Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs

Ekawat Homsirikamol Marcin Rogawski Kris Gaj

George Mason University

ehomsiri, mrogawsk, kgaj@gmu.edu

This work has been supported in part by NIST through the Recovery Act Measurement Science and Engineering Research Grant Program, under contract no. 60NANB10D004.

Contents

Abstract

Performance in hardware has been demonstrated to be an important factor in the evaluation of candidates for cryptographic standards. Up to now, no consensus exists on how such an evaluation should be performed in order to make it fair, transparent, practical, and acceptable for the majority of the cryptographic community. In this report, we formulate a proposal for a fair and comprehensive evaluation methodology, and apply it to the comparison of hardware performance of 14 Round 2 SHA-3 candidates. The most important aspects of our methodology include the definition of clear performance metrics, the development of a uniform and practical interface, generation of multiple sets of results for several representative FPGA families from two major vendors, and the application of a simple procedure to convert multiple sets of results into a single ranking. The VHDL codes for 256 and 512-bit variants of all 14 SHA-3 Round 2 candidates and the old standard SHA-2 have been developed and thoroughly verified. These codes have been then used to evaluate the relative performance of all aforementioned algorithms using seven modern families of Field Programmable Gate Arrays (FPGAs) from two major vendors, Xilinx and Altera. All algorithms have been evaluated using four performance measures: the throughput to area ratio, throughput, area, and the execution time for short messages. Based on these results, the 14 Round 2 SHA-3 candidates have been divided into several groups depending on their overall performance in FPGAs.

Chapter 1

Introduction and Motivation

Starting from the Advanced Encryption Standard (AES) contest organized by NIST in 1997-2000 [1], open contests have become a method of choice for selecting cryptographic standards in the U.S. and over the world. The AES contest in the U.S. was followed by the NESSIE competition in Europe [2], CRYPTREC in Japan, and eSTREAM in Europe [3].

Four typical criteria taken into account in the evaluation of candidates are: security, performance in software, performance in hardware, and flexibility. While security is commonly recognized as the most important evaluation criterion, it is also a measure that is most difficult to evaluate and quantify, especially during a relatively short period of time reserved for the majority of contests. A typical outcome is that, after eliminating a fraction of candidates based on security flaws, a significant number of remaining candidates fail to demonstrate any easy to identify security weaknesses, and as a result are judged to have adequate security.

Performance in software and hardware are next in line to clearly differentiate among the candidates for a cryptographic standard. Interestingly, the differences among the cryptographic algorithms in terms of hardware performance seem to be particularly large, and often serve as a tiebreaker when other criteria fail to identify a clear winner. For example, in the AES contest, the difference in hardware speed between the two fastest final candidates (Serpent and Rijndael) and the slowest one (Mars) was by a factor of seven [1][4]; in the eSTREAM competition the spread of results among the eight top candidates qualified to the final round was by a factor of 500 in terms of speed (Trivium x64 vs. Pomaranch), and by a factor of 30 in terms of area (Grain v1 vs. Edon80) [5][6].

At this point, the focus of the attention of the entire cryptographic community is on the SHA-3 contest for a new hash function standard, organized by NIST [7][8]. The contest is now in its second round, with 14 candidates remaining in the competition. The evaluation is scheduled to continue until the second quarter of 2012.

In spite of the progress made during previous competitions, no clear and commonly

accepted methodology exists for comparing hardware performance of cryptographic algorithms [9]. The majority of the reported evaluations have been performed on an ad-hoc basis, and focused on one particular technology and one particular family of hardware devices. Other pitfalls included the lack of a uniform interface, performance metrics, and optimization criteria. These pitfalls are compounded by different skills of designers, using two different hardware description languages, and no clear way of compressing multiple results to a single ranking. In this paper, we address all the aforementioned issues, and propose a clear, fair, and comprehensive methodology for comparing hardware performance of SHA-3 candidates and any future algorithms competing to become a new cryptographic standard. Our methodology is based on the use of FPGA devices from various vendors. The advantages of using FPGAs for comparison include short development time, wide availability of tools, and a limited number of vendors dominating the market.

The hardware evaluation of SHA-3 candidates started shortly after announcing the specifications and reference software implementations of 51 algorithms submitted to the contest [7][8][10]. The majority of initial comparisons were limited to less than five candidates, and their results have been published at [10]. The more comprehensive efforts became feasible only after NISTs announcement of 14 candidates qualified to the second round of the competition in July 2009. Since then, two comprehensive studies have been reported in the Cryptology ePrint Archive [11][12]. The first, from the University of Graz, has focused on ASIC technology, the second from two institutions in Japan, has focused on the use of the FPGA-based SASEBO-GII board from AIST, Japan. Although both studies generated quite comprehensive results for their respective technologies, they did not quite address the issues of the uniform methodology, which could be accepted and used by a larger number of research teams. Our study is intended to fill this gap, and put forward the proposal that could be evaluated and commented on by a larger cryptographic community.

Chapter 2

Methodology

2.1 Choice of a Language, FPGA Devices, and Tools

Out of two major hardware description languages used in industry, VHDL and Verilog HDL, we choose VHDL. We believe that either of the two languages is perfectly suited for the implementation and comparison of SHA-3 candidates, as long as all candidates are described in the same language. Using two different languages to describe different candidates may introduce an undesired bias to the evaluation.

FPGA devices from two major vendors, Xilinx and Altera, dominate the market with about 90% of the market share. We therefore feel that it is appropriate to focus on FPGA devices from these two companies. In this study, we have chosen to use seven families of FPGA devices from Xilinx and Altera. These families include two major groups, those optimized for minimum cost (Spartan 3 from Xilinx, and Cyclone II and III from Altera) and those optimized for high performance (Virtex 4 and 5 from Xilinx, and Stratix II and III from Altera). Within each family, we use devices with the highest speed grade, and the largest number of pins.

As CAD tools, we have selected tools developed by FPGA vendors themselves: Xilinx ISE Design Suite v. 11.1 (including Xilinx XST, used for synthesis) and Altera Quartus II v. 9.1 Subscription Edition Software.

2.2 Performance Metrics for FPGAs

Choosing proper performance metrics for the implementation of hash functions (or any other cryptographic transformations) using FPGAs is a non-trivial task, and no clear consensus exists so far on how these metrics should be defined. Below we summarize our proposed approach, which we applied in our study.

Speed.

In order to characterize the speed of the hardware implementation of a hash function, we suggest using Throughput, understood as a throughput (number of input bits processed per unit of time) for long messages. To be exact, we define Throughput using the following formula:

$$Throughput = \frac{block\_size}{T \cdot (HTime(N+1) - HTime(N))} \tag{2.1}$$

where *block_size* is a message block size, characteristic for each hash function, $HTime(N)$ is a total number of clock cycles necessary to hash an N-block message, T is a clock period, different and characteristic for each hardware implementation of a specific hash function.

Throughput defined this way is typically independent of N (and thus the size of the message), as in all hash function architectures we investigated so far, the expression $HTime(N+1) - HTime(N)$ is a constant that corresponds to the number of clock cycles between processing of two subsequent input blocks.

The effective throughput for short messages is always smaller, and is expressed by the formula

$$Throughput_{eff} = \frac{N \cdot block\_size}{T \cdot HTime(N)} \tag{2.2}$$

In this paper, we provide the exact formulas for $HTime(N)$ for each SHA-3 candidate (see Table 4.2), and values of $f = 1/T$ for each algorithm–FPGA device pair (see Tables 4.8 and 4.9). Therefore, we provide sufficient information to calculate and compare values of the effective throughputs for each specific message size, which may be of interest in a given application.

For short messages, it is more important to evaluate the total time required to process a message of a given size (rather than throughput). The size of the message can be chosen depending on the requirements of an application. For example, in the eBASH study of software implementations of hash functions, execution times for all sizes of messages, from 0-bytes (empty message) to 4096 bytes, are reported, and five specific sizes 8, 64, 576, 1536, and 4096 are featured in the tables [13]. The generic formulas we include in this paper (see Table 4.2) allow the calculation of the execution times for any message size.

In order to characterize the capability of a given hash function implementation for processing short messages, we present in this study the comparison of execution times for an empty message (one block of data after padding) and a 100-byte (800-bits) message before padding (which becomes equivalent for majority, but not all, of the investigated functions to 1024 bits after padding).

To be exact our parameters are defined as follows

$$T_{empty} = T \cdot HTime(1) \tag{2.3}$$

$$T_{100B} = T \cdot HTime\left(\frac{padlen(800)}{block\_size}\right), \tag{2.4}$$

where $padlan(800)$ denotes the size of an 800-bit message after padding.

Resource Utilization/Area.

Resource utilization is particularly difficult to compare fairly in FPGAs, and is often a source of various evaluation pitfalls. First, the basic programmable block (such as CLB slice in Xilinx FPGAs) has a different structure and different capabilities for various FPGA families from different vendors. For example, in Virtex 5, a CLB slice includes four 6-input Look-Up-Tables (LUTs); in Spartan 3 and Virtex 4, a CLB slice includes two 4-input LUTs. In Cyclone II and Cyclone III, the basic programmable block is called Logic Element (LE); in Stratix II and III, the basic programmable component has a different structure and is called ALUT (Adaptive Look-Up Table). Taking this issue into account, we suggest avoiding any comparisons across family lines. Secondly, all modern FPGAs include multiple dedicated resources, which can be used to implement specific functionality. These resources include Block RAMs (BRAMs), multipliers (MULs), and DSP units in Xilinx FPGAs, and memory blocks, multipliers, and DSP units in Altera FPGAs. In order to implement a specific operation, some of these resources may be interchangable, but there is no clear conversion factor to express one resource in terms of the other.

Therefore, we suggest in the general case, treating resource utilization as a vector, with coordinates specific to a given FPGA family. For example,

$$Resource\_Utilization_{Spartan3} = (\#CLBslices, \#BRAMs, \#MULs) \tag{2.5}$$

$$Resource\_Utilization_{CycloneIII} = (\#LE, \#memory\_bits, \#MULs) \tag{2.6}$$

Taking into account that vectors cannot be easily compared to each other, we have decided to opt out of using any dedicated resources in the hash function implementations used for our comparison. Thus, all coordinates of our vectors, other than the first one have been forced (by choosing appropriate options of the synthesis and implementation tools) to be zero. This way, our resource utilization (further referred to as Area) is characterized using a single number, specific to the given family of FPGAs, namely the number of CLB slices ($\#CLBslices$) for Xilinx FPGAs, the number of Logic Elements ($\#LE$) for Cyclone II and Cyclone III, and the number of Adaptive Look-Up Tables ($\#ALUT$) in Stratix II and Stratix III.

The resource utilization vector in FPGAs (or even its simplified one-coordinate form, referred to as Area above) cannot be easily translated to an equivalent area or the number of transistors in ASICs. Any attempts to define a resource utilization unit that would apply to both technologies (such as an equivalent logic gate) have been mostly unsuccessful, and of limited value in practice. The only common denominator is cost, but unfortunately the prices of integrated circuits, and FPGAs in particular, are not commonly available, and are affected by multiple non-technical factors (including the number of units ordered, the relationship between companies, etc.)

2.3 Uniform Interface

In order to remove any ambiguity in the definition of our hardware cores for SHA-3 candidates, and in order to make our implementations as practical as possible, we have developed an interface shown in Fig. 2.1a, and described below. In a typical scenario, the SHA core is assumed to be surrounded by two standard FIFO modules: Input FIFO and Output FIFO, as shown in Fig. 2.1b. In this configuration, SHA core is an active module, while a surrounding logic (FIFOs) is passive. Passive logic is much easier to implement, and in our case is composed of standard logic components, FIFOs, available in any major library of IP cores.

Each FIFO module generates signals empty and full, which indicate that the FIFO is empty and/or full, respectively. Each FIFO accepts control signals write and read, indicating that the FIFO is being written to and/or read from, respectively.

The aforementioned assumptions about the use of FIFOs as surrounding modules are very natural and easy to meet. For example, if a SHA core implemented on an FPGA communicates with an outside world using PCI, PCI-X, or PCIe interface, the implementations of these interfaces most likely already include Input and Output FIFOs, which can be directly connected to a SHA core. If a SHA core communicates with another core implemented on the same FPGA, then FIFOs are often used on the boundary between the two cores in order to accommodate for any differences between the rate of generating data by one core and the rate of accepting data by another core.

Additionally, the inputs and outputs of our proposed SHA core interface do not need to be necessarily generated/consumed by FIFOs. Any circuit that can support control signals src_ready and src_read can be used as a source of data. Any circuit that can support control signals dst_ready and dst_write can be used as a destination for data.

The exact format of an input to the SHA core, for the case of pre-padded messages, is shown in Fig. 2.2. Two scenarios of operation are supported. In the first scenario, the message bitlength after padding is known in advance and is smaller than 2^w. In this scenario, shown in Fig. 2.2a, the first word of input represents message length after padding,

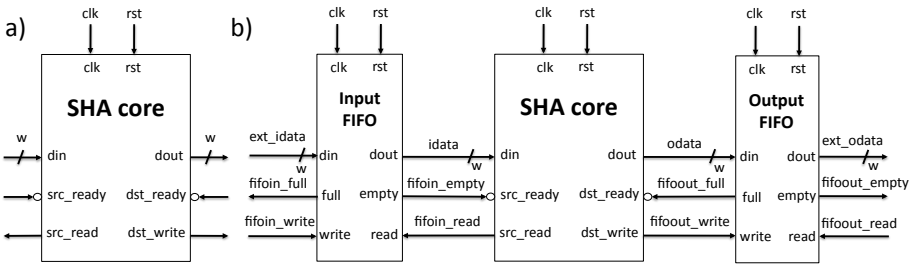

Figure 2.1: a) Input/output interface of a SHA core. b) A typical configuration of a SHA core connected to two surrounding FIFOs.

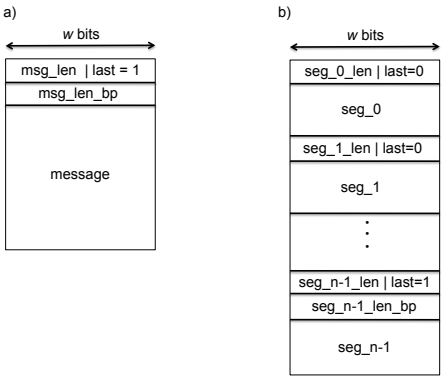

Figure 2.2: Format of input data for two different operation scenarios: a) with message bitlength known in advance, and b) with message bitlength unknown in advance. Notation: msg_len – message length after padding, msg_len_bp – message length before padding, seg_i_len – segment i length after padding, 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), "|" – bitwise OR.

expressed in bits. This word has the least significant bit, representing a flag called last, set to one. This word is followed by the message length before padding. This value is required by several SHA-3 algorithms using internal counters (such as BLAKE, ECHO, Shavite-3, and Skein), even if padding is done outside of the SHA core. These two control words are followed by all words of the message.

The second format, shown in Fig. 2.2b, is used when either message length is not known in advance, or it is greater than 2^w. In this case, the message is processed in segments of data denoted as seg_0, seg_1,...,seg_n-1. For the ease of processing data by the hash core, the size of the segments, from seg_0 to seg_n-2 is required to be always an integer multiple of the block size b, and thus also of the word size w. The least significant bit of the segment length expressed in bits is thus naturally zero, and this bit, treated as a flag called last, can be used to differentiate between the last segment and all previous segments of the message. The last segment before padding can be of arbitrary length $< 2^w$. This segment is processed in the same way as the entire message in scenario a). This way there is no need for any additional signal to distinguish between these two scenarios. Scenario a) is a special case of scenario b). In case the SHA core supports padding, the protocol can be even simpler, as explained in [14].

Please note that scenario b) is very similar to the way data is processed by a typical software API for hash functions, such as [15]. The Update function of the software API corresponds to processing segments from seg_0 to seg_n-2. The function Final corresponds to the processing of the last segment of data, seg_n-1.

2.4 Assumptions and Simplifications

Our study is performed using the following assumptions. Only the SHA-3 candidate variants with the 256-bit and the 512-bit outputs have been implemented and compared at this point. Other variants, treated either independently or as combinations of multiple variants (all-in-one hash cores) may be subjects of future comparisons.

Padding is assumed to be done outside of the hash cores (e.g., in software). All investigated hash functions have very similar padding schemes, which would lead to similar absolute area overhead if implemented as a part of the hardware core. The relative area penalty will be higher for cores with smaller area used for main processing. The complexity of the padding circuit will also depend on the assumptions regarding the smallest unit of a message (bit, byte, or word), which may be different for specific applications.

Only the primary mode of operation is supported for all functions. Special modes, such as tree hashing or MAC mode are not implemented (their implementation would actually work against the respective candidates, because of the area and speed penalty introduced by these extra features). There is also no support for providing salt specific to each message. The salt values are fixed to all zeros in all SHA-3 candidates supporting this special input (namely BLAKE, ECHO, SHAvite-3, and Skein).

2.5 Optimization Target

We believe that the choice of the primary optimization target is one of the most important decisions that needs to be made before the start of the comparison. The optimization target should drive the design process of every SHA-3 candidate, and it should also be used as a primary factor in ranking the obtained SHA-3 cores. The most common choices are: Maximum Throughput, Minimum Latency, Minimum Area, Throughput to Area Ratio, Product of Latency times Area, etc.

Our choice is the Throughput to Area Ratio, where Throughput is defined as Throughput for long messages, and Area is expressed in terms of the number of basic programmable logic blocks specific to a given FPGA family. This choice has multiple advantages. 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. Otherwise, the input/output latency overhead associated with using a hardware accelerator dominates the total processing time, and the cost of using dedicated hardware (FPGA) is not justified. Optimizing for the best ratio provides a good balance between the speed and the cost of the solution.

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.

2.6 Design Methodology

Our design of all 14 SHA-3 candidates followed an identical design methodology. Each SHA core is composed of the Datapath and the Controller. The Controller 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. The finite state machines responsible for input and output are almost identical for all hash function candidates; the third state machine, responsible for main data processing, is based on a similar template. The similarity of all designs and reuse of common building blocks assures a high fairness of the comparison.

The design of the Datapath starts from the high level architecture. At this point, the most complex task that can be executed in an iterative fashion, with the minimum overhead associated with multiplexing inputs specific to a given iteration round, is identified. The proper choice of such a task is very important, as it determines both the number of clock cycles per block of the message and the circuit critical path (minimum clock period).

It should be stressed that the choice of the most complex task that can be executed in an iterative fashion should not follow blindly the specification of a function. In particular, quite often one round (or one step) from the description of the algorithm is not the most suitable component to be iterated in hardware. Either multiple rounds (steps) or fractions thereof may be more appropriate. In Table 2.1 we summarize our choices of the main iterative tasks of SHA-3 candidates. Each such task is implemented as combinational logic, surrounded by registers.

The next step is an efficient implementation of each combinational block within the DataPath. In Table 2.2, we summarize major operations of all SHA-3 candidates that require logic resources in hardware implementations. Fixed shifts, fixed rotations, and

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	$G_i..G_{i+3}$	**JH**	Round function R_8
BMW	entire function	**Keccak**	Round R
CubeHash	one round	**Luffa**	The Step Function, Step
ECHO	AES round/AES round/ BIG.SHIFTROWS, BIG.MIXCOLUMNS	**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**	4 rounds of Threefish-512

Table 2.2: Major operations of SHA-3 candidates (other than permutations, fixed shifts and fixed rotations). mADDn denotes a multioperand addition with n operands.

Function	**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..x05, 0x07				XOR
Hamsi		LC	Serpent 4x4					XOR
JH			4x4	x2, x5				XOR
Keccak								NOT,AND,XOR
Luffa			4x4	x02				XOR
Shabal					x3, x5		ADD,SUB	NOT,AND,XOR
SHAvite-3			AES 8x8	x02, x03				NOT,XOR
SIMD	NTT				x185, x233	mADD3	ADD	NOT,AND,OR
Skein							ADD	XOR
SHA-256						mADD5		NOT,AND,XOR

other more complex permutations are omitted because they appear in all candidates and require only routing resources (programmable interconnects). The most complex out of logic operations are the Number Theoretic Transform (NTT) [16] in SIMD, linear code (LC) [17] in Hamsi, basic operations of AES (8x8 AES S-box and multiplication by a constant in the Galois Field GF(2^8)) in ECHO, Fugue, Groestl, and SHAvite-3; and multioperand additions in BLAKE, BMW, SIMD, and SHA-256.

For each of these operations we have implemented at least two alternative architectures. NTT was optimized by using a Fast Fourier Transform (FFT) [16]. In Hamsi, the linear code was implemented using both logic (matrix by vector multiplications in GF(4)), and using look-up tables. AES 8x8 S-boxes (SubBytes) were implemented using both look-up tables (stored in distributed memories), and using logic only (following method described in [18], Section 10.6.1.3). Multi-operand additions were implemented using the following four methods: carry save adders (CSA), tree of two operand adders, parallel counter, and a "+" in VHDL [19]). Finally, integer multiplications by 3 and 5 in Shabal have been replaced by a fixed shift and addition.

All optimized implementations of basic operations have been applied uniformly to all SHA-3 candidates. In case the initial testing did not provide a strong indication of superiority of one of the alternative methods, the entire hash function unit was implemented using two alternative versions of the basic operation code, and the results for a version with the better throughput to area ratio have been listed in the result tables.

All VHDL codes have been thoroughly verified using a universal testbench, capable of testing an arbitrary hash function core that follows interface described in Section 2.3 [20]. A special padding script was developed in Perl in order to pad messages included in the Known Answer Test (KAT) files distributed as a part of each candidates submission package. An output from the script follows a similar format as its input, but includes apart from padding bits also the lengths of the message segments, defined in Section 2.3, and shown schematically in Fig. 2.2b. The generation of a large number of results was facilitated by an open source tool ATHENa (Automated Tool for Hardware EvaluatioN) [20]. This benchmarking environment was also used to optimize requested synthesis and implementation frequencies and other tool options.

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. Only the 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. The full specification of each of the algorithms can be found in [21–35].

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.
SIPO	Serial-in-parallel-out unit.
PISO	Parallel-in-serial-out unit.
switch endian word	Wordwise endianness switching.
switch endian byte	Bytewise endianness switching.

3.2 Basic Component Description

This section describes implementations of selected 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)

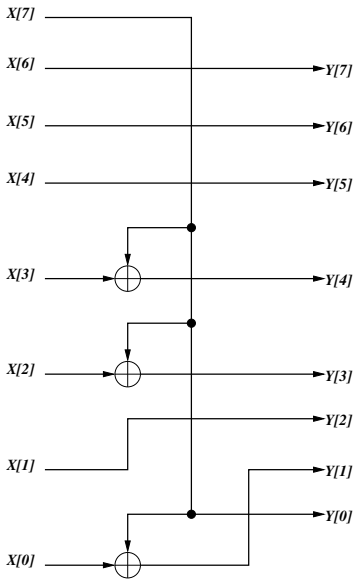

Figure 3.1: Basic : x2

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

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

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 ShiftRows 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 bits. Hence, four instances of MixColumns are required to process the entire AES block of 128 bits. The SBOX and ShiftBytes operations of AES and their full specifications can be found in [36] and [37].

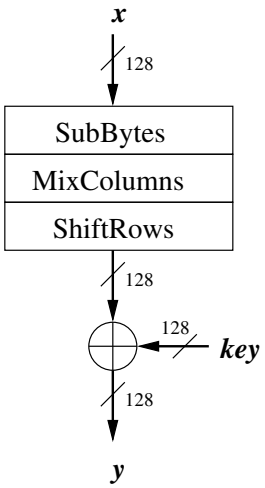

Figure 3.2: Basic : AES Round

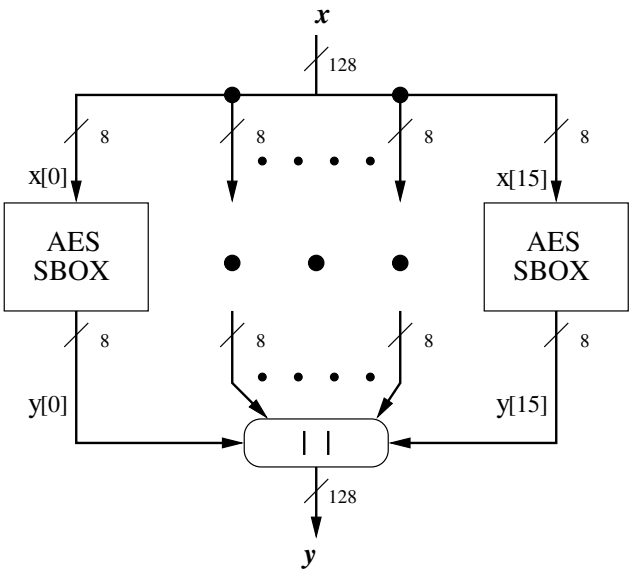

Figure 3.3: Basic : AES SubBytes

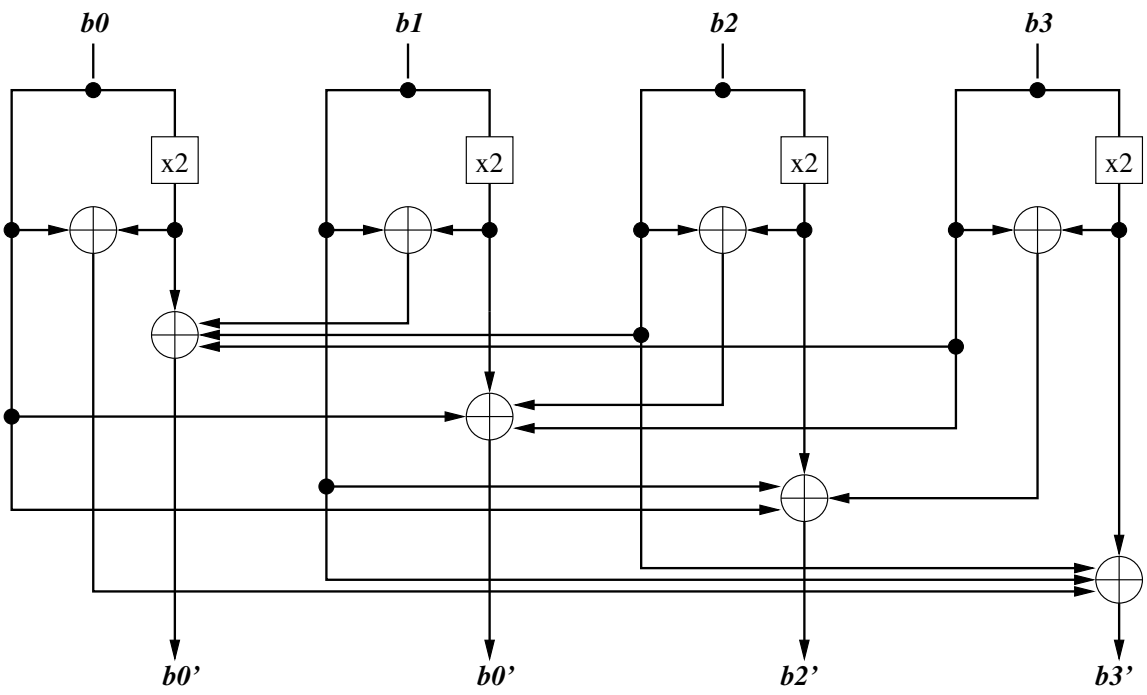

Figure 3.4: Basic : AES MixColumns

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 [21]. 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 applied as inputs to the function PERMUTE and the obtained output is passed to the design's CORE. Simultaneously, the chaining value, CV, 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 an 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:

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

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.

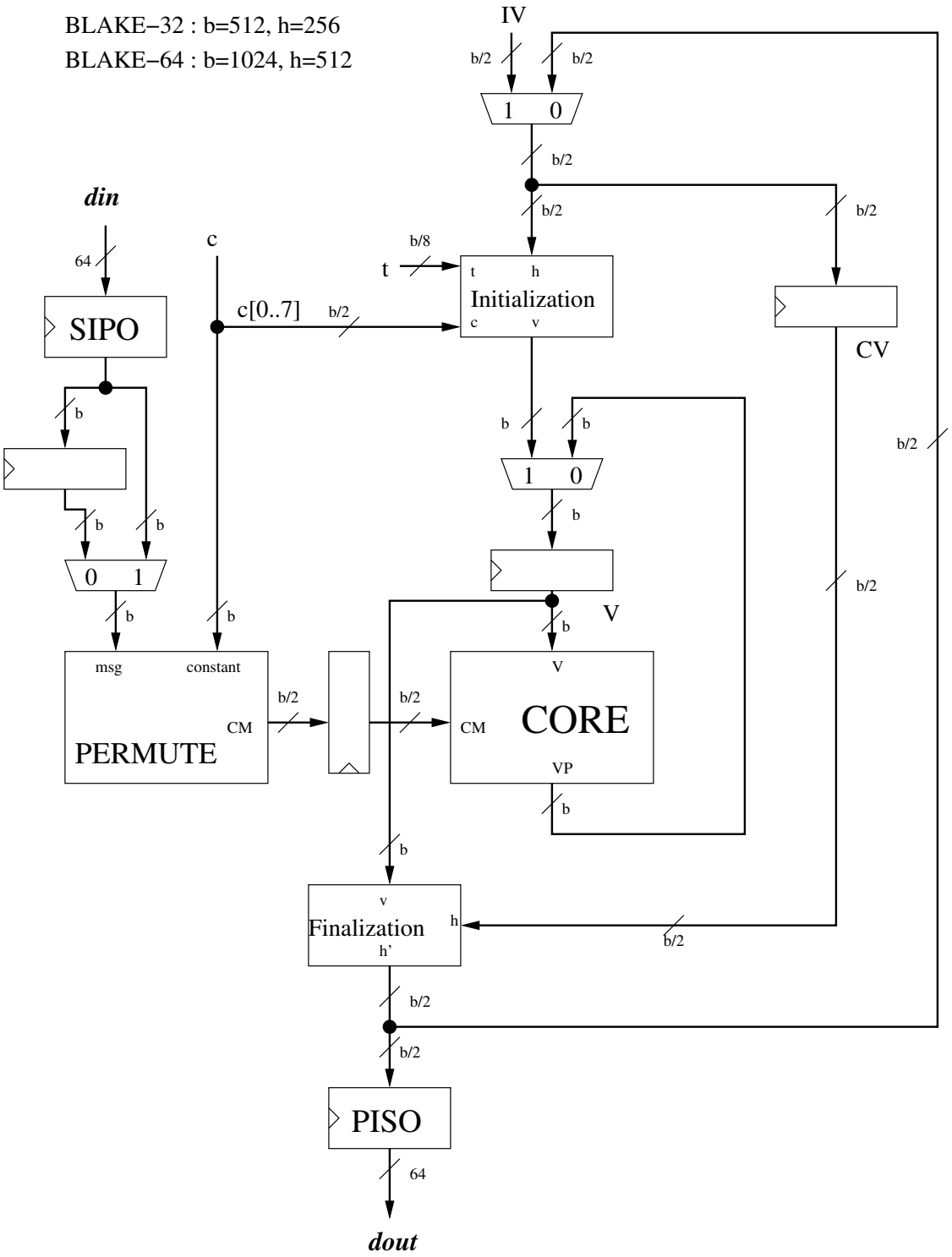

Figure 3.5: BLAKE : Datapath

Table 3.3: BLAKE : Permutation Constants

	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

The CORE unit is shown in Figure 3.7 and represents one half of the BLAKE's round. As specified in [21], 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 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

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

Table 3.5: BLAKE : Rotation Constants of BLAKE-32

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

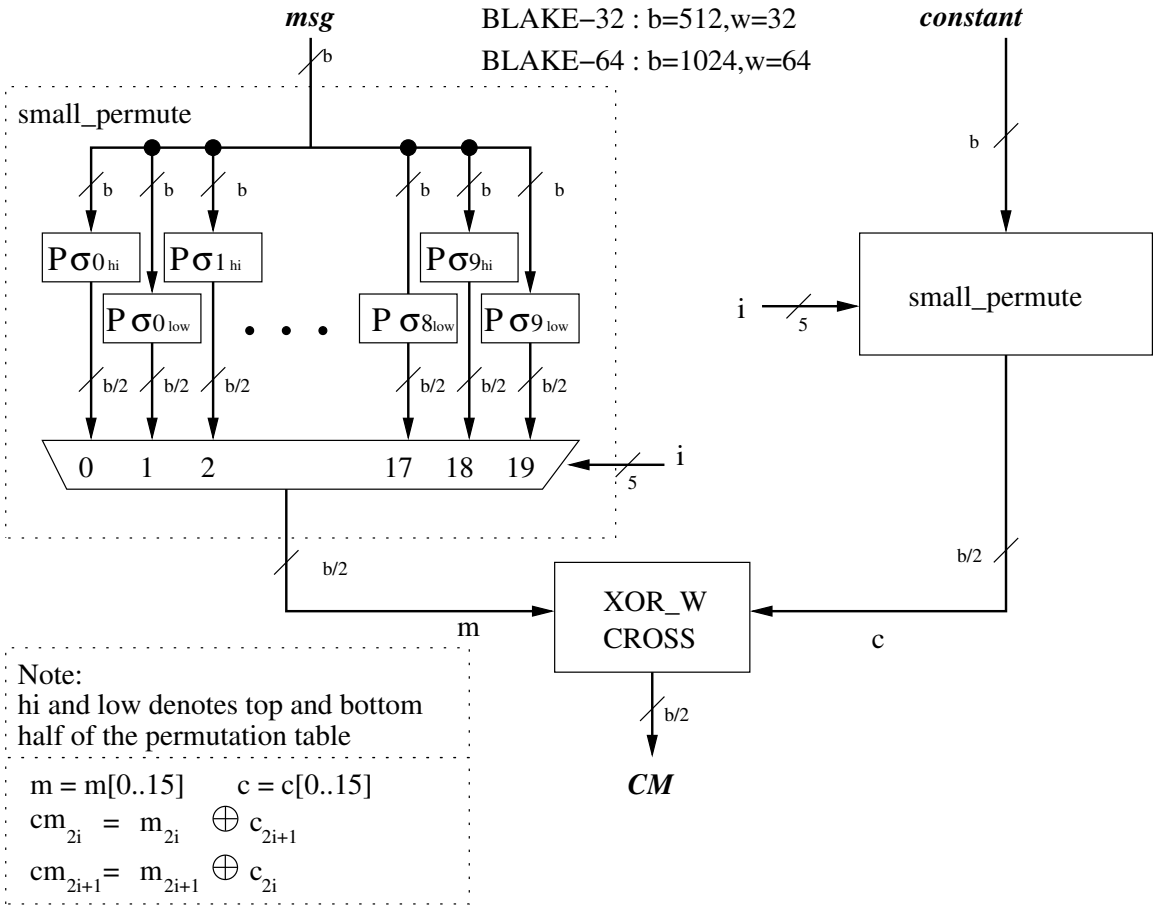

Figure 3.6: BLAKE : PERMUTE

Table 3.6: BLAKE: Rotating Constants of BLAKE-64

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

be found in Section 2.2.1 of [21]. 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 rotation constants are adjusted to reflect the increased word size. These values are described in Table 3.6.

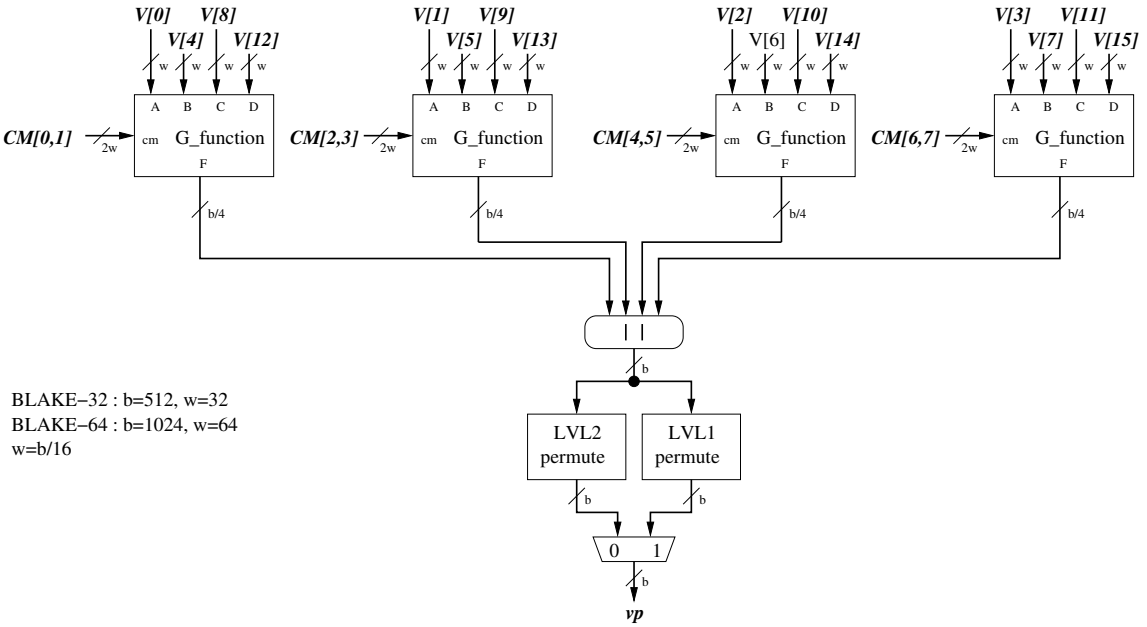

Figure 3.7: BLAKE : CORE

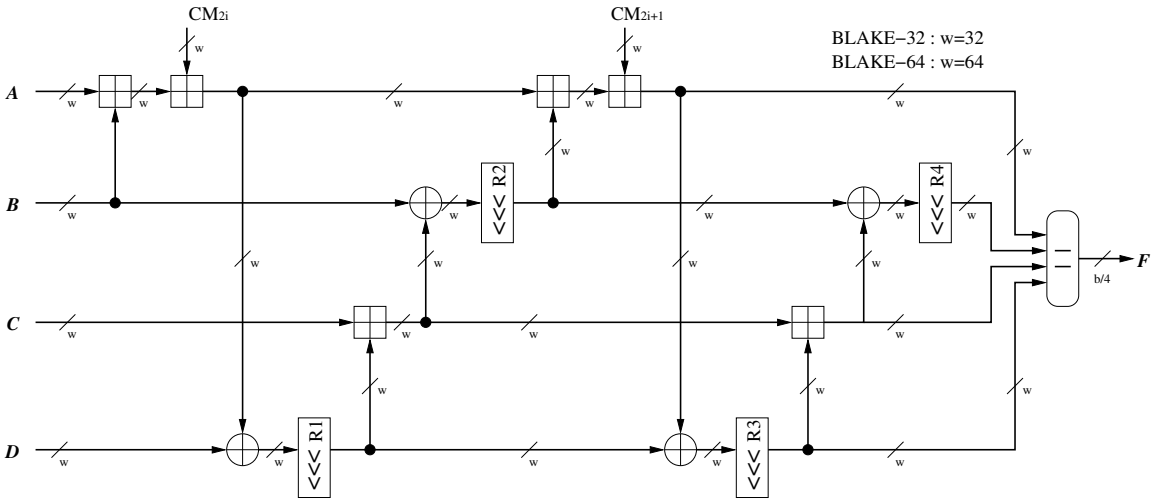

Figure 3.8: BLAKE : G-function

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 in order 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 [22].

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 [22]. All other operations remain the same.

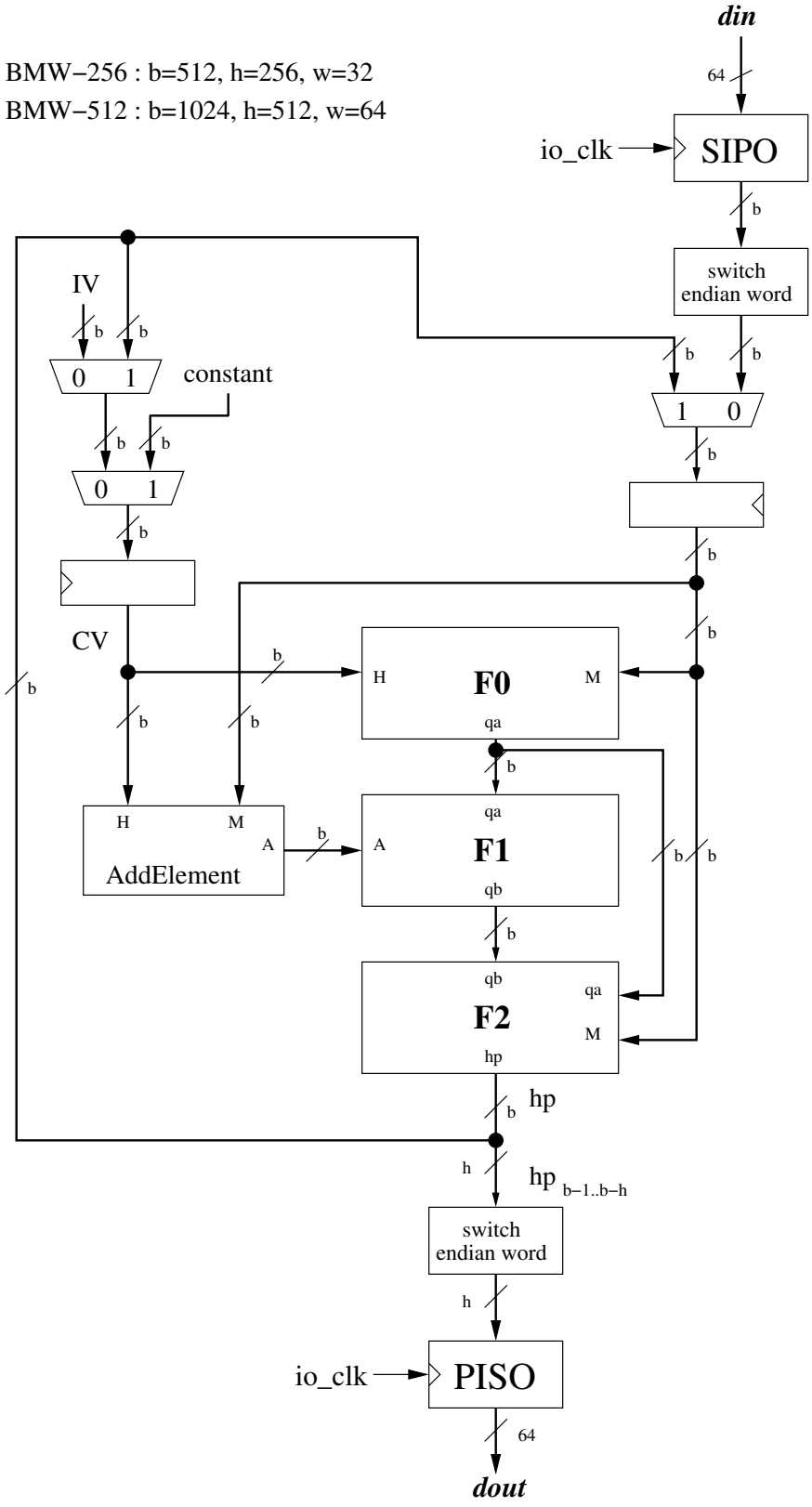

Figure 3.9: BMW : Datapath

3.5 CubeHash

3.5.1 Block Diagram Description

A straightforward 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, the chaining value A 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. Swaps used inside of the round are described in Figure 3.12. 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.

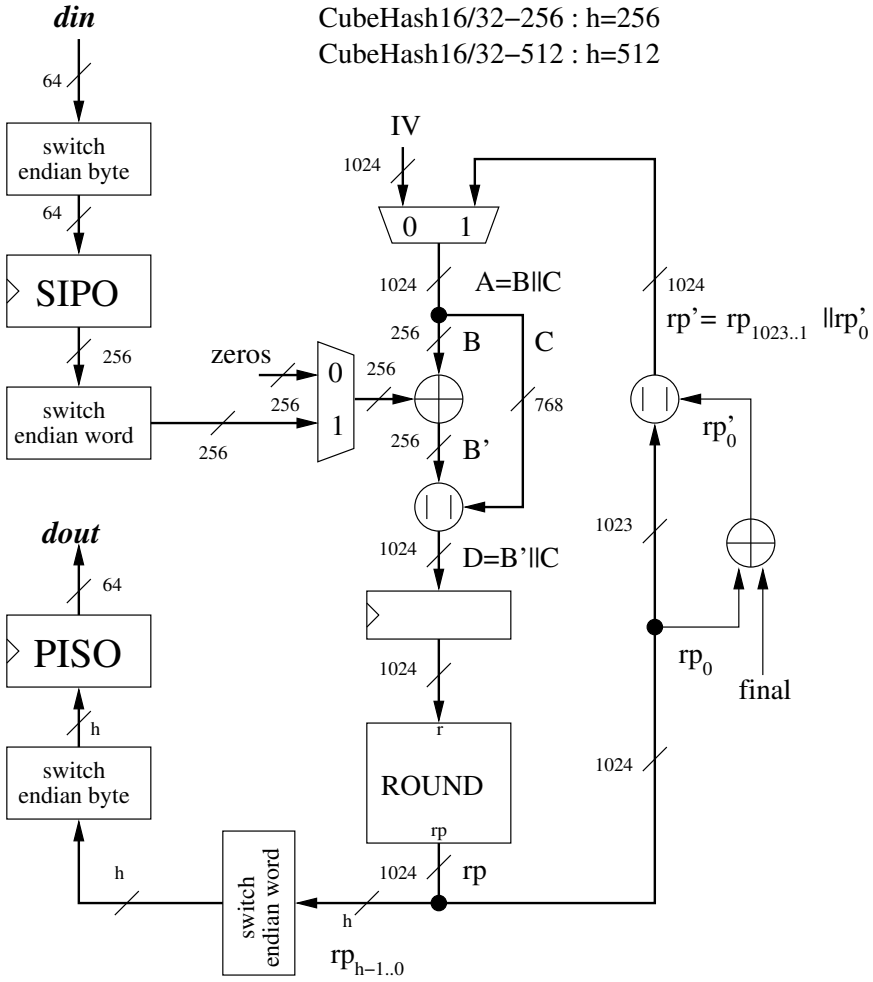

Figure 3.10: CubeHash : Datapath

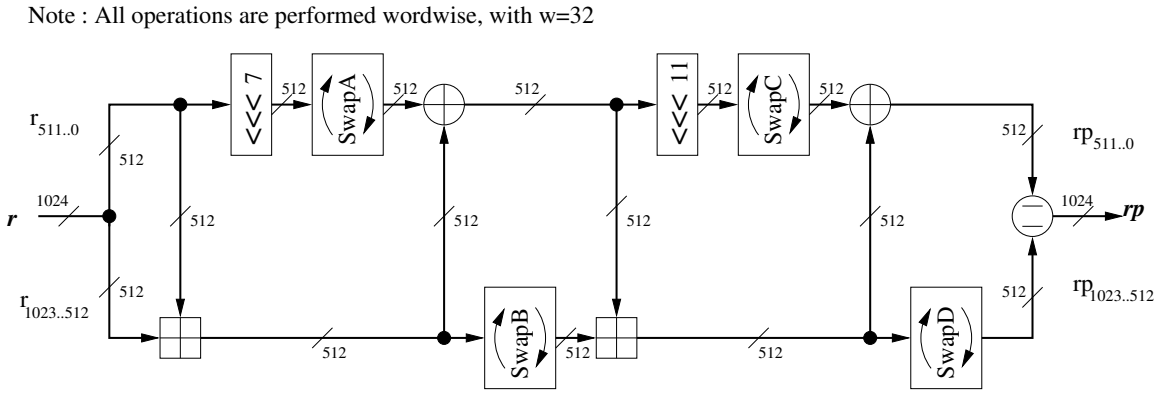

Figure 3.11: CubeHash : Round

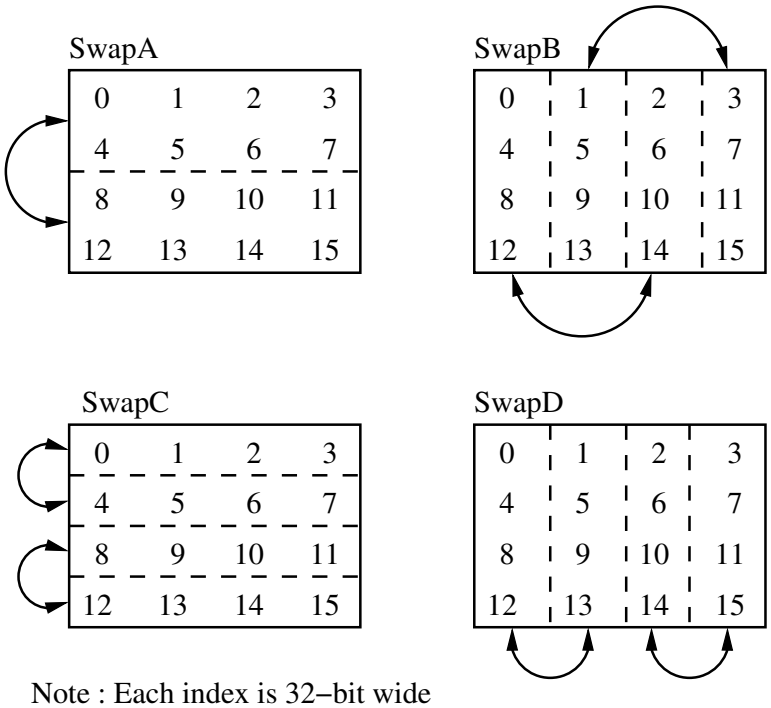

Figure 3.12: CubeHash : Swaps

3.6 ECHO

3.6.1 Block Diagram Description

ECHO's top level datapath is shown in Figure 3.13. 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:

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

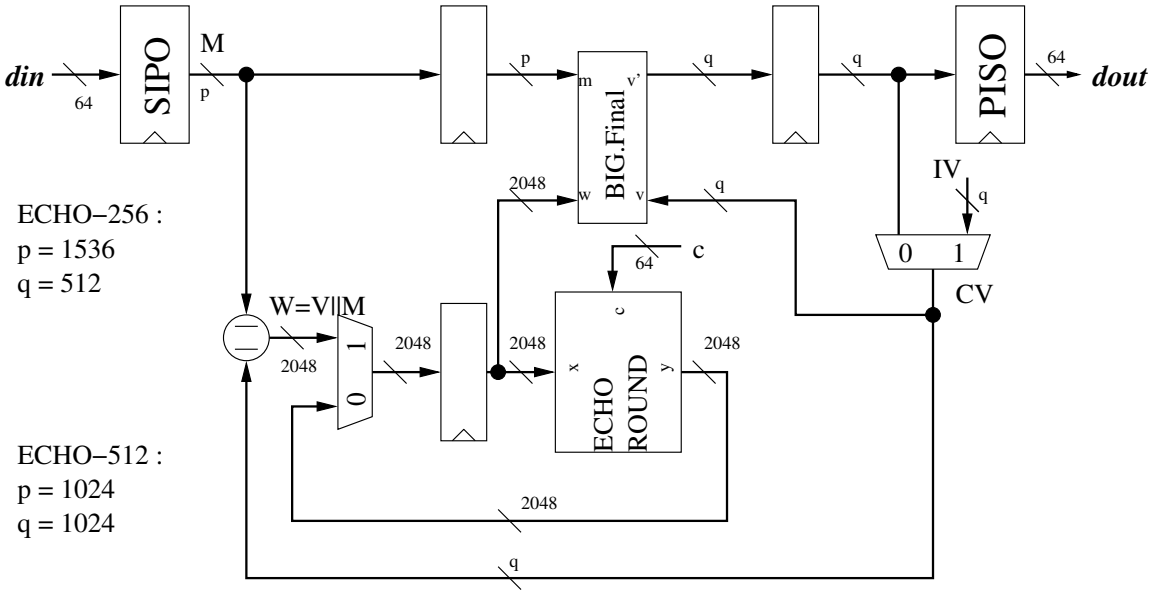

Figure 3.13: ECHO : Datapath

In Figure 3.14, 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 two clock cycles and BIG.ShiftRows and BIG.MixColumns in the third cycle. BigSubBytes operation is shown in Figure 3.15. 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

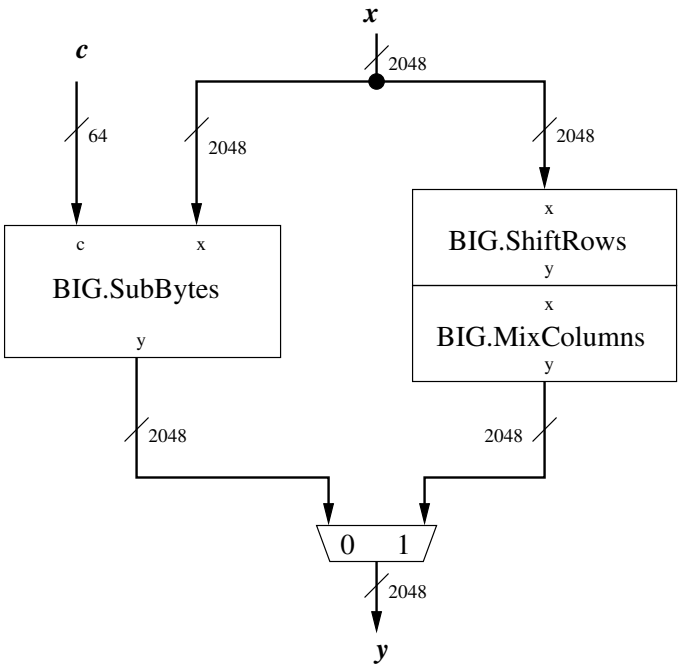

Figure 3.14: ECHO : Round

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.

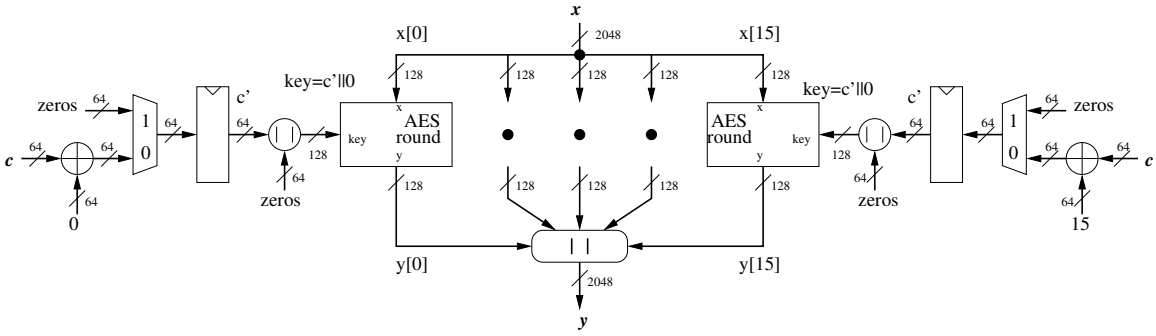

Figure 3.15: ECHO : BIG.SubBytes

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 3.16, a diagram of BIG.MixColumns is shown. BIG.MixColumns separates the state into 4 blocks, each block containing 4 128-bit 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

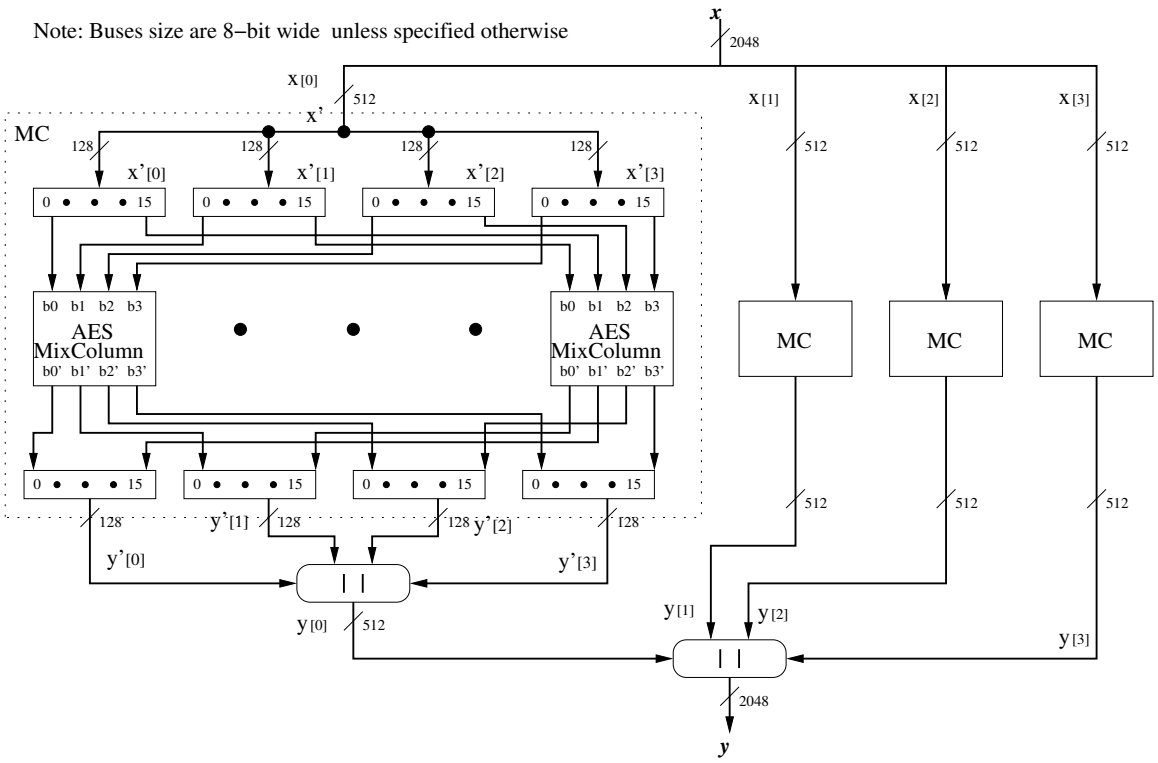

Figure 3.16: ECHO : BIG.MixColumns

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

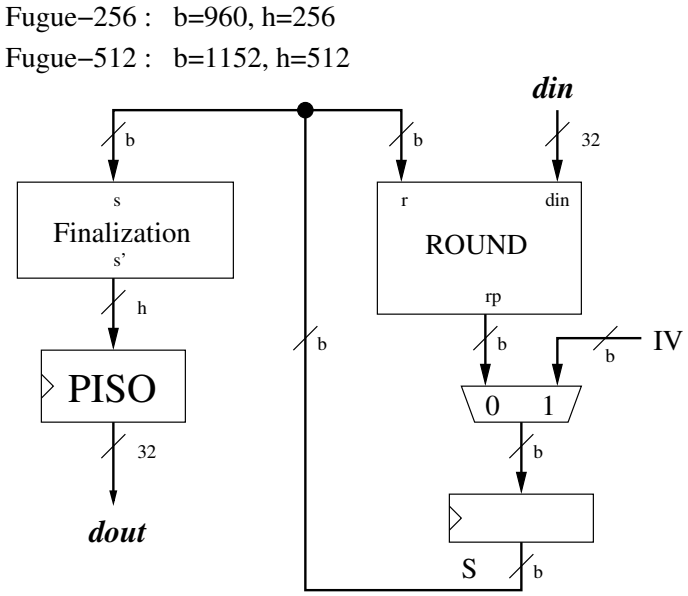

Figure 3.17: Fugue : Datapath

In Figure 3.17, 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:

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

A round of Fugue is shown in Figure 3.18. 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 [25]. TIX operates in parallel as follows:

$$\begin{array}{c} S'[0] = din \\ S'[1] = S[1] \oplus S[24] \\ S'[8] = S[8] \oplus din \\ S'[10] = S[10] \oplus S[0] \end{array}$$

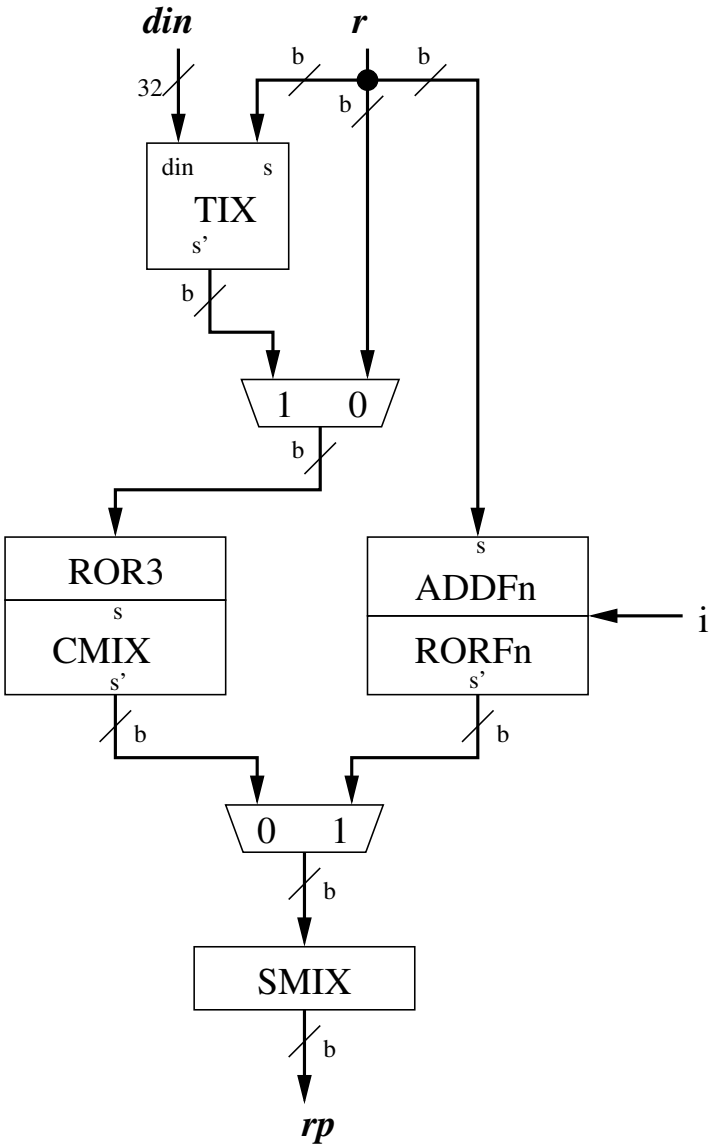

Figure 3.18: Fugue : Round

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	$S'[4] = S[4] \oplus S[0]$ $S'[16] = S[16] \oplus S[0]$ $ROR14$

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:

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

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 3.19. 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. Empty positions in this table correspond to the values zero. All multiplications are defined as multiplications 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 in parallel as follows:

$$\begin{aligned} S'[0] &= din \\ S'[1] &= S[1] \oplus S[24] \\ S'[4] &= S[4] \oplus S[27] \\ S'[7] &= S[7] \oplus S[30] \\ S'[8] &= S[8] \oplus din \\ S'[22] &= S[22] \oplus S[0] \end{aligned}$$

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				

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	$S'[4] = S[4] \oplus S[0]$ $S'[9] = S[9] \oplus S[0]$ $S'[19] = S[19] \oplus S[0]$ $S'[28] = S[28] \oplus S[0]$ $ROR8$

CMIX is now performed as follows:

$$
\begin{aligned}
S'[0] &= S[0] \oplus S[4] \\
S'[1] &= S[1] \oplus S[5] \\
S'[2] &= S[2] \oplus S[6] \\
S'[18] &= S[18] \oplus S[4] \\
S'[19] &= S[19] \oplus S[5] \\
S'[20] &= S[20] \oplus S[6]
\end{aligned}
$$

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

Finally, Finalization is performed as follows:

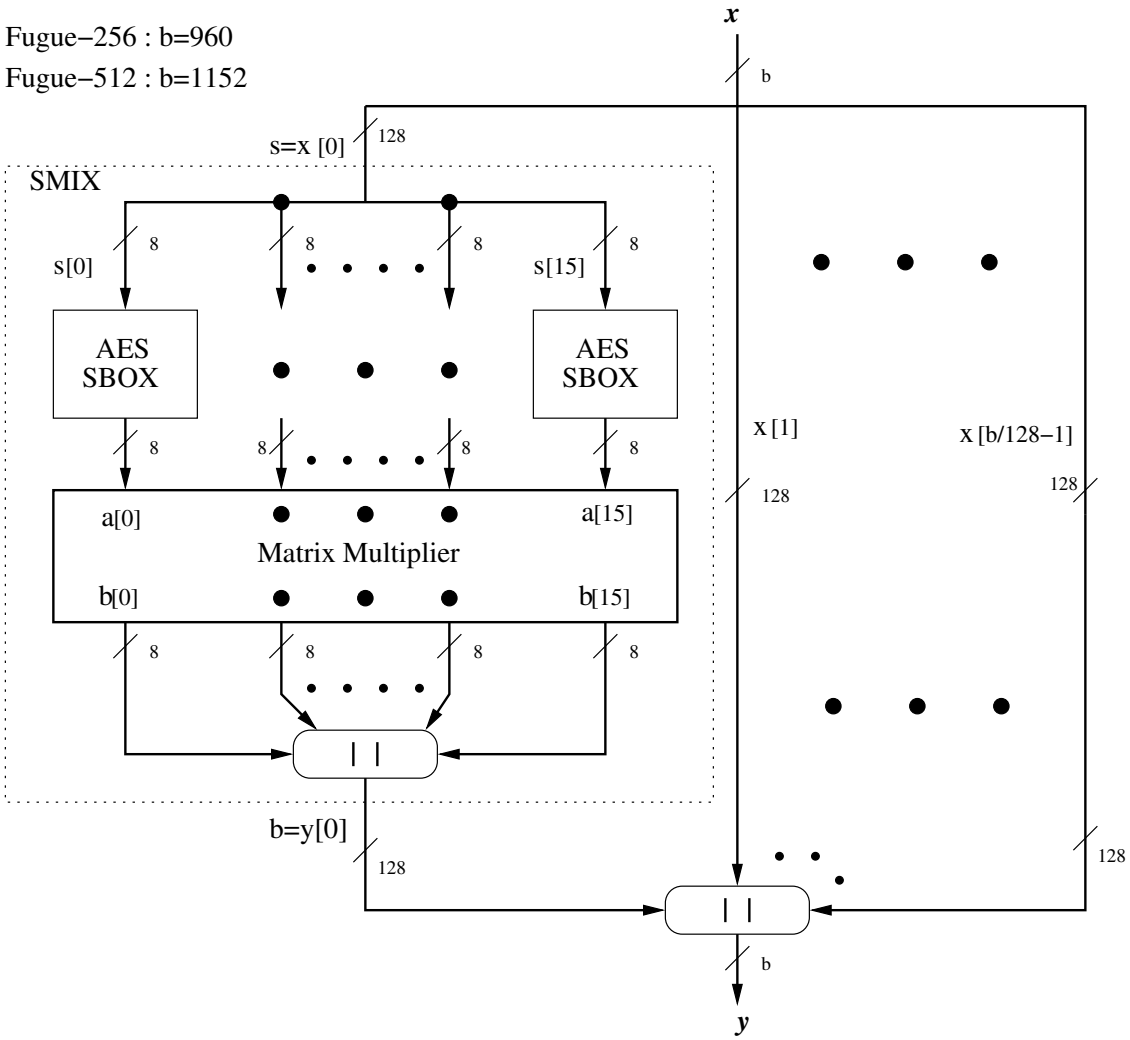

Figure 3.19: Fugue : SMIX

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

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

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 3.20 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 3.21, 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 3.22 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.

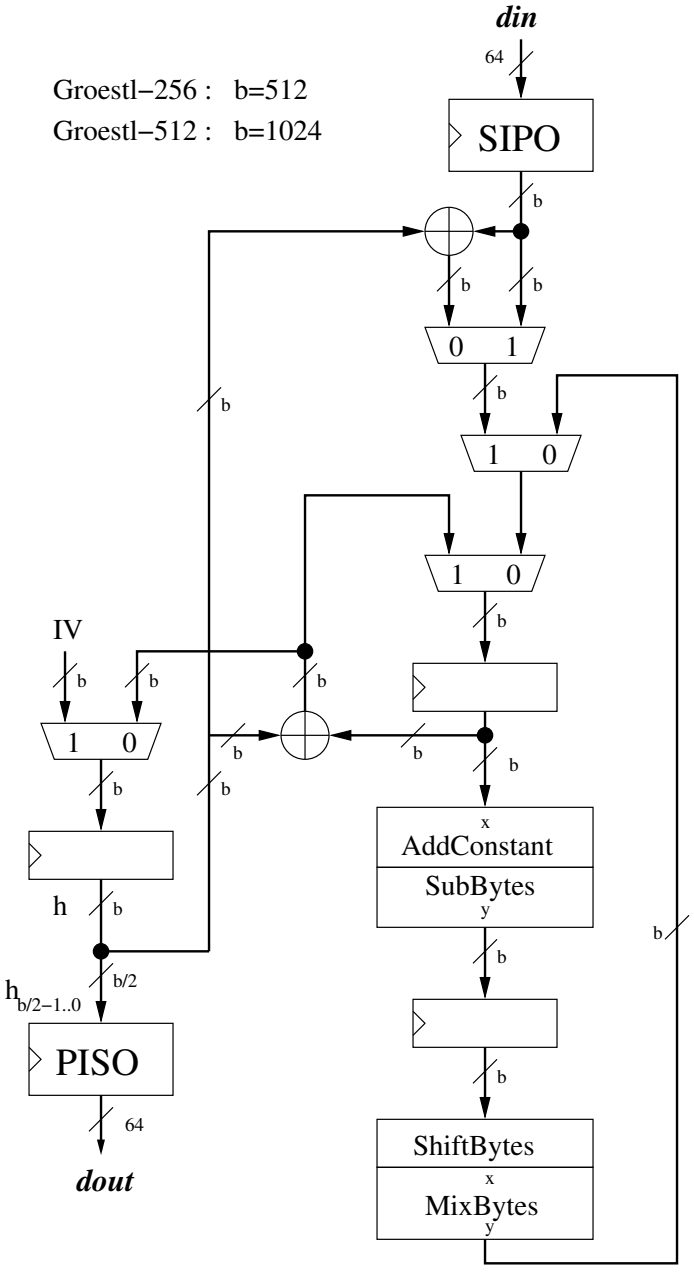

Figure 3.20: Groestl : Datapath

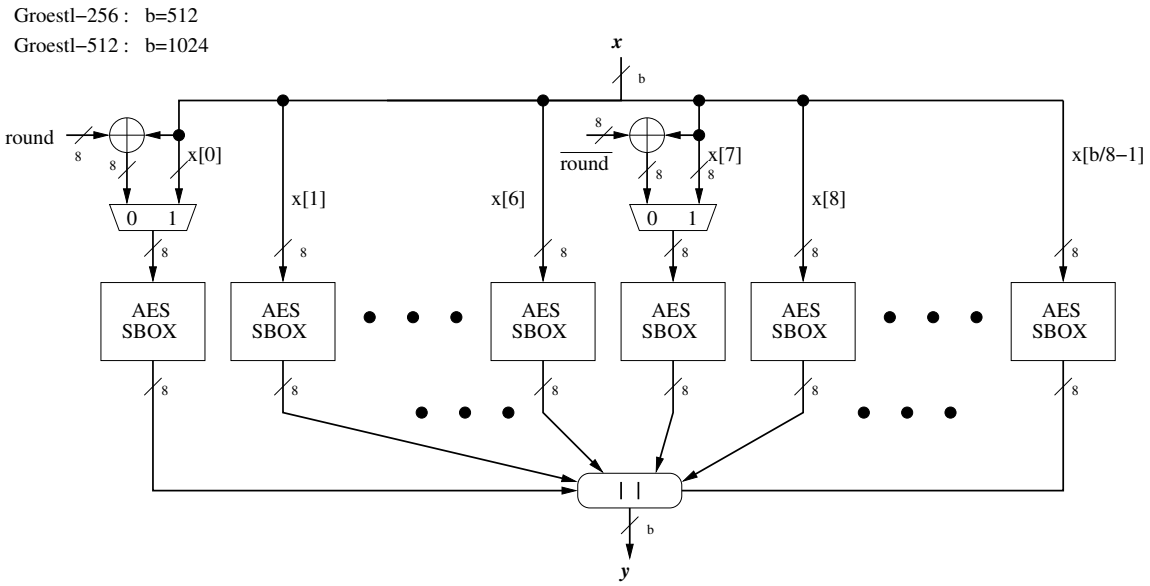

Figure 3.21: Groestl : AddConstant and SubBytes

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.

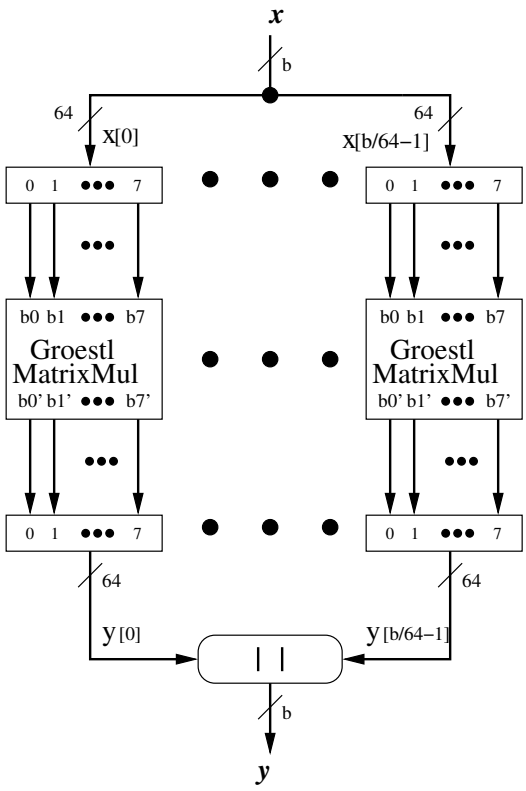

Figure 3.22: Groestl : MixBytes

3.9 Hamsi

3.9.1 Block Diagram Description

The datapath of Hamsi is shown in Figure 3.23. 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 3.24, 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 [27].

Concatenation is performed as follows:

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

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 3.25. 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.

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

The Diffusion Layer is based on the logic function L, shown in 3.26. This function performs the following sequence of operations:

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

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

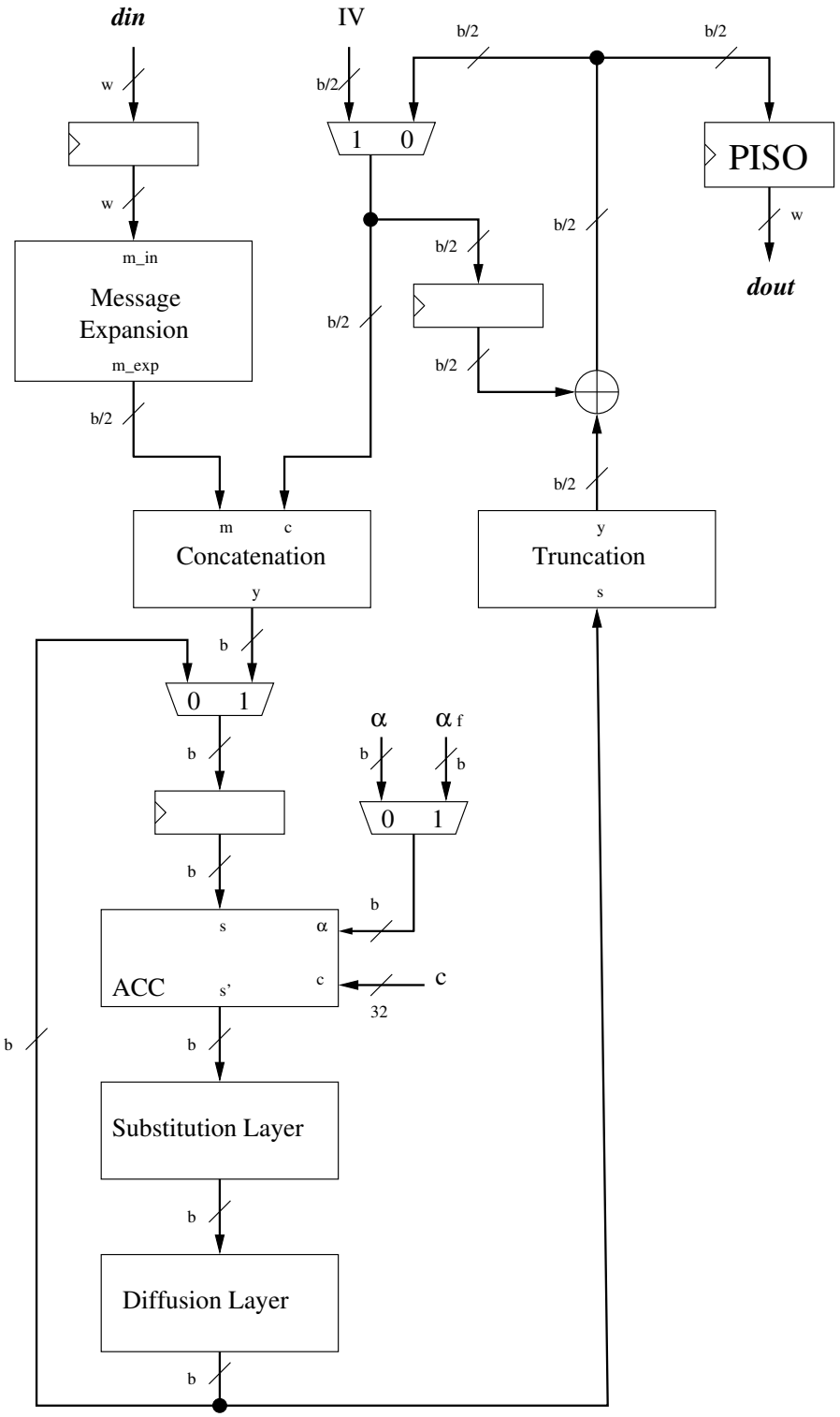

Figure 3.23: Hamsi : Datapath

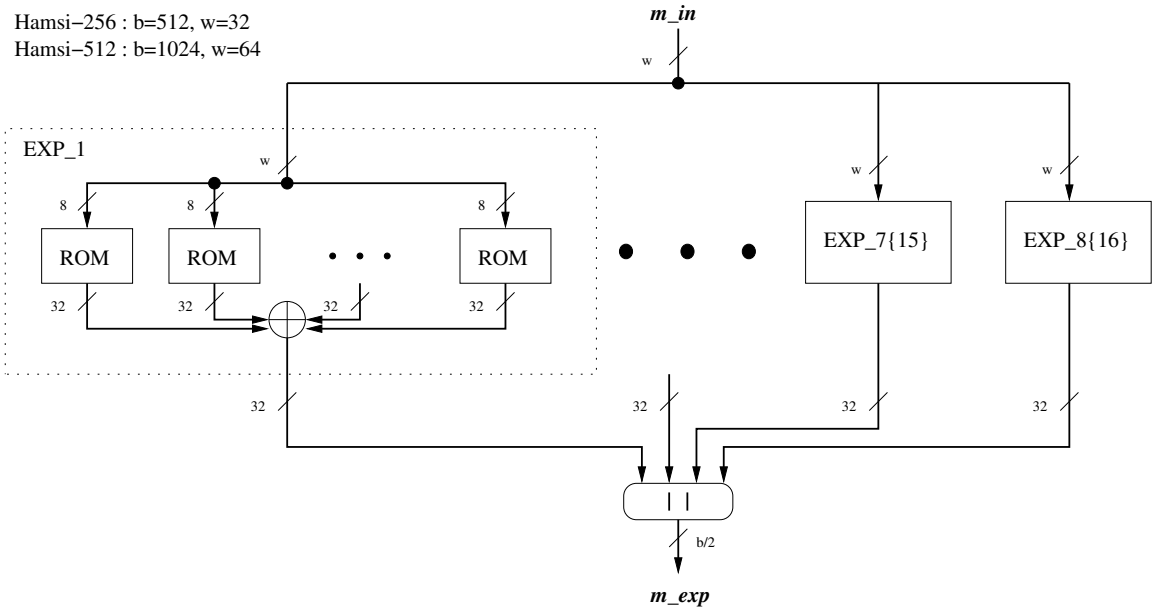

Figure 3.24: Hamsi : Message Expansion

Finally, Truncation is performed as follows:

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

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:

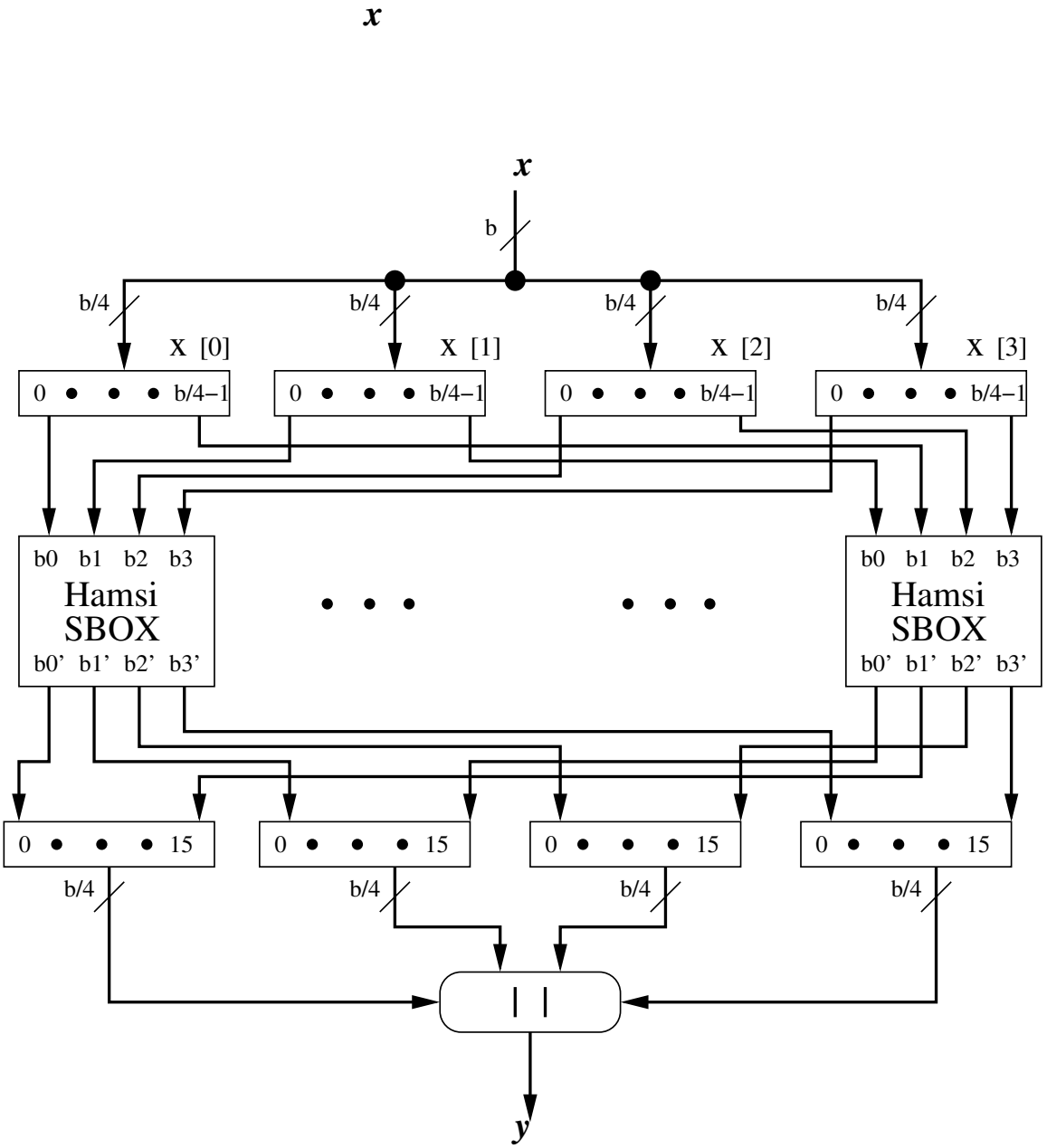

Figure 3.25: Hamsi : Substitution Layer

Note : All bus sizes are 32 bits

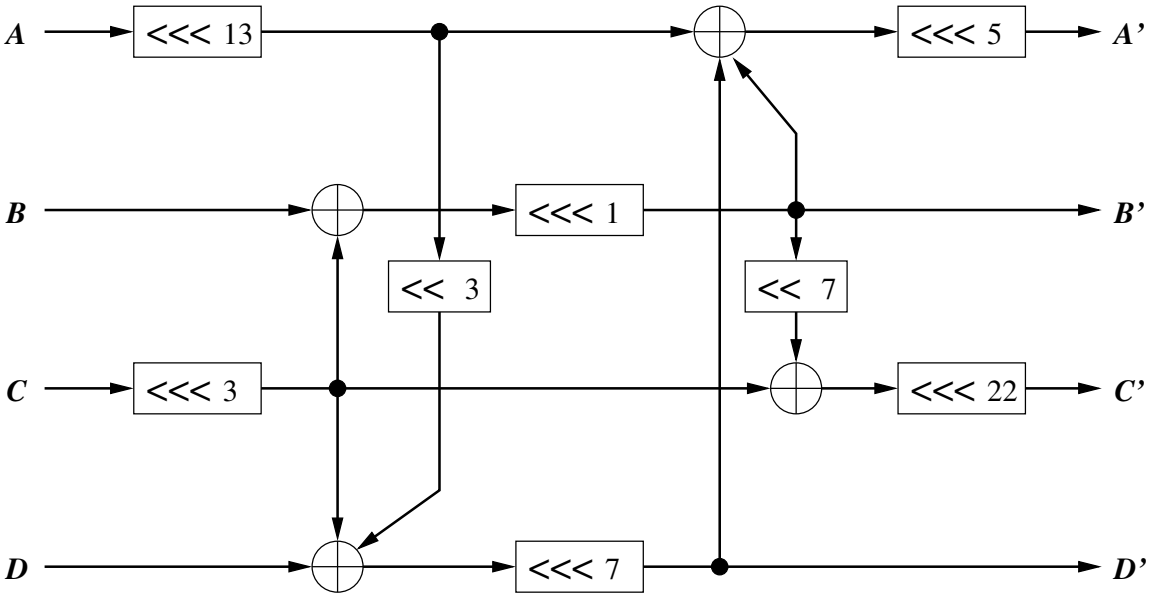

Figure 3.26: Hamsi : L

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

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 3.27. 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 key 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 3.28, 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.

Table 3.13: JH: SBOX

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_0[x]	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
S_1[x]	3	12	5	13	5	7	1	9	15	2	0	4	11	10	14	8

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 3.29. The transformed outputs are then permuted by the PERMUTE

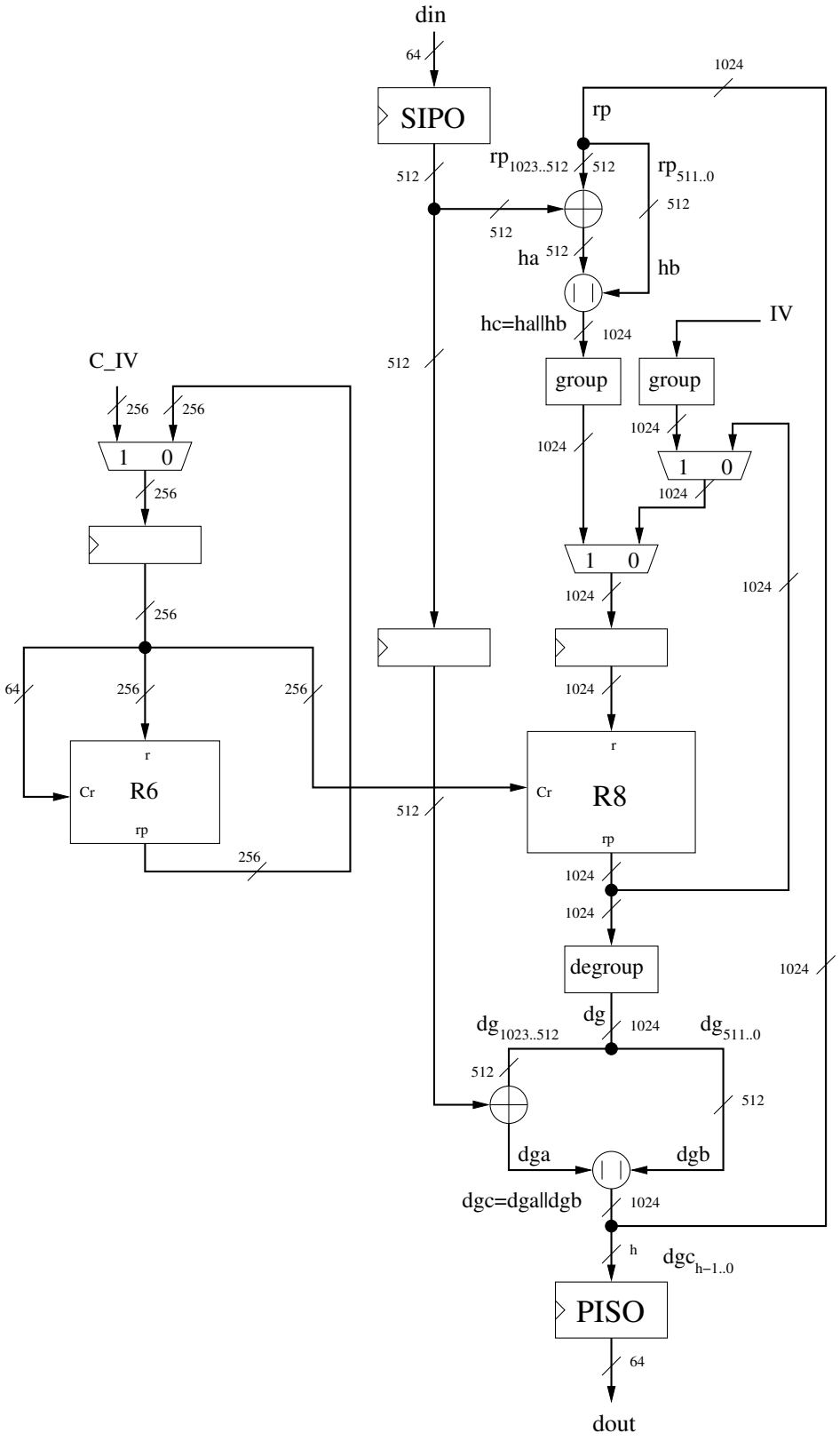

Figure 3.27: JH:Datapath

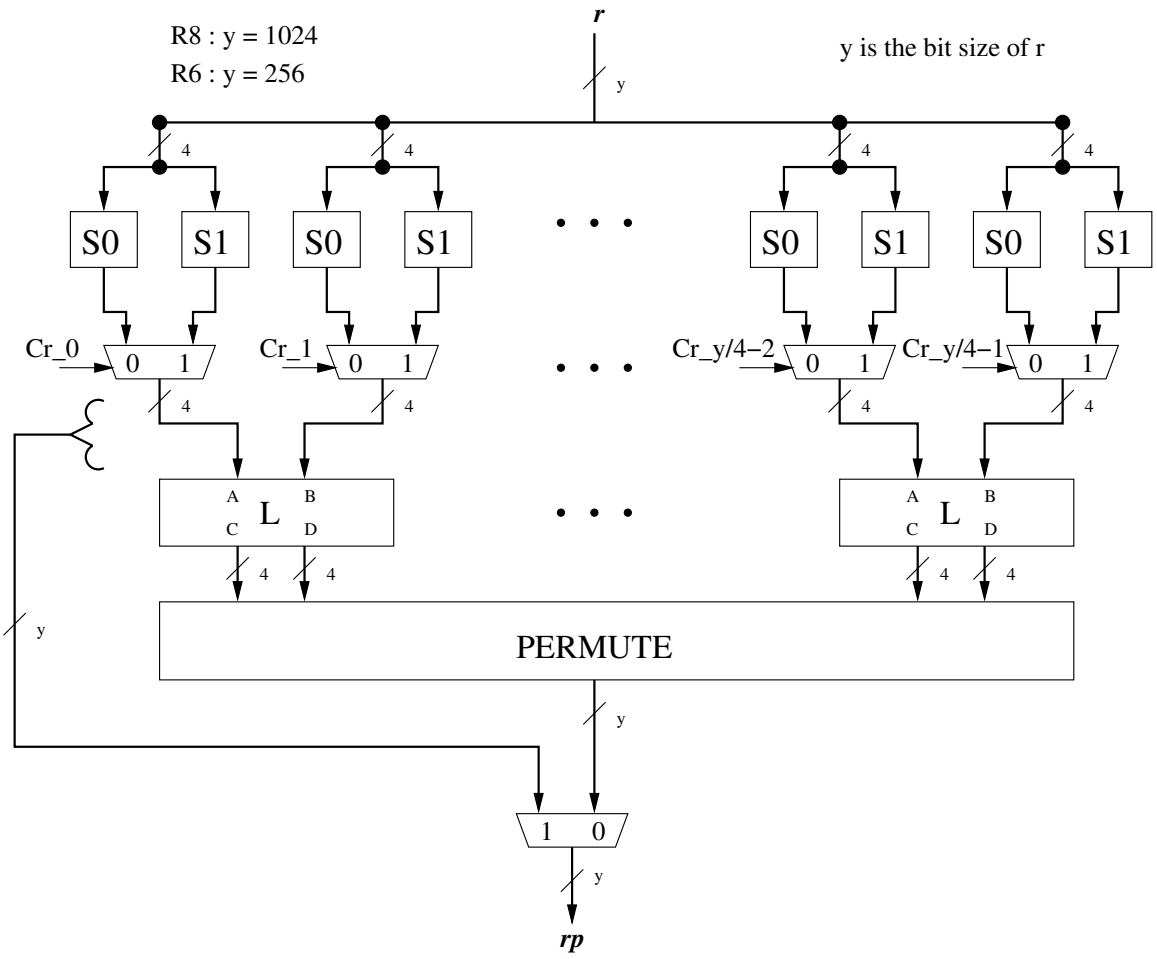

Figure 3.28: JH : Rn

block. PERMUTE can be described by a series of permutations given by the code below. x, y and k refer 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;
```

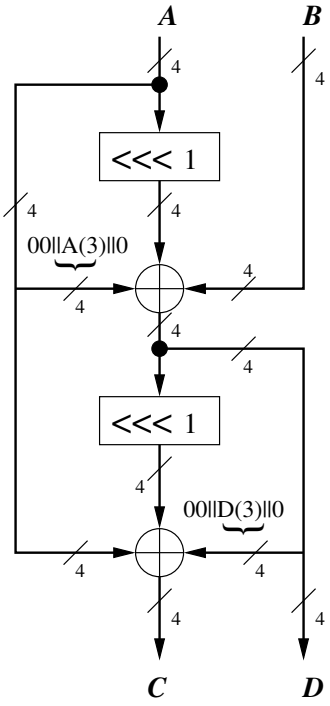

Figure 3.29: JH : Linear Transformation

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.

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 datapath is shown in Figure 3.30. 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 3.31. 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 3.32, 3.33.

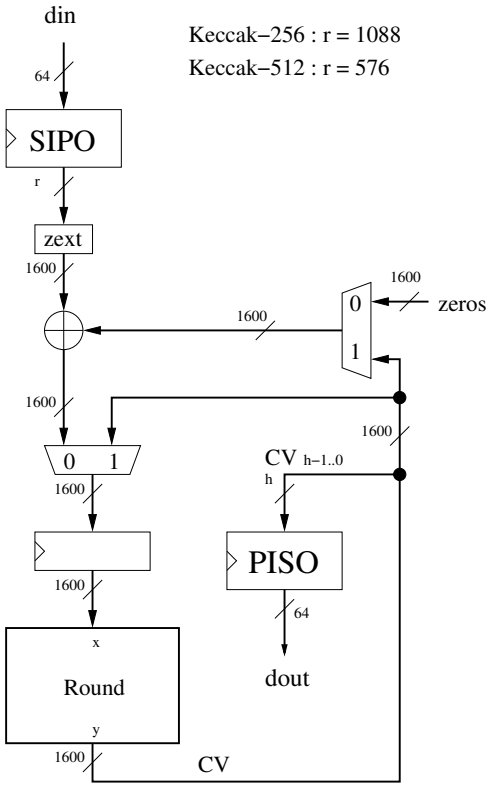

Figure 3.30: Keccak : Datapath

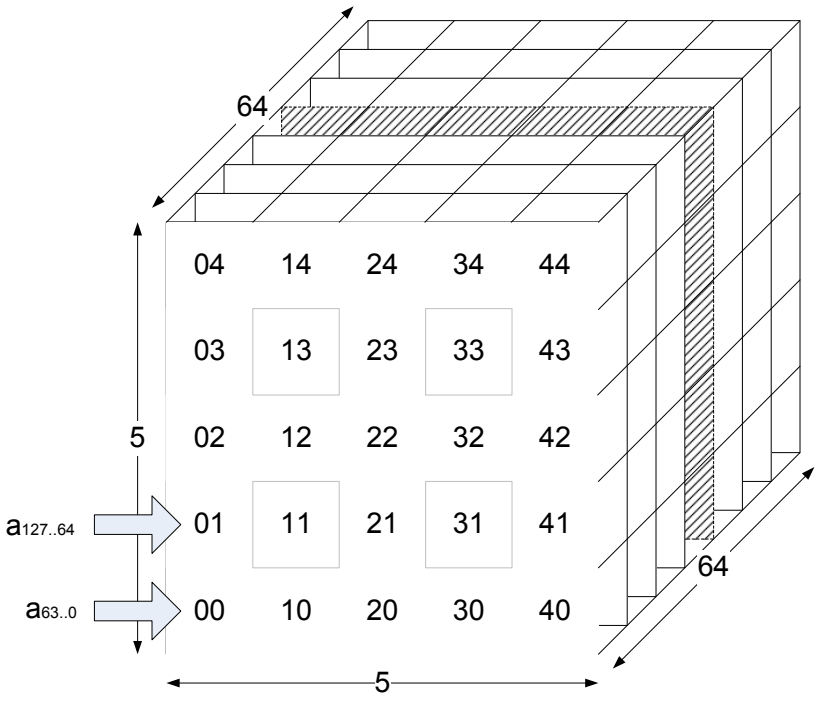

Figure 3.31: Keccak : State

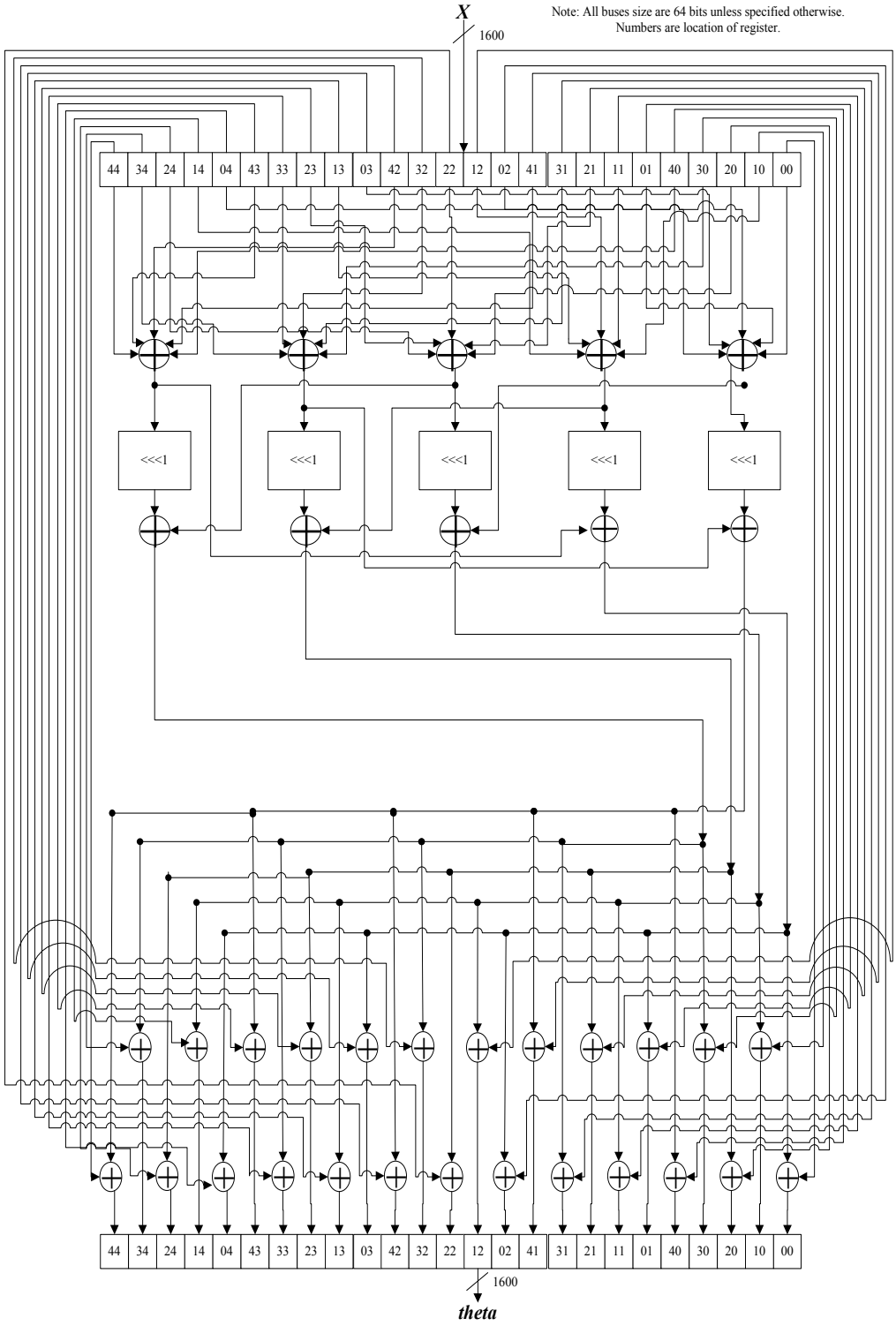

Figure 3.32: Keccak : Round Part 1

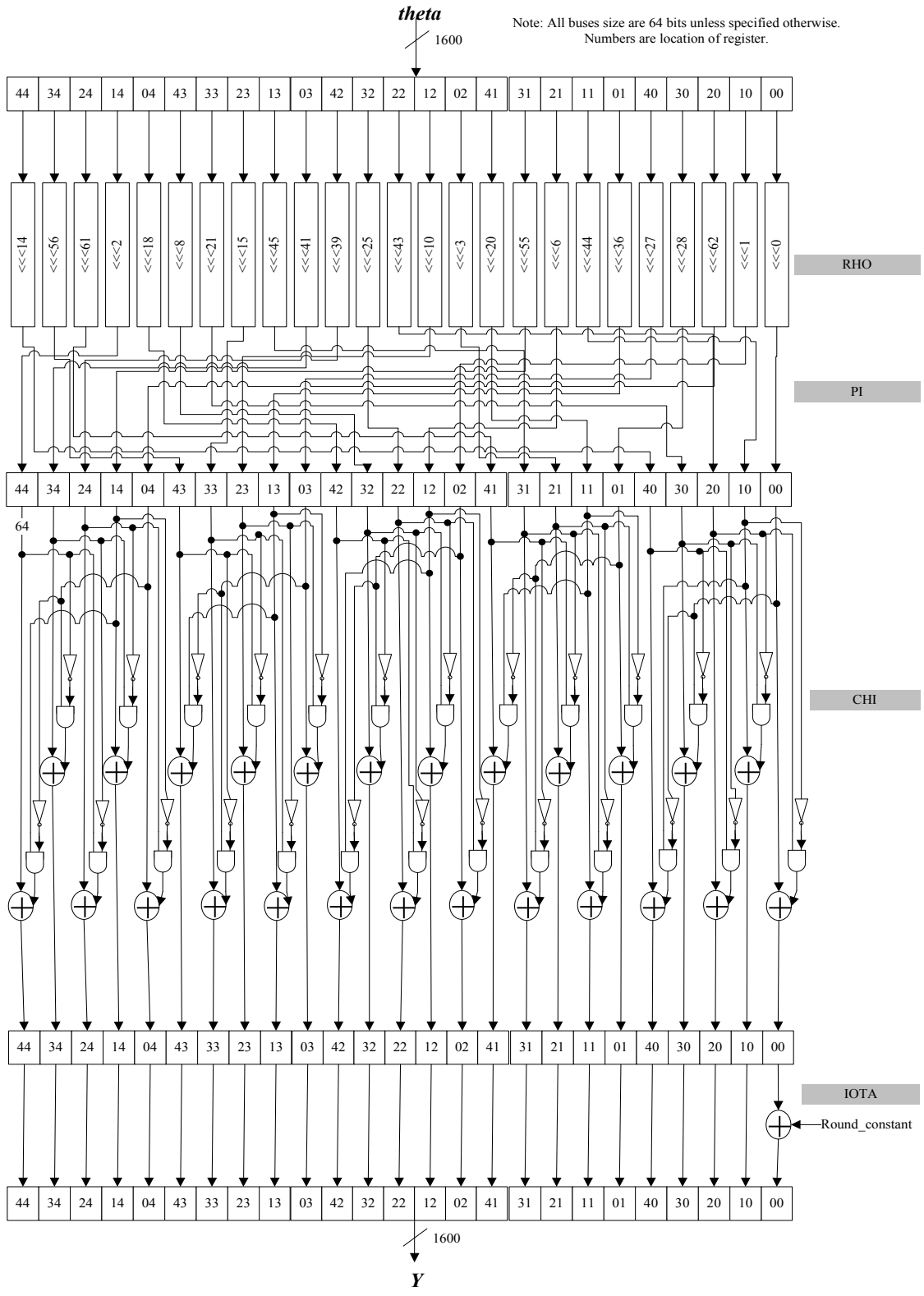

Figure 3.33: Keccak : Round Part 2

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.

3.12 Luffa

3.12.1 Block Diagram Description

The datapath of Luffa is shown in Figure 3.34. 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 3.35. The Galois field multiplication (x2), used in the MI unit, is also shown in Figure 3.1.

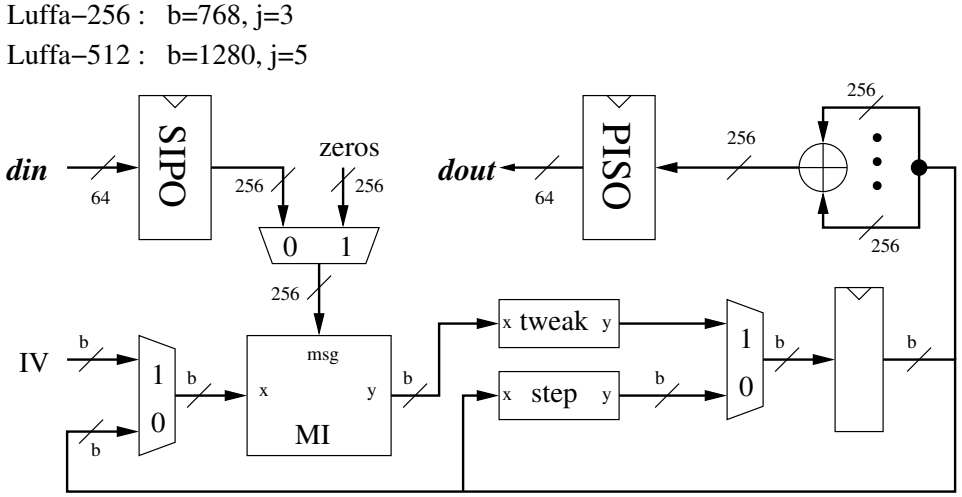

Figure 3.34: Luffa : Datapath

The state is then rotated wordwise using the Tweak operation shown in Figure 3.36. 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 3.37. The Step function consists of SubCrumb, MixWord and AddConstant operations. SubCrumb and MixWord are shown in Figures 3.38 and 3.39, 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 [30]. 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.

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 3.40. Since the finalization process only produces 256 bits at a time, the chain

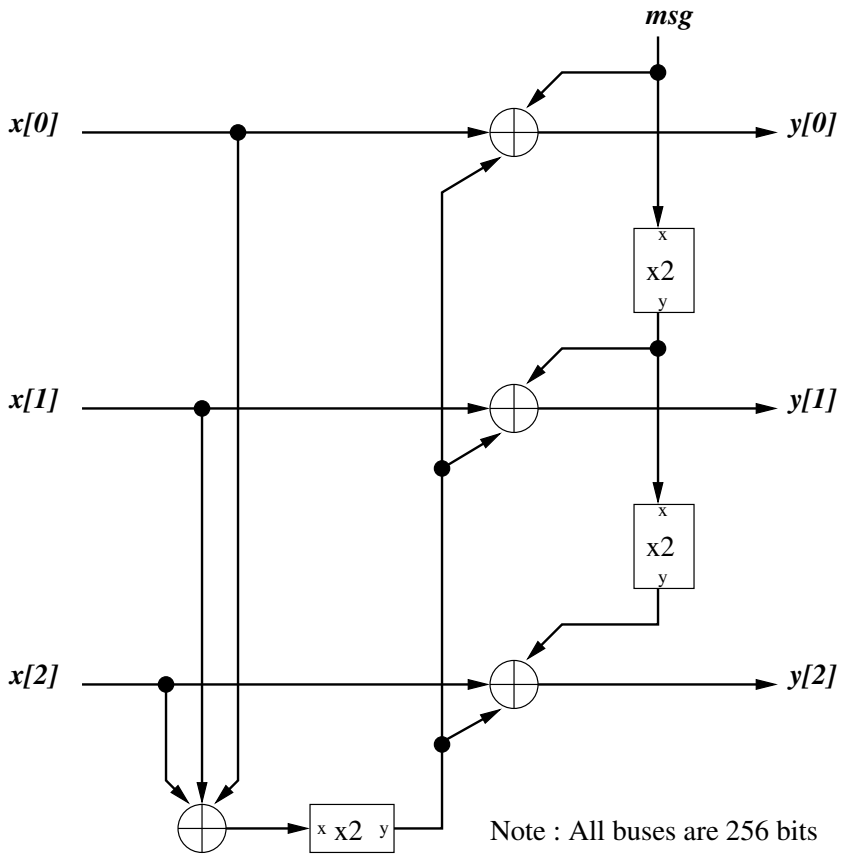

Figure 3.35: Luffa : Message Injection (256 bits)

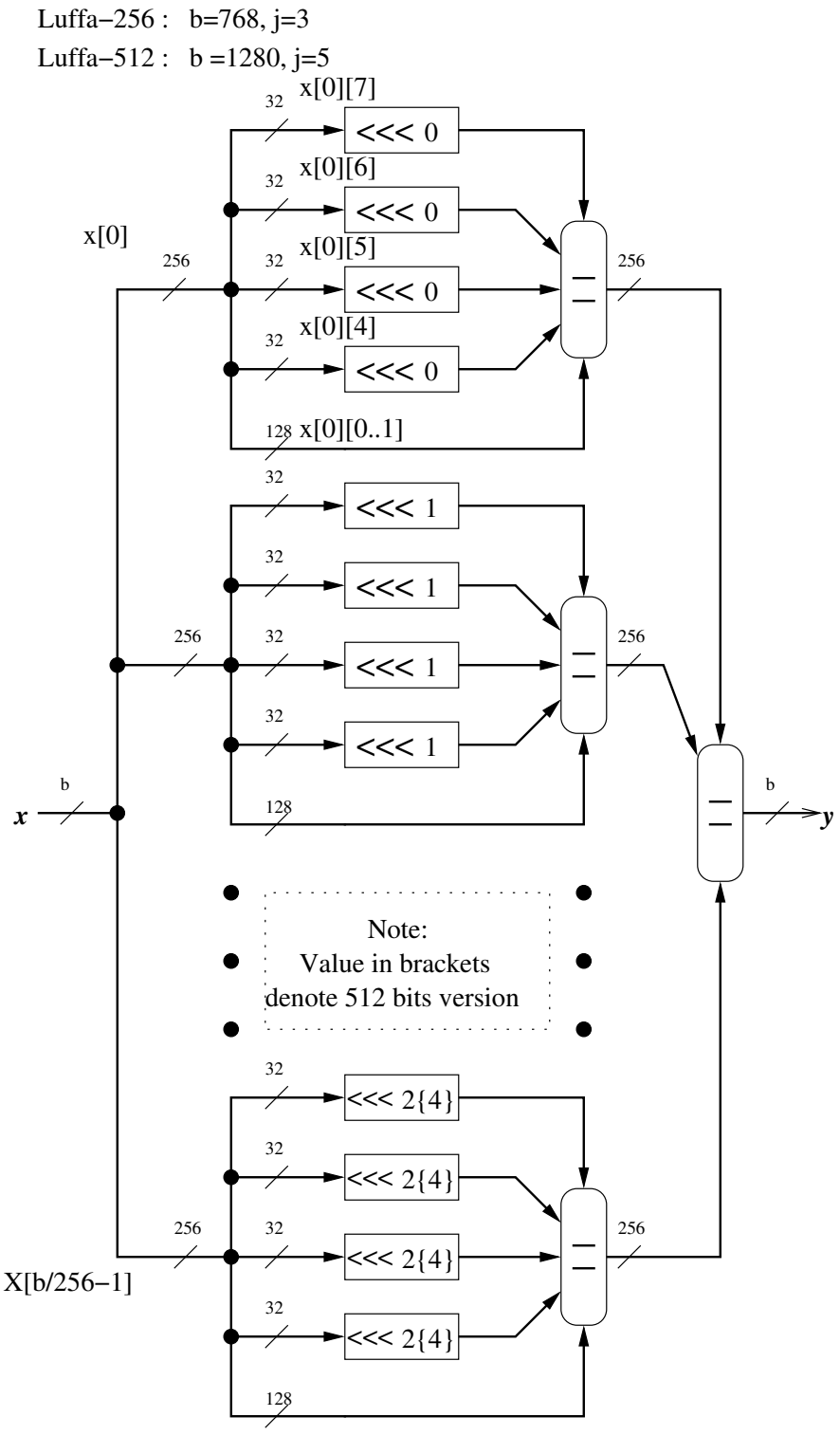

Figure 3.36: Luffa : Tweak

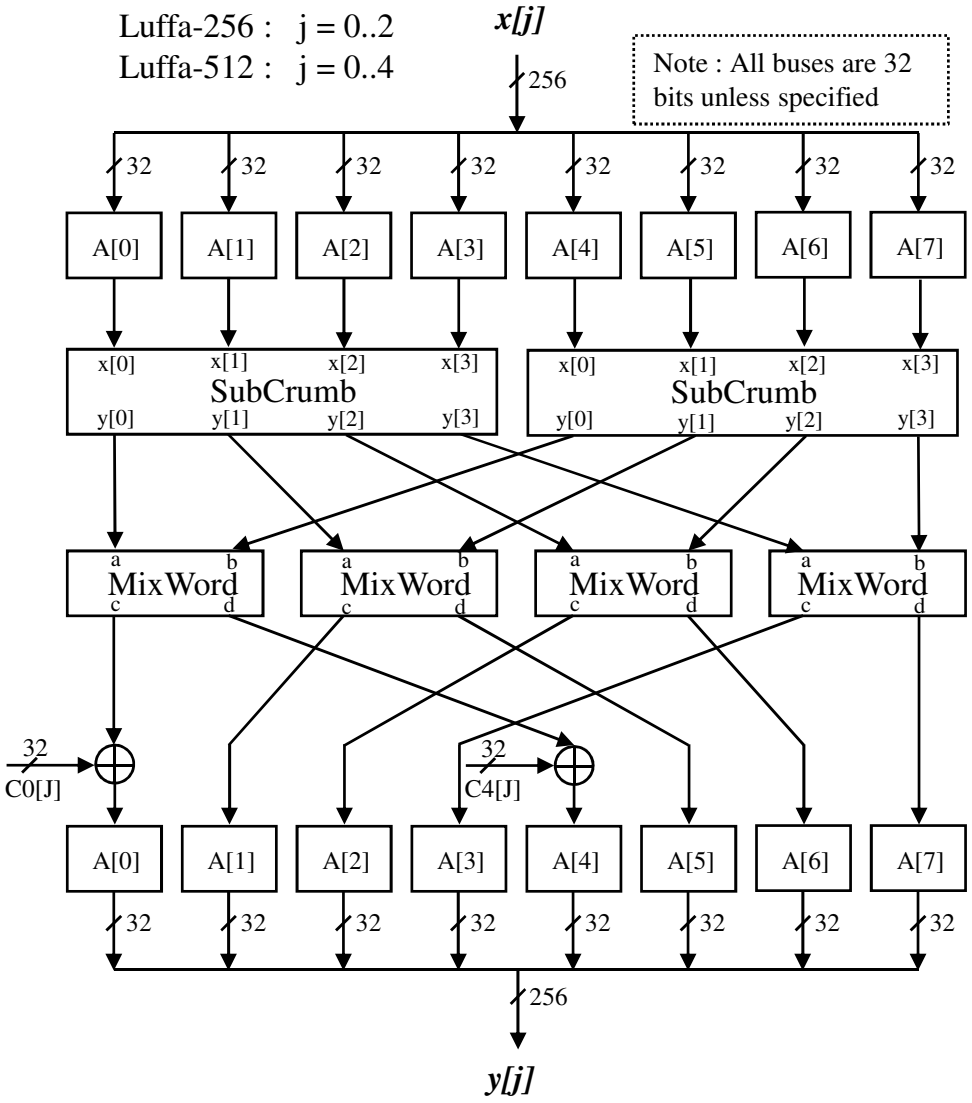

Figure 3.37: Luffa : Step function

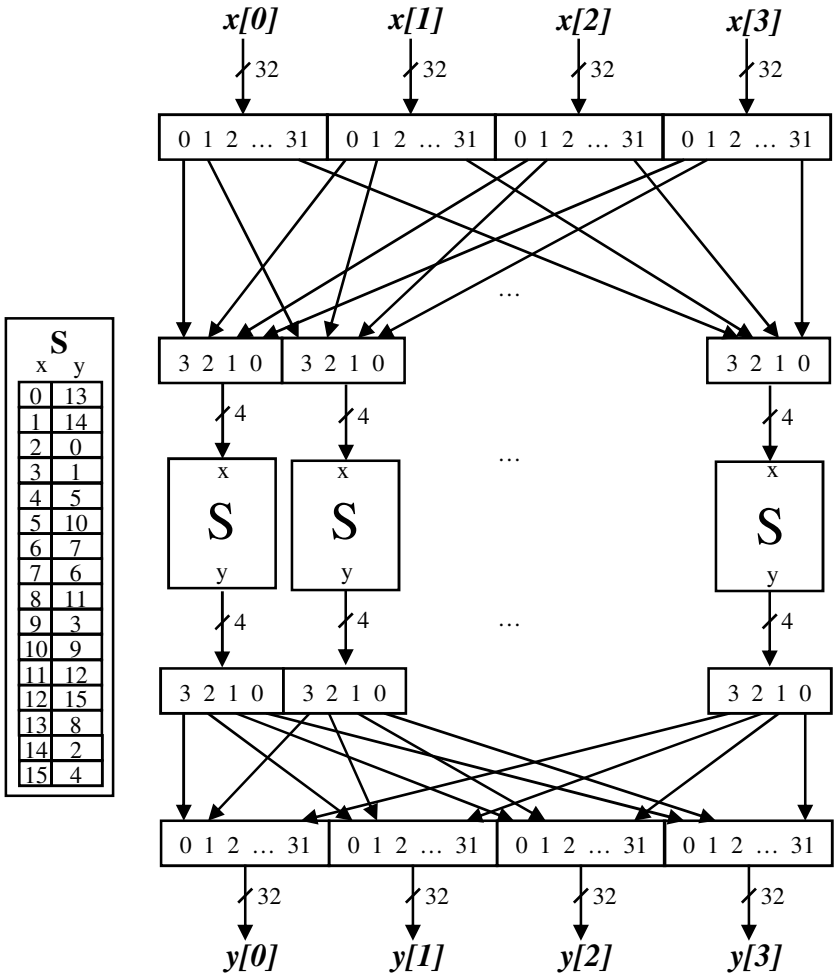

Figure 3.38: Luffa : SubCrumb

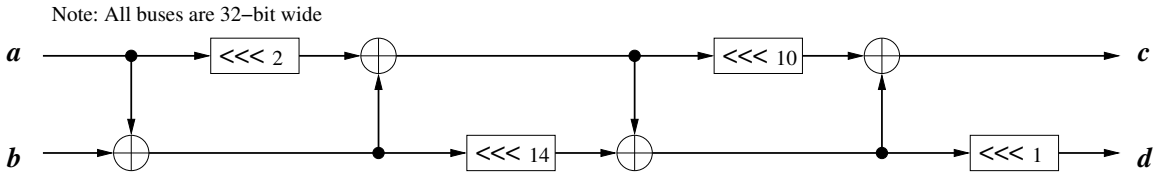

Figure 3.39: Luffa : Mix

value is hashed with another message block of value zero to produce the second half of a 512-bit hash value.

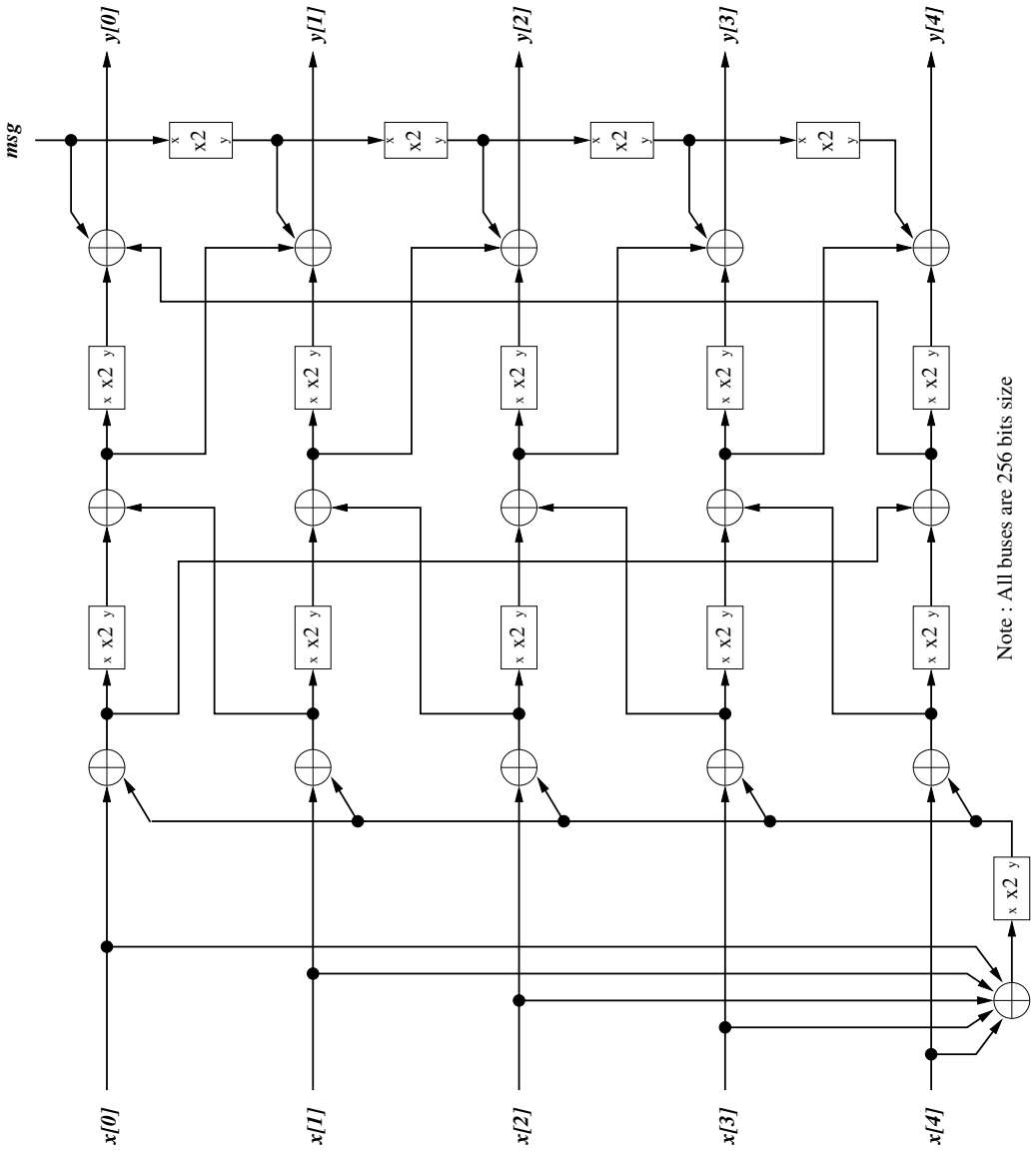

Figure 3.40: Luffa : Message Injection (512 bits)

3.13 SHA-2

3.13.1 Block Diagram Description

Our design of SHA-2 is based on [38]. A diagram of our SHA-2 circuit is shown in Figure 3.42. The detailed definitions of all SHA-2 operations shown in our diagram can be found in [31].

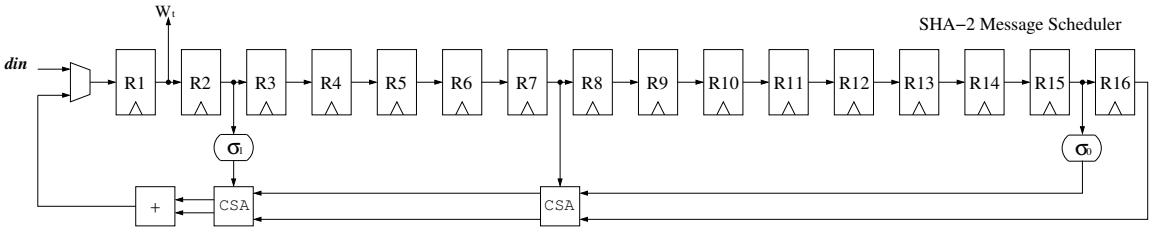

Figure 3.41: SHA2 : Message Scheduler

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.

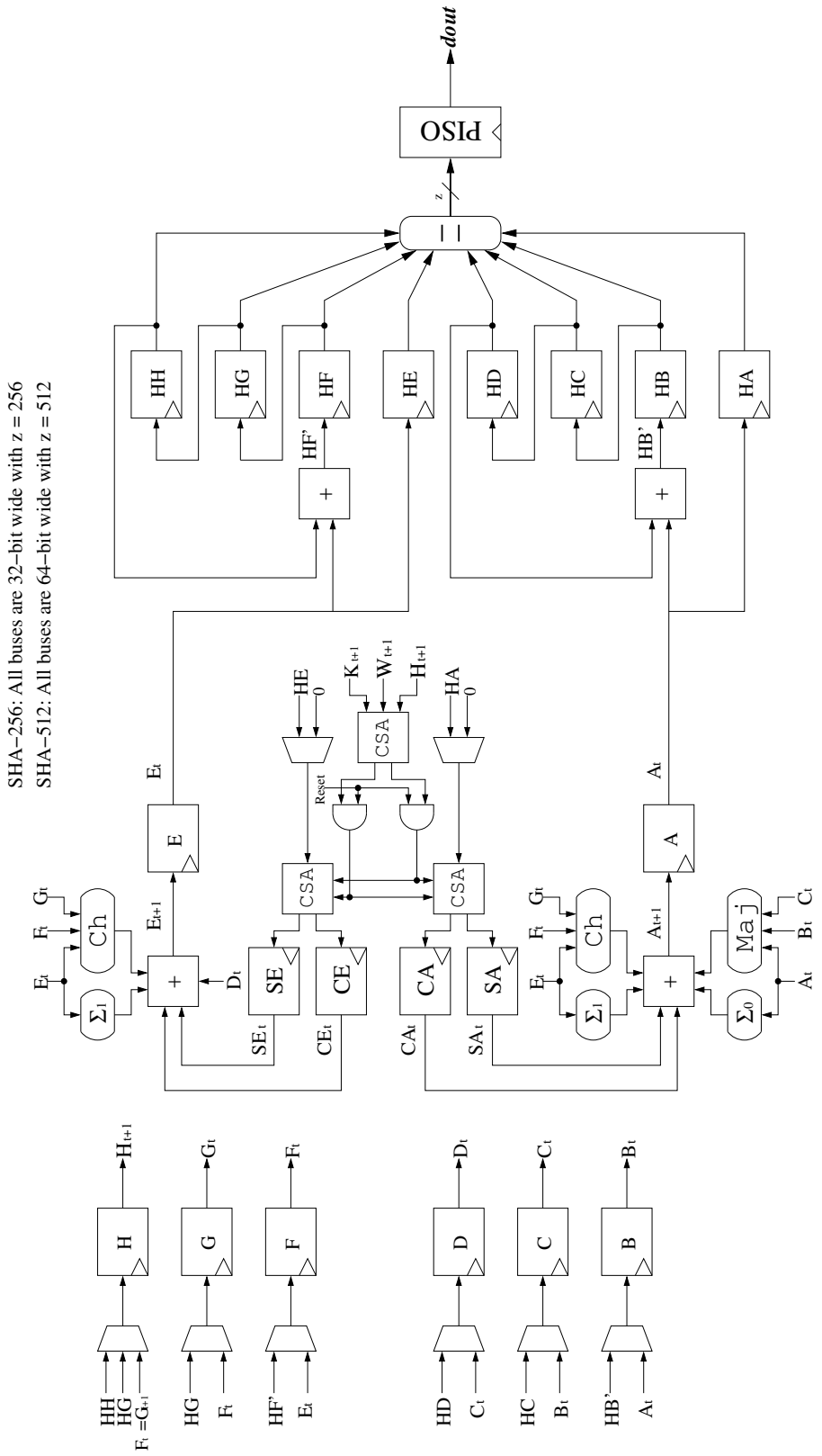

Figure 3.42: SHA2 : Datapath

3.14 Shabal

3.14.1 Block Diagram Description

The block diagram of Shabal is shown in Figure 3.43. 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:

$$\begin{aligned} A \leftarrow &\quad A_{iv}[0..1] \oplus w[0..1] \mid\mid A_{iv}[2..11] \\ B \leftarrow &\quad ((B_{iv} + M_w) <<< 17) \\ C \leftarrow &\quad C_{iv} \\ M \leftarrow &\quad M_w \end{aligned}$$

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:

$$\begin{aligned} A \leftarrow &\quad ap[0..1] \oplus w[0..1] \mid\mid ap[2..11] \\ B \leftarrow &\quad (((C - M) + M_w) <<< 17) \\ C \leftarrow &\quad B \\ M \leftarrow &\quad M_w \end{aligned}$$

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

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 3.44. For SHAvite-3-256, a block of message contains 512 bits, and 128 bits are loaded into

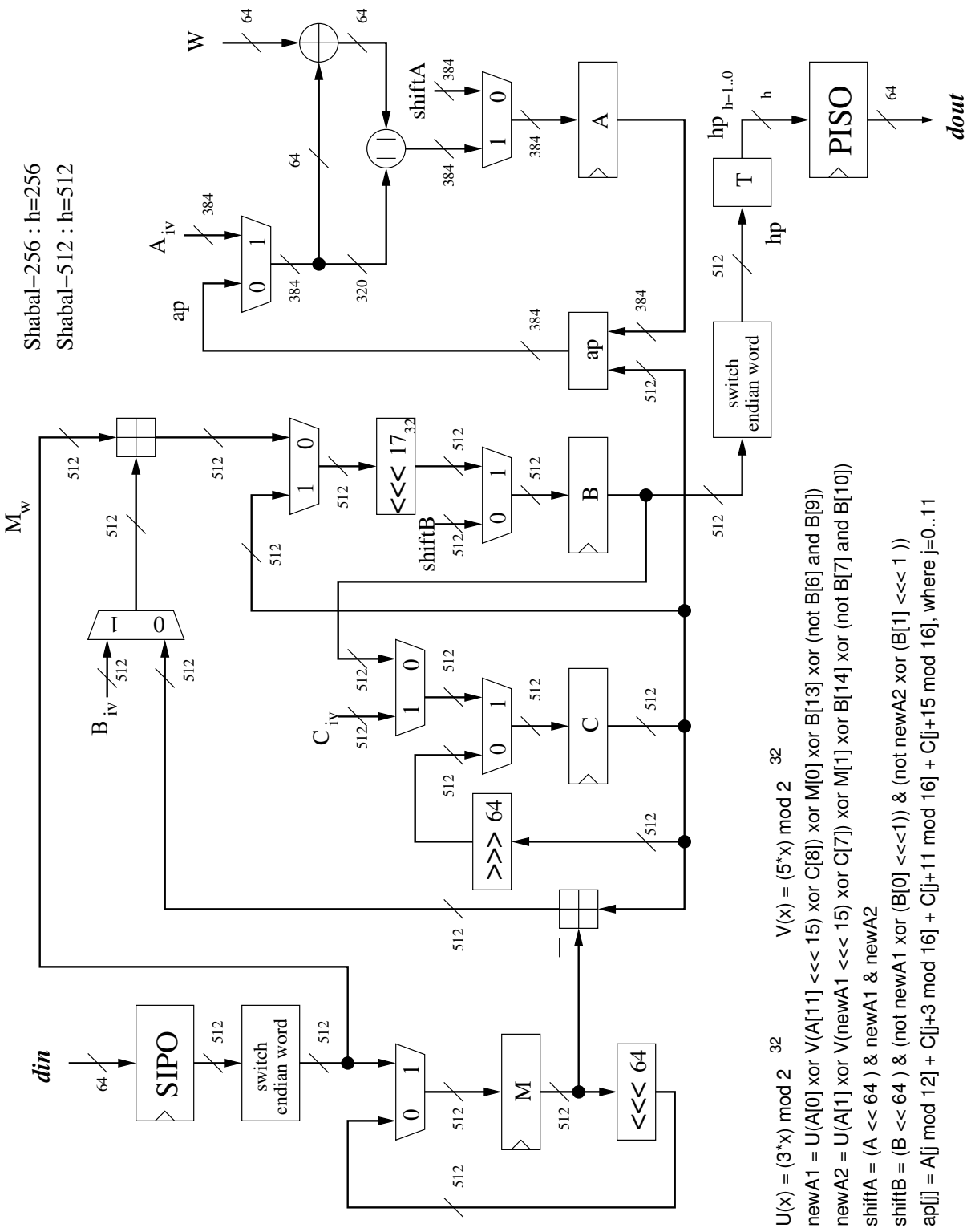

Figure 3.43: Shabal : Datapath

the key generation unit at a time. For every message, the state register S and the chain value CV 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.

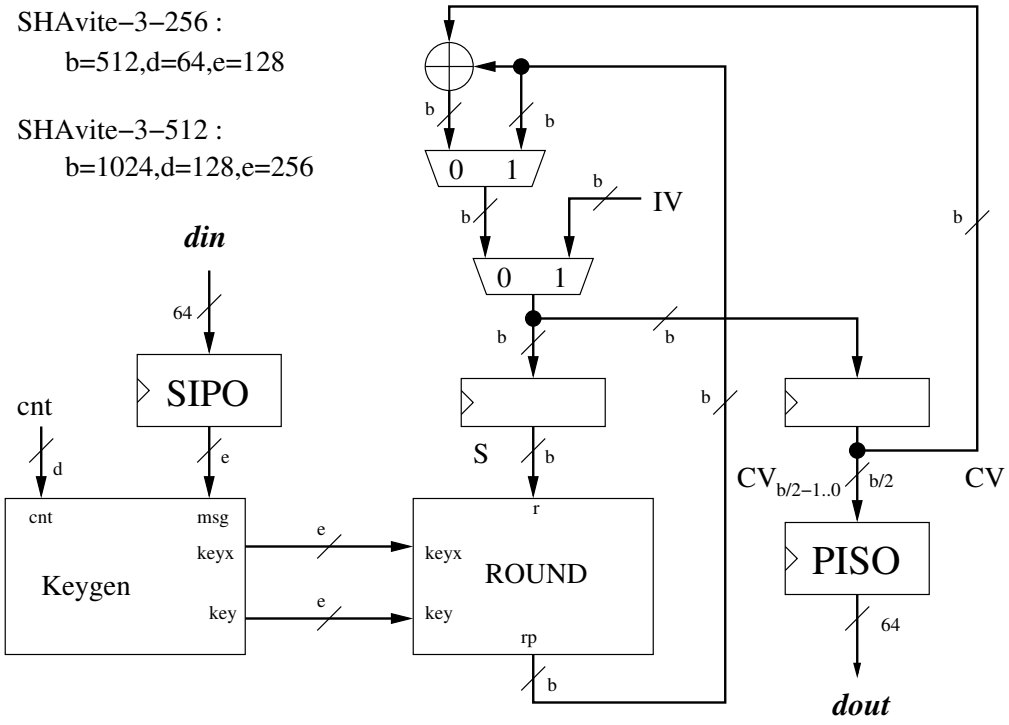

Figure 3.44: SHAvite-3: Datapath

The SHAvite-3 ROUND unit is shown in Figure 3.45. 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 3.46. Key and/or keyx are generated for each main round using this circuit.

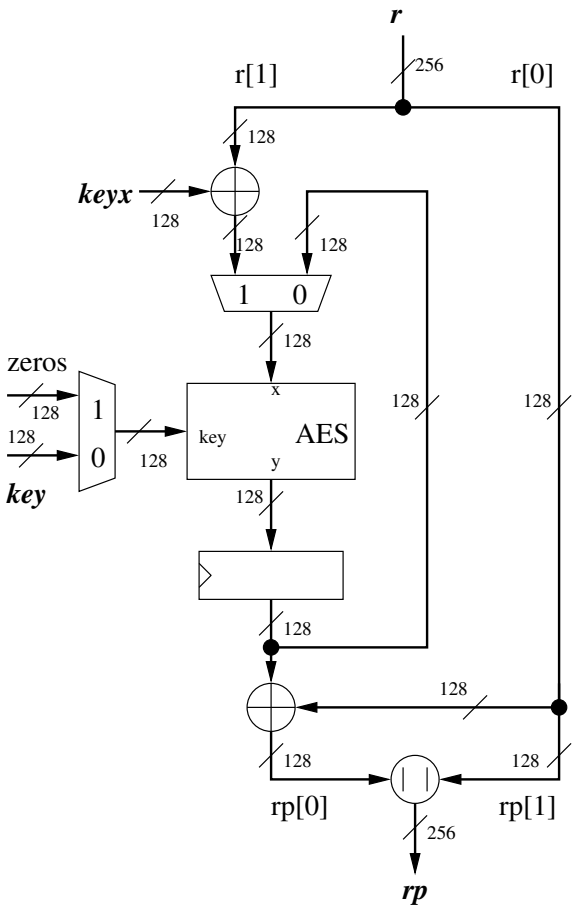

Figure 3.45: SHAvite-3: Round (256 bits)

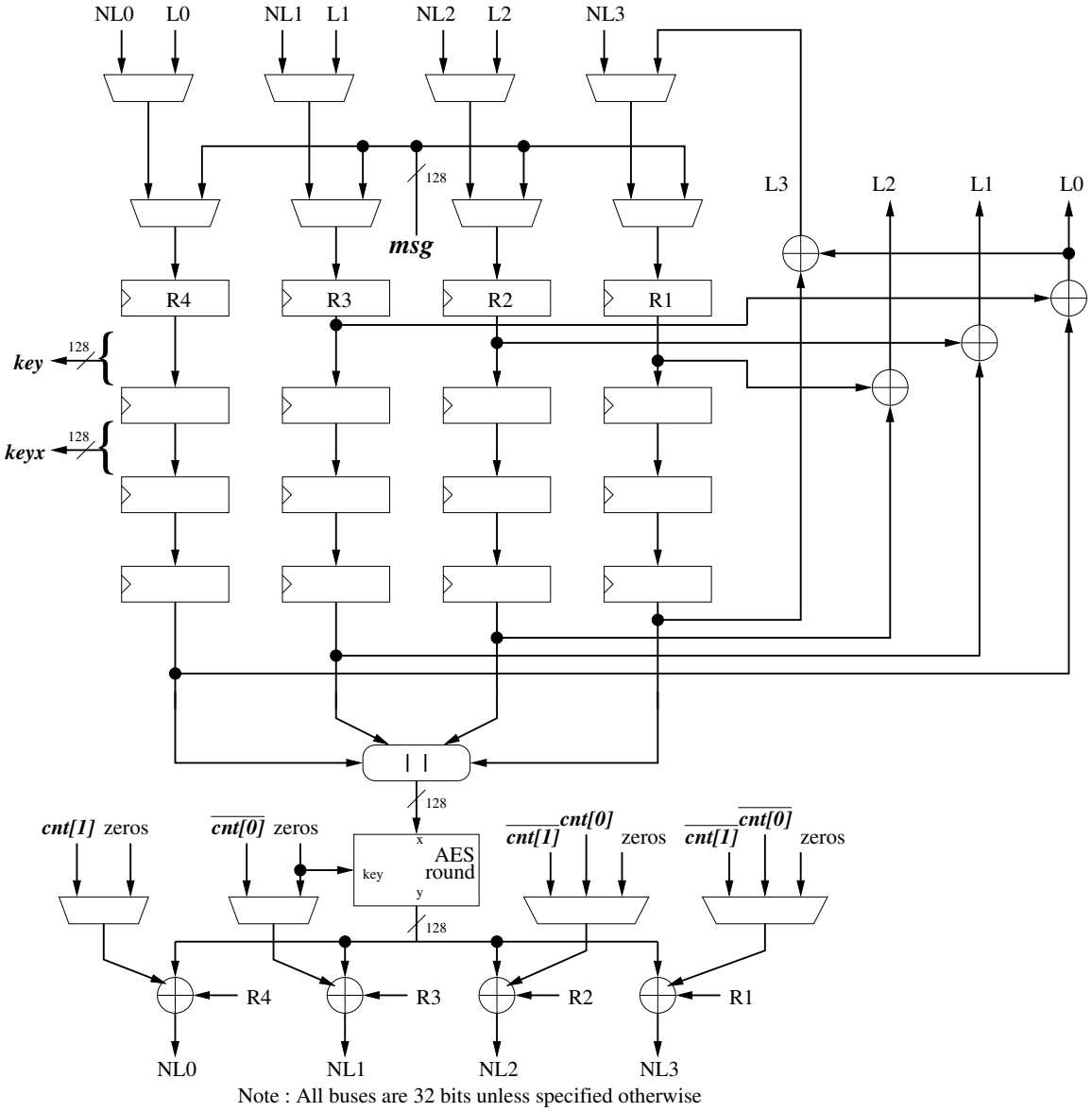

Figure 3.46: SHAvite-3: Keygen (256 bits)

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 3.47 illustrates changes in the ROUND unit for SHAvite-3-512. The same operation as $Round_256$ is performed with the exception that the number of internal rounds is increased from 3 to 4. Figure 3.48 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.

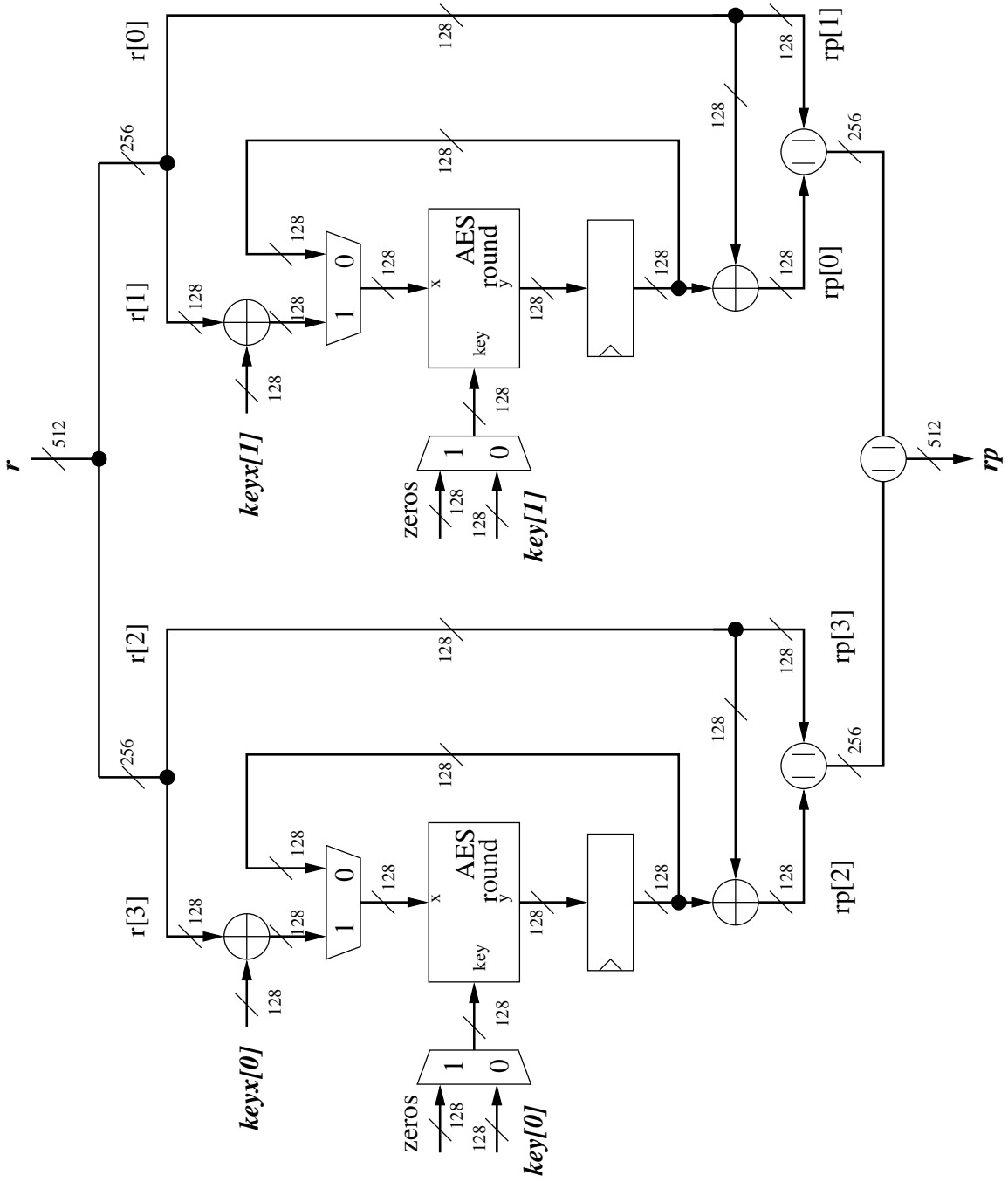

Figure 3.47: SHAvite-3 : Round (512 bits)

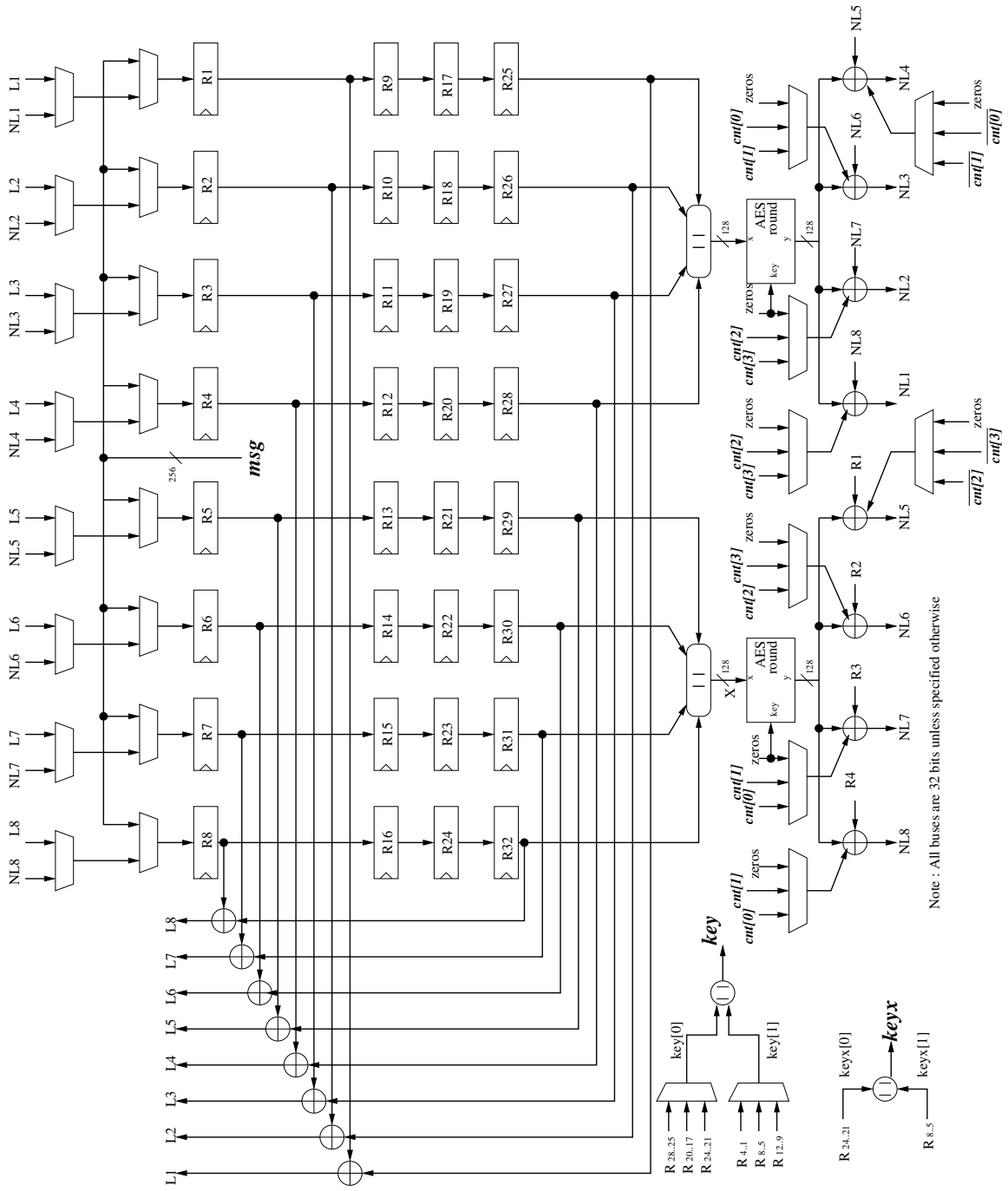

Figure 3.48: SHAvite-3 : Keygen (512 bits)

3.16 SIMD

3.16.1 Block Diagram Description

The block diagram of SIMD is shown in Figure 3.49. 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 rounds). 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 DFT stage as its basic building block. First, each byte of an input is zero-extended to 9 bits. Then, these 9-bit blocks are inserted into the DFT stage with its respective twiddle factor as an input. The twiddle factor is chosen based on the DFT 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. The value of *point* is equal to $2^{stage+1}$, where *stage* is the number of the current DFT stage executed by the unit. The variable *pts* represents the maximum value of the variable *point*. For SIMD-256, *pts* is always equal to 128 as there are seven stages of DFT with 128 points. For SIMD-512, *pts* is always equal to 256.

The main building block of NTT, referred to as DFT stage, is shown in Figure 3.51. This unit is built using several repetitions of the butterfly unit. 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

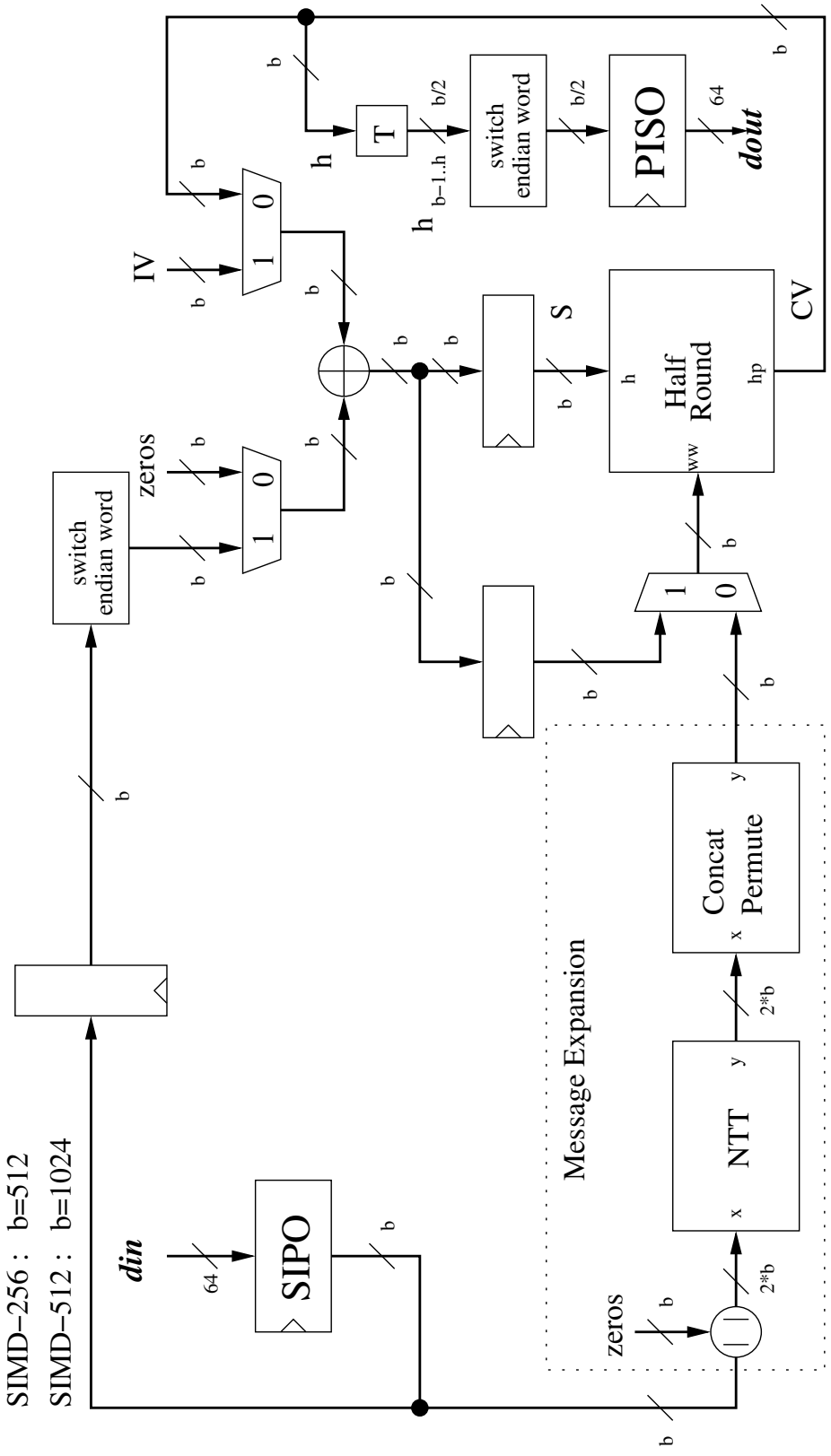

Figure 3.49: SIMD : Datapath

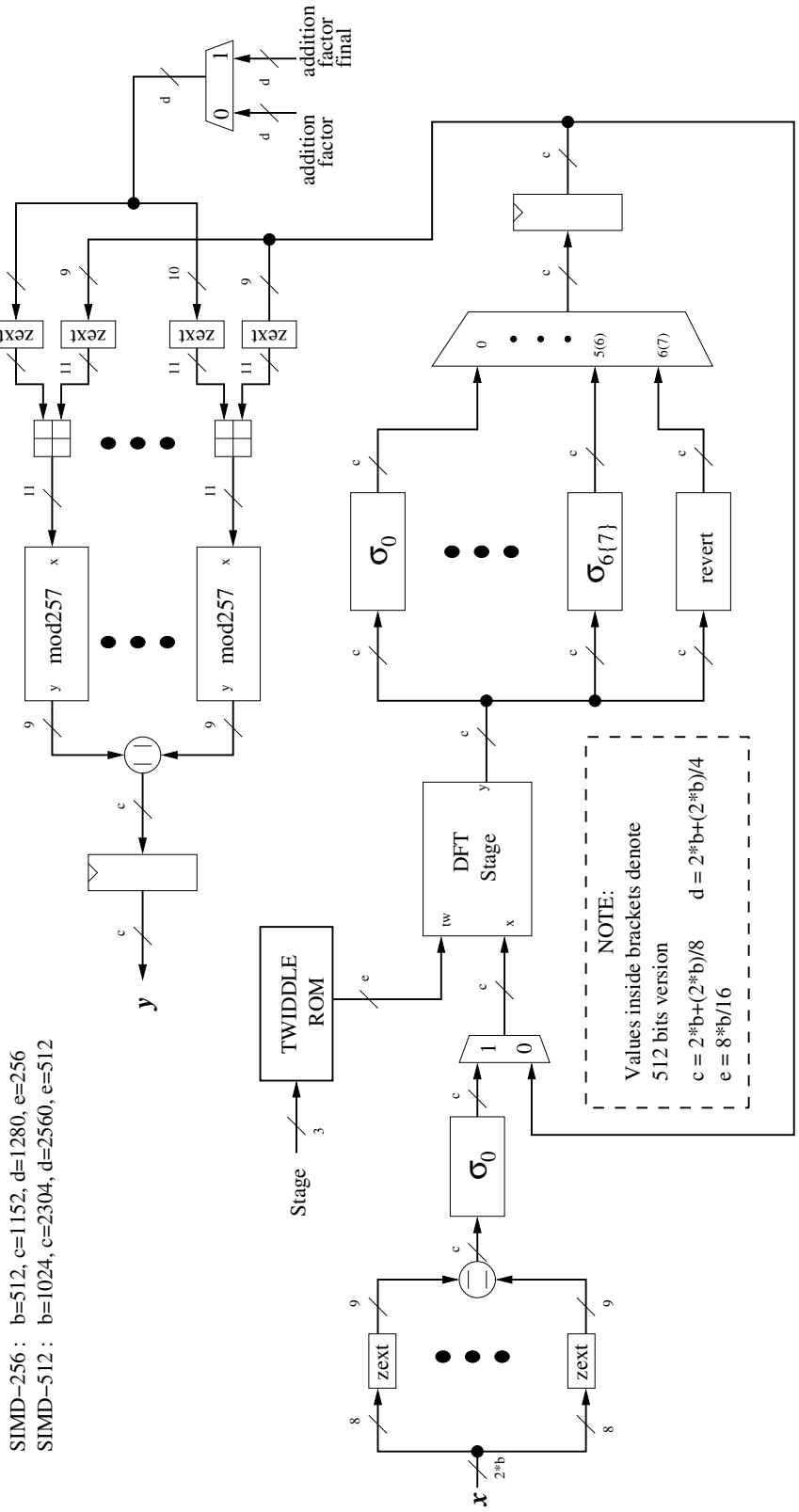

Figure 3.50: SIMD : NTT

ensure that there is no bit growth. The Modulo 257 unit is shown in Figure 3.52.

Since there are 7 stages for a 128-point DFT, it is necessary to permute an input before entering each next stage, so that the inputs are applied in the correct order. The permutation can be derived from any diagrams illustrating the operation of FFT (many of such diagrams are available on the internet). The final operation of the NTT unit involves an addition between the output of the 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 is equal to 128 for SIMD-256 and 256 for SIMD-512:

```
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;

  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 [34]. A diagram of Concat Permute (CP) is shown in Figure 3.53. 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:

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

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

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

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

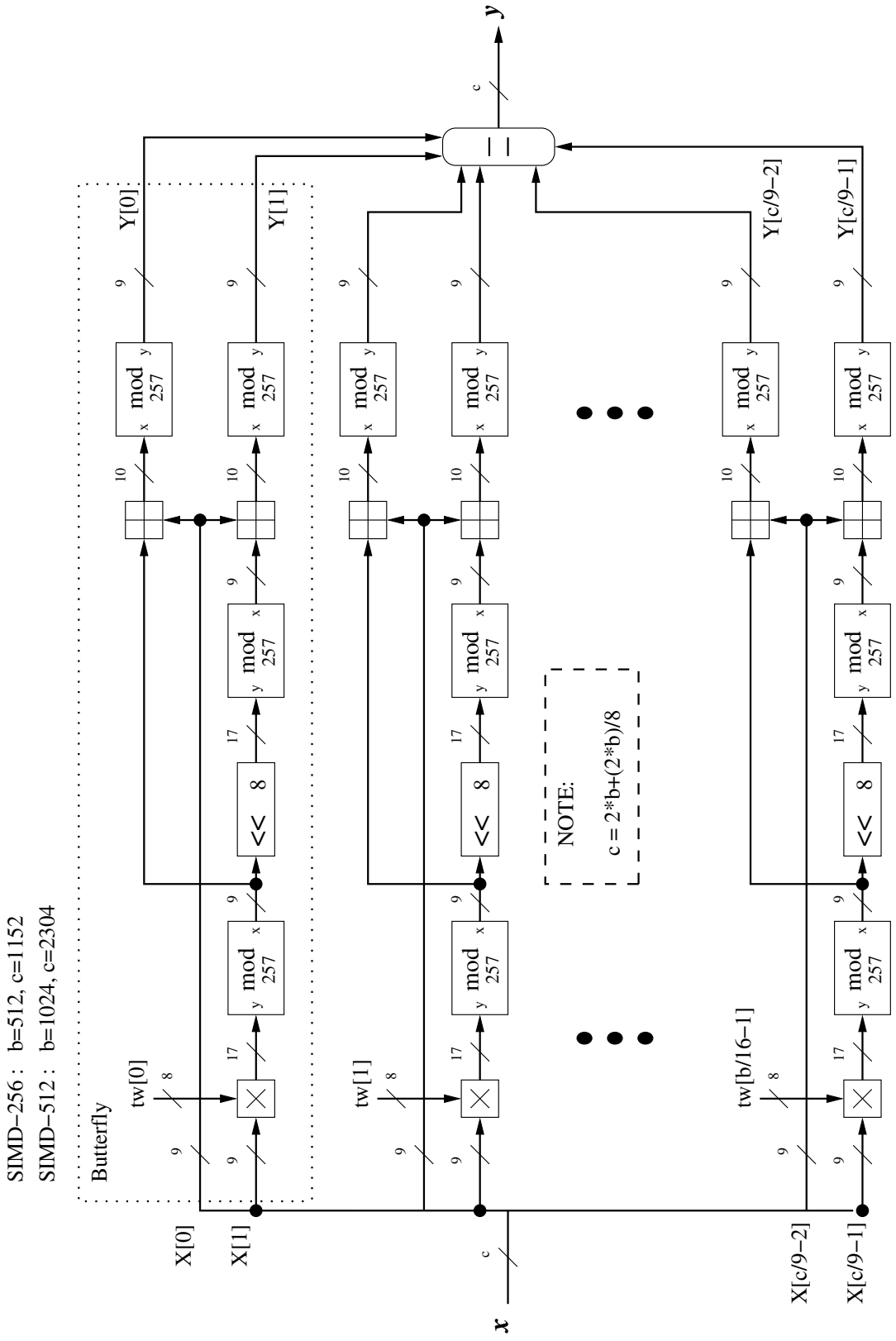

Figure 3.51: SIMD : DFT stage

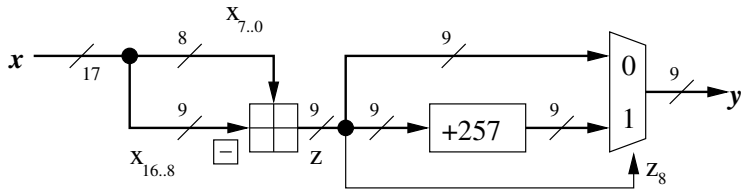

Figure 3.52: SIMD : Modulo 257

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

half and entered into Lift module shown in Figure 3.54. 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.

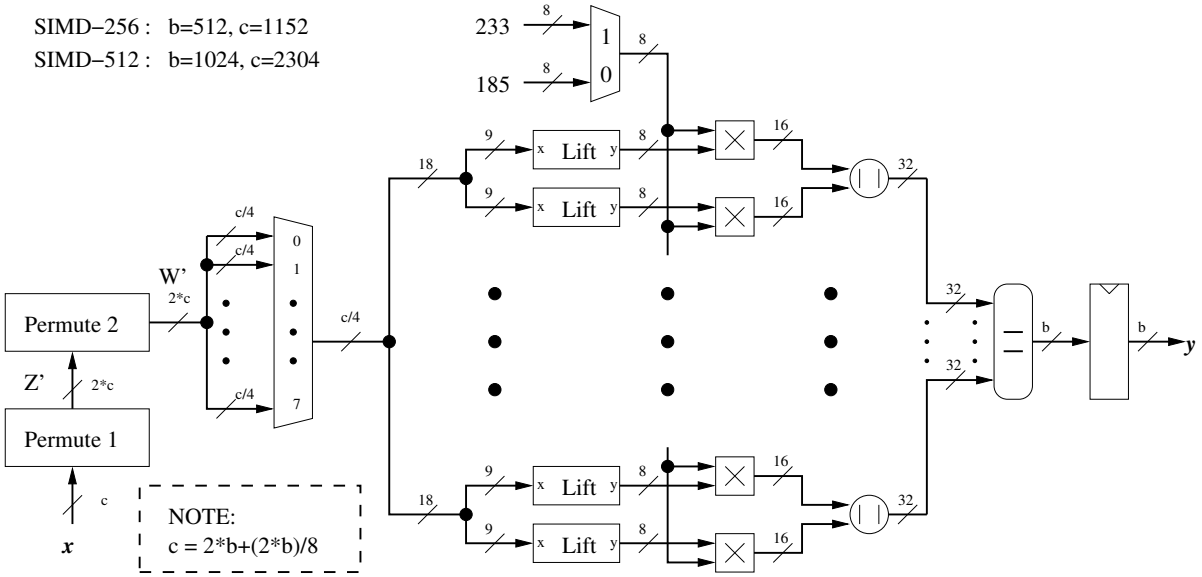

Figure 3.53: SIMD : Concatenate and Permute (CP)

The core operation of our SIMD's design is the Half Round module. This module is

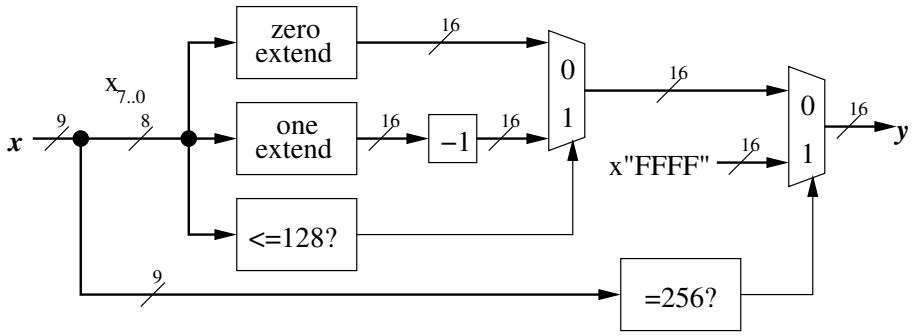

Figure 3.54: SIMD : Lift

equivalent to four steps of the SIMD round. A block diagram of the Half Round operation is shown in Figure 3.55. Half Round is based of 16 QS units with *quarterstep* as its core. *quarterstep* is shown in Figure 3.56. 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:

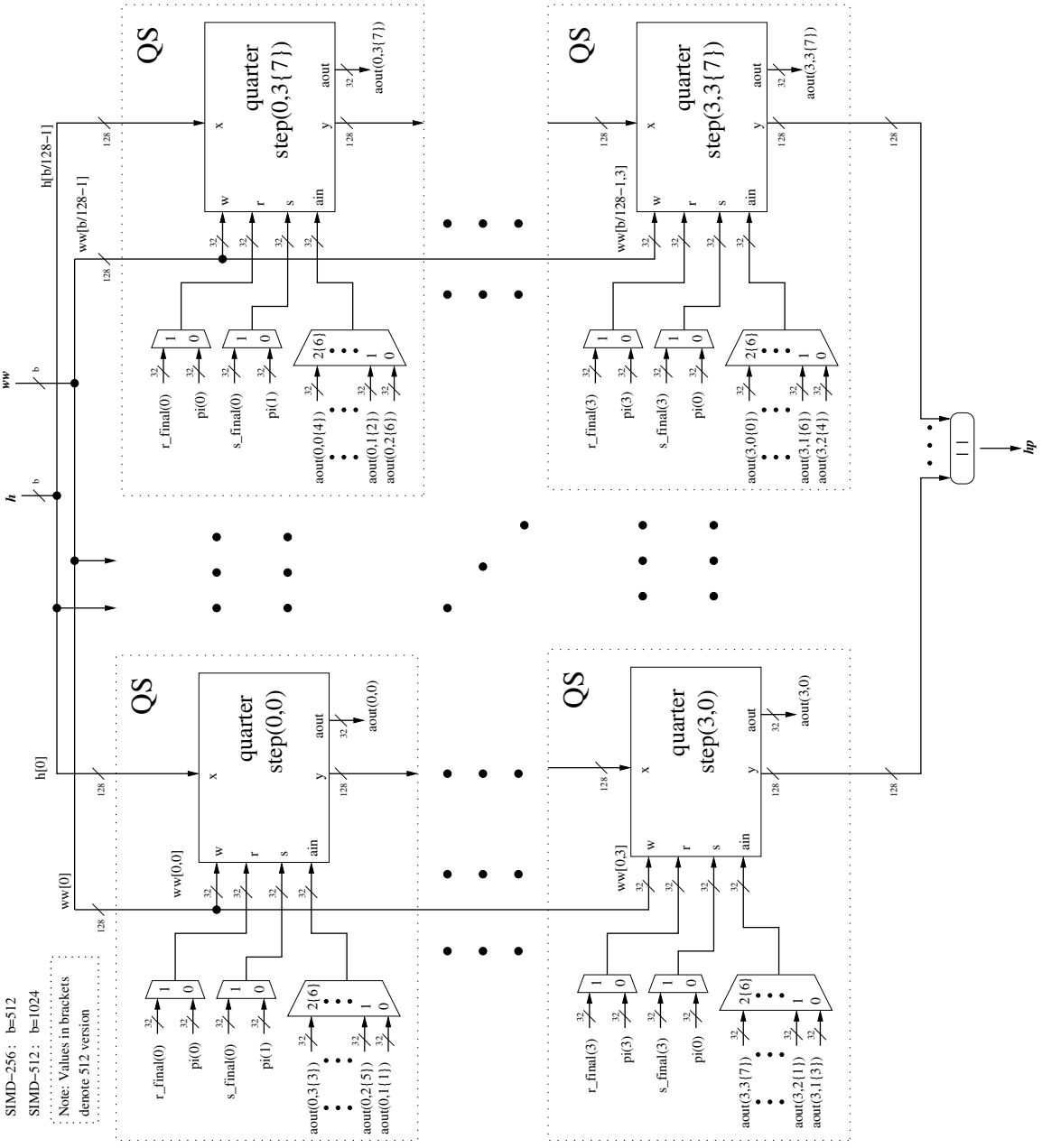

Figure 3.55: SIMD : Half Round

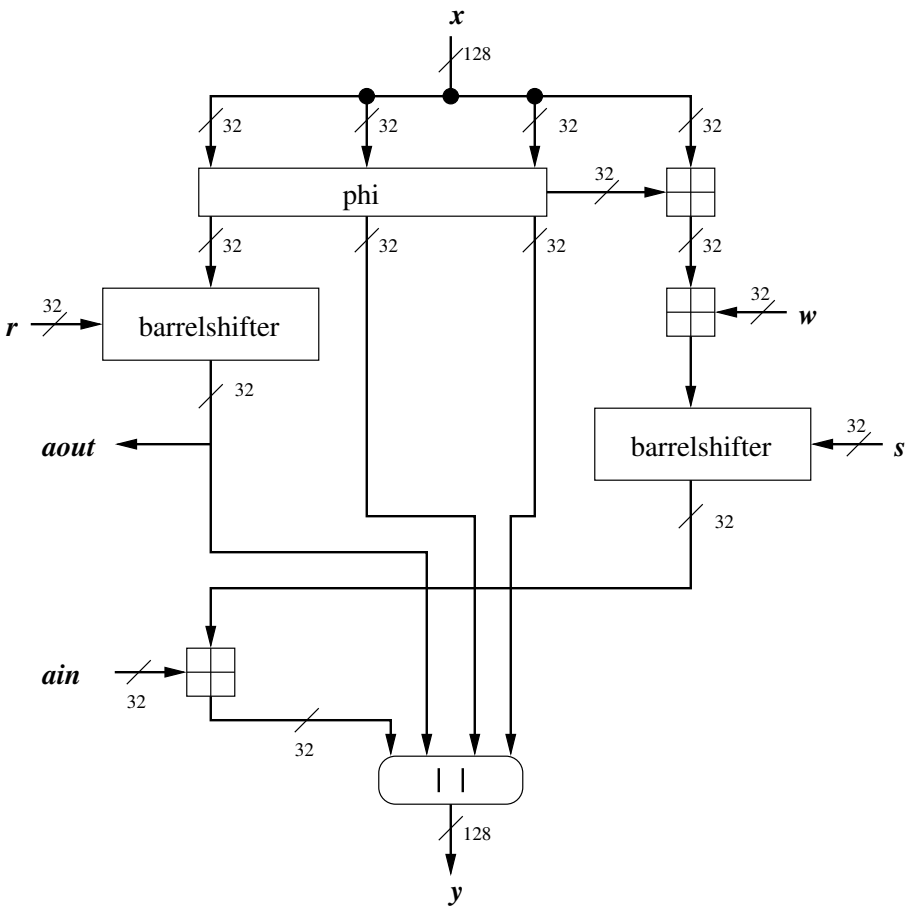

Figure 3.56: SIMD : Quarter Step

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

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:

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

3.17 Skein

3.17.1 Block Diagram Description

The datapath of Skein is shown in Figure 3.57. This diagram is based on the Skein-512-256 construction. The datapath of Skein can be separated into two main parts, key generation and the Skein's round. The round includes a layer of 64-bit additions and 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 eight 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.

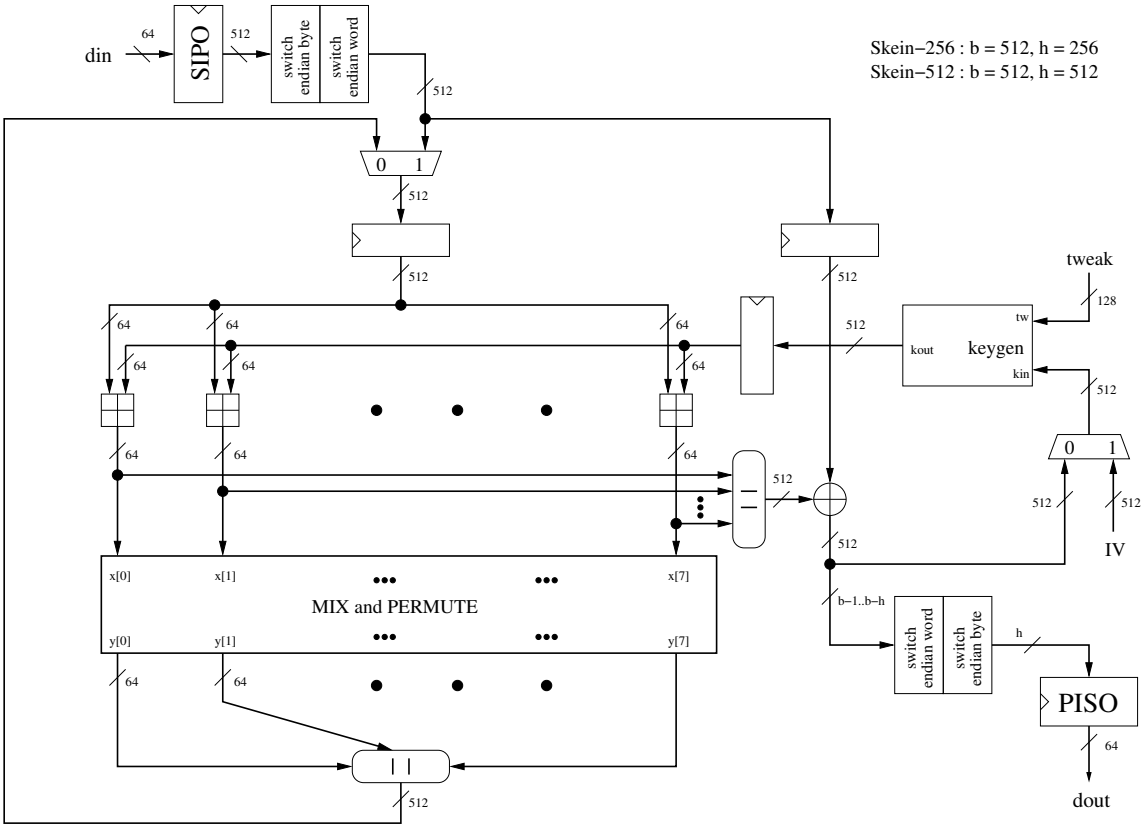

Figure 3.57: Skein : Datapath

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 at the beginning of the message. A tweak is controlled by the controller. Its full specification can be found under Section 3.4 of [35]. In Figure 3.58, a key generation unit for our design is shown. s is the subkey counter. It gets reset for every new message block.

In Figure 3.59, a 4-times unrolled MIX and PERMUTE unit is shown. This unit is based on 16 instantiations of the MIX operation. The MIX operation is shown in Figure 3.60. 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.

Table 3.15: Skein: Rotation Constants, N_w is the number of words, N_w=8 in the implemented variants of the function

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

x	0	1	2	3	4	5	6	7
y	2	1	4	7	6	5	0	3

3.17.2 256 vs. 512 Variant Differences

Skein-512 as submitted to the SHA-3 contest is based on Skein-512-512. The same design as used in Skein-256 is applied to Skein-512, with the exception of the output register (PISO), where 512-bit instead of 256-bit register is used.

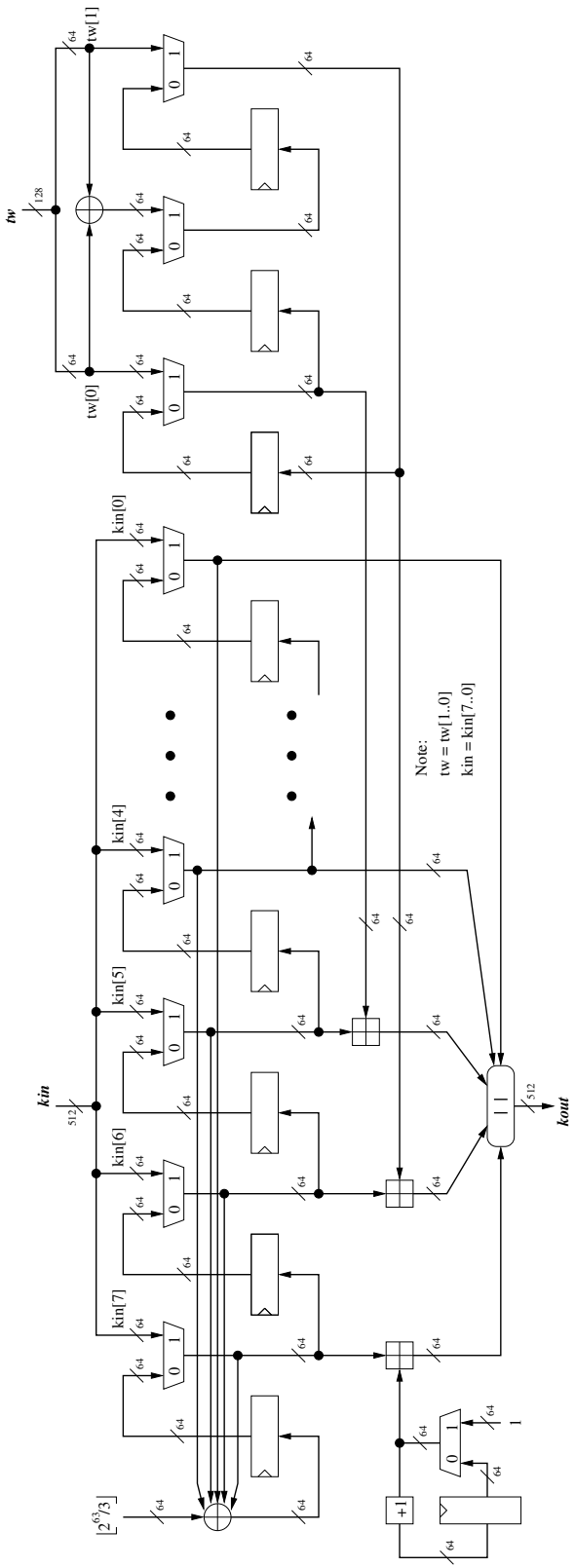

Figure 3.58: Skein : Key Generation

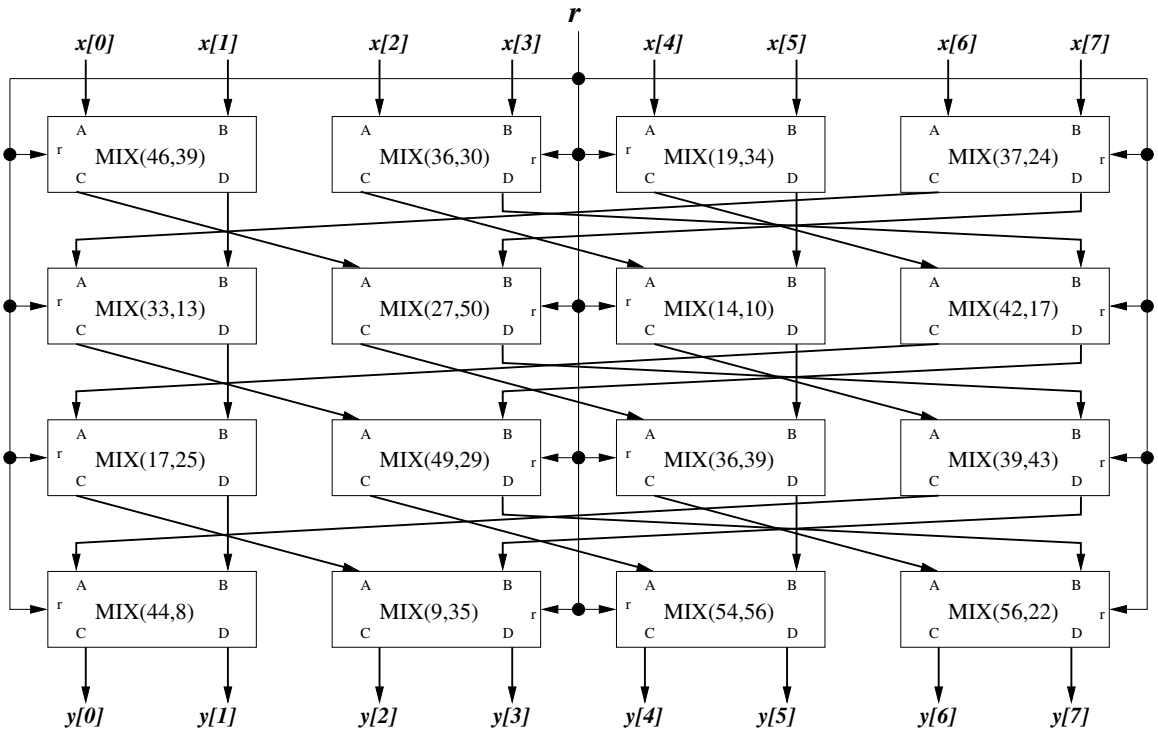

Figure 3.59: Skein : Round

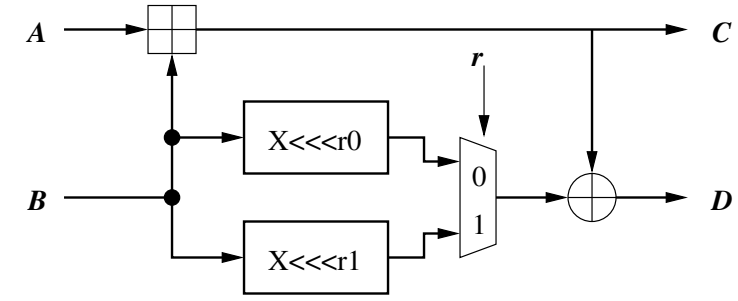

Figure 3.60: Skein : Mix

Chapter 4

Design Summary and Results

4.1 Design Summary

The major parameters of both hash function variants (with a 256-bit and a 512-bit output) are summarized in Table 4.1.

In Table 4.2, we provide the I/O Data Bus Widths, and the exact formulas for the Hash Function Execution Time (in clock cycles) and Throughput (in Mbits/s) for our designs of all SHA-3 candidates and the current standard, SHA-2. The equations are derived from the analysis of block diagrams of the respective designs, and have been confirmed through simulation. All numerical values of timing parameters presented in this report are based on these equations.

The I/O Data Bus Width, w, is a feature of our interface described in Section 2.3. It is the size of the data buses, din and dout, used to connect the SHA core with external logic (such as Input and Output FIFOs). The parameter w has been chosen to be equal to 64, unless there was a compelling reason to make it smaller. The value of 64 was considered to be small enough so that the SHA cores fit in all investigated FPGAs (even the smallest ones) without exceeding the maximum number of user pins. At the same time, setting this value to any smaller power of two (e.g., 32) would increase the time necessary to load input data from the input FIFO and store the hash value to the output FIFO. In some cases, it would also mean that the time necessary for processing a single block of data would be smaller than the time of loading the next block of data, which would decrease the overall throughput. The only exceptions are Fugue-256, Hamsi-256, and Fugue-512, for which we choose w=32, because they all have block size equal to 32 bits, and thus cannot be sped up by using a wider I/O data bus. Similarly, SHA-256 can start processing data after receiving just one 32-bit word, and cannot be easily sped-up by using a wider input data bus. In case of BMW, an additional faster i/o clock was used on top of the main clock shown in Fig. 2.1a. This faster clock is driving input/output interfaces of the SHA core, as well as surrounding FIFOs. The ratio of the i/o clock frequency to the main clock frequency was

selected to be 8 for BMW-256 and 16 for BMW-512, so the entire block of message (512 bits for BMW-256 and 1024 for BMW-512) can be loaded in a single clock cycle of the main clock (8 and 16 cycles of the fast i/o clock, respectively).

The next column of Table 4.2 contains the detailed formulas for the number of clock cycles necessary to hash N blocks of the message after padding. The formulas include the time necessary to load the message length, load input data from the FIFO, perform all necessary initializations, perform main processing, perform all required finalizations, and then send the result to the output FIFO. Finally, the last column (per each hash function variant) contains the formula for the circuit throughput for long messages as defined by Equation (2.1).

Table 4.1: 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	512	512	72	64	512	512	72	64	1	1
SHA-2	**256**	**512**	64	**32**	**512**	**1024**	**80**	**64**	**2**	**1.6**

4.2 Relative Performance of the 512 and 256-bit Variants of the SHA-3 Candidates

In the last two columns of Table 4.1, we provide the first rough approximation of the predicted area ratio (512 vs. 256-bit variant) and the predicted throughput ratio (512 vs. 256-bit variant). In general, the area of the circuit optimized for the maximum throughput to area ratio 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: 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+21·N+4	512/(21·T)	64	2+**16**+29·N+**8**	**1024**/(**29**·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	3+**16**+29·N+**8**	**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	3+**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+25·N+3·49+**8**	512/(25·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+19·N+4	512/(19·T)	64	2+**8**+19·N+**8**	512/(19·T)
SHA-256	32	2+1+65·N+8	512/(65·T)	64	2+1+**81**·N+**8**	**1024**/(81·T)

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

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 under the assumption that the critical path, and thus the clock period, are similar in both variants.

$$Predicted\_Throughput\_Ratio_{512/256} = \frac{Thr_{512}}{Thr_{256}} = \frac{\frac{Block\_size_{512}}{Round\_no_{512}}}{\frac{Block\_size_{256}}{Round\_no_{256}}}. \quad (4.2)$$

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.

Based on the above 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, Skein.*
- Group 2: area and throughput both double: *BMW, SIMD.*
- 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.*

From the point of view of the throughput to area ratio, Groups 1 and 2 are the best, followed by Groups 3, 4, and 5, and ending with the Group 6, with the worst trend. Among the Groups 1 and 2, belonging to the Group 2 is less desirable, especially for the algorithms that already take significant area for a 256-bit variant, such as BMW and SIMD.

In Table 4.3, we report ratios of the two major performance measures for a 512-bit variant vs. a 256-bit variant (namely, Area_ratio = Area(512)/Area(256) and Thr_ratio = Thr(512)/Thr(256). Both actual (A) and predicted (P) values of the respective ratios are reported, together with the relative difference (RD), and major reasons for this difference. The actual values (A) are averaged (using geometric mean) over all seven FPGA families. This table demonstrates a relatively good agreement between our predictions and actual experimental results.

4.3 Results

In Tables 4.4 and 4.5, 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.

In Tables 4.6 and 4.7, the absolute results obtained for our implementations of the current standard SHA-2 are summarized. The results are repeated across seven selected

Table 4.3: Ratio of the respective performance measures (Area and Throughput) for a 512-bit variant vs. 256-bit variant. Notation: A – actual ratio (averaged, using geometric mean, over all 7 FPGA families), P – predicted ratio (based on Equations 4.1 and 4.2), RD – relative difference in % ((A-P)/P*100%).

	Area ratio (512 vs. 256 bit variant)				**Throughput ratio (512 vs. 256 bit variant)**			
	A	**P**	**RD [%]**	**Major Reasons**	**A**	**P**	**RD [%]**	**Major Reasons**
BLAKE	1.89	2	-5.61	-	1.13	1.43	-21.17	64-bit vs. 32-bit addition
BMW	1.99	2	-0.29	-	1.11	2	-44.48	routing congestion
CubeHash	0.97	1	-3.42	-	0.97	1	-2.9	-
ECHO	1.09	1	9	-	0.46	0.54	-14.81	routing congestion
Fugue	1.09	1.2	-9.17	-	0.54	0.5	8	-
Groestl	1.96	2	-1.94	-	1.42	1.43	-0.45	-
Hamsi	2.4	2	19.8	look-up tables of the message expansion unit increase by a factor of 4	0.69	1	-30.93	table look-up time in the message expansion unit increase by a factor of 2
JH	1.03	1	3.46	-	1	1	-0.18	-
Keccak	0.89	1	-10.96	smaller message block size (576 vs. 1088 bits) = smaller input shift reg.	0.54	0.53	1.01	-
Luffa	2.08	1.67	24.53	larger constants in the $GF(2^8)$ muls in the Message Injection phase	0.94	1	-5.72	-
Shabal	1.02	1	1.55	-	1.02	1	2.36	-
SHAvite-3	2.07	2	3.64	-	1.19	1.29	-7.41	-
SIMD	2.11	2	5.26	-	1.86	2	-7.14	-
Skein	1.02	1	1.78	-	1	1	0	-
SHA-2	1.67	2	-16.62	control unit relatively large compared to the datapath	1.58	1.6	-1.4	-

Table 4.4: 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	106.01	117.06	0.91	3743.28	2853.91	1.31	3276	1871	1.75	1.14	1.53	0.75
BMW	8.45	10.89	0.78	8655.87	5576.70	1.55	10401	4400	2.36	0.83	1.27	0.66
CubeHash	215.33	219.30	0.98	3508.77	3445.31	1.02	707	764	0.93	4.59	4.87	0.94
ECHO	200.97	234.85	0.86	6430.88	13874.33	0.46	5958	5445	1.09	1.08	2.55	0.42
Fugue	138.49	98.47	1.41	1107.88	3151.17	0.35	924	956	0.97	1.20	3.30	0.36
Groestl	325.63	350.51	0.93	11498.00	8545.72	1.35	3155	1716	1.84	3.64	4.98	0.73
Hamsi	171.38	248.08	0.69	1828.05	2646.15	0.69	2201	946	2.33	0.83	2.80	0.30
JH	275.48	278.09	0.99	3917.97	3955.02	0.99	1165	1108	1.05	3.36	3.57	0.94
Keccak	276.86	238.38	1.16	6644.52	10806.51	0.61	1236	1229	1.01	5.38	8.79	0.61
Luffa	220.12	281.53	0.78	7043.81	8008.02	0.88	2164	1154	1.88	3.25	6.94	0.47
Shabal	135.30	128.12	1.06	2770.94	2623.96	1.06	1372	1266	1.08	2.02	2.07	0.97
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	104.34	104.34	1.00	2811.72	2811.72	1.00	1520	1463	1.04	1.85	1.92	0.96
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 4.5: 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	93.41	124.55	0.75	3298.34	3036.65	1.09	3414	1779	1.92	0.97	1.71	0.57
BMW	7.44	16.45	0.45	7618.56	8422.40	0.90	25225	12632	2.00	0.30	0.67	0.45
CubeHash	218.05	236.07	0.92	3488.80	3777.12	0.92	1924	1928	1.00	1.81	1.96	0.93
ECHO	246.00	N/A	N/A	7872.00	N/A	N/A	20085	N/A	N/A	0.39	N/A	N/A
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	358.94	364.96	0.98	5104.92	5190.54	0.98	3222	3107	1.04	1.58	1.67	0.95
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	126.44	126.87	1.00	2589.49	2598.30	1.00	3753	3600	1.04	0.69	0.72	0.96
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	92.10	92.10	1.00	2481.85	2481.85	1.00	4563	4499	1.01	0.54	0.55	0.98
SHA-2	234.80	212.81	1.10	2968.34	1676.29	1.77	1620	963	1.68	1.83	1.74	1.05

FPGA families. In terms of the design, an architecture by Chaves et al. [38], [39] is selected, as it is considered one of the best known SHA-2 architectures, and is optimized specifically for the maximum throughput to area ratio.

Table 4.6: 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 4.7: 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

Tables 4.8 and 4.9 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, BMW is unable to fit in Cyclone II, Cyclone III and Stratix II due to the routing congestion.

This congestion is due to multi-operand additions exhausting 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 4.8: 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	45.98	85.90	117.06	56.97	66.38	103.21	124.55
BMW	4.19	12.37	10.89	7.69	8.41	13.45	16.45
CubeHash	91.27	186.50	219.30	115.89	133.19	179.37	236.07
ECHO	52.10	131.90	234.85	N/A	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	129.75	271.67	278.09	174.61	222.72	268.31	364.96
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	36.93	81.20	104.34	47.06	54.73	70.64	92.10
SHA-512	90.84	183.02	207.00	111.04	126.86	158.08	212.81

Table 4.9: 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	36.59	71.26	106.01	41.57	50.24	73.78	93.41
BMW	N/A	6.03	8.45	N/A	N/A	N/A	7.44
CubeHash	90.84	188.89	215.33	113.43	129.20	164.69	218.05
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	113.68	281.37	325.63	124.10	133.96	187.72	250.38
Hamsi	69.00	158.05	171.38	103.31	117.16	128.68	181.16
JH	130.12	277.32	275.48	173.94	221.93	267.52	358.94
Keccak	94.12	208.86	276.86	161.39	173.07	207.68	269.61
Luffa	93.41	210.88	220.12	143.53	172.98	192.49	268.02
Shabal	29.87	113.62	135.30	69.38	81.70	103.58	126.44
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	36.93	81.20	104.34	47.06	54.73	70.64	92.10
SHA-512	90.06	168.75	215.84	93.54	113.15	177.34	234.80

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* normalized result is a geometric mean of normalized results for all investigated FPGA families.

Tables 4.10, 4.11, and 4.12 summarize normalized results for the 256-bit variants of all SHA-3 candidates in terms of throughput, area, and throughput to area ratio, respectively. In terms of throughput, the best performance is accomplished by Keccak, ECHO, Luffa, BMW, and Groestl. No candidate is slower than SHA-256 in terms of throughput. In terms of area, none of the candidates is smaller than SHA-256. The ones that come closest are CubeHash, Hamsi, BLAKE, and Luffa. BMW, SIMD, and ECHO have their areas over 10 times bigger than the area of SHA-256. In terms of the throughput to area ratio, the best performers are Keccak, Luffa, and CubeHash, which are the only candidates outperforming SHA-256. The worst results in terms of this measure, belong to ECHO, and SIMD, which loose to SHA-256 by a factor of at least 3.

Table 4.10: 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
Keccak	6.10	6.37	6.63	8.56	7.91	7.23	8.01	7.21
ECHO	4.30	5.41	8.51	N/A	6.25	5.20	5.79	5.78
Luffa	5.16	5.14	4.91	5.58	4.94	5.02	5.21	5.13
BMW	3.00	4.39	3.42	4.50	4.31	5.53	5.02	4.48
Groestl	3.60	3.97	5.32	3.68	3.62	4.24	3.93	4.02
JH	2.58	2.68	2.43	2.84	3.17	3.06	3.10	2.82
CubeHash	2.03	2.10	2.11	2.12	2.13	2.30	2.25	2.15
Fugue	1.77	1.62	1.93	1.95	1.94	2.15	2.36	1.95
BLAKE	1.57	1.45	1.75	1.59	1.62	2.02	1.81	1.68
Hamsi	1.35	1.49	1.62	1.82	1.96	1.66	1.88	1.67
SHAvite-3	1.64	1.46	1.77	1.51	1.58	1.89	2.11	1.70
Skein	1.39	1.46	1.72	1.45	1.48	1.53	1.48	1.50
Shabal	0.89	1.62	1.61	1.63	1.41	1.73	1.55	1.46
SIMD	1.37	1.15	1.43	1.41	1.36	1.69	1.61	1.38

In Fig. 4.1, we present a two dimensional diagram, with the Overall Normalized Area on the X-axis and the Overall Normalized Throughput on the Y-axis. The algorithms seem to fall into several major groups. Group with the high normalized throughput (>5), medium normalized area (<4), and the high normalized throughput to area ratio (>1.5), include Keccak and Luffa. Groestl, BMW, and ECHO, have all high normalized throughput (>4), but their normalized area varies significantly from about 6 in case of Groestl, through 12 for BMW, up to over 25 in case of ECHO. SIMD is both relatively slow (less then 1.4 times faster than SHA-256) and big (more than 24 times bigger than SHA-256). The last group includes 8 candidates covering the range of the normalized throughputs from 0.8 to 2.8, and the normalized areas from 2.1 to 4.1.

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

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
CubeHash	1.76	1.75	1.63	1.87	1.86	2.00	2.00	1.84
Hamsi	2.17	2.16	2.18	1.92	1.94	2.40	2.41	2.16
BLAKE	4.91	4.87	4.32	2.20	2.32	1.86	1.85	2.92
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
JH	4.43	4.42	2.56	4.15	4.10	3.17	3.23	3.66
Keccak	3.97	3.99	2.84	3.77	3.62	4.20	4.63	3.82
SHAvite-3	4.91	4.91	2.61	5.68	5.64	2.57	2.59	3.89
Skein	4.01	3.95	3.38	3.98	4.12	4.64	4.67	4.09
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
BMW	12.07	13.45	10.16	12.00	12.02	12.99	13.12	12.24
SIMD	20.97	19.99	21.45	18.53	18.57	23.03	23.24	20.39
ECHO	30.87	28.48	12.58	0.00	39.77	22.29	22.52	24.58

Table 4.12: 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
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
CubeHash	1.16	1.20	1.29	1.13	1.14	1.15	1.13	1.17
Hamsi	0.62	0.69	0.74	0.94	1.01	0.69	0.78	0.77
JH	0.58	0.61	0.95	0.68	0.77	0.97	0.96	0.77
Groestl	0.23	0.25	1.22	0.80	0.81	1.32	1.22	0.69
BLAKE	0.32	0.30	0.41	0.72	0.70	1.09	0.98	0.57
Fugue	0.42	0.36	0.88	0.33	0.33	0.58	0.63	0.47
SHAvite-3	0.33	0.30	0.68	0.27	0.28	0.74	0.81	0.44
Shabal	0.24	0.42	0.55	0.44	0.38	0.44	0.41	0.40
BMW	0.25	0.33	0.34	0.38	0.36	0.43	0.38	0.37
Skein	0.35	0.37	0.51	0.36	0.36	0.43	0.38	0.37
ECHO	0.14	0.19	0.68	N/A	0.16	0.23	0.26	0.23
SIMD	0.07	0.06	0.07	0.08	0.07	0.07	0.07	0.07

Table 4.13: 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
Groestl	3.53	4.66	4.21	3.71	3.31	2.96	2.98	3.58
Luffa	2.63	3.16	2.58	3.88	3.87	2.75	2.89	3.07
BMW	N/A	2.90	3.17	N/A	N/A	N/A	2.57	2.87
ECHO	2.39	2.85	2.36	0.00	3.03	2.38	2.65	2.60
Keccak	1.98	2.35	2.44	3.28	2.90	2.22	2.18	2.45
JH	1.63	1.85	1.44	2.09	2.21	1.70	1.72	1.79
SIMD	N/A	1.52	1.52	1.93	1.90	1.64	1.66	1.69
CubeHash	1.28	1.40	1.29	1.53	1.45	1.18	1.18	1.32
SHAvite-3	1.19	1.36	1.41	1.32	1.30	1.13	1.30	1.28
BLAKE	1.13	1.18	1.37	1.24	1.24	1.16	1.11	1.21
Skein	0.87	1.03	1.03	1.07	1.03	0.85	0.84	0.96
Shabal	0.54	1.09	1.02	1.20	1.17	0.95	0.87	0.95
Hamsi	0.65	0.79	0.67	0.93	0.87	0.61	0.65	0.73
Fugue	0.45	0.46	0.41	0.59	0.56	0.51	0.56	0.50

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
CubeHash	1.28	1.24	1.18	1.16	1.17	1.18	1.19	1.20
Fugue	2.22	2.16	1.43	2.54	2.52	1.72	1.71	2.00
Keccak	2.25	2.17	1.91	1.81	1.81	2.19	2.21	2.04
Shabal	2.28	2.25	2.12	2.10	2.11	2.29	2.32	2.21
JH	2.81	2.74	1.80	2.41	2.37	1.97	1.99	2.27
Skein	2.51	1.32	2.35	2.28	2.36	2.79	2.82	2.29
BLAKE	5.86	5.44	5.07	2.52	2.44	2.12	2.11	3.02
Hamsi	3.19	3.10	3.41	2.61	2.61	3.50	3.50	3.11
Luffa	3.92	3.82	3.35	3.60	3.63	4.28	4.25	3.82
SHAvite-3	5.85	6.09	3.02	7.01	7.01	3.36	3.46	4.83
Groestl	10.68	10.28	4.88	5.08	5.16	3.71	3.88	5.73
BMW	N/A	13.50	16.10	N/A	N/A	N/A	15.57	15.01
ECHO	19.56	18.43	9.22	N/A	23.89	12.26	12.40	15.15
SIMD	N/A	28.29	26.34	22.09	22.13	29.15	29.43	26.05

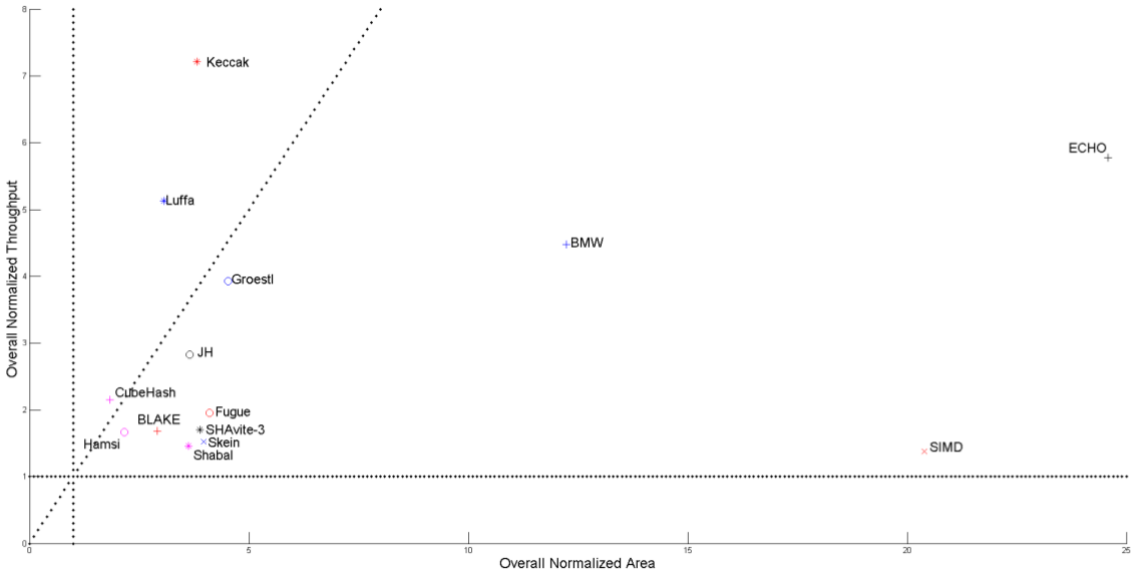

Figure 4.1: Relative performance of all Round 2 SHA-3 Candidates (256-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-256 used as a reference point).)

Table 4.15: 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
Keccak	0.88	1.09	1.27	1.81	1.60	1.02	0.99	1.20
CubeHash	1.00	1.13	1.09	1.33	1.24	1.00	0.99	1.10
Luffa	0.67	0.83	0.77	1.08	1.07	0.64	0.68	0.80
JH	0.58	0.68	0.80	0.87	0.93	0.86	0.86	0.79
Groestl	0.33	0.45	0.86	0.73	0.64	0.80	0.77	0.62
Shabal	0.24	0.48	0.48	0.57	0.55	0.41	0.38	0.43
BLAKE	0.19	0.22	0.27	0.49	0.51	0.55	0.53	0.40
Skein	0.35	0.43	0.44	0.47	0.44	0.30	0.30	0.38
SHAvite-3	0.20	0.22	0.46	0.19	0.19	0.34	0.38	0.27
Fugue	0.20	0.21	0.28	0.23	0.22	0.30	0.32	0.25
Hamsi	0.20	0.25	0.20	0.36	0.34	0.17	0.19	0.24
BMW	N/A	0.21	0.20	N/A	N/A	N/A	0.16	0.19
ECHO	0.12	0.15	0.26	0.00	0.13	0.19	0.21	0.17
SIMD	N/A	0.05	0.06	0.09	0.09	0.06	0.06	0.06

Interestingly, only two candidates, Keccak and CubeHash outperform SHA-512 in terms of the throughput to area ratio. Only three more candidates, Luffa, JH and Groestl, have the overall normalized ratio higher than 0.5. In terms of throughput, only five candidates, Groestl, Luffa, BMW, ECHO, and Keccak, outperform SHA-512 by a factor larger than

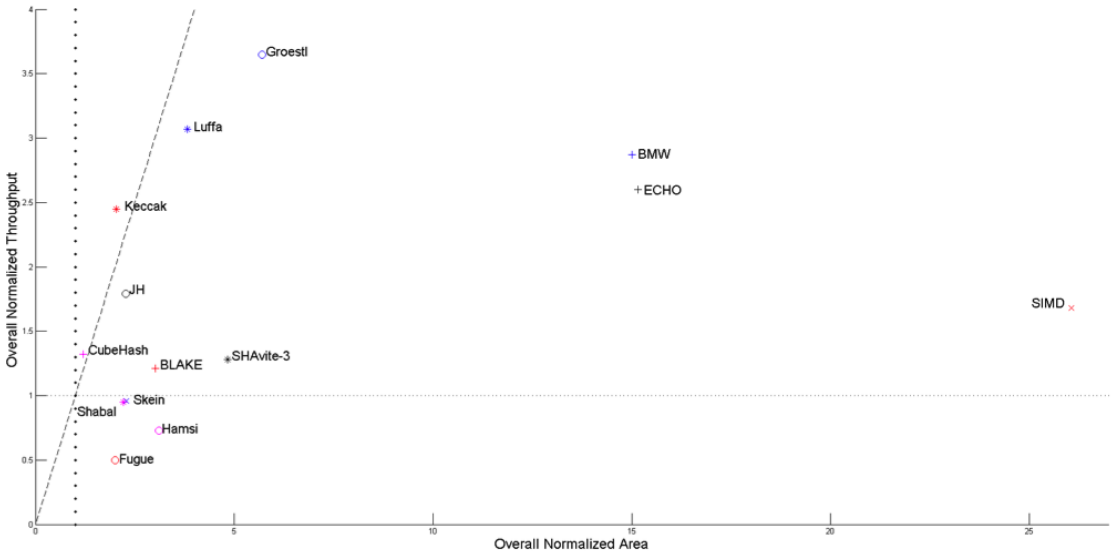

Figure 4.2: Relative performance of all Round 2 SHA-3 Candidates (512-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-512 used as a reference point).)

two. The additional five candidates have a normalized throughput in the range from 1 to 2. Four candidates, Skein, Shabal, Hamsi, and Fugue, are slower than SHA-512, with Fugue, slower by a factor of two. In terms of area, all SHA-3 candidates, in their 512-bit variants, are larger than SHA-512. The spread of results is much larger than in the case of throughput, with the smallest SHA-3 candidate, CubeHash, almost the same size as SHA-512, and the largest SIMD, lagging behind by a factor of 26. The group following CubeHash in terms of area, including Fugue, Keccak, Shabal, JH and Skein, covers the range between 2.0 and 2.3, and includes only one candidate, Keccak, which excels also in terms of speed.

In Fig. 4.2, we presents a two dimensional diagram, with the Overall Normalized Area on the X-axis and the Overall Normalized Throughput on the Y-axis. Only two algorithms, Keccak and CubeHash, outperform SHA-512 in terms of the throughput to area ratio. Out of them Keccak is almost twice as fast, but CubeHash is almost twice as small. SIMD is approximately 20 times worse than Keccak in terms of the throughput to area ratio, and ECHO and BMW are more than 6 times worse. The implementations of these algorithms are not likely to scale to the same performance region as implementations of majority of other candidates, even if significantly trading speed for reduced area.

A throughput based on the performance for long messages does not reliably describe the behavior of the developed hash modules for short messages. For some applications, an

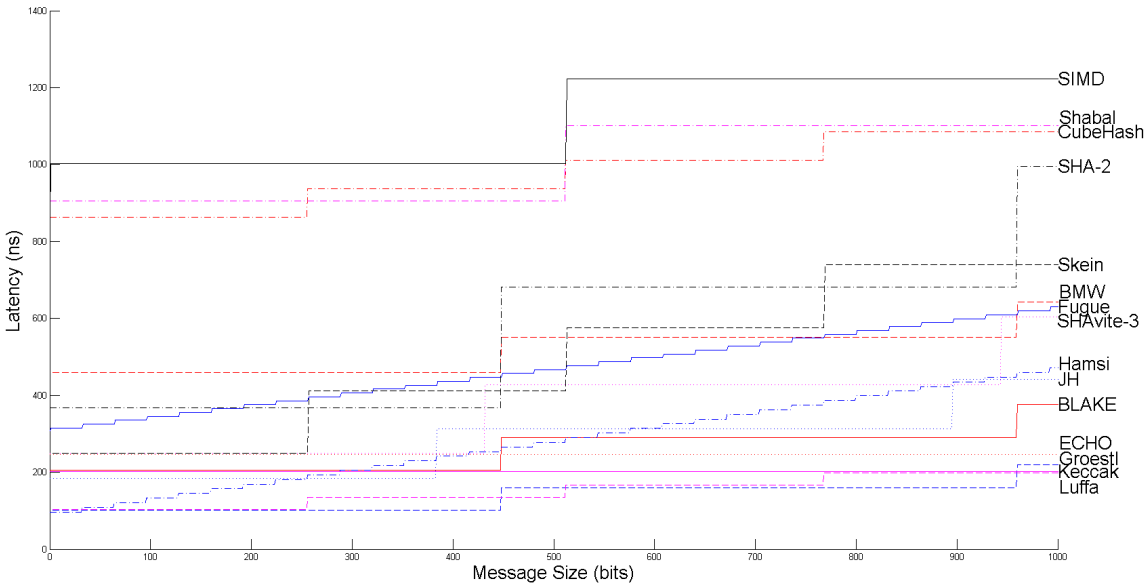

Figure 4.3: Execution time vs. message size for short messages up to 1,000 bits. 256-bit variants of all SHA-3 Candidates and SHA-256 in Virtex 5.

algorithm that can perform particularly well for short messages may be favored over an algorithm that is exceptionally good for long messages, but terribly slow for short ones. In Figure 4.3 we present the execution time as a function of the message length, varying between 0 and 1000 bits, for 256-bit variants of all SHA-3 candidates and SHA-256. Similar graphs for 512-bit variants of all algorithms are presented in Figure 4.3. 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.

For the 256-bit variants of all algorithms, the only ones performing worse than SHA-256 are SIMD, Shabal, and CubeHash. The best performers are Luffa, Keccak, Groestl, and ECHO. For the 512-bit variants, the worst performing algorithms are the same as for the SHA-256 case, with the addition of Fugue. The best performers are the same as in the SHA-256 case, with the addition of JH and Skein.

In Figure 4.5, we summarize all our results for both 256 and 512 variants of all algorithms. Each variant of each algorithm is characterized using four performance measures: the throughput to area ratio, throughput, area, and the execution time for short messages. The performance is graded on the 3-point scale and denoted using the following color code: green – best, yellow – medium, red–worst. The best performing algorithms are those that have the largest number of green boxes (high scores), and no red box (low scores) associated with them. Additionally, taking into account that our designs are intended as high-speed designs, and not low-cost designs, the throughput to area ratio and throughput

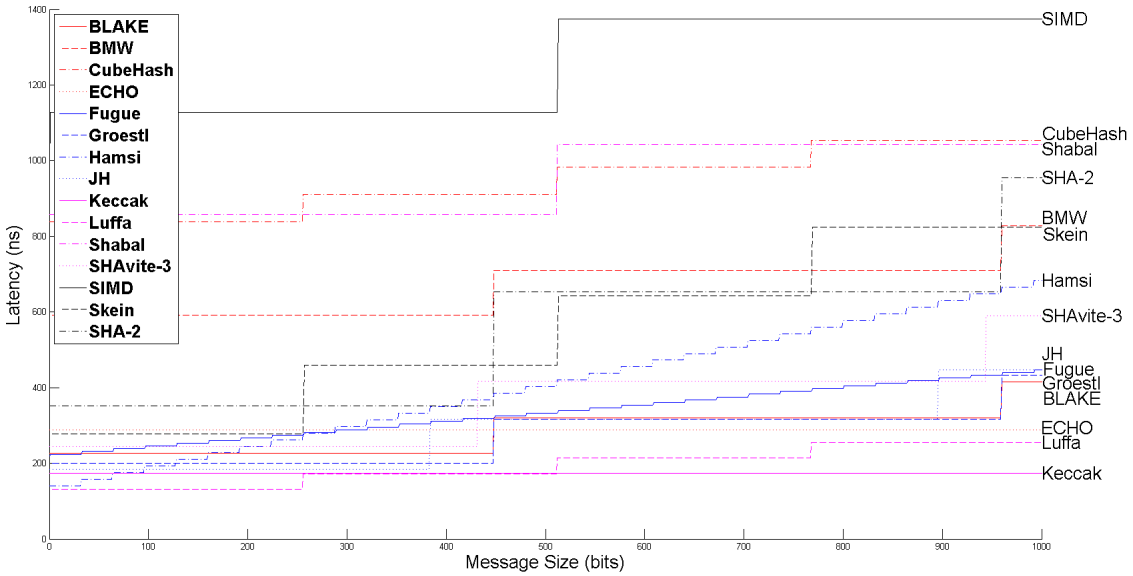

Figure 4.4: Execution time vs. message size for short messages up to 1,000 bits. 512-bit variants of all SHA-3 Candidates and SHA-512 in Virtex 5.

are treated as two primary performance measures, while area and the execution time for short messages are considered as secondary criteria. Under these assumptions, the two best performing candidates are Keccak and Luffa, scoring high in 7 out of 8 categories. These two algorithms are followed by Groestl, which excels in 4 categories, namely throughput, and the execution time for short messages in both function variants. SIMD is the worst performing algorithm so far, with the low scores in 6 out of 8 categories. Additionally, ECHO and BMW perform quite poorly in terms of area. Out of the remaining candidates, the ones with the highest potential are CubeHash, excelling in terms of area and throughput to area ratio for both variants of the algorithm, JH, performing particularly well in its 512-bit variant, and BLAKE, which does not receive low scores in any of the 8 categories.

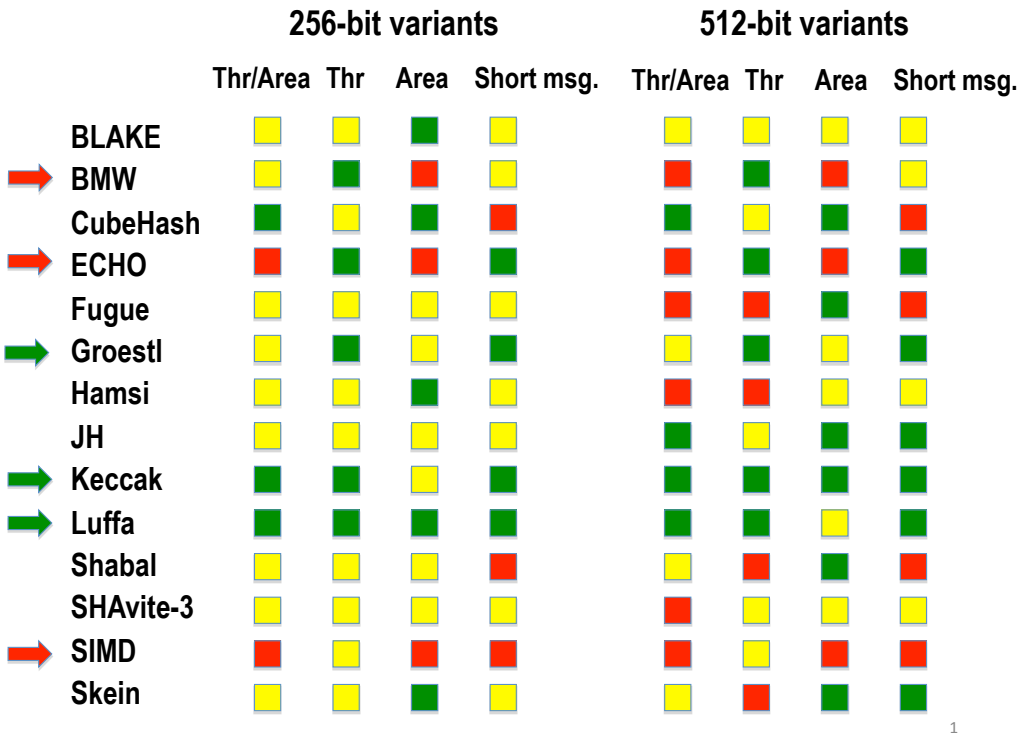

Figure 4.5: Summary of major features of all SHA-3 candidates in terms of performance in FPGAs. Color code: green - best, yellow - medium, red - worst. The performance is characterized using four metrics: throughput to area ratio, throughput, area, and execution time for short messages.

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 paper. 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 modules residing between FSMs 1, 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) to the best designs reported in the literature to date. Selected algorithms for which our results are worse are discussed below:

- *BLAKE*: The design by Aumasson et al. [21] 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*: An interface overhead puts our design at a disadvantage with some other designs following different interfaces

- *Groestl*: Similarly to CubeHash, an interface overhead causes our throughput to area ratio to drop. Also, we may yet overlook a possible way to reduce resource utilization.

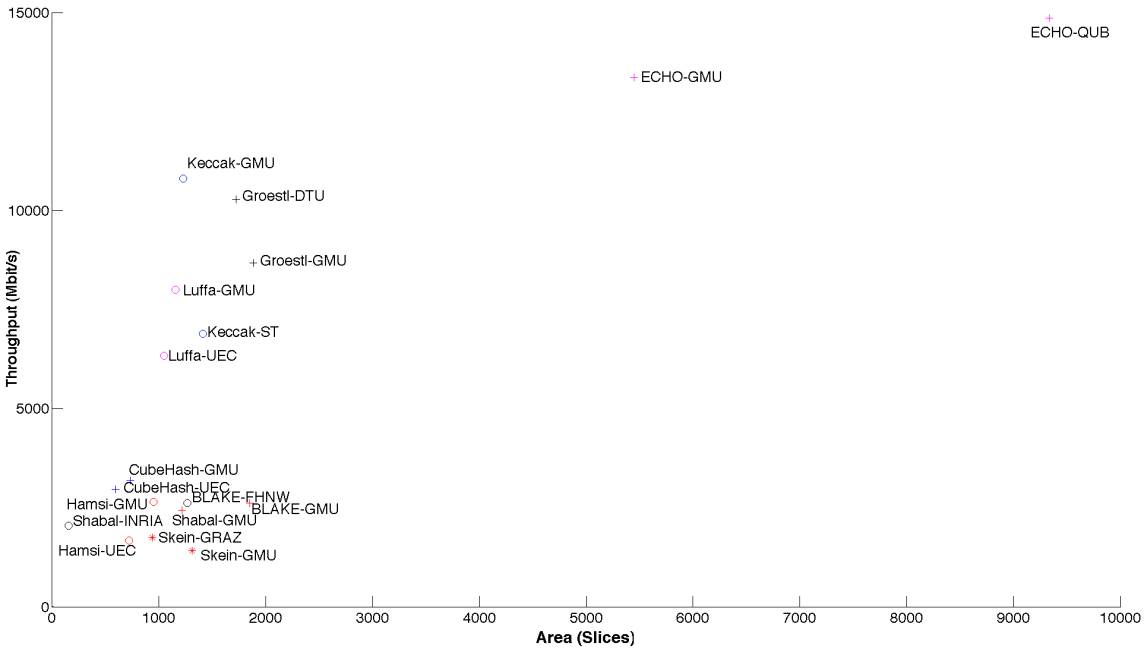

Figure 5.1: Best *published* results vs. GMU results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput to area ratio in Xilinx Virtex 5

- *Shabal*: Detrey et al. [40] utilizes a shift register mode (SRL16) of Xilinx Multipurpose Look-Up Tables. This optimization applies only to Xilinx FPGAs, and has not been yet introduced in our 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. [21]	1851	2610.6	1.41
CubeHash	590	2960	5.02	Kobayashi et al. [12]	730	3189.8	4.37
ECHO	9333	14860	1.59	Lu et al. [41]	5445	13360.5	2.45
Groestl	1722	10276	5.97	Gauvaram et al. [26]	1884	8676.5	4.61
Hamsi	718	1680	2.34	Kobayashi et al. [12]	946	2646.2	2.80
Keccak	1412	6900	4.89	Bertoni et al. [29]	1229	10806.5	8.79
Luffa	1048	6343	6.05	Kobayashi et al. [12]	1154	8008.0	6.94
Shabal	153	2051	13.41	Detrey et al. [40]	1266	2624.0	2.07

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 demonstrates the best throughput to area ratio for Virtex 5. Additionally, Shabal will most likely retains its lead in comparison of 512-bit variants as there is practically no functional change between 256 and 512-bit variants of this algorithm.

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. [21]
BMW	4400	5576.7	1.27	GMU
CubeHash	590	2960.0	5.02	Kobayashi et al. [12]
ECHO	5445	13360.5	2.45	GMU
Fugue	956	3151.2	3.30	GMU
Groestl	1722	10276.0	5.97	Gauvaram et al. [26]
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	1463	2811.7	1.92	GMU

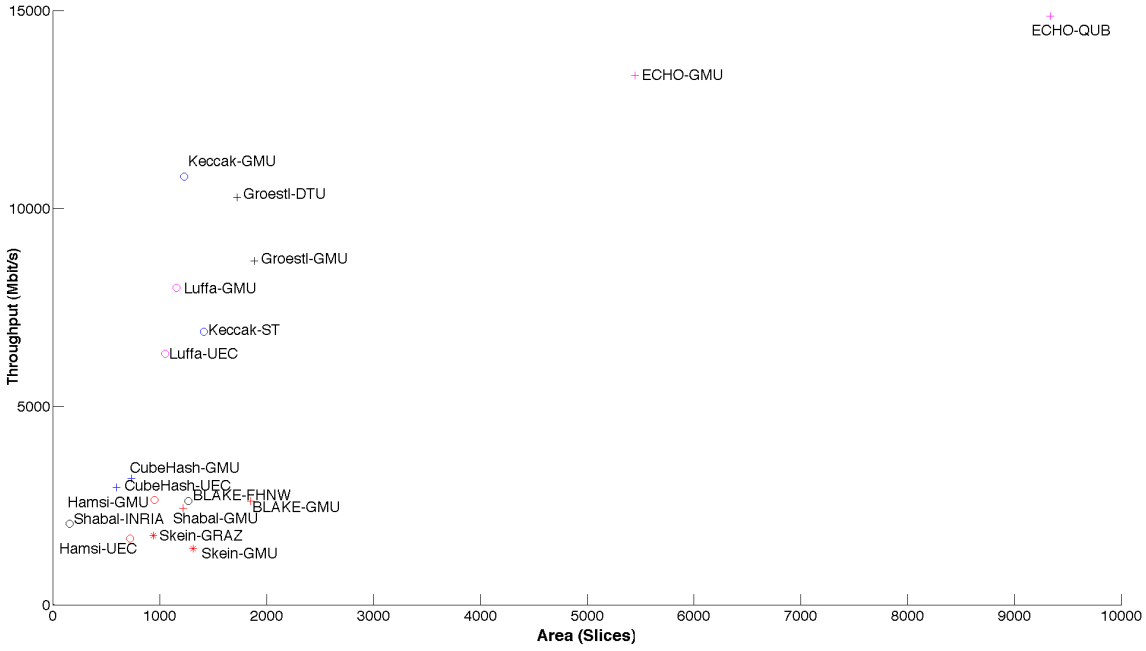

Figure 5.2: Best results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput vs. area in Virtex 5

Chapter 6

Conclusions and Future Work

Our evaluation methodology, applied to 14 Round 2 SHA-3 candidates, has demonstrated large differences among competing candidates.

For the 256-bit variants of the SHA-3 candidates, the ratio of the best result to the worst result was equal to about 9 in terms of the throughput (Keccak vs. Skein), over 13 times in terms of area (CubeHash vs. ECHO), and about 27 in terms of our primary optimization target, the throughput to area ratio (Keccak vs. SIMD). Only three candidates, Keccak, Luffa, and CubeHash, have demonstrated the throughput to area ratio better than the current standard SHA-256. Out of these three algorithms, Keccak and Luffa have also demonstrated very high throughputs, while CubeHash outperformed other candidates in terms of minimum area. All candidates except Skein outperform SHA-256 in terms of the throughput, but at the same time none of them matches SHA-256 in terms of the area.

For the 512-bit variants, the ratio of the best result to the worst result was equal to about 6 in terms of the throughput (Luffa vs. Fugue), about 23 in terms of area (CubeHash vs. SIMD), and about 20 in terms of our primary optimization target, the throughput to area ratio (Keccak vs. SIMD). Only two candidates, Keccak and CubeHash, have demonstrated the throughput to area ratio better than the current standard SHA-512. Out of these two algorithms, Keccak has also demonstrated very high throughputs, while CubeHash outperformed other candidates in terms of minimum area. Almost all candidates, except Fugue, Hamsi, Shabal, and Skein, outperform SHA-512 in terms of the throughput, but at the same time none of them, except CubeHash, matches SHA-512 in terms of the area.

Future work will include the evaluation of the remaining variants of SHA-3 candidates (such as variants with 224 and 384 bit outputs, and an all-in-one architecture). The uniform padding units will be added to each SHA core, and their cost estimated. In terms of FPGA families, our study will be extended to the most recent families of FPGAs from two major vendors, namely Spartan 6 and Virtex 6 from Xilinx, and Cyclone IV, Stratix IV, and Arria II from Altera. We will also investigate the influence of synthesis tools from

different vendors (e.g., Synplify Pro from Synopsys). The evaluation may be also extended to the cases of hardware architectures optimized for the minimum area (cost), maximum throughput (speed), or minimum power consumption. Each algorithm will be also evaluated in terms of its suitability for implementation using dedicated FPGA resources, such embedded memories, dedicated multipliers, and DSP units. Our methodology can be also applied to the implementations of MACs based on the SHA-3 candidates (in particular, HMAC), with added countermeasures against side channel attacks. Finally, an extension of our methodology to the standard-cell ASIC technology will be investigated.

Acknowledgments.

The authors would like to acknowledge all George Mason University students from the Fall 2009 edition of the course entitled "Digital System Design with VHDL," and the Spring 2010 edition of the course entitled "Computer Arithmetic" for conducting initial exploration of the design space of all SHA-3 candidates.

Bibliography

[1] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback, "Report on the development of the Advanced Encryption Standard (AES)." `http://csrc.nist.gov/archive/aes/round2/r2report.pdf`.

[2] "NESSIE." `https://www.cosic.esat.kuleuven.be/nessie`.

[3] "eSTREAM." `http://www.ecrypt.eu.org/stream`.

[4] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays," in *Progress in Cryptology - CT-RSA 2001* (D. Naccache, ed.), vol. 2020 of *Lecture Notes in Computer Science (LNCS)*, pp. 84–99, Springer, Apr. 2001.

[5] D. Hwang, M. Chaney, S. Karanam, N. Ton, and K. Gaj, "Comparison of FPGA-targeted hardware implementations of eSTREAM stream cipher candidates," in *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*, pp. 151–162, Feb. 2008.

[6] T. Good and M. Benaissa, "Hardware performance of eSTREAM phase-iii stream cipher candidates," in *State of the Art of Stream Ciphers Workshop (SASC 2008)*, pp. 163–173, Feb 2008.

[7] "SHA-3 Contest." `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[8] "SHA-3 Zoo." `http://ehash.iaik.tugraz.at/wiki/TheSHA-3Zoo`.

[9] S. Drimer, *Security for volatile FPGAs. Chapter 5: The Meaning and Reproducibility of FPGA Results.* Ph.d. dissertation, University of Cambridge, Computer Laboratory, Nov 2009. UCAM-CL-TR-763, `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-763.pdf`.

[10] "SHA-3 Zoo: SHA-3 hardware implementations." `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations`.

[11] 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, Groestl, Hamsi, JH, Keccak, Luffa, Shabal, Shavite-3, SIMD, and Skein." Cryptology ePrint Archive, Report 2009/510, 2009. `http://eprint.iacr.org/`.

[12] 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/`.

[13] "ECRYPT Benchmarking of Cryptographic Systems." `http://bench.cr.yp.to`.

[14] C. G. Group, "Hardware Interface of a Secure Hash Algorithm (SHA)." `http://cryptography.gmu.edu/athena/index.php?id=interfaces`.

[15] "ANSI C cryptographic API profile for SHA-3 candidate submissions." `http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html`.

[16] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, ch. Fourier Transforms (6) and Advanced Topics (7), pp. 343–475. Signals, Springer, third edition ed., 2007.

[17] J. van Lint, *Introduction to Coding Theory*, vol. 86 of *Graduate Texts in Mathematics*. Springer, second ed., 1992.

[18] K. Gaj and P. Chodowiec, *Cryptographic Engineering*, ch. Chapter 10, FPGA and ASIC Implementations of AES, pp. 235–294. Springer, 2009.

[19] J.-P. Deschamps, G. J. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. Wiley-Interscience, 2006.

[20] G. CERG, "ATHENa project website." Online, 2010. `http://cryptography.gmu.edu/athena/`.

[21] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "SHA-3 proposal BLAKE." Submission to NIST, 2008.

[22] 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.

[23] D. J. Bernstein, "CubeHash specification (2.b.1)." Submission to NIST (Round 2), 2009.

[24] 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.

[25] S. Halevi, W. E. Hall, and C. S. Jutla, "The hash function Fugue." Submission to NIST (updated), 2009.

[26] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlffer, and S. S. Thomsen, "Grøstl – a SHA-3 candidate." Submission to NIST, 2008.

[27] O. Kucuk, "The hash function Hamsi." Submission to NIST (updated), 2009.

[28] H. Wu, "The hash function JH." Submission to NIST (updated), 2009.

[29] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Keccak specifications." Submission to NIST (Round 2), 2009.

[30] C. D. Canniere, H. Sato, and D. Watanabe, "Hash function Luffa: Specification." Submission to NIST (Round 2), 2009.

[31] N. I. of Standards and T. (NIST), "Secure Hash Standard (SHS) FIPS publication 180-3," Oct 2008.

[32] 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 NIST's cryptographic hash algorithm competition." Submission to NIST, 2008.

[33] E. Biham and O. Dunkelman, "The SHAvite-3 hash function." Submission to NIST (Round 2), 2009.

[34] G. Leurent, C. Bouillaguet, and P.-A. Fouque, "SIMD is a message digest." Submission to NIST (Round 2), 2009.

[35] 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.

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

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

[38] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Improving SHA-2 hardware implementations," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, vol. 4249 of *LNCS*, pp. 298–310, Springer, Oct. 2006.

[39] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Cost efficient SHA hardware accelerators," *IEEE Trans. Very Large Scale Integration Systems,*, vol. 16, pp. 999–1008, 2008.

[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] 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`.

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