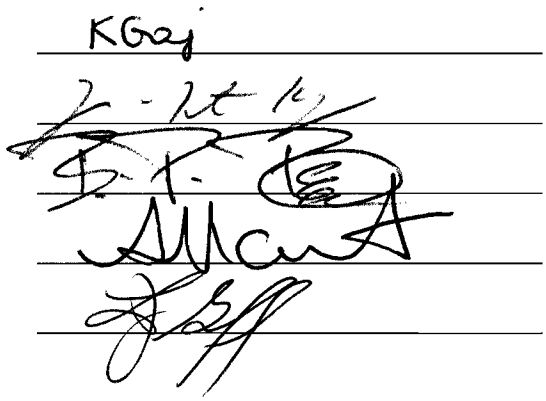
FAIR AND COMPREHENSIVE COMPARISON OF HARDWARE PERFORMANCE OF SHA-3 ROUND 2 CANDIDATES USING FPGAS

by

Ekawat Homsirikamol

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. Kris Gaj, Thesis Director

Dr. Jens-Peter Kaps, Committee Member

Dr. Bernd-Peter Paris, Committee Member

Dr. Andre Manitius, Department Chairman

Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: 07/30/2010

Summer Semester 2010
George Mason University
Fairfax, VA

Fair and Comprehensive Comparison of Hardware Performance of SHA-3 Round 2 Candidates Using FPGAs

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

By

Ekawat Homsirikamol
Bachelor of Science in Electrical Engineering
Volgenau School of Information Technology and Engineering, 2008

Director: Dr. Kris Gaj, Professor
Department of Electrical and Computer Engineering

Summer Semester 2010
George Mason University
Fairfax, VA

Dedication

I dedicate this thesis to the advancement of mankind.

Acknowledgments

First, I would like to thank my advisor Dr. Kris Gaj for his kind advise and support throughout the entire course of working on this project. This thesis would never be as completed as it is now without him. I also thank Marcin Rogawski for his significant contributions to this project. Marcin has contributed the final designs for many architectures, and played a major role in generating results for this thesis.

Special thanks go to Dr. Jens-Peter Kaps for keeping me connected to the system whenever I log myself out. Additionally, I would like to thank my fellow colleagues at CERG for providing several insightful comments (and entertainments). Most of all, this work would not be able to be completed without the help from students from the class of ECE545 in Fall 2009 and ECE645 in Spring 2010 in developing the initial and, in many cases, the final designs of the hardware architectures presented in this paper.

Last, but not least, I would like to thank my parents, Thanakosin Homsirikamol and Wipa Buranchai, for providing me with the opportunity to study abroad and for their support during my times of need.

This work was supported in part by NIST through the Recovery ActMeasurement Science and Engineering Research Grant Program, under contract no. 60NANB10D004.

Table of Contents

List of Tables

List of Figures

Abstract

FAIR AND COMPREHENSIVE COMPARISON OF HARDWARE PERFORMANCE OF SHA-3 ROUND 2 CANDIDATES USING FPGAS

Ekawat Homsirikamol, MS

George Mason University, 2010

Thesis Director: Dr. Kris Gaj

In 2007, National Institute of Standards and Technology has announced a contest for a new American cryptographic hash function standard, called SHA-3. At the time of writing, after eliminating 37 algorithms in Round 1, due to security and performance weaknesses, only 14 Round 2 candidate algorithms remain in the competition. A comprehensive methodology for fair comparison of hash algorithms competing in the SHA-3 contest from the point of view of hardware performance in FPGAs has been proposed in this thesis. Based on this methodology, hardware designs optimized for the maximum throughput to area ratio have been developed for all Round 2 SHA-3 candidates. The obtained results have been compared with results from other groups. In our study, only three candidates, namely CubeHash, Keccak and Luffa, have consistently outperformed SHA-2 in terms of the throughput to area ratio for both 256 and 512 bits versions of all hash algorithms.

Chapter 1: Introduction

1.1 Hash Function

Imagine you are a detective who is about to interrogate a suspect. The suspect happens to fit your witnesses's description. Based on your outstanding detective instinct, you are certain that the suspect is the right one. The only task left for you is to verify his identity. How would you do that? How can you verify his identity even though he may be lying to you and reinforcing his story by showing a fake identification card. The answer is quite simple. You would check his fingerprints and verify them against the police database.

A similar situation can occur in digital world. As data are being transferred across the globe through the internet filled with hackers, the need to verify data integrity (or identity) is paramount. It is also very important to ensure that data stored in a database is unique and not tampered with. Hash functions provide a solution to these two problems. Hashing is a process that reduces an input file or a message to a fixed size output through a sequence of operations that is computationally fast. This output is called hash value or message digest (MD). A hash value can be used to verify an integrity of a file by comparing an output of the hash function applied to the current version of the file with the original hash value computed when the file was created. It can also be used as a file id for protection of database records.

Another important application of hash functions is its use in digital signatures. In majority of practical signature schemes, messages are hashed before applying actual signature scheme based on a public key cryptosystem. This way, all signatures have the same size, independent of the size of the original message, and the generation and verification of the signature is computationally efficient. In order for a hash function to be suitable for use in digital signatures, it must have a special property called "collision resistance". A hash

function has a collision resistance if and only if it is computationally infeasible to find two different messages that hash to the same value.

One of the most widely used secure hash families is commonly called Secure Hash Algorithm-2 (SHA-2). This family includes four functions with the following hash value sizes: 224, 256, 384, and 512. All functions were designed by National Security Agency (NSA), and approved as an American federal standard by National Institute of Science and Technology (NIST) in August 2002. The SHA-2 family is based on very similar basic building blocks and has a very similar overall structure as an earlier American hash function standard, called SHA-1, which was also designed by NSA. In 2005, a substantial security flaw was discovered in SHA-1 [1]. Since SHA-2 is based on very similar principles to SHA-1, it is prudent to expect that the similar attack can be found against SHA-2. Such attack can either break SHA-2 completely, or at least significantly reduce its strength compared to the theoretical strength assumed today.

In response to this potential threat, NIST initiated the SHA-3 contest in search of a new cryptographic hash function family. The winner of this competition will become the next standard for secure hash algorithm. As of now, the competition has proceeded to Round 2, with only 14 out of the original 51 candidates remaining.

1.2 Motivation

The selection of the contest winner based only on its security strength is inadequate. Would user care if one algorithm performed 10 times slower and took 1000 years to break as opposed to a faster algorithm that takes 800 years to break with the current technology? Most likely not. This can be seen from the previous cryptographic competitions such as the Advanced Encryption Standard (AES) contest in 1997-2000. The winner, Rijndael, was chosen based on its performance in hardware and software, when its security was judged adequate by NSA and the rest of the cryptographic community [2].

Nevertheless, the criteria used in the AES contest, where performance was considered equivalent to speed, will most likely be inadequate in the SHA-3 competition. AES contest

was held before a major boom in low-power portable, hand-held devices. A decade has changed the usage of low power devices from occasional to ubiquitous. Radio-Frequency Identification (RFID) tags are being used to track shipments worldwide. Kindergartners are carrying mobile phones. Laptops are getting smaller. New inventions tend to have smaller and greener functionality than its predecessors. With all these trends, one can see that speed and security are not enough in judging the winner of SHA-3 competition. Area, cost, power and energy consumption will definitely play a role in this contest.

1.3 Previous Studies

As of writing this thesis, relatively few hardware performance reports for SHA-3 candidates have been published. The summary of existing hardware implementations can be found in [3]. Thus far, only [4–7] provide comprehensive hardware performance reports, and [8–19] present reports for specific hash functions. With the exception of [6], all of the hardware performance papers were focused only on 256-bit versions of SHA-3 candidates and the implementation goals of all these paper are either throughput or throughput to area ratio. In compact designs, only [20] presents the study of Blake for low area implementation. Regarding the interface, [7,21] have proposed uniform interface but none of these interfaces has been widely adopted by other groups yet. Unclear optimization targets, the use of different hardware description languages, unclear performance measure, and varying design methodologies further complicated the comparison procedure. While not all papers contain these deficiencies, even the ones that report multiple implementations within the same paper often make at least one mistake.

Chapter 2: Design Methodology

To avoid the common fallacies in designing toward hardware comparison, this thesis follows the established ground rules as proposed by [5]. These rules are summarized below:

- **FPGA**s are selected as our primary implementation platform.
- **Uniform input/output interface and protocol** is used in implementations of all of the SHA-3 candidates.
- **Optimization target** is throughput to area ratio.
- **The same basic building blocks** are used in implementations of all candidates, by reusing the same source codes for low level operations. This approach rules out the possible inconsistencies in optimizations of basic logic operations, possible if these operations were implemented separately for each candidate.
- **Hardware description language** is VHDL. Different languages may have different level of optimization capability. Using the same language ensures that a design will not get a better result simply because of the different treatment of different languages by the current generation of CAD tools.
- **Same assumptions and simplifications**, such as no padding in hardware and no support for special modes of operation.
- **CAD tools** selected are tools developed by the FPGA vendors :
  - *Xilinx* : Xilinx ISE Design Suite v. 11.1
  - *Altera* : Quartus II v. 9.1 Subscription Edition Software

- **No dedicated resources**. This means that only Configurable Logic Blocks (CLBs) in Xilinx FPGAs, and Logic Elements or Adaptive Look-Up Tables (ALUTs) in Altera FPGAs are used for synthesis and implementation. No Block RAMs, memory bits, DSP units, or multipliers are used.

- **Identical and easy to repeat tool options** are possible through the use of an open source benchmarking environment, called ATHENa (Automated Tool for Hardware EvaluatioN), developed at George Mason University [22].

- **Results for several families**. The results are generated across 7 FPGA families from two major FPGA vendors, Xilinx and Altera.

- **Generalized design** is used for all SHA-3 candidates when applicable. See Section 5 for the detailed diagrams.

- **Universal testbench** is applied to all of the SHA-3 candidates to ensure the correct functionality. The testbench accepts test vectors generated by the padding script developed in Perl. The script uses the Known Answer Test (KAT) test vector files available as a part of each candidate's submission package.

Major points are discussed in detail in subsections.

2.1 Uniform Interface and Protocol

Utilizing the same interface and protocol for all algorithms reduces a possible negative impact that using different interfaces can have on the comparison process. Assuming that two algorithms accept the same block size of an input message, if one design uses a larger input bus width than the other, the number of clock cycles required for loading a message may be different. This difference may give the design with the larger input bus width an edge over the design with the smaller bus. Different interface or protocol can also cause different overhead in terms of area. In general, algorithms are required to keep track of the size of the message. This task can be done inside or outside of the cryptographic

unit. If an interface assumes performing this task outside the core, the required resources are minimized, but the requirements on the sophistication of the source circuit increase. Similarly, padding can be performed either inside or outside of the core, with an impact at least on the area (and possibly also on speed) of the hash core.

As noted previously, only two interfaces have been developed. The selected interface is the one proposed by [5]. This interface of our cryptographic core is shown in Figure 2.1a. The core is assumed to be surrounded by standard FIFO modules. These modules provide necessary control signals and input/output buses to the core. The core assumes an active role and makes the decisions about receiving or transmitting data based on its internal state and two external control signals. In general, the interface has two variable parameters:

- w = the width of the input data bus, *din*, and the output data bus, *dout*. These buses are independent of each other, but both have the same width w.
- $r_{io} = \lceil f_{ioclk}/f_{clk} \rceil$, i.e., the ratio of the clock frequency for the fast I/O clock (used only for the fast communication with the surrounding circuits, typically Input and Output FIFOs), and the clock frequency for the main clock used for data processing. If only one clock is used for both functions, r_{io}=1.

A protocol for this interface is shown in Figure 2.2. In essence, the protocol transmits the length of the message in bits followed by the message itself. This protocol also allows segmentation of messages. In other words, the message can be split into multiple segments. The lengths of all but the last segment are required to be multiples of the message block length. Figure 2.2a describes the format of incoming data in case the message consists of only one segment. Figure 2.2b describes the case when the message is split into multiple segments. The latter scenario is useful in case the message is greater or equal than 2^w bits, i.e., its length cannot be represented using a single word, or if the message length is not known in advance, and becomes available only during message processing.

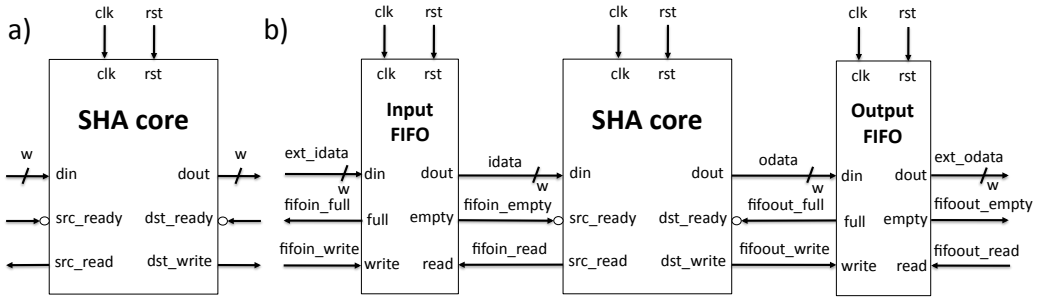


Figure 2.1: a) Input/output interface of a SHA core. b) A typical configuration of a SHA core connected to two surrounding FIFOs.

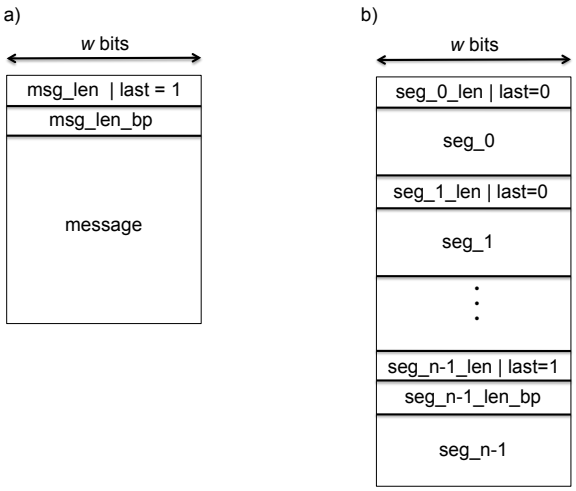


Figure 2.2: Format of input data for two different operation scenarios: a) with message bitlength known in advance, and b) with message bitlength unknown in advance. Notation: msg_len – message length after padding, msg_len_bp – message length before padding, seg_i_len - segment i length after padding, seg_i_len_bp – segment i length before padding, last – a one-bit flag denoting the last segment of the message (or one-segment message)

2.2 Optimization Target

Optimization Target is one of the most important decisions we have to make in order to develop a fair comparison. The possible choices include Maximum Throughput, Minimum Area, Maximum Throughput to Area Ratio, Minimum Latency, etc. All of the aforementioned targets can be used to make a comparison. Out of them, we have selected Maximum Throughput to Area Ratio as our criteria of choice. This selection has advantages over other possible choices. First, it is practical, as hardware cores are typically applied in situations, where the size of the processed data is significant and the speed of processing is essential.

Secondly, this optimization criterion is a very reliable guide throughout the entire design process. At every junction where the decisions must be made, starting from the choice of a high-level hardware architecture down to the choice of the particular FPGA tool options, this criterion facilitates the decision process, leaving very few possible paths for further investigation.

On the contrary, optimizing for Throughput alone, leads to highly unrolled hash function architectures, in which a relatively minor improvement in speed is associated with a major increase in the circuit area. In hash function cores, Latency, defined as a delay between providing an input and obtaining the corresponding output, is a function of the input size. Since various sizes may be most common in specific applications, this parameter is not a well-defined optimization target. Finally, optimizing for Area leads to highly sequential designs, resembling small general-purpose microprocessors, and the final product depends highly on the maximum amount of area (e.g., a particular FPGA device) assumed to be available.

With the maximum throughput to area ratio as our optimization target, the main iterative task of each candidate needs to be investigated to ensure the best result. The general rule applies that if two identical (or similar) tasks are performed in parallel, then there is no gain in terms of the throughput to area ratio over executing them sequentially. If however the same two identical (or similar) tasks are performed sequentially, one by one, then executing them using the same hardware significantly reduces the area of the circuit, with almost no penalty in terms of the speed. After taking this rule into account, and after performing the more detailed comparative analysis if needed, the optimum main iterative tasks used in all our SHA-3 candidate designs are described in Table 2.1.

2.3 Basic Components

The same basic components (if applicable) are used in all our designs. This eliminates the possibility that the same component when used in one design is more optimized than when appearing in other designs. The major operations of all SHA-3 candidates that require

Table 2.1: Main iterative tasks of the hardware architectures of SHA-3 candidates optimized for the maximum Throughput to Area ratio

Function	Main Iterative Task	Function	Main Iterative Task
BLAKE	$G_0..G_3$	**JH**	Round function R_4
BMW	f_0, f_1, f_2	**Keccak**	Round R
CubeHash	one round	**Luffa**	The Step Function, Step
ECHO	AES round	**Shabal**	Two iterations of the main loop
Fugue	2 subrounds (ROR3, CMIX, SMIX)	**SHAvite-3**	AES round
Groestl	Modified AES round	**SIMD**	4 steps of the compression function
Hamsi	Truncated Non-Linear Permutation P	**Skein**	8 rounds of Threefish-256

logic resources in hardware implementations are summarized in Table 2.2. Fixed shifts, fixed rotations and fixed permutations are not included as they use only routing resources.

2.4 Generalized Design Template

The general design template is based on the interface as described in the previous section.

2.4.1 Top Level

Top level consists of four control signals and two data buses. The control signals going into the controller are the signals indicating the availability of the input data and the readiness of the destination circuit to accept any output from the datapath unit. The output signals going out of the controller provide the handshake signals. Likewise, the data bus going into the datapath comes from the source circuit, providing both the message header and its body. The output from the datapath contains the message digest.

2.4.2 Datapath

For most designs, the datapath can be separated into two main units as shown in Figure 2.4, the message expansion unit and the hash core. The message expansion unit performs any necessary modifications to a message block before this block is processed by the hash core.

Table 2.2: Major operations of SHA-3 candidates (other than permutations, fixed shifts and fixed rotations)

	NTT	**Linear code**	**S-box**	**GF MUL**	**MUL**	**mADD**	**ADD /SUB**	**Boolean**
BLAKE						mADD3	ADD	XOR
BMW						mADD17	ADD, SUB	XOR
CubeHash							ADD	XOR
ECHO			AES 8x8	x02, x03				XOR
Fugue			AES 8x8	x04..x07				XOR
Groestl			AES 8x8	x02..x07				XOR
Hamsi		LC[128, 16,70]	Serpent 4x4					XOR
JH			Serpent 4x4	x2, x5				XOR
Keccak								NOT:AND:XOR
Luffa			4x4	x2				XOR
Shabal					x3, x5		ADD, SUB	NOT:AND:XOR
SHAvite-3			AES 8x8	x02, x03				NOT:XOR
SIMD	NTT_{128}				x185, x233	mADD3	ADD	NOT:AND:OR
Skein							ADD	XOR
SHA-256						mADD5		NOT:AND:XOR

The core mixes the current state and the expanded message block together to produce the next state, and finally (after all message blocks have been processed) the message digest.

2.4.3 Controller

The Controller, shown in Figure 2.5, is implemented using three main Finite State Machines, working in parallel, and responsible for the Input, Main Processing, and the Output, respectively. As a result, each circuit can simultaneously perform the following three tasks: output hash value for the previous message, process a current block of data, and read the next block of data. The parameters of the interface are selected in such a way that the time necessary to process one block of data is always larger or equal to the time necessary to read the next block of data. This way, the processing of long streams of data can happen at full speed, without any visible input interface overhead. A generalized design of Finite State Machine 1, 2 and 3 are shown in Figures 2.6, 2.7 and 2.8, respectively.

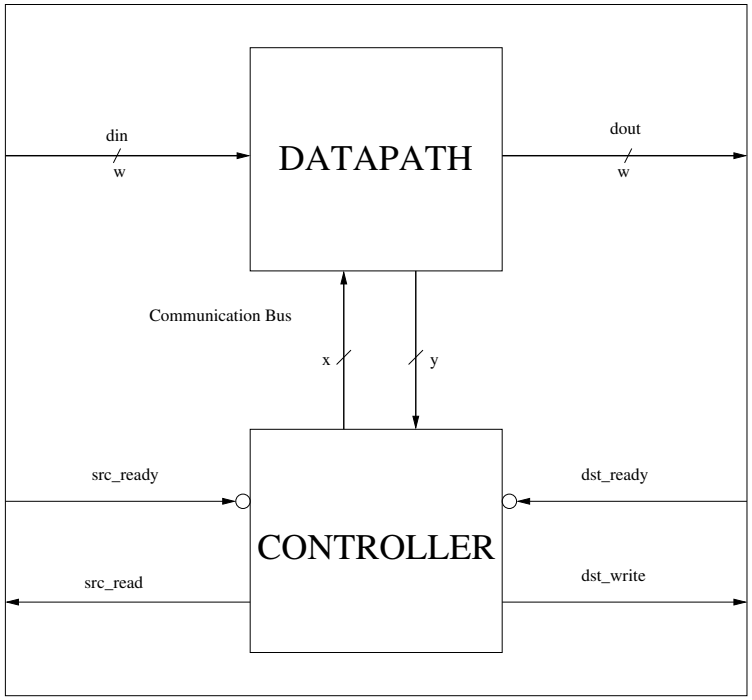


Figure 2.3: Generalized Top Level

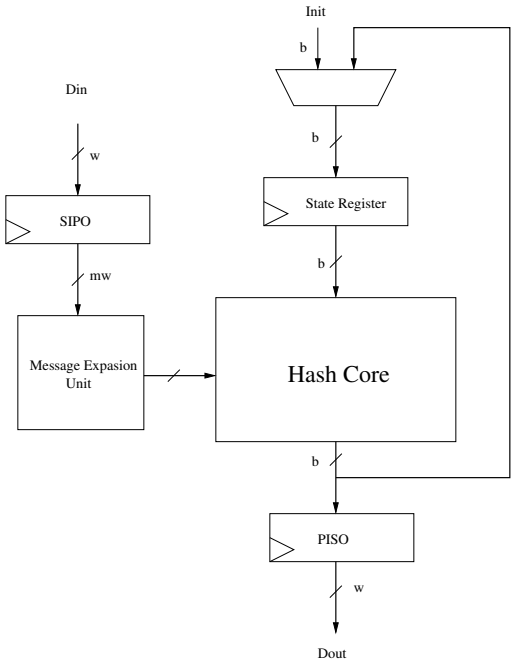


Figure 2.4: Generalized Datapath

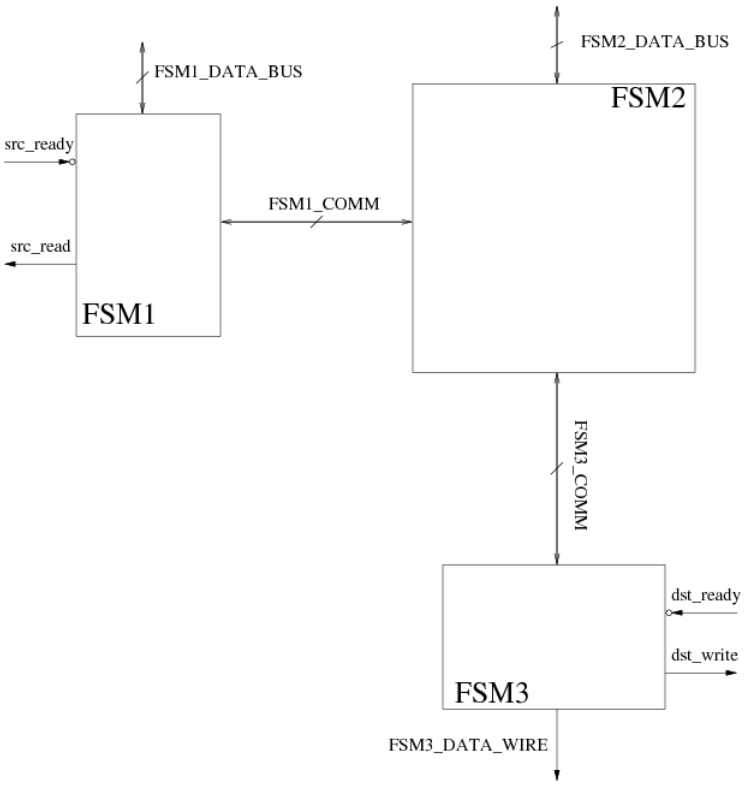


Figure 2.5: Generalized Top Level Controller

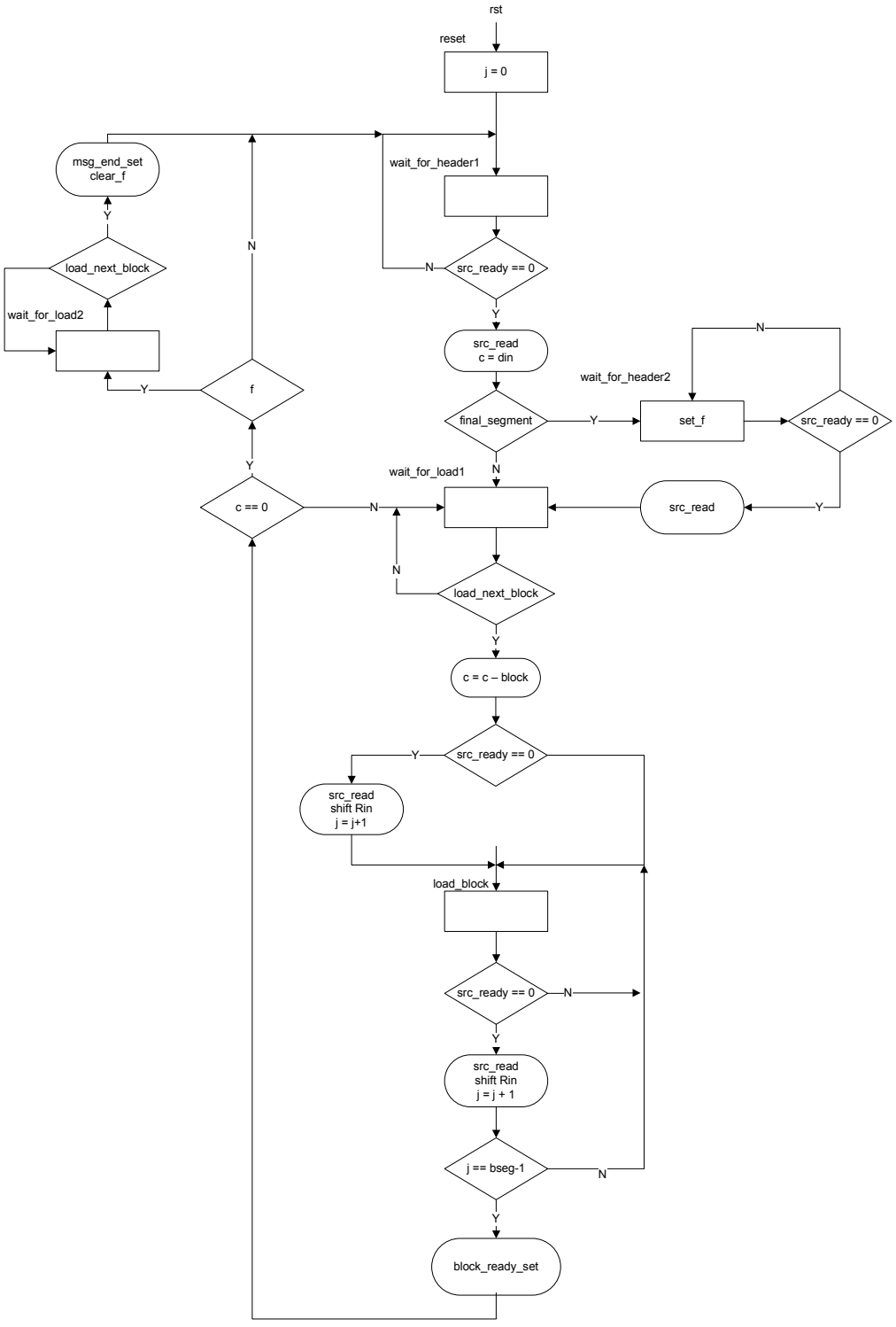


Figure 2.6: Generalized FSM 1 State Diagram

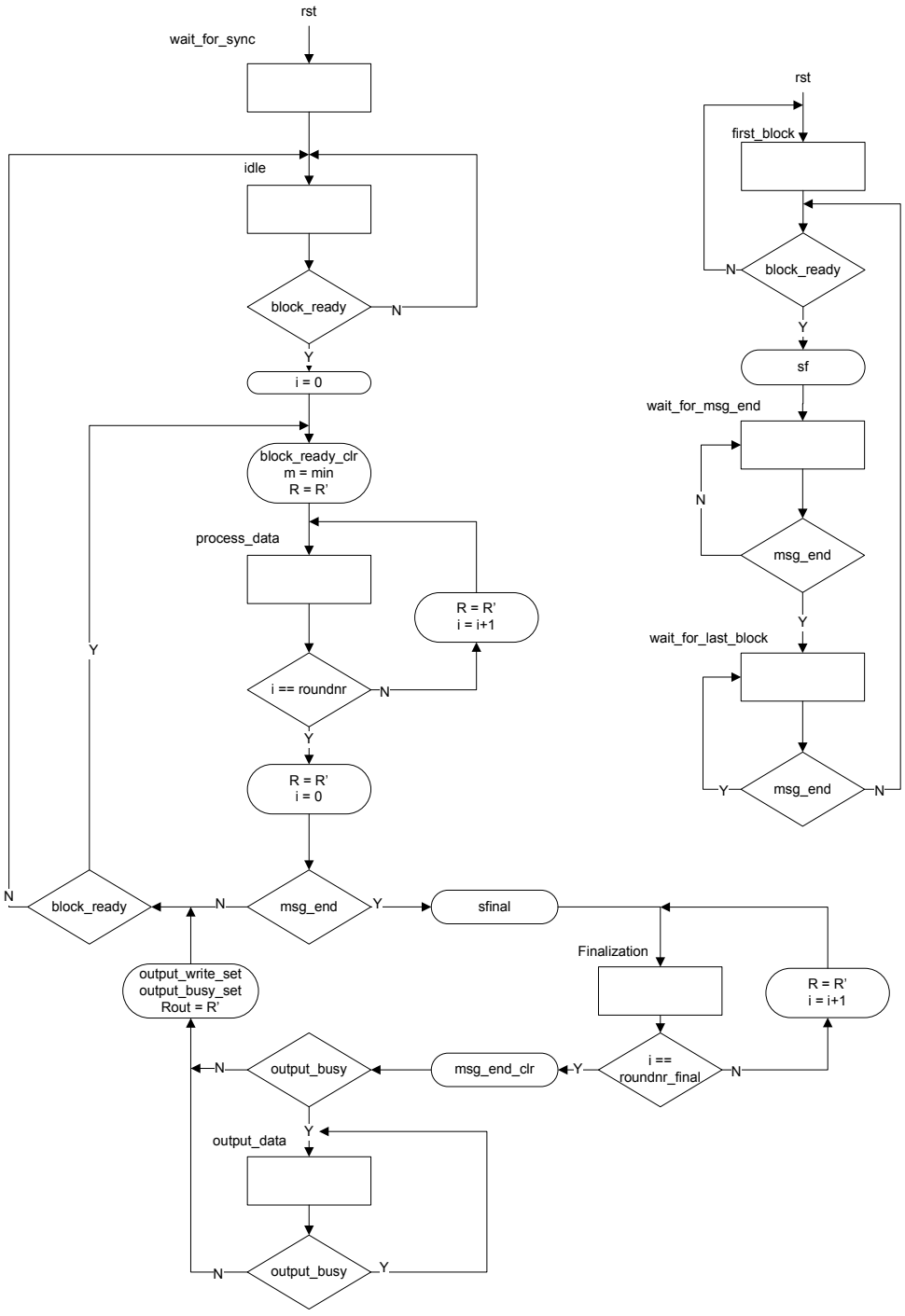


Figure 2.7: Generalized FSM 2 State Diagram

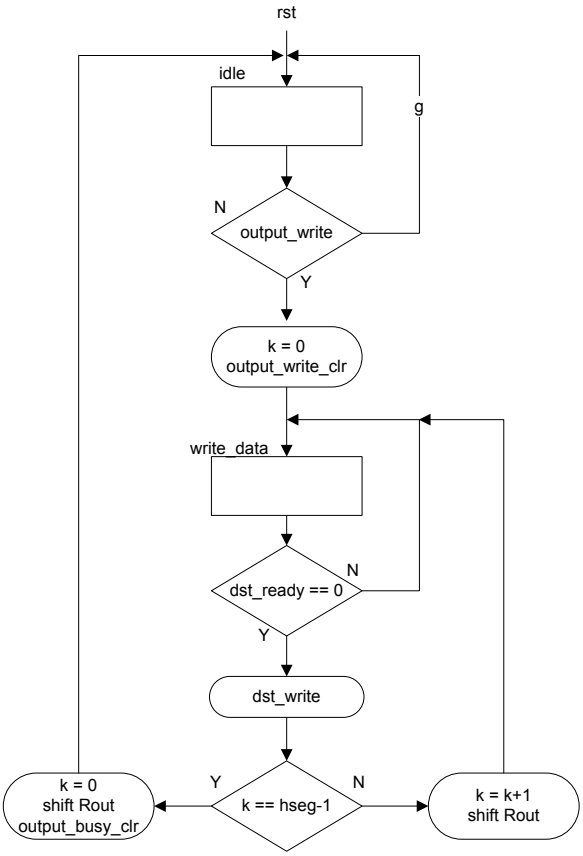


Figure 2.8: Generalized FSM 3 State Diagram

2.5 Performance Metric

Latency, Throughput, Area and Throughput to Area ratio are the basic performance metrics used in our study.

2.5.1 Latency

Latency is defined as the time to process a single message. This includes the time taken for reading data from the source circuit (typically Input FIFO), initialization, main processing, finalization, and writing result to the destination circuit (typically Output FIFO). The formula for Latency can be written as follows:

$$Latency = T \cdot HCycle(N) \tag{2.1}$$

where T is a clock period of a specific hash function, *HCycle(N)* is the total number of clock cycles required to process an N-block message.

The general formula for the time necessary to hash an N-blocks message can be written as:

$$HCycle(N) = c_{init} + \lceil \frac{c_{in}}{r_{io}} \rceil + c_{block} \cdot N + c_{final} + \lceil \frac{c_{out}}{r_{io}} \rceil \tag{2.2}$$

In this formula:

- c_{init} is the number of clock cycles necessary to establish communication with the source of data (typically, Input FIFO) and read the length of the message (in our formulas we assume that the length of the message is smaller than 2^w).
- c_{in} is the number of clock cycles required to read the very first block of the message. $c_{in} = Block\_size/w$.
- c_{block} is the number of clock cycles required to process one block of the message.
- c_{final} is the number of clock cycles required for the finalization. We assume that only one finalization is required per entire message (if the finalization needs to be repeated

for every block of the message, its number of clock cycles is included in c_{block}).

- c_{out} is the number of clock cycles required to write hash value to the destination circuit (typically Output FIFO). $c_{out} = output\_size/w$.
- r_{io} is the ratio between the frequency of the input/output clock, f_{ioclk}, and the frequency of the main clock, f_{clk}, $r_{io} = \lceil f_{ioclk}/f_{clk} \rceil$.

2.5.2 Throughput

Throughput is a measure of how many bits of a message can be processed within a given time. For hash functions, we define throughput based on the circuit performance for long messages. Hence, the time required for initialization and finalization (performed once per message) and message input/output are ignored. Under these assumptions, the formula for Throughput becomes:

$$Thr = \frac{Block\_size}{T \cdot (HCycle(N+1) - HCycle(N))} \tag{2.3}$$

where *Block_size* is the block size of a specific candidate and variant.

Alternatively, throughput can be represented as:

$$Thr = \frac{Block\_size}{T \cdot (\#cycles)} \tag{2.4}$$

where $\#cycles$ is the number of clock cycles required to process one block of the message.

Throughput can also be written as a formula based on latency as:

$$Thr = \frac{Block\_size}{(Latency(N+1) - Latency(N))} \tag{2.5}$$

where *Latency(N)* is the latency required to process an N-block message.

2.5.3 Area

Since our designs contain no dedicated resources such as Block RAMs, DSP units, multipliers, etc., area in our study is defined as follows:

$$Area = \#Basic\_elements \tag{2.6}$$

Basic_elements are specific to a given FPGA family. For Xilinx, the $\#Basic\_elements$ is defined as the number of CLB slices (#CLB_slices). For the Altera Cyclone families and Stratix, $\#Basic\_elements$ is defined as the number of Logic Elements (#LEs). For the Altera Stratix II and more recent high performance families, $\#Basic\_elements$ is defined as the number of Adaptive Look-up Tables (#ALUTs).

2.5.4 Throughput to Area Ratio

The formula for the Throughput to Area Ratio is:

$$Ratio = \frac{Thr}{Area} \tag{2.7}$$

where *Thr* and *Area* are the metrics described above.

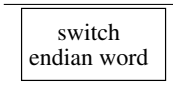
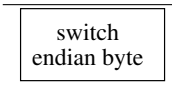
Chapter 3: Comprehensive Designs of SHA-3 Candidates

The designs of all 14 SHA-3 candidates followed the same basic design principle with the core separated into two main units, the Datapath and the Controller (see Section 1.2). Only Datapath diagrams are provided in this chapter as the Controller can be derived from the Datapath and the specification of the function, and described using ASM charts similar to those given in Section 1.2. The full specification of each of the algorithms can be found in [23–37].

3.1 Notations and Symbols

Table 3.1 provides the notation and symbols that are being used throughout this chapter.

Table 3.1: Notations and Symbols

Word	A group of bits used in arithmetic and logic operations, typically of the size of 32 or 64 bits.
Block	A group of words.
X[i]	Refers to an array position i in X.
X_i	Refers to a bit position i in X.
salt	Salt values are always assumed to be zero and as a result they are omitted from the diagrams.
b	Block size in bits.
h	Hash value size in bits.
w	Word size in bits.
IV	Initialization vector
SEXT	Sign extension.
ZEXT	Zero extension.
<<<R	Rotation left by R positions. If R is a constant: fixed rotation; if R is a variable: variable rotation implemented using a barrel rotator.
>>>R	Rotation right by R positions. If R is a constant: fixed rotation; if R is a variable: variable rotation implemented using a barrel rotator.
<<S	Shift left by S positions. If S is a constant: fixed shift; if S is a variable: variable shift implemented using a barrel shifter.
>>S	Shift right by S positions. If S is a constant: fixed shift; if S is a variable: variable shift implemented using a barrel shifter.
||	Concatenation. By default, the buses concatenate back to the same arrangement as before the separation (split) occurs.
	Serial-in-parallel-out unit.
	Parallel-in-serial-out unit.
	Wordwise endianness switching.
	Bytewise endianness switching.

3.2 Basic Component Description

This section describes implementations of basic components used in more than one algorithm. These components include multiplication by 2 in GF(2^8), SubBytes, MixColumns, and AES Round.

3.2.1 Multiplication by 2 in the Galois Field GF(2^8)

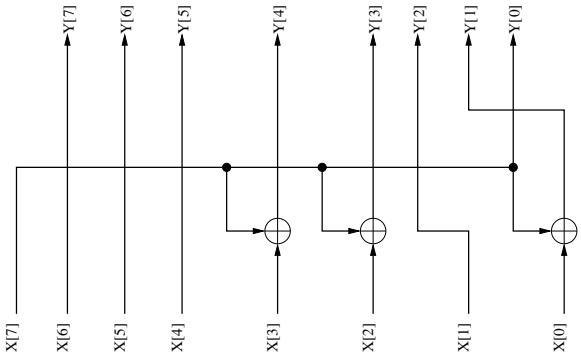


Figure 3.1: Base : X2

3.2.2 Multiplication by n in the Galois Field GF(2^8)

Galois Field multiplication by n other than 2 is summarized in Table 3.2.

Table 3.2: Galois Field Multiplication by n

$x3 = x2(X) \oplus X$
$x4 = x2(x2(X))$
$x5 = x4(X) \oplus X$
$x6 = x4(X) \oplus x2(X)$
$x7 = x4(X) \oplus x3(X)$

3.2.3 AES

AES is a basic building block of many SHA-3 candidates. An AES round consists of three basic operations, SubBytes, MixColumns and ShiftRow shown in Figure 3.2. SubByte operation, shown in Figure 3.3, performs direct substitution on all bytes of its input. MixColumns performs matrix multiplication on each word of its input. A word of AES contains

32-bit. Hence, 4 instances of MixColumn are required. SBOX and ShiftBytes operation of AES and its full specification can be found from [38].

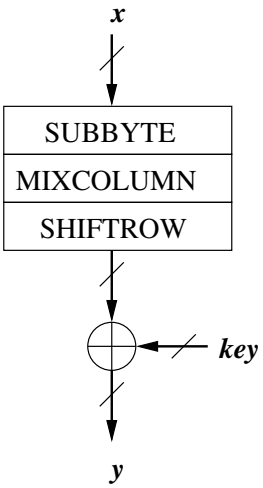


Figure 3.2: Base : AES Round

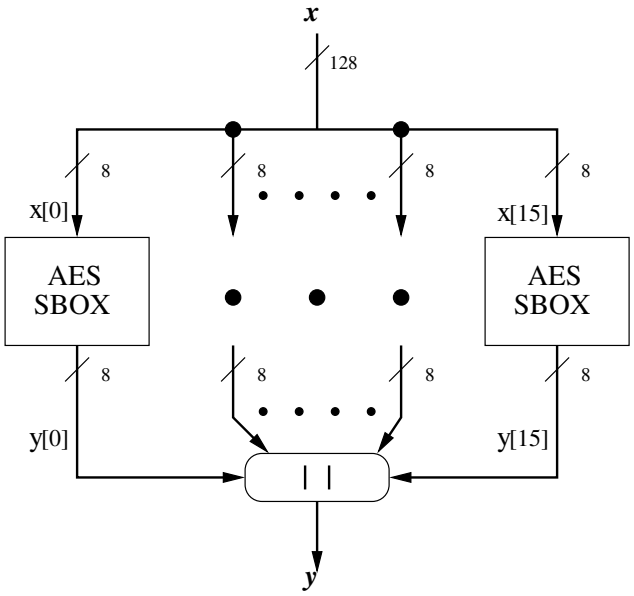


Figure 3.3: Base : AES SubBytes

Note : All buses are 8 bits

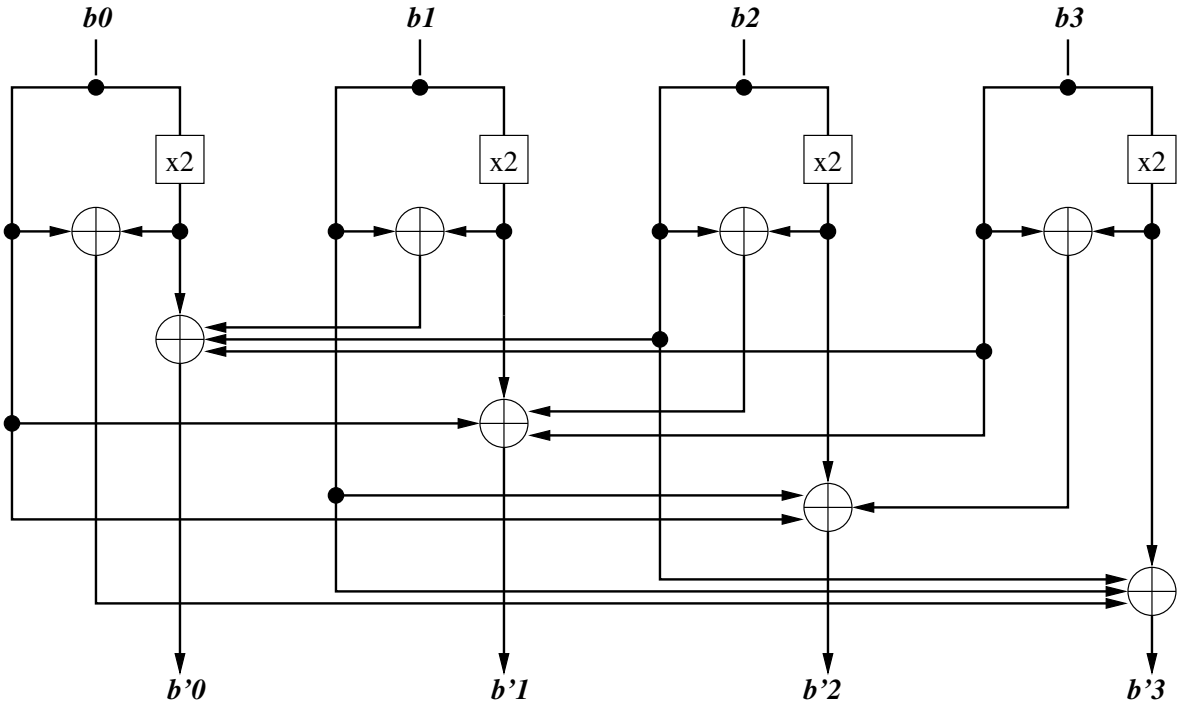


Figure 3.4: Base : AES MixColumn

3.3 Blake

3.3.1 Block Diagram Description

Figure 3.5 shows the datapath of BLAKE. In this design, the combinational CORE implements one half of the BLAKE's round [23]. Thus, two clock cycles are necessary to implement the full round. First, a message block is loaded into SIPO. Once done, the block is stored in a temporary register, used to hold the message block until this block is fully processed by the CORE. This temporary register allows the next message block to be loaded simultaneously into SIPO. The message block msg and the constant c are then permuted and passed to the design's CORE. Simultaneously, the chaining value is initialized with the the Initialization Vector, IV, and an input to the CORE, V, is initialized with the value dependent on the chaining value, the counter, t, and a lower half of the constant c. The initial value of V is mixed by the CORE with the output of the block PERMUTE, CM, for twenty clock cycles (10 rounds). Once this operation is completed, an additional clock cycle is required for finalization. The output of Finalization is used as the next chaining value, for intermediate message blocks, or as the final hash value for the last message block.

The Initialization unit performs the following function:

$$\begin{pmatrix} v[0] & v[1] & v[2] & v[3] \\ v[4] & v[5] & v[6] & v[7] \\ v[8] & v[9] & v[10] & v[11] \\ v[12] & v[13] & v[14] & v[15] \end{pmatrix} \longleftarrow \begin{pmatrix} h[0] & h[1] & h[2] & h[3] \\ h[4] & h[5] & h[6] & h[7] \\ c[0] & c[1] & c[2] & c[3] \\ t[0] \oplus c[4] & t[1] \oplus c[5] & c[6] & c[7] \end{pmatrix}$$

The Finalization unit performs the following operation:

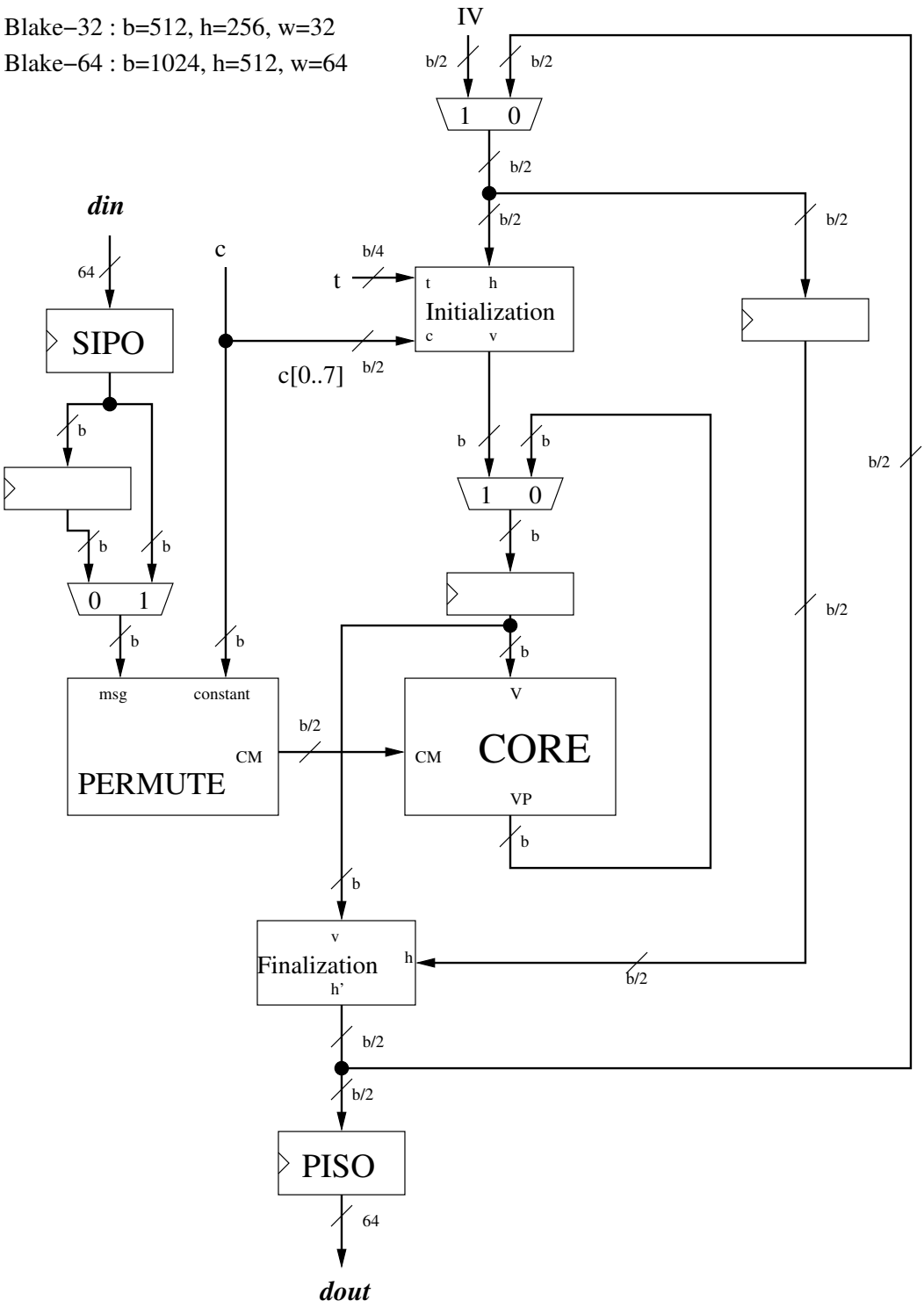


Figure 3.5: Blake : Datapath

Table 3.3: Blake: Permutation Constant

	hi								*low*							
σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

$$h'[0] \leftarrow h[0] \oplus v[0] \oplus v[8]$$
$$h'[1] \leftarrow h[1] \oplus v[1] \oplus v[9]$$
$$h'[2] \leftarrow h[2] \oplus v[2] \oplus v[10]$$
$$h'[3] \leftarrow h[3] \oplus v[3] \oplus v[11]$$
$$h'[4] \leftarrow h[4] \oplus v[4] \oplus v[12]$$
$$h'[5] \leftarrow h[5] \oplus v[5] \oplus v[13]$$
$$h'[6] \leftarrow h[6] \oplus v[6] \oplus v[14]$$
$$h'[7] \leftarrow h[7] \oplus v[7] \oplus v[15]$$

In Figure 3.6, an operation of the BLAKE's PERMUTE module is presented. A new value of the variable m is selected depending on the round number using a wide multiplexer preceded by constant permutations. A permutation table is shown in Table 3.3. The selection signal of the multiplexer cycles from 0 to 19 (and then again back to 0 for BLAKE-64) until all BLAKE's rounds are executed. Each output of the multiplexer is then mixed with the respective constant using the transformation XOR_W_CROSS, defined in the note to Fig. 3.6, and registered afterwards.

The CORE unit is shown in Figure 3.7 and represents one half of the BLAKE's round. As specified in [23], there are two levels of G functions and therefore a permutation between the first and the second half-round is required. This permutation is performed wordwise and is shown in Table 3.4. LVL2 permute transforms the state matrix (output of 4 parallel

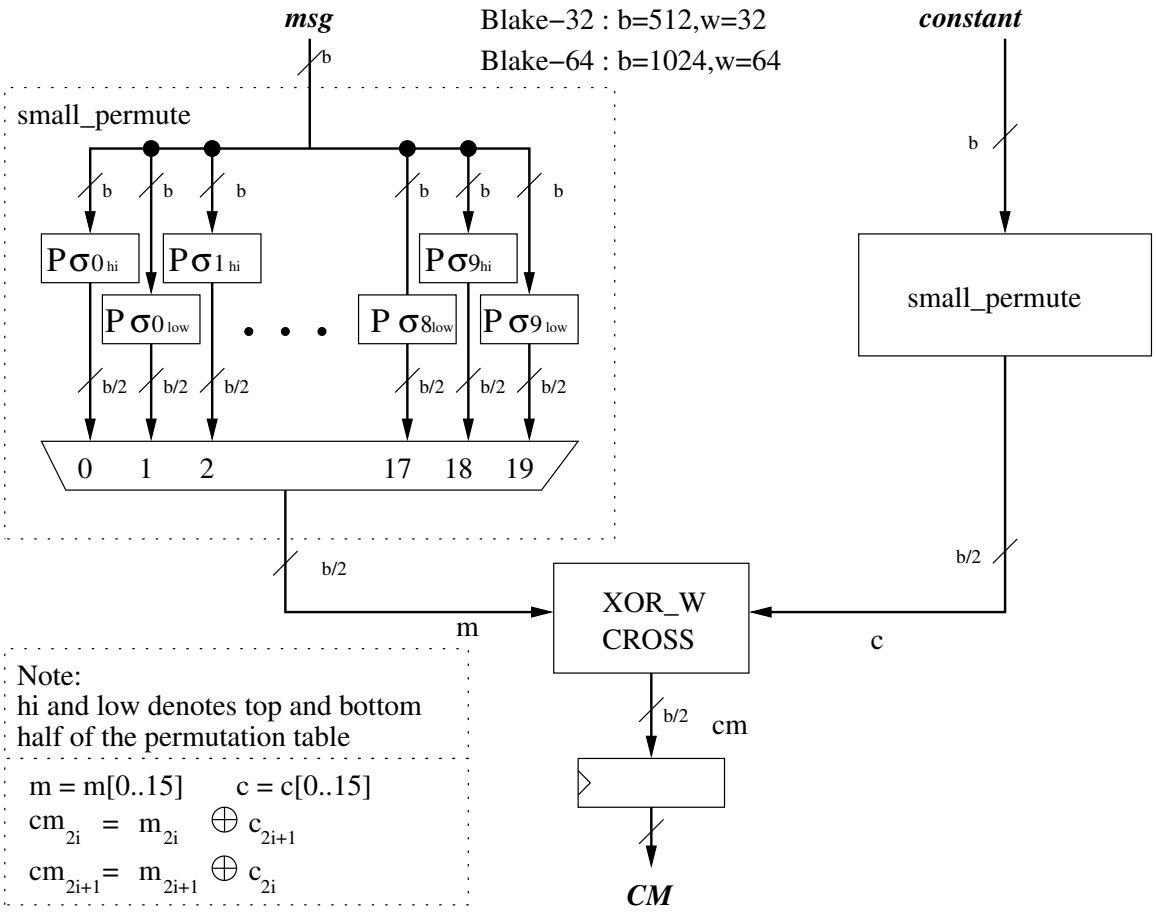


Figure 3.6: Blake : PERMUTE

G functions) into a new matrix appropriate for the second half-round. LVL1 permute is a permutation inverse to LVL2 permute.

Table 3.4: Blake : Half Round's Permutation

LVL2 (forward)	0	1	2	3	5	6	7	4	10	11	8	9	15	12	13	14
LVL1 (revert)	0	1	2	3	7	4	5	6	10	11	8	9	13	14	15	12

The G-function in the CORE unit is shown in Figure 3.8. Note that the XOR operations used to calculate input values CM_{2i} and CM_{2i+1}, which are normally depicted as a part of the G-function, are omitted in our design. These operations were placed as a part of the PERMUTE unit and therefore skipped here. R1, R2, R3 and R4 are rotating constants. The values of these constants are shown in Table 3.5

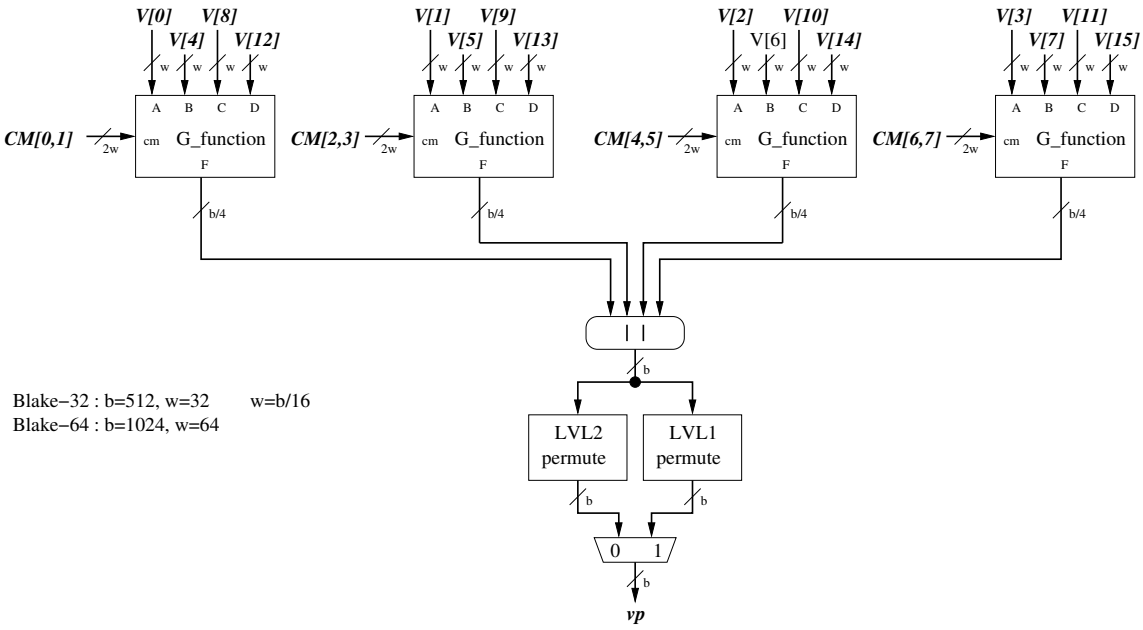


Figure 3.7: Blake : CORE

Table 3.5: Blake: Rotating Constants of Blake-32

R1	**R2**	**R3**	**R4**
16	12	8	7

3.3.2 256 vs 512 Variant Differences

Blake-64 doubles the word size of Blake-32, thereby increasing the block size as well. Hence, the IV and the constant are changed from 512 bits to 1024 bits. These values can be found in Section 2.2.1 of [23]. Blake-64 introduces also an increase in the number of mixing rounds from 10 to 14. As a result, the number of clock cycles required in our design for processing a single block of message increases from 21 to 29. The multiplexer selection signal in the PERMUTE unit loops back when the round number reaches 10. Hence, after reaching 19, this selection signal goes back to 0. Finally, the rotating constants are adjusted to reflect the increased word size. These values are described in Table 3.6.

Table 3.6: Blake: Rotating Constants of Blake-64

R1	**R2**	**R3**	**R4**
32	25	16	11

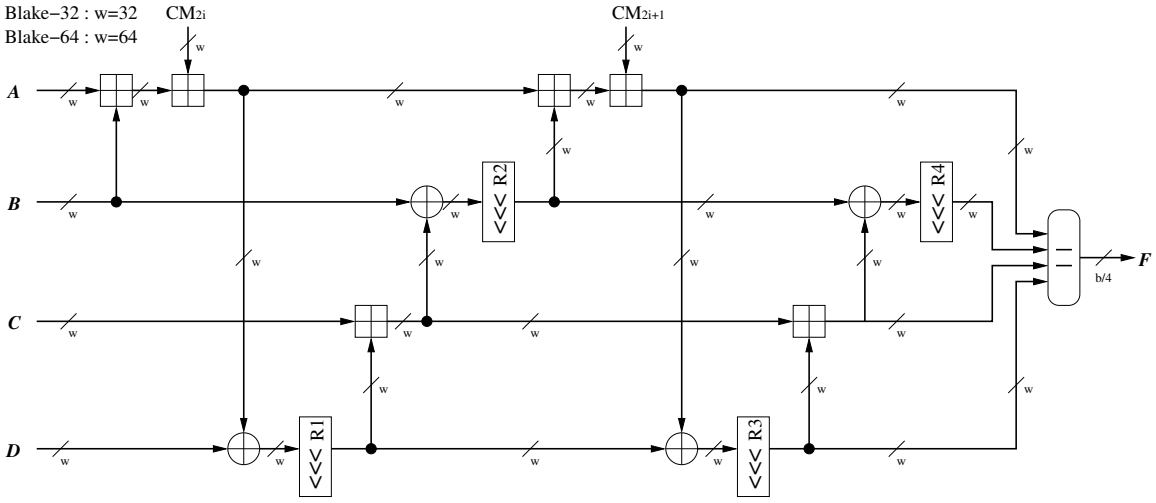


Figure 3.8: Blake : G-function

3.4 Blue Midnight Wish (BMW)

3.4.1 Block Diagram Description

Our design for Blue Midnight Wish (BMW) hashes a block of data within one clock cycle. Since the number of clock cycles necessary to read a block of a message is greater than the number of clock cycles required to hash it, an additional clock signal is used in the circuit as shown in Figure 3.9. This faster clock (*io_clk*) is used to drive the SIPO and PISO units, allowing them to read and write data at a faster rate than the operation of other units in the circuit. The rate of reading and writing is determined by the block size and the number of cycles required to process a block. Since only one clock cycle is used to process a message block, the frequency of *io_clk* is *block_size/word_size* times higher than the main clock. This ratio is equal to 8 for BMW-256. BMW requires each message block to go through the endianness switching before the start of processing. A message block is then mixed with the chaining value to obtain the next chaining value. Once all blocks of the message are processed, a finalization round is initiated. Since there is no incoming message block, the chaining value and the input message block are replaced by the constant and the chaining value, respectively. The descriptions of F0, F1, F2 and AddElements and its associated logical operations can be found in Table 1.3 and Table 2.2-2.4 in [24].

3.4.2 256 vs 512 Variant Differences

BMW-512 increases the word size of BMW-256 from 32 to 64 bits. As a result, the block size is doubled as well. Since the block size increases, the number of clock cycles required to load a message block also increases for *io_clk* from 8 cycles to 16 cycles. Furthermore, logic functions, specifically shifts and rotations, are adjusted to accommodate the increased word size. These changes are shown in Table 1.3 of [24]. All other operations remain the same.

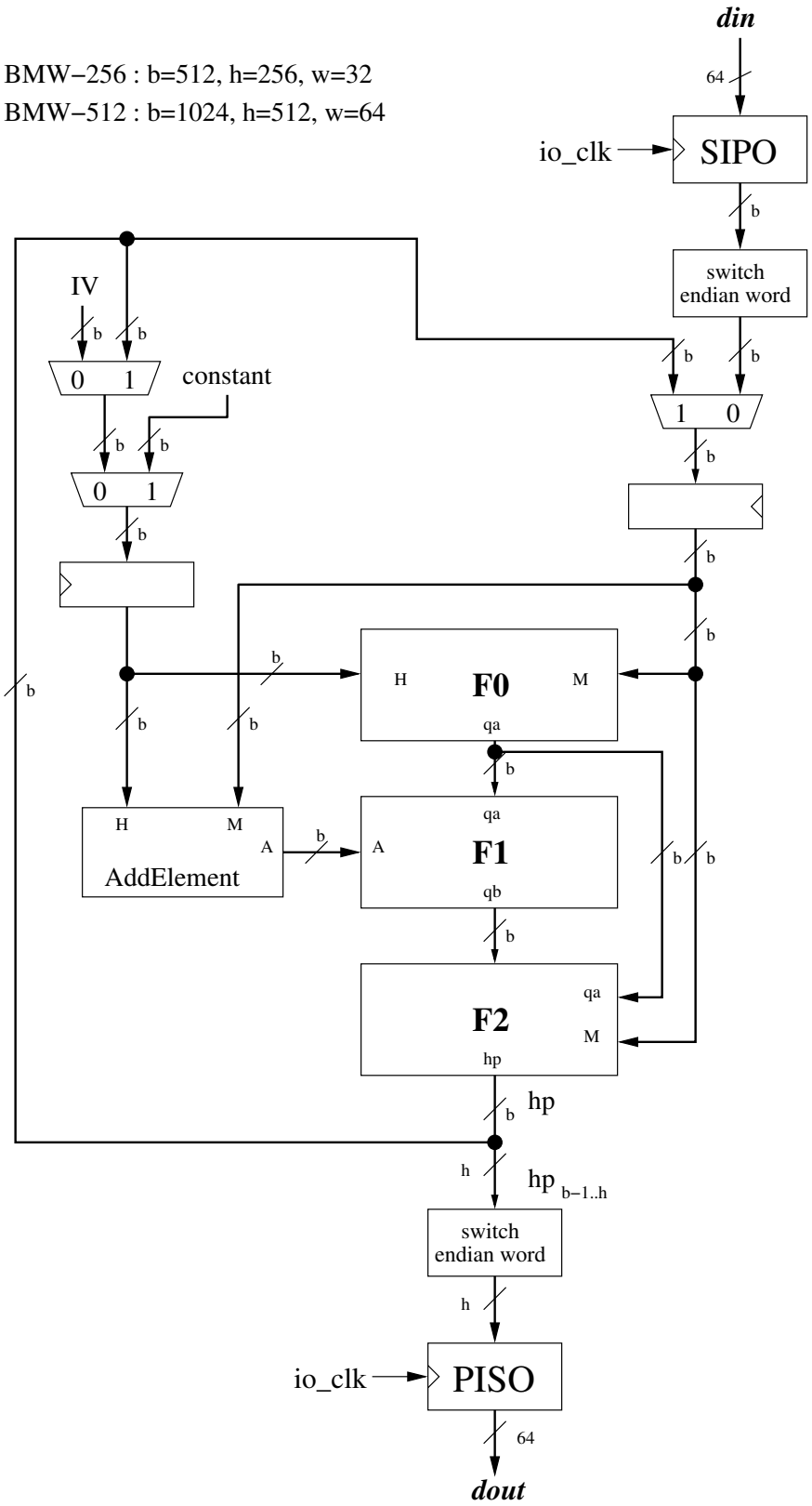


Figure 3.9: BMW : Datapath

3.5 CubeHash

3.5.1 Block Diagram Description

A straight-forward iterative architecture is used in our design. The datapath of CubeHash is shown in Figure 3.10. Due to endianness issue, the input message is required to go through endianness switching twice. First, the bytewise endianness switching is applied, which is then followed by the wordwise endianness switching. A word of CubeHash consists of 32 bits.

For each message, a chaining value is initialized to IV. The 256 leftmost bits of the chaining value are xored with an input message block. The state is then transformed for 16 rounds. A round is described in Figure 3.11. All operations inside the round are performed wordwise. This process repeats until all message blocks are processed. In the last round of the last message block, an integer one is xored with the position zero of the chaining value, rp, by activating the control signal $final$ before the chaining value is inserted back into the state register. Then, the chaining value is transformed for 160 rounds to get the final hash value. The hash value is required to go through the endianness switching process again to reach the correct hash output.

3.5.2 256 vs 512 Variant Differences

Everything is the same for both variants with the exception of truncation size. CubeHash16/32-256 truncates the state to 256 bits to obtain the hash value, as opposed to CubeHash16/32-512 which truncates the state to 512 bits.

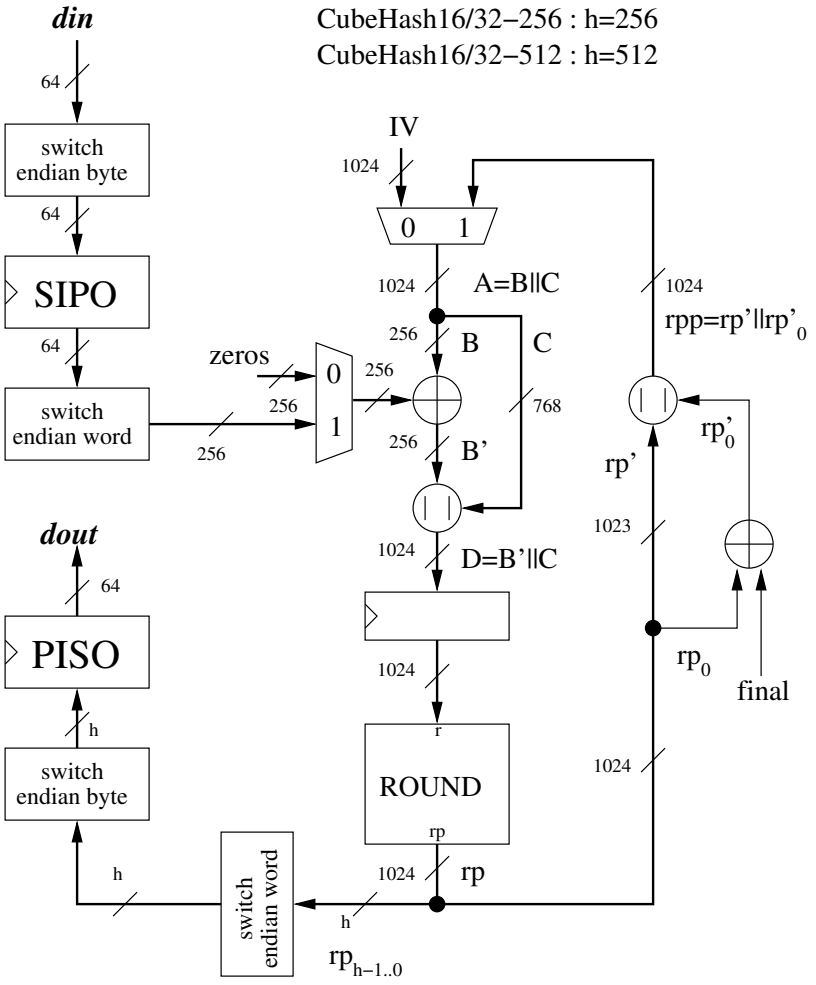


Figure 3.10: CubeHash : Datapath

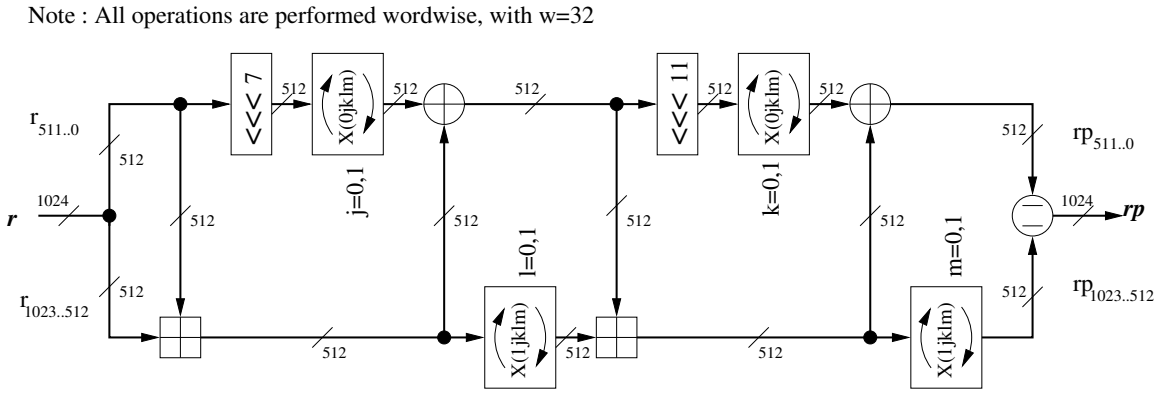


Figure 3.11: CubeHash : Round

3.6 ECHO

3.6.1 Block Diagram Description

ECHO's top level datapath is shown in Figure 3.12. A message block is first concatenated with the chaining value to produce the state matrix. The state matrix is viewed as an array of 16 words with each word representing 128 bits. The state then goes through 10 rounds of iteration for ECHO-256. Note that C represents the number of bits hashed so far. This value also includes bits of the currently processed block. Once the state matrix is thoroughly mixed, a new chaining value is computed from the state matrix by the BIG.Final unit. This operation is described as follows:

$$v'[0] \leftarrow v[0] \oplus m[0] \oplus m[4] \oplus m[8] \oplus w[0] \oplus w[4] \oplus w[8] \oplus w[12]$$
$$v'[1] \leftarrow v[1] \oplus m[1] \oplus m[5] \oplus m[9] \oplus w[1] \oplus w[5] \oplus w[9] \oplus w[13]$$
$$v'[2] \leftarrow v[2] \oplus m[2] \oplus m[6] \oplus m[10] \oplus w[2] \oplus w[6] \oplus w[10] \oplus w[14]$$
$$v'[3] \leftarrow v[3] \oplus m[3] \oplus m[7] \oplus m[11] \oplus w[3] \oplus w[7] \oplus w[11] \oplus w[15]$$

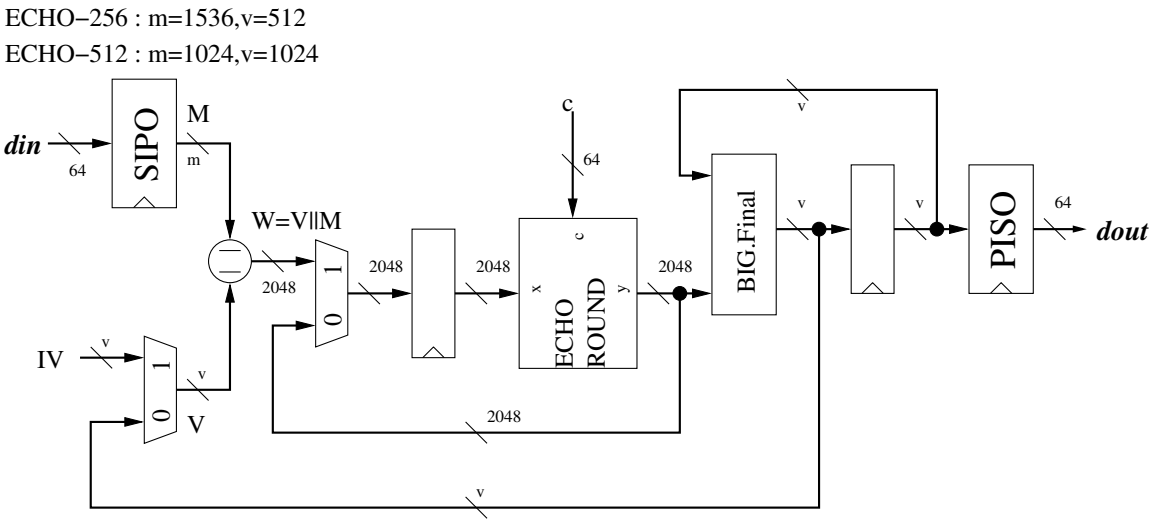


Figure 3.12: ECHO : Datapath

In Figure 3.13, operations inside of the ECHO round are shown. In our design, each ECHO round is executed in three clock cycles. BIG.SubBytes is performed in the first

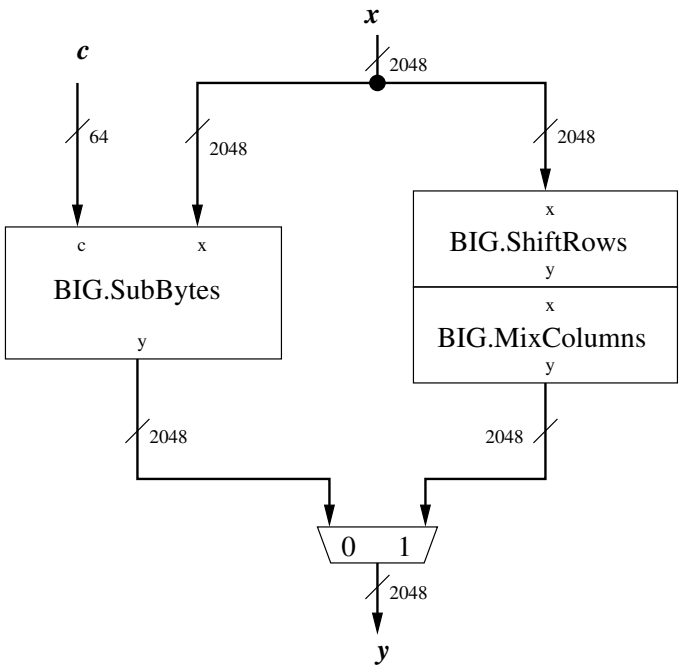


Figure 3.13: ECHO : Round

two clock cycles and BIG.ShiftRows and BIG.MixColumns in the third cycle. BigSubBytes operation is shown in Figure 3.14. The unit takes in the state matrix and the message length counter, C, and produces the next state. In the first clock cycle of the round operation, the key is chosen to be the length counter plus the numbers between 0 and 15. These added values follow the word number. Hence, the fourteenth word gets the key as $C + 14$. In the next cycle, salt is selected as the key. Since in our implementation, salt is not used, zero is selected instead.

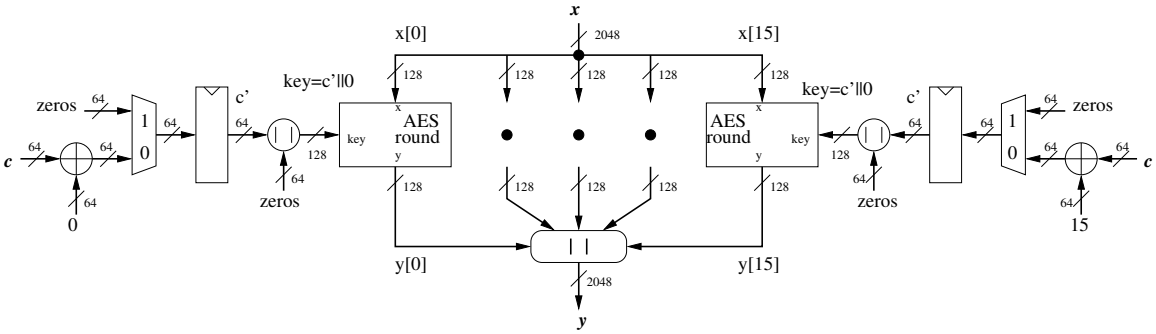


Figure 3.14: ECHO : BIG.SubBytes

Two operations are performed in the third cycle of a round. First, BIG.ShiftRows is performed. This operation is equivalent to the word permutation given in Table 3.7. Next, BIG.MixColumns transforms the permuted state to obtain the final value of a round. In Figure 3.15, a diagram of BIG.MixColumns is shown. BIG.MixColumns separates the state into 4 blocks, each block containing 4 words. A byte of data from each word is selected to go through the AES MixColumn. All data is then combined together to produce the final state.

Table 3.7: ECHO : BIG.ShiftRows

Word	0	1	2	3	5	6	7	4	10	11	8	9	15	12	13	14
Permuted Word	0	5	10	15	4	9	14	3	8	13	2	7	12	1	6	11

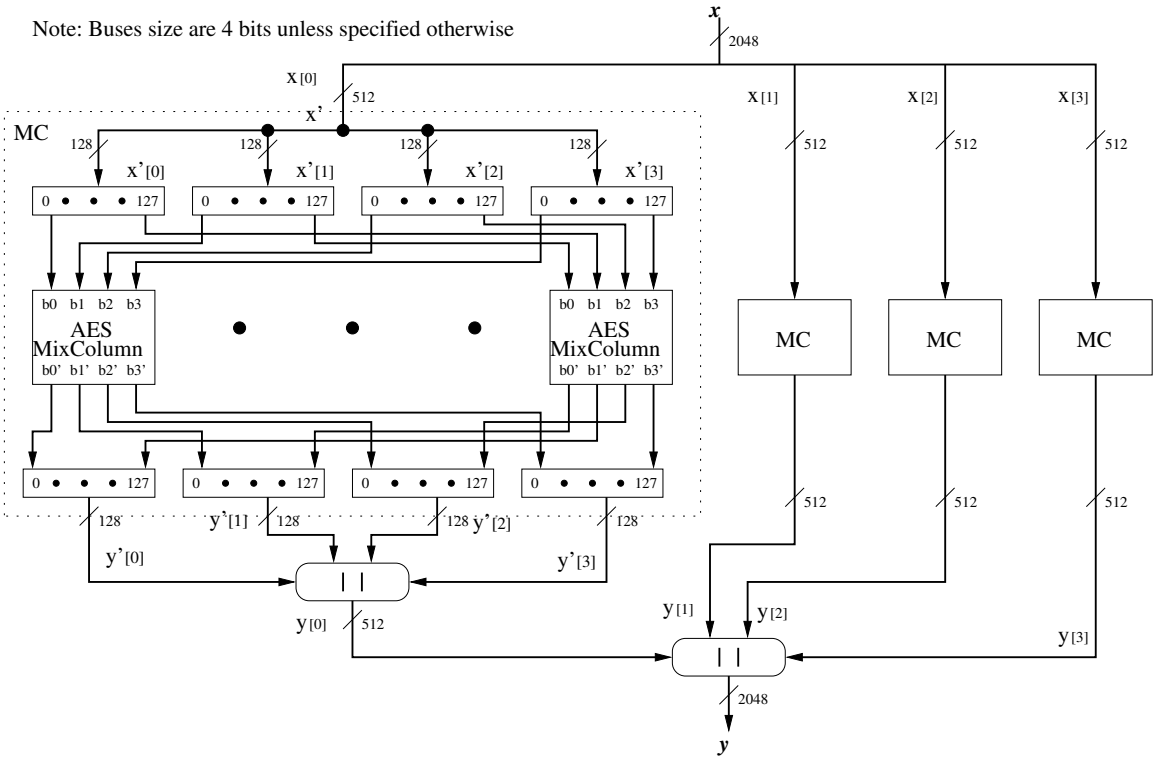


Figure 3.15: ECHO : BIG.MixColumns

3.6.2 256 vs 512 Variant Differences

ECHO-512 differs from ECHO-256 in its message block and chaining value sizes. The message block is reduced from 1536 bits to 1024. On the other hand, the chaining value is increased from 512 to 1024 bits. This change increases the security of ECHO and therefore a smaller number of rounds is used. Only 8 rounds are used in ECHO-512. Finally, only BIG.Final is altered. ECHO-512's BIG.Final is described as follows:

$$
\begin{aligned}
v'[0] &\leftarrow v[0] \oplus m[0] \oplus w[0] \oplus w[8] \\
v'[1] &\leftarrow v[1] \oplus m[1] \oplus w[1] \oplus w[9] \\
v'[2] &\leftarrow v[2] \oplus m[2] \oplus w[2] \oplus w[10] \\
v'[3] &\leftarrow v[3] \oplus m[3] \oplus w[3] \oplus w[11] \\
v'[4] &\leftarrow v[4] \oplus m[4] \oplus w[4] \oplus w[12] \\
v'[5] &\leftarrow v[5] \oplus m[5] \oplus w[5] \oplus w[13] \\
v'[6] &\leftarrow v[6] \oplus m[6] \oplus w[6] \oplus w[14] \\
v'[7] &\leftarrow v[7] \oplus m[7] \oplus w[7] \oplus w[15]
\end{aligned}
$$

3.7 Fugue

3.7.1 Block Diagram Description

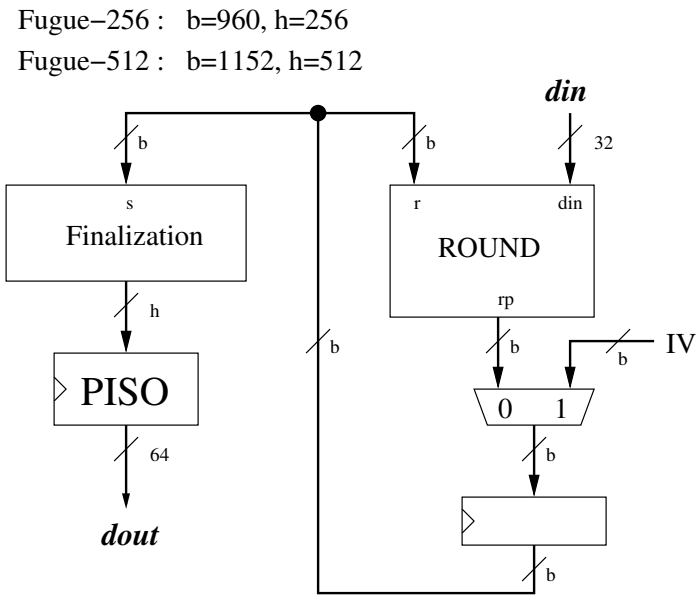


Figure 3.16: Fugue : Datapath

In Figure 3.16, the datapath of Fugue is shown. For every message, the state register is initialized to IV. The state is viewed as a matrix of 4 by X bytes, where X is the column length dependent on the block size of Fugue. For Fugue-32, the block size is equal to 960 bits. Hence, the matrix have dimensions 4 x 30. The state is mixed with input message blocks through the ROUND unit. Once all message blocks are processed, the state goes through Finalization. For Fugue-32, Finalization is described below:

$$Finalization = \begin{array}{c} S[1..3] \;||\; (S[4] \oplus S[0]) \;|| \\ (S[15] \oplus S[0]) \;||\; S[16..18] \end{array}$$

A round of Fugue is shown in Figure 3.17. The path through the ROUND unit is selected based on the sequence of operations as described in Section 4.3.5 of F-256 in [27]. TIX operates in parallel as follows:

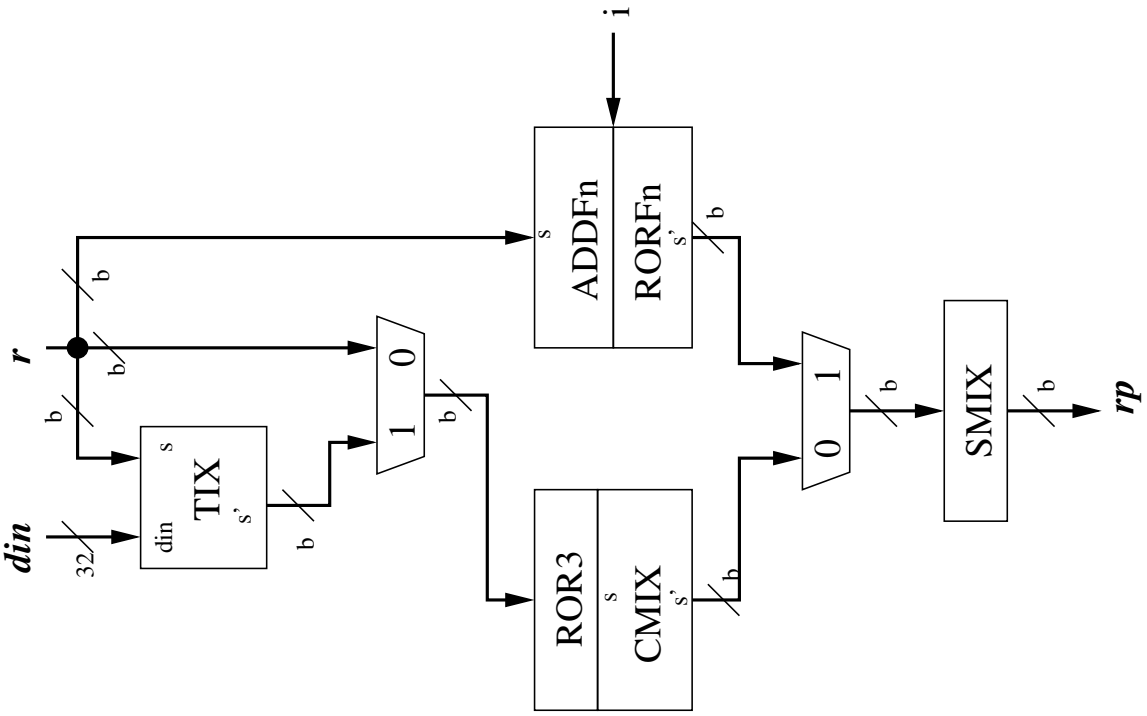


Figure 3.17: Fugue : Round

$$S'[0] = din$$
$$S'[1] = S[1] \oplus S[24]$$
$$S'[8] = S[8] \oplus din$$
$$S'[10] = S[10] \oplus S[0]$$

ROR3 and CMIX are performed consecutively. All RORn operations are bytewise rotations by n bytes. This is equivalent to $>>> (n * 8)$. As such, ROR3 can be considered as $>>> 24$. CMIX operates as follows:

$$\begin{aligned}
S'[0] &= S[0] \oplus S[4] \\
S'[1] &= S[1] \oplus S[5] \\
S'[2] &= S[2] \oplus S[6] \\
S'[15] &= S[15] \oplus S[4] \\
S'[16] &= S[16] \oplus S[5] \\
S'[17] &= S[17] \oplus S[6]
\end{aligned}$$

Table 3.8: Fugue: F-256 ADDFn and RORFn Operation

i	y
0	$S'[4] = S[4] \oplus S[0]$ $S'[15] = S[15] \oplus S[0]$ $ROR15$
1	$S'[4] = S[4] \oplus S[0]$ $S'[16] = S[16] \oplus S[0]$ $ROR14$

Table 3.9: Fugue: Matrix Multiplier Table

	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]	X[9]	X[10]	X[11]	X[12]	X[13]	X[14]	X[15]
Y[0]	1	4	7	1	1				1				1			
Y[1]		1			1	1	4	7		1				1		
Y[2]			1				1		7	1	1	4			1	
Y[3]				1				1				1	4	7	1	1
Y[4]						4	7	1	1				1			
Y[5]		1							1	4	7			1		
Y[6]			1				1						7	1		4
Y[7]	4	7	1					1				1				
Y[8]					7				6	4	7	1	7			
Y[9]		7								7			1	6	4	7
Y[10]	7	1	6	4			7								7	
Y[11]				7	4	7	1	6				7				
Y[12]					4				4				5	4	7	1
Y[13]	1	5	4	7						4				4		
Y[14]			4		7	1	5	4							4	
Y[15]				4				5	4	7	1	5				

The ADDFn and RORFn operations are selected by the i control signal. The selection process is described in Table 3.8.

Finally, the SMIX operation is described in Figure 3.18. The SMIX operation first splits an input into an array of 128-bit blocks. Then, each block is further splitted into 16 bytes. These bytes are transformed using AES SBOX and the resulting vector of 16 bytes is used as an input to the Matrix Multiplier. The Matrix Multiplier performs multiplication of a constant matrix by an input vector. The value of the constant matrix is shown in Table 3.9. The multiplication is performed and based on the multiplication by 2 in $GF(2^8)$.

3.7.2 256 vs 512 Variant Differences

Fugue-512 increases the state size to 4 x 36 which is equivalent to 1152 bits. Additionally, TIX, CMIX, ADDFn, RORFn and Finalization have been modified. TIX is now performed

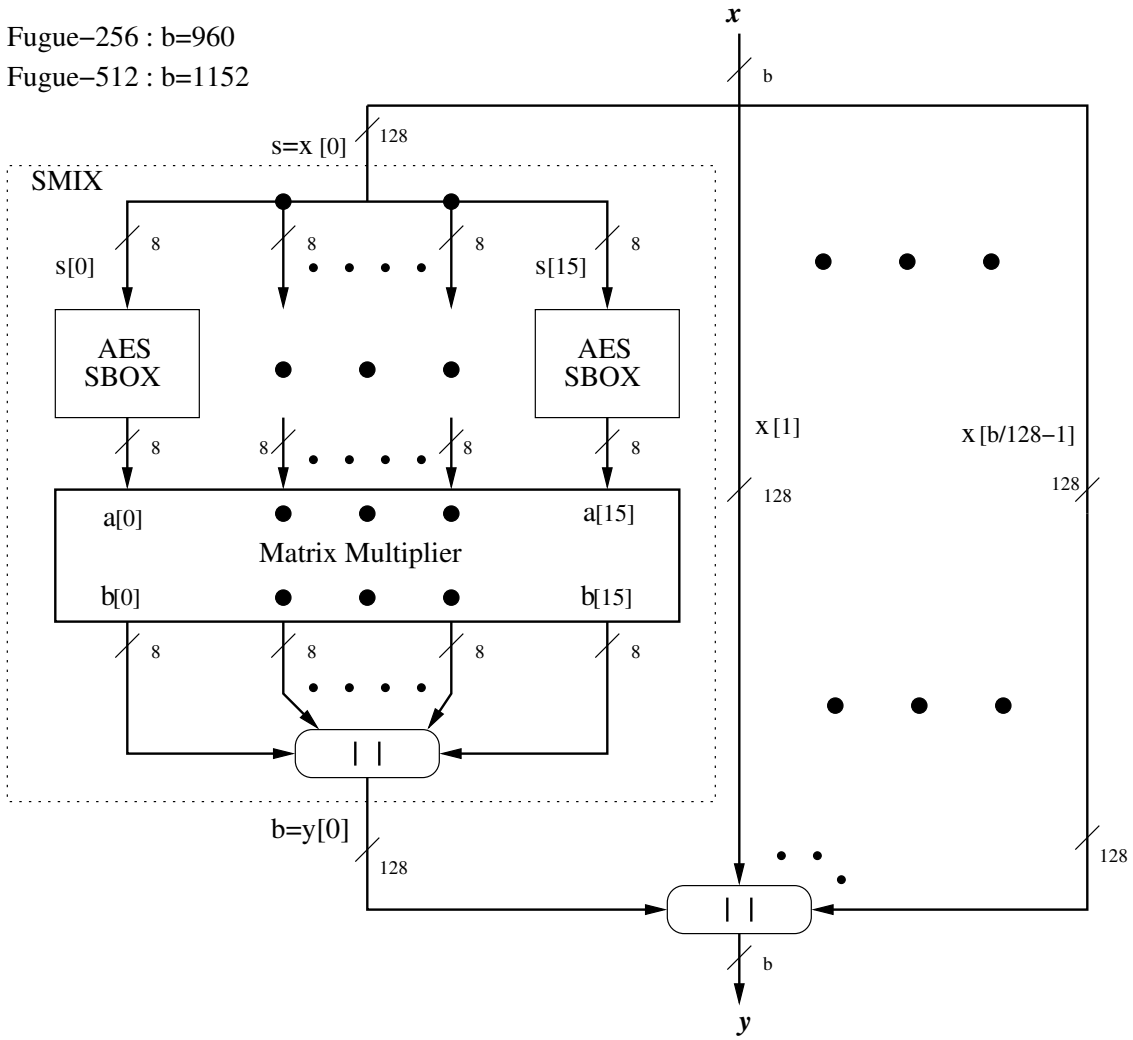


Figure 3.18: Fugue : SMIX

in parallel as follows:

$$S'[0] = din$$
$$S'[1] = S[1] \oplus S[24]$$
$$S'[4] = S[4] \oplus S[27]$$
$$S'[7] = S[7] \oplus S[30]$$
$$S'[8] = S[8] \oplus din$$
$$S'[22] = S[22] \oplus S[0]$$

CMIX is now performed as follows:

Table 3.10: ADDFn and RORFn Operation

i	y	i	y
0	$S'[4] = S[4] \oplus S[0]$ $S'[9] = S[9] \oplus S[0]$ $S'[18] = S[18] \oplus S[0]$ $S'[27] = S[27] \oplus S[0]$ $ROR9$	2	$S'[4] = S[4] \oplus S[0]$ $S'[9] = S[9] \oplus S[0]$ $S'[19] = S[19] \oplus S[0]$ $S'[27] = S[27] \oplus S[0]$ $ROR9$
1	$S'[4] = S[4] \oplus S[0]$ $S'[10] = S[10] \oplus S[0]$ $S'[18] = S[18] \oplus S[0]$ $S'[27] = S[27] \oplus S[0]$ $ROR9$	3	$S'[4] = S[4] \oplus S[0]$ $S'[9] = S[9] \oplus S[0]$ $S'[19] = S[19] \oplus S[0]$ $S'[28] = S[28] \oplus S[0]$ $ROR8$

$$S'[0] = S[0] \oplus S[4]$$

$$S'[1] = S[1] \oplus S[5]$$

$$S'[2] = S[2] \oplus S[6]$$

$$S'[18] = S[18] \oplus S[4]$$

$$S'[19] = S[19] \oplus S[5]$$

$$S'[20] = S[20] \oplus S[6]$$

The ADDFn and RORFn operations are adjusted to the Fugue-512 and described in Table 3.10.

Finally, Finalization is performed as follows:

$$Finalization = \begin{array}{c} S[1..3] \,||\, (S[4] \oplus S[0]) \,||\, (S[9] \oplus S[0]) \,||\, S[10..12] \,||\, \\ (S[18] \oplus S[0]) \,||\, S[19..21] \,||\, (S[27] \oplus S[0]) \,||\, S[10..12] \end{array}$$

3.8 Groestl

3.8.1 Block Diagram Description

Groestl is an example of another SHA-3 candidate based on AES. A block diagram in Figure 3.19 shows datapath used in our design. As opposed to a straightforward design, a pipelined architecture is applied. The pipeline register is inserted between SubBytes and ShiftBytes operations. A message block is xored with an initialized chain register to create an input for the operation P in the first cycle of processing. In the next cycle, an input message is loaded directly to the state register as an input to the operation Q. At the same time when the first stage of the pipeline starts executing the operation Q, the second stage of the pipeline continues the execution of the operation P. The first stage of the pipeline consists of the ADD_SUB unit. The second stage of the pipeline consists of the ShiftBytes and MixBytes units. A part of the function P is always performed one cycle ahead of the corresponding part of function Q. Finalization in this design takes two clock cycles. First, the chaining value is xored with the final value of P, while Q is being still processed. In the subsequent cycle the final result of Q is mixed with the chaining value as well. The entire process is repeated until all blocks of a message are thoroughly mixed. Finally, a hash value is taken from the bottom half of the chaining value.

Figure 3.20, describes how the AddConstant and SubBytes are performed in our design. A round number is xored with the first byte of a message in the P operation. In the Q operation, a complemented round number is xored into the 8th byte. After that, all bytes go through the SBOX of AES.

ShiftBytes operation is performed by rotating all bytes in row i to the right by σ_i, where σ is given as $\sigma = [0, 1, 2, 3, 4, 5, 6, 7]$. Figure 3.21 describes MixBytes operation. The MixBytes operation splits an input into b/64 64-bit words. Each word becomes an input into Groestl matrix multiplication. The constant matrix multiplication table used in Groestl is given in Table 3.11. All operations are performed in $GF(2^8)$, the same as in AES, as shown in Table 3.2.

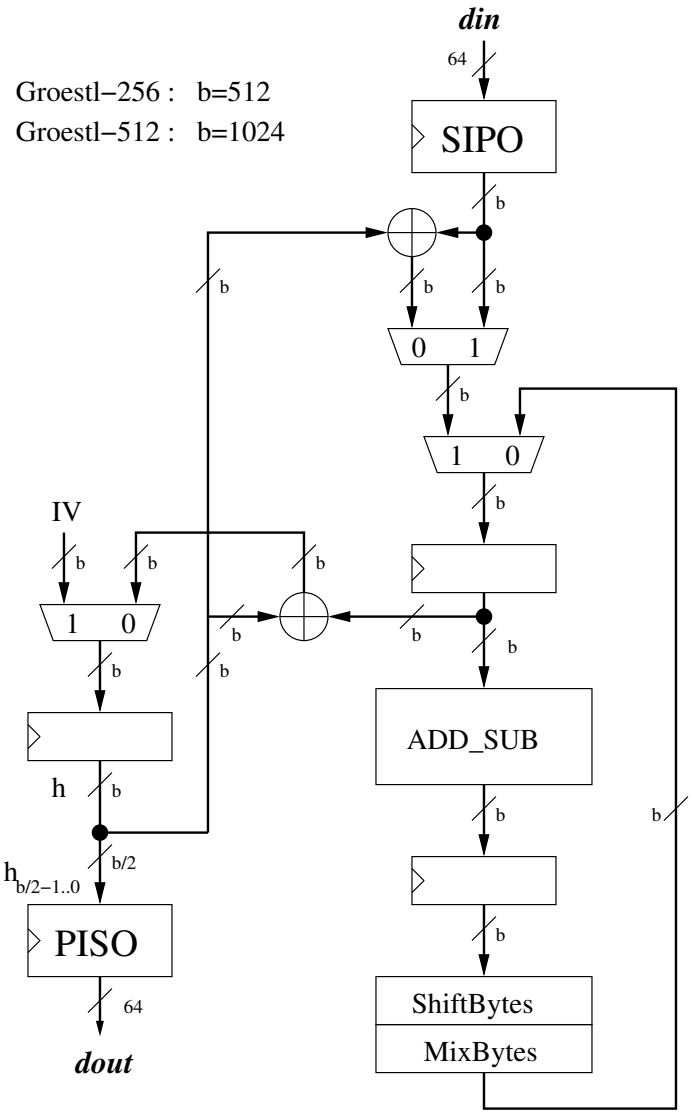


Figure 3.19: Groestl : Datapath

Table 3.11: Groestl: Matrix Multiplier Table

	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
B'[0]	2	2	3	4	5	3	5	7
B'[1]	7	2	2	3	4	5	3	5
B'[2]	5	7	2	2	3	4	5	3
B'[3]	3	5	7	2	2	3	4	5
B'[4]	5	3	5	7	2	2	3	4
B'[5]	4	5	3	5	7	2	2	3
B'[6]	3	4	5	3	5	7	2	2
B'[7]	2	3	4	5	3	5	7	2

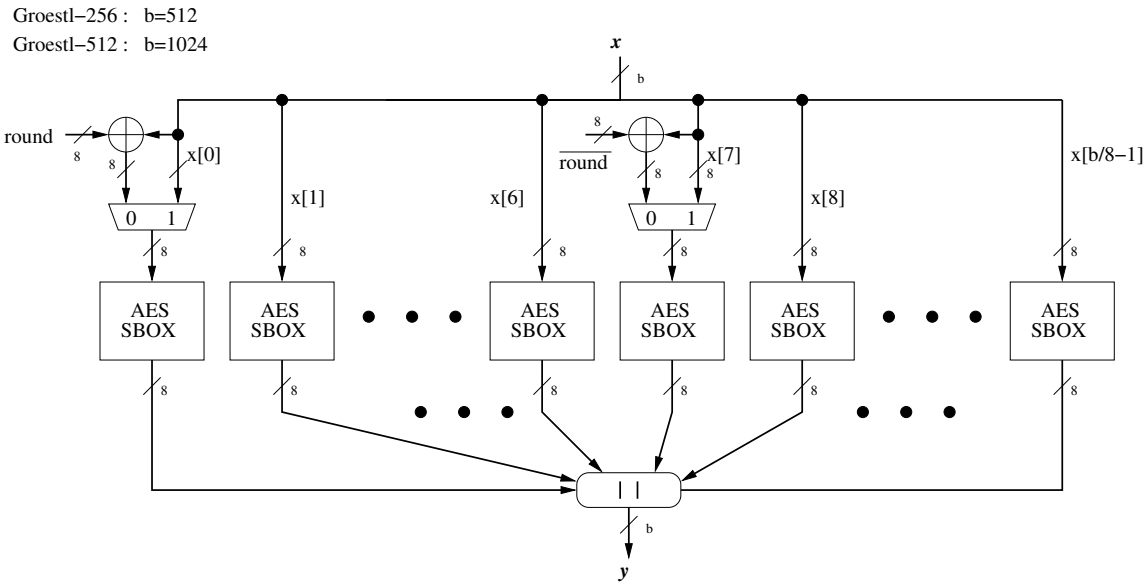


Figure 3.20: Groestl : AddConstant and SubBytes

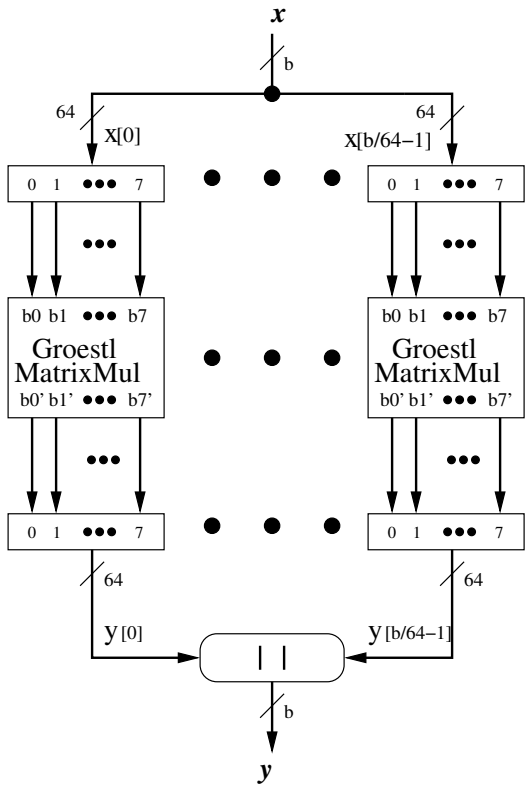


Figure 3.21: Groestl : MixBytes

3.8.2 256 vs 512 Variant Differences

In Groestl-512 the block size is doubled. This means that the state size is increased by a factor of two as well. All basic operations of Groestl remain the same with the exception of ShiftRows. The ShiftRows rotation constants for each row are now changed to $\sigma = [0, 1, 2, 3, 4, 5, 6, 11]$. Finally, the number of rounds for Groestl-512 is increased to 14.

3.9 Hamsi

3.9.1 Block Diagram Description

Hamsi's datapath is shown in Figure 3.22. For every message block, an expanded message is concatenated with the chaining value to form a state. This state is viewed as an array of 32-bit words. The state is transformed through P or P_f rounds, using ACC, Substitution Layer and Diffusion Layer in each round. For Hamsi-256, P and P_f are equal to 3 and 8, respectively. P_f is selected as a number of rounds during processing of the last block of a message. After completing all rounds, the state is truncated and xored with the previous chaining value to form a new one.

In Figure 3.23, Message Expansion is shown. Message Expansion expands an input word of the size of w bits to an output of the size of half of the block size $b/2$. Each word is split into an array of bytes. Each byte becomes an input to a ROM-based look-up table, which produces a 32-bit output. The outputs from w/8 neighboring look-up tables are xored together to produce a portion of the overall output of the Message Expansion. All ROMs contain different dataset values, which can be obtained from a reference software implementation included in the submission package of [29].

Concatenation is performed as follows:

$$y = m[0..1]||c[0..1]||c[2..3]||m[2..3]||m[4..5]||c[4..7]||m[6..7]$$

ACC refers to Addition of Constants and Counter step. This step can be described by the following sequence of operations:

$$s' = s \oplus \alpha$$
$$s'[2] = s' \oplus c$$

Substitution Layer is shown in Figure 3.24. An input is split into four equal blocks. Then the corresponding bits of each block form an input to the Hamsi SBOX. This SBOX is defined in Table 3.12.

Hamsi−256 : b=512,w=32
Hamsi−512 : b=1024,w=64

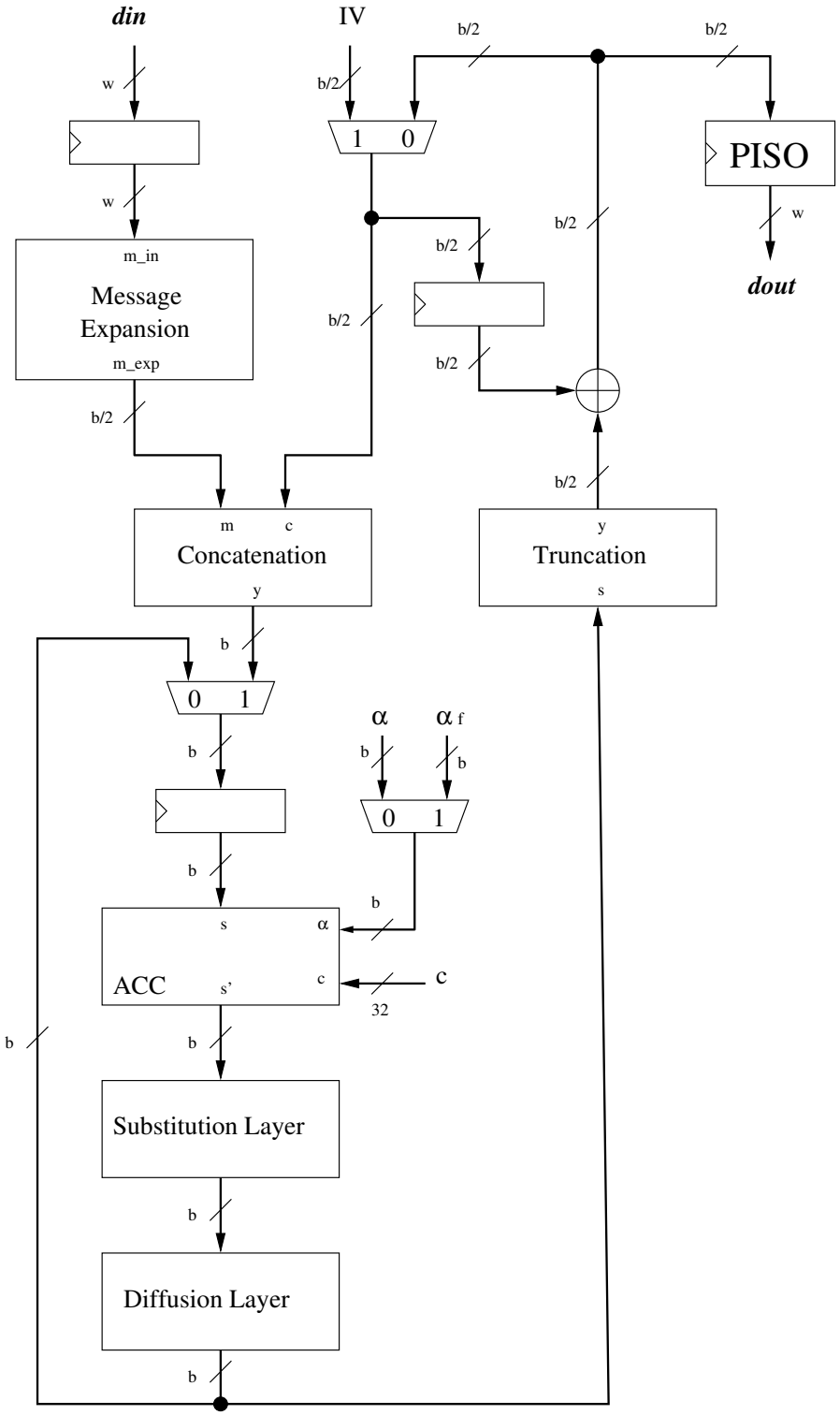


Figure 3.22: Hamsi : Datapath

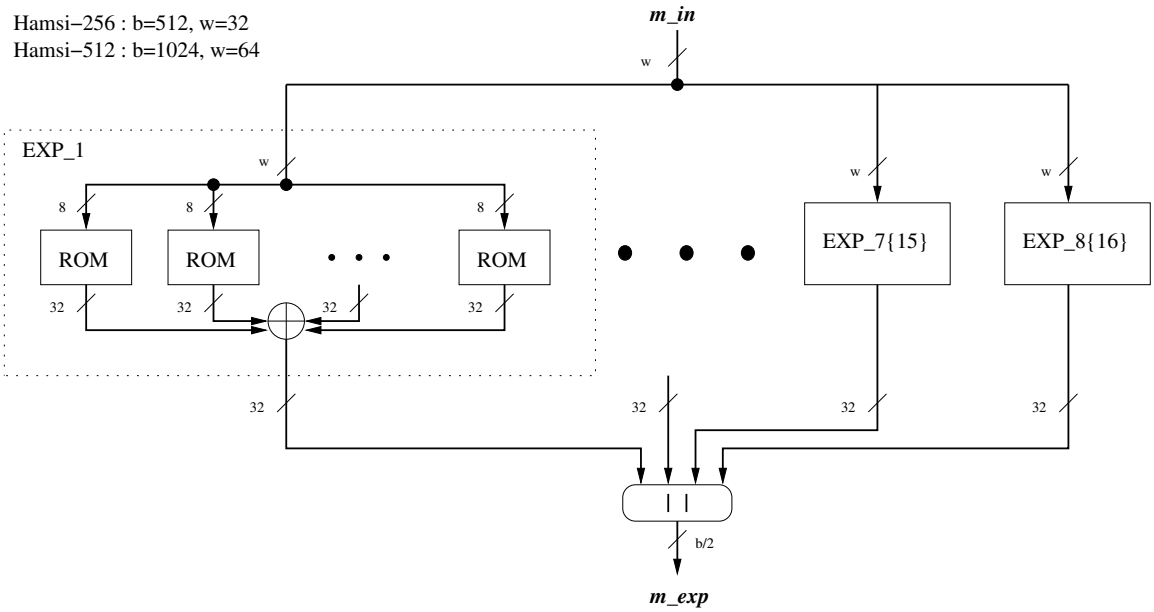


Figure 3.23: Hamsi : Message Expansion

Diffusion Layer is based on the logic function L, shown in 3.25. This function performs the following sequence of operations:

$$(s[0], s[5], s[10], s[15]) = L(s[0], s[5], s[10], s[15])$$
$$(s[1], s[6], s[11], s[12]) = L(s[1], s[6], s[11], s[12])$$
$$(s[2], s[7], s[8], s[13]) = L(s[2], s[7], s[8], s[13])$$
$$(s[3], s[4], s[9], s[14]) = L(s[3], s[4], s[9], s[14])$$

Finally, Truncation is performed as follows:

$$y = s[0..3] \;||\; s[8..11]$$

Table 3.12: Hamsi: SBOX

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
s[X]	8	6	7	9	3	C	A	F	D	1	E	4	0	B	5	2

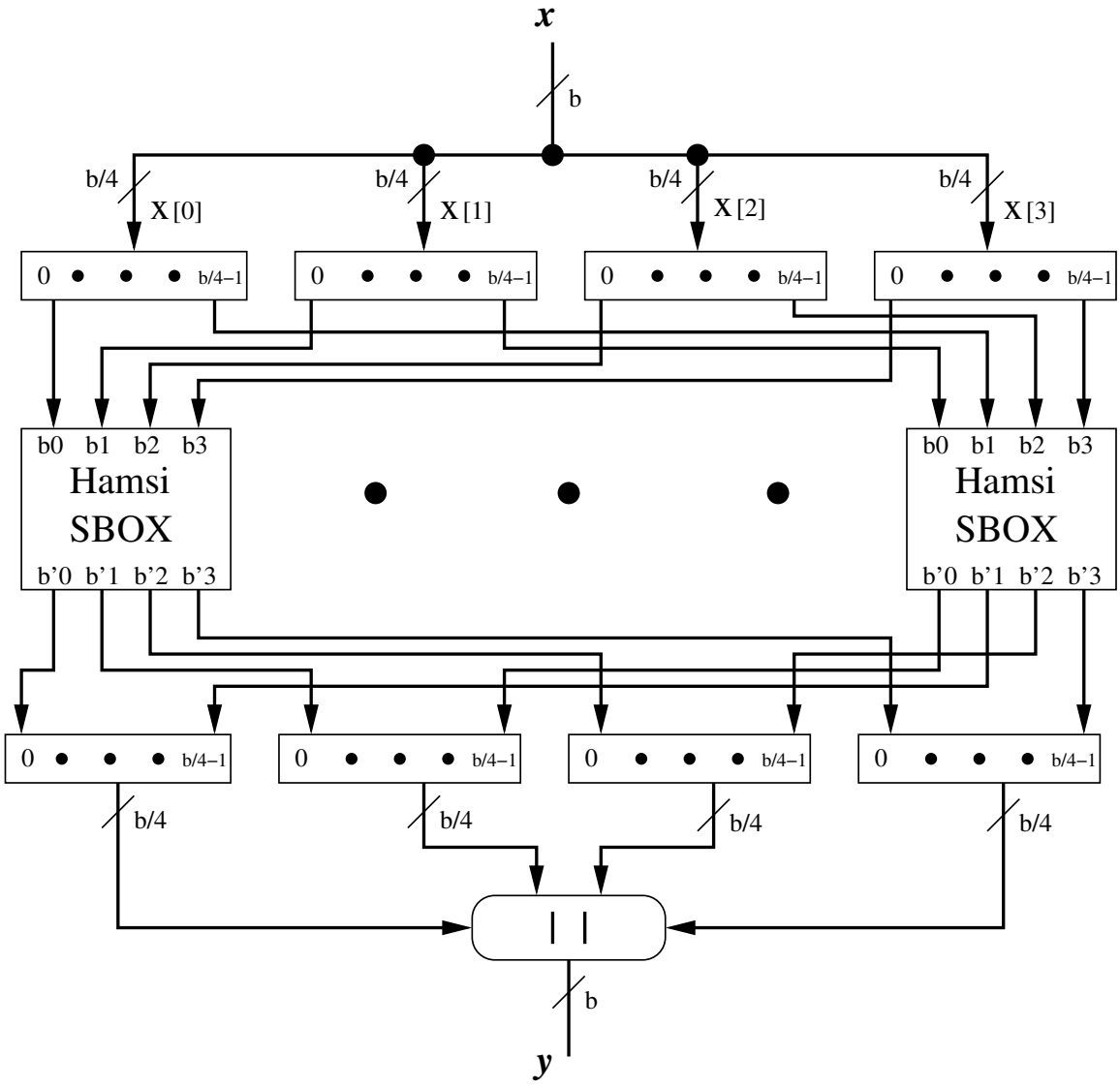


Figure 3.24: Hamsi : Substitution Layer

Note : All bus sizes are 32 bits

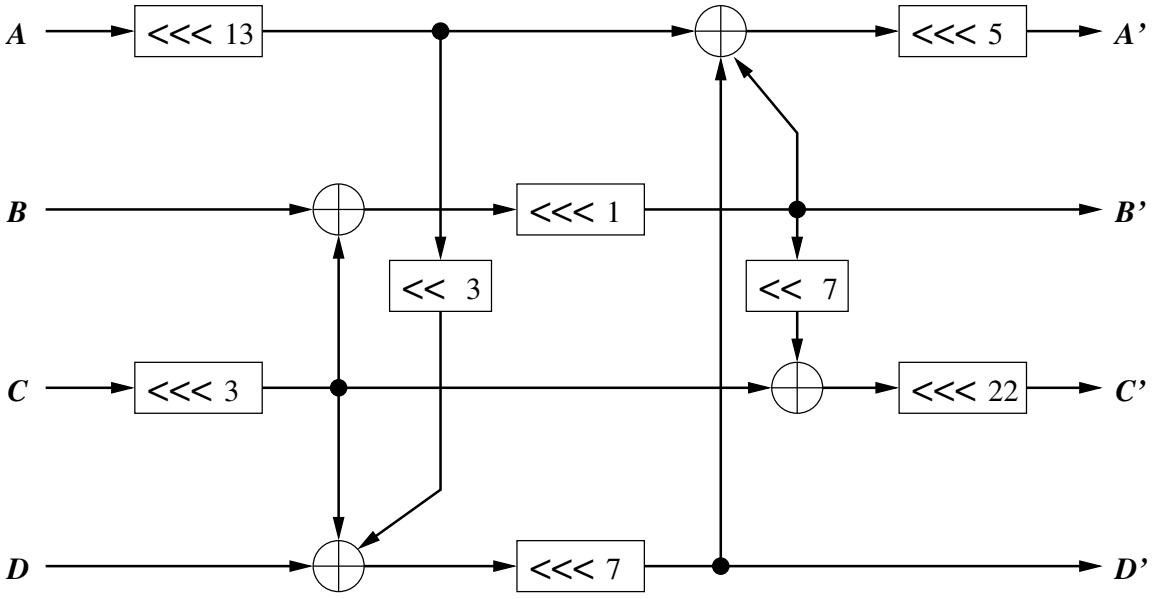


Figure 3.25: Hamsi : L

3.9.2 256 vs 512 Variant Differences

An input to Hamsi-512 is increased to 64 bits. As a result, the size of ROMs used in the Message Expansion unit is increased as well. Similar to Hamsi-256, the data to populate these ROM-based look-up tables can be found in the reference software implementation. The rest of the operations remain largely the same with the following exceptions: Concatenation, Diffusion Layer, and Truncation.

Concatenation of Hamsi-512 is performed as follows:

$$y = \begin{matrix} m[0..1]||c[0..3]||m[2..5]||c[4..7], m[6..9]||c[8..9]|| \\ m[10..11]||c[10..13]||m[12..13]||c[14..5]||m[14..15] \end{matrix}$$

Diffusion Layer of Hamsi-512 is defined using the following sequence of operations:

$$(s[0], s[9], s[18], s[27]) = L(s[0], s[9], s[18], s[27])$$
$$(s[1], s[10], s[19], s[28]) = L(s[1], s[10], s[19], s[28])$$
$$(s[2], s[11], s[20], s[29]) = L(s[2], s[11], s[20], s[29])$$
$$(s[3], s[12], s[21], s[30]) = L(s[3], s[12], s[21], s[30])$$
$$(s[4], s[13], s[22], s[31]) = L(s[4], s[13], s[22], s[31])$$
$$(s[5], s[14], s[23], s[24]) = L(s[5], s[14], s[23], s[24])$$
$$(s[6], s[15], s[16], s[25]) = L(s[6], s[15], s[16], s[25])$$
$$(s[7], s[8], s[17], s[26]) = L(s[7], s[8], s[17], s[26])$$
$$(s[0], s[2], s[5], s[7]) = L(s[0], s[2], s[5], s[7])$$
$$(s[16], s[19], s[21], s[22]) = L(s[16], s[19], s[21], s[22])$$
$$(s[9], s[11], s[12], s[14]) = L(s[9], s[11], s[12], s[14])$$
$$(s[25], s[26], s[28], s[31]) = L(s[25], s[26], s[28], s[31])$$

Truncation of Hamsi-512 is performed as follows:

$$y = s[0..7] \;||\; s[16..23]$$

3.10 JH

3.10.1 Block Diagram Description

The block diagram of JH is shown in Figure 3.26. To process a message, the state register is initialized to IV, the temporary register takes the value of an input message block and the key register is initialized to C_IV. The state is transformed using R8 for 36 rounds. Each of these rounds use different keys generated by the key generator, R6. Once processing is completed, the output from R8 is degrouped and xored with an input message stored in the temporary register to create a new chaining value. If there are more message blocks, the chaining value is xored with an input message and grouped together. The aforementioned steps are repeated until all message blocks are processed. The hash value is taken from the new chaining value of the last block processed.

The operations Group and Degroup are permutations specific to JH. Group and Degroup can be described by the following sequence of operations. Note that k is the keysize and b is equal to the input block size.

```
Group:
for i = 0:k/2-1
  y(b-i*8-1..b-i*8-4) = x(b-1 - i) || x(b-1 - (i+k)) || x(b-1 - (i+2*k)) || x(b-1 - (i+3*k));
  y(b-i*8-5..b-i*8-8) = x(b-1 - (i + k/2)) || x(b-1 - ((i+k) + (k/2))) ||
                x(b-1 - (i+2*k + k/2)) || x(b-1 - (i+3*k + k/2));
end
```

```
Degroup:
for i in 0 to k/2-1 loop
  dg(b-1 - i)          := rd(b-i*8-1);
  dg(b-1 - (i+k))        := rd(b-i*8-2);
  dg(b-1 - (i+2*k))    := rd(b-i*8-3);
  dg(b-1 - (i+3*k))      := rd(b-i*8-4);
  dg(b-1 - (i + k/2))      := rd(b-i*8-5);
  dg(b-1 - (i+k + k/2))    := rd(b-i*8-6);
  dg(b-1 - (i+2*k + k/2))    := rd(b-i*8-7);
  dg(b-1 - (i+3*k + k/2))    := rd(b-i*8-8);
end loop;
```

In Figure 3.27, a generic description of a JH round is shown. The same unit is used for R6 and R8. The differences are the key and input sizes. Where R6 uses values 64 and 256 for the key and the input sizes respectively, R8 uses values 256 and 1024. In a JH Round, an input is viewed as an array of 4-bit blocks. These blocks go through either S_0 or S_1 s-boxes, defined in Table 3.13.

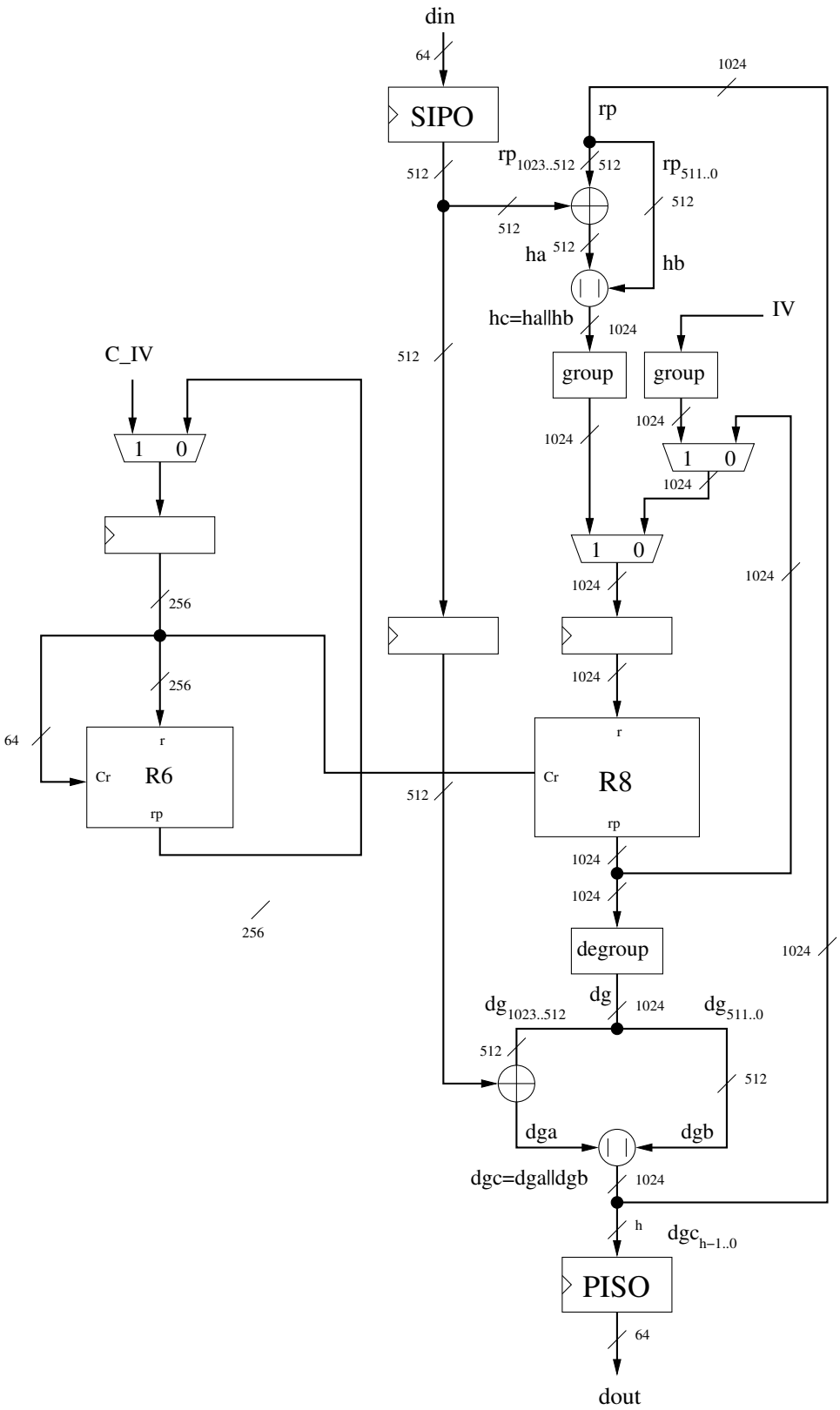


Figure 3.26: JH:Datapath

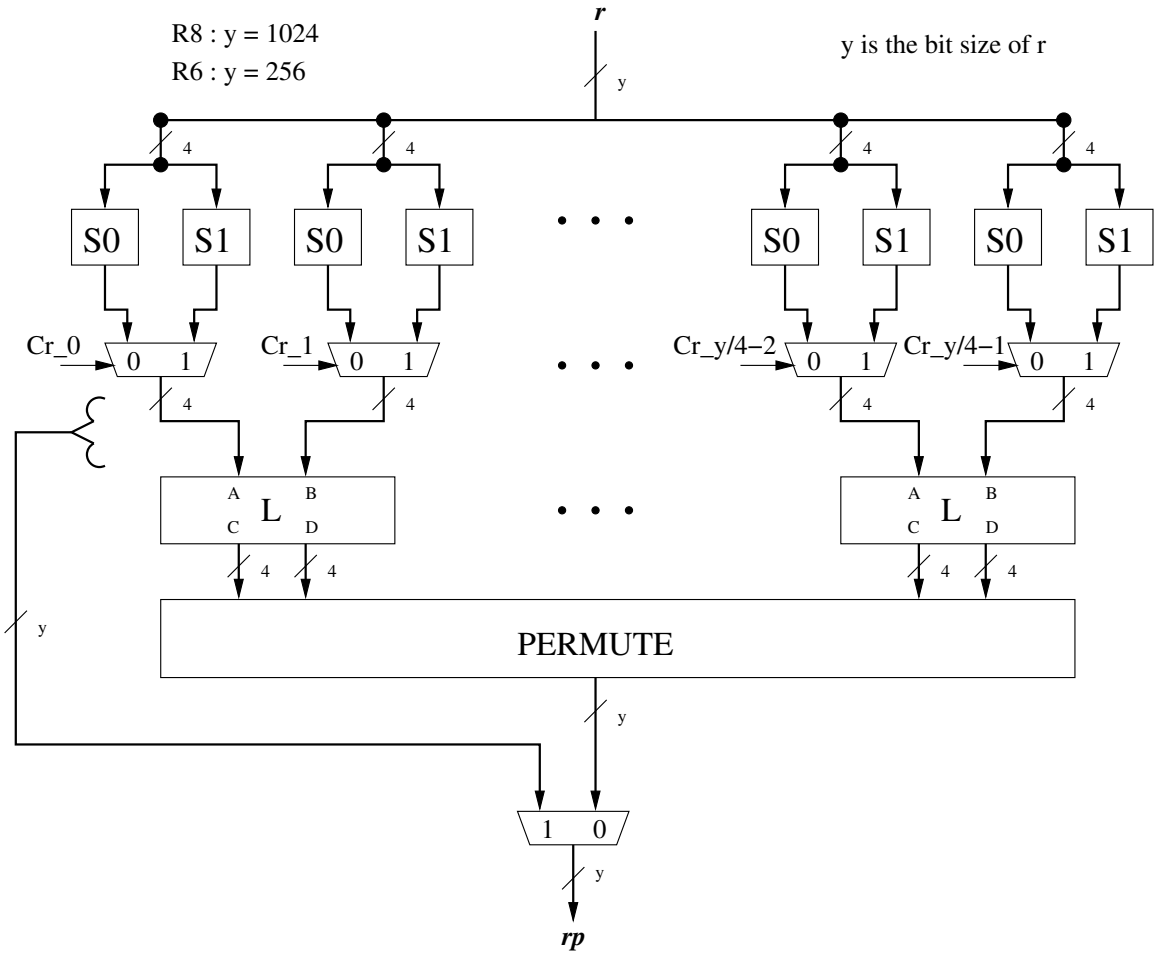


Figure 3.27: JH : Round

Next, outputs from these sboxes are selected by a corresponding input key. Two consecutive outputs form an input to the linear transformation unit, L. A diagram of this unit is shown in Figure 3.28. The transformed outputs are then permuted by the PERMUTE block. PERMUTE can be described by a series of permutations given by the code below. x, y and k refers to input, output and the size of the key, respectively.

```
for i = k/4-1:0
  a(i*4 + 0) <= x(i*4 + 0);
  a(i*4 + 1) <= x(i*4 + 1);
  a(i*4 + 2) <= x(i*4 + 3);
  a(i*4 + 3) <= x(i*4 + 2);
end generate;
for i = k/2-1:0
  b(i)       <= a(i*2);
  b(i + k/2) <= a(i*2 + 1);
end
for i = k/2-1:0
  y(i) <= b(i);
end generate;
for i = k/4-1:0
  y(i*2 + k/2)     <= b(i*2 + 1 + k/2);
  y(i*2 + 1 + k/2) <= b(i*2 + k/2);
end generate;
```

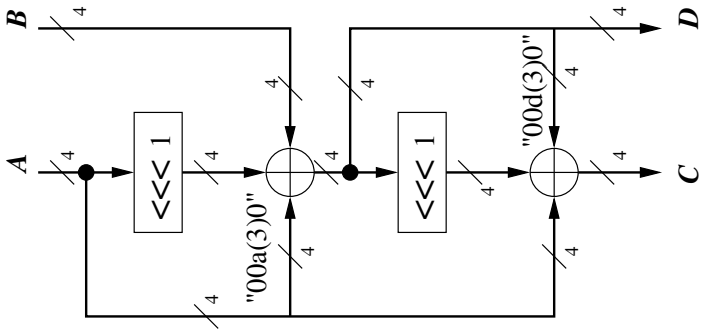


Figure 3.28: JH : Linear Transformation

3.10.2 256 vs 512 Variant Differences

All operations are the same for both variants. The only exception is the output selection function of JH-512, where 512 bits of the chaining value are selected instead of 256 bits in JH-256.

Table 3.13: JH: SBOX

y	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	5	13	5	7	1	9	15	2	0	4	11	10	14	8

3.11 Keccak

3.11.1 Block Diagram Description

Keccak is based on four basic logic operations: xor, and, not and rotate. Based on the authors' recommendations, Keccak-1600 is chosen as a candidate for SHA-3. For Keccak-256, an input message block has the size of 1088 bits. The full details of our datapath are shown in Figure 3.29. For every message block, an input is zero-extended to produce a 1600-bit state. This state can be viewed as a 5x5 array of 64-bit words as shown in Figure 3.30. An extended input is xored with the chaining value. For the first message block, the chaining value is zero. The state is then transformed using Keccak Round for 24 rounds. Finally, a hash value is selected from the chaining value of the last message block. The description of the Keccak's Round is shown in Figures 3.31, 3.32.

3.11.2 256 vs 512 Variant Differences

All operations are the same for both variants. The only exception is the output selection of Keccak-512, where 512 bits of the chaining value are selected instead of 256 bits in Keccak-256.

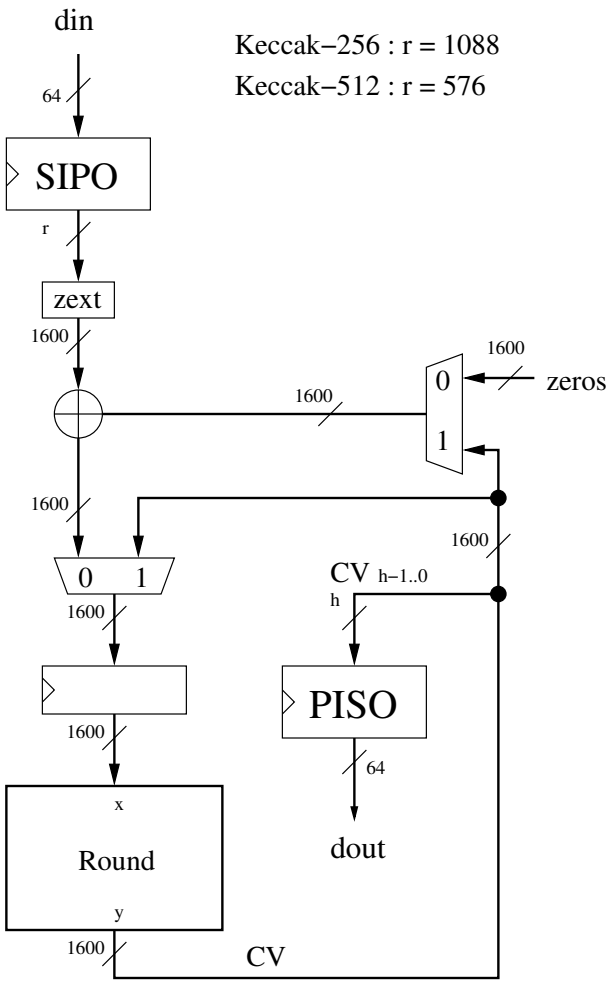


Figure 3.29: Keccak : Datapath

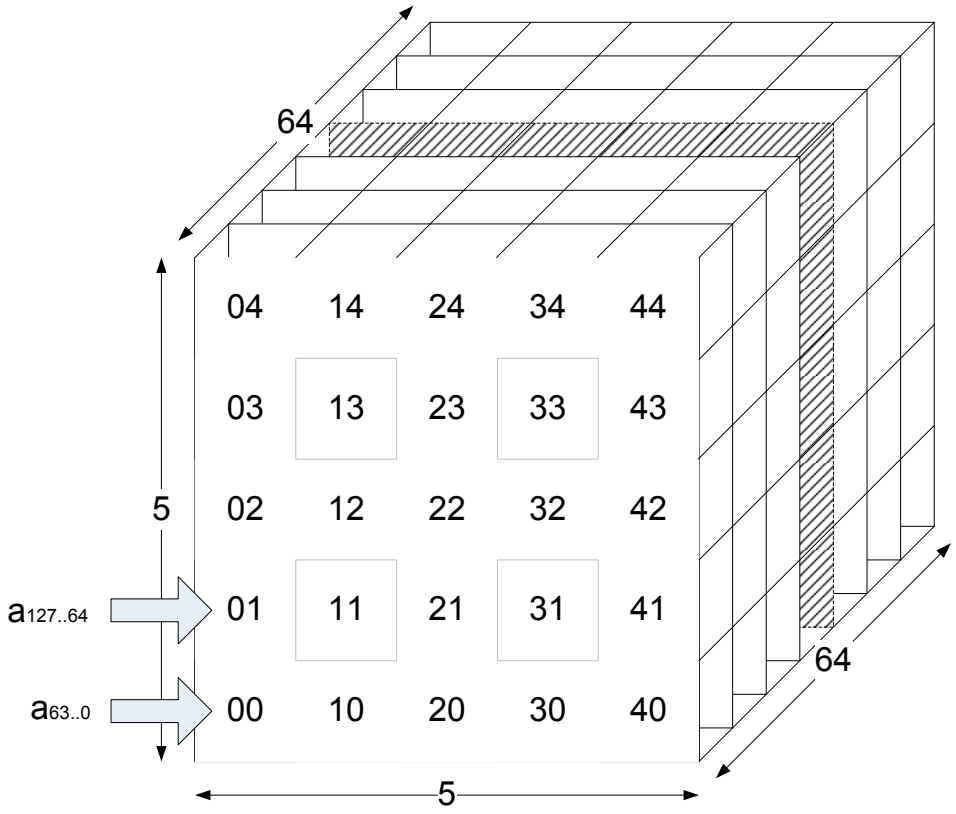


Figure 3.30: Keccak : State

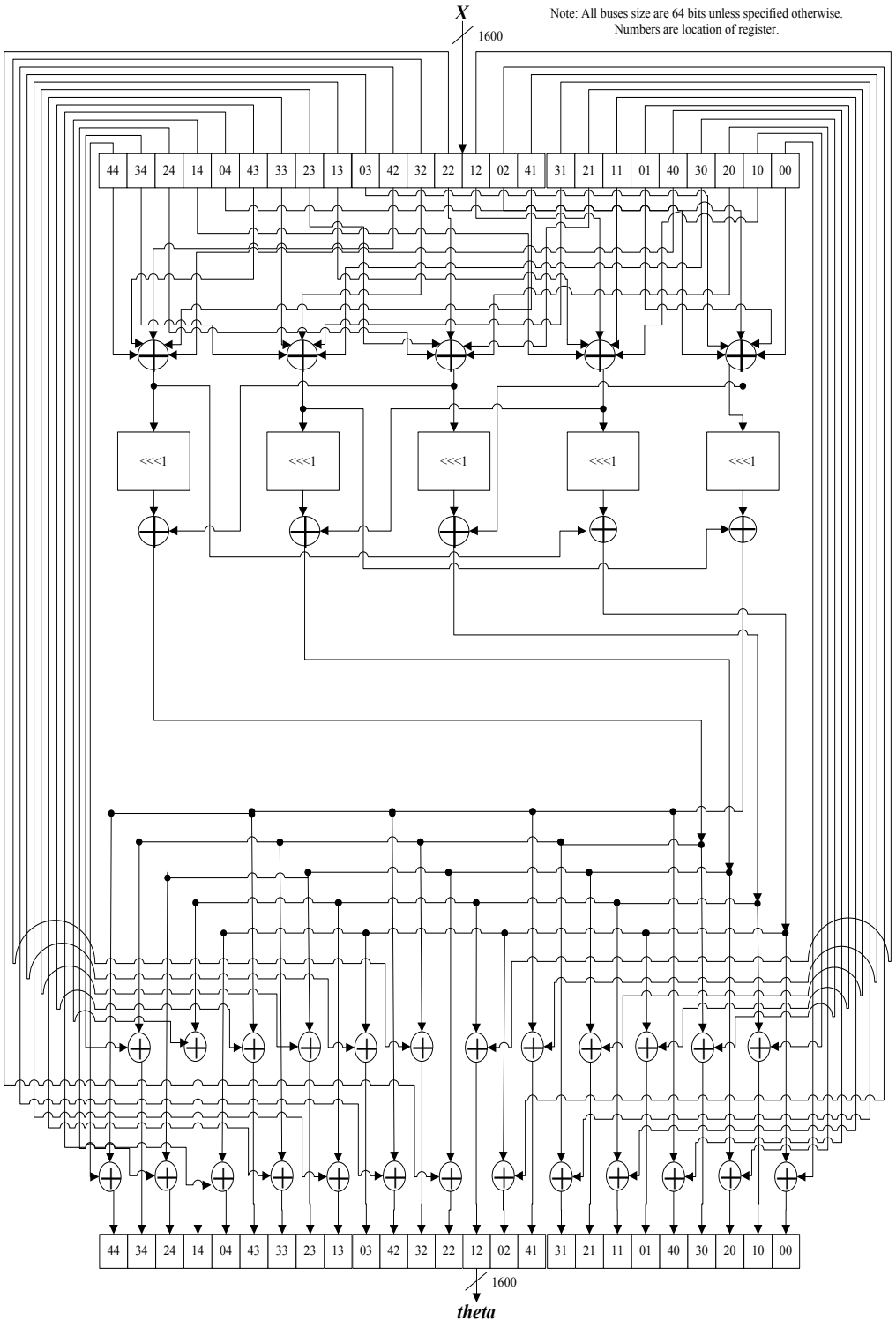


Figure 3.31: Keccak : Round Part 1

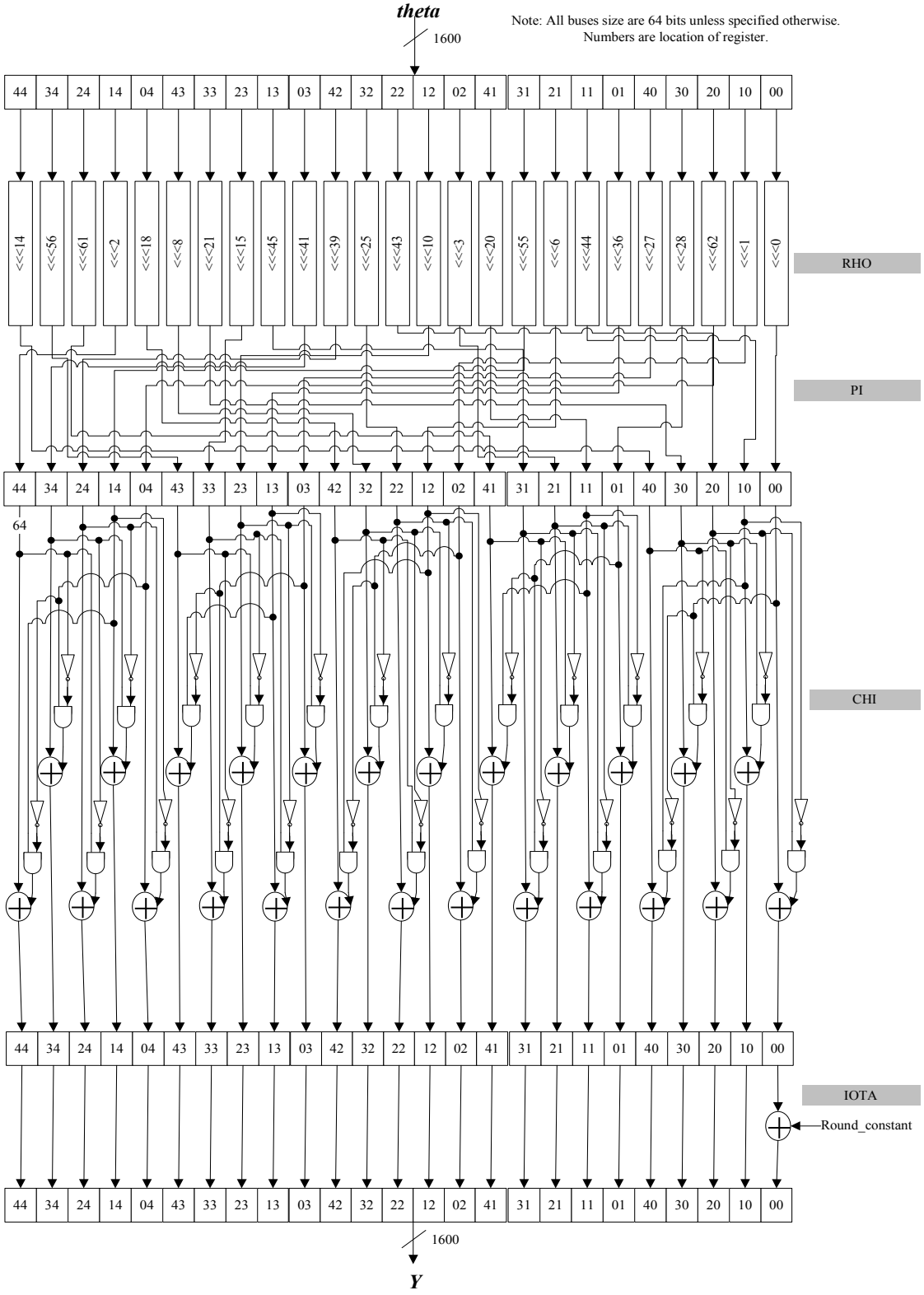


Figure 3.32: Keccak : Round Part 2

3.12 Luffa

3.12.1 Block Diagram Description

The datapath of Luffa is shown in Figure 3.33. For every message block, an input block is injected into the chain value via the Message Injection (MI) unit. The initial chaining value is equal to IV. The MI unit is shown in Figure 3.34. The Galois field multiplication (x2), used in the MI unit, is also shown in Figure 3.1.

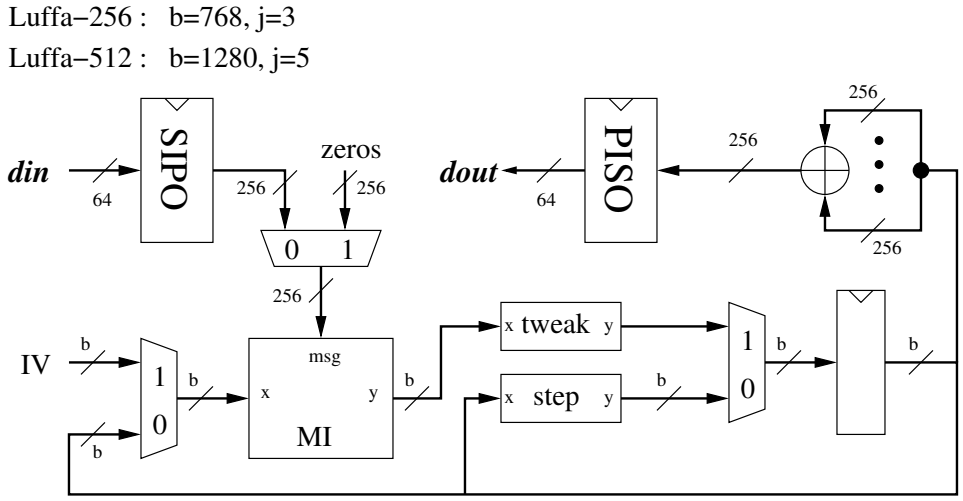


Figure 3.33: Luffa : Datapath

The state is then rotated wordwise using the Tweak operation shown in Figure 3.35. The number of positions by which each word is rotated depends on the position of the word in the input to the Tweak function. Next, the state is transformed through the Step function for 8 rounds. A diagram of the Step function is shown in Figure 3.36. The Step function consists of SubCrumb, MixWord and AddConstant operations. SubCrumb and MixWord are shown in Figures 3.37 and 3.38, respectively. AddConstant is an addition of a constant to the first and the fifth word of the state array. The constant is selected depending on the round number. The values of these constants can be found from Appendix B of [32]. The process repeats itself until all message blocks are fully injected. Once processing is completed, the state's 256-bit blocks are xored together to form the hash value.

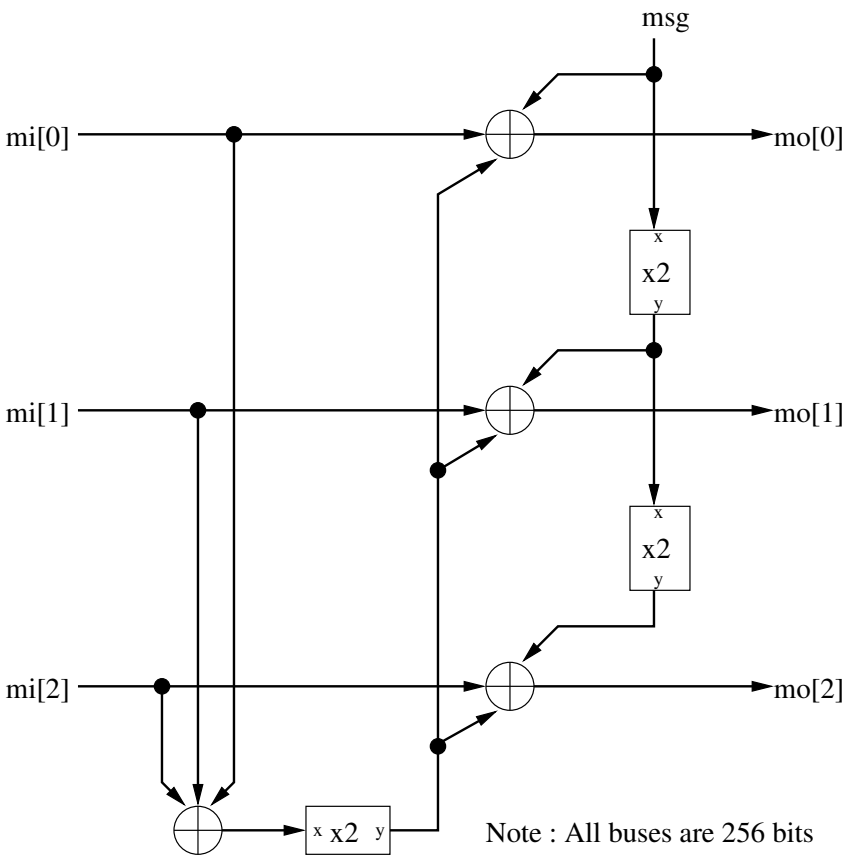


Figure 3.34: Luffa : Message Injection (256 bits)

Luffa−256 : b=768, j=3

Luffa−512 : b =1280, j=5

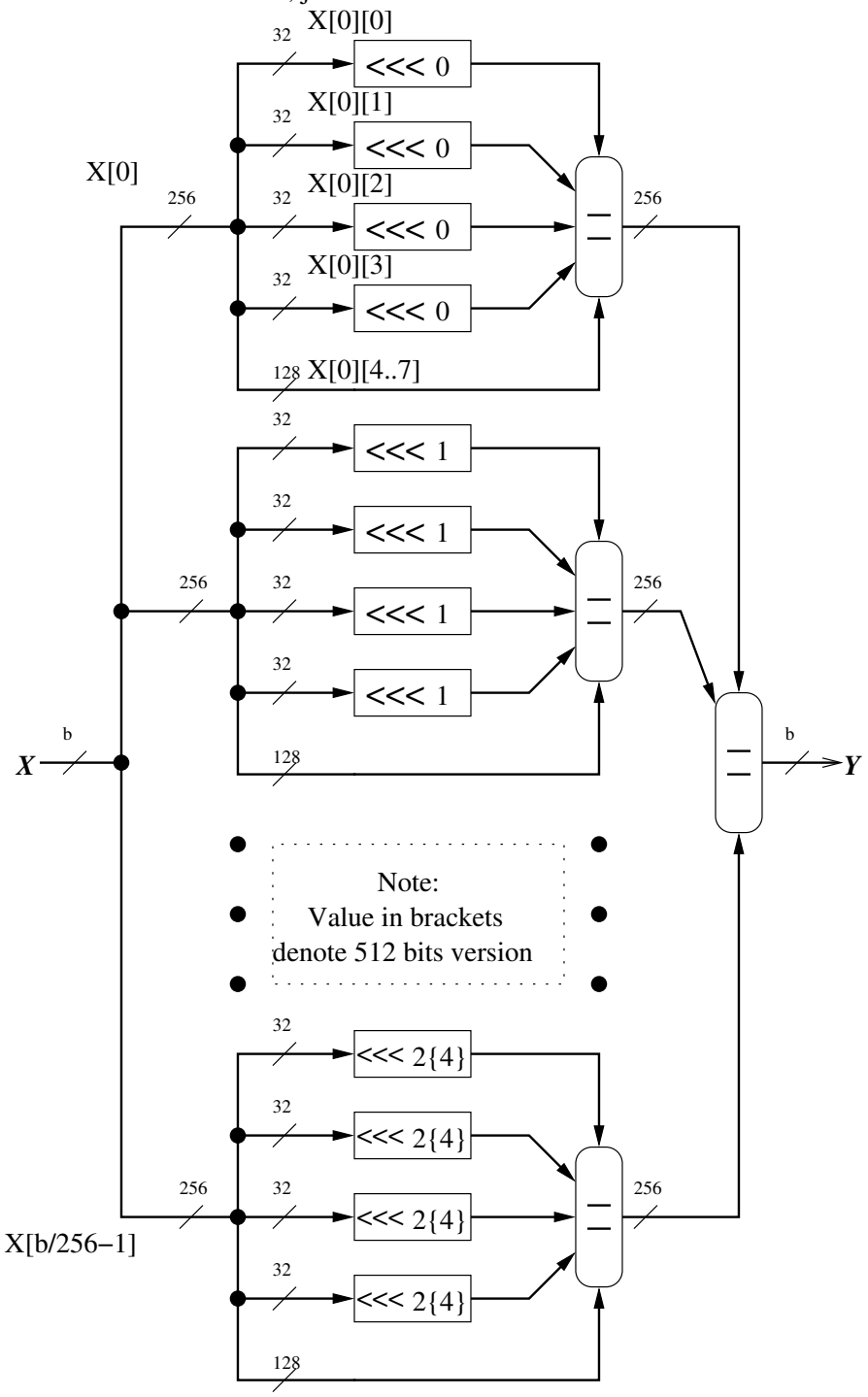


Figure 3.35: Luffa : Tweak

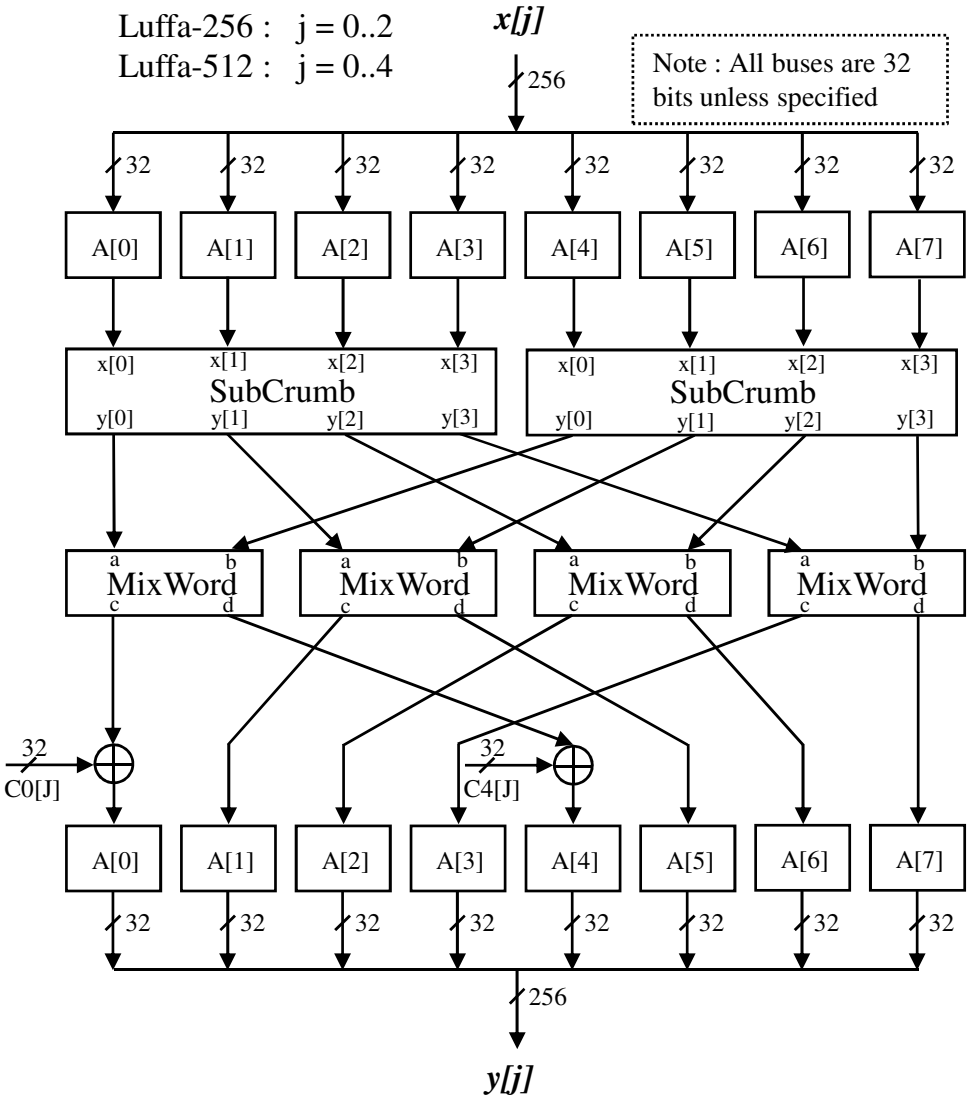


Figure 3.36: Luffa : Step function

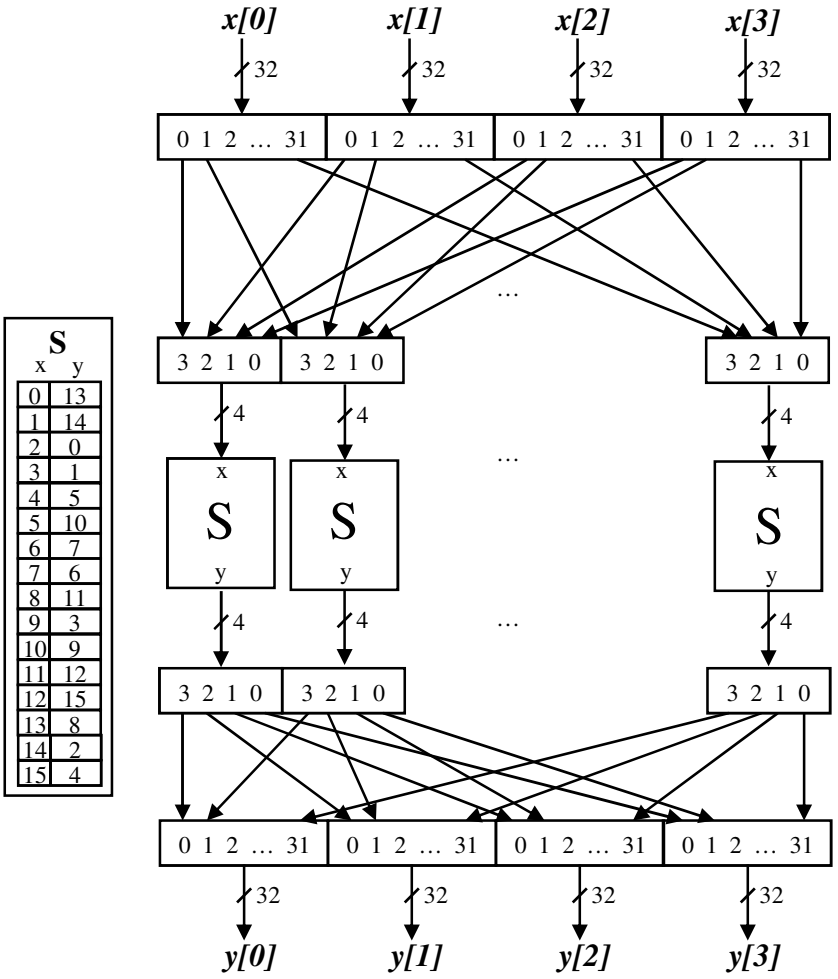


Figure 3.37: Luffa : SubCrumb

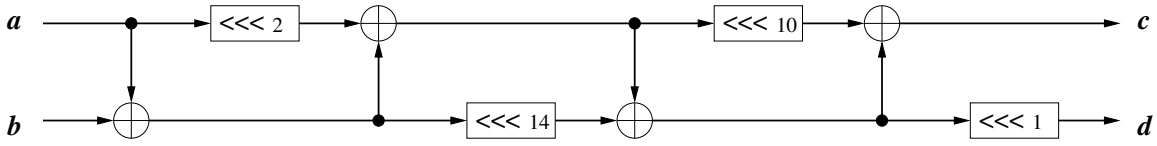


Figure 3.38: Luffa : Mix

3.12.2 256 vs 512 Variant Differences

Luffa-512 increases the state array size from 3 to 5. This means that the state's size is equal to 1280 bits. The Message Injection block is also redefined appropriately, as shown in Figure 3.39. Since the finalization process only produces 256 bits at a time, the chain value is hashed with another message block of value zero to produce the second half of a 512-bit hash value.

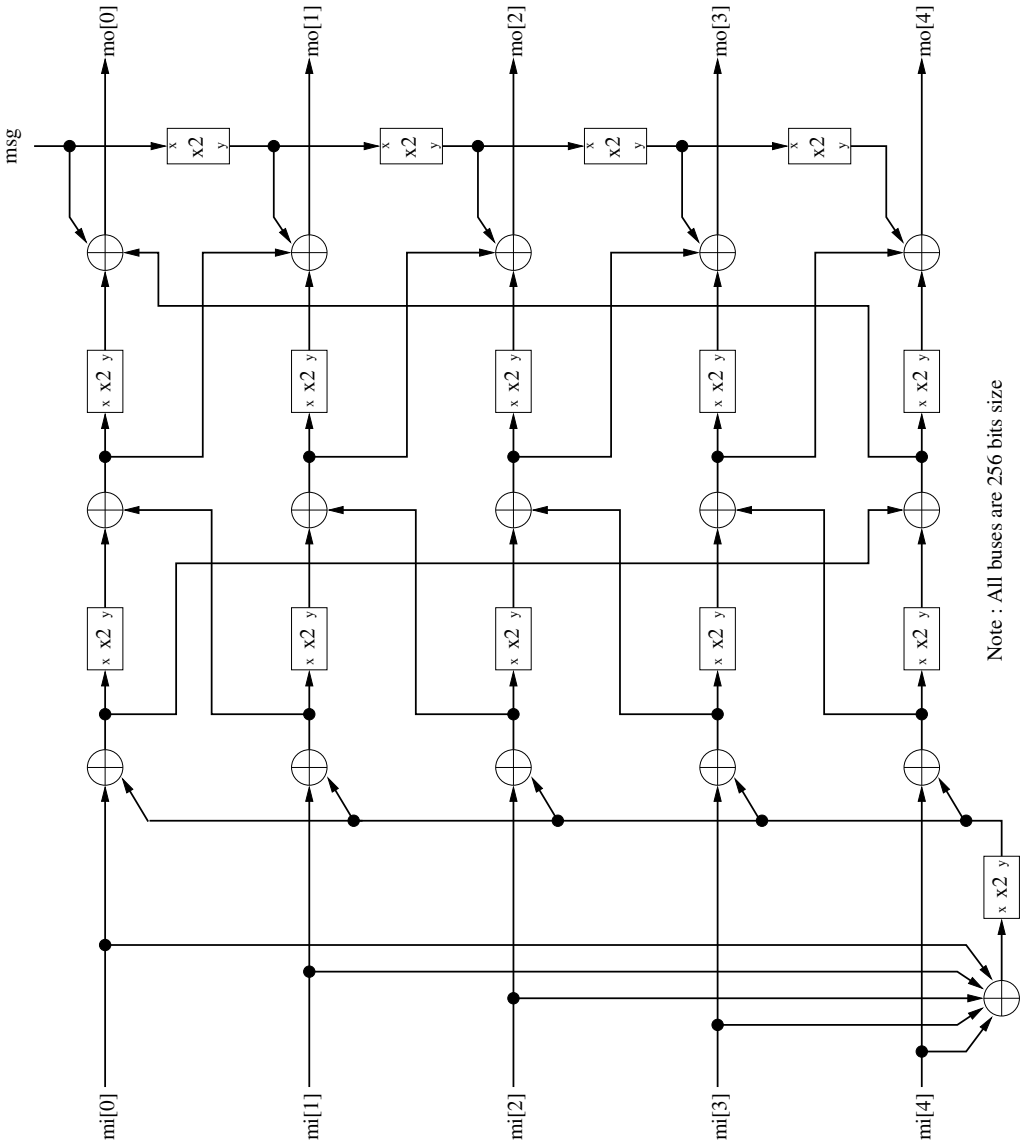


Figure 3.39: Luffa : Message Injection (512 bits)

3.13 SHA-2

3.13.1 Block Diagram Description

Our design of SHA-2 is based on [39]. A diagram of our SHA-2 circuit is shown in Figure 3.41. The detailed definitions of all SHA-2 operations shown in our diagram can be found in [33].

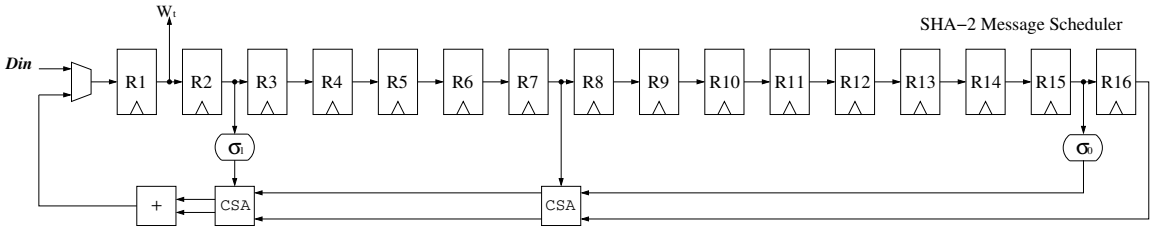


Figure 3.40: SHA2 : Message Scheduler

3.13.2 256 vs 512 Variant Differences

The differences between the two variants include: change in the word size from 32 bits to 64 bits, word selection in the Message Scheduling unit, different operations Σ_0 and Σ_1, and different constants K_t.

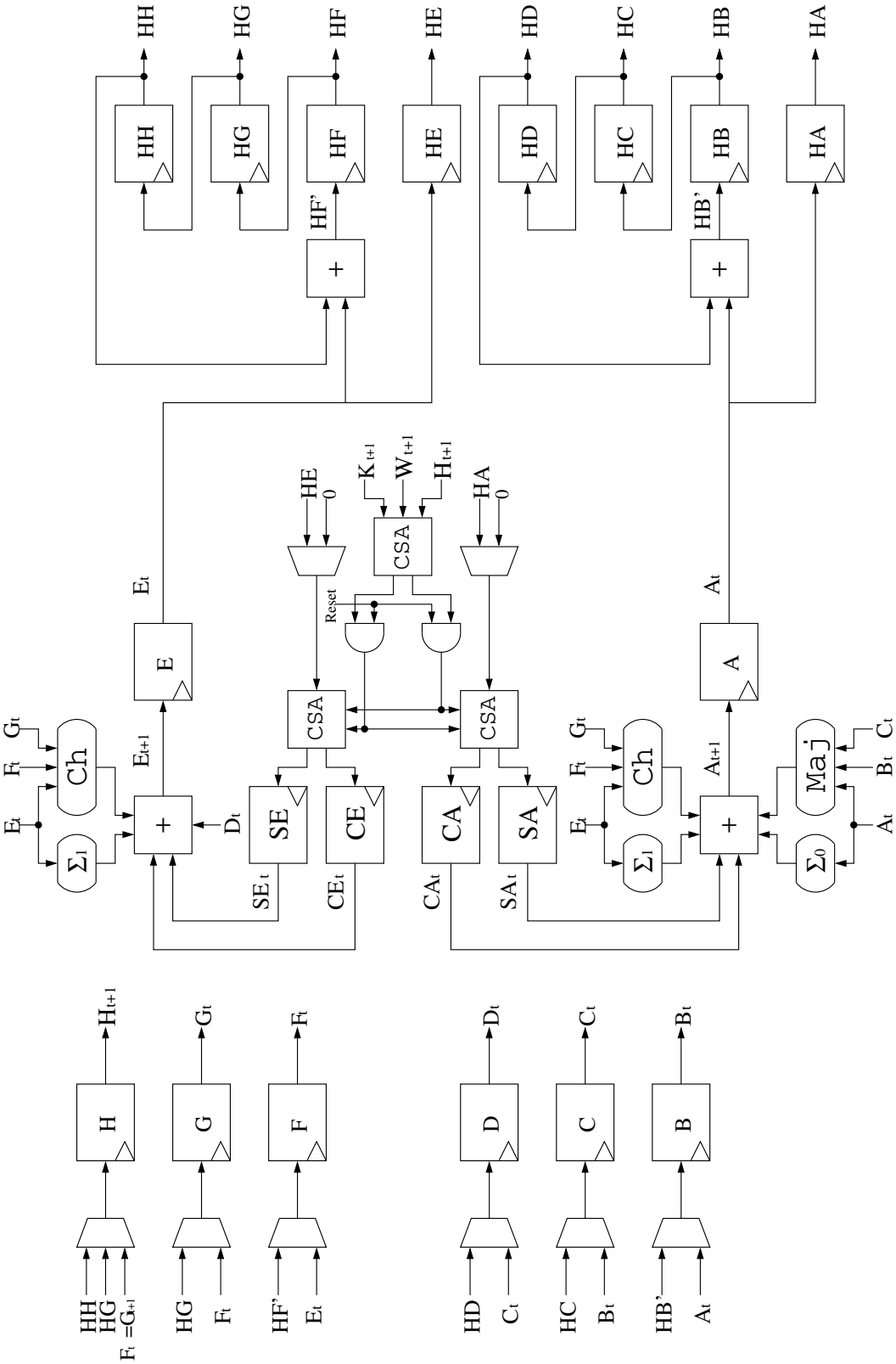


Figure 3.41: SHA2 : Datapath

3.14 Shabal

3.14.1 Block Diagram Description

The block diagram of Shabal is shown in Figure 3.42. W in the diagram represents a counter. This counter counts the number of message blocks that has been hashed so far. This value also includes the current message block. At a minimum, W has a value of one. Shabal contains four state registers, A, B, C and M, each containing an array of 32-bit words. The Shabal architecture used in this thesis is a twice unrolled architecture. An input message block has its endianess switched before the start of processing. Hash value is also required to switch its endianness before the data can be transmitted out.

In the first clock cycle, the following operations are performed:

$$\begin{aligned} A \leftarrow &\quad A_{iv}[0..1] \oplus w[0..1] \;||\; A_{iv}[2..11] \\ B \leftarrow &\quad ((B_{iv} + M_w) <<< 7) \\ C \leftarrow &\quad C_{iv} \\ M \leftarrow &\quad M_w \end{aligned}$$

In the next 24 clock cycles, two rounds of the main iteration unit are executed. This means that all state registers get their data shifted. The last clock cycle prepares the state for the next message block, if any. The performed operation, somewhat similar to the operation executed in the first clock cycle, is shown below:

$$\begin{aligned} A \leftarrow &\quad ap[0..1] \oplus w[0..1] \;||\; ap[2..11] \\ B \leftarrow &\quad (((C - M) + M_w) <<< 7) \\ C \leftarrow &\quad B \\ M \leftarrow &\quad M_w \end{aligned}$$

Finally, if the message block is the last one, an output is produced out of the truncated and endian-switched state B.

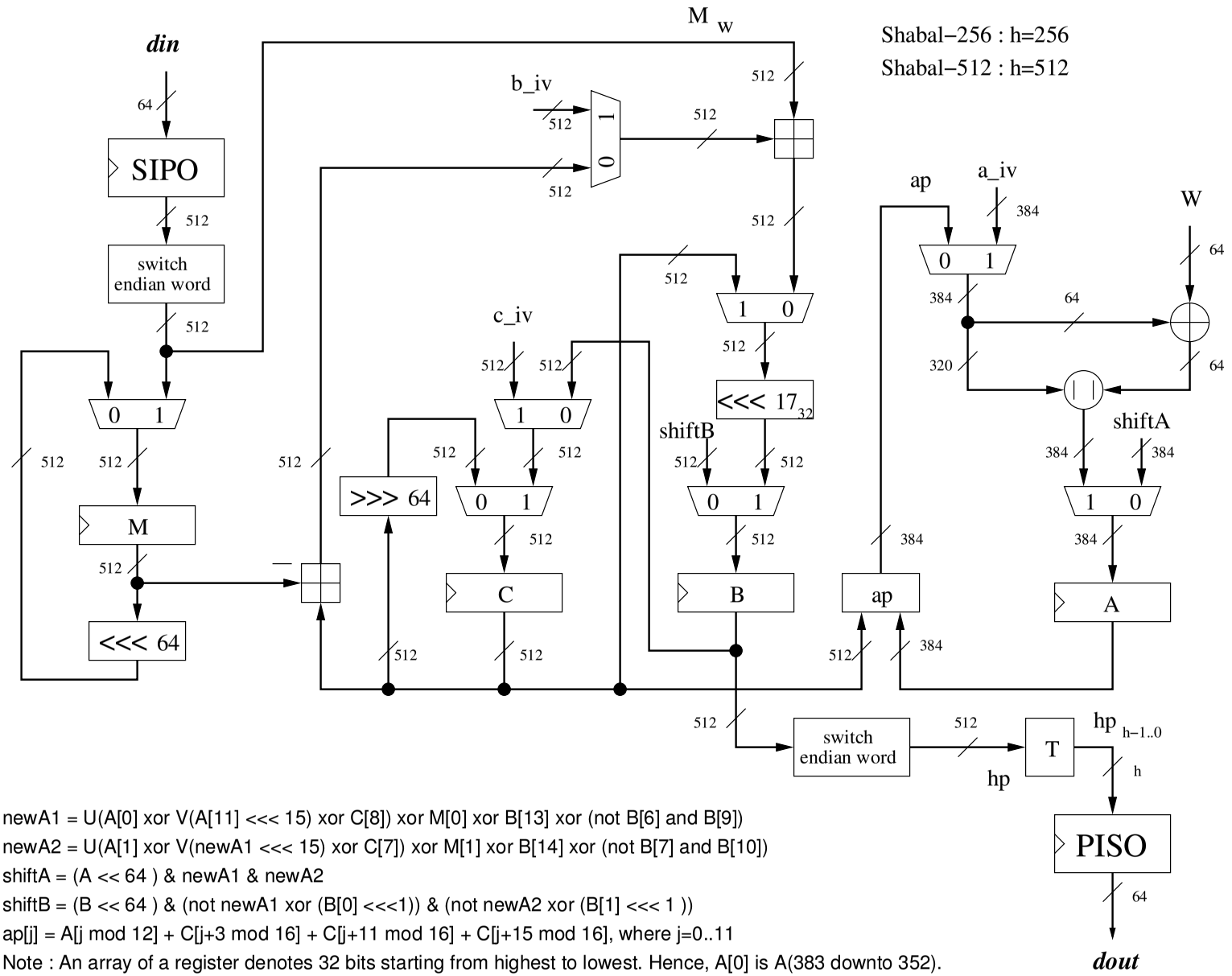


Figure 3.42: Shabal : Datapath

3.14.2 256 vs 512 Variant Differences

There is no difference between the two variants except that no truncation is needed for Shabal-512.

3.15 SHAvite-3

3.15.1 Block Diagram Description

SHAvite-3 works like a block cipher. Three round keys are generated for every round, based on an input message block. The datapath of SHAvite-3-256 is shown in Figure 3.43. For SHAvite-3-256, a block of message contains 512 bits, and 128 bits are loaded into the key generation unit at a time. For every message, the state register and the chain value are initialized to IV. The state register is then processed for 12 rounds. When the processing is completed, the obtained output is xored with the current chain value to generate a new chain value. If it is the last block of the message, the bottom half of the chain value is used as a hash value.

The SHAvite-3 ROUND unit is shown in 3.44. Each main round, executed by the ROUND unit, consists of 3 internal rounds. At the beginning of each main round, the top half of the state is xored with the round key, keyx. The result is applied to the input of the AES round. All internal rounds are provided with a key from the key generation unit, with the exception of the last internal round where the string of zeros is used as a key. After executing three internal rounds, the obtained result is xored with the bottom half of the state and concatenated back to create a new state. This process is repeated until all 12 main rounds are completed. As a result, 36 clock cycles are required to hash a single message block.

The key generation unit is shown in Figure 3.45. Key and/or keyx are generated for each main round using this circuit.

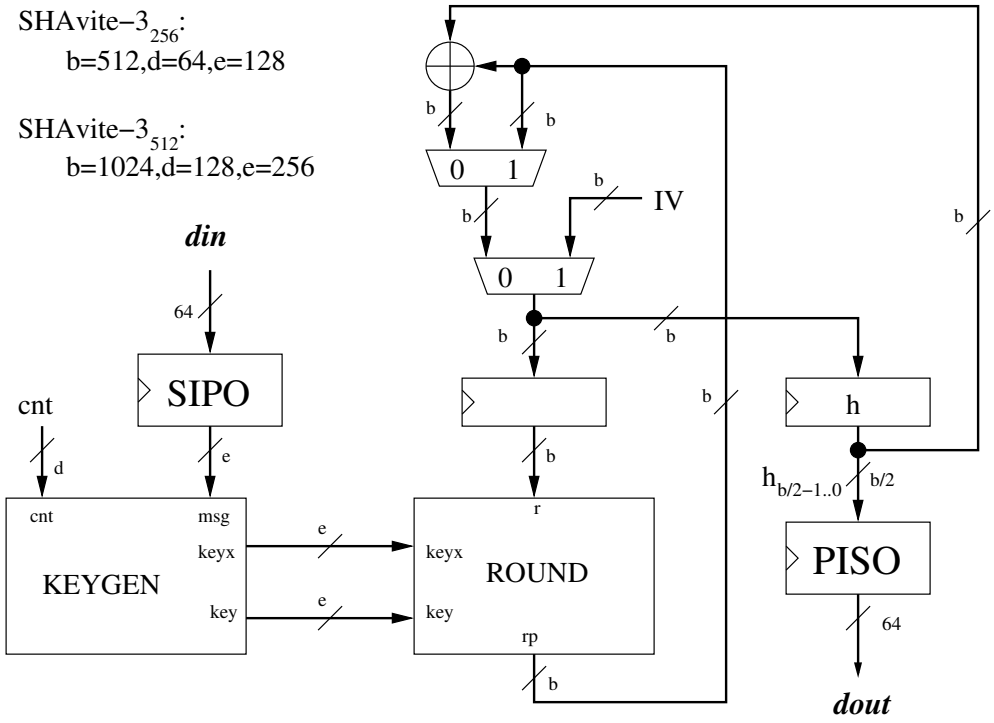


Figure 3.43: Shavite3 : Datapath

3.15.2 256 vs 512 Variant Differences

SHAvite-3-512 has the state size doubled compared to SHAvite-3-256. The basic operation in the top level datapath remains the same. The number of main rounds is increased from 12 to 14. The ROUND unit is also doubled in size. Figure 3.46 illustrates changes in the ROUND unit for SHAvite-3-512. The same operation as $Round_2 56$ is performed with the exception that the number of internal rounds is increased from 3 to 4. Figure 3.47 describes a new key generation circuit. Once again, one can find similarity in terms of the design, with the exception that all major building blocks are duplicated. Note that in this design four clock cycles are required to load a 1024-bit message block, 256 bits per clock cycle.

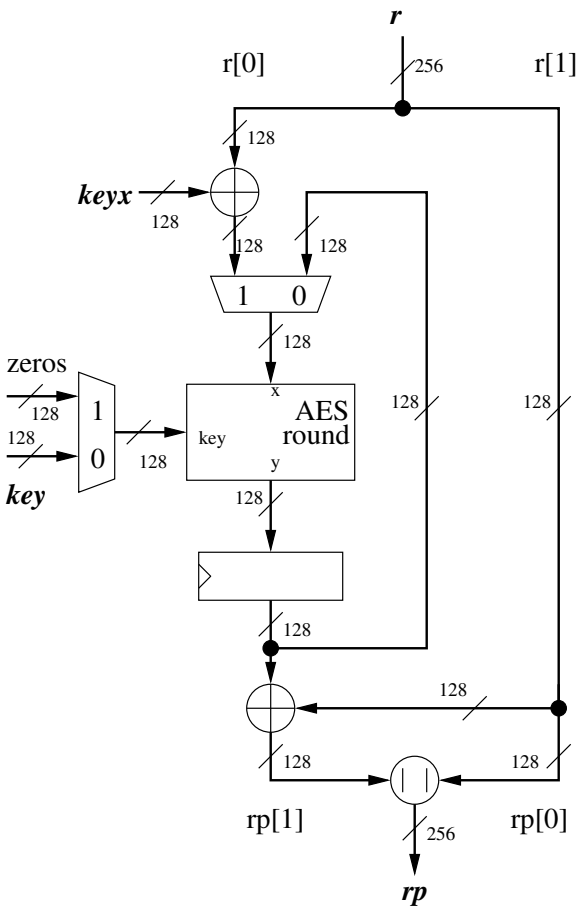


Figure 3.44: Shavite3 : Round (256 bits)

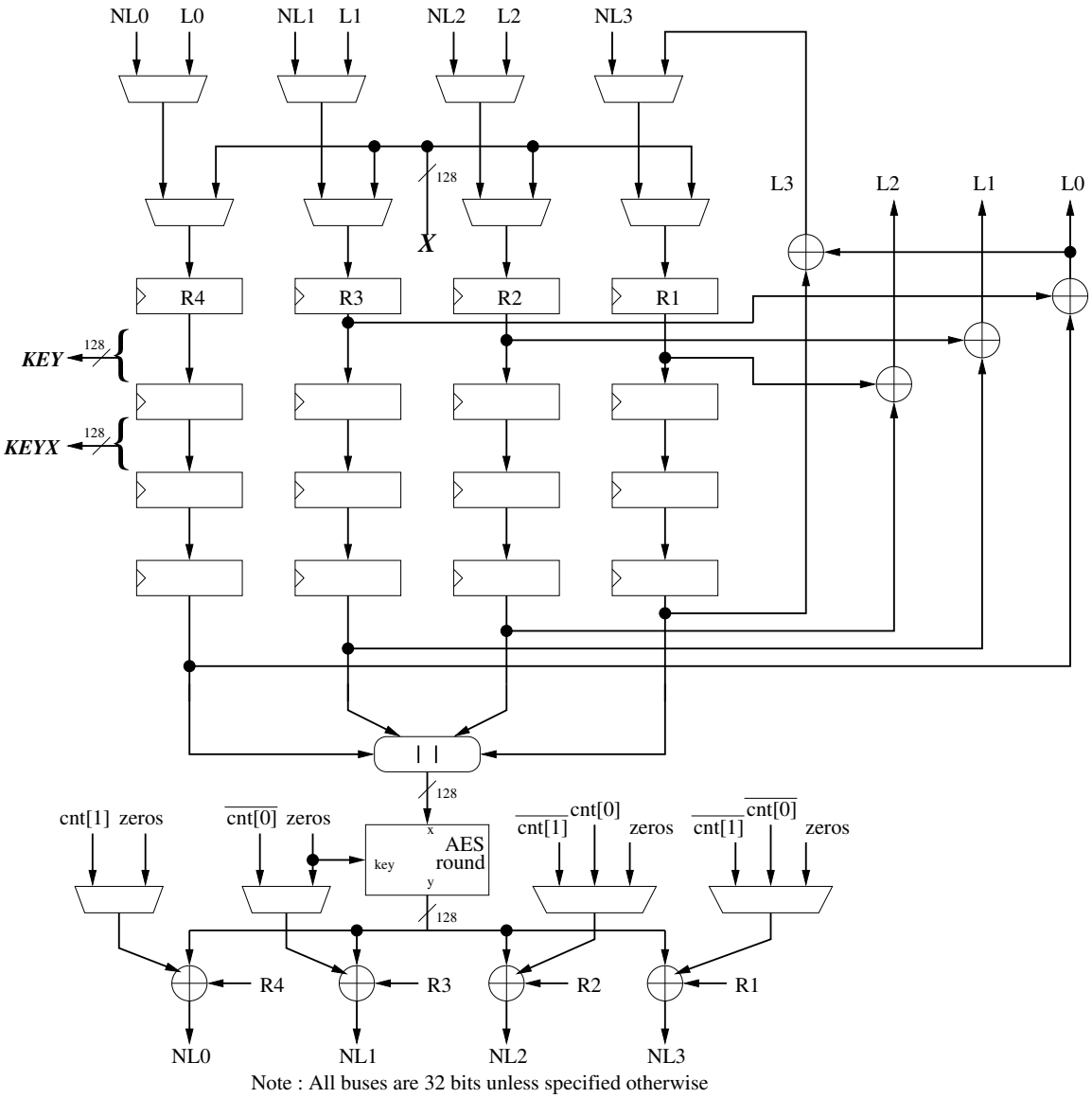


Figure 3.45: Shavite3 : Keygen (256 bits)

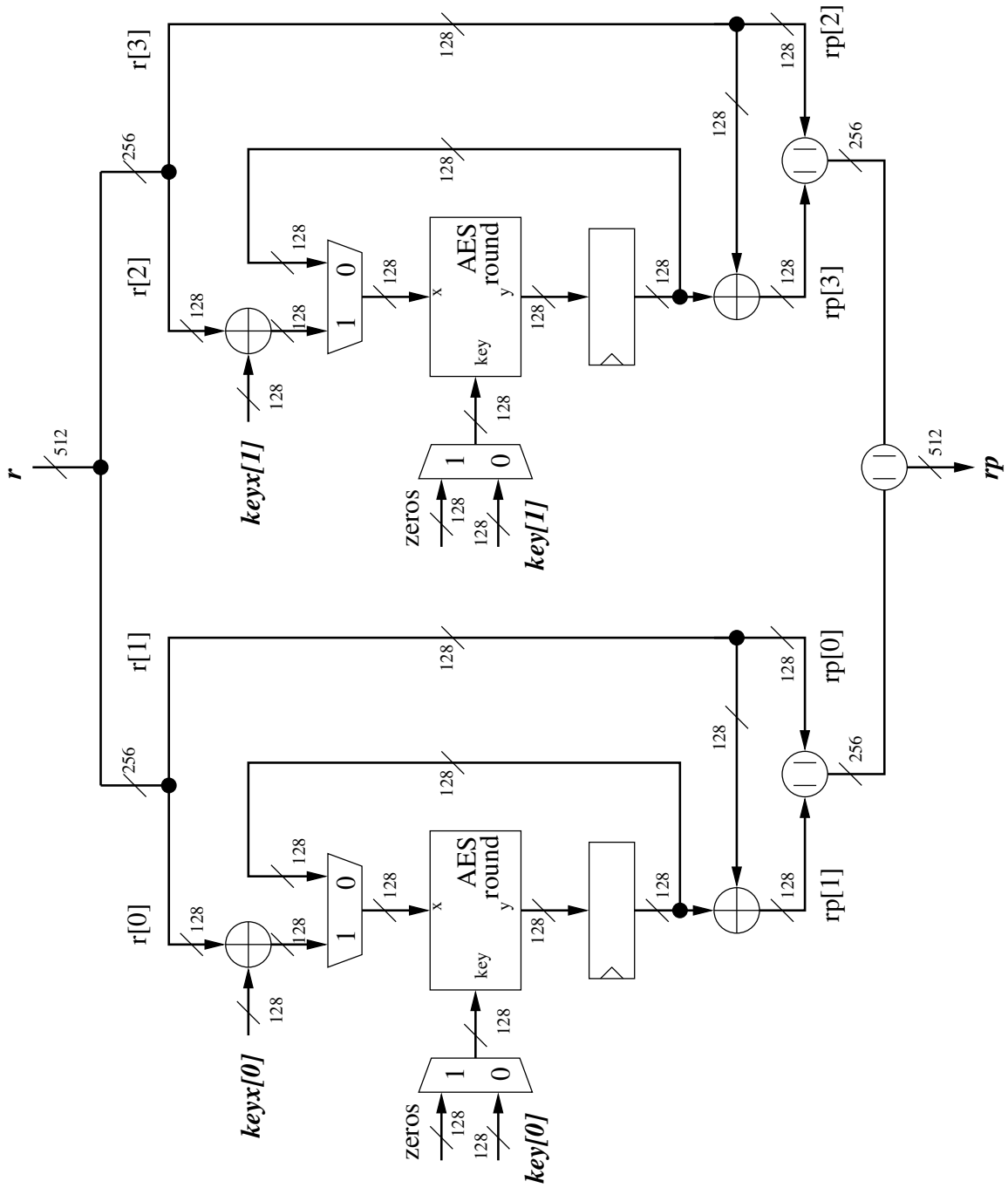


Figure 3.46: Shavite3 : Round (512 bits)

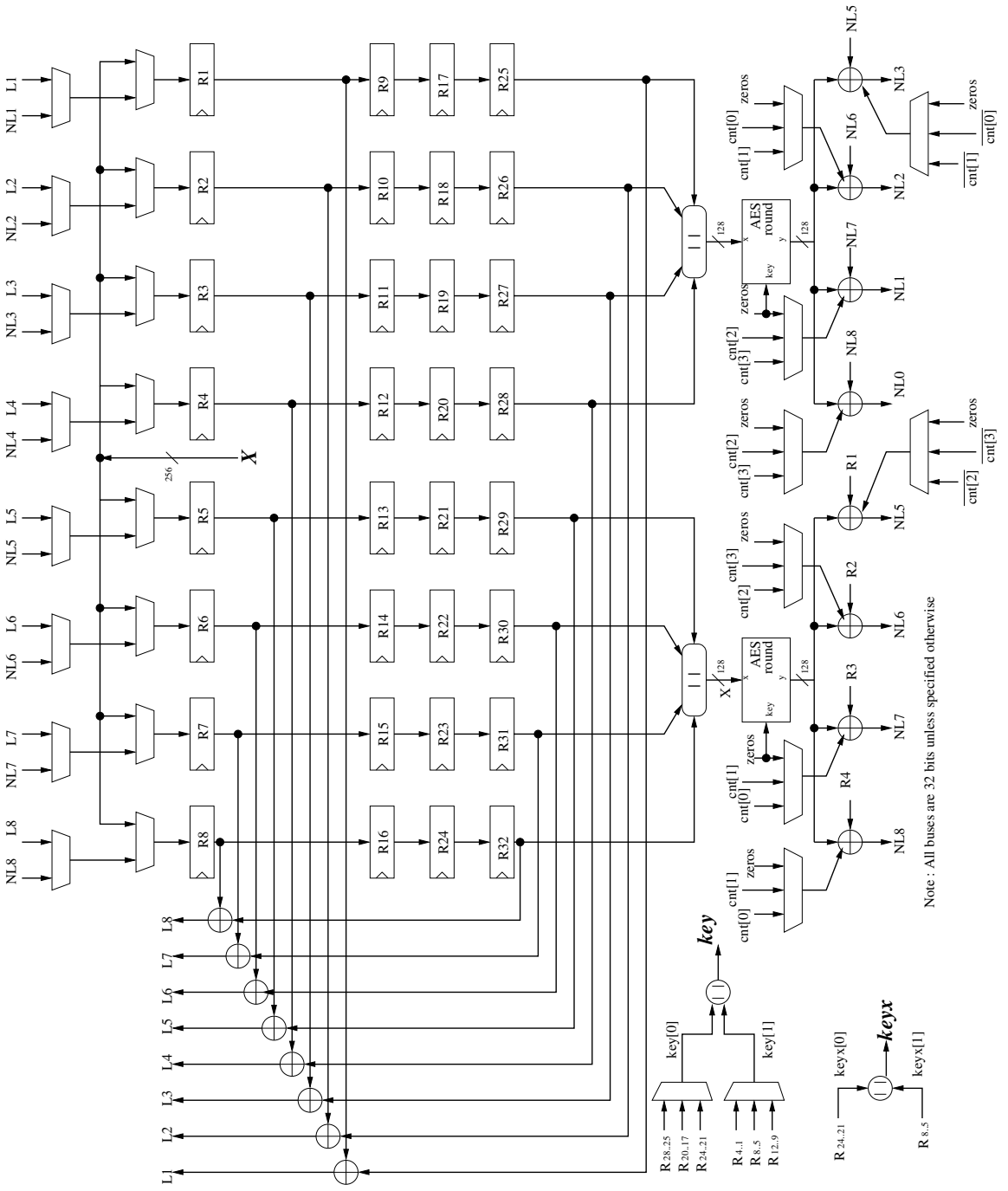


Figure 3.47: Shavite3 : Keygen (512 bits)

3.16 SIMD

3.16.1 Block Diagram Description

The block diagram of SIMD is shown in Figure 3.48. The design executes four steps at a time, and requires 9 clock cycles to process a message block. The Message Expansion unit requires a total of 8 clock cycles to fully expand a message block. Assuming that the message block is expanded, the first block of the message is xored with IV and used as a state. This state is transformed for 9 clock cycles (or four and a half round). If there is more than one message block, the chaining value is xored with a new message block and the same process is repeated again. The final hash value is obtained by truncating the chaining value. Both the input message block and the output hash value are required to switch their endianness in order to maintain correct operation.

The Message Expansion unit can be considered as a whole core by itself. In our design, an additional controller is added between FSM1 and FSM2 to control this unit. This ensures that we can keep processing an input block while reading the next message at the same time. The first part of Message Expansion is its NTT unit. The NTT unit is based on a folded 7-stage DFT that uses a 2-point DFT as its base. First, each byte of an input is zero-extended to 9 bits. Then, these 9-bit blocks are inserted into the 2-point NTT unit with its respective twiddle factor as an input. The twiddle factor is chosen based on the DFT's stage number. The calculation of the twiddle factors can be performed using the following VHDL code:

```
type halfptsx8 is array (natural range <>) OF std_logic_vector(7 downto 0);

function twiddle_gen(point : integer; pts : integer) return halfptsx8 is
  variable twiddle_factor : halfptsx8( 0 to pts/2 -1 ) := twiddle_factor_gen( pts );
  variable y : halfptsx8( 0 to pts/2 -1 );
  variable step : integer := (pts/point);
  variable cur_step : integer := 0;
begin
  if ( step = pts/2 ) then
    step := 0;
  end if;
  for i in 0 to pts/2-1 loop
    y(i) := twiddle_factor(cur_step);
    cur_step := cur_step + step;
    if (cur_step >= pts/2) then
      cur_step := cur_step - pts/2;
    end if;
  end loop;
  return y;
end twiddle_gen;
```

The function takes two inputs, *point* and *pts*, and returns one output, y. *point* refers

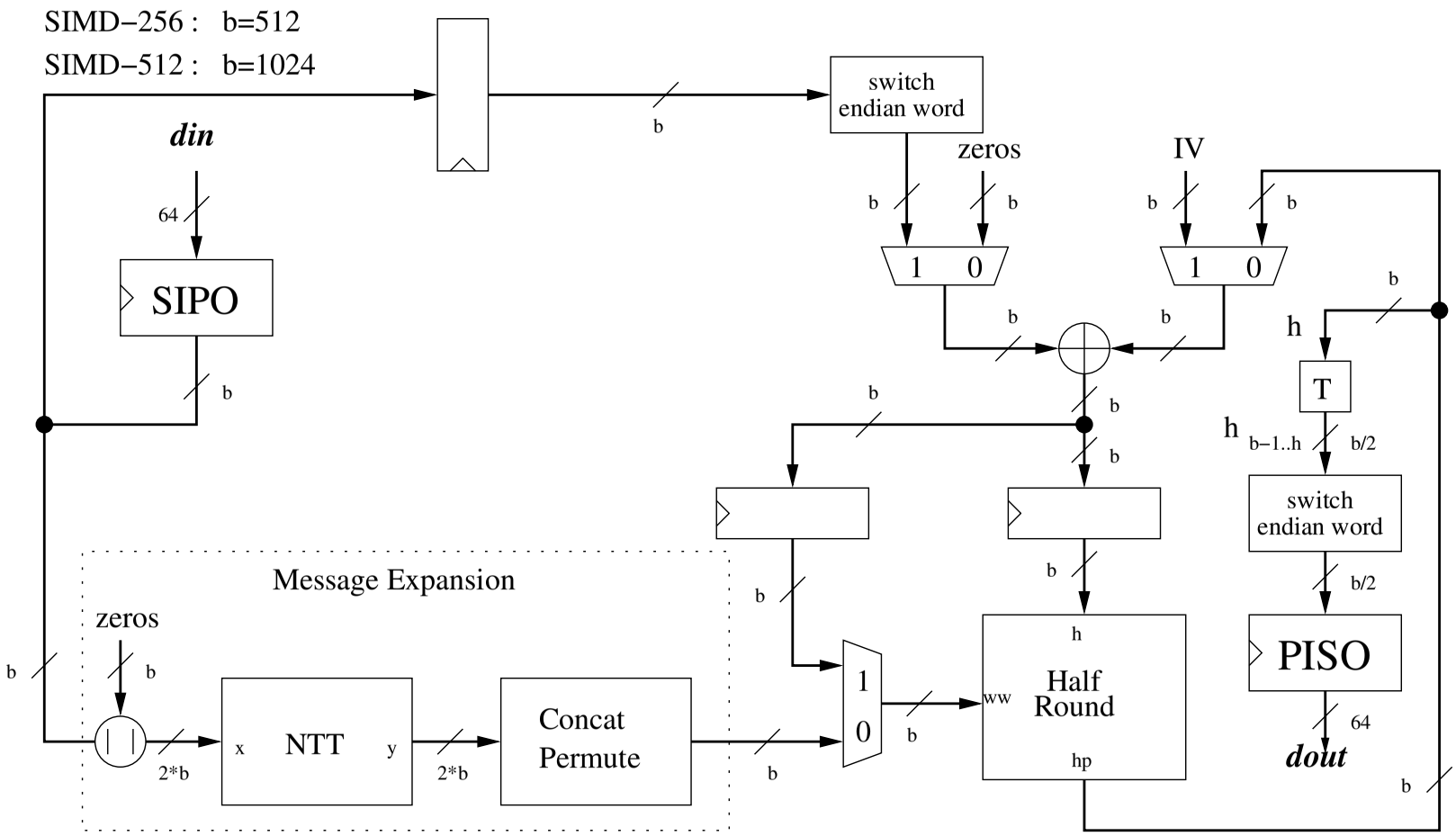


Figure 3.48: SIMD : Datapath

to the current DFT stage and *pts* stands for the maximum stage. For SIMD-256, *pts* is always equal to 128 as there are seven stages of NTT with 128 points. Seven values of the twiddle factor need to be generated for NTT with 128 points. These are 2,4,8,16,32,64 and 128. The output, y, returns an array of 64 by 8 bits.

The building base of a 2-point DFT is shown in Figure 3.50. The unit is built using several repetitions of the butterfly units. The input is viewed as an array of 9 bit values. Two consecutive values go into each butterfly unit, and their outputs are combined to form a result. In a Butterfly unit, a modulo 257 reduction is applied to ensure that there is no bit growth. The Modulo 257 unit is shown in Figure 3.51.

Since there are 7 stages for a 128-point DFT, it is necessary to permute the input before entering 2-point DFTs so that the unit can perform correctly. The permutation can be derived from any diagrams showing at least 4 to 8 point DFT (many of such diagrams are available on the internet). In the last cycle, where a 128-point DFT is performed, the result is permuted to its correct position. The final operation of the NTT unit involves the addition between the output of the 128-point DFT and an addition factor. Addition factor final is selected if the expanded message block is the last block of the message. Addition factor and addition factor final can be calculated using the following VHDL function, where $final$ is high for calculation of addition factor final and *pts* refers to the maximum-point DFT used in the design (128 for SIMD-256) :

```
type ptsx10 is array (natural range <>) OF std_logic_vector(9 downto 0);

function af_gen ( final : integer; pts : integer ) return ptsx10 is
  variable y : ptsx10(0 to pts-1);
  variable beta_i : std_logic_vector(17 downto 0);
  variable beta : std_logic_vector(7 downto 0);
begin
  if ( pts = 128 ) then
    beta := conv_std_logic_vector(98,8);
  else
    beta := conv_std_logic_vector(163,8);
  end if;
  y(0) := conv_std_logic_vector(1,10);
  for i in 1 to pts-1 loop
    beta_i := y(i-1) * beta;
    beta_i := conv_std_logic_vector((conv_integer(beta_i) mod 257),18);
    y(i) := beta_i(9 downto 0);
  end loop;
  if ( final = 1 ) then
    if ( pts = 128 ) then
      beta := conv_std_logic_vector(58,8);
    else
      beta := conv_std_logic_vector(40,8);
    end if;
    beta_i := "000000000000000001";
    for i in 0 to pts-1 loop
      y(i) := y(i) + beta_i(9 downto 0);
      beta_i := beta_i(9 downto 0) * beta;
      beta_i := conv_std_logic_vector((conv_integer(beta_i) mod 257),18);
    end loop;
  end if;
```

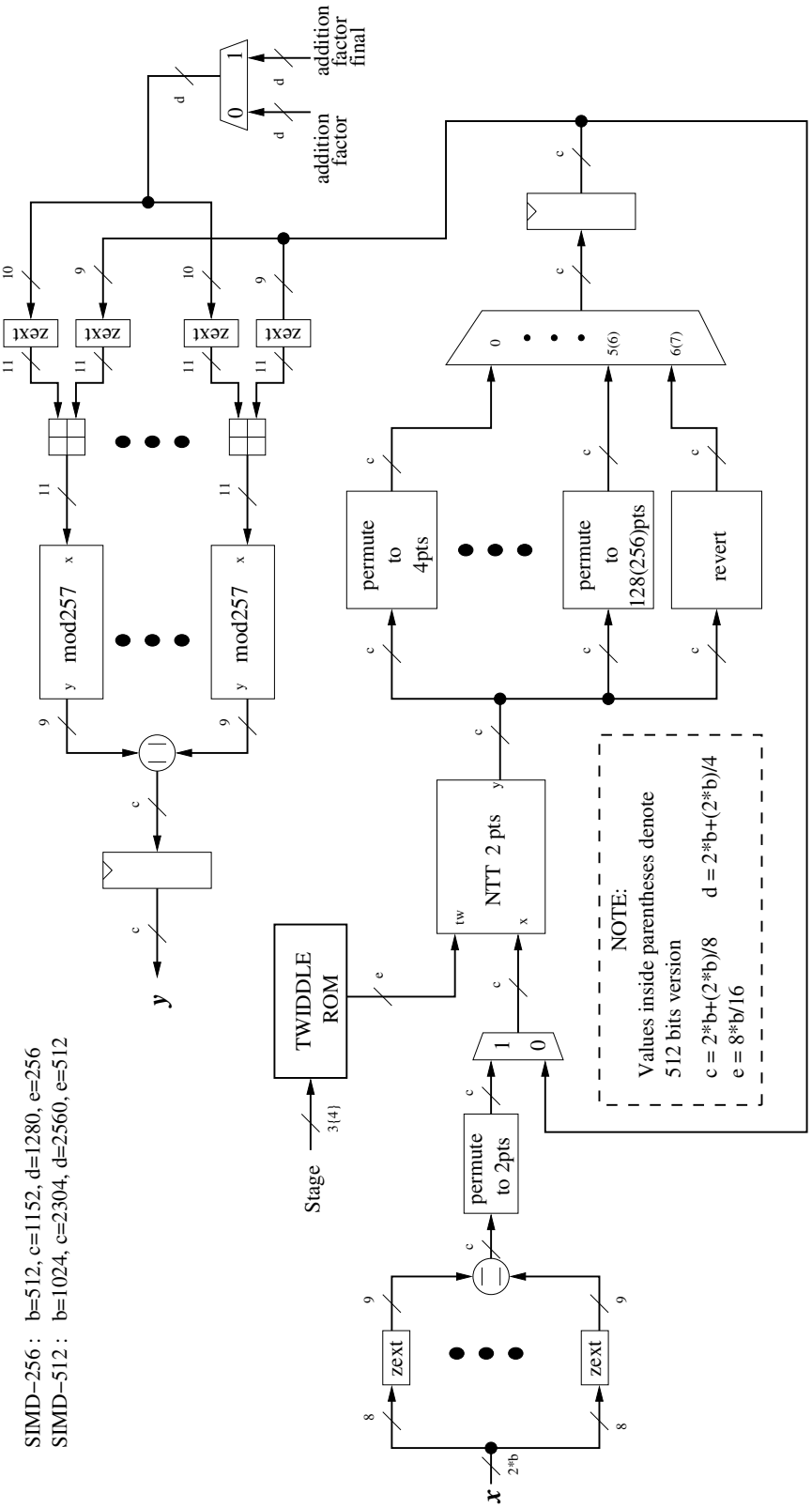


Figure 3.49: SIMD : NTT

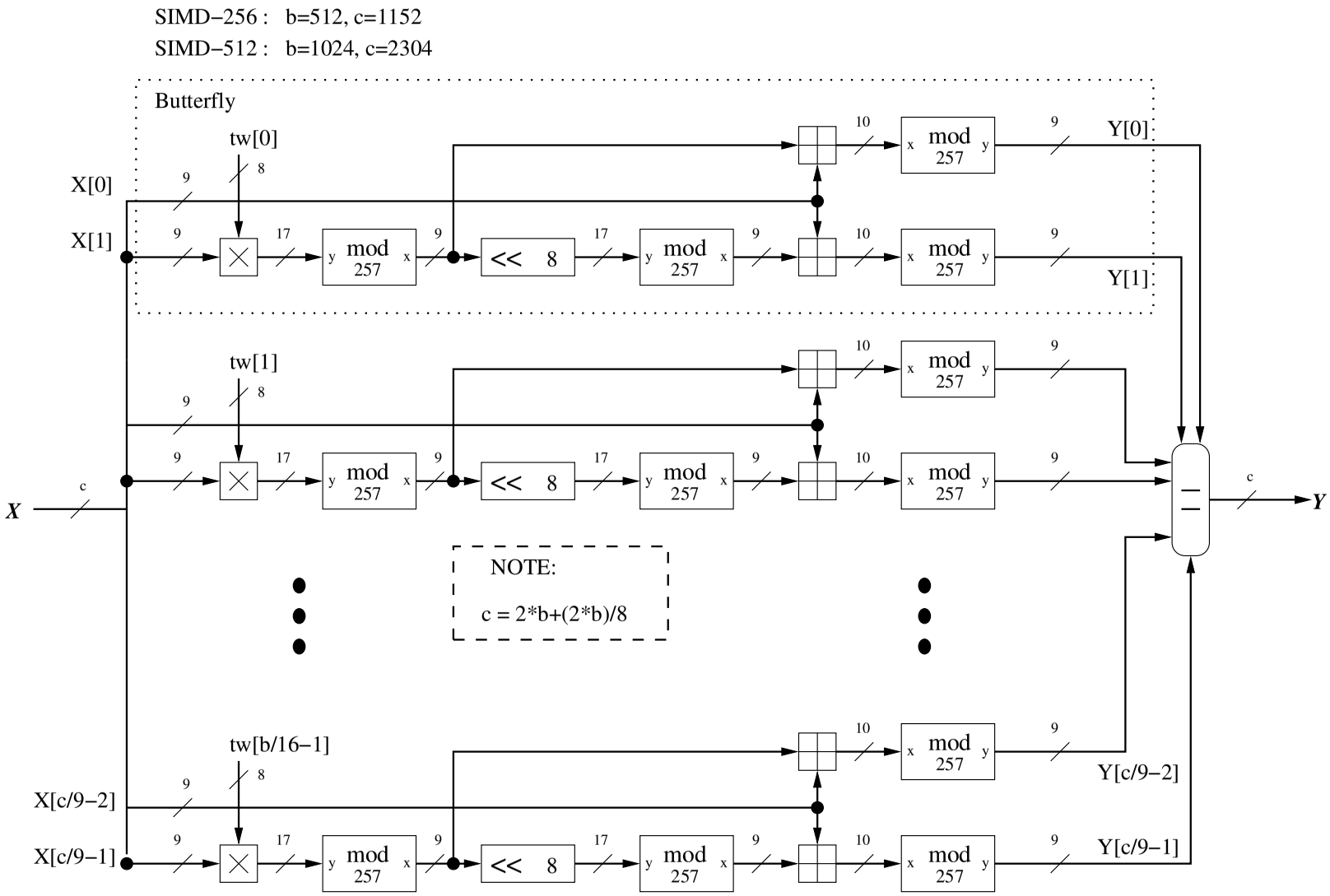


Figure 3.50: SIMD : NTT 2 Points

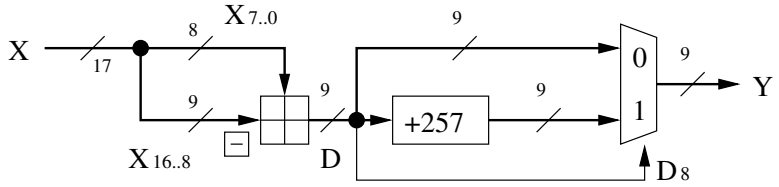


Figure 3.51: SIMD : Modulo 257

```
    return y;
end af_gen;
```

The last step of the Message Expansion unit is to perform Concatenated Code and Permute. These operations are described in Section 1.2.2 of [36]. A diagram of Concat Permute (CP) is shown in Figure 3.52. To reduce the resource requirements in Concatenated Code, Permute is performed first. An input is viewed as an array of 9 bit values. For SIMD-256, this array size is equal to 128. Permute 1 forms a matrix of 32 x 4 of 18 bits each. This doubles the size of an input. The permutation of Permute 1 is given as follows:

$$with 0 \leq j \leq 3$$

$$Z_j'^i = \begin{Bmatrix} x[8i+2j] \,||\, x[8i+2j+1] & when & 0 \leq i \leq 15 \\ x[8i+2j-128] \,||\, x[8i+2j-64] & when & 16 \leq i \leq 23 \\ x[8i+2j-191] \,||\, x[8i+2j-127] & when & 24 \leq i \leq 31 \end{Bmatrix}$$

Next, the matrix Z' is permuted to form W' in Permute 2. The permutation table is given in Table 3.14, where $W_j'^i = Z_j'^{P(i)}$.

Table 3.14: SIMD: Permute 2

cycle	0				1				2				3			
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	4	6	0	2	4	5	3	1	15	11	12	8	9	13	10	14
cycle	4				5				6				7			
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	17	18	23	20	22	21	16	19	30	24	25	31	27	29	28	26

A multiplexer selects appropriate data depending on the cycle number. A selected value is viewed as an array of 4 x 4 with 18 bits at each location. Each 18-bit value is split in

half and entered into Lift module shown in Figure 3.53. An output from the Lift module is then multiplied by a constant. The constant is 185 for the first four cycles and 233 for the last four cycles. The outputs are combined back into a 4 x 4 matrix of 32-bit words.

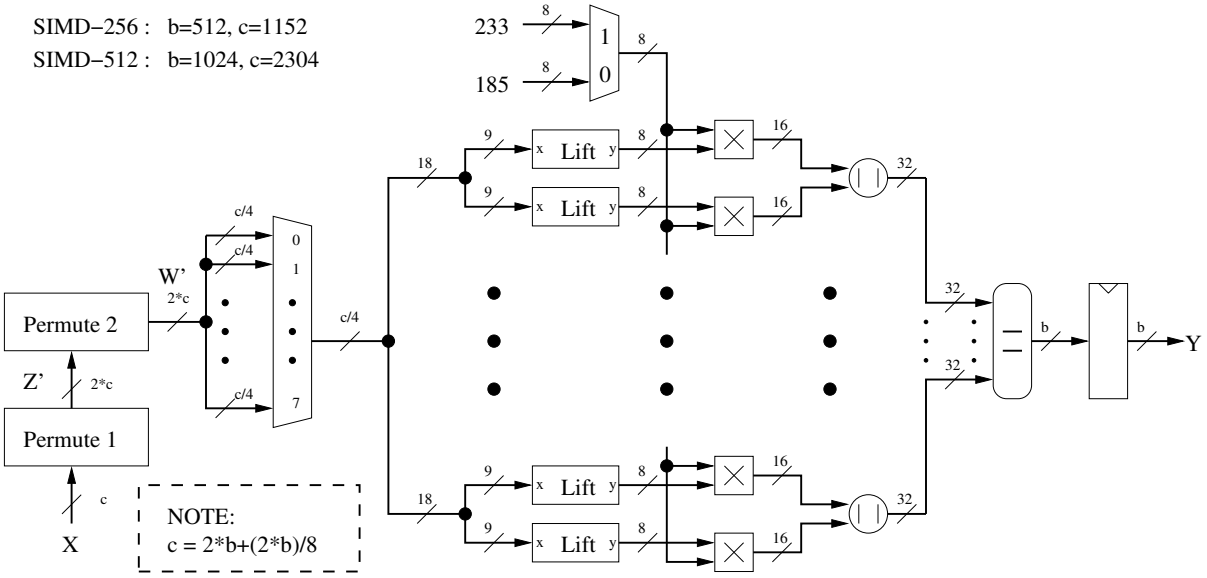


Figure 3.52: SIMD : Concatenate and Permute (CP)

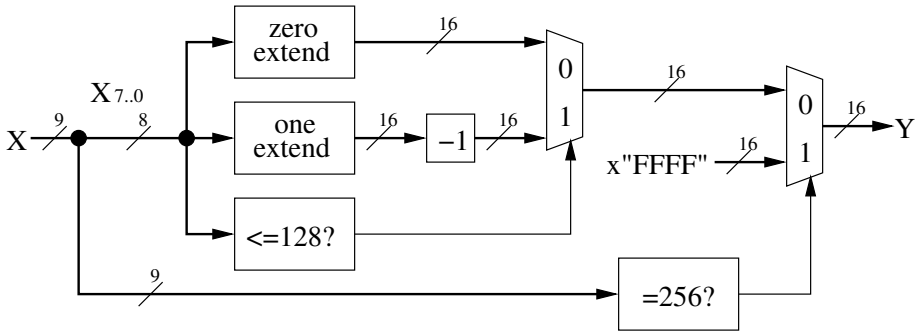


Figure 3.53: SIMD : Lift

The core operation of our SIMD's design is the Half Round module. This module is equivalent to four steps of the SIMD round. A block diagram of the Half Round operation is shown in Figure 3.54. Half Round is based of 16 QS units with *quarterstep* as its core. *quarterstep* is shown in Figure 3.55. There are four inputs to *quarterstep*. *ain* comes from its adjacent *quarterstep* controlled by a multiplexer. w comes from the message expansion unit. r and s are rotation constants depending on the round number. Additionally, *phi*

is selected to perform IF or MAJ depending on the round number. These constants are given as follows:

$\phi^{(i)}$	$r^{(i)}$	$s^{(i)}$
IF	π_0	π_1
IF	π_1	π_2
IF	π_2	π_3
IF	π_3	π_0
MAJ	π_0	π_1
MAJ	π_1	π_2
MAJ	π_2	π_3
MAJ	π_3	π_0

Round	π_0	π_1	π_2	π_3
0	3	23	17	27
1	28	19	22	7
2	29	9	15	5
3	4	13	10	25

Finally, a permutation given for Ain is given as follows :

$$p^{(0)}(j) = j \oplus 1$$
$$p^{(1)}(j) = j \oplus 2$$
$$p^{(2)}(j) = j \oplus 3$$

3.16.2 256 vs 512 Variant Differences

The biggest change in SIMD-512 is the increase in the block size. This change causes the size of NTT to increase. DFT now requires 8 stages instead of 7 and the size of the butterfly increases by a factor of two. In the CP unit, Permute 1 is defined as follows:

$$with 0 \leq j \leq 7$$
$$Z_j'^i = \begin{Bmatrix} x[8i+2j] \,||\, x[8i+2j+1] & when & 0 \leq i \leq 15 \\ x[8i+2j-256] \,||\, x[8i+2j-128] & when & 16 \leq i \leq 23 \\ x[8i+2j-383] \,||\, x[8i+2j-255] & when & 24 \leq i \leq 31 \end{Bmatrix}$$

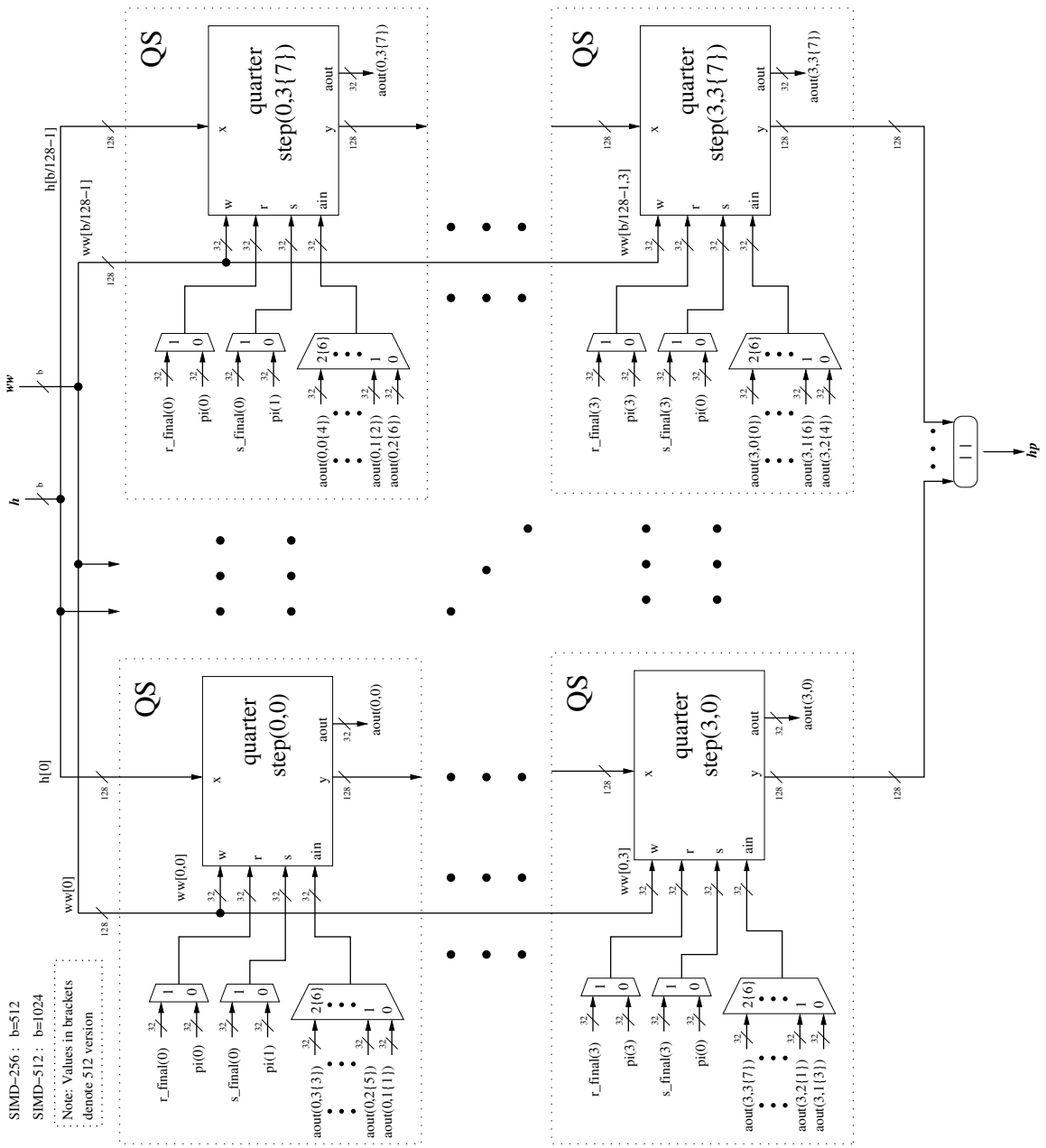


Figure 3.54: SIMD : Half Round

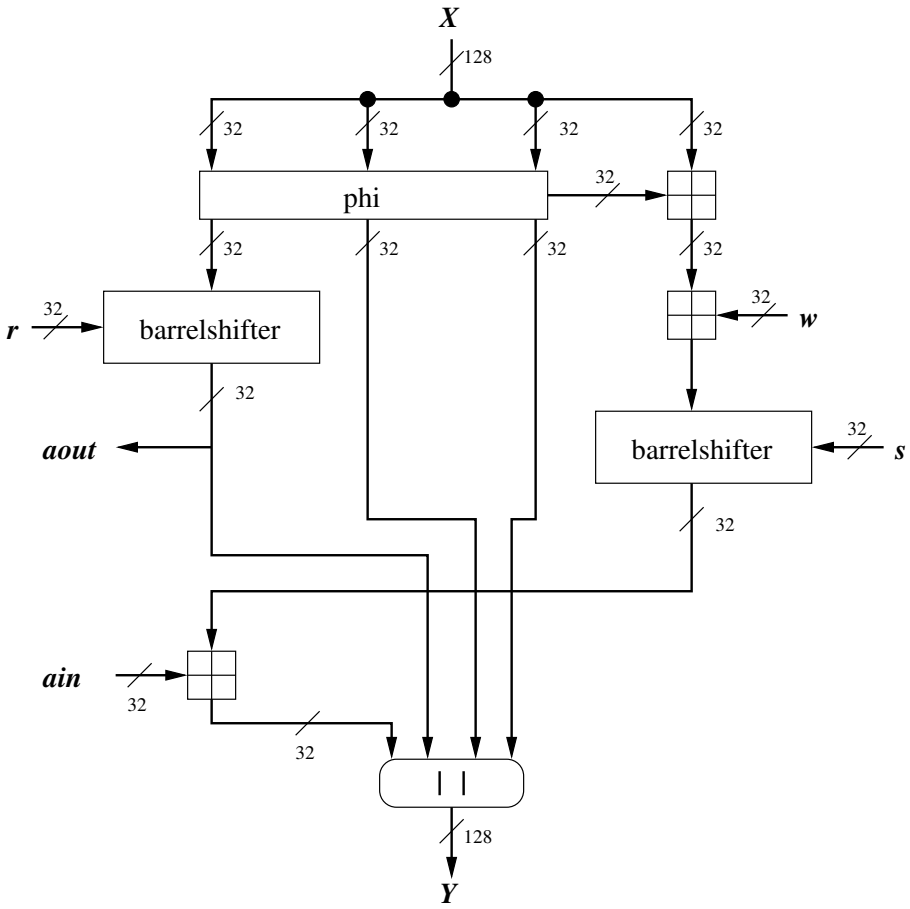


Figure 3.55: SIMD : Quarter Step

Additionally, the Feistal Ladder is increased from 4 to 8. This necessitates the change in the Ain permutation to the mux. The permutation for SIMD-512 of Ain is given as follows:

$$p^{(0)}(j) = j \oplus 1$$
$$p^{(1)}(j) = j \oplus 6$$
$$p^{(2)}(j) = j \oplus 2$$
$$p^{(3)}(j) = j \oplus 3$$
$$p^{(4)}(j) = j \oplus 5$$
$$p^{(5)}(j) = j \oplus 7$$
$$p^{(6)}(j) = j \oplus 4$$

3.17 Skein

3.17.1 Block Diagram Description

The datapath of Skein is shown in Figure 3.56. This diagram is based on the Skein-256-256 construction. The datapath of Skein can be separated into two main parts, the key generation and the Skein's round. The round includes a layer of 64 bit additions and the 4x unrolled MIX and PERMUTE unit. An input message block is used to initialize the internal state of Skein. This state is viewed as an array of four 64-bit words. For every message block, a subkey is added to the state once for every 4 rounds of the MIX and PERMUTE operation. The total number of rounds for Skein-256 is 72. Because of the 4x unrolled architecture, these rounds are executed in 18 clock cycles. Then, the finalization is performed after the last round is executed. The finalization is performed at the end of each message block processing, in order to generate a new chaining value. This operation is equivalent to an addition between the state and the key, followed by an xor with the current message block.

The key generation unit takes two input sources, the chaining value and the tweak. The chaining value acts as a key to the key generation unit. It is computed from the previous message block or taken as an initialization vector in the beginning of the message. A tweak is controlled by the controller. Its full specification can be found under Section 3.4 of [37]. In Figure 3.57, a key generation unit for our design is shown. s is the subkey counter. It gets reset for every new message block.

In Figure 3.58, a 4-times unrolled MIX and PERMUTE unit is shown. This unit is based on eight instantiations of the MIX operation. The MIX operation is shown in Figure 3.59. The rotation constants are given in Table 3.15. The round number is calculated modulo 8. The permutation executed between each round of MIX is also given in Table 3.16.

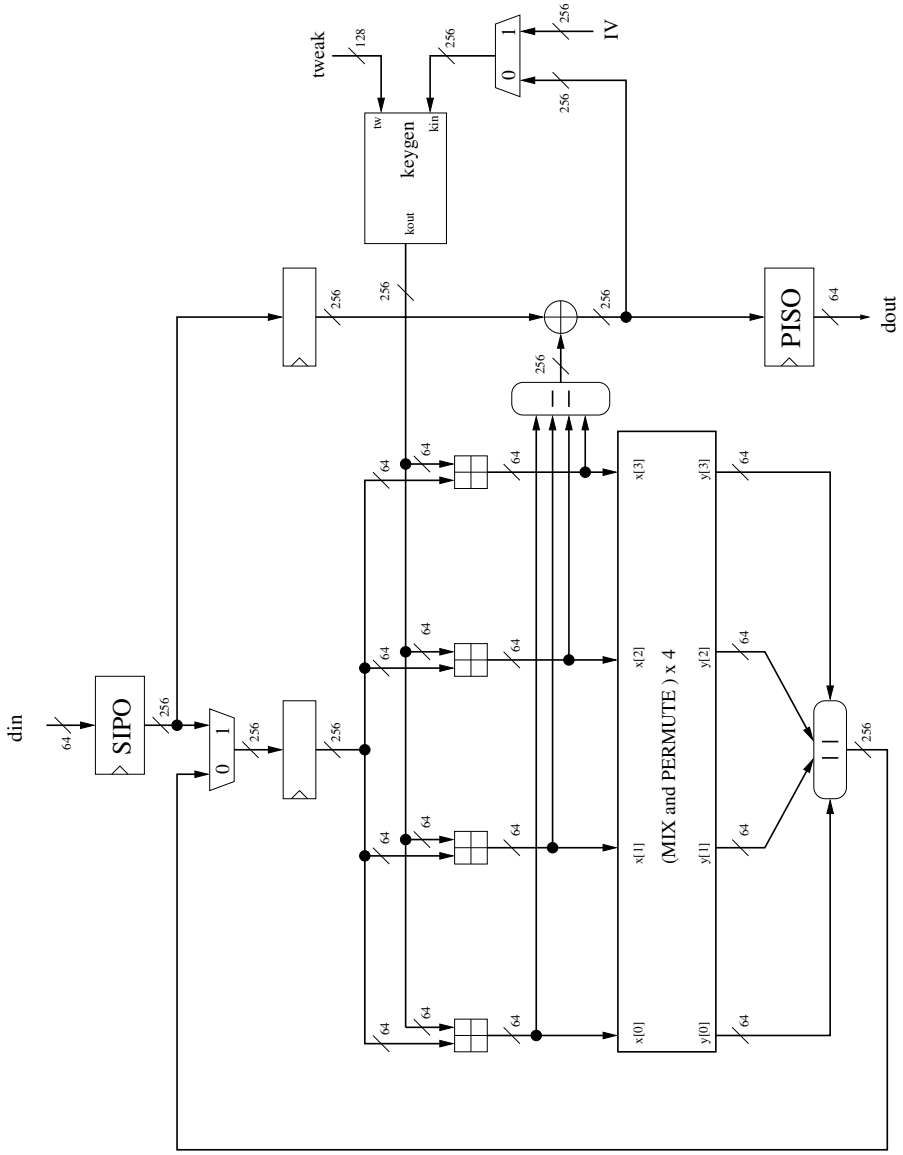


Figure 3.56: Skein : Datapath

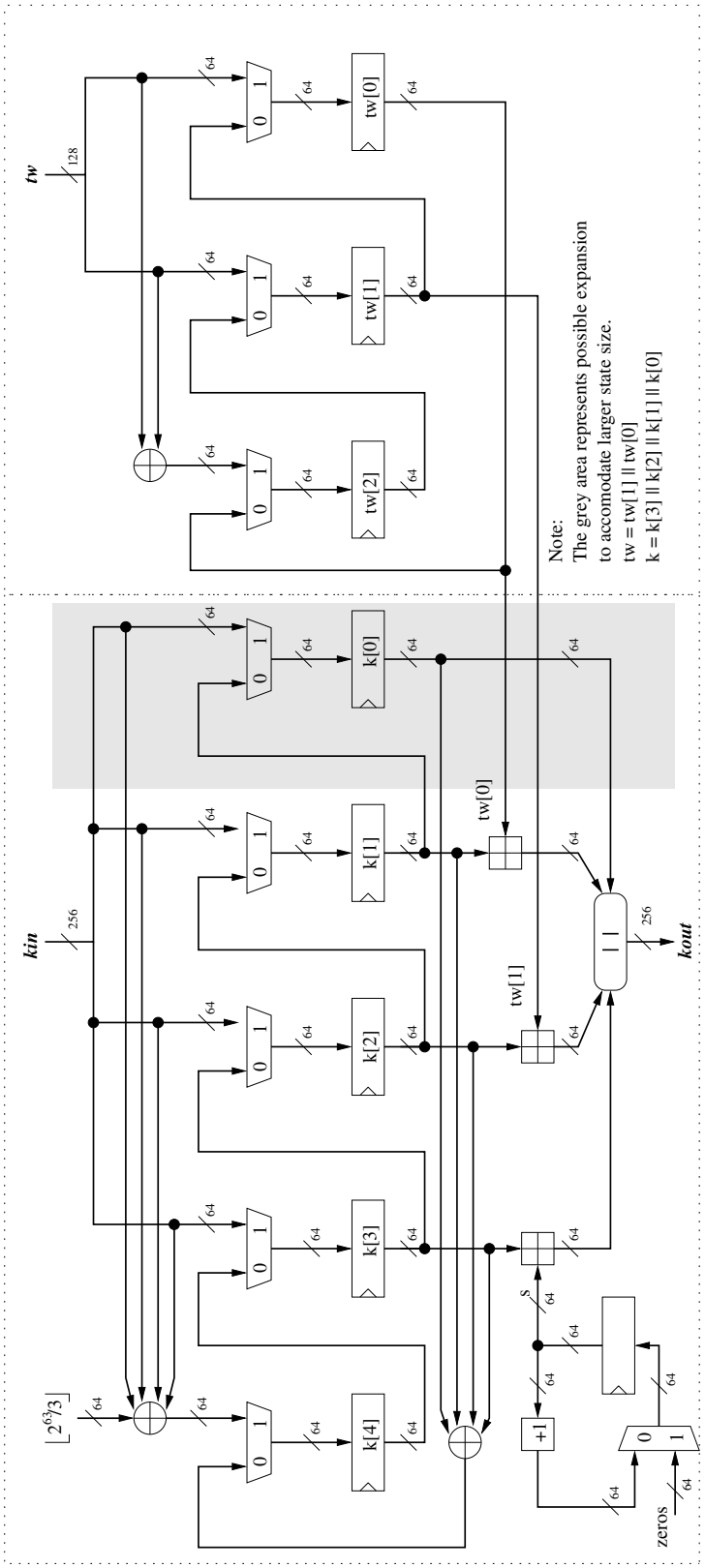
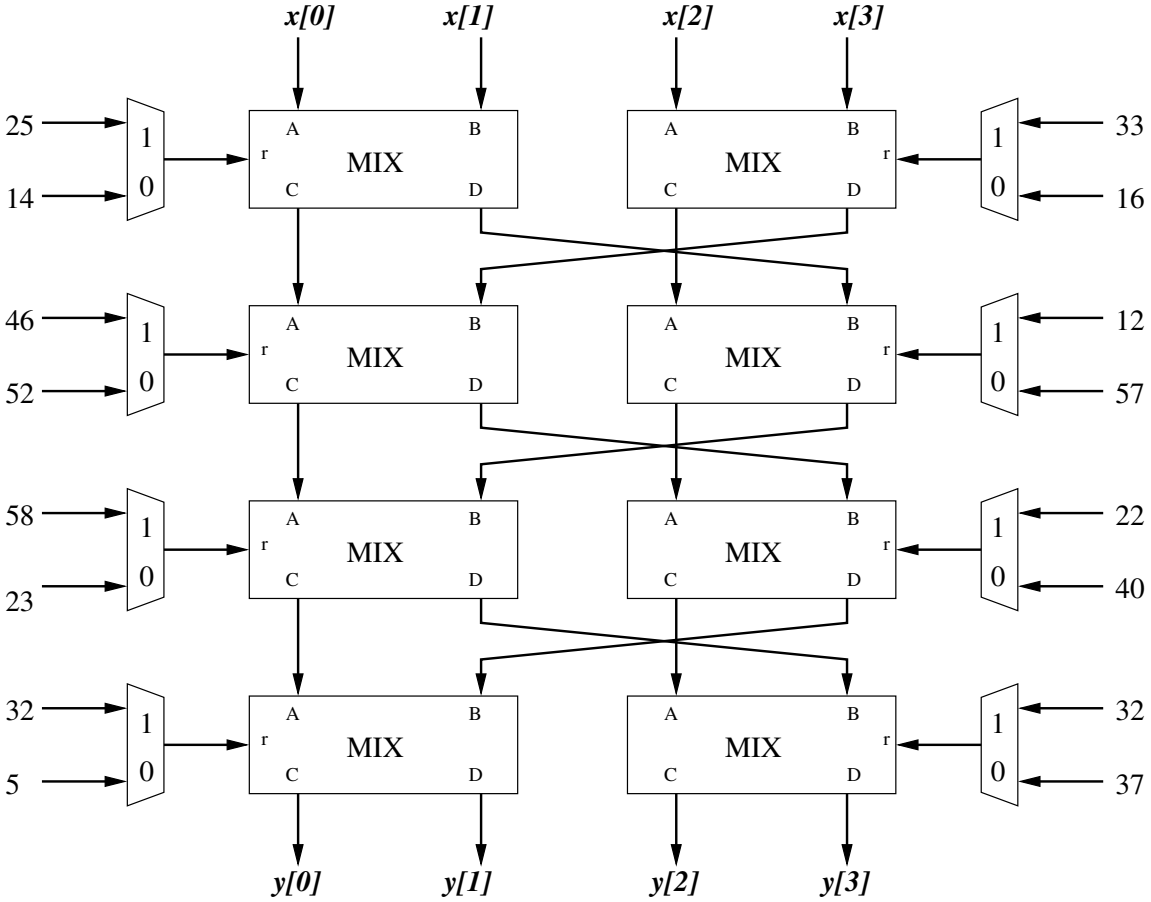


Figure 3.57: Skein : Key Generation



Note: All buses are 64 bits

Figure 3.58: Skein : Round

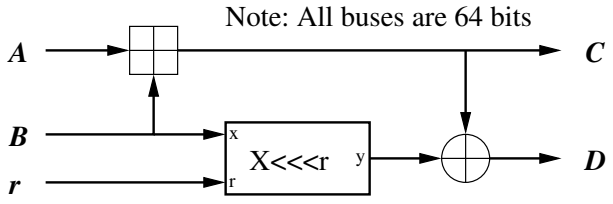


Figure 3.59: Skein : Mix

Table 3.15: Skein: Rotation Constants, N_w is the number of words

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

Table 3.16: Skein: Permutation, N_w is the number of words

		y															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x	4	0	3	2	1												
	$N_w = 8$	2	1	4	7	6	5	0	3								
	16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

3.17.2 256 vs 512 Variant Differences

Skein-512 as submitted to the SHA-3 contest is based on Skein-512-512. The state size in this version is doubled compared to the state size in the 256 bit version. Changes in the key generation unit involve the extension of the register to the right side of the grey area in Figure 3.57. In essence, k[3] becomes k[7], k[2] becomes k[6] and so on. In the MIX and PERMUTE unit, the MIX column is expanded to 4 columns instead of 2. Hence, the permutation and rotation constants are changed. Finally, the

Chapter 4: Design Summary and Results

4.1 Design Summary

All timing data presented in this paper are based on the equations for the Hash Function Execution Time (in clock cycles) and Throughput (in Mbit/s) as summarized in Table 4.1. Additionally, the major parameters of both hash function variants (with a 256-bit and a 512-bit output), and the predicted area and throughput ratios are summarized in Table 4.2.

Table 4.1: The I/O Data Bus Width (in bits), Hash Function Execution Time (in clock cycles), and Throughput (in Mbits/s) for the 256-bit and 512-bit variants of all SHA-3 candidates and the current standard, SHA-2. T denotes the clock period in μs. Values different between 256-bit and 512-bit variants are shown in bold.

	256-bit variants			**512-bit variants**		
Function	**I/O Bus width**	**Hash Time [cycles]**	**Throughput [Mbit/s]**	**I/O Bus width**	**Hash Time [cycles]**	**Throughput [Mbit/s]**
BLAKE	64	2+8+20·N+4	512/(20·T)	64	2+**16/2**+14·N+**8/2**	**1024**/(**14**·T)
BMW	64	2+8/8+N+1	512/T	64	2+**16/16**+N+**8/16**	**1024**/T
CubeHash	64	2+4+16·N+160+4	256/(16·T)	64	2+4+16·N+160+**8**	256/(16·T)
ECHO	64	3+24+27·N+4	1536/(27·T)	64	3+**16**+**31**·N+**8**	**1024**/(**31**·T)
Fugue	32	2+N+18+8	32/T	32	2+**4**·N+**21**+16	32/(**4**·T)
Groestl	64	3+8+21·N+4	512/(21·T)	64	2+**16/2**+29·N+**8/2**	**1024**/(**29**·T)
Hamsi	32	3+1+3·(N-1)+6+8	32/(3·T)	**64**	3+1+6·(N-1)+6+8	**64**/(**6**·T)
JH	64	3+8+36·N+4	512/(36·T)	64	3+8+36·N+**8**	512/(36·T)
Keccak	64	3+17+24·N+4	1088/(24·T)	64	2+**9**+24·N+**8**	**576**/(24·T)
Luffa	64	3+4+9·N+9+4	256/(9·T)	64	3+4+9·N+**2**·**9**+**8**	256/(9·T)
Shabal	64	3+8+1+25·N+3·25+4	512/(25·T)	64	3+8+1+49·N+3·49+**8**	512/(49·T)
Shavite-3	64	3+8+37·N+4	512/(37·T)	64	3+16+**57**·N+**8**	**1024**/(**57**·T)
SIMD	64	3+8+8+9·N+4	512/(9·T)	64	3+16+**9**+9·N+**8**	**1024**/(9·T)
Skein	64	2+4+9·N+4	256/(9·T)	64	2+8+9·N+**8**	**512**/(9·T)
SHA-256	32	2+1+65·N+8	512/(65·T)	64	2+1+**81**·N+**8**	**1024**/(81·T)

In general, area of the circuit is most affected by the state size. As a result, the predicted area ratio between the 512 and 256-bit variants can be roughly approximated as shown in Eq. 4.1 below.

Table 4.2: Major parameters of the 256-bit and 512-bit variants of all SHA-3 candidates and the current standard, SHA-2. Values different between 256-bit and 512-bit variants are shown in bold. The first approximations of the predicted area ratio (512 vs. 256-bit variant) and the predicted throughput ratio (512 vs. 256-bit variant) are given in the last two columns.

	256-bit variant				**512-bit variant**				**Predicted**	**Predicted**
	state size	**Block size**	**Round no**	**Word size**	**State size**	**Block size**	**Round no**	**Word size**	**Area Ratio**	**Thr Ratio**
BLAKE	**512**	**512**	**10**	32	**1024**	**1024**	**14**	**64**	**2**	**1.43**
BMW	**512**	**512**	**16**	32	**1024**	**1024**	**16**	**64**	**2**	**2**
CubeHash	1024	256	16	32	1024	256	16	32	1	1
ECHO	2048	**1536**	**8**	32	2048	**1024**	**10**	32	1	**0.53**
Fugue	**960**	32	**2**	32	**1152**	32	**4**	32	**1.2**	**0.5**
Groestl	**512**	**512**	**10**	64	**1024**	**1024**	**14**	64	**2**	**1.43**
Hamsi	**512**	**32**	**3**	32	**1024**	**64**	**6**	32	**2**	**1**
JH	1024	512	36	64	1024	512	36	64	1	1
Keccak	1600	**1088**	24	64	1600	**576**	24	64	1	**0.53**
Luffa	**768**	256	8	32	**1280**	256	8	32	**1.67**	**1**
Shabal	1408	512	48	32	1408	512	48	32	1	1
SHAvite-3	**512**	**512**	**36**	32	**1024**	**1024**	**56**	32	**2**	**1.29**
SIMD	**512**	**512**	36	32	**1024**	**1024**	36	32	**2**	**2**
Skein	**256**	**256**	72	64	**512**	**512**	72	64	**2**	**2**
SHA-2	**256**	**512**	64	**32**	**512**	**1024**	**80**	**64**	**2**	**1.6**

$$Predicted\_Area\_Ratio_{512/256} = \frac{State\_size_{512}}{State\_size_{256}} \tag{4.1}$$

Additional factors that can affect the actual area ratio include:

- *message block size*, which determines the size of the input shift register.
- *output size*, which determines the size of the output shift register.
- *logic of the main round*, which may be more complex in case of a 512-bit variant of a function.
- logic required for *message expansion or key generation*, which may be more complex in case of a 512-bit variant of a function.
- logic required for *initialization and finalization*, which may not follow the datapath width.
- *size of the control unit*, which is likely to remain constant between two variants, but typically contributes only small percentage to the total circuit area.

Similarly, the throughput ratio between 512 and 256-bit variants can be estimated based on the equation for throughput, Eq. 2.4, under the assumption that the critical path, and thus the clock period, are similar in both variants.

$$Predicted\_Throughput_{512/256} = \frac{Thr_{512}}{Thr_{256}} = \frac{\frac{Block\_size_{512}}{Round_{512}}}{\frac{Block\_size_{256}}{Round_{256}}} \tag{4.2}$$

In the actual circuits, the clock period, T, may change due to the increase in the critical path in case of a 512-bit variant of a function.

For both predictions, the actual results will most likely vary and be dependent on a particular FPGA family, and selected tools. This includes:

- *Tools performance.* The results may vary depending on the tool version and vendor.
- *Elementary resource behavior.* The compatibility between a logic operation and a corresponding resource may affect the area, e.g., in case of the look-up table size.
- *Design congestion.* Particularly in the low cost FPGAs, the low number of available resources may limit the tool's ability for successful routing, resulting in longer than expected critical path and/or increase in resource utilization.

Based on these predictions, we can divide the 15 investigated algorithms into 6 major groups:

- Group 1: area and throughput are not affected by the change of the output size: *CubeHash, JH, Shabal.*
- Group 2: area and throughput both double: *BMW, SIMD, Skein.*
- Group 3: area and throughput both increase, but area increases more: *BLAKE, Groestl, SHAvite-3, and SHA-2.*
- Group 4: area stays the same and throughput decreases: *ECHO, Keccak.*

- Group 5: area increases and throughput stays the same: *Hamsi, Luffa.*
- Group 6: area increases and throughput decreases: *Fugue.*

The groups are ranked ascendingly from the point of view of the throughput to area ratio, with Group 1 being the best and Group 6 the worst.

4.2 Results

In Tables 4.3 and 4.4, the absolute results obtained for our implementations of the current standard SHA-2 are summarized. The results are repeated across seven selected FPGA families. In terms of the design, an architecture by Chaves et al. [39] is selected, as it is considered one of the best known SHA-2 architectures.

Table 4.3: Results for the reference implementation of SHA-256 (architecture with rescheduling)

	Spartan 3	**Virtex 4**	**Virtex 5**	**Cyclone II**	**Cyclone III**	**Stratix II**	**Stratix III**
Max. Clk Freq. [MHz]	90.84	183.02	207.00	111.04	126.86	158.08	212.81
Throughput [Mbit/s]	715.56	1441.60	1630.49	874.65	999.27	1245.18	1676.29
Area	838	838	433	1655	1653	973	963
Throughput to Area Ratio	0.85	1.72	3.77	0.53	0.60	1.28	1.74

Table 4.4: Results for the reference implementation of SHA-512 (architecture with rescheduling)

	Spartan 3	**Virtex 4**	**Virtex 5**	**Cyclone II**	**Cyclone III**	**Stratix II**	**Stratix III**
Max. Clk Freq. [MHz]	90.06	168.75	215.84	93.54	113.15	177.34	234.80
Throughput [Mbit/s]	1138.51	2133.31	2728.68	1182.53	1430.44	2241.93	2968.34
Area	1367	1403	646	2916	2915	1639	1620
Throughput to Area Ratio	0.83	1.52	4.22	0.41	0.49	1.37	1.83

In Tables 4.5 and 4.6, the actual performance measures of the 256 and 512-bit variants of all investigated algorithms are reported for the case of Xilinx Virtex 5 and Altera Stratix III, respectively.

Tables 4.7 and 4.8 summarize the clock frequencies of the implemented algorithms across seven selected FPGAs. For 512-bit variants, some algorithms are unable to fit in the selected FPGAs. These cases are denoted by 'N/A' in the following tables. Specifically, BLAKE is unable to fit in Spartan 3 due to the routing congestion. The same can be said

Table 4.5: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Xilinx Virtex 5 FPGAs

	Max. Clk Freq. [MHz]			**Throughput [Mbit/s]**			**Area [CLB slices]**			**Throughput/Area**		
	512	**256**	**ratio**	**512**	**256**	**ratio**	**512**	**256**	**ratio**	**512**	**256**	**ratio**
BLAKE	27.14	101.98	0.27	1985.02	2610.64	0.76	5429	1851	2.93	0.37	1.41	0.26
BMW	8.45	10.89	0.78	8655.87	5576.70	1.55	10401	4400	2.36	0.83	1.27	0.66
CubeHash	211.00	199.36	1.06	3376.00	3189.79	1.06	775	730	1.06	4.36	4.37	1.00
ECHO	200.97	234.85	0.86	6638.33	13360.47	0.50	5958	5445	1.09	1.11	2.45	0.45
Fugue	138.49	98.47	1.41	1107.88	3151.17	0.35	924	956	0.97	1.20	3.30	0.36
Groestl	180.15	355.87	0.51	6361.09	8676.50	0.73	3466	1884	1.84	1.84	4.61	0.40
Hamsi	171.38	248.08	0.69	1828.05	2646.15	0.69	2201	946	2.33	0.83	2.80	0.30
JH	220.95	282.20	0.78	3142.33	4013.51	0.78	1319	1275	1.03	2.38	3.15	0.76
Keccak	157.18	238.38	0.66	3772.39	10806.51	0.35	1417	1229	1.15	2.66	8.79	0.30
Luffa	220.12	281.53	0.78	7043.81	8008.02	0.88	2164	1154	1.88	3.25	6.94	0.47
Shabal	156.30	128.12	1.22	1633.17	2623.96	0.62	1355	1266	1.07	1.21	2.07	0.58
SHAvite-3	213.45	208.55	1.02	3834.56	2885.89	1.33	1954	1130	1.73	1.96	2.55	0.77
SIMD	36.37	40.89	0.89	4138.55	2325.90	1.78	17016	9288	1.83	0.24	0.25	0.97
Skein	27.20	49.79	0.55	1547.32	1416.14	1.09	2120	1312	1.62	0.73	1.08	0.68
SHA-2	215.84	207.00	1.04	2728.68	1630.49	1.67	646	433	1.49	4.22	3.77	1.12

Table 4.6: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Stratix III FPGAs

	Max. Clk Freq. [MHz]			**Throughput [Mbit/s]**			**Area [ALUTs]**			**Throughput/Area**		
	512	**256**	**ratio**	**512**	**256**	**ratio**	**512**	**256**	**ratio**	**512**	**256**	**ratio**
BLAKE	39.07	109.21	0.36	2857.69	2795.78	1.02	10927	1969	5.55	0.26	1.42	0.18
BMW	7.44	16.45	0.45	7618.56	8422.40	0.90	25225	12632	2.00	0.30	0.67	0.45
CubeHash	216.12	237.64	0.91	3457.92	3802.24	0.91	1942	1935	1.00	1.78	1.96	0.91
ECHO	246.00	164.20	1.50	8125.94	9341.16	0.87	20085	21689	0.93	0.40	0.43	0.94
Fugue	206.27	123.64	1.67	1650.16	3956.48	0.42	2775	3594	0.77	0.59	1.10	0.54
Groestl	250.38	270.27	0.93	8841.00	6589.44	1.34	6288	3103	2.03	1.41	2.12	0.66
Hamsi	181.16	294.81	0.61	1932.37	3144.64	0.61	5668	2320	2.44	0.34	1.36	0.25
JH	359.45	364.30	0.99	5112.18	5181.16	0.99	3354	3117	1.08	1.52	1.66	0.92
Keccak	269.61	296.30	0.91	6470.64	13432.27	0.48	3575	4458	0.80	1.81	3.01	0.60
Luffa	268.02	307.31	0.87	8576.64	8741.26	0.98	6888	3304	2.08	1.25	2.65	0.47
Shabal	219.73	126.87	1.73	2295.95	2598.30	0.88	3413	3600	0.95	0.67	0.72	0.93
SHAvite-3	215.38	255.00	0.84	3869.28	3528.65	1.10	5610	2497	2.25	0.69	1.41	0.49
SIMD	43.38	47.40	0.92	4935.68	2696.53	1.83	47671	22376	2.13	0.10	0.12	0.86
Skein	50.91	52.29	0.97	2896.21	1487.36	1.95	6396	3602	1.78	0.45	0.41	1.10
SHA-2	234.80	212.81	1.10	2968.34	1676.29	1.77	1620	963	1.68	1.83	1.74	1.05

about BMW in Cyclone II, Cyclone III and Stratix II, where the design congestion due to multi-operand additions exhausts available routing resources. For BMW and SIMD in Spartan 3 and ECHO in Cyclone II, resource utilization of its 512-bit variant exceeds the available resources of the largest FPGA device in a given family.

Table 4.7: Clock frequencies of all SHA-3 candidates (256-bit variants) and SHA-256 expressed in MHz (post placing and routing)

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III
BLAKE	41.87	79.82	101.98	52.40	52.37	85.77	109.21
BMW	4.19	12.37	10.89	7.69	8.41	13.45	16.45
CubeHash	84.70	187.58	199.36	115.67	133.83	179.40	237.64
ECHO	52.10	131.90	234.85	0.00	105.70	109.50	164.20
Fugue	39.67	72.86	98.47	53.25	60.71	83.75	123.64
Groestl	105.72	234.74	355.87	132.00	148.46	216.73	270.27
Hamsi	90.37	200.88	248.08	148.83	183.52	193.87	294.81
JH	119.36	221.58	282.20	173.43	215.89	267.45	364.30
Keccak	96.32	202.47	238.38	165.07	174.28	198.65	296.30
Luffa	129.84	260.28	281.53	171.64	173.43	219.88	307.31
Shabal	30.99	114.03	128.12	69.57	68.76	105.40	126.87
SHAvite-3	84.60	152.23	208.55	95.40	114.40	170.00	255.00
SIMD	17.20	29.25	40.89	21.66	23.97	37.07	47.40
Skein	18.22	38.16	49.79	22.30	25.14	38.89	52.29
SHA-256	90.84	183.02	207.00	111.04	126.86	158.08	212.81

Table 4.8: Clock frequencies of all SHA-3 candidates (512-bit variants) and SHA-512 expressed in MHz (post placing and routing)

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III
BLAKE	N/A	25.02	27.14	17.22	21.14	31.01	39.07
BMW	N/A	6.03	8.45	N/A	N/A	N/A	7.44
CubeHash	93.99	183.79	211.00	114.34	130.77	163.00	216.12
ECHO	85.17	190.30	200.97	N/A	135.24	166.64	246.00
Fugue	64.25	122.84	138.49	86.61	100.74	142.05	206.27
Groestl	66.18	202.76	180.15	124.10	133.96	187.72	250.38
Hamsi	69.00	158.05	171.38	103.31	117.16	128.68	181.16
JH	113.66	238.60	220.95	173.55	208.07	265.67	359.45
Keccak	80.39	135.63	157.18	161.39	173.07	207.68	269.61
Luffa	93.41	210.88	220.12	143.53	172.98	192.49	268.02
Shabal	61.97	131.01	156.30	120.16	136.02	181.03	219.73
SHAvite-3	75.31	161.97	213.45	86.71	103.73	140.53	215.38
SIMD	N/A	28.57	36.37	20.09	23.87	32.36	43.38
Skein	16.53	32.23	27.20	20.51	24.29	39.16	50.91
SHA-512	90.06	168.75	215.84	93.54	113.15	177.34	234.80

While high clock frequency shows that a cryptographic core can run fast, it is by no means an indication that an algorithm will perform well in a specific application. A throughput based on the performance for long messages alone is also not enough to justify the selection of a candidate. For many applications, an algorithm that can perform really well

for short messages may be more favorable than another that is exceptionally good for long messages but terribly slow for short ones. In Figures 4.1 and 4.2, we present the execution time as a function of the message length for 256-bit variants of all SHA-3 candidates and SHA-256. Similar graphs for 512-bit variants of all algorithms are presented in Figures 4.1 and 4.4.

Message sizes used in these diagrams represent sizes before padding. Padding is assumed to be done outside of a hash core (e.g., in software), and its time is not included in the execution time. Execution time (Latency) is a function of the clock period, T, and the total number of clock cycles required to process an N-block message, *HTime(N)*, given by Eq. 2.1. T is equal to an inverse of the clock frequency, which is given in Tables 4.7 and 4.8. The function *HTime(N)* for each candidate is given in Table 4.1.

For 256 and 512-bit variants, ECHO, Groestl and Luffa perform exceptionally well for both long (10,000 bits or more) and short (less than 1,000 bits) messages. Keccak also performs fairly well except in its 512-bit variant for long message sizes. In general, for 256 and 512-bit variants, most functions seem to outperform SHA-256 in terms of the execution time for long message sizes. For short message sizes (less than 1000 bits), CubeHash, Shabal and SIMD are not performing so well. Note that the results shown are dependent upon one specific design and one FPGA device only.

In Tables 4.9 and 4.10, we report ratios of major performance measures (Area, Throughput, and Throughput to Area Ratio) for a 512-bit variant vs. a 256-bit variant, averaged (using geometric mean) over:

- all seven FPGA families,
- three Xilinx families (Spartan 3, Virtex 4 and Virtex 5),
- four Altera families (Cyclone II, Cyclone III, Stratix II and Stratix III),
- three Low Cost families (Spartan 3, Cyclone II and Cyclone II), and
- four High Performance families (Virtex 4, Virtex 5 , Stratix II and Stratix III).

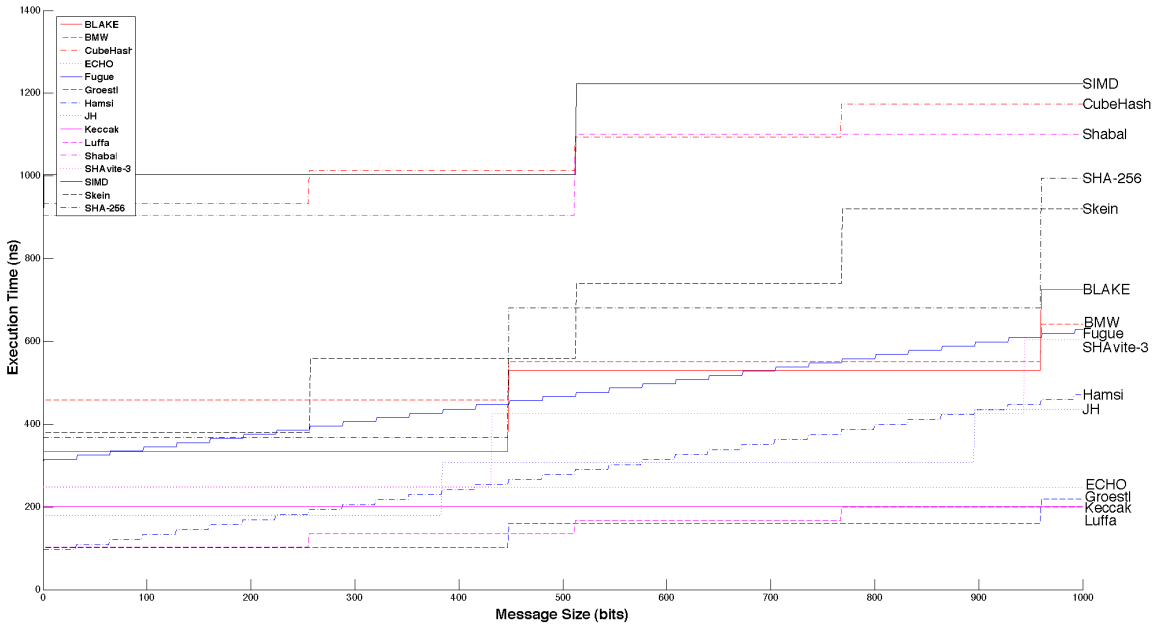


Figure 4.1: Execution time vs. message size for short messages up to 1,000 bits. 256-bit variants of all SHA-3 Candidates and SHA-256 in Virtex 5.

Comparing averaged throughput to area ratio between Altera and Xilinx families, Xilinx outperforms Altera in BLAKE, BMW, SHAvite-3 and SHA-256 by at least 15%.

Based on Table 2.2, 3 out of 4 of Xilinx favored algorithms rely heavily on multi-operand additions (more than three inputs). Interestingly, with the exception of SHAvite-3, which excels in Xilinx, Altera FPGAs give better results for algorithms that are based on AES (ECHO, Fugue and Groestl). This may due to the fact that the structure of SHAvite-3 is more complex than the structure of other AES-based algorithms. Serpent-based algorithms perform equally well in FPGAs of both vendors. In general, the averaged throughput to area ratio of 256 and 512-bit variants are within 25% between Xilinx and Altera with the exception of Keccak where Altera outperforms Xilinx by almost 50%. In terms of the Low Cost versus High Performance families, all algorithms seem to be equally matched between the two categories. (Note: Shabal result need to be regenerated in 512 bit version)

The comparisons between predicted throughput and area ratios given in the last two columns of Table 4.2 and the overall average (or geomeatric mean) of throughput and area ratios provide a good agreement with the experimental results. The only algorithms, for

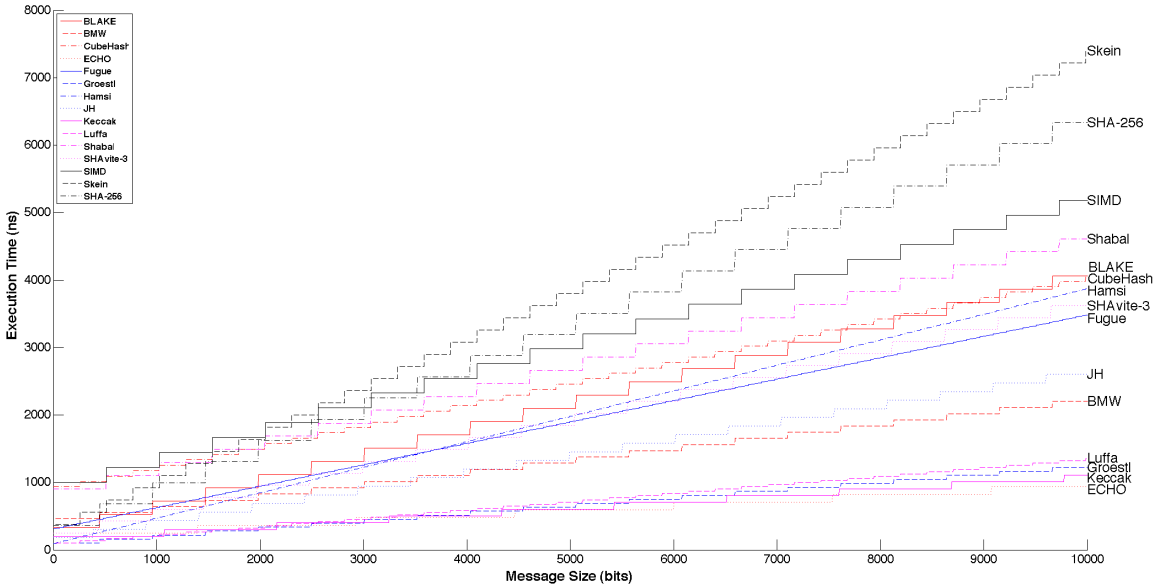


Figure 4.2: Execution time vs. message size for short messages up top 10,000 bits. 256-bit variants of all SHA-3 Candidates and SHA-512 in Virtex 5.

which the ratios are substantially different are listed below:

Blake: An underoptimized design may have caused the problem.

BMW: The throughput ratio is only half of what expected. This is most likely due to the increase in the word size for a 512-bit variant, where the word size is increased from 32 to 64-bits, causing the critical path to increase as well. The reduction in throughput is not by a factor of two because FPGAs are optimized for fast carry chain addition.

Hamsi: The area is larger than expected (2.4 vs. 2) because the message expansion unit does not increase by a factor of two but rather by a factor of four (256 kbit vs. 1 Mbit). Also, there is an increase in the area requirement of the main iterative round, where the number of diffusion layers is increased from 4 to 12 units. Hence, the reduction of throughput (0.69 vs. 1). This tripling of the number of diffusion layers also required the layers to be separated into two groups. Whereas in a 256-bit variant all 4 layers are computed in parallel, in a 512-bit variant 8 layers must be computed first, and the remaining 4 layers later.

Luffa: The area ratio is larger than predicted (2.08 vs. 1.67). This effect can be explained by the more complex computations performed in the 512-bit variant of Luffa during the Message Injection phase. In particular, the GF(2^8) constants used as inputs in

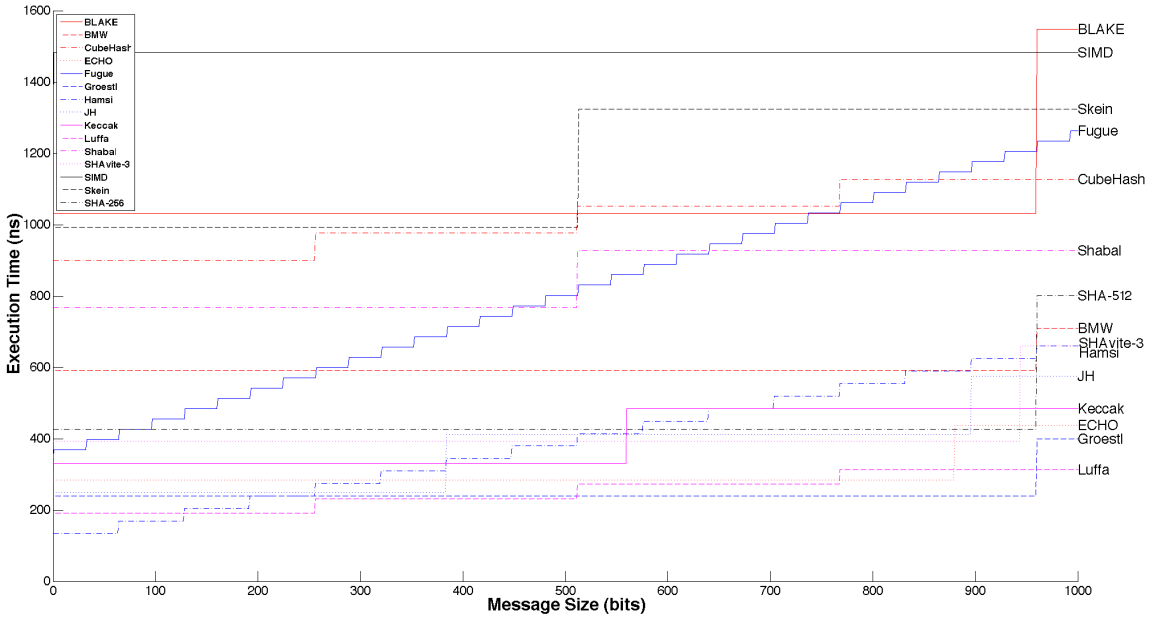


Figure 4.3: Execution time vs. message size for short messages up to 1,000 bits. 512-bit variants of all SHA-3 Candidates and SHA-512 in Virtex 5.

the Galois Field multiplications, change from small values of 1, 2, 3, 4 to the larger values including 01, 02, 04, 08, 10, 0A, 0F.

Modern FPGA families are created using different fabrication process, layout, and basic resources, which make comparison across several families in absolute terms difficult, if not impossible. To mitigate this problem, the normalized results are defined and calculated to provide a more direct comparison. A *normalized* result is calculated by dividing an absolute result for a SHA-3 candidate by the corresponding result for the reference implementation of the current standard SHA-2 with the same strength. Normalized results have no units, and can be reasonably compared across multiple families of FPGAs. An *overall* result is a geometric mean of results for all investigated FPGA families.

Tables 4.11, 4.12 and 4.13 summarize normalized results for the 256-bit variants of all SHA-3 candidates in terms of area, throughput, and throughput to area ratio, respectively. Similarly, Tables 4.14, 4.15 and 4.16 summarize normalized results for the 512-bit variants of all SHA-3 candidates in terms of the same parameters.

In conjunction with the normalized results, Figures 4.5 and 4.6 provide two-dimensional

Table 4.9: Ratio of the respective performance measures (Throughput (Thr), Area, Throughput to Area Ratio (Thr/Area)) for a 512-bit variant vs. 256-bit variant averaged (using geometric mean) over all 7 FPGA families (Overall), 3 families, and 4 Altera Families.

	Overall 512 vs. 256 variant			**Xilinx Families 512 vs. 256 variant**			**Altera Families 512 vs. 256 variant**		
	Area	**Thr**	**Thr/Area**	**Area**	**Thr**	**Thr/Area**	**Area**	**Thr**	**Thr/Area**
BLAKE	4.29	0.96	0.22	2.88	0.83	0.29	5.23	1.03	0.20
BMW	1.99	1.11	0.56	1.99	1.23	0.62	2.00	0.90	0.45
CubeHash	1.09	0.99	0.91	1.14	1.05	0.92	1.05	0.95	0.90
ECHO	1.02	0.78	0.77	1.07	0.73	0.69	0.97	0.83	0.86
Fugue	0.81	0.41	0.50	0.87	0.39	0.45	0.77	0.42	0.54
Groestl	1.87	1.14	0.61	1.74	0.94	0.54	1.98	1.32	0.66
Hamsi	2.40	0.69	0.29	2.37	0.75	0.31	2.41	0.65	0.27
JH	1.06	0.96	0.91	1.02	0.93	0.91	1.08	0.99	0.91
Keccak	0.92	0.45	0.49	1.03	0.38	0.37	0.85	0.52	0.61
Luffa	2.08	0.94	0.45	1.92	0.87	0.45	2.21	1.00	0.46
Shabal	1.00	0.82	0.83	1.04	0.72	0.69	0.96	0.91	0.94
SHAvite-3	2.07	1.19	0.58	1.91	1.28	0.67	2.20	1.13	0.51
SIMD	2.11	1.86	0.88	2.08	1.86	0.89	2.12	1.85	0.88
Skein	1.80	1.73	0.96	1.77	1.50	0.85	1.82	1.93	1.06
SHA-2	1.67	1.58	0.95	1.36	1.59	1.16	1.72	1.58	0.91

Table 4.10: Ratio of the respective performance measures (Throughput (Thr), Area, Throughput to Area Ratio (Thr/Area)) for a 512-bit variant vs. 256-bit variant averaged (using geometric mean) over all 7 FPGA families (Overall), 3 Low Cost families, and 4 High Performance Families.

	Overall 512 vs. 256 variant			**Low Cost Families 512 vs. 256 variant**			**High Performance Families 512 vs. 256 variant**		
	Area	**Thr**	**Thr/Area**	**Area**	**Thr**	**Thr/Area**	**Area**	**Thr**	**Thr/Area**
BLAKE	4.29	0.96	0.22	4.88	1.04	0.21	4.02	0.92	0.23
BMW	1.99	1.11	0.56	N/A	N/A	N/A	1.99	1.11	0.56
CubeHash	1.09	0.99	0.91	1.12	1.02	0.91	1.06	0.96	0.91
ECHO	1.02	0.78	0.77	1.05	0.84	0.80	1.00	0.75	0.75
Fugue	0.81	0.41	0.50	0.79	0.41	0.52	0.83	0.40	0.48
Groestl	1.87	1.14	0.61	1.79	1.17	0.65	1.94	1.11	0.58
Hamsi	2.40	0.69	0.29	2.38	0.70	0.29	2.41	0.69	0.29
JH	1.06	0.96	0.91	1.06	0.97	0.91	1.05	0.95	0.91
Keccak	0.92	0.45	0.49	0.89	0.49	0.56	0.95	0.43	0.45
Luffa	2.08	0.94	0.45	2.18	0.95	0.43	2.00	0.94	0.47
Shabal	1.00	0.82	0.83	0.98	0.97	0.99	1.01	0.73	0.72
SHAvite-3	2.07	1.19	0.58	2.10	1.17	0.56	2.05	1.21	0.59
SIMD	2.11	1.86	0.88	2.10	1.92	0.91	2.11	1.83	0.87
Skein	1.80	1.73	0.96	1.85	1.86	1.01	1.76	1.64	0.93
SHA-2	1.67	1.58	0.95	1.72	1.45	0.85	1.63	1.68	1.03

Table 4.11: Area (utilization of programmable logic blocks) of all SHA-3 candidates (256-bit variants) normalized to the area of SHA-256

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	4.96	4.87	4.27	2.17	2.16	2.00	2.04	2.96
BMW	12.07	13.45	10.16	12.00	12.02	12.99	13.12	12.24
CubeHash	1.81	1.81	1.69	1.87	1.88	1.99	2.01	1.86
ECHO	30.87	28.48	12.58	0.00	39.77	22.29	22.52	24.58
Fugue	4.26	4.44	2.21	5.85	5.87	3.70	3.73	4.11
Groestl	15.96	16.01	4.35	4.60	4.50	3.21	3.22	5.86
Hamsi	2.17	2.16	2.18	1.92	1.94	2.40	2.41	2.16
JH	4.84	4.78	2.94	4.37	4.31	3.18	3.24	3.88
Keccak	3.97	3.99	2.84	3.77	3.62	4.20	4.63	3.82
Luffa	3.28	3.29	2.67	2.74	2.77	3.40	3.43	3.07
Shabal	3.75	3.84	2.92	3.67	3.68	3.90	3.74	3.63
SHAvite-3	4.91	4.91	2.61	5.68	5.64	2.57	2.59	3.89
SIMD	20.97	19.99	21.45	18.53	18.57	23.03	23.24	20.39
Skein	3.41	3.45	3.03	3.28	3.34	3.68	3.74	3.41

Table 4.12: Throughput of all SHA-3 candidates (256-bit variants) normalized to the throughput of SHA-256

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	1.50	1.42	1.60	1.53	1.34	1.76	1.67	1.54
BMW	3.00	4.39	3.42	4.50	4.31	5.53	5.02	4.48
CubeHash	1.89	2.08	1.96	2.12	2.14	2.31	2.27	2.10
ECHO	4.14	5.21	8.19	0.00	6.02	5.00	5.57	5.56
Fugue	1.77	1.62	1.93	1.95	1.94	2.15	2.36	1.95
Groestl	3.60	3.97	5.32	3.68	3.62	4.24	3.93	4.02
Hamsi	1.35	1.49	1.62	1.82	1.96	1.66	1.88	1.67
JH	2.37	2.19	2.46	2.82	3.07	3.05	3.09	2.70
Keccak	6.10	6.37	6.63	8.56	7.91	7.23	8.01	7.21
Luffa	5.16	5.14	4.91	5.58	4.94	5.02	5.21	5.13
Shabal	0.89	1.62	1.61	1.63	1.41	1.73	1.55	1.46
SHAvite-3	1.64	1.46	1.77	1.51	1.58	1.89	2.11	1.70
SIMD	1.37	1.15	1.43	1.41	1.36	1.69	1.61	1.38
Skein	0.72	0.75	0.87	0.73	0.72	0.89	0.89	0.79

Table 4.13: Throughput to Area Ratio of all SHA-3 candidates normalized to the throughput to area ratio of SHA-256

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	0.30	0.29	0.37	0.71	0.62	0.88	0.82	0.52
BMW	0.25	0.33	0.34	0.38	0.36	0.43	0.38	0.37
CubeHash	1.04	1.15	1.16	1.13	1.14	1.16	1.13	1.13
ECHO	0.13	0.18	0.65	0.00	0.15	0.22	0.25	0.23
Fugue	0.42	0.36	0.88	0.33	0.33	0.58	0.63	0.47
Groestl	0.23	0.25	1.22	0.80	0.81	1.32	1.22	0.69
Hamsi	0.62	0.69	0.74	0.94	1.01	0.69	0.78	0.77
JH	0.49	0.46	0.84	0.65	0.71	0.96	0.95	0.70
Keccak	1.54	1.60	2.34	2.27	2.18	1.72	1.73	1.89
Luffa	1.57	1.56	1.84	2.04	1.78	1.48	1.52	1.67
Shabal	0.24	0.42	0.55	0.44	0.38	0.44	0.41	0.40
SHAvite-3	0.33	0.30	0.68	0.27	0.28	0.74	0.81	0.44
SIMD	0.07	0.06	0.07	0.08	0.07	0.07	0.07	0.07
Skein	0.21	0.22	0.29	0.22	0.21	0.24	0.24	0.23

Table 4.14: Area (utilization of programmable logic blocks) of all SHA-3 candidates (512-bit variants) normalized to the area of SHA-512

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	N/A	8.24	8.40	5.99	5.98	6.75	6.75	6.95
BMW	N/A	13.50	16.10	N/A	N/A	N/A	15.57	15.01
CubeHash	1.31	1.27	1.20	1.17	1.17	1.19	1.20	1.21
ECHO	19.56	18.43	9.22	0.00	23.89	12.26	12.40	15.15
Fugue	2.22	2.16	1.43	2.54	2.52	1.72	1.71	2.00
Groestl	14.35	18.57	5.37	5.08	5.16	3.71	3.88	6.59
Hamsi	3.19	3.10	3.41	2.61	2.61	3.50	3.50	3.11
JH	3.03	2.90	2.04	2.65	2.69	2.05	2.07	2.46
Keccak	2.27	2.41	2.19	1.81	1.81	2.19	2.21	2.12
Luffa	3.92	3.82	3.35	3.60	3.63	4.28	4.25	3.82
Shabal	2.23	2.51	2.10	2.04	2.05	2.19	2.11	2.17
SHAvite-3	5.85	6.09	3.02	7.01	7.01	3.36	3.46	4.83
SIMD	N/A	28.29	26.34	22.09	22.13	29.15	29.43	26.05
Skein	3.79	3.87	3.28	3.50	3.49	3.90	3.95	3.67

Table 4.15: Throughput of all SHA-3 candidates (512-bit variants) normalized to the throughput of SHA-512

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	N/A	0.86	0.73	1.07	1.08	1.01	0.96	0.94
BMW	N/A	2.90	3.17	N/A	N/A	N/A	2.57	2.87
CubeHash	1.32	1.38	1.24	1.55	1.46	1.16	1.16	1.32
ECHO	2.47	2.95	2.43	0.00	3.12	2.46	2.74	2.68
Fugue	0.45	0.46	0.41	0.59	0.56	0.51	0.56	0.50
Groestl	2.05	3.36	2.33	3.71	3.31	2.96	2.98	2.90
Hamsi	0.65	0.79	0.67	0.93	0.87	0.61	0.65	0.73
JH	1.42	1.59	1.15	2.09	2.07	1.69	1.72	1.65
Keccak	1.69	1.53	1.38	3.28	2.90	2.22	2.18	2.07
Luffa	2.63	3.16	2.58	3.88	3.87	2.75	2.89	3.07
Shabal	0.57	0.64	0.60	1.06	0.99	0.84	0.77	0.76
SHAvite-3	1.19	1.36	1.41	1.32	1.30	1.13	1.30	1.28
SIMD	N/A	1.52	1.52	1.93	1.90	1.64	1.66	1.69
Skein	0.83	0.86	0.57	0.99	0.97	0.99	0.98	0.87

Table 4.16: Throughput to Area Ratio of all SHA-3 candidates normalized to the throughput to area ratio of SHA-512

Candidate	Spartan 3	Virtex 4	Virtex 5	Cyclone II	Cyclone III	Stratix II	Stratix III	Overall
BLAKE	N/A	0.10	0.09	0.18	0.18	0.15	0.14	0.14
BMW	N/A	0.21	0.20	N/A	N/A	N/A	0.16	0.19
CubeHash	1.01	1.08	1.03	1.33	1.25	0.98	0.97	1.09
ECHO	0.13	0.16	0.26	0.00	0.13	0.20	0.22	0.18
Fugue	0.20	0.21	0.28	0.23	0.22	0.30	0.32	0.25
Groestl	0.14	0.18	0.43	0.73	0.64	0.80	0.77	0.44
Hamsi	0.20	0.25	0.20	0.36	0.34	0.17	0.19	0.24
JH	0.47	0.55	0.56	0.79	0.77	0.82	0.83	0.67
Keccak	0.75	0.63	0.63	1.81	1.60	1.02	0.99	0.98
Luffa	0.67	0.83	0.77	1.08	1.07	0.64	0.68	0.80
Shabal	0.25	0.26	0.29	0.52	0.48	0.39	0.37	0.35
SHAvite-3	0.20	0.22	0.46	0.19	0.19	0.34	0.38	0.27
SIMD	N/A	0.05	0.06	0.09	0.09	0.06	0.06	0.06
Skein	0.22	0.22	0.17	0.28	0.28	0.26	0.25	0.24

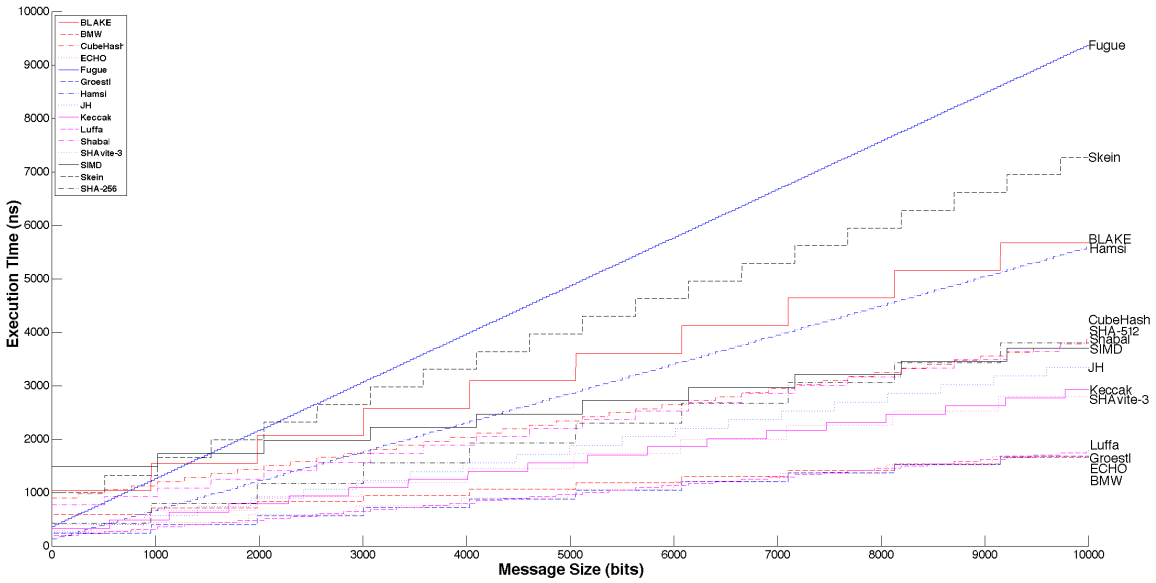


Figure 4.4: Execution time vs. message size for short messages up top 10,000 bits. 512-bit variants of all SHA-3 Candidates and SHA-512 in Virtex 5.

graphs showing the dependence between the normalized area and the normalized throughput for Xilinx Virtex 5 and Altera Stratix III families, respectively. Both graphs concern 256-bit variants of all algorithms. In these diagrams, the higher the gradient, the better the function in terms of the throughput to area ratio. The dotted line represents a normalized line drawn based on the reference implementation of SHA-256. Any algorithms that reside on the left side of the line perform better than SHA-256 in terms of the throughput to area ratio. We can categorize the results into 3 groups as:

- Group 1: Better than SHA-256 (CubeHash, Groestl, Keccak and Luffa)
- Group 2: Worse than SHA-256 with small area (BLAKE, Fugue, Hamsi, JH, Shabal, SHAvite-3 and Skein)
- Group 3: Worse than SHA-256 with large area (BMW, ECHO and SIMD)

The similar graphs calculated for the 512-bit variants of all algorithms are presented in Figures 4.7 and 4.8. The criterion used for the classification of the 256-bit variants of the SHA-3 candidates cannot be applied towards the 512-bit variants. With the exception

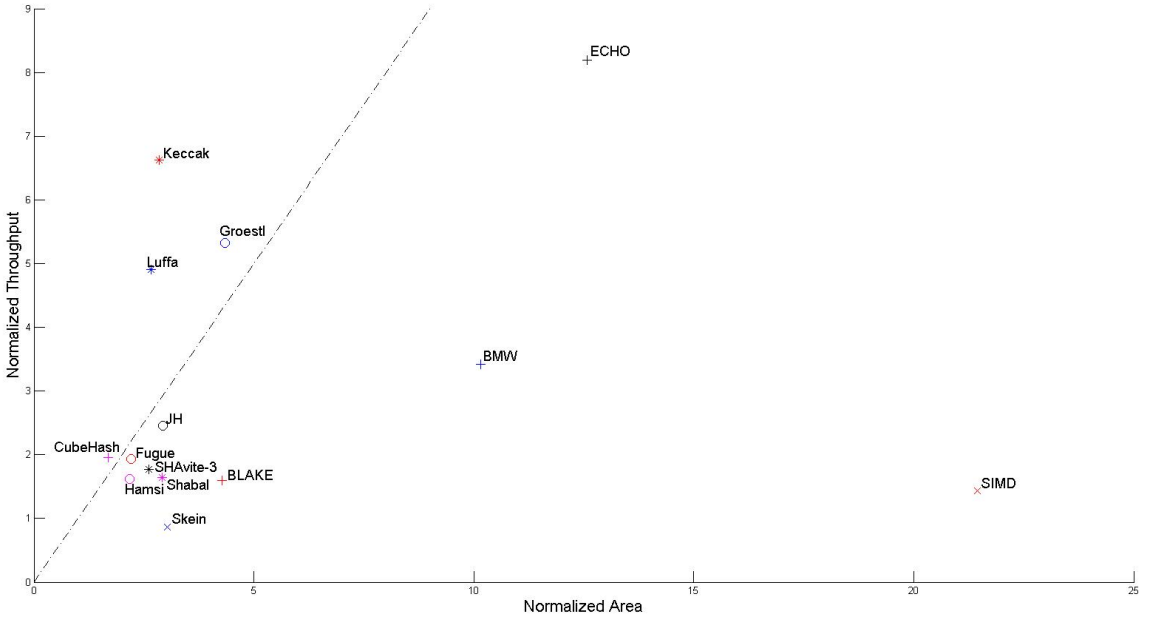


Figure 4.5: Relative performance of all Round 2 SHA-3 Candidates (256-bit variants) in terms of the normalized throughput and the normalized area in Xilinx Virtex 5 (with SHA-256 used as a reference point).)

of CubeHash, for which the performance point barely touches the normalized line in case of both presented families, other candidates fail to meet the same level of performance compared to SHA-512. Additionally, by comparing diagrams for Virtex 5 and Stratix III, it can be seen that the algorithms' performance vary depending on the FPGA family.

Additionally, the overall normalized graphs for 256 and 512-bit variants are plotted in Figures 4.9 and 4.10. The same criterion as used earlier for 256-bit variants of SHA-3 candidates in case of Virtex 5 and Stratix III, can be applied to the case of overall normalized results. In case of 512-bit variants, two groups can be formed based on the area requirements of candidates, with BMW, ECHO and SIMD demonstrating prohibitively large area requirements.

Finally, we present a combined graph representing an overall normalized throughput vs. overall normalized area of 256 and 512-bit variants normalized to SHA-256 in Figure 4.11.

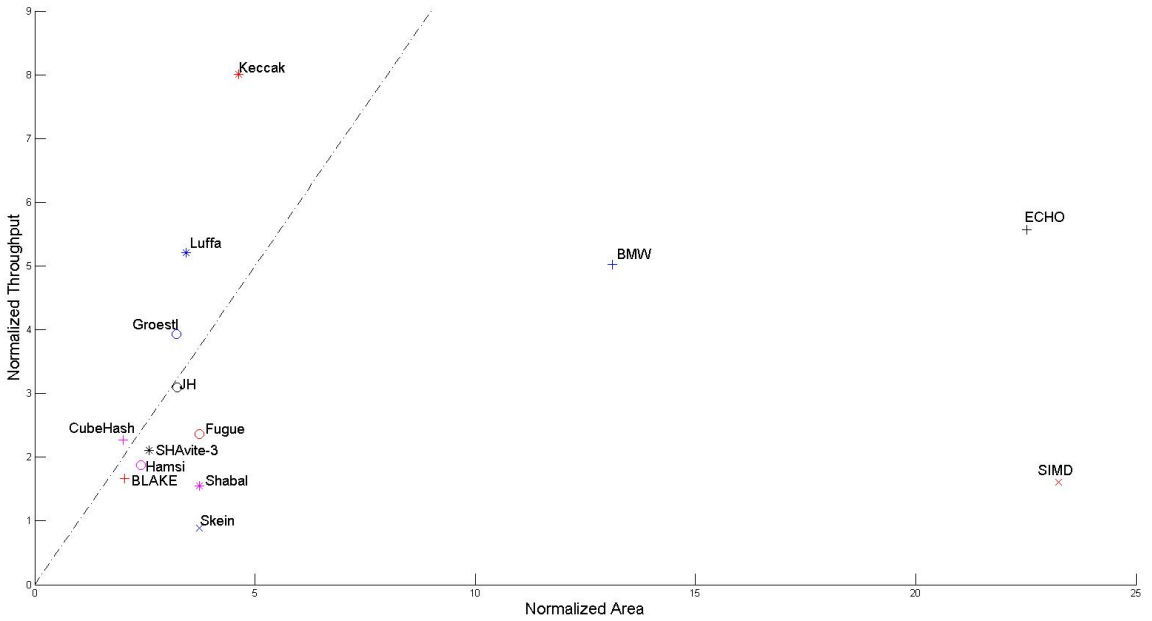


Figure 4.6: Relative performance of all Round 2 SHA-3 Candidates (256-bit variants) in terms of the normalized throughput and the normalized area in Stratix III (with SHA-256 used as a reference point).)

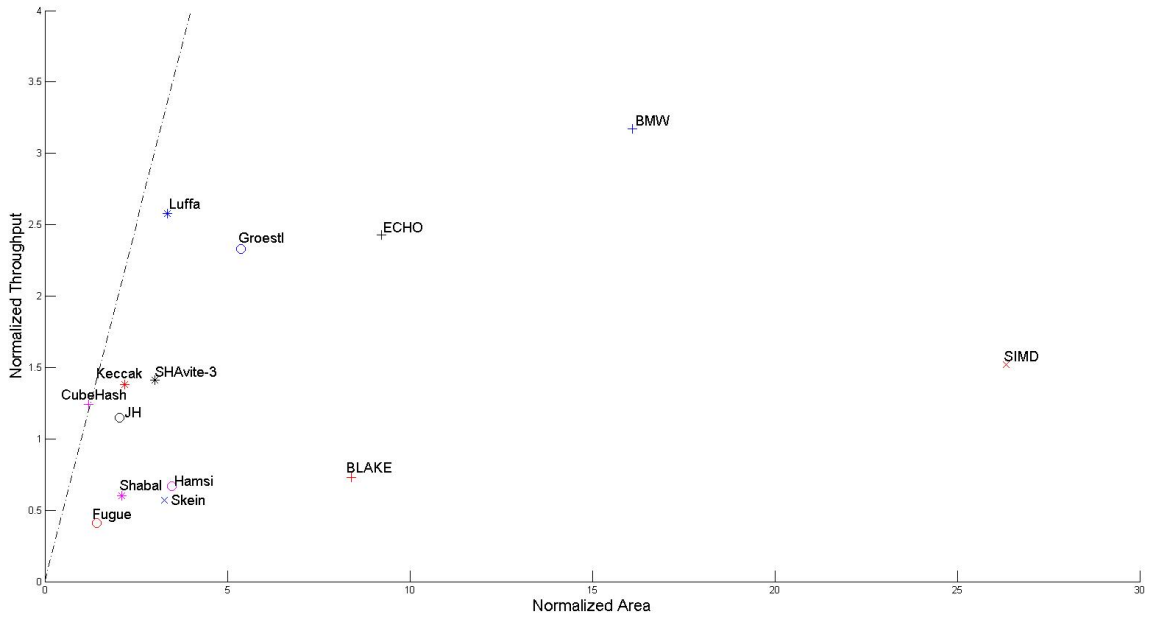


Figure 4.7: Relative performance of all Round 2 SHA-3 Candidates (512-bit variants) in terms of the normalized throughput and the normalized area in Virtex 5 (with SHA-512 used as a reference point).)

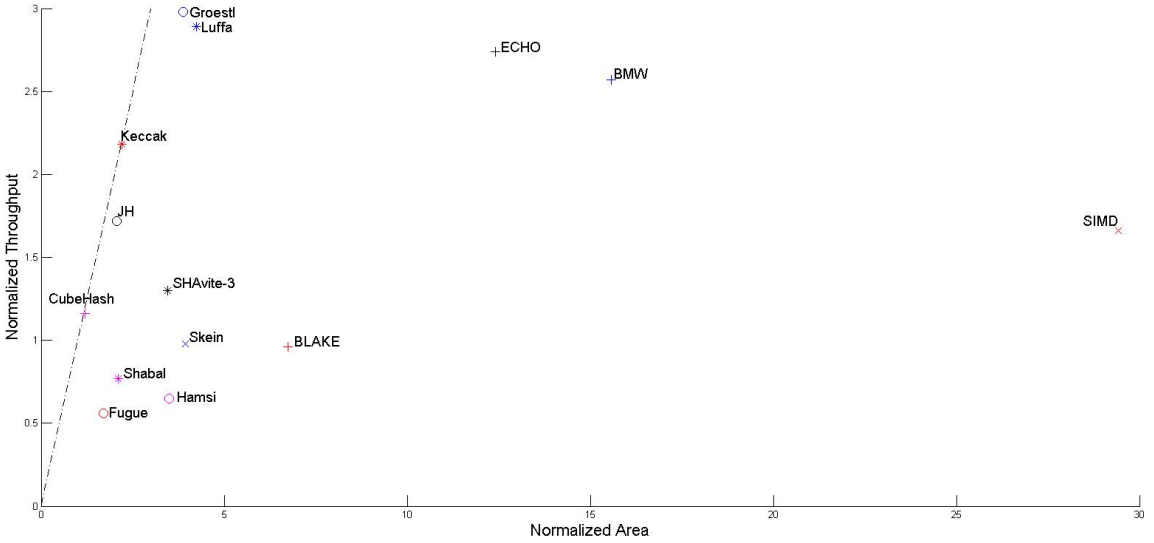


Figure 4.8: Relative performance of all Round 2 SHA-3 Candidates (512-bit variants) in terms of the normalized throughput and the normalized area in Stratix III (with SHA-512 used as a reference point).)

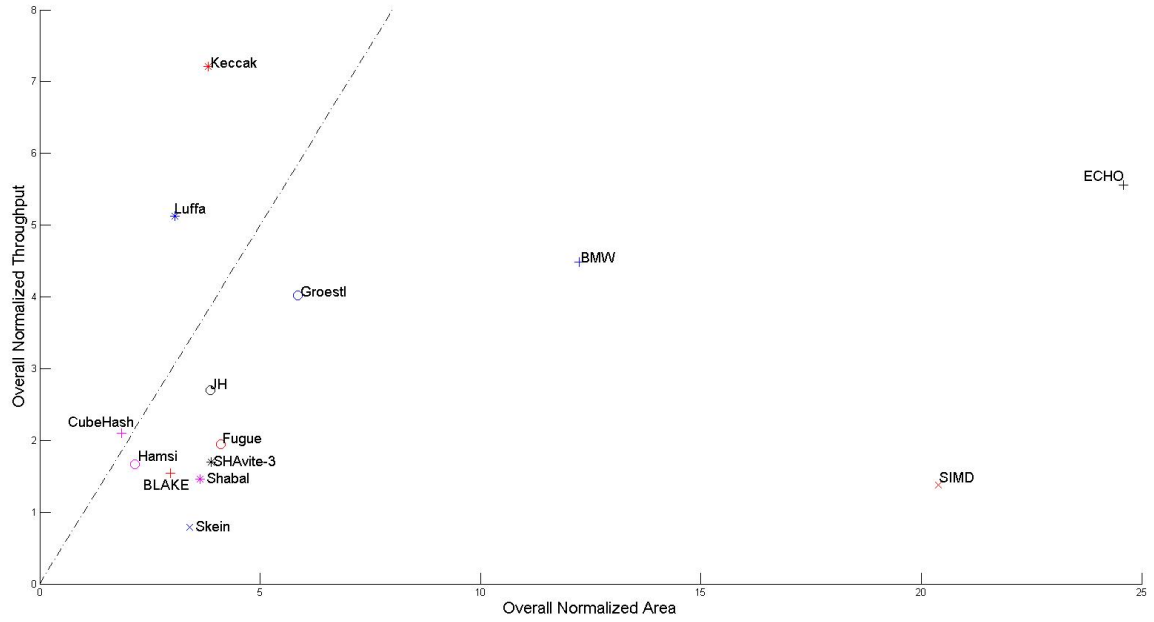


Figure 4.9: Relative performance of all Round 2 SHA-3 Candidates (256-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-256 used as a reference point).)

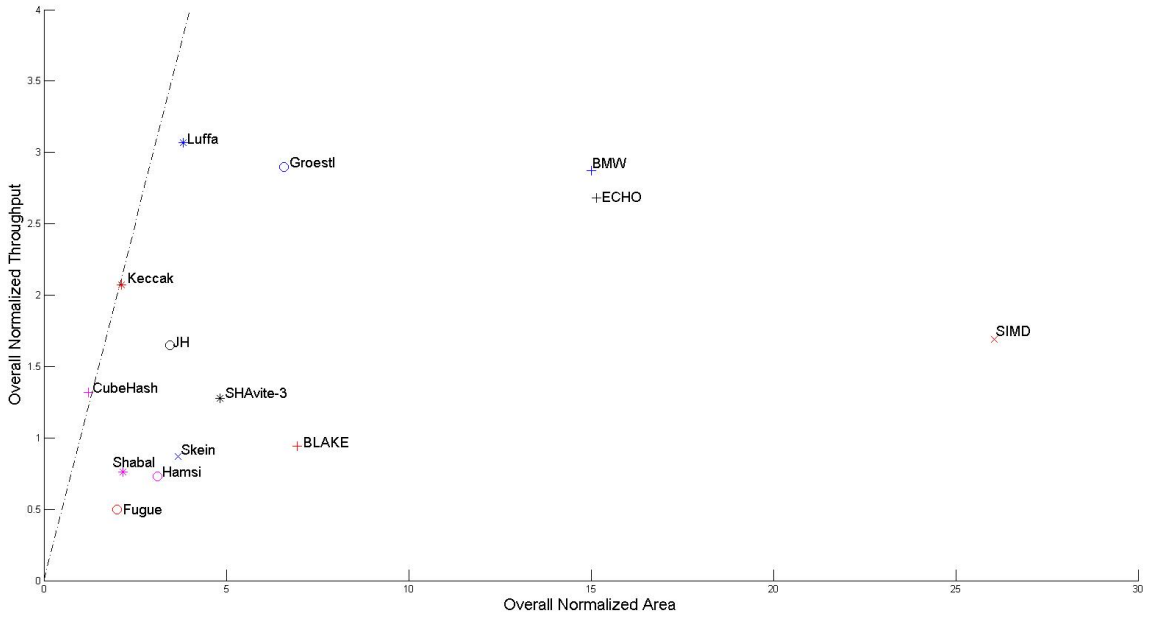


Figure 4.10: Relative performance of all Round 2 SHA-3 Candidates (512-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-512 used as a reference point).)

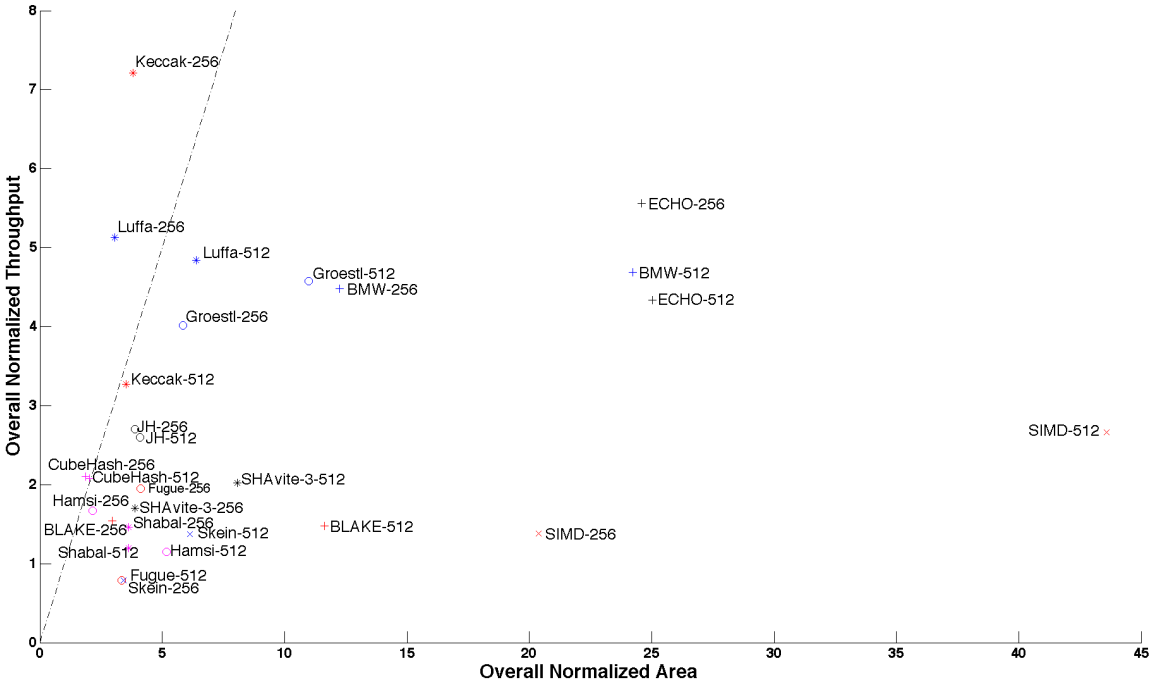


Figure 4.11: Relative performance of all Round 2 SHA-3 Candidates (256 and 512-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-256 used as a reference point).

Chapter 5: Results from Other Groups

5.1 Best Results from Other Groups

Table 5.1 presents the best published results in terms of the throughput to area ratio for 256-bit variants of the SHA-3 Round 2 Candidates, and contrasts them with the best results reported in this thesis. The implementation platform is Xilinx Virtex 5 family. This platform has been selected because majority of papers from other groups target this particular family. The corresponding throughput vs. area graph is also shown in Figure 5.1.

In general, due to our selection of the interface/protocol, the controller associated with all our designs cost us between 80 and 150 slices. This overhead mainly originates from the counter required to store the message length, communication module residing between FSM1, 2 and 3, and some additional control logic. As a result, small designs such as CubeHash may be at a disadvantage in terms of the throughput to area ratio compared to the designs from other groups, following different interfaces. However, with the exception of Shabal, most of our designs perform comparatively close (within 25%), if not better than the best designs reported in the literature to date. Selected algorithms that underperform are discussed below:

- *BLAKE*: The design by Aumasson et al. [23] is much smaller than our design. This may be due to our inefficient implementation of the Permute unit (Figure 3.6). Additionally, the removal of the temporary message block register and its corresponding multiplexer should be able to further reduce our resource utilization.
- *CubeHash*: Interface overhead causes our throughput to area ratio to drop.
- *Groestl*: Interface overhead causes our throughput to area ratio to drop. Also, we may yet overlook a possible way to reduce resource utilization.

- *Shabal*: Detrey et al. [40] utilizes a shift register mode (SRL16) of Xilinx Multipurpose Look-Up Tables.

- *Skein*: The differences may have been caused by an inefficient implementation of the proposed design.

Table 5.1: Comparison of the best designs from other groups in terms of the Throughput to Area Ratio with designs presented in this paper. All designs concern 256-bit variants of the SHA-3 candidates.

	Other Groups				**This Paper**		
	Area (CLB slices)	**Thr (Mbit/s)**	**Thr/Area**	**Source**	**Area (CLB slices)**	**Thr (Mbit/s)**	**Thr/Area**
BLAKE	1217	2438	2.00	Aumasson et al. [23]	1851	2610.6	1.41
CubeHash	590	2960	5.02	Kobayashi et al. [7]	730	3189.8	4.37
ECHO	9333	14860	1.59	Lu et al. [15]	5445	13360.5	2.45
Groestl	1722	10276	5.97	Gauvaram et al. [28]	1884	8676.5	4.61
Hamsi	718	1680	2.34	Kobayashi et al. [7]	946	2646.2	2.80
Keccak	1412	6900	4.89	Bertoni et al. [31]	1229	10806.5	8.79
Luffa	1048	6343	6.05	Kobayashi et al. [7]	1154	8008.0	6.94
Shabal	153	2051	13.41	Detrey et al. [40]	1266	2624.0	2.07
Skein	937	1751	1.87	Tillich [11]	1312	1416.1	1.08

5.2 Best Results

The best results (including our results and results from other groups) in terms of the throughput to area ratio for 256-bit variants of all SHA-3 Round 2 candidates in Xilinx Virtex 5 are summarized in Table 5.2. A corresponding throughput vs. area diagram is shown in Figure 5.2.

There is no significant change in an overall ranking of SHA-3 Round 2 candidates in terms of the throughput to area ratio compared to the ranking based exclusively on our own results, with the exception of Shabal, which jumps to the lead. Additionally, Shabal will most likely retains its lead in comparison of 512-bit variants as there is no functional change between 256 and 512-bit variants of this algorithm.

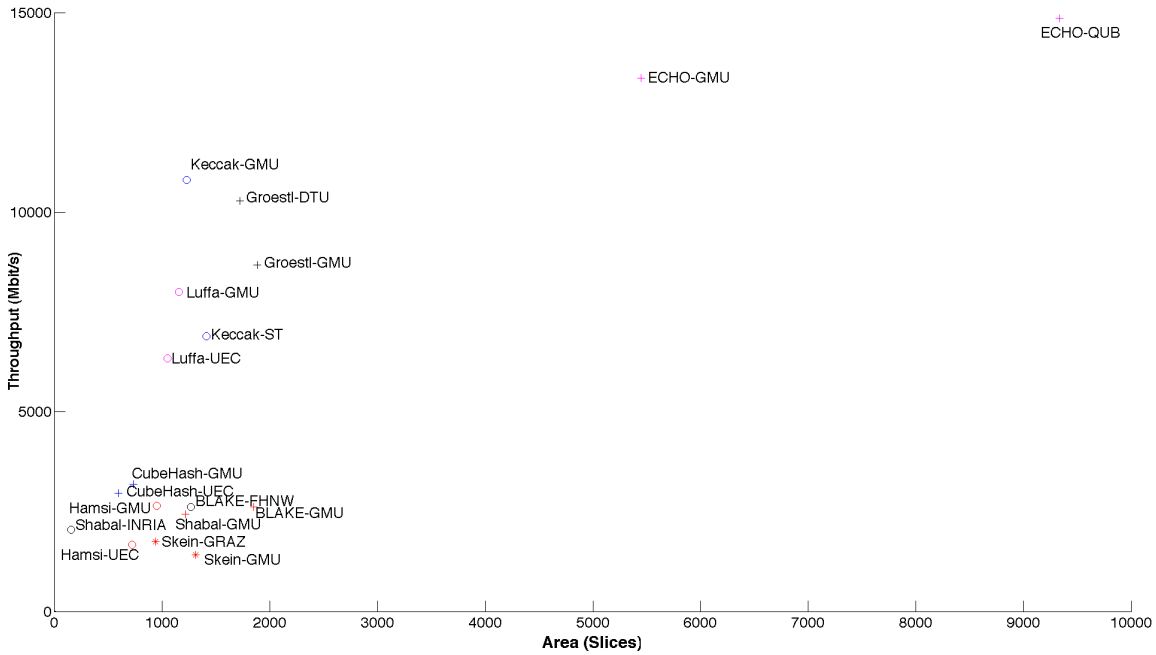


Figure 5.1: Best *published* results vs. GMU results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput to area ratio in Xilinx Virtex 5

Table 5.2: Best results in terms of the Throughput to Area Ratio for 256-bit variants of all SHA-3 Round 2 candidates in Xilinx Virtex 5

	Area (CLB slices)	**Thr (Mbit/s)**	**Thr/Area**	**Source**
BLAKE	1217	2438.0	2.00	Aumasson et al. [23]
BMW	4400	5576.7	1.27	GMU
CubeHash	590	2960.0	5.02	Kobayashi et al. [7]
ECHO	5445	13360.5	2.45	GMU
Fugue	956	3151.2	3.30	GMU
Groestl	1722	10276.0	5.97	Gauvaram et al. [28]
Hamsi	946	2646.2	2.797	GMU
JH	1275	4013.5	3.15	GMU
Keccak	1229	10806.5	8.793	GMU
Luffa	1154	8008.0	6.939	GMU
Shabal	153	2051.0	13.41	Detrey et al. [40]
SHAvite-3	1130	2886.9	2.55	GMU
SIMD	9288	2325.9	0.25	GMU
Skein	937	1751.0	1.87	Tillich [11]

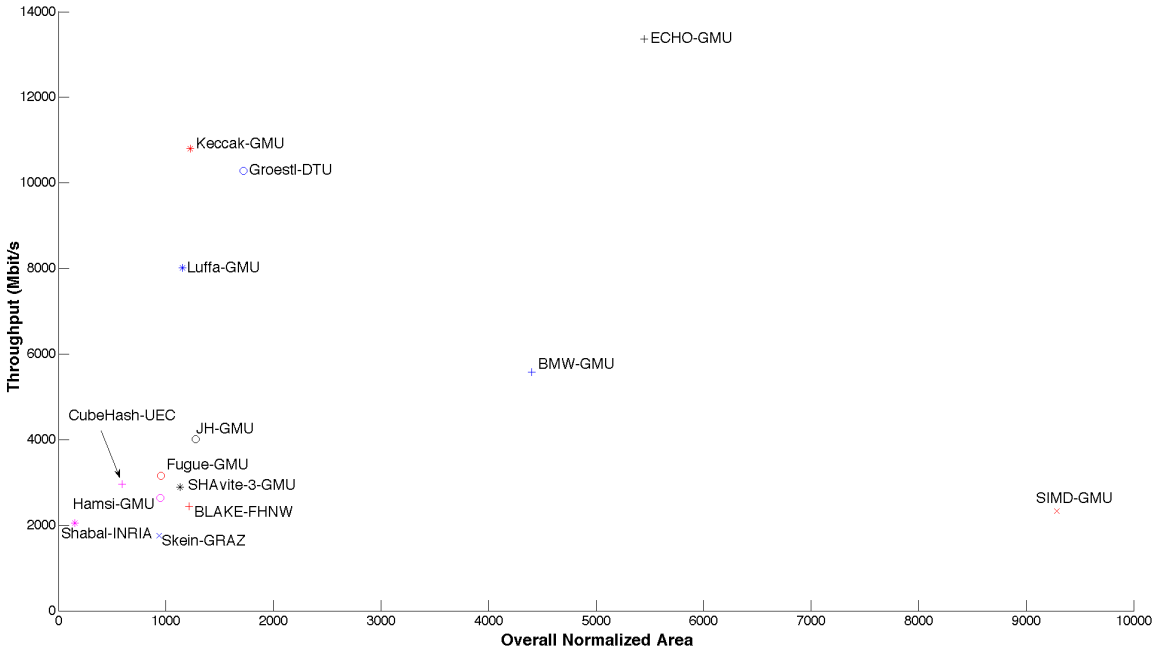


Figure 5.2: Best results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput vs. area in Virtex 5

Chapter 6: Conclusions and Future Works

Based on our study, only one candidate, namely CubeHash, consistently outperforms SHA-2 in terms of the Maximum Throughput to Area Ratio in the Overall category across 7 FPGA families for both 256 and 512-bit outputs. Keccak and Luffa clearly outperform SHA-2 when 256-bit variants of all functions are taken into account, but fell short in case of 512-bit variants.

While we can base our decision on choosing the best candidates to advance to the next round on this study alone, some caution must be exercised. This is because there are at least two more implementation targets that can be investigated: Maximum Throughput and Minimum Area. In essence, an algorithm that performs extremely well for one optimization target may fail to produce the same results in other studies. For instance, Keccak outperforms other candidates in terms of the Throughput to Area Ratio and in terms of the Maximum Throughput. However, if we were to focus on area, Keccak would not likely rank equally high. This is due to its extensive use of multiple layers of permutations, which make it hard to apply folding techniques leading to a compact architecture.

However, one can at least draw some conclusions regarding which candidates should be dismissed in the next round. The three candidates that require large area, and as a result demonstrate an extremely bad throughput to area ratio, namely BMW, ECHO and SIMD, can be easily eliminated. This is due to the fact that even if we applied folding techniques to their designs, it would be most likely infeasible to produce a reasonable throughput for their compact (folded) architectures. In fact, a compact design of ECHO has been developed recently by [41] with the throughput of 72 Mbps in Virtex 5. The throughput of this design is approximately three times smaller than the throughput of the compact design for Blake [20]. This comparison is performed under the assumption that the increased usage of Block RAMs in compact Blake is offset by the increased use of CLB slices in

compact ECHO. On top of that, it is indisputable that compact ECHO has almost thirty times smaller throughput with almost the same slice count as the best design for Shabal from [40]. Needless to say, very area consuming hash functions, such as ECHO should be eliminated assuming that all candidates guarantee the same security level.

Comparing our results with results from different groups also proves that our designs are comparable to others with the exception of Shabal which overtakes other candidates in terms of the throughput to area ratio based on the study from Detrey et al. [40]. In fact, Shabal's extraordinary performance in that study will most likely allow this algorithm to pass to the next round without much of a problem, unless, of course, a security flaw is found. While this is very promising for Shabal, its victory as a next SHA-3 standard is not yet assured in light of its ability to optimize for high-speed design. In fact, due to its highly sequential nature, it is unlikely that the algorithm will be able to perform faster than reported in the study by Feron and Francq [19]. Moreover, Shabal is quite slow in hashing messages smaller than 10,000 bits compared to other SHA-3 candidates. As shown in Figures 4.1, 4.2, 4.1 and 4.4, Shabal consistently underperformed when a message was under 1000 bits and only performed adequately as compared to SHA-2 for messages in the range of 10,000 bits.

Even though this thesis presents one of the most comprehensive studies of all SHA-3 candidates, it is still far from complete. As noted earlier, other optimization targets could be investigated. However, rather than using these two straightforward targets, it may be better to investigate the algorithms' performance and flexibility using folding and unrolling techniques. This is because optimizing for the Maximum Throughput will most likely lead to fully unrolled designs, and optimizing for Minimum Area will likely lead to the microcontroller-like designs. Both of these design types are impractical as the speed-optimized design will be extremely large and the area-optimized design will be extremely slow. Assuming that we provide maximum acceptable area and minimum acceptable speed for each optimization criterion, how do we know that we are not unintentionally giving an advantage to one or more algorithms that easily meet these specific requirements?

Rather than optimizing exclusively for Maximum Throughput or Minimum Area, it is more practical to study a range of designs using our basic architectures presented in this paper as a base. In general, this should include at least two and four times unrolled and folded architectures (where applicable). This way, each algorithm would be represented not by one, but by at least 5 different points. This study would also evaluate the flexibility of all algorithms in terms of trading area for speed and speed for area. Indeed, maybe the ability for an algorithm to adjust according to the need of a user is more important than a specific single performance point in terms of area or speed. As such, our future project will include investigation of architectures based on folding and unrolling, using our current designs as a starting point for investigation.

A uniform padding circuit will also be developed and its cost evaluated. Additionally, an investigation on the cost of the interface and its associated protocol will be performed if the time permits. Furthermore, an analysis of power and energy will be performed for currently implemented and future designs. Results across a broader range of FPGA families will be gathered, including Spartan 6 and Virtex 6 from Xilinx, and Cyclone IV, Stratix IV and Arria II from Altera. An influence of synthesis tools and high level description languages will be investigated. A similar methodology will be applied toward designs implemented using standard-cell ASIC technology. Additionally, candidates' capability of using dedicated FPGA resources, such as dedicated multipliers and DSP units, will be analyzed. The performance of these candidates for other modes of operation will also be studied.

The information obtained from all these studies will allow us to compile a guideline for developing hardware implementations targeting various hardware performance measures, such as Maximum Throughput to Area Ratio, Maximum Throughput, Minimum Area, etc. This will enable us to create a software program that can estimate the cost and performance of a cryptographic core, or any hardware design for that matter, without the need to implement this core in a hardware description language at the Register-Transfer Level. Instead, a high level description, such as data flow diagram or a pseudocode, will be used as an input sufficient to generate approximate results. While it is unlikely that such

program would be able to evaluate an algorithm as well as an experienced human designer, it should have sufficient accuracy to serve as a guide for developing future cryptographic algorithms.

Bibliography

Bibliography

[1] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *CRYPTO*, ser. LNCS, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 17–36. [Online]. Available: http://dx.doi.org/10.1007/11535218_2

[2] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays," *LNCS 2020, Progress in Cryptology - CT-RSA 2001, Ed. D. Naccache, RSA Conference 2001 - Cryptographers' Track*, pp. 84–99, Apr. 2001.

[3] S.-. Zoo, "Sha-3 hardware implementations," http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations.

[4] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, "High-speed hardware implementations of blake, blue midnight wish, cubehash, echo, fugue, grøstl, hamsi, jh, keccak, luffa, shabal, shavite-3, simd, and skein," Cryptology ePrint Archive, Report 2009/510, 2009, http://eprint.iacr.org/.

[5] K. Gaj, E. Homsirikamol, and M. Rogawski, "Comprehensive comparison of hardware performance of fourteen round 2 sha-3 candidates with 512-bit outputs using field programmable gate arrays," SHA-3 Workshop, Santa Barbara, Aug, 2010, submission.

[6] ——, "Fair and comprehensive methodology for comparing hardware performance of fourteen round two sha-3 candidates using fpgas," Proc. Cryptographic hardware and Embedded Systems Workshop, CHES 2010, Santa Barbara, Aug, 2010, in print.

[7] K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta, "Evaluation of hardware performance for the sha-3 candidates using sasebo-gii," Cryptology ePrint Archive, Report 2010/010, 2010, http://eprint.iacr.org/.

[8] J. A. Bernhard Jungk, Steffen Reith, "On optimized fpga implementations of the sha-3 candidate groestl," Cryptology ePrint Archive, Report 2009/206, 2009, http://eprint.iacr.org/.

[9] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan, and W. P. Marnane, "Fpga implementations of sha-3 candidates:cubehash, grøstl, lane, shabal and spectral hash," Cryptology ePrint Archive, Report 2009/342, 2009, http://eprint.iacr.org/.

[10] S. Tillich, M. Feldhofer, W. Issovits, T. Kern, H. Kureck, M. Mühlberghuber, G. Neubauer, A. Reiter, A. Köfler, and M. Mayrhofer, "Compact hardware implementations of the sha-3 candidates arirang, blake, grøstl, and skein," Cryptology ePrint Archive, Report 2009/349, 2009, http://eprint.iacr.org/.

[11] S. Tillich, "Hardware implementation of the sha-3 candidate skein," Cryptology ePrint Archive, Report 2009/159, 2009, http://eprint.iacr.org/.

[12] M. Bernet, L. Henzen, H. Kaeslin, N. Felber, and W. Fichtner, "Hardware implementations of the SHA-3 candidates shabal and cubehash," *Circuits and Systems*, pp. 515–518, 2009, 52nd IEEE International Midwest Symposium on 2-5 Aug.

[13] J. Fan, "Hardware evaluation of the hash function hamsi," Hamsi's website, 2009, http://www.cosic.esat.kuleuven.be/publications/article-1322.pdf.

[14] M. Long, "Implementing skein hash function on xilinx virtex-5 fpga platform," Online, 2010, http://www.skein-hash.info/sites/default/files/.

[15] L. Lu, M. O'Neill, and E. Swartzlander, "Hardware evaluation of sha-3 hash function candidate echo," Online, 2009, http://www.ucc.ie/en/crypto/CodingandCryptographyWorkshop/TheClaudeShannonWorkshoponCodingCryptography2009/DocumentFile,75649,en.pdf.

[16] I. V. Miroslav Kneevic, "Hardware evaluation of the luffa hash family," COSIC Publication, Online, 2009, http://www.cosic.esat.kuleuven.be/publications/article-1282.pdf.

[17] A. Namin and M. Hasan, "Hardware implementation of the compression function for selected SHA-3 candidates," in *CACR 2009-28*, July 2009, p. 29.

[18] J. Strømbergson, "Implementation of the keccak hash function in fpga devices," Online, 2008, http://www.strombergson.com/files/.

[19] R. Feron and J. Francq, "Fpga implementation of shabal: Our first results," Online, 2010. [Online]. Available: http://ehash.iaik.tugraz.at/uploads/d/d4/FPGA_Implementation_of_Shabal_-_First_Results.pdf

[20] 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, http://eprint.iacr.org/.

[21] G. CERG, "Hardware interface of a secure hash algorithm sha v1.4," Online, 2010, http://cryptography.gmu.edu/athena/interfaces/.

[22] ——, "Athena project website," Online, 2010, http://cryptography.gmu.edu/athena/.

[23] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "Sha-3 proposal blake," Submission to NIST, 2008. [Online]. Available: http://131002.net/blake/blake.pdf

[24] D. Gligoroski, V. Klima, S. J. Knapskog, M. El-Hadedy, J. Amundsen, and S. F. Mjolsnes, "Cryptographic hash function blue midnight wish," Submission to NIST (Round 2), 2009. [Online]. Available: http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf

[25] D. J. Bernstein, "Cubehash specification (2.b.1)," Submission to NIST (Round 2), 2009. [Online]. Available: http://cubehash.cr.yp.to/submission2/spec.pdf

[26] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin, "Sha-3 proposal: Echo," Submission to NIST (updated), 2009. [Online]. Available: http://crypto.rd.francetelecom.com/echo/doc/echo_description_1-5.pdf

[27] S. Halevi, W. E. Hall, and C. S. Jutla, "The hash function fugue," Submission to NIST (updated), 2009. [Online]. Available: http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html/{$}FILE/fugue_09.pdf

[28] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer, and S. S. Thomsen, "Grstl – a sha-3 candidate," Submission to NIST, 2008. [Online]. Available: http://www.groestl.info/Groestl.pdf

[29] zgl Kck, "The hash function hamsi," Submission to NIST (updated), 2009. [Online]. Available: http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf

[30] H. Wu, "The hash function jh," Submission to NIST (updated), 2009. [Online]. Available: http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf

[31] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Keccak specifications," Submission to NIST (Round 2), 2009. [Online]. Available: http://keccak.noekeon.org/Keccak-specifications-2.pdf

[32] C. D. Canniere, H. Sato, and D. Watanabe, "Hash function luffa: Specification," Submission to NIST (Round 2), 2009. [Online]. Available: http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_Specification_20091002.pdf

[33] N. I. of Standards and T. (NIST), "Secure hash standard (SHS) FIPS publication 180-3," Oct 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[34] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau, "Shabal, a submission to nists cryptographic hash algorithm competition," Submission to NIST, 2008. [Online]. Available: http://ehash.iaik.tugraz.at/uploads/6/6c/Shabal.pdf

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

[36] G. Leurent, C. Bouillaguet, and P.-A. Fouque, "Simd is a message digest," Submission to NIST (Round 2), 2009. [Online]. Available: http://www.di.ens.fr/~leurent/files/SIMD.pdf

[37] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family," Submission to NIST (Round 2), 2009. [Online]. Available: http://www.skein-hash.info/sites/default/files/skein1.2.pdf

[38] J. Daemen and V. Rijmen, "AES proposal: Rijndael," in *First Advanced Encryption Standard AES Conference*, Ventura, California, USA, 1998, updated version from 1999.

[39] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Improving sha-2 hardware implementations," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, Oct 2006, pp. 298–310.

[40] J. Detrey, P. Gaudry, and K. Khalfallah, "A low-area yet performant fpga implementation of shabal," Cryptology ePrint Archive, Report 2010/292, 2010, http://eprint.iacr.org/.

[41] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "A compact fpga implementation of the sha-3 candidate echo," Cryptology ePrint Archive, Report 2010/364, 2010, http://eprint.iacr.org/.

Curriculum Vitae

Ekawat Homsirikamol was born in December of 1986 in Bangkok, Thailand. He received his Bachelor of Science in Electrical Engineering from the Volgenau School of Information Technology and Engineering of George Mason University, Fairfax, USA in Spring 2008. He continued his studies in the same school directly afterwards. During the course of his studies, he was involved in teaching various undergraduate courses at George Mason University both as a Teaching Assistant and as a Lab Instructor. He also worked as a Research Assistant in the Cryptographic Engineering Research Group (CERG) with funding from the National Institute of Standards and Technology (NIST). His research interest include but are not limited to efficient hardware implementations of cryptographic algorithms and embedded hardware design.