non linear key is obtained in 10 clock cycles and one 128 bit linear key is obtained in 8 clock cycles. So all the keys are produced in 288 clock cycles(10*16 + 8 *16). Each round of processing takes 38 clock cycles.The total number of clock cycles to process one block of message are 744(38*12 + 28).

Chapter 4: Results and Comparisions

4.1 Implementations results

The results are updated in table 4.1 and 4.2. All the implementations are under the area constraint of 400-600 slices. The designs are optimized for maximum throughput within this limit. The delays of Luffa and BMW are on the higher side. Introducing a pipeline register can improve the delay in Luffa. The large delay in BMW is due to the 32-bit adders used. The area of BMW is hugely influenced by the controller (Fig.4.4). Since BMW is a single round function, and there are no regular patterns in the memory addresses, the controller area is very huge. Luffa has the least area among the four, followed by SHAvite-3 on Spartan device. But the situation changes when it is targeted to Virtex device. SHAvite-3 has the least area among the four. This is due to the fact that the four S-boxes used in SHAvite occupy 256 slices(4 * 64) on Spartan-3 whereas it just takes 32 slices(4 * 8) on Virtex-5. Throughputs of algorithms which have finalization rounds are adversely affected for short messages.

4.2 Comparision with other groups

Firstly, it is to be noted that our primary design target is Spartan-3 device. So some of the algorithms perform worse when they are compared to designs on a different platform as they could not take advantage of that particular device features. The results from other groups are shown in table 4.3.

Our BMW outperforms the implementation by El Hadedy [8] by a significant margin. The throughput is much better then theirs as the message block is processed in less number

Table 4.1: Implementation Results of Lightweight Implementations of Grøstl , Luffa , SHAvite-3 and BMW

Algorithm	Xilinx xc3s50-5			Xilinx xc6slx4csg-3			Xilinx xc5vlx20-2			Altera ep2c5f256c6		
	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (LEs)	Memory Bits	Maximum Delay (ns) T
BMW	561	1	9.99	183	1	7.15	233	1	5.16	1,104	8,192	9.45
Grøstl	483	1	11.42	148	1	7.66	183	1	4.80	1,157	8,704	11.12
Luffa	474	1	10.17	107	1	6.86	176	1	5.08	946	8,192	7.66
Shavite-3	501	1	7.60	120	1	5.24	136	1	3.37	471	16,384	7.01

of clock cycles. The other implementation of BMW, again by El Hadedy [9] is primarily based on their previous design and this time published results targeting Virtex family. Both the designs take huge number of clock cycles to process a block. This is due to their rather simple design which has an ALU consisting of adders, shifters, rotators and xor gates. Each instant, two words are accesed from the memory and one of the above mentioned operations is performed and the result is again stored back into the memory. Our design have multiple adders and xor gates and also the shifter is separated from the rotater. So operations can be performed in parallel and this reduces the number of clock cycles by a huge margin at the cost of marginal increase in area.

There are no lightweight reults for Luffa targeting Spartan 3 Device. Our Luffa is better when compared to the lowest(in terms of area) reported result by Shugo [24]. Their smallest reported result on Virtex 5 family is 355 slices as against our 144 slices. But our throughput is better than theirs resulting in a very good throughput to area ratio. When compared to their next higher versions, the result is not promising. The performance of our luffa is effected due to the area constraint. Given a little more budget, its performance can be improved by implementing a full step function instead of the half that is implemented.

Table 4.2: Throughput Results of Lightweight implementations of Grøstl , Luffa , SHAvite-3 and BMW

Message Algorithm	Xilinx xc3s50-5				Xilinx xc6slx4csg225-3			
	Long		Short		Long		Short	
	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)
BMW	67.3	0.12	33.7	0.060	93.9	0.51	47.1	0.257
Grøstl	38.9	0.08	25.9	0.050	58.0	0.39	38.6	0.26
Luffa	40.5	0.09	19.8	0.042	60.0	0.56	29.4	0.275
Shavite-3	86.8	0.17	83.1	0.166	126.0	1.05	120.6	1.005

Message Algorithm	Xilinx xc5vlx20-2				Altera ep2c5f256c6			
	Long		Short		Long		Short	
	TP (Mbps)	(Mbps /slice)	TP (Mbps)	(Mbps /slice)	TP (Mbps)	(Mbps /LE)	TP (Mbps)	(Mbps /LE)
BMW	130.3	0.56	65.3	0.280	71.1	0.06	35.6	0.032
Grøstl	92.6	0.50	61.6	0.33	39.9	0.03	26.4	0.022
Luffa	81.1	0.46	39.7	0.225	53.8	0.06	26.3	0.028
Shavite-3	196.1	1.44	187.6	1.380	94.1	0.20	90.1	0.191

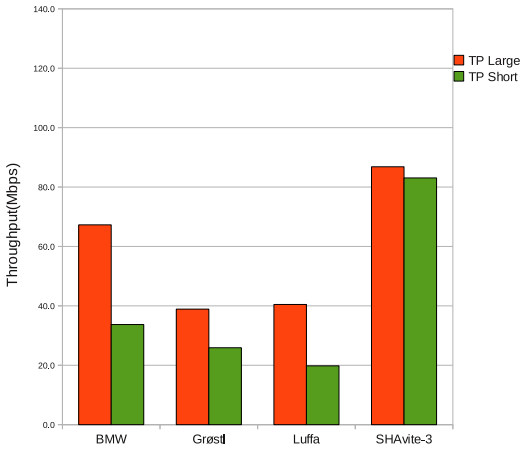


Figure 4.1: Throughput

Grøstl performs worse then the one reported by Jung [19]. But design is significantly changed and performs much better than [19]. This is expalined in the next chapter. There is no reported result on lightweight implementation of SHAvite-3 on any platform.

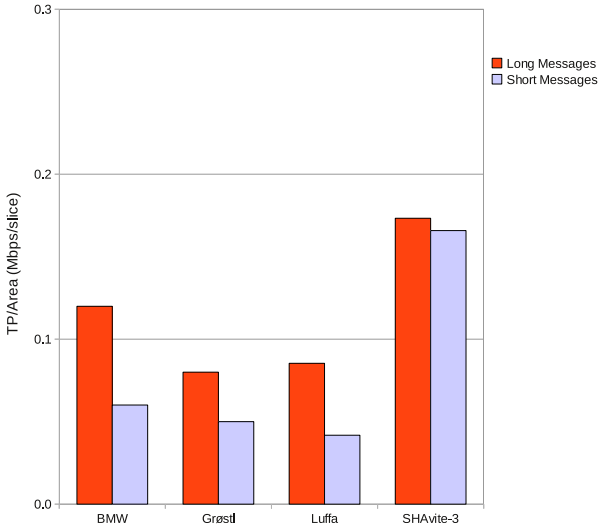


Figure 4.2: Throughput to Area ratio

Table 4.3: Comparison of Lightweight Implementations of Grøstl , Luffa , SHAvite-3 and BMW on Xilinx FPGAs ([TW] – this work))

Algorithm	Reference	Device	I/O Width	Datapath Width	Clock Cycles $(l+p)$	Area (slices)	Block RAMs	Maximum Delay (ns)	Throughput (Mbps)	TP/Area (Mbps/slice)
BMW	[8]	xc3s400a-5	32	32	24832	1,440	0	10.31	2.0	0.001
BMW	[TW]	xc3s200-5	16	32	762	582	1	8.23	81.6	0.140
Grøstl	[19]	xc3s200	64	64	160	1,276	0	16.67	192.0	0.150
Grøstl	[TW]	xc3s200-5	16	32	1120	483	1	11.48	38.7	0.080
Luffa	[24]	xc5vlx75t-1	256	128	50	503	0	4.57	1120.0	2.227
Luffa	[24]	xc5vlx75t-1	256	128	50	355	0	20.0	33.0	0.090
Luffa	[TW]	xc5vlx75t-1	16	32	622	144	1	5.59	81.1	0.460

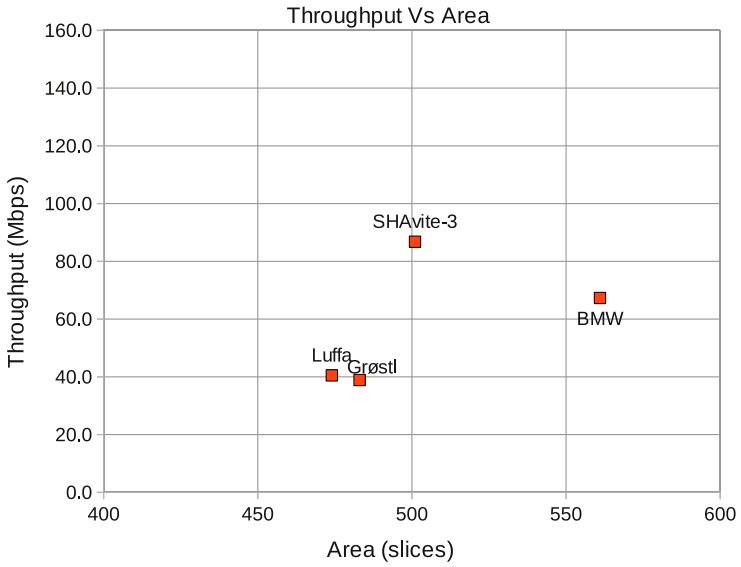


Figure 4.3: Throughput Vs Area plot.

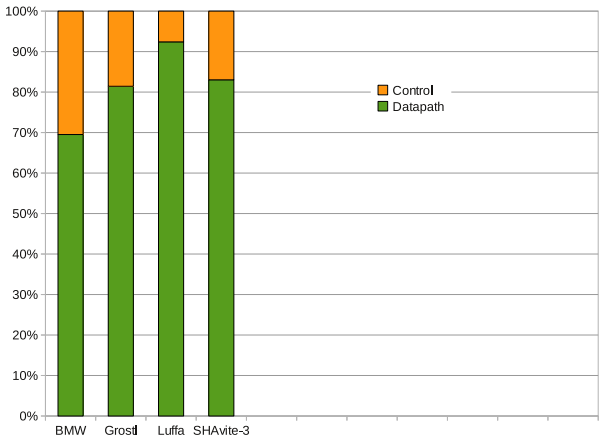


Figure 4.4: Datapath vs Controller

Chapter 5: Scalability of Grøstl

For round-3 of the SHA-3 competition, a tweak was introduced which changes the shifts in the ShiftByte operation and introduces a different AddRoundConstant function for Q. This has minimal efect on the area consumption and does not change the overall architecture.

The drawback of the previous design of Grøstl is that each access of the BRAM can produce only two valid bytes. This is due to the ShiftByte operation. The matrix before and after ShiftByte operation is shown in fig 5.1 . Since the shift operation cannot be achieved within the BRAM, the column is formed by accessing various words and tapping the required bytes. To avoid this, the state(P & Q) can be removed from the BRAM and stored in DRAMs in a way described in the following section.

5.1 Grøstl BRAM with 4 sboxes and half GF multiplier (GrøstlB)

In this implementation the state, consisting of two 512 bit matrices P and Q, is stored in 16 4x8 Distributed RAMs. Each row is stored in one Distributed RAM. In order to get the first 64-bit column we access byte0 from RAM0, byte1 from RAM1. . . etc. This access scheme performs the ShiftByte operation with which we start each round. The AddRoundConstant operation is taken care by the controller. Four bytes are accessed from the memory in each clock cycle. The second and third byte is xored with either zeros or ones based on whther the bytes belong to a P column or a Q column. The first word is xored with the round counter of P or ones and the fourth byte is xored with the round counter of Q or zeros, agian based on which column they belong to. SubBytes is implemented using 4 pipelined S-Boxes which are described as logic functions. The multiplier takes a column from SubBytes and

Table 5.1: Throughput formulae for implementations of Grøstl

Algorithm	Speci-fication	Block Size (bits) b	Clock Cycles to hash N blocks $clk = st + (\ l + \ p) \cdot N + \ end$	Throughput $\frac{b}{(l+p) \cdot T}$
GrøstlB	[13]	512	$2 + (\ 32 + 515) \cdot N + \ 532$	$512/(547 \cdot T)$
GrøstlD0	[13]	512	$2 + (\ 32 + 515) \cdot N + \ 532$	$512/(547 \cdot T)$
GrøstlD1	[13]	512	$2 + (\ 32 + 357) \cdot N + \ 374$	$512/(389 \cdot T)$
GrøstlD2	[13]	512	$2 + (\ 32 + 187) \cdot N + \ 204$	$512/(219 \cdot T)$

0	8	16	24	32	40	48	56
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63

0	8	16	24	32	40	48	56
9	17	25	33	41	49	57	1
18	26	34	42	50	58	2	10
27	35	43	51	59	3	11	19
36	44	52	60	4	12	20	28
45	53	61	5	13	21	29	37
54	62	6	14	22	30	38	46
63	7	15	23	31	39	47	55

Figure 5.1: Grøstl ShiftByte Operation

produces 32 bits of the new column in one clock cycle, the remaining 32 bits in the second clock cycle. It takes a total 3 clock cycles to produce a new column. Each round of P and Q computes 16 new columns which takes 48 clock cycles. BRAM is used in dual port mode and stores the initialization vector and the intermediate hash.

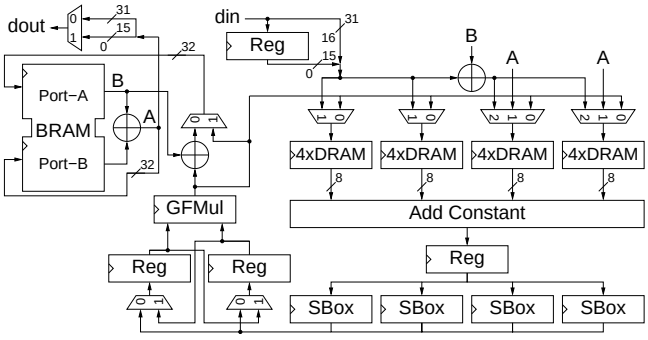


Figure 5.2: GrøstlB implementation.

5.2 Grøstl DRAM with 4 sboxes and half GF multiplier (GrøstlD0)

The block RAM in the previous design is replaced with a distributed RAM.

5.3 Grøstl DRAM with 4 sboxes and full GF multiplier (GrøstlD1)

In this implementation, a full Galois field multiplier is used which takes in 64 bit column from the SubBytes and produces a 64 bit new column. DRAM, instead of BRAM, is used to store the initialization vector and the intermediate hash. It takes a total 2 clock cycles to produce a new column. Each round of P and Q computes 16 new columns, a column of P followed by a column of Q, which takes 32 clock cycles. The total number of clock cycles to process one block of message are 325(16*2*10 + 5(pipeline))clock cycles.

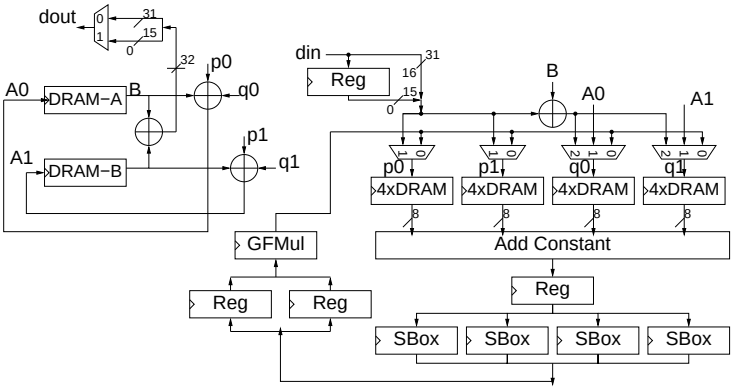


Figure 5.3: GrøstlD1 implementation.

5.4 Grøstl DRAM with 8 sboxes and full GF multiplier (GrøstlD2)

In this implementation, SubBytes is implemented using 8 pipelined S-Boxes which are described as logic functions. A full Galois field multiplier is used which takes in 64 bit column from the SubBytes and produces a 64 bit new column. It now takes only 1 clock cycle to produce a new column. Each round of P and Q computes 16 new columns, a column of P followed by a column of Q, which takes 16 clock cycles. The total number of clock cycles to process one block of message are 165(16*1*10 + 5(pipeline))clock cycles.

5.5 Results and Comparisions

Grøstl by Jung [19]is the only lightweight implementation targeted at a Spartan Device. But their area is more than double of our size as they are using 8 S-boxes. Their delay is also on the higher side as S-boxes used are described as logic without any pipeline registers. They produce a new column for every clock cycle resulting in a high throughput. But our throughput to area ratio is better than theirs.

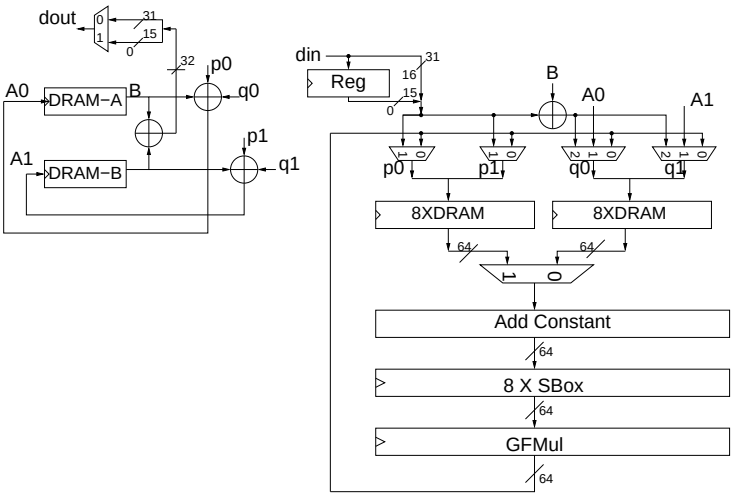


Figure 5.4: GrøstlD2 implementation.

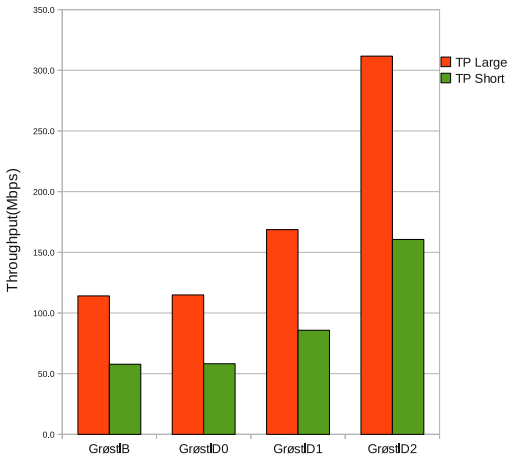


Figure 5.5: Throughput

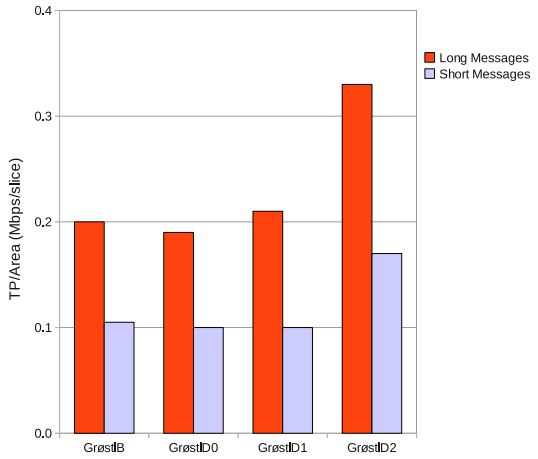


Figure 5.6: Throughput to Area ratio

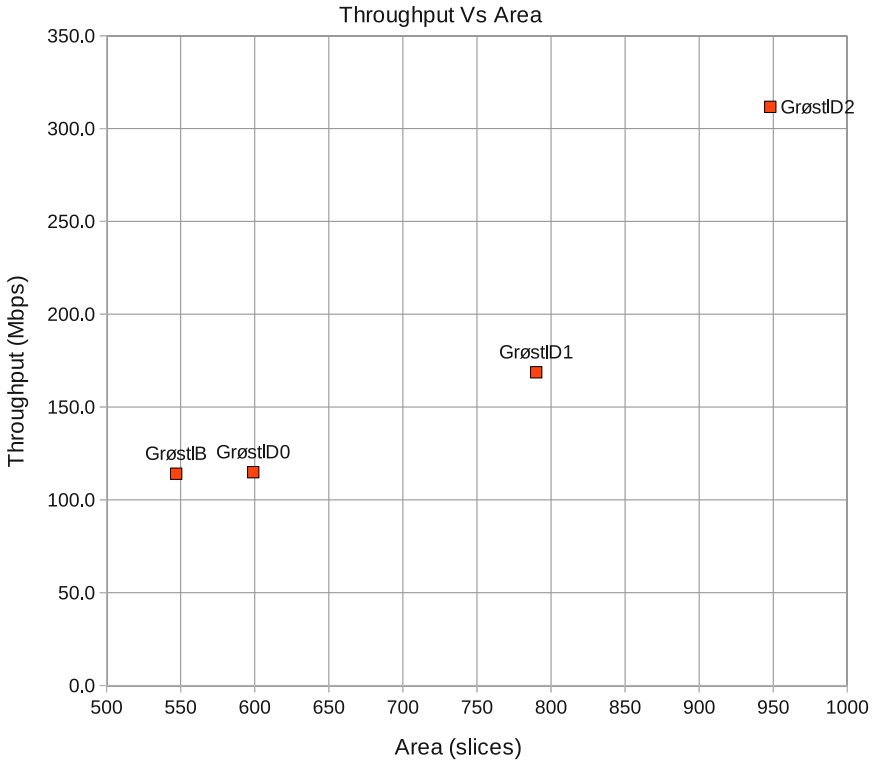


Figure 5.7: Throughput Vs Area plot.

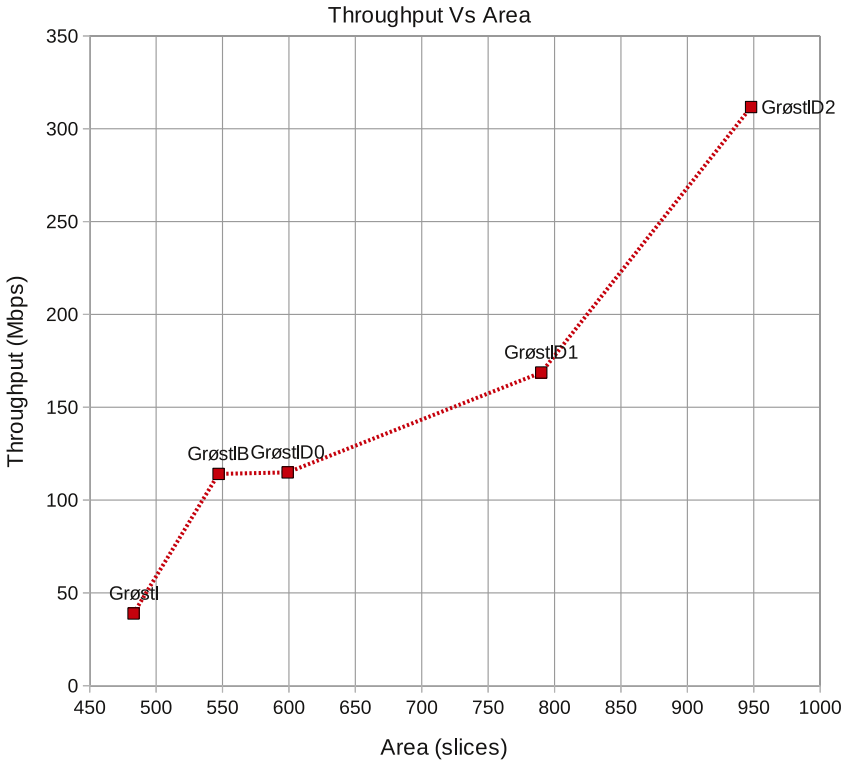


Figure 5.8: Scalability of Grøstl.

Table 5.2: Implementation Results of Lightweight Implementations of Grøstl

Algorithm	Xilinx xc3s200-5			Xilinx xc6slx4csg-3			Xilinx xc5vlx20-2			Altera ep2c5f256c6		
	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (slices)	Block RAMs	Maximum Delay (ns) T	Area (LEs)	Memory Bits	Maximum Delay (ns) T
GrøstlB	547	1	8.20	180	1	5.44	232	1	3.61	1240	3,072	6.72
GrøstlD0	599	0	8.14	229	0	7.80	262	0	3.40	1196	2,560	7.02
GrøstlD1	790	0	7.80	284	0	6.80	367	0	3.10	1459	2,560	6.78
GrøstlD2	948	0	7.50	342	0	6.36	428	0	3.83	1647	2,560	6.73

The resource efficient implementation of Grøstl by Kerchoff[20] and Jung [17], [18] are targeted at Virtex-6 and Virtex-5 families. Both the implementations outperform our GrostlB implementation. Even when they are compared to our GrostlD2 implementation, they perform better . The implementation of Kerchoff takes agvantage of Virtex-6 family. They use 8 Sboxes implemented as LUTs and consume only 64 slices. Our GrostlD2 also uses 8 Sboxes but described as logic functions with pipeline registers which occupy (8 * 17) 136 slices. As result our area is comaparitvely higher than theirs. Additionally the XOR operation (P XOR Q XOR H) after the 10th round takes 32 clock cycles which is avoided in [20] by executing them on the fly. The implementation by Jung is almost similar to our design including the Sbox implementation except that the XOR operation is done on the fly. Our area is on the higher side due to the controller area. The controller designed for the BRAM version is slightly tweaked to work for GrøstlD2 and the area can be decreased significantly if it is designed exclusively for the unfolded version Grøstl(GrøstlD2).

Table 5.3: Throughput Results of Lightweight implementations of Grøstl

Message Algorithm	Xilinx xc3s200-5				Xilinx xc6slx4csg225-3			
	Long		Short		Long		Short	
	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)	Throughput (Mbps)	TP/Area (Mbps/slice)
GrøstlB	114.1	0.20	57.8	0.105	172.0	0.95	87.2	0.484
GrøstlD0	114.9	0.19	58.2	0.100	120.0	0.52	60.7	0.270
GrøstlD1	168.7	0.21	85.8	0.100	193.5	0.68	98.4	0.350
GrøstlD2	311.7	0.33	160.6	0.170	367.2	1.07	189.2	0.550

Message Algorithm	Xilinx xc5vlx20-2				Altera ep2c5f256c6			
	Long		Short		Long		Short	
	TP (Mbps)	(Mbps /slice)	TP (Mbps)	(Mbps /slice)	TP (Mbps)	(Mbps /LE)	TP (Mbps)	(Mbps /LE)
GrøstlB	259.2	1.11	131.4	0.566	139.3	0.11	70.6	0.056
GrøstlD0	275.3	1.05	139.3	0.530	133.3	0.11	67.6	0.056
GrøstlD1	424.5	1.15	215.9	0.590	194.1	0.13	98.9	0.067
GrøstlD2	610.4	1.42	314.5	0.731	347.4	0.21	179.8	0.109

Table 5.4: Comparison of Lightweight Implementations of Grøstl on Xilinx FPGAs

Algorithm	Reference	Device	I/O Width	Datapath Width	Clock Cycles $(l+p)$	Area (slices)	Block RAMs	Maximum Delay (ns)	Throughput (Mbps)	TP/Area (Mbps/slice)
Grøstl	[19]	xc3s200	64	64	160	1,276	0	16.67	192.0	0.150
GrøstlB	[TW]	xc3s200-5	16	32	547	547	1	8.20	114.1	0.200
GrøstlD0	[TW]	xc3s200-5	16	32	547	599	0	8.14	114.9	0.190
Grøstl	[20]	xc6vlx75t-1	64	64	176	285	0	3.57	815.0	2.860
GrøstlD2	[TW]	xc6vlx75t-1	16	32	219	403	0	4.51	518.0	1.285
Grøstl	[18]	xc5v	32	64	160	368	0	3.27	975.0	2.640
GrøstlD2	[TW]	xc5v	16	32	219	428	0	3.83	610.4	1.420

Chapter 6: Power Measurements

6.1 Implementation

Two FPGA boards are used for the setup. One is the Nexys2 board and the other one is the Spartan 3E starter kit[29]. The Nexys2 board is used for the wrapper and the Spartan 3E starter kit is used for the hash function. The two boards are connected using a bridge connecter.

Nexys2 Board : The Nexys2 is USB powered and the Spartan 3E FPGA on it is programmed with the wrapper. This FPGA produces the trigger signal according to the state of the HASH function. Digilent's Adept tool is used for programming the FPGA on the Nexys2 board.

Spartan 3E starter kit Board : This board is used for loading the hash function and measuring its power consumption. Firstly, to separate the core FPGA from the rest, the jumper j7 is removed. Now, the core FPGA is powered using an external power supply. A series circuit is formed with a 1ohm resistor and the FPGA core with 1.2V DC. The current flowing through the circuit will be in the range of mA and hence the voltage drop across the resistor is very negligible. This ensures almost all the voltage is supplied to the core and is within the acceptable range of 1.14V to 1.26V [30].

The power measurements are taken across the resistor with the help of a digital oscilloscope and the power consumed by the core FPGA is calculated accordingly. In one of the channels, the voltage across the resister is traced and to the other channel the trigger signal produced by the Nexys2 board is connected.

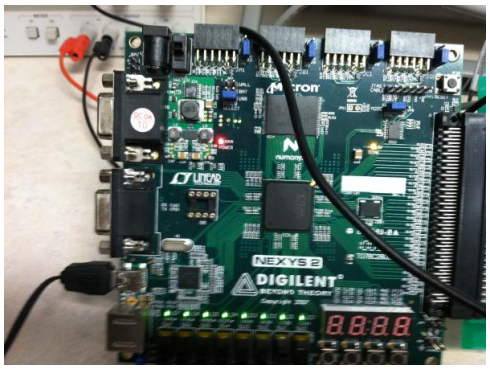


Figure 6.1: Nexys2 Board

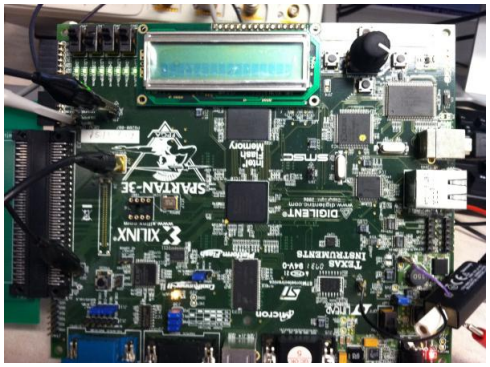


Figure 6.2: Spartan 3E starter kit

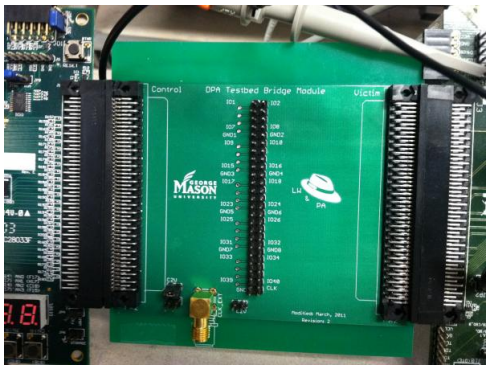


Figure 6.3: Bridge Connector

Bridge connector: The bridge connector connects the Nexys2 board and the Spartan 3E kit. The UCF files are written accordingly so that a proper communication is achieved between the two boards.

First the two boards are connected using the bridge connector. Then the Spartan 3E starter kit is externally powered and jumper j7 is removed to separate the core FPGA. The core FPGA, which is made as part of the series circuit is powered using the power supply with the voltage fixed at 1.2V. The wrapper is programmed onto the FPGA in the Nexys2 board. The Nexys2 board has the external control signals rst and start and also the trigger output.

Without giving the inputs, the power consumed by the core FPGA of the starter kit is measured which is nothing but the static power of the design loaded. When the rst signal is enabled, all the signals of the hash function remain the same except the clock. So the power measured in this state is the clock + static power. After that the rst signal is disabled and start signal is enabled which makes the src_ready signal = '0'. So the hash function generates a src_read signal and starts loading the message block from the wrapper. Now the power measured includes the I/O power as well. After the message is loaded, the SHA

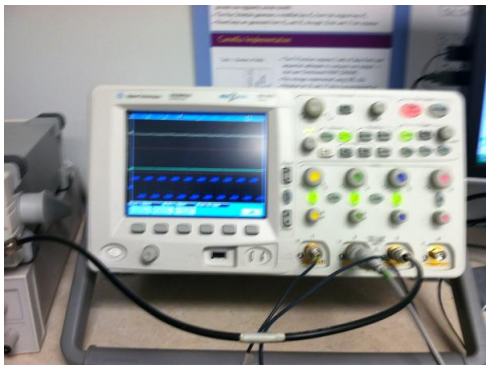


Figure 6.4: Digital Oscilloscope

function startes processing and the power measured is the total power which inludes static, clock and (logic power + interconnect power). After processing the message, the SHA core writes output hash digest back to the wrapper by enabling Dst_write signal.

The digital oscilloscope is capable of recording multiple number of points and these recordings are separated into each state with the help of the trigger signal connected to the other channel of the oscilloscope. The maximum, minimum and also the average value of the readings are obtained from the readings.

This process is repeated for all the five hash functions and the corresponding logic powers are measured. Since we know the number of bits the core is processing and also the time it is taking to process one block, the energy per bit is also calculated.

6.2 Results

The Power measurements of the SHA-3 finalists are shown in Table 6.2. JH consumes more power among all the finalists and skein consumes the least. When it comes to energy, JH is again the top among the energy consumptions.But now the order differs slightly. BLAKE

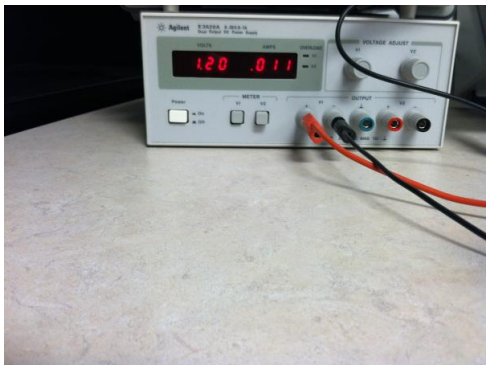


Figure 6.5: Power Supply

Figure 6.6: Communication between Nexys2 board and Spartan 3E starter board

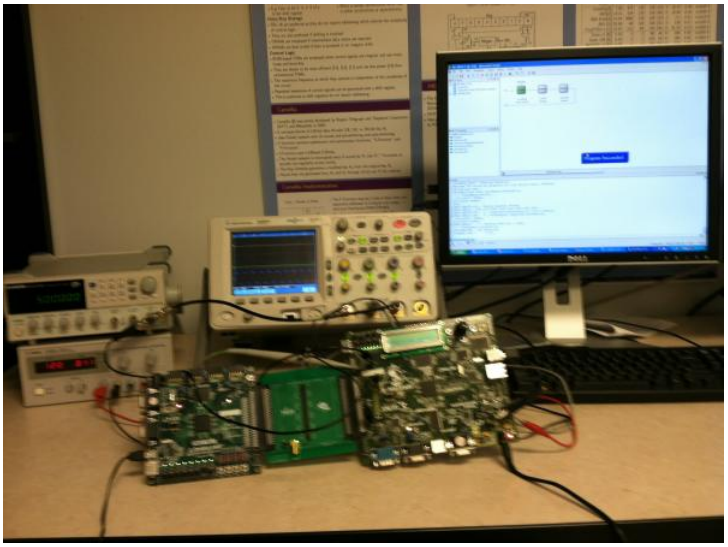


Figure 6.7: Power Measurement Setup

is the least among all overtaking Skein. This is due to the fact that Skein takes more number of clock cycles(more than 9 times) to process than BLAKE. Also both the algorithms process a block of same size. So the energy consumed per bit of Skein is also higher than BLAKE.

The estimated values obtained by XPOWER are shown in table 6.4. From the results obtained, it is observed that the difference between the estimated power and measured power varies from 10% to 42%. BLAKE and Grøstl measurements are close to their estimated values whereas JH and Skein have significant difference.

The Power vs Area plot is shown in fig 6.15. It is observed that Dynamic power consumption is not proportional to area but Staic power is directly proportional to the area occupied. JH is the least among area but consumes the highest power among all. BLAKE and Grøstl occupy the same area but Grøstl consume more power comparitively.

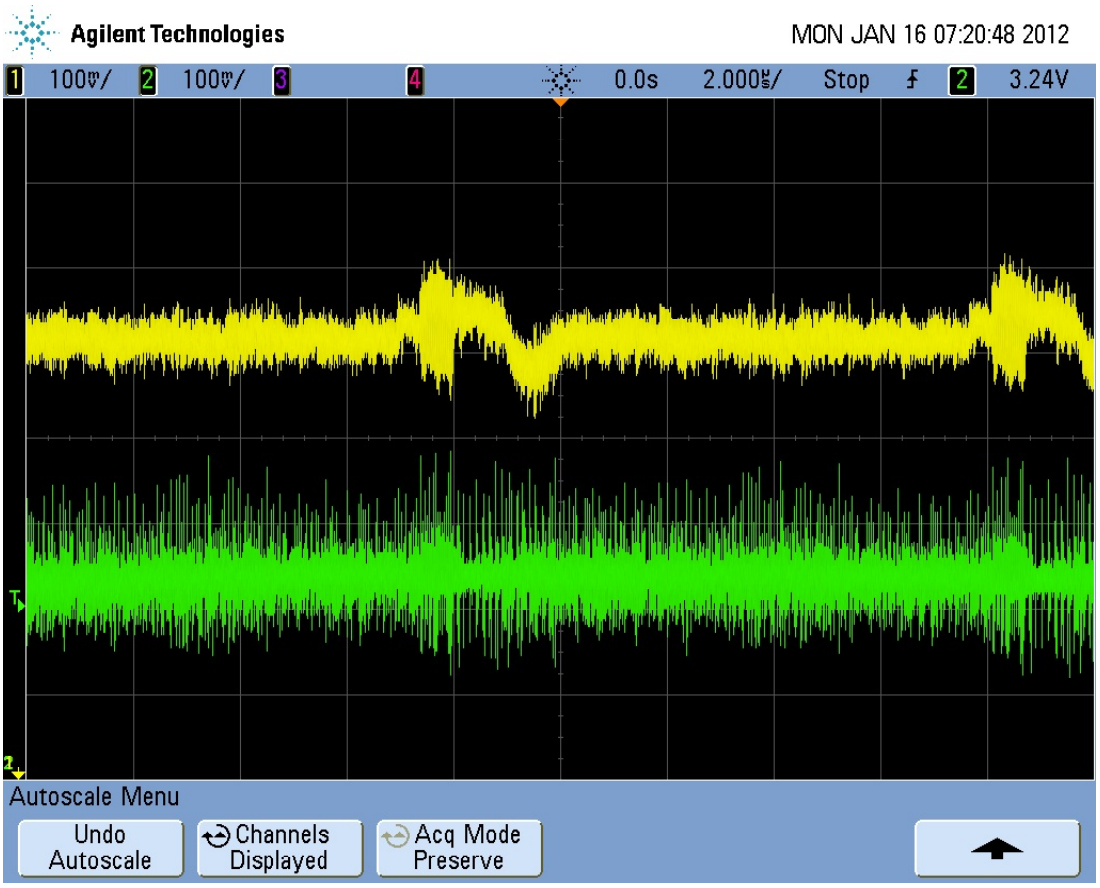


Figure 6.8: Power Trace

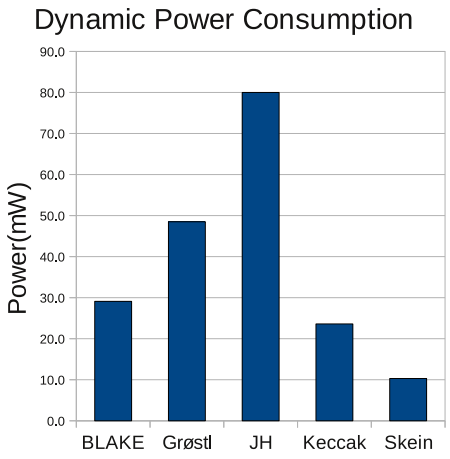


Figure 6.9: Dynamic Power Consumption

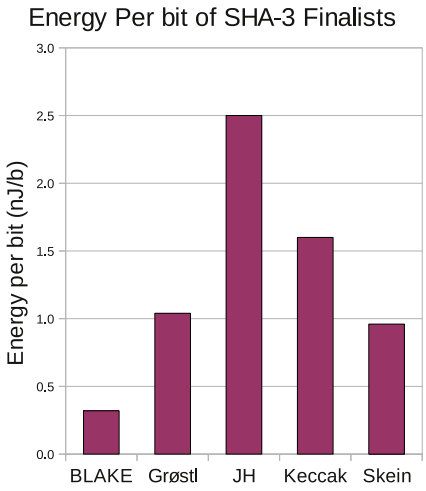


Figure 6.10: Energy Consumption

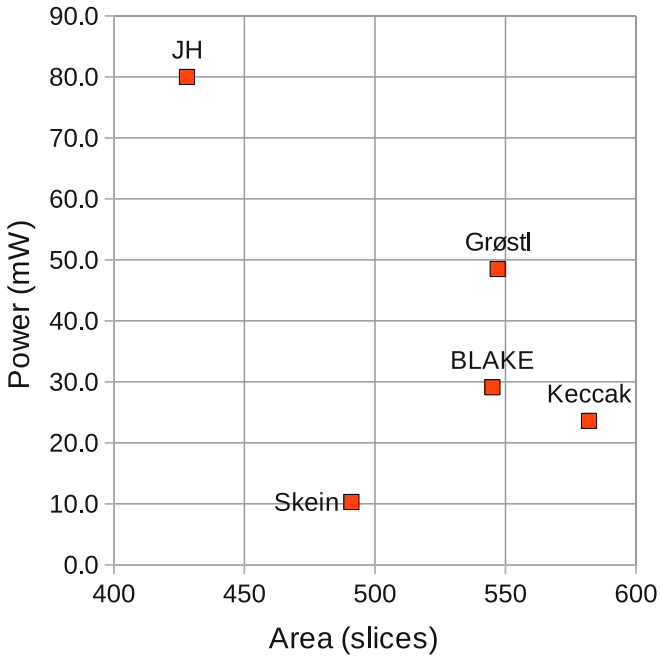


Figure 6.11: Power vs Area Plot

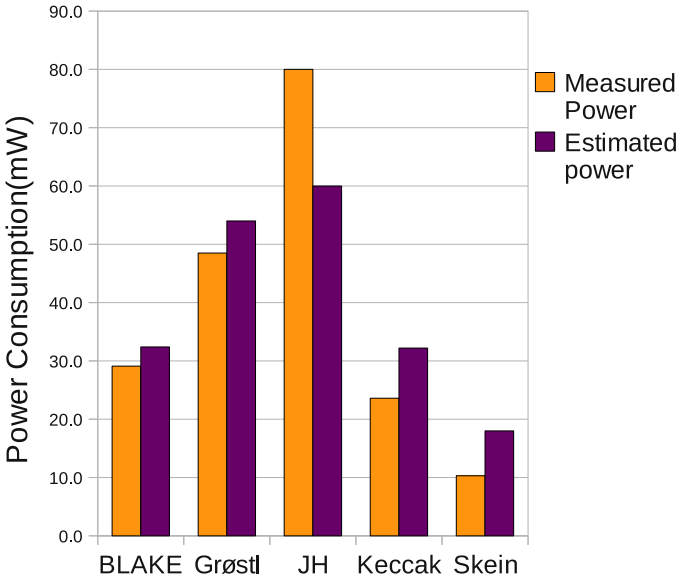


Figure 6.12: Estimated Power vs Measured Power

Table 6.1: Power Measurements of SHA-3 fianlists

Algorithm	Specification	Area	(Static + Clock) Power (mW)	Dynamic power (mW)	Total Power (mW)
BLAKE	[3]	545	110.0	29.1	139.1
Grøstl	[13]	537	99.9	48.5	148.4
JH	[28]	428	68.4	80.0	148.4
Keccak	[5]	582	138.0	23.6	161.6
Skein	[10]	491	89.6	10.3	99.9

Table 6.2: Energy per bit of SHA-3 fianlists

Algorithm	Block Size (b)	Clock Cycles	Dynamic power (mW)	Energy (nJ)	Energy per bit (nJ/b)
BLAKE	512	290	29.1	168.78	0.32
Grøstl	512	547	48.5	530.59	1.04
JH	512	800	80.0	1280.00	2.50
Keccak	1088	3764	23.6	1776.61	1.63
Skein	512	2398	10.3	493.99	0.96

Table 6.3: Power Estimations of SHA-3 fianlists

Algorithm	Specification	(Static + Clock) Power (mW)	Dynamic power (mW)	Total Power (mW)
BLAKE	[3]	33.6	32.4	66.0
Grøstl	[13]	32.4	54.0	86.4
JH	[28]	32.4	60.0	92.4
Keccak	[5]	31.4	32.2	63.6
Skein	[10]	31.2	18.0	49.2

Table 6.4: Power Measurements against Power Estimations

Algorithm	Estimated Power (mW)	Measured Power (mW)	Difference (%)
BLAKE	32.4	29.1	-10.18%
Grøstl	54.0	48.5	-10.18%
JH	60.0	80.0	33.33%
Keccak	32.2	23.6	-26.70%
Skein	18.0	10.3	-42.77%

Chapter 7: Conclusions and Future Work

7.1 Lightweight implementations

7.1.1 Conclusion

From the results obtained, it can be concluded that Grøstl is nicely scalable. Since it is mainly based on AES, the desigm optimizations made to AES can be adapted to Grøstl. The relationship between the throughput and area is definitely not linear. As you unfold the design, there is significant amount of increase in the throughtput. But the performance of Grøstl implemented with 4 S-boxes and a full GF multiplier is not significant. The throughput to area ratio is almost the same as that of the one implemented with a lesser area (4 S-boxes and half GF multiplier). So GrøstlD and GrøstlD2 are better choices depending on the area constraint. The imact of dedicated resources like BRAM is minimal for Grøstl.

7.1.2 Future work

The controllers of all the designs are not fully optimized. Allmost all the controller areas are more than 100 slices. The area of these controllers can reduced atleast by 30% by optimizing them further.

7.2 Power measurements

7.2.1 Conclusion

Skein consumes the least power and JH, the most. BLAKE consumes the least energy per bit. The Power estimation tools could not provide accurate information about the power consumed by designs. They vary considerably from the measured values.

7.2.2 Future work

The accuracy of the measurements can be increased further by maintaining the voltage across the FPGA at a constant value(Vccint = 1.2V). Also, the routing or interconnect power can be separated from the logic power by following the methodologies described by Jevtic et all [2]. A dedicated board can developed from the available ones by removing all additional components with only the core FPGA on it. This eliminates the leakage currents(if any). The setup can be interfaced with a PC and the data can be sent from the PC to the FPGA and the HASh digest computed by the SHA function on the core is returened back to the PC and can be verified.

Bibliography

Bibliography

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

[2] Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register/ Vol. 72, No. 212 (Nov 2007), notices 62212

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

[4] Baldwin, B., Hanley, N., Hamilton, M., Lu, L., Byrne, A., O'Neill, M., Marnane, W.P.: FPGA implementations of the round two SHA-3 candidates. Tech. rep., Second SHA-3 Candidate Conference (2010)

[5] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family main document. http://keccak.noekeon.org (Apr 2009), version 1.2

[6] Biham, E., Dunkelman, O.: The SHAvite-3 hash function. Submission to NIST (Round 2) (2009), http://www.cs.technion.ac.il/~orrd/SHAvite-3/Spec.15.09.09.pdf

[7] De Cannière, C., Sato, H., Watanabe, D.: Hash function Luffa: Specification. Submission to NIST (Round 2) (Oct 2009), http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_Specification_20091002.pdf

[8] El-Hadedy, M., Gligoroski, D., Knapskog, S.: Low area implementation of the hash function "Blue Midnight Wish 256" for FPGA platforms. In: Intelligent Networking and Collaborative Systems, INCOS '09. pp. 100–104. IEEE (2009)

[9] El-Hadedy, M., Margala, M., Gligoroski, D., Knapskog, S.J.: Resource-efficient implementation of Blue Midnight Wish-256 hash function on Xilinx FPGA platform. Tech. rep., Second SHA-3 Candidate Conference (2010)

[10] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (Round 3) (2010), http://www.skein-hash.info/sites/default/files/skein1.3.pdf

[11] Gaj, K., Homsirikamol, E., Rogawski, M.: Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGA. In: Mangard, S., Standaert, F.X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 264–278. Springer Berlin / Heidelberg (2010)

[12] Gaj, K., Kaps, J.P., Amirineni, V., Rogawski, M., Homsirikamol, E., Brewster, B.Y.: ATHENa – Automated Tool for Hardware EvaluatioN: Toward fair and comprehensive benchmarking of cryptographic hardware using FPGAs. In: FPL 2010. pp. 414–421. IEEE (2010)

[13] Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schäffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (Oct 2008), http://www.groestl.info/

[14] Gligoroski, D., Klima, V., Knapskog, S.J., El-Hadedy, M., Amundsen, J., Mjølsnes, S.F.: Cryptographic hash function Blue Midnight Wish. Submission to NIST (Round 2) (Sep 2009), http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf

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

[16] Jevtic, R., Carreras, C.: Power measurement methodology for FPGA devices. IEEE Transactions on Instrumentation and Measurement 60(1), 237–247 (Jan 2011)

[17] Jungk, B.: Compact implementations of Grøstl, JH and Skein for FPGAs (May 2011), ECRYPT II Hash Workshop 2011

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

[19] Jungk, B., Reith, S.: On FPGA-based implementations of Grøstl. Cryptology ePrint Archive, Report 2010/260 (2010)

[20] Kerckhof, S., Durvaux, F., Veyrat-Charvillon, N., Regazzoni, F., de Dormale, G.M., Standaert, F.X.: Compact FPGA implementations of the five SHA-3 finalists (May 2011), ECRYPT II Hash Workshop 2011

[21] Kobayashi, K., Ikegami, J., Matsuo, S., Sakiyama, K., Ohta, K.: Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII. http://eprint.iacr.org/2010/010 (Jan 2010)

[22] Matsuo, S., Knežević, M., Schaumont, P., Verbauwhede, I., Satoh, A., Sakiyama, K., Ota, K.: How can we conduct "fair and consistent" hardware evaluation for SHA-3 candidate? Tech. rep., Second SHA-3 Candidate Conference (2010)

[23] Meintanis, D., Papaefstathiou, I.: Power consumption estimations vs measurements for FPGA-based security cores. In: International Conference on Reconfigurable Computing FPGAs. IEEE (2008)

[24] Mikami, S., Mizushima, N., Nakamura, S., Watanabe, D.: A compact hardware implementation of SHA-3 candidate Luffa. http://www.sdl.hitachi.co.jp/crypto/luffa/ACompactHardwareImplementationOfSHA-3CandidateLuffa_20101105.pdf (2010)

[25] Sönmez Turan, M., Perlner, R., Bassham, L.E., Burr, W., Chang, D., jen Chang, S., Dworkin, M.J., Kelsey, J.M., Paul, S., Peralta, R.: Status report on the second round of the SHA-3 cryptographic hash algorithm competition. NIST Interagency Report 7764, NIST, Gaithersburg, MD, USA (Feb 2011)

[26] Tuan, T., Kao, S., Rahman, A., Das, S., Trimberger, S.: A 90nm low-power FPGA for battery-powered applications. In: FPGA '06. pp. 3–11. ACM/SIGDA, ACM, New York, NY, USA (2006)

[27] Westermann, B., Gligoroski, D., Knapskog, S.: Comparision of the power consumption of the 2nd round sha-3 candidates (2010)

[28] Wu, H.: The hash function JH. Submission to NIST (round 3) (2011), `http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf`

[29] Xilinx: Spartan-3E Starter Kit Board User Guide (march 2006)

[30] Xilinx: Spartan-3 FPGA Family Data Sheet (December 2009)

Curriculum Vitae

Kishore Kumar Surapathi received his Bachelor of Technology Degree from Gayatri Vidya Parishad College of Engineering, Visakhapatnam, India in May 2009. He started working towards his Master of Science degree in Computer Engineering from August 2009 at George Mason University.

As a Graduate Research Assistant in Cryptographic Engineering Research Group at George Mason, he worked on several projects and gained hands on experience in design and implementation of cryptographic algorithms on FPGAs and ASICs. He received several academic awards for his projects. His work is partially supported by NIST. He also worked as Graduate Teaching Assistant in the Department of Electrical and Computer Engineering and achieved very good reviews from the students.