
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

Last revised: December 21, 2010

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

1	Introduction and Motivation	2
2	Methodology	4
2.1	Choice of a Language, FPGA Devices, and Tools	4
2.2	Performance Metrics for FPGAs	4
2.3	Uniform Interface	7
2.4	Assumptions and Simplifications	9
2.5	Optimization Target	9
2.6	Design Methodology	10
3	Comprehensive Designs of SHA-3 Candidates	13
3.1	Notations and Symbols	13
3.2	Basic Component Description	15
3.2.1	Multiplication by 2 in the Galois Field $GF(2^8)$	15
3.2.2	Multiplication by n in the Galois Field $GF(2^8)$	15
3.2.3	AES	16
3.3	BLAKE	19
3.3.1	Block Diagram Description	19
3.3.2	256 vs. 512 Variant Differences	21
3.4	Blue Midnight Wish (BMW)	24
3.4.1	Block Diagram Description	24
3.4.2	256 vs. 512 Variant Differences	24
3.5	CubeHash	26
3.5.1	Block Diagram Description	26
3.5.2	256 vs. 512 Variant Differences	26
3.6	ECHO	29
3.6.1	Block Diagram Description	29
3.6.2	256 vs. 512 Variant Differences	31
3.7	Fugue	33
3.7.1	Block Diagram Description	33

3.7.2	256 vs. 512 Variant Differences	35
3.8	Groestl	39
3.8.1	Block Diagram Description	39
3.8.2	256 vs. 512 Variant Differences	41
3.9	Hamsi	43
3.9.1	Block Diagram Description	43
3.9.2	256 vs. 512 Variant Differences	45
3.10	JH	48
3.10.1	Block Diagram Description	48
3.10.2	256 vs. 512 Variant Differences	48
3.11	Keccak	53
3.11.1	Block Diagram Description	53
3.11.2	256 vs. 512 Variant Differences	57
3.12	Luffa	58
3.12.1	Block Diagram Description	58
3.12.2	256 vs. 512 Variant Differences	58
3.13	SHA-2	65
3.13.1	Block Diagram Description	65
3.13.2	256 vs. 512 Variant Differences	65
3.14	Shabal	67
3.14.1	Block Diagram Description	67
3.14.2	256 vs 512 Variant Differences	67
3.15	SHAvite-3	69
3.15.1	Block Diagram Description	69
3.15.2	256 vs. 512 Variant Differences	70
3.16	SIMD	75
3.16.1	Block Diagram Description	75
3.16.2	256 vs. 512 Variant Differences	82
3.17	Skein	86
3.17.1	Block Diagram Description	86
3.17.2	256 vs. 512 Variant Differences	87
4	Design Summary and Results	90
4.1	Design Summary	90
4.2	Relative Performance of the 512 and 256-bit Variants of the SHA-3 Candidates	91
4.3	Results	93
5	Results from Other Groups	108
5.1	Best Results from Other Groups	108
5.2	Best Results	109

6	Conclusions and Future Work	114
A	Absolute Results for Ten FPGA Families	116

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 ten 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, several comprehensive studies have been reported in the Cryptology ePrint Archive [11][12], at the CHES 2010 workshop [13][14], and during the Second SHA-3 Candidate Conference [15][16][17][18]. This report is an extension of our earlier papers presented at CHES 2010 and the SHA-3 Candidate Conference [13][16]. To our best knowledge, this is the first report that presents the detailed block diagrams of all 14 Round 2 SHA-3 Candidates, and a comprehensive set of results covering two major SHA-3 variants (SHA-3-256 and SHA-3-512) implemented using 10 FPGA families from two major vendors, Xilinx and Altera.

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 ten 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, III, and IV from Altera) and those optimized for high performance (Virtex 4, 5, and 6 from Xilinx, and Stratix II, III, and IV 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. 12.3 (including Xilinx XST, used for synthesis) and Altera Quartus II v. 10.0 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:

$$\text{Throughput} = \frac{\text{block_size}}{T \cdot (\text{HTime}(N + 1) - \text{HTime}(N))} \quad (2.1)$$

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

$$\text{Throughput}_{eff} = \frac{N \cdot \text{block_size}}{T \cdot \text{HTime}(N)} \quad (2.2)$$

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 [19]. 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 messages ranging in size from 0 to 1000 bits before 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 and Virtex 6, 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, III, and IV, the basic programmable block is called Logic Element (LE); in Stratix II, III, and IV, 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 interchangeable, 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,

$$Resource_Utilization_{Spartan3} = (\#CLBslices, \#BRAMs, \#MULs) \quad (2.3)$$

$$Resource_Utilization_{CycloneIII} = (\#LE, \#memory_bits, \#MULs) \quad (2.4)$$

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 families, and the number of Adaptive Look-Up Tables ($\#ALUT$) in Stratix families.

Unfortunately, in four out of ten families, namely Cyclone II, Cyclone III, Cyclone IV, and Stratix II, the complete elimination of the use of memory blocks is very difficult, and might require a substantial redesign of the circuit. This is because these families have no concept of distributed memory, i.e., memory implemented inside of basic configurable building blocks of Altera FPGAs, such as Logic Elements or Adaptive Look-Up Tables. As a result, each time memory is inferred by the VHDL code, this memory is implemented using memory blocks characteristic for a given family (of the size of 4 kbits in Cyclone II, 9 kbits in Cyclone III and IV, and of the sizes of 512 bytes, 4 kbits, or 512 kbits in Stratix II). In our analysis below we neglect these memory requirements, as typically they amount to a relatively small percentage of memory resources available in target FPGAs.

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,

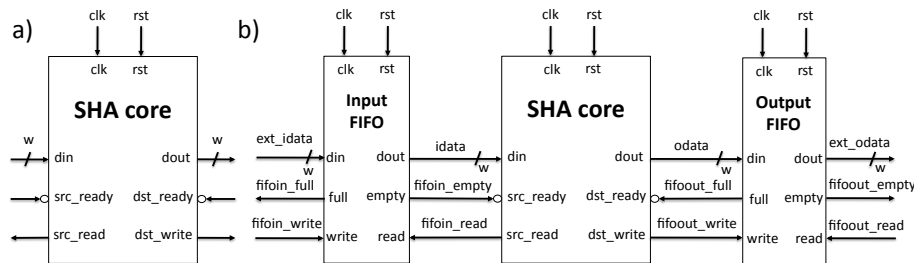


Figure 2.1: a) Input/output interface of a SHA core. b) A typical configuration of a SHA core connected to two surrounding FIFOs.

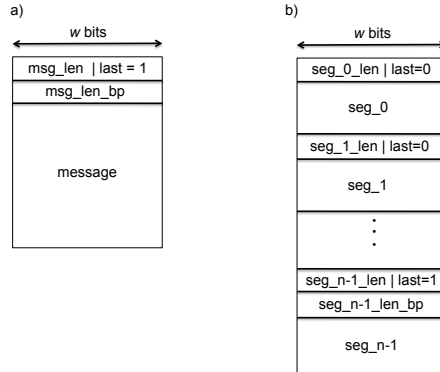


Figure 2.2: Format of input data for two different operation scenarios: a) with message bitlength known in advance, and b) with message bitlength unknown in advance. Notation: msg_len – message length after padding, msg_len_bp – message length before padding, $\text{seg}_i.\text{len}$ – segment i length after padding, $\text{seg}_i.\text{len_bp}$ – 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 [20].

Please note that scenario b) is very similar to the way data is processed by a typical software API for hash functions, such as [21]. 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) [22] in SIMD, linear code (LC) [23] 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) [22]. 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 [24], 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 [25]). 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 [26]. 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) [26]. This benchmarking environment was also used to optimize requested synthesis and implementation frequencies and other tool options.

Chapter 3



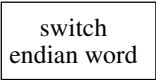
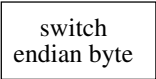
Comprehensive Designs of SHA-3 Candidates

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 [27–41].

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.
$\ll\ll R$	Rotation left by R positions. If R is a constant: fixed rotation; if R is a variable: variable rotation implemented using a barrel rotator.
$\gg\gg R$	Rotation right by R positions. If R is a constant: fixed rotation; if R is a variable: variable rotation implemented using a barrel rotator.
$\ll\ll S$	Shift left by S positions. If S is a constant: fixed shift; if S is a variable: variable shift implemented using a barrel shifter.
$\gg\gg S$	Shift right by S positions. If S is a constant: fixed shift; if S is a variable: variable shift implemented using a barrel shifter.
	Concatenation. By default, the buses concatenate back to the same arrangement as before the separation (split) occurs.
	Serial-in-parallel-out unit.
	Parallel-in-serial-out unit.
	Wordwise endianness switching.
	Byte-wise 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 $\text{GF}(2^8)$, SubBytes, Mix-Columns, and AES Round.

3.2.1 Multiplication by 2 in the Galois Field $\text{GF}(2^8)$

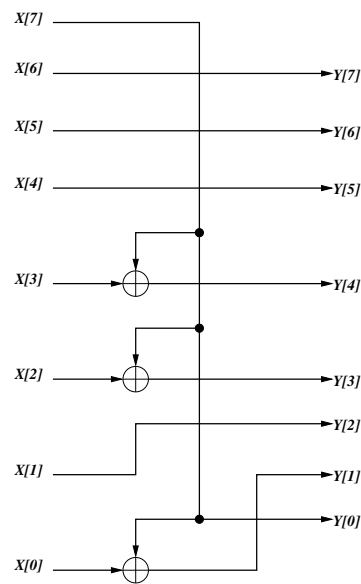


Figure 3.1: Basic : x2

3.2.2 Multiplication by n in the Galois Field $\text{GF}(2^8)$

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

Table 3.2: Galois Field Multiplication by n

$$\begin{aligned} x3(X) &= x2(X) \oplus X \\ x4(X) &= x2(x2(X)) \\ x5(X) &= x4(X) \oplus X \\ x6(X) &= x4(X) \oplus x2(X) \\ x7(X) &= x4(X) \oplus x3(X) \end{aligned}$$

3.2.3 AES

AES is a basic building block of many SHA-3 candidates. An AES round consists of three basic operations, SubBytes, ShiftRows, and MixColumns as shown in Figure 3.2. The SubBytes operation, shown in Figure 3.3, performs direct substitution on all bytes of its input. MixColumns performs multiplication of a constant 4x4 matrix by each word of its input. A word of AES contains 32 bits or 4 bytes. Hence, four instances of MC (MixColumn) are required to process the entire AES block of 128 bits, as shown in Figure 3.4 . The SBOX and ShiftBytes operations of AES and their full specifications can be found in [42] and [43].

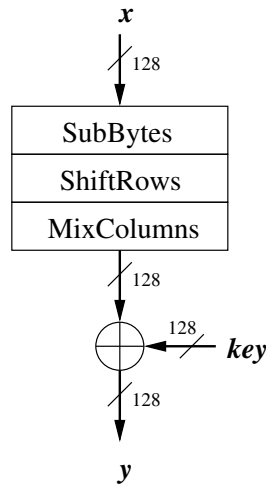


Figure 3.2: Basic : AES Round

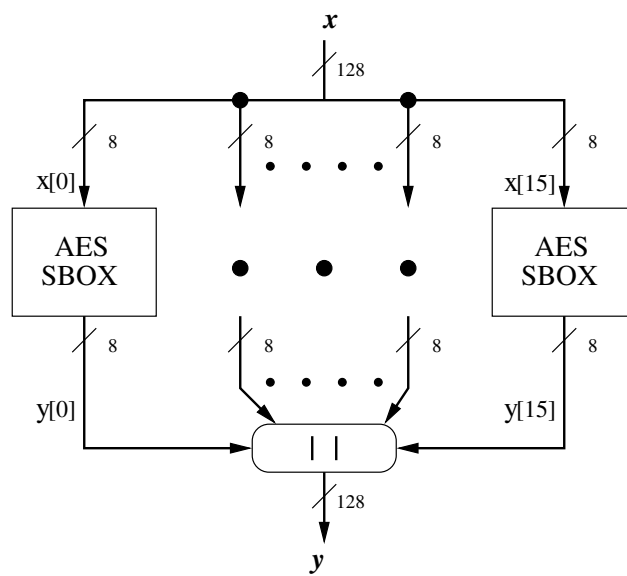


Figure 3.3: Basic : AES SubBytes

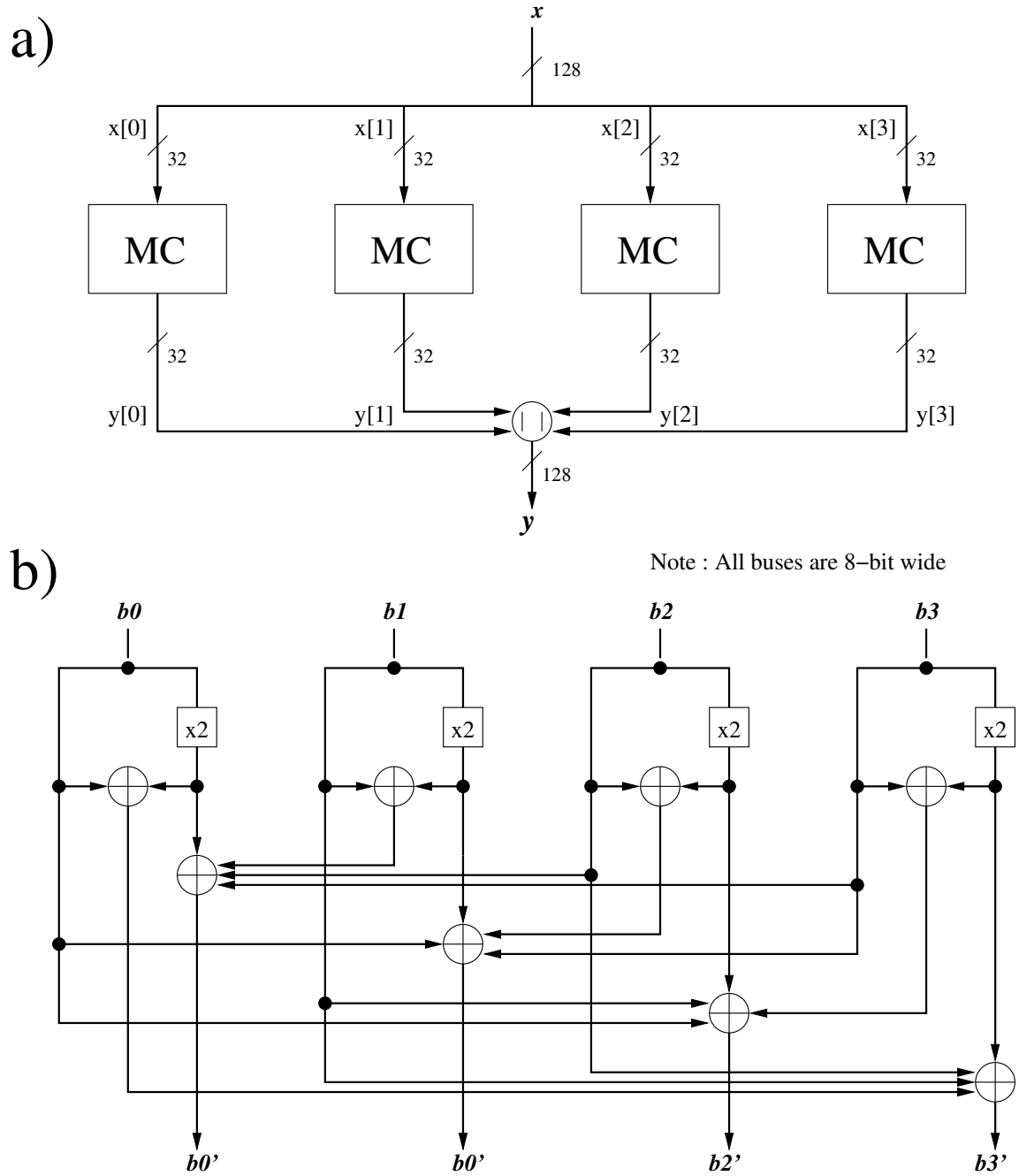


Figure 3.4: Basic : AES MixColumns. a) Applying MixColumns to the entire AES block, and b) MC: Applying MixColumns to a single word.

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 [27]. 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:

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

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.

BLAKE-32 : $b=512, h=256$
 BLAKE-64 : $b=1024, h=512$

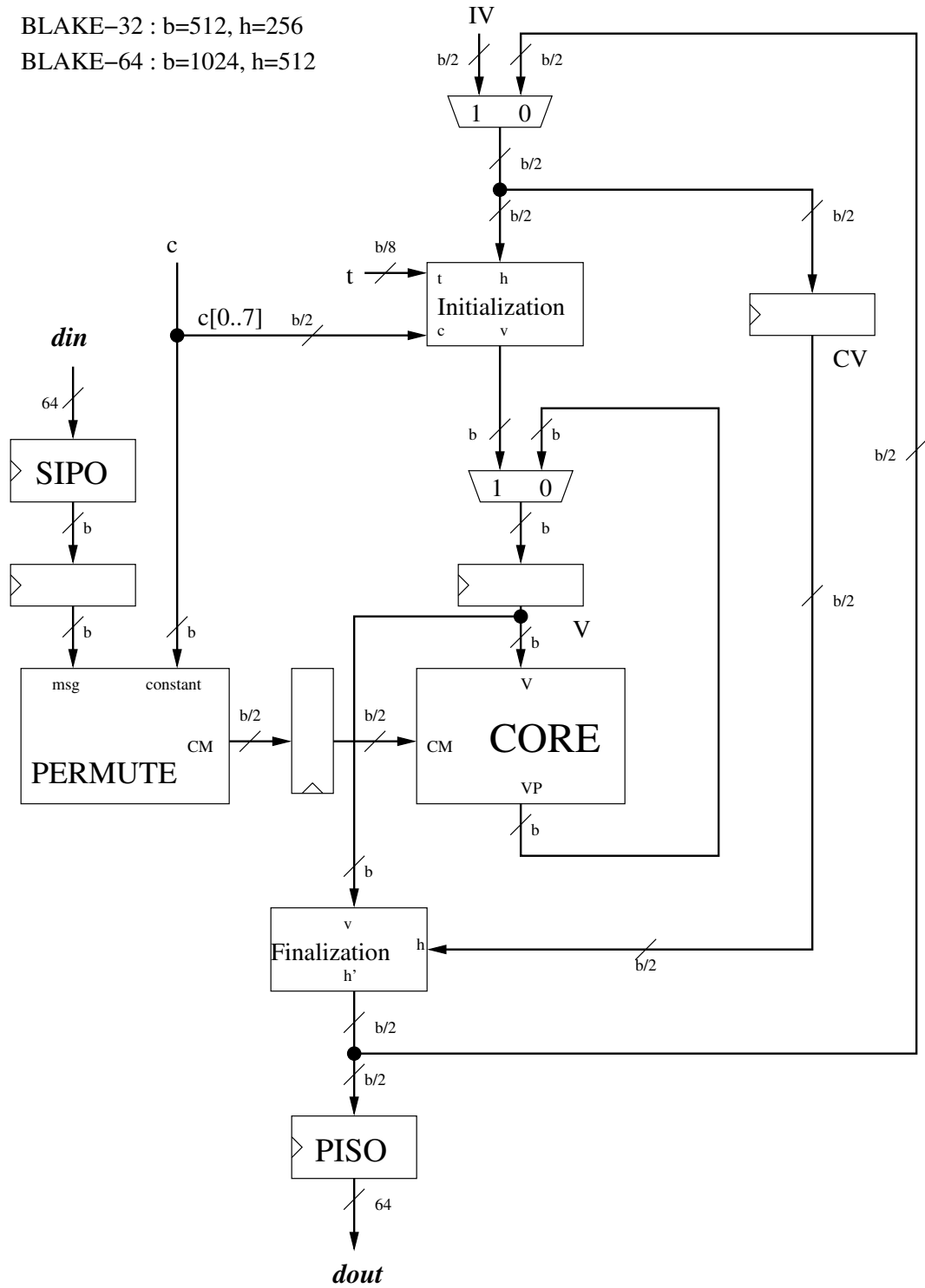


Figure 3.5: BLAKE : Datapath

Table 3.3: BLAKE : Permutation Constants

	<i>hi</i>								<i>low</i>							
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 [27], 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 transforms the state matrix (output of 4 parallel G functions) into a new matrix appropriate for the second half-round. LVL1 is a permutation inverse to LVL2.

Table 3.4: Blake : Half Round Permutations

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

LVL1 (inverse)																
input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
output	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

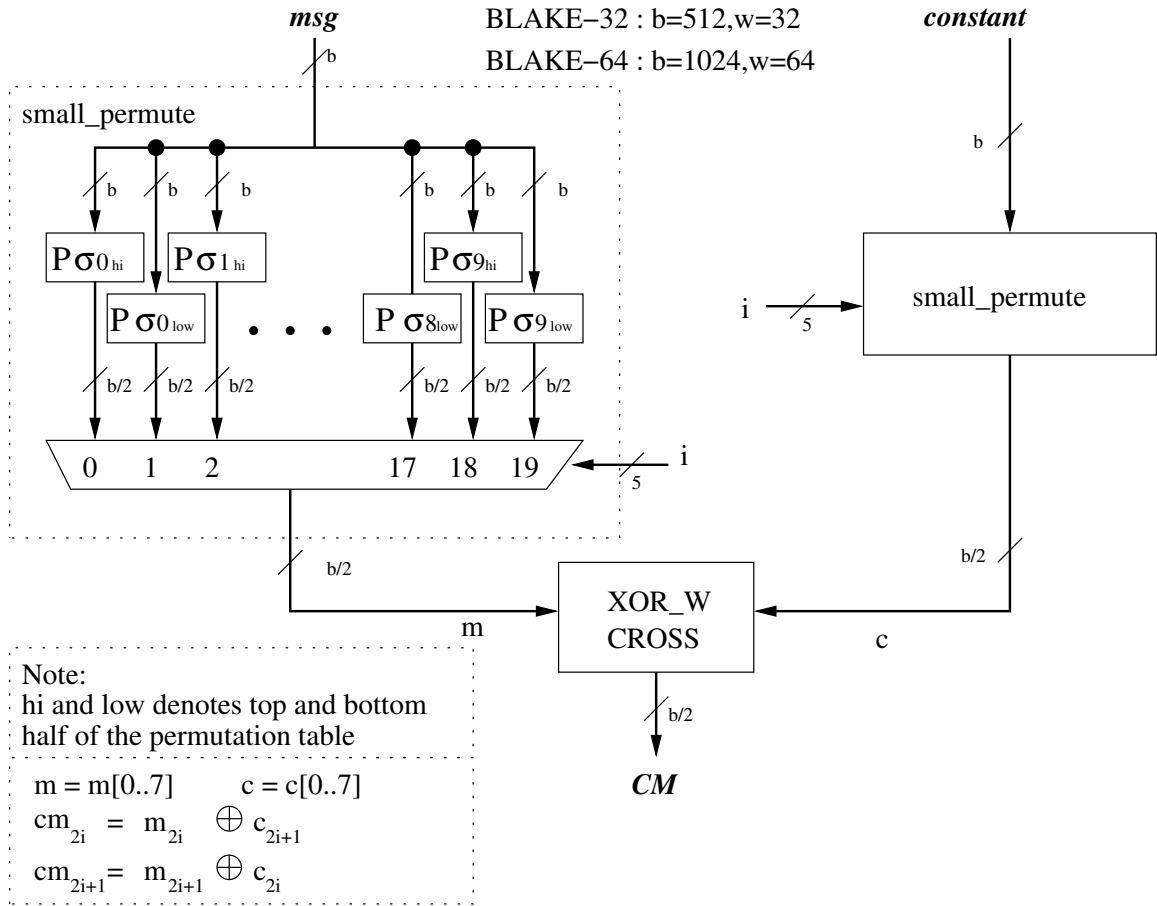


Figure 3.6: BLAKE : PERMUTE

Table 3.5: BLAKE : Rotation Constants of BLAKE-32

R1	R2	R3	R4
16	12	8	7

be found in Section 2.2.1 of [27]. 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.

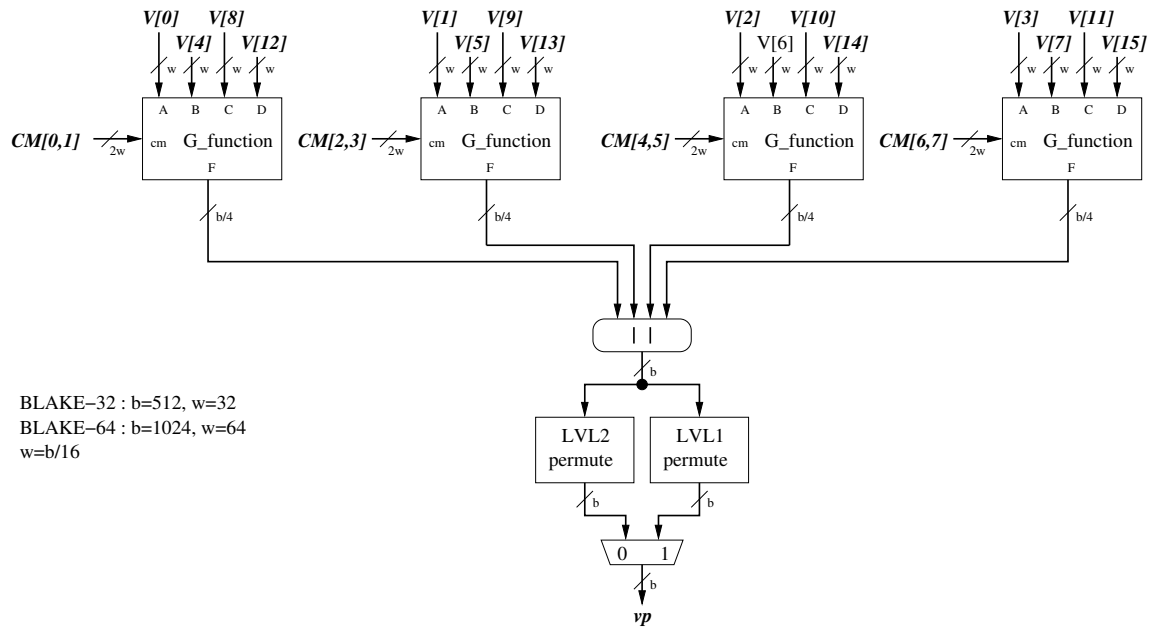


Figure 3.7: BLAKE : CORE

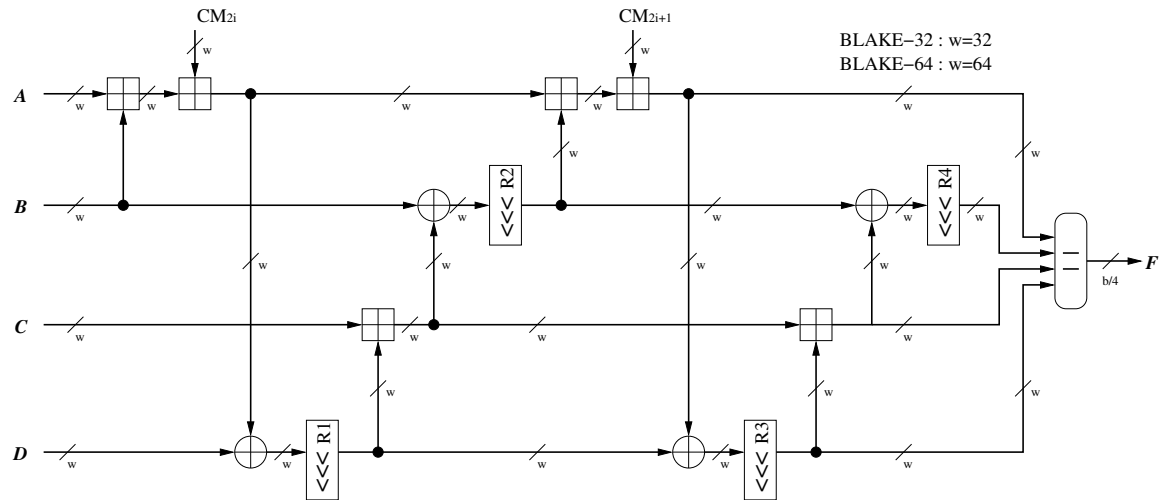


Figure 3.8: BLAKE : G-function

Table 3.6: BLAKE: Rotating Constants of BLAKE-64

R1	R2	R3	R4
32	25	16	11

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 [28].

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 [28]. All other operations remain the same.

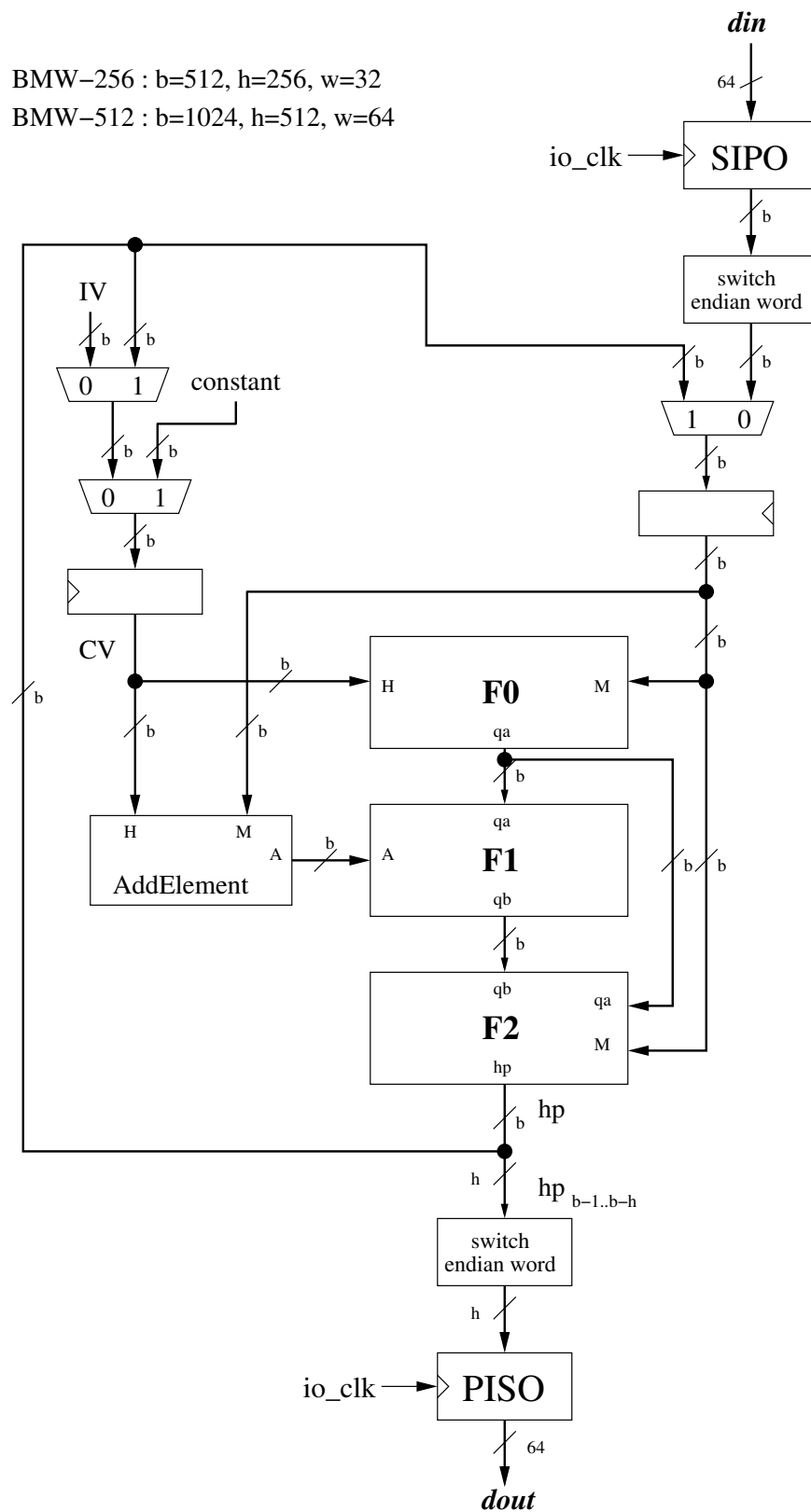


Figure 3.9: BMW : Datapath

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 bitwise 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.

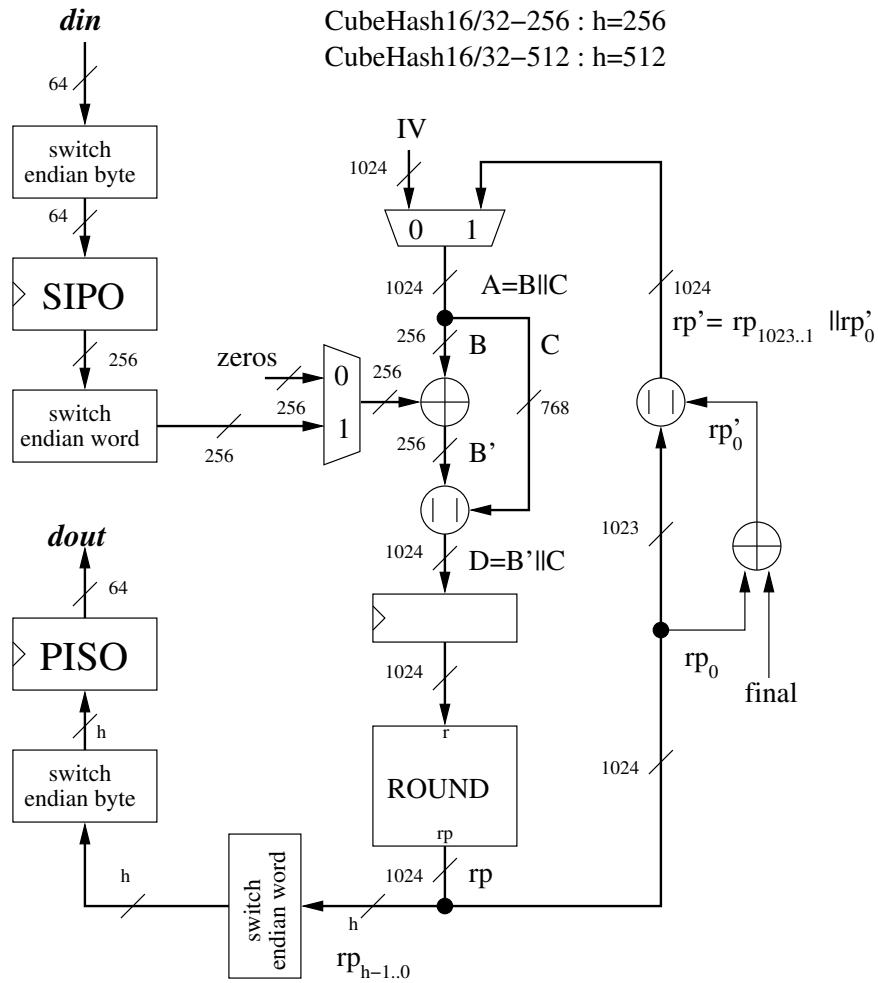


Figure 3.10: CubeHash : Datapath

Note : All operations are performed wordwise, with $w=32$

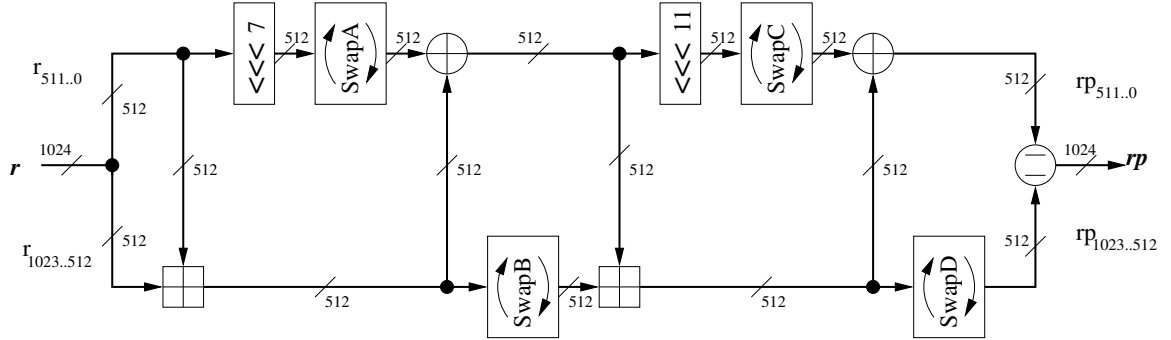
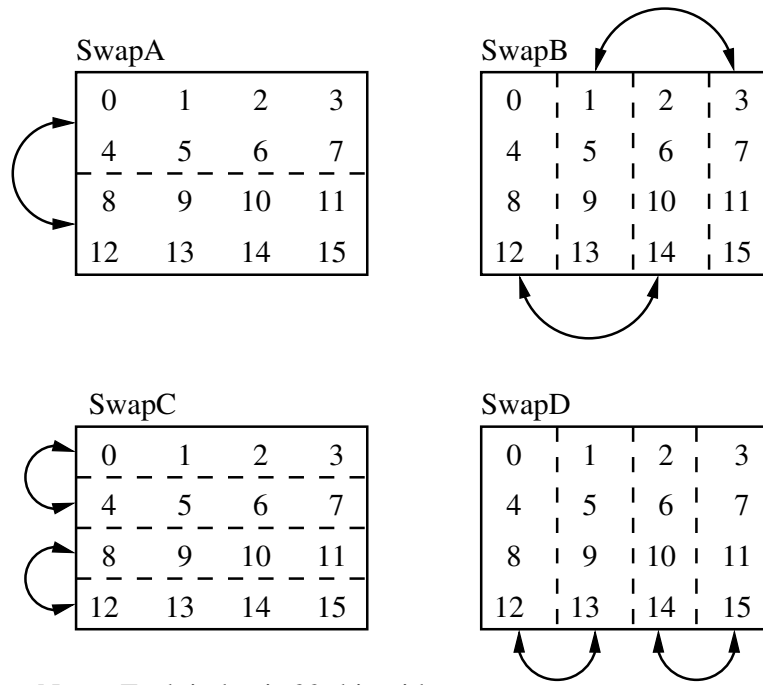


Figure 3.11: CubeHash : Round



Note : Each index is 32-bit wide

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 8 iterations of the ECHO Round. 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:

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

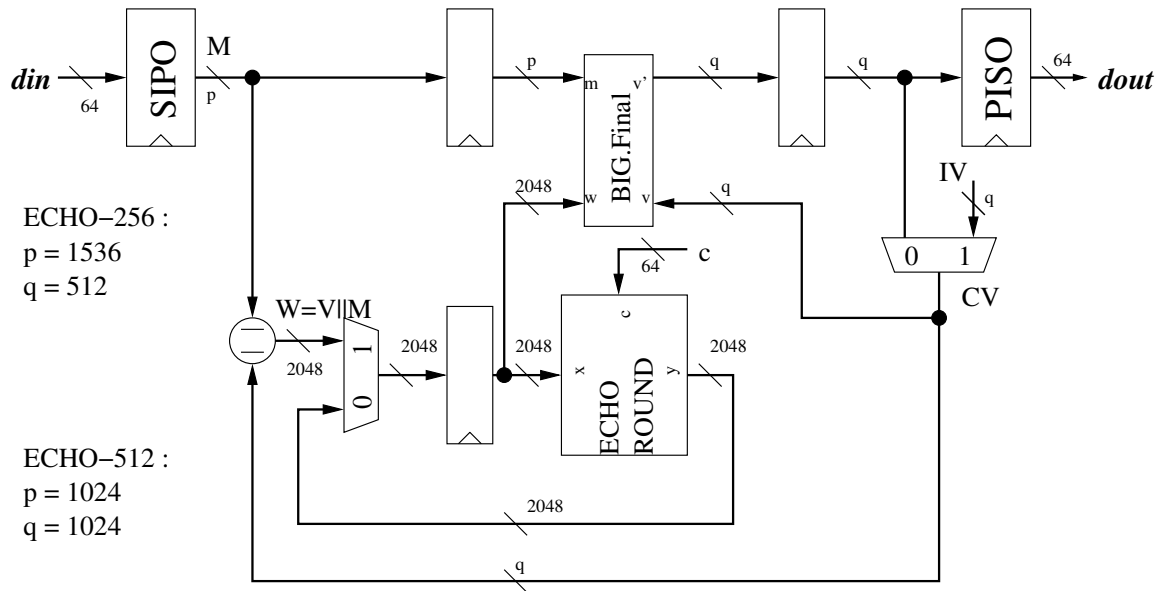


Figure 3.13: ECHO : Datapath

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

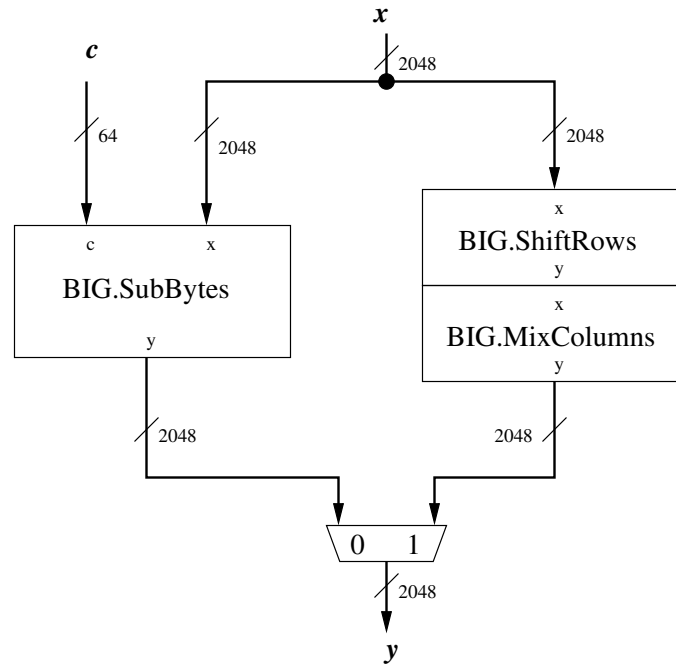


Figure 3.14: ECHO : Round

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.

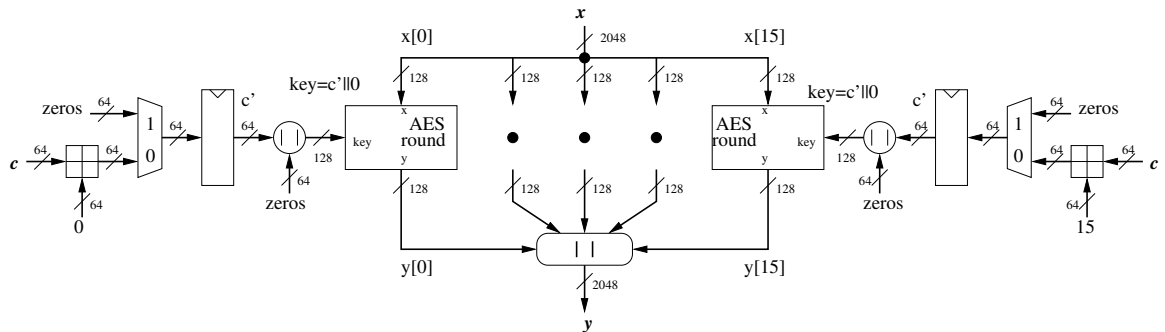


Figure 3.15: ECHO : BIG.SubBytes

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 MixColumns. All data is then combined together to produce the final state.

Table 3.7: ECHO : BIG.ShiftRows

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

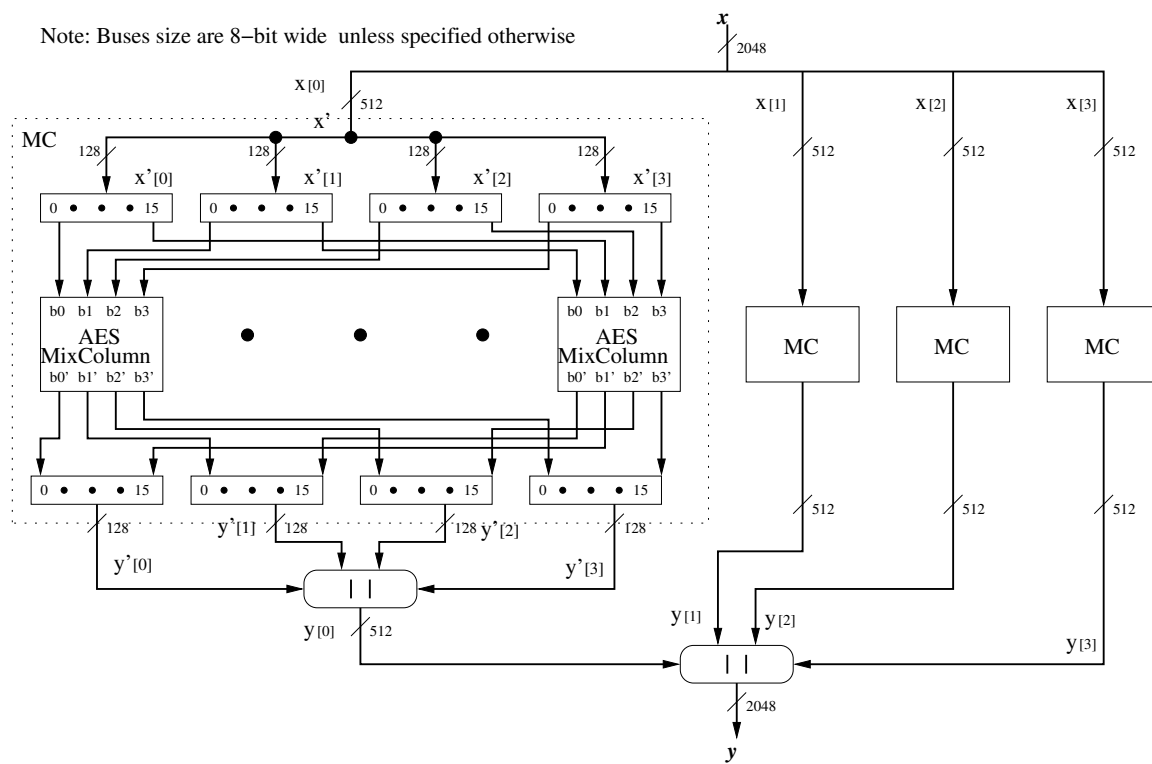


Figure 3.16: ECHO : BIG.MixColumns

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. The number of rounds changes from 8 to 10. All other

operations, except BIG.Final, remain the same. The BIG.Final operation of ECHO-512 is described below:

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

3.7 Fugue

3.7.1 Block Diagram Description

Fugue-256 : $b=960, h=256$ size of 0s : $b-h$
 Fugue-512 : $b=1152, h=512$ size of IV : h

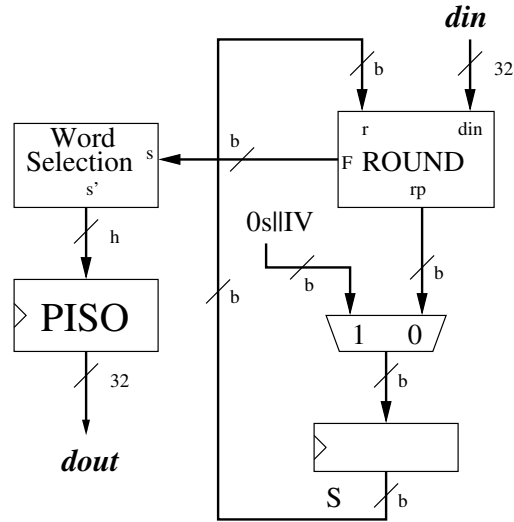


Figure 3.17: Fugue : Datapath

In Figure 3.17, the datapath of Fugue is shown. For every message, the state register is initialized to zeros concatenated with the h -bit IV. The state is viewed as a matrix of 4 by X bytes, where X is the number of columns in the matrix, dependent on the block size of Fugue. For Fugue-256, 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 ADDF0 followed by Word Selection. For Fugue-256, Word Selection is described below:

$$S' = \begin{matrix} S[1..3] || S[4] || \\ S[15] || S[16..18]. \end{matrix}$$

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 Fugue-256 in [31]. TIX operates in parallel as follows:

$$\begin{aligned} S'[0] &= din \\ S'[1] &= S[1] \oplus S[24] \\ S'[8] &= S[8] \oplus din \\ S'[10] &= S[10] \oplus S[0]. \end{aligned}$$

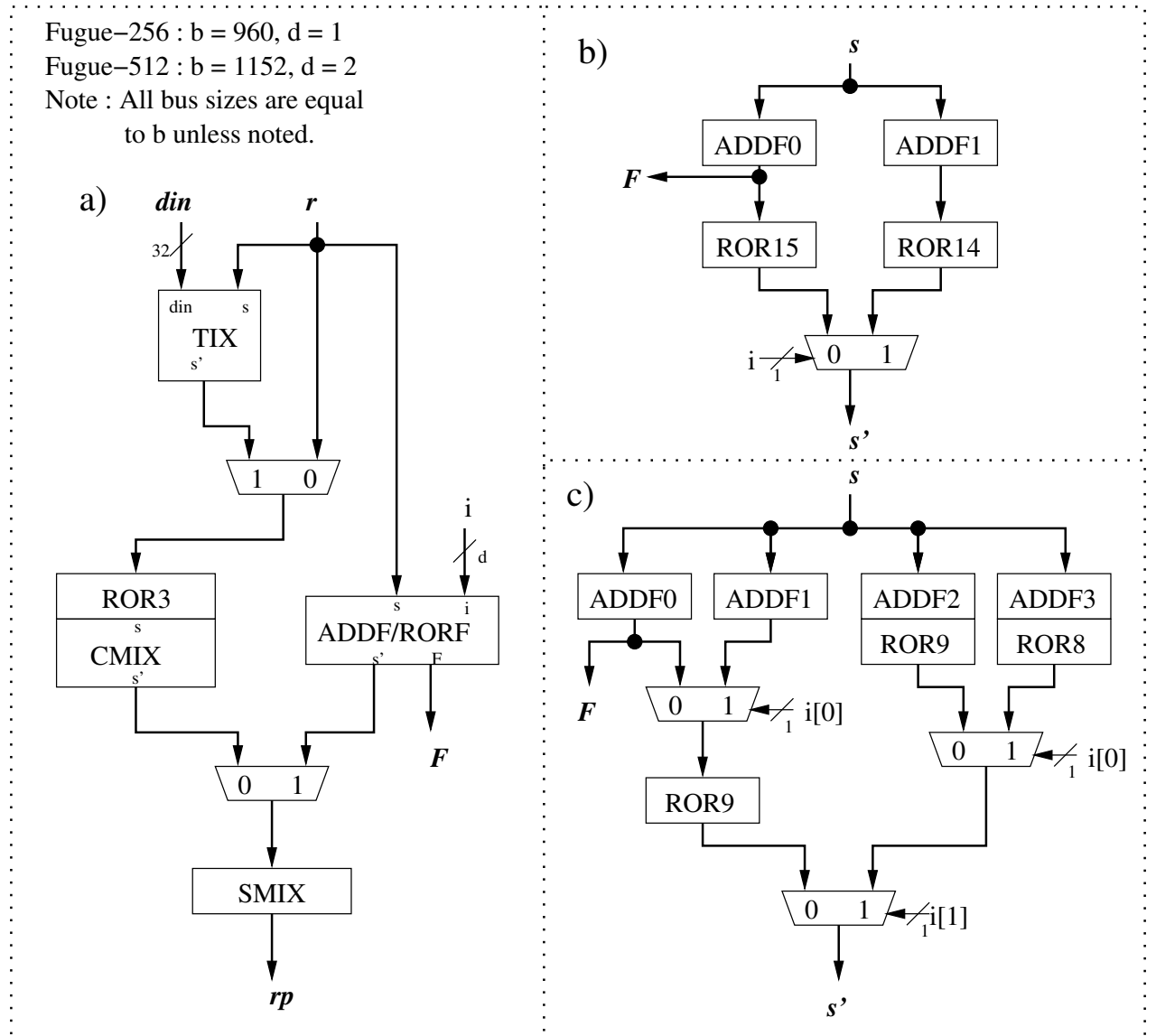


Figure 3.18: Fugue : a) Round, b) ADDF/RORF for Fugue-256, c) ADDF/RORF for Fugue-512.

Table 3.8: Fugue-256: ADDFi Operation

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

ROR3 and CMIX are performed consecutively.

RORn sets

$$S'[i] = S[(i - n) \bmod 30], \text{ for } i = 0..29 \text{ in Fugue-256, and}$$

$$S'[i] = S[(i - n) \bmod 36], \text{ for } i = 0..35 \text{ in Fugue-512.}$$

As such, RORn can be considered as $\ggg (n * 32)$.

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 ADDF0 and ADDF1 operations are defined in Table 3.8.

Finally, the SMIX operation is described in Figure 3.19. The SMIX operation affects only the first 128-bits of the state. These 128 bits are split 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 bytes, which is equivalent to 1152 bits. Additionally, TIX, CMIX, ADDF/RORF and Word Selection have been modified. TIX is now performed in parallel as follows:

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				

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

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 ADDFi operations, for $i = 0..3$ are defined in Table 3.10.

Finally, Word Selection is performed as follows:

$$S' = \begin{matrix} S[1..3] || S[4] || S[9] || S[10..12] || \\ S[18] || S[19..21] || S[27] || S[10..12]. \end{matrix}$$

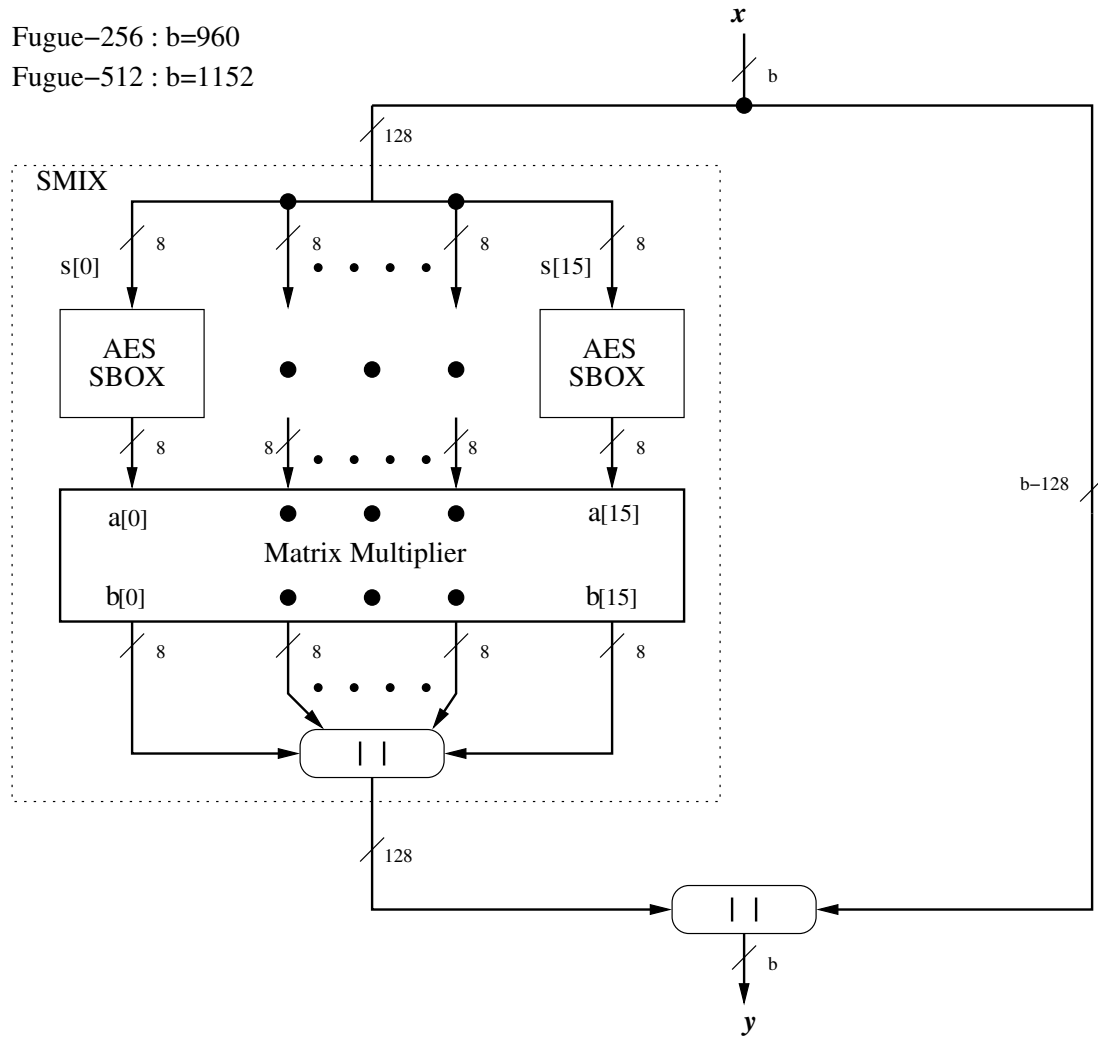


Figure 3.19: Fugue : SMIX

Table 3.10: Fugue-512: ADDFi Operation

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

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.

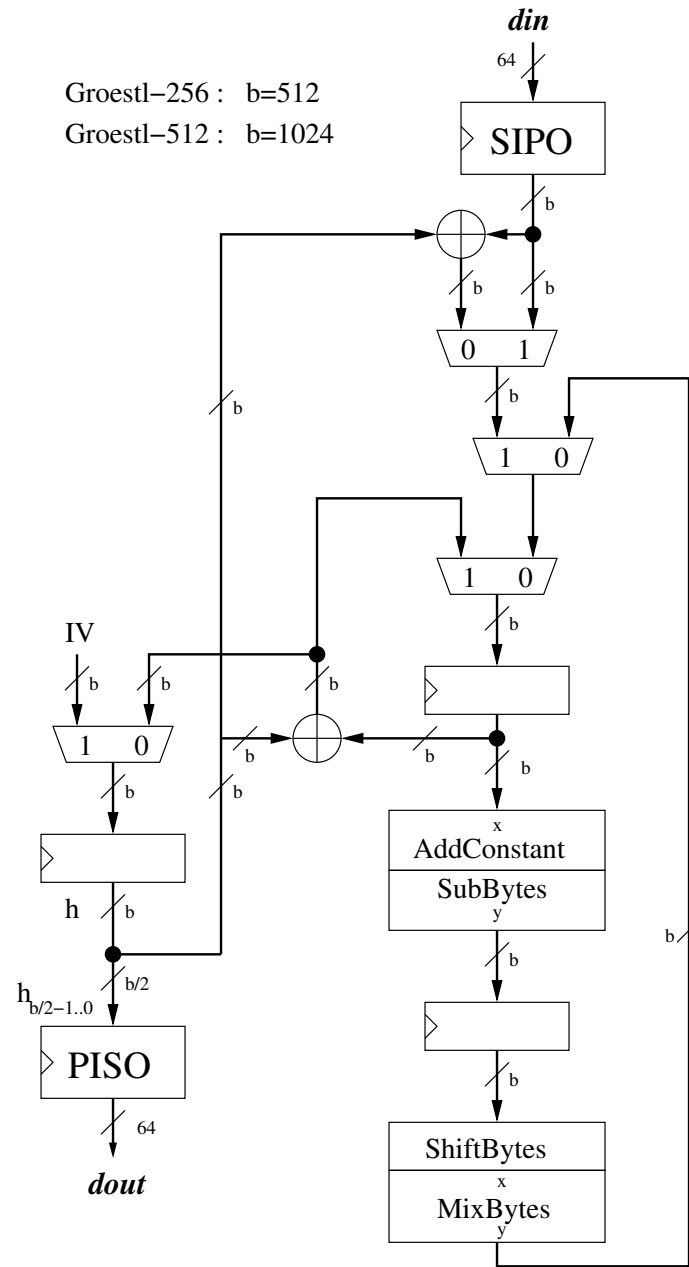


Figure 3.20: Groestl : Datapath

Groestl-256 : $b=512$
 Groestl-512 : $b=1024$

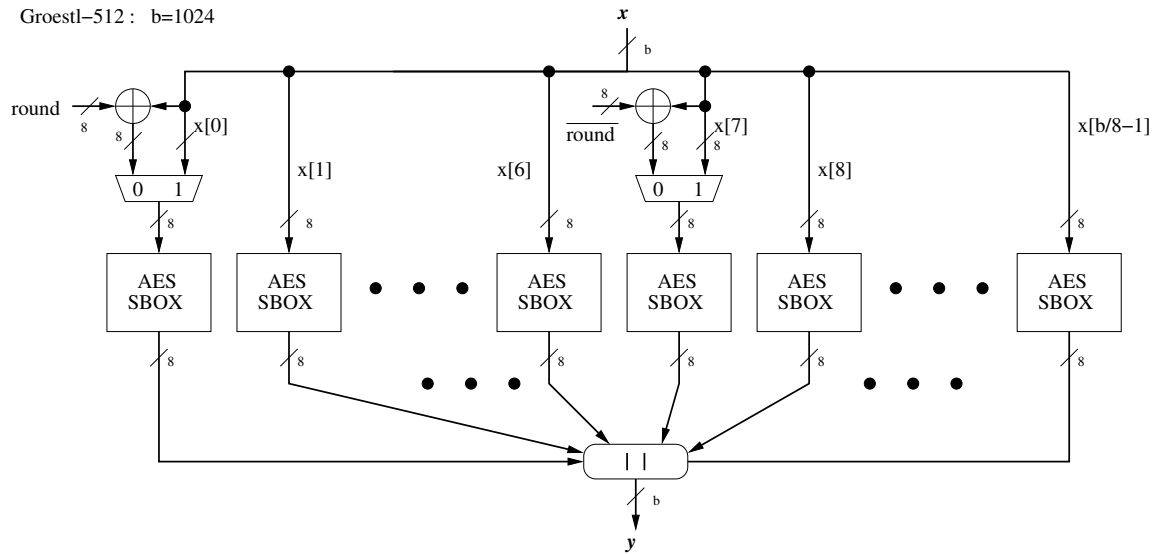


Figure 3.21: Groestl : AddConstant and SubBytes

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.

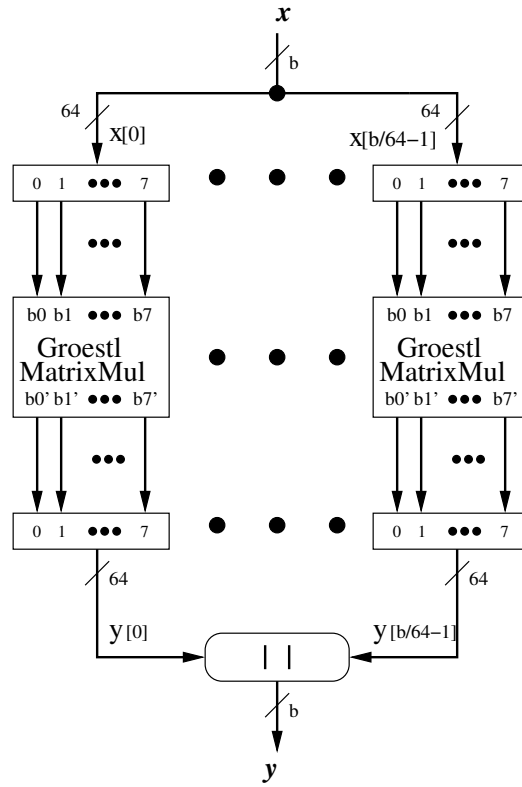


Figure 3.22: Groestl : MixBytes

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 [33].

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:

$$\begin{aligned} s'[1] &= s[1] \oplus \alpha[1] \oplus c \\ s'[i] &= s[i] \oplus \alpha[i] \\ &\text{for } i=0, 2..15. \end{aligned}$$

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:

$$\begin{aligned} (s[0], s[5], s[10], s[15]) &= L(s[0], s[5], s[10], s[15]) \\ (s[1], s[6], s[11], s[12]) &= L(s[1], s[6], s[11], s[12]) \\ (s[2], s[7], s[8], s[13]) &= L(s[2], s[7], s[8], s[13]) \\ (s[3], s[4], s[9], s[14]) &= L(s[3], s[4], s[9], s[14]) \end{aligned}$$

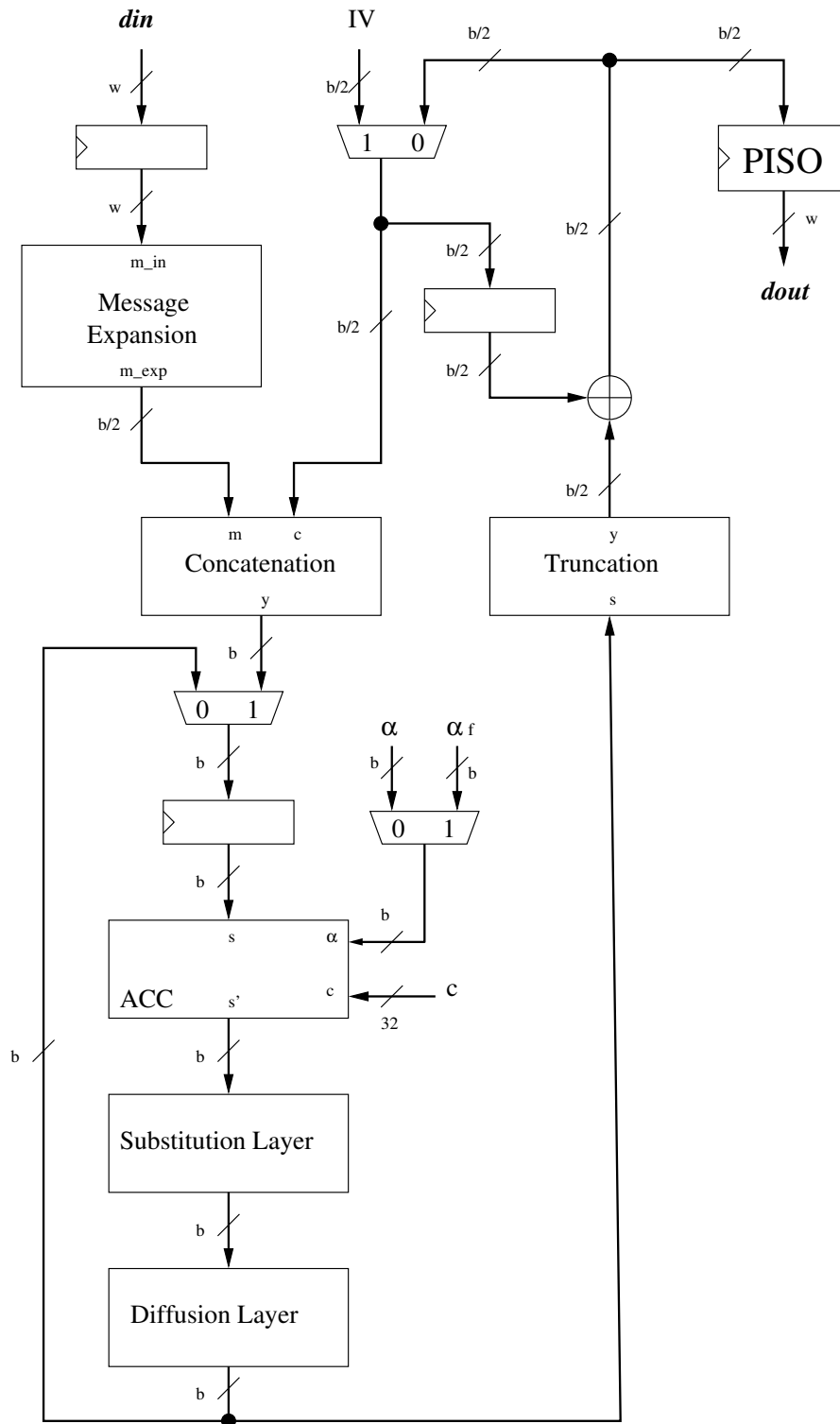
Hamsi-256 : $b=512, w=32$ Hamsi-512 : $b=1024, w=64$ 

Figure 3.23: Hamsi : Datapath

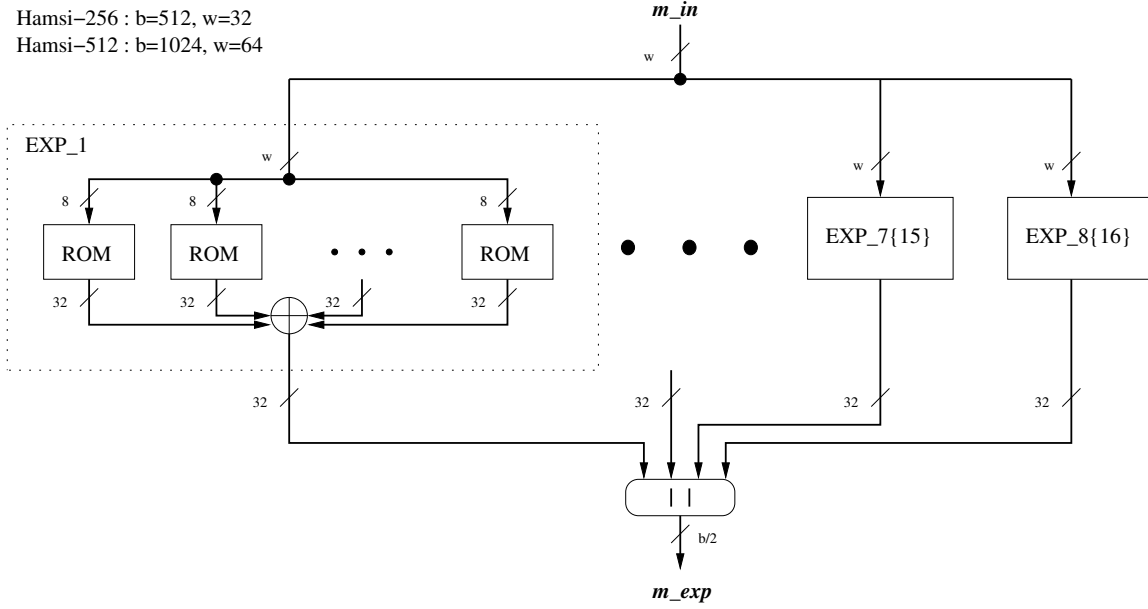


Figure 3.24: Hamsi : Message Expansion

Finally, Truncation is performed as follows:

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

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, Addition of Constants and Counter, Diffusion Layer, and Truncation.

Concatenation of Hamsi-512 is performed as follows:

$$y = m[0..1] \parallel c[0..3] \parallel m[2..5] \parallel c[4..7], m[6..9] \parallel c[8..9] \parallel m[10..11] \parallel c[10..13] \parallel m[12..13] \parallel c[14..5] \parallel m[14..15]$$

The Addition of Constants and Counter step is defined as follows:

$$\begin{aligned} s'[1] &= s[1] \oplus \alpha[1] \oplus c \\ s'[i] &= s[i] \oplus \alpha[i] \\ &\text{for } i=0, 2..31. \end{aligned}$$

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

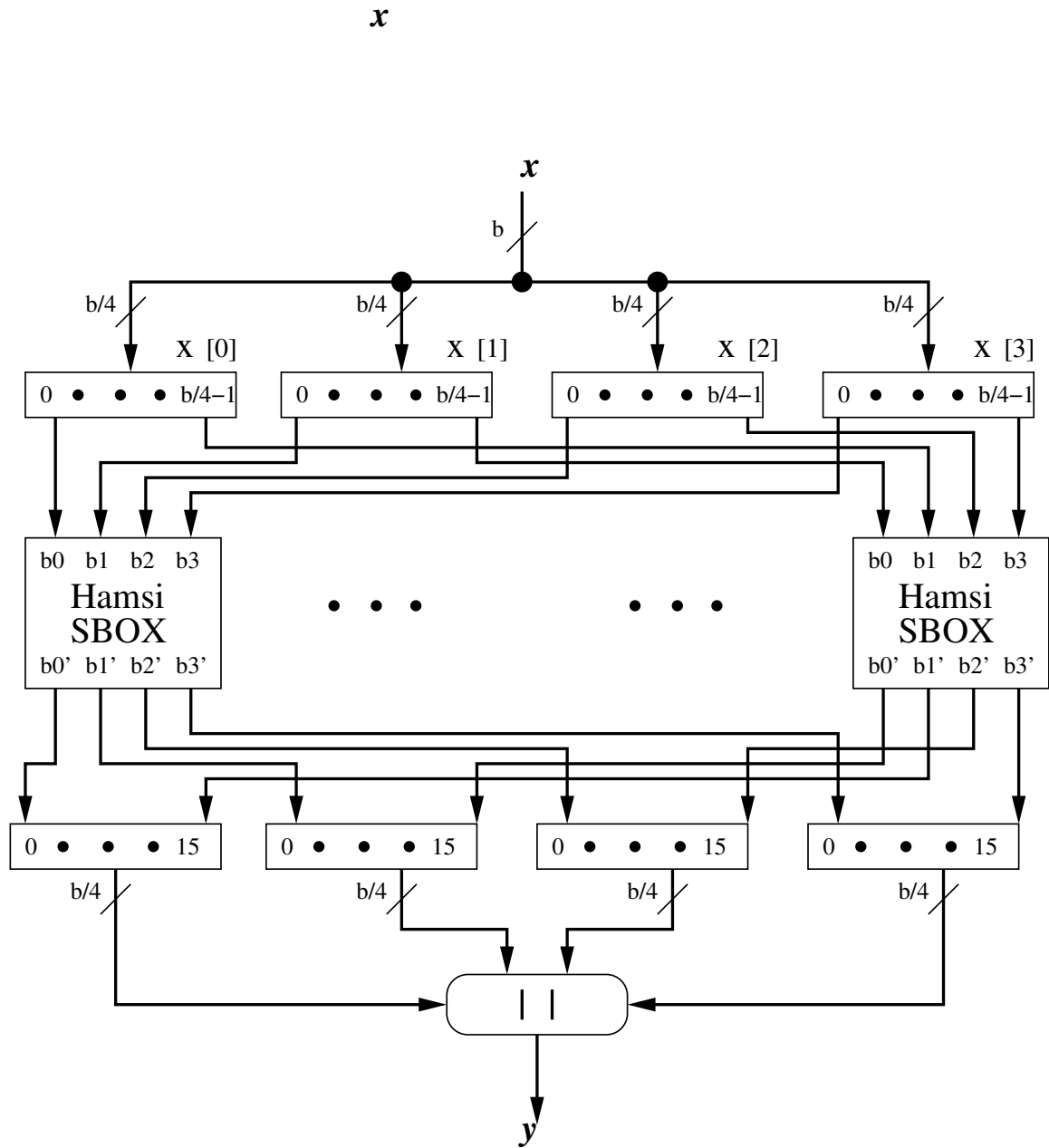


Figure 3.25: Hamsi : Substitution Layer

Note : All bus sizes are 32 bits

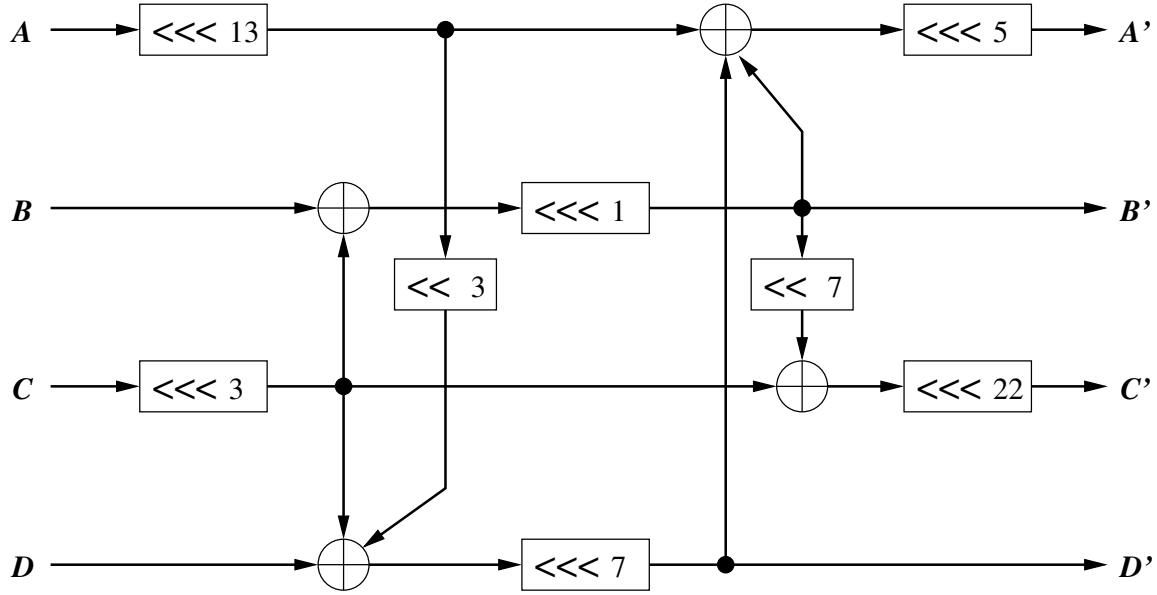


Figure 3.26: Hamsi : L

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

Truncation of Hamsi-512 is performed as follows:

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

3.10 JH

3.10.1 Block Diagram Description

The block diagram of JH is shown in Figure 3.27. At the beginning of computations, the three registers shown in the diagram are initialized respectively with C_{IV} , first message block, and an initial state dependent on IV and the first message block. The internal state, held by the rightmost register, is transformed using the round transformation $R8$ for 36 rounds. In each of these rounds, a different round constant, Cr , generated using transformation $R6$, is used. Once processing is completed, the output from $R8$ is degrouped and its lower half is xored with the message block stored in the temporary register, to create a new chaining value, dgc . If there are more message blocks, the upper half of the chaining value is xored with the next block, and the result processed through the group transformation. 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 shown in Figure 3.28.

In Figure 3.29, a generic description of a JH round R_n is shown. The same unit is used for $R6$ and $R8$. The differences are the sizes of the input r , and the round constant Cr . These sizes are equal to y and $y/4$ bits respectively, where $y = 4 \cdot 2^8 = 1024$ for $R8$ and $y = 4 \cdot 2^6 = 256$ for $R6$. In the JH Round R_n , an input is viewed as an array of 2^n 4-bit blocks. These blocks go through either S_0 or S_1 S-boxes, defined in Table 3.13.

Next, outputs from these S-boxes are selected by the corresponding bit of the round constant, Cr . Two consecutive outputs form an input to the linear transformation unit, L . A diagram of this unit is shown in Figure 3.30. The transformed outputs are then permuted by the PERMUTE unit. 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 round constant, respectively.

```

for i = k/4-1 downto 0 do
  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 for;
for i = k/2-1 downto 0 do
  b(i) <= a(i*2);
  b(i + k/2) <= a(i*2 + 1);
end for;
for i = k/2-1 downto 0 do
  y(i) <= b(i);
end for;
for i = k/4-1 downto 0 do
  y(i*2 + k/2) <= b(i*2 + 1 + k/2);
  y(i*2 + 1 + k/2) <= b(i*2 + k/2);
end for;

```

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.

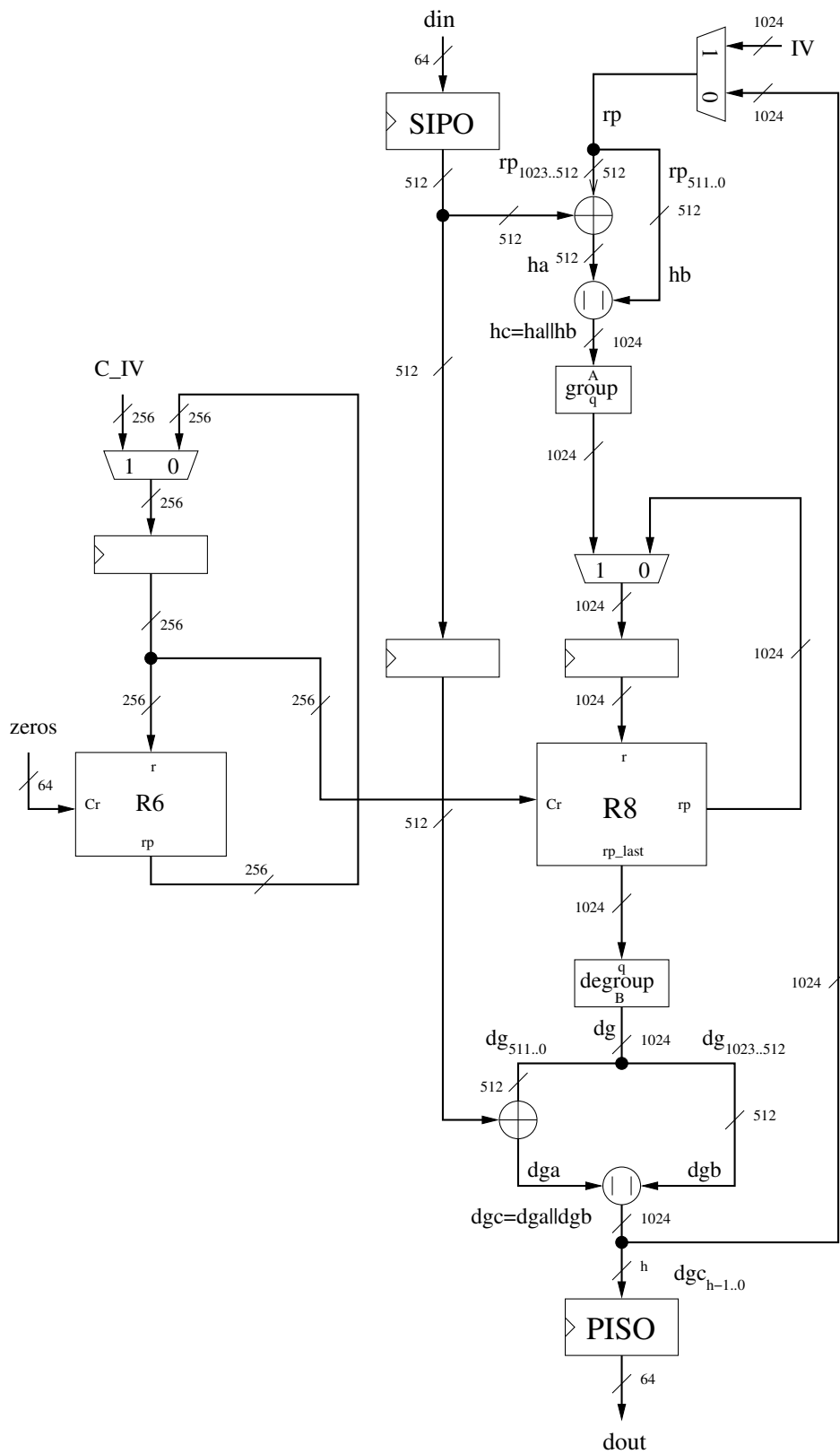
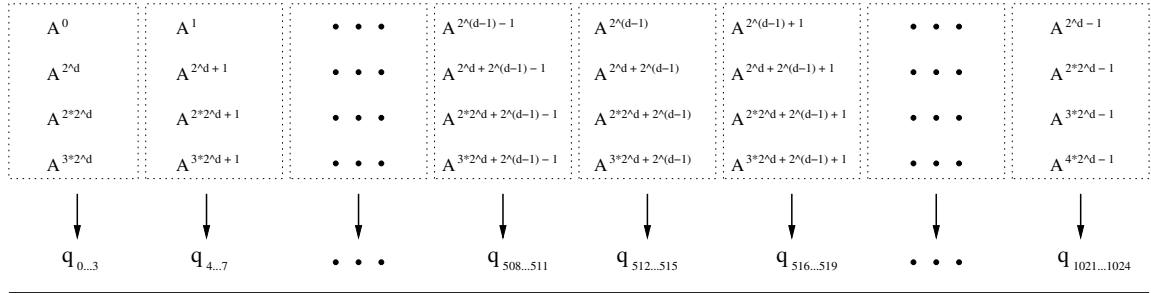


Figure 3.27: JH : Datapath

a) grouping



b) de-grouping

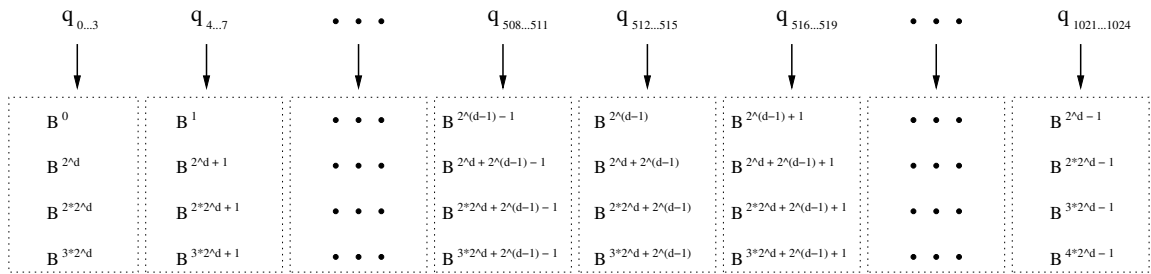


Figure 3.28: JH : Group and Degroup Operations

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

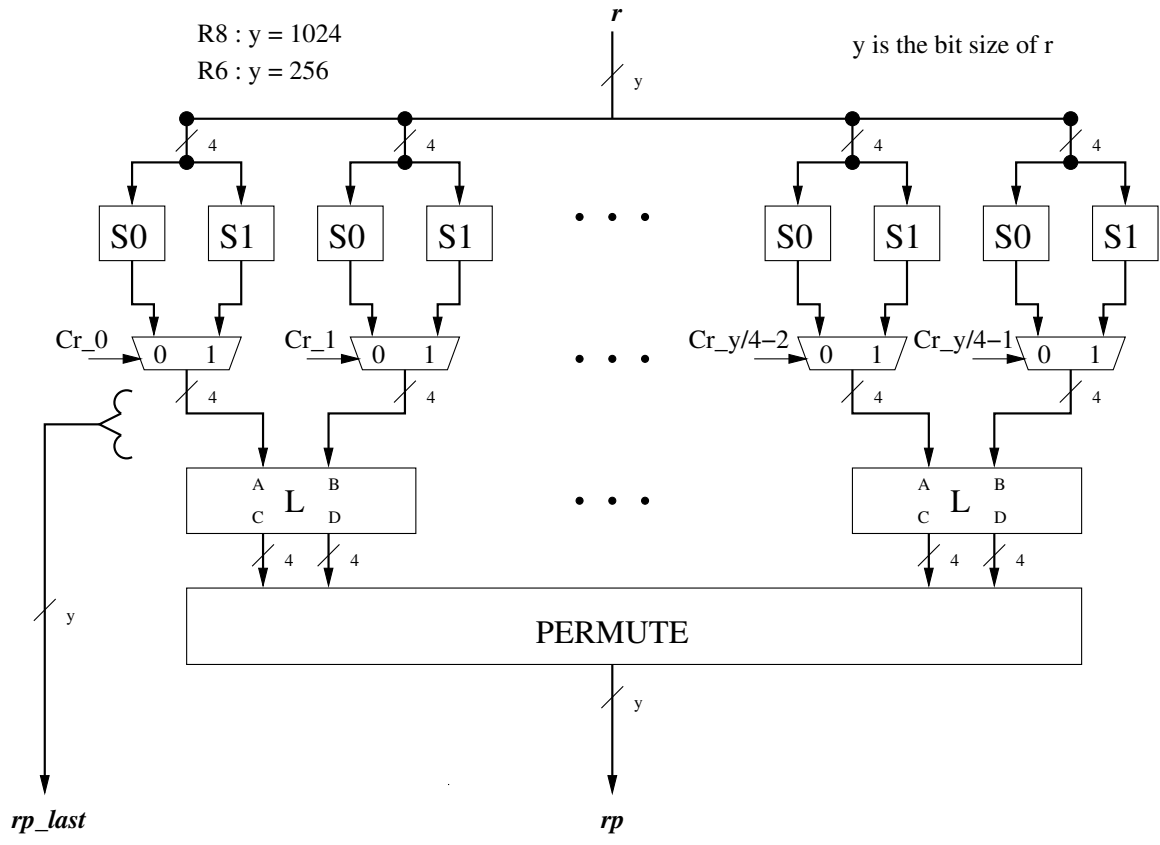
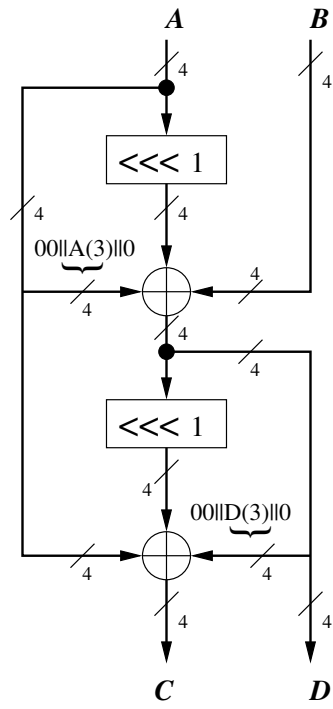


Figure 3.29: JH : R_n

Figure 3.30: JH : Linear Transformation

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 3.31. 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 3.32. 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 3.33, 3.34.

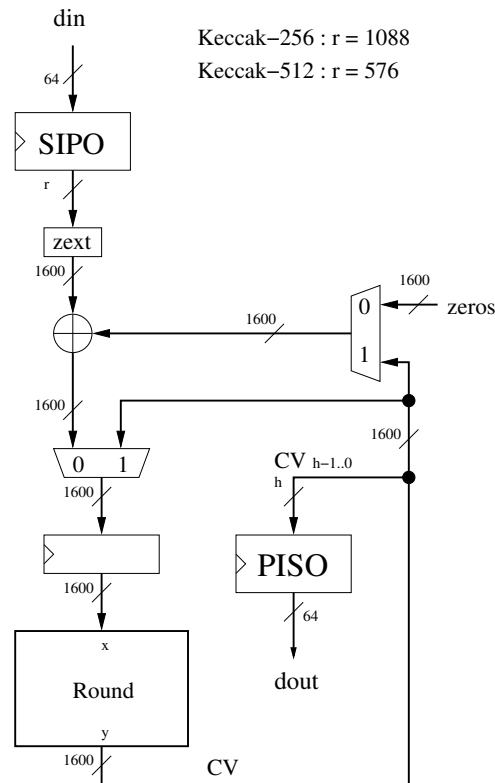


Figure 3.31: Keccak : Datapath

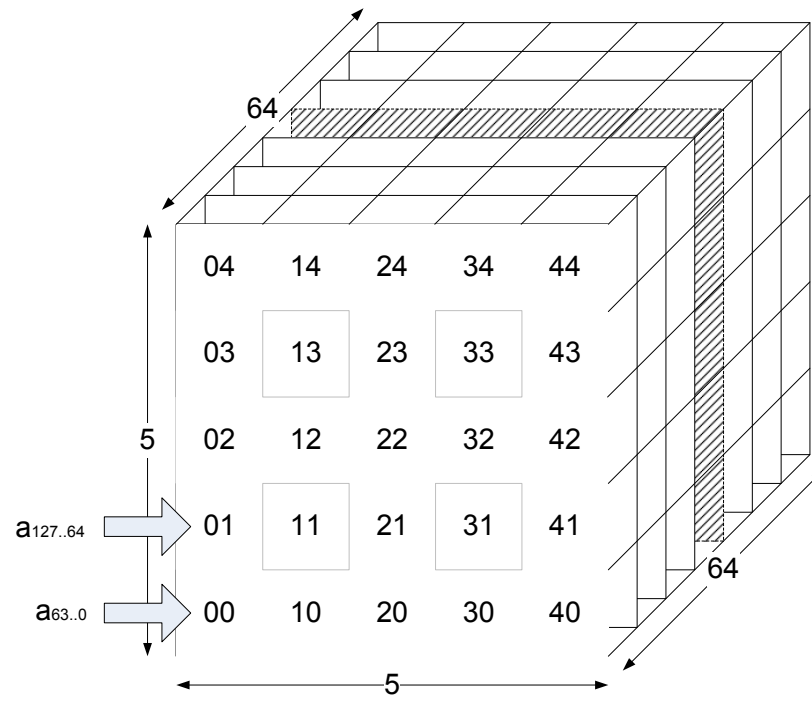


Figure 3.32: Keccak : State

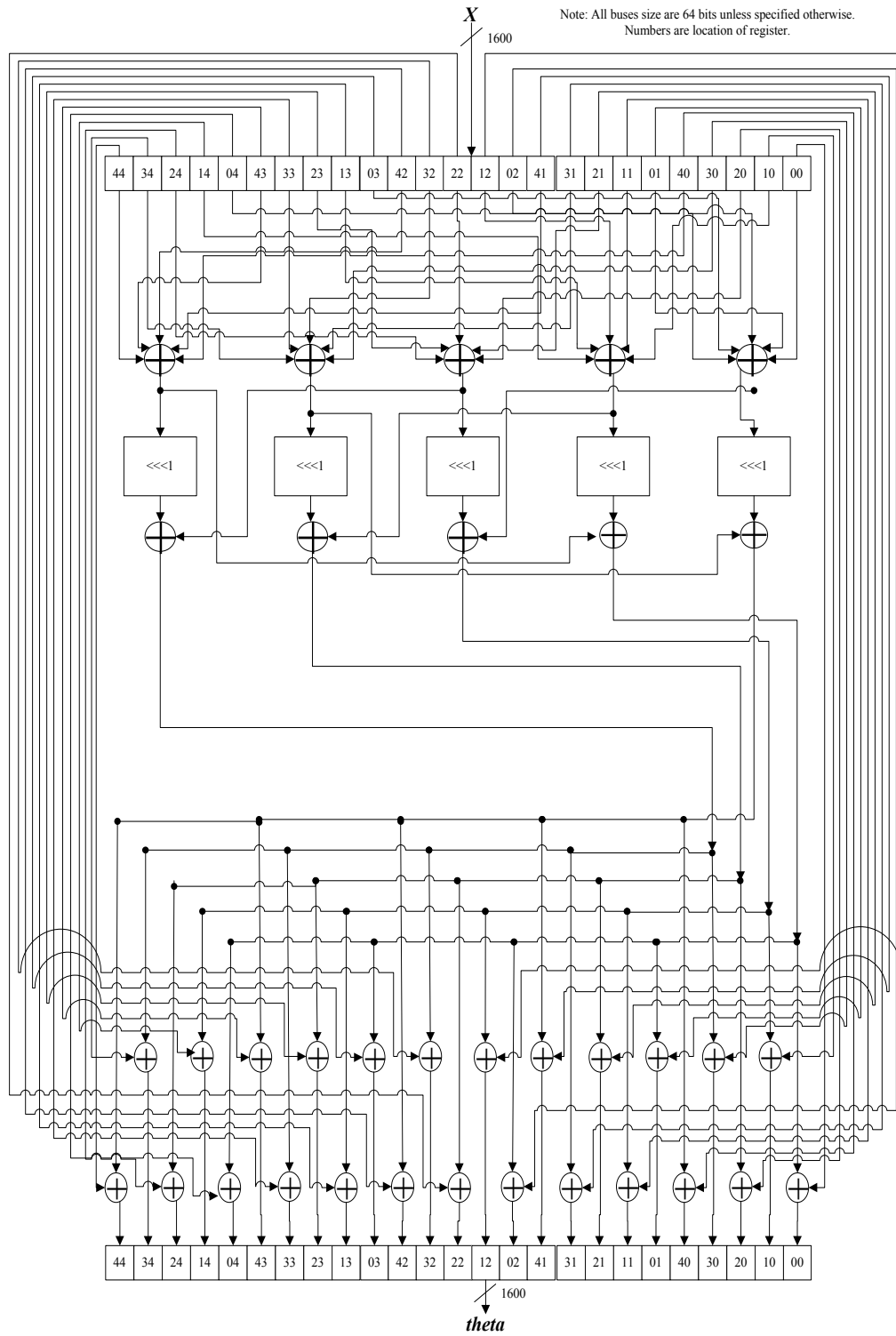


Figure 3.33: Keccak : Round Part 1

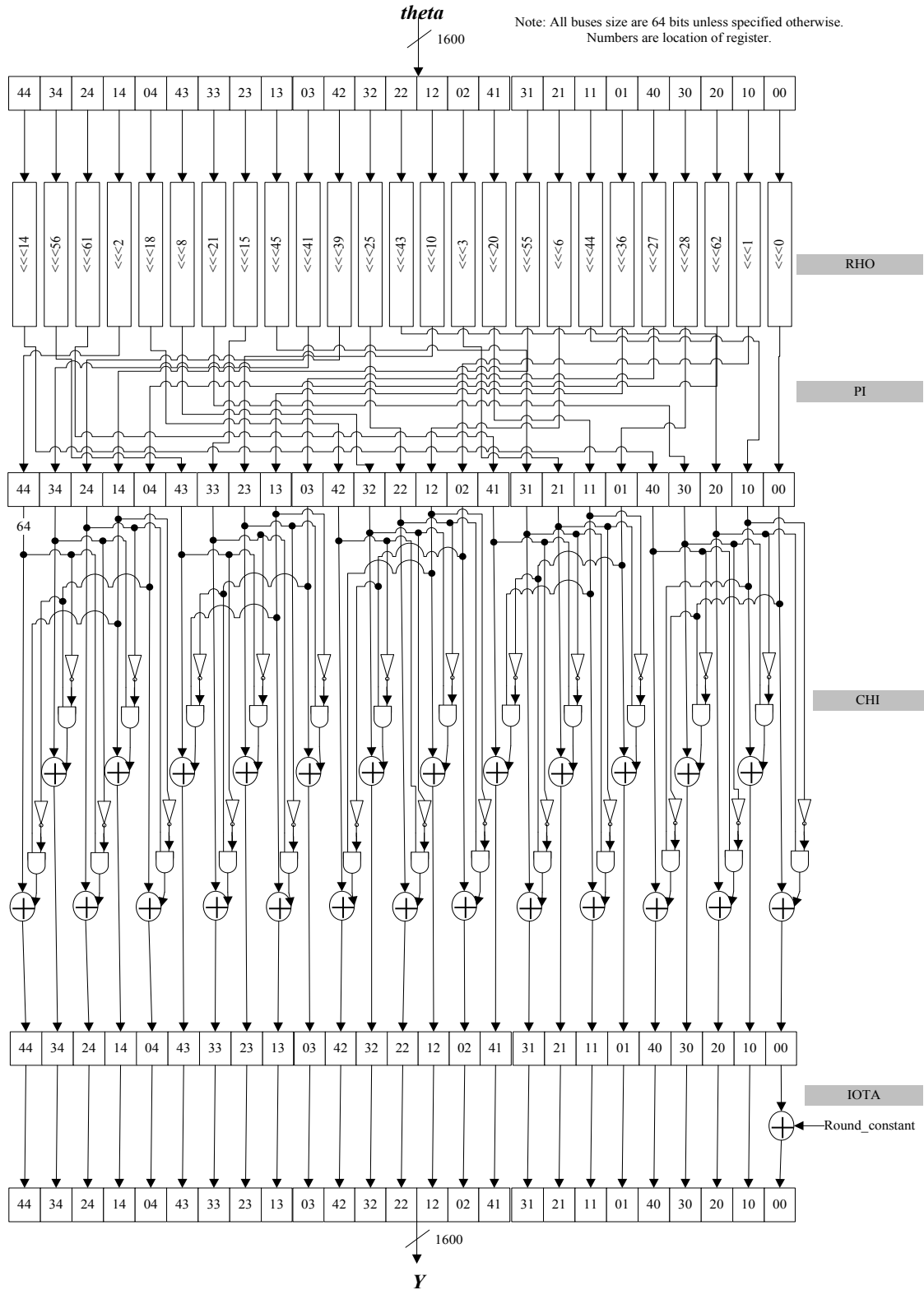


Figure 3.34: Keccak : Round Part 2

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 3.35. 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 3.36. The Galois field multiplication ($\times 2$), used in the MI unit, is also shown in Figure 3.1.

The state is then rotated wordwise using the Tweak operation shown in Figure 3.37. 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 3.38. The Step function consists of SubCrumb, MixWord and AddConstant operations. SubCrumb and MixWord are shown in Figures 3.39 and 3.40, 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 [36]. 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 3.41. Since the finalization process only produces 256 bits at a time, the chain value is hashed with another message block of value zero to produce the second half of a 512-bit hash value.

Luffa-256 : $b=768, h=256, j=3$
 Luffa-512 : $b=1280, h=512, j=5$

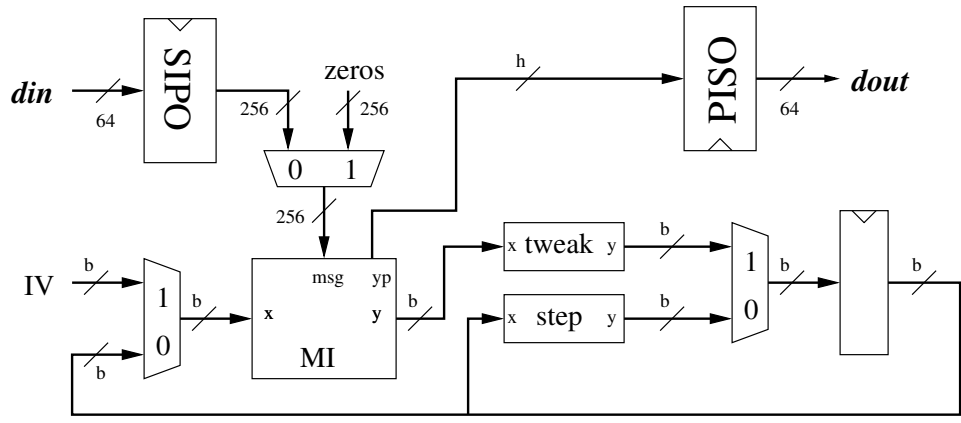


Figure 3.35: Luffa : Datapath

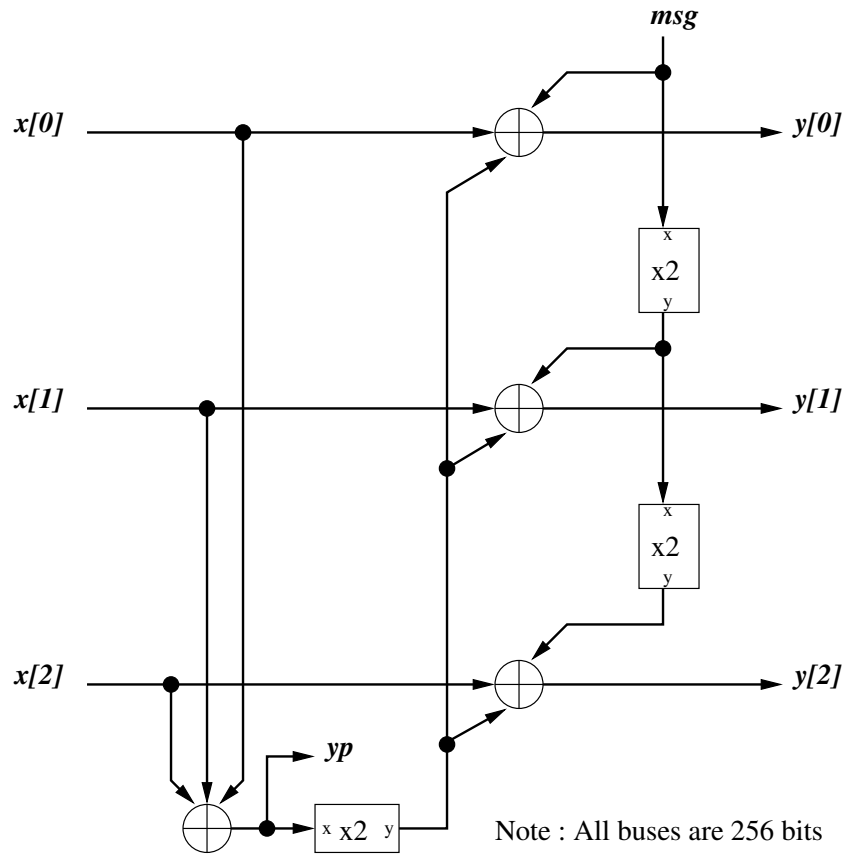


Figure 3.36: Luffa : MI – Message Injection (256 bits)

Luffa-256 : $b=768, j=3$
 Luffa-512 : $b=1280, j=5$

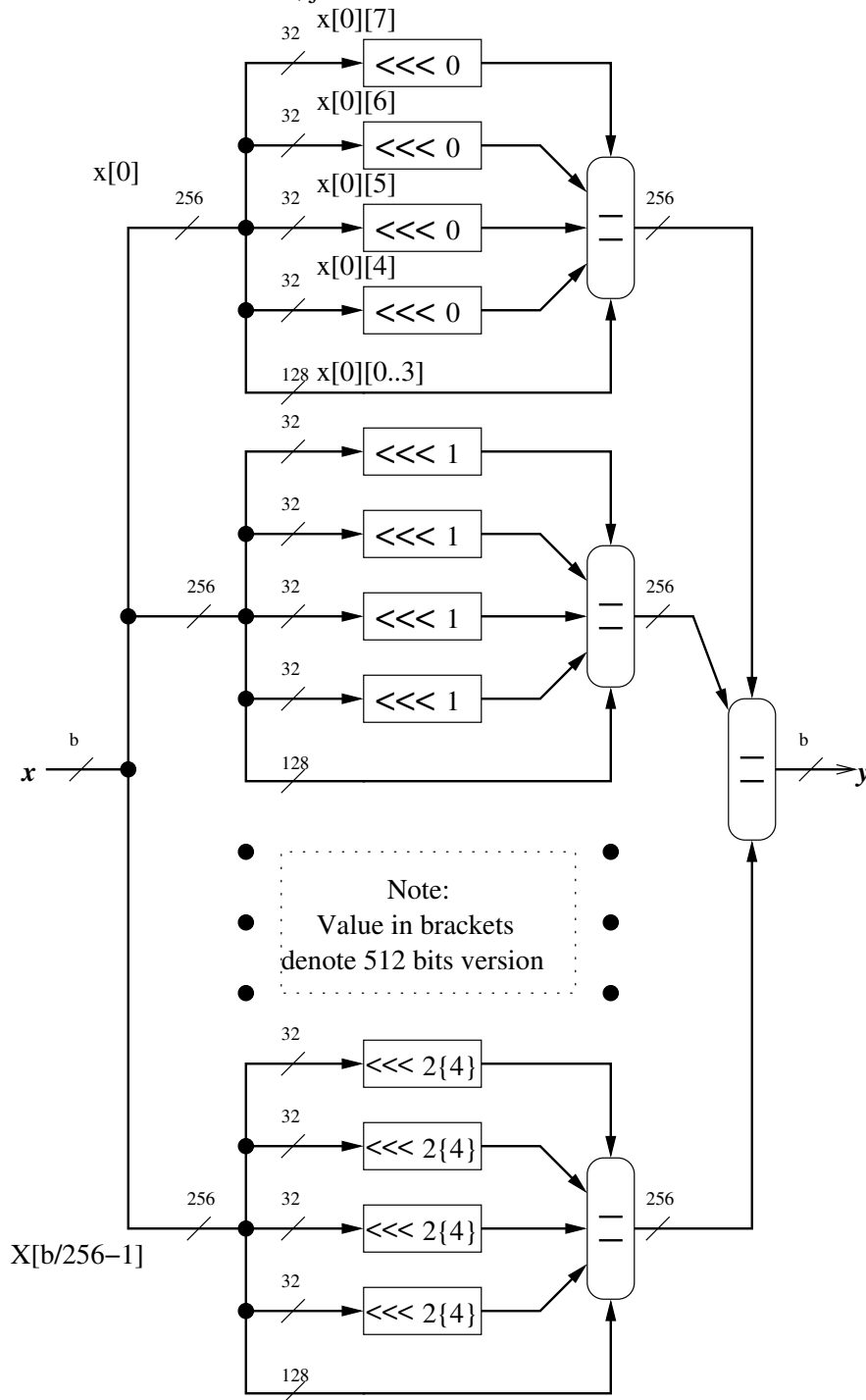


Figure 3.37: Luffa : Tweak

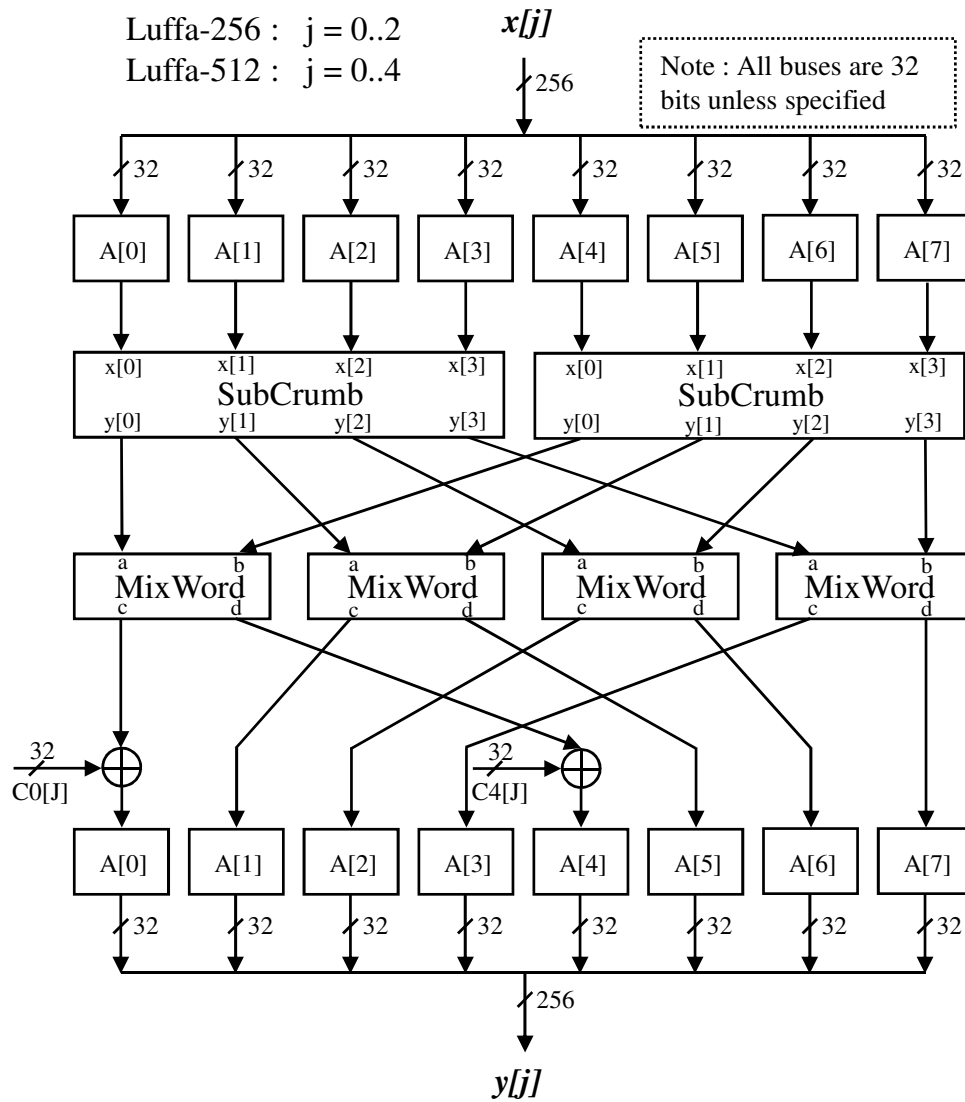


Figure 3.38: Luffa : Step function

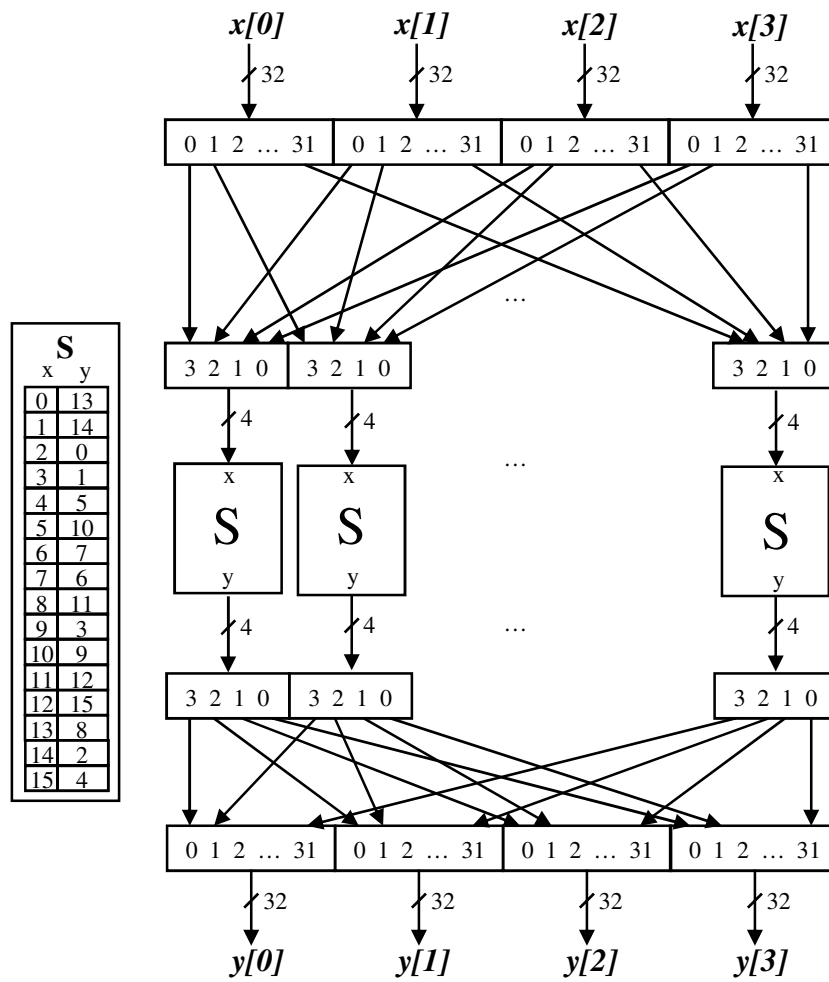


Figure 3.39: Luffa : SubCrumb

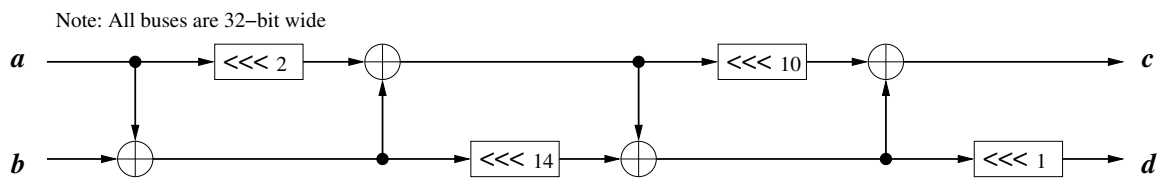


Figure 3.40: Luffa : MixWord

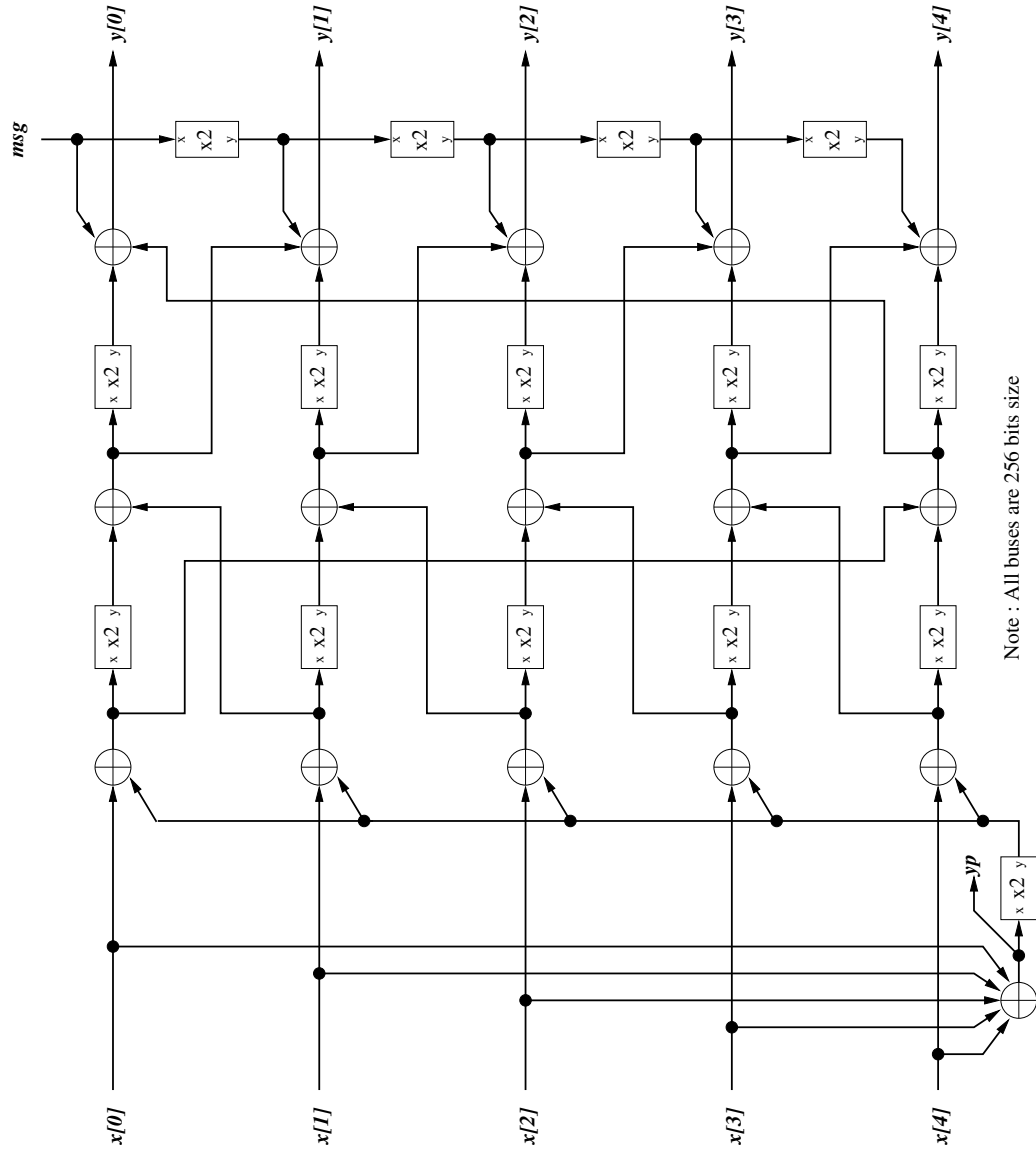


Figure 3.41: Luffa : MI – Message Injection (512 bits)

3.13 SHA-2

3.13.1 Block Diagram Description

Our design of SHA-2 is based on [44]. A diagram of our SHA-2 circuit is shown in Figure 3.43. The detailed definitions of all SHA-2 operations shown in our diagram can be found in [37].

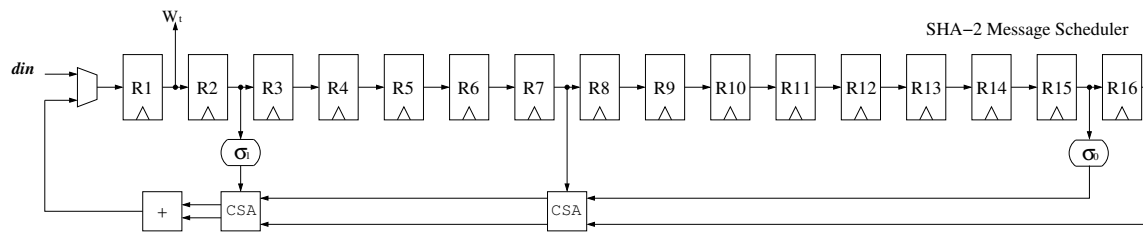


Figure 3.42: SHA2 : Message Scheduler

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 .

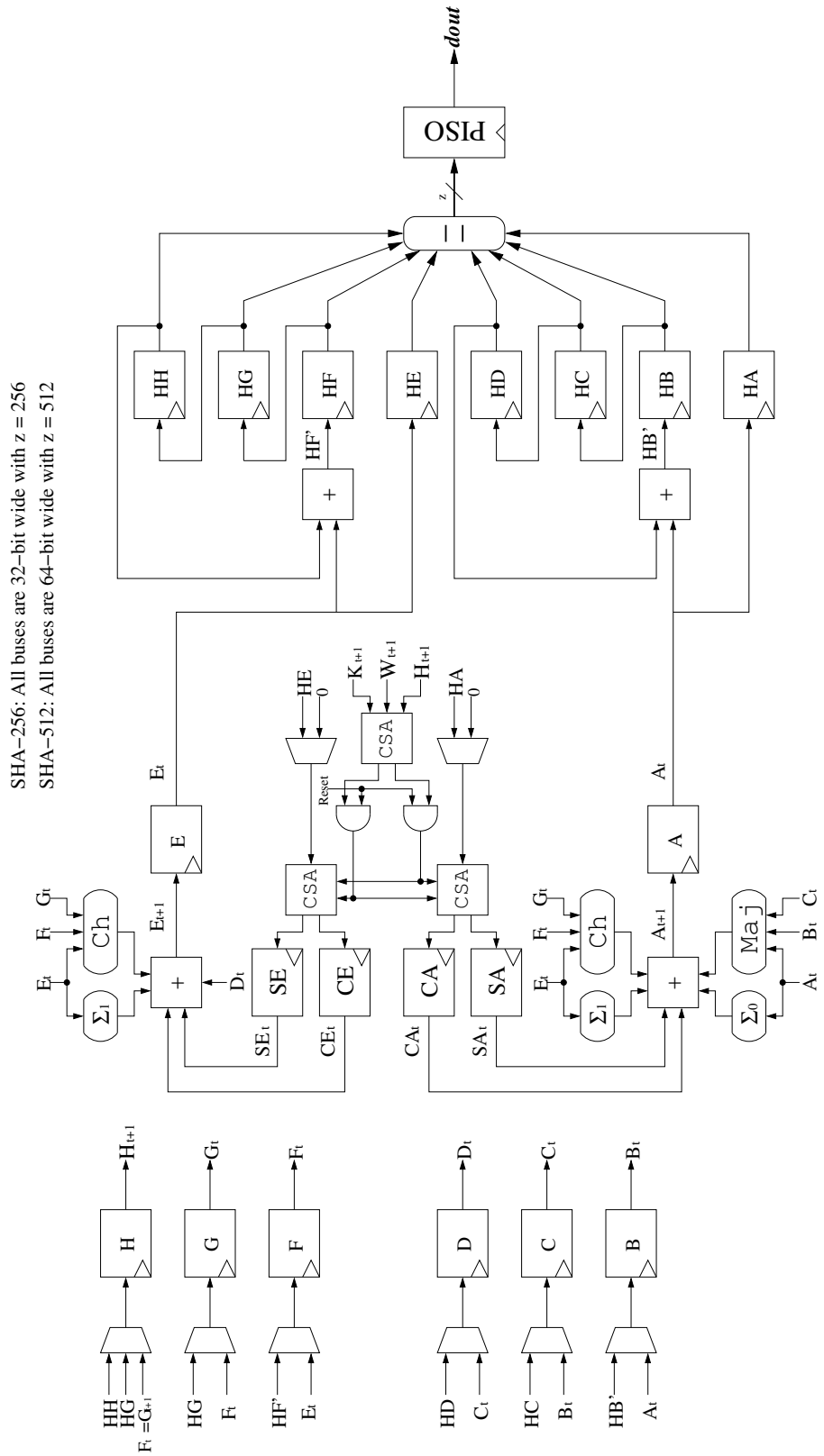


Figure 3.43: SHA2 : Datapath

3.14 Shabal

3.14.1 Block Diagram Description

The block diagram of Shabal is shown in Figure 3.44. Our design of Shabal is a modified version of the design by Detrey et al. [45] Apart from adjusting the design to conform to our interface, there are two main modifications. The first modification is the scheduling for the following operation :

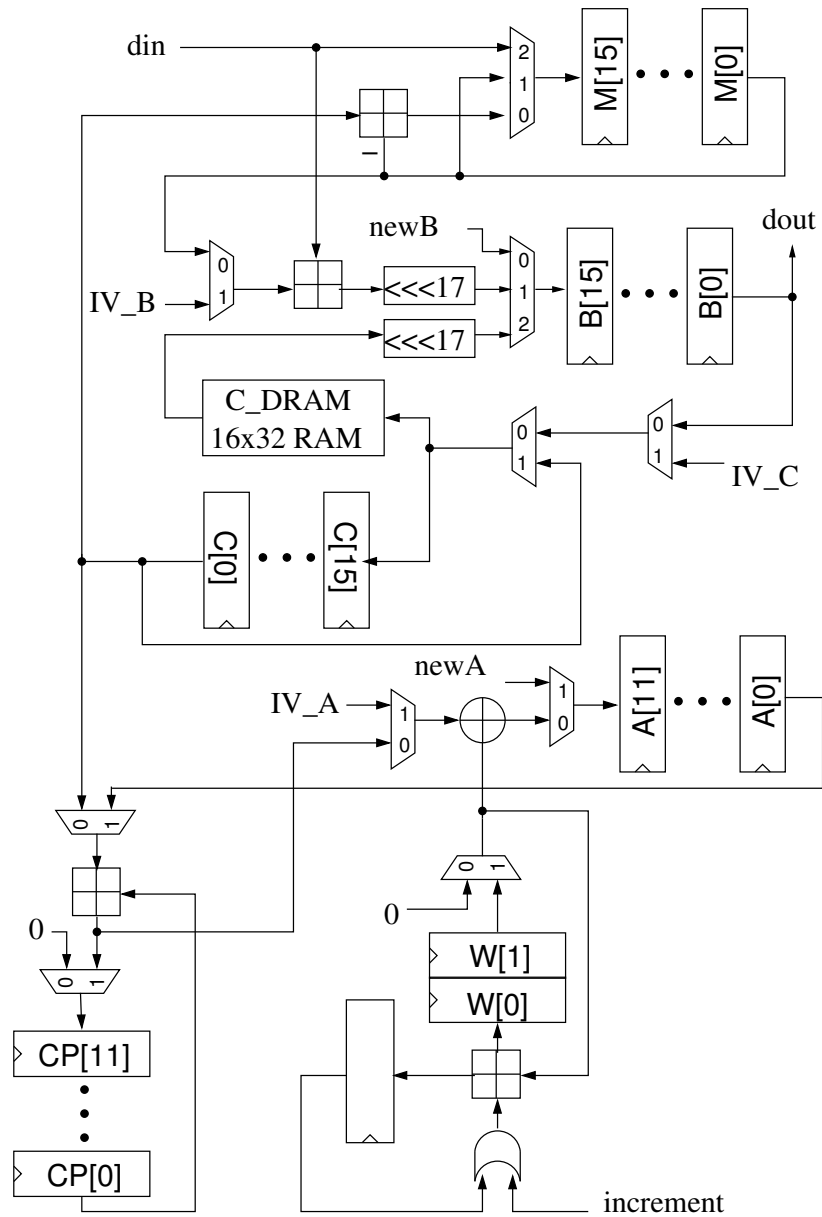
$$A[i \bmod 12] \leftarrow A[i \bmod 12] + C[(i+3) \bmod 16].$$

In the design by Detrey, the aforementioned operation starts in the 16th clock cycle. We begin the computation in the 19th clock cycle. This change should allow better utilization of Xilinx shift registers, as the output from $C[3]$ is no longer required.

The second modification is the block counter, W . A one-stage shift register is used instead of 16-stage shift register. The 16-stage shift register was used to reduce the complexity of the controller in Detrey's design. However, as our interface and protocol are different, this advantage is reduced. Furthermore, a larger number of stages may cause an adverse effect for Altera families, as they are not able to utilize the same optimizations as Xilinx families (namely, shift register mode of Xilinx LUT).

3.14.2 256 vs 512 Variant Differences

There is no difference between the two variants except that different initialization vectors are used for all state registers in the 512-bit variant.



Note: All buses are 32-bit

$$U(x) = (3*x) \bmod 2^{32}$$

$$V(x) = (5*x) \bmod 2^{32}$$

$$\text{newA} = U(A[0] \text{ xor } V(A[11] \lll 15) \text{ xor } C[8] \text{ xor } M[0] \text{ xor } B[13] \text{ xor } (\text{not } B[6] \text{ and } B[9]))$$

$$\text{newB} = \text{not newA} \text{ xor } (B[0] \lll 1)$$

Figure 3.44: Shabal : Datapath

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 3.45. For SHAvite-3-256, a block of message contains 512 bits, and 128 bits are loaded into the key generation unit at a time. For every message, the state register 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.

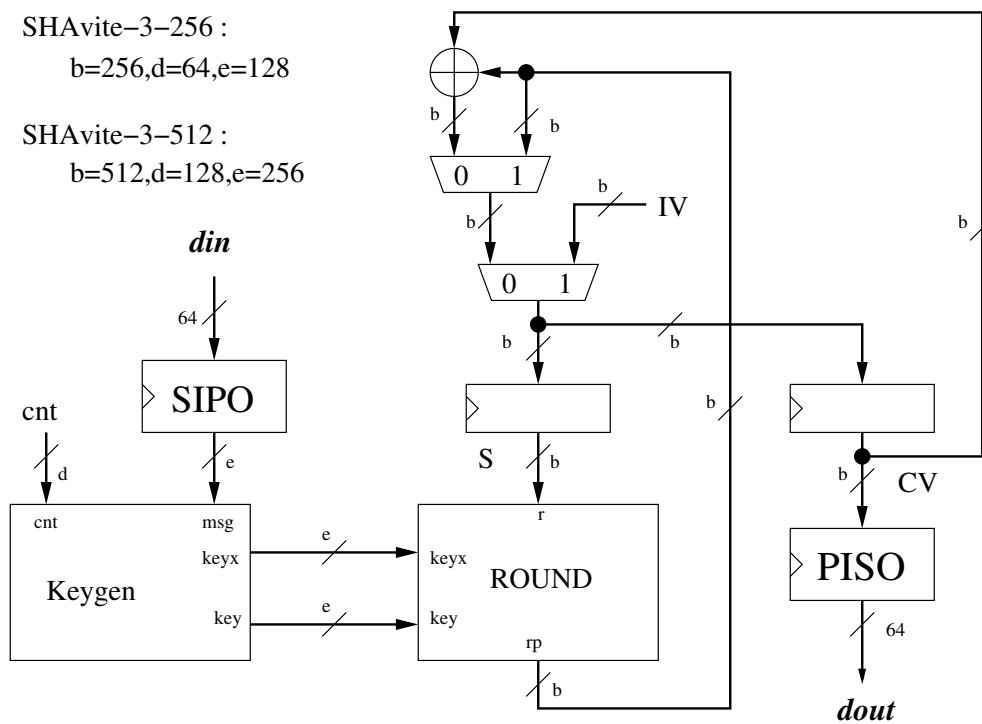


Figure 3.45: SHAvite-3: Datapath

The SHAvite-3 ROUND unit is shown in Figure 3.46. 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.

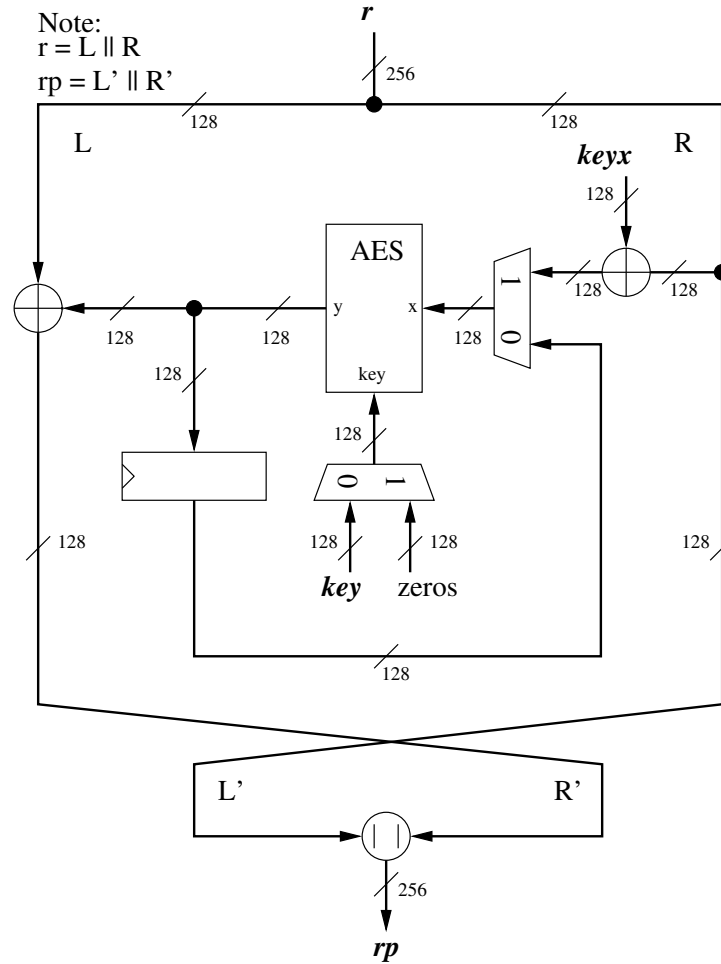


Figure 3.46: SHAvite-3: Round (256 bits)

The key generation unit is shown in Figure 3.47. Key and/or $keyx$ are generated for each main round using this circuit.

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 3.48 illustrates changes in the

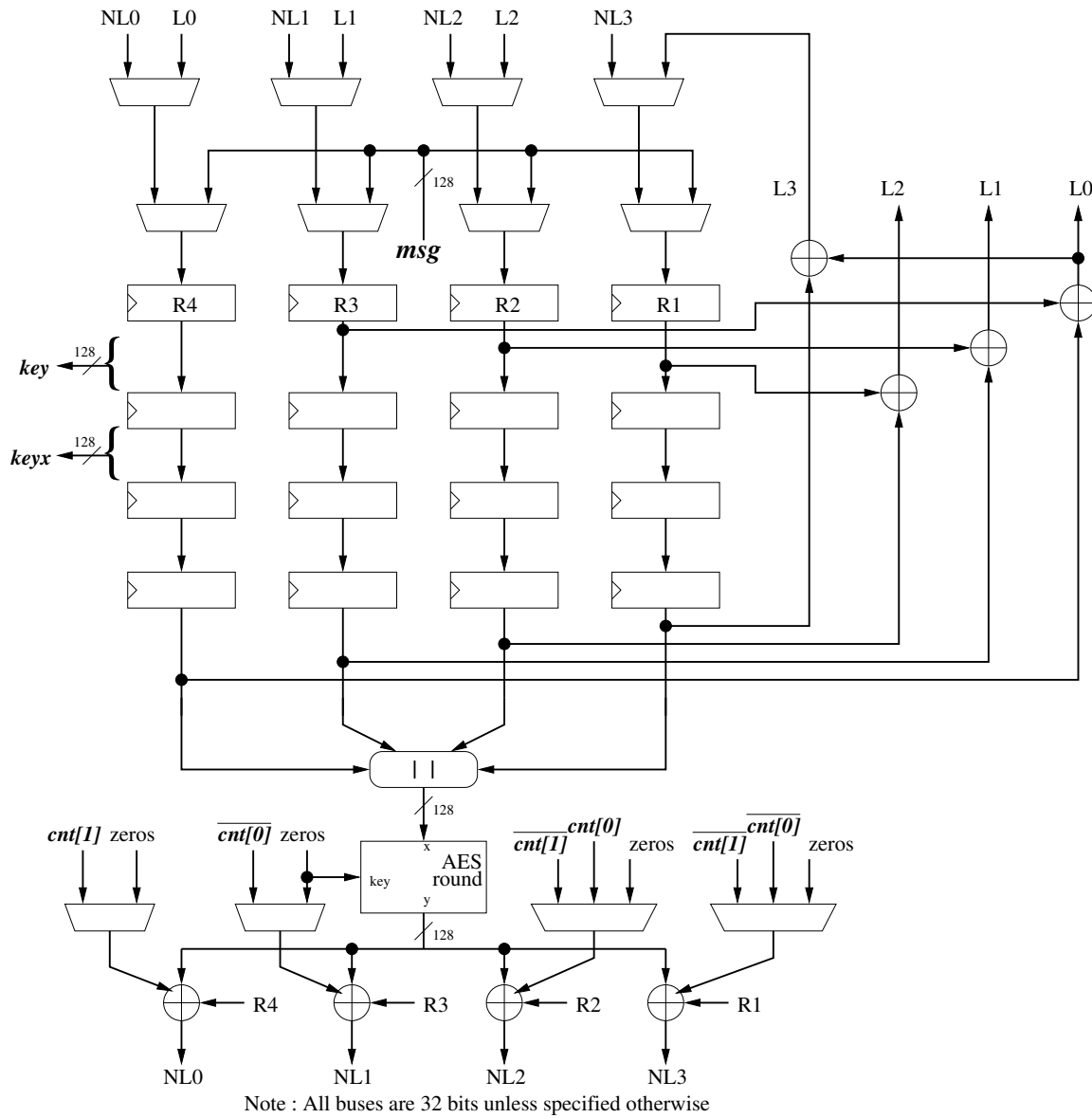


Figure 3.47: SHAvite-3: Keygen (256 bits)

ROUND unit for SHA_{vite}-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 3.49 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.

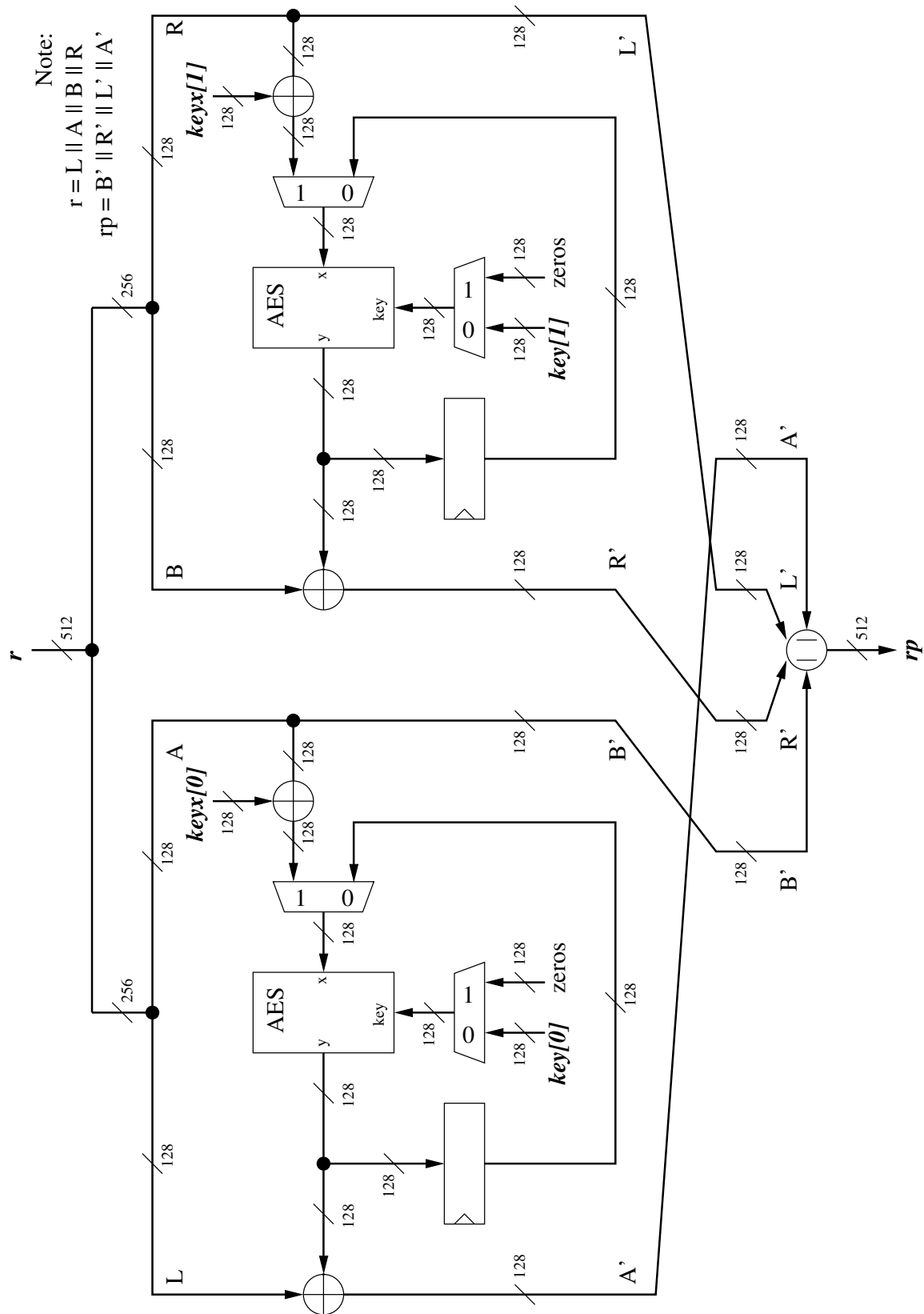


Figure 3.48: SHAvite-3 : Round (512 bits)

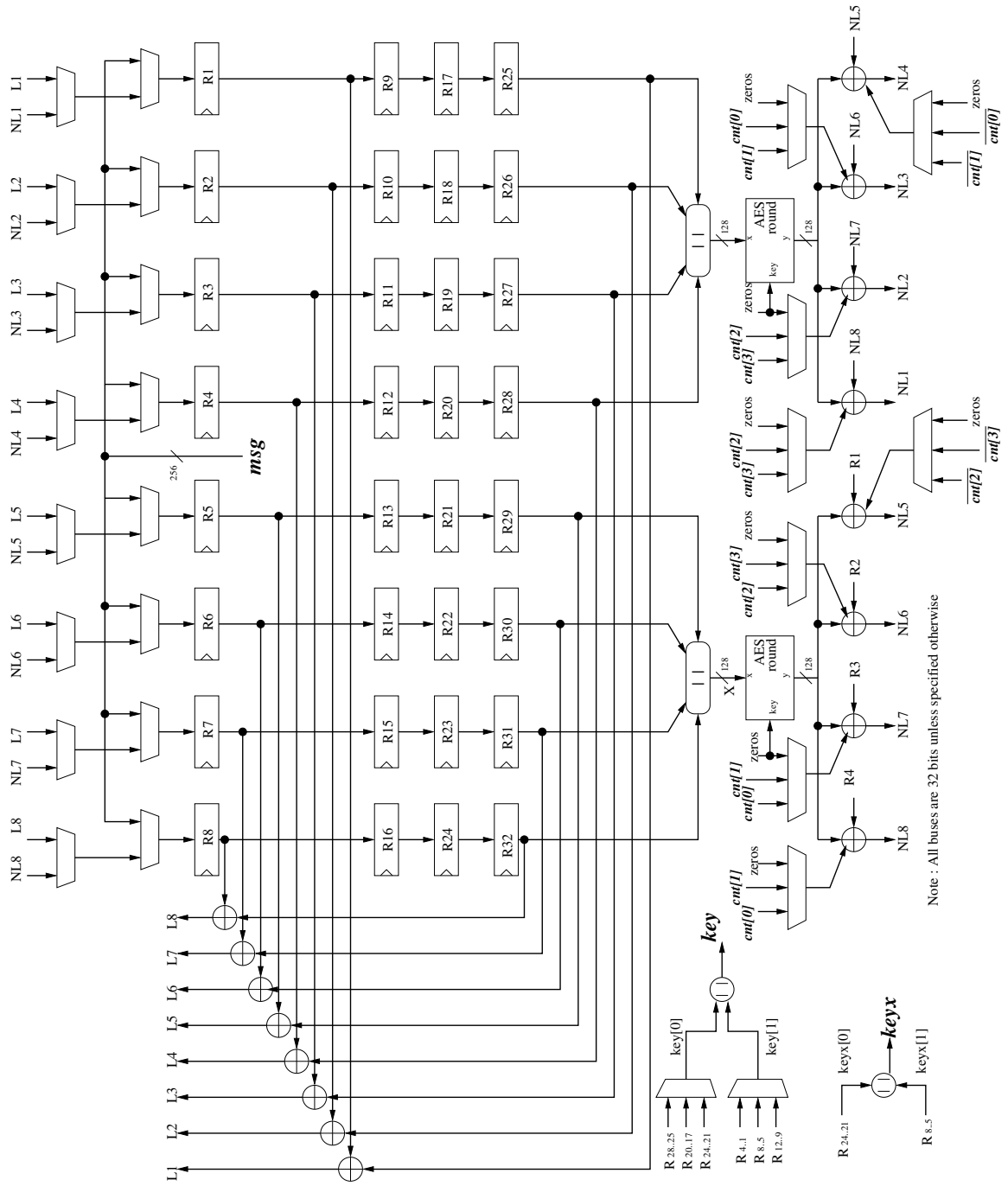


Figure 3.49: SHA-3 : Keygen (512 bits)

3.16 SIMD

3.16.1 Block Diagram Description

The block diagram of SIMD is shown in Figure 3.50. 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.

As opposed to majority of other SHA-3 candidates, the core unit of SIMD requires an output of the Message Expansion unit instead of the SIPO unit. Hence, we can consider the Message Expansion unit as a separate module requiring an additional controller. This special controller is introduced between the FSM1 and FSM2 units allowing them to operate independently. The Message Expansion unit is separated into two parts, the NTT unit and the Concatenated Code and Permutation (CP) unit. The NTT unit is based on a folded 7-stage DFT that uses its first stage, referred to as *DFT stage* onwards, as a basic building block. First, each byte of an input is zero-extended to 9-bits and permuted by σ_0 . The permuted values are inserted into the *DFT stage* together with its respective twiddle factor as a second input. The twiddle factor is chosen based on the DFT stage number, which can be calculated 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 3.52. 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 ensure that there is no bit growth. The Modulo 257 unit is shown in Figure 3.53.

Our DFT unit is based on a 7-stage FFT (Fast Fourier Transform) for the 256-bit variant of SIMD, and on an 8-stage FFT for the 512-bit variant of SIMD. Before the input data enters the first stage of FFT, its 9-bit words are rearranged using the permutation σ_0 . After each pass through the DFT stage, another permutations σ_i is used to rearrange inputs to make them ready for the next stage of FFT. All permutations can be described using the following pseudocode:

$$\sigma_0(X) = M(X)$$

$$\sigma_1(X) = P_2(X)$$

$$\sigma_k(X) = P_{k+1}(P_k^{-1}(X)) \text{ for } k=2..stage_no-1$$

$$\sigma_{last}(X) = P_{stage_no-1}^{-1}(X)$$

$Y = M(X)$ can be defined as follows:

```
for i=0 to pt-1 loop
  y(i) = x(bit_mirror(i))
end loop
```

where `bit_mirror(i)` is a function that converts `i` to binary, inverts the order of bits, and converts the result back to an integer.

$Y = P_m(X)$ is defined as follows:

```
pt = 2^stage_no
group_size = 2^m
groups = pt/group_size
for i in 0 to groups-1 loop
  for j in 0 to group_size/2-1 loop
    y(group_size*i+2*j) <= x(group_size*i+j);
    y(group_size*i+2*j+1) <= x(group_size*i+j+group_size/2);
  end loop;
end loop;
```

$Y = P_m^{-1}(X)$ is defined as follows:

```
pt = 2^stage_no
group_size = 2^m
groups = pt/group_size
for i in 0 to groups-1 loop
  for j in 0 to group_size/2-1 loop
    y(group_size*i+j) <= x(group_size*i+2*j) ;
    y(group_size*i+j+group_size/2) <= x(group_size*i+2*j+1);
  end loop;
end loop;
```

where, `stage_no` is 7 and 8, for 256 and 512-bit variant, respectively.

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:

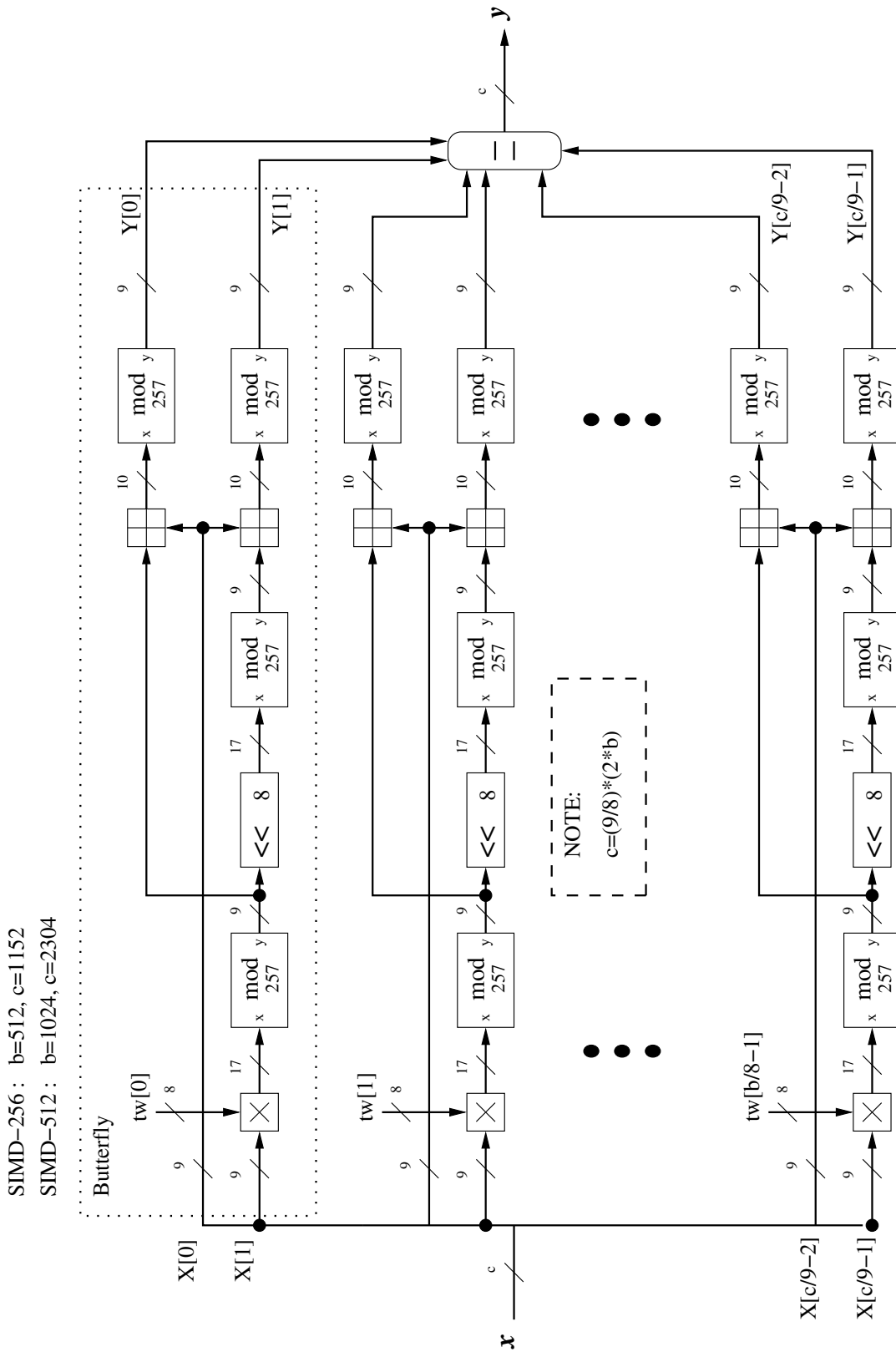


Figure 3.52: SIMD : DFT stage

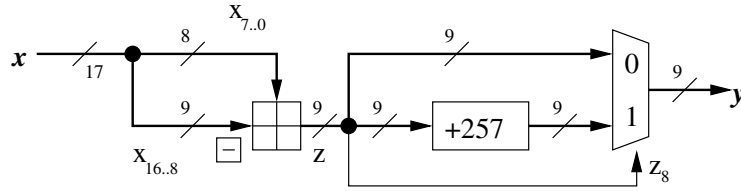


Figure 3.53: SIMD : Modulo 257

```

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 [40]. A diagram of Concat Permute (CP) is shown in Figure 3.54. 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:

$$Z_j^i = \begin{cases} x[8i + 2j] \parallel x[8i + 2j + 1] & \text{when } 0 \leq i \leq 15 \\ x[8i + 2j - 128] \parallel x[8i + 2j - 64] & \text{when } 16 \leq i \leq 23 \\ x[8i + 2j - 191] \parallel x[8i + 2j - 127] & \text{when } 24 \leq i \leq 31 \end{cases} \quad \text{with } 0 \leq j \leq 3$$

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

A multiplexer selects appropriate data depending on the cycle number. A selected value is viewed as an array of 4 x 4 with 18 bits at each location. Each 18-bit value is split in half and entered into Lift module shown in Figure 3.55. An output from the Lift

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

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.

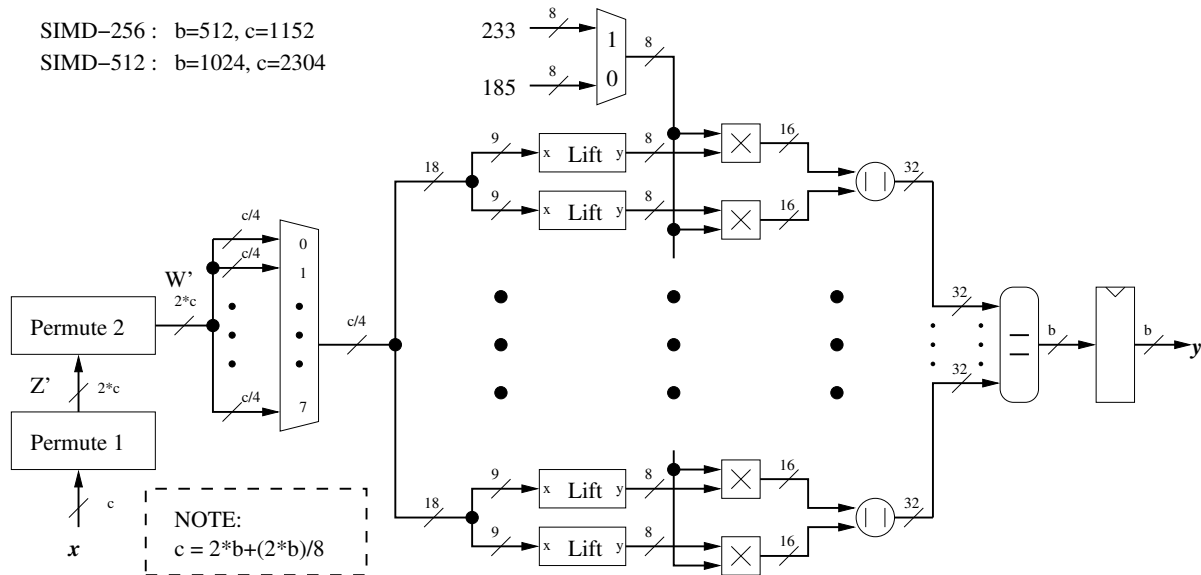


Figure 3.54: SIMD : Concatenate and Permute (CP)

The core operation of our SIMD's design is the Half Round module. This module is equivalent to four steps of the SIMD round. A block diagram of the Half Round operation is shown in Figure 3.56. Half Round is based of 16 *QS* units with *quarterstep* as their core. *quarterstep* is shown in Figure 3.57. 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:

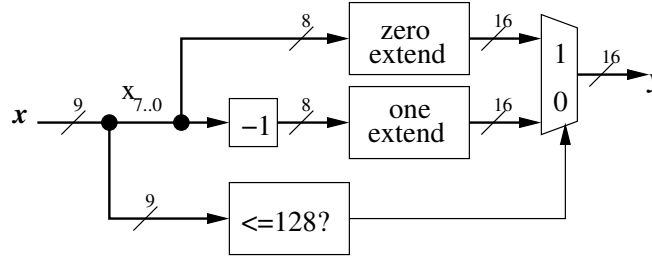


Figure 3.55: SIMD : Lift

$\phi^{(i)}$	$r^{(i)}$	$s^{(i)}$					
<i>IF</i>	π_0	π_1					
<i>IF</i>	π_1	π_2	<i>Round</i>	π_0	π_1	π_2	π_3
<i>IF</i>	π_2	π_3	0	3	23	17	27
<i>IF</i>	π_3	π_0	1	28	19	22	7
<i>MAJ</i>	π_0	π_1	2	29	9	15	5
<i>MAJ</i>	π_1	π_2	3	4	13	10	25
<i>MAJ</i>	π_2	π_3					
<i>MAJ</i>	π_3	π_0					

Due to the each step's non-uniform structure of permutation, the inputs to each mux inside of QS in Figure 3.56 are defined as follows:

```

for x in 0 to 3 loop -- row
  for y in 0 to feistel_ladder_no-1 loop --column
    for m in 0 to permute_size-1 loop --mux input
      mux(x,y)(m) <= aout( x, p^((4*m+x) mod permute_size)(y) );
    end loop;
  end loop;
end loop;

```

where, *feistel_ladder_no* is a number of Feistel Ladders, which is 4 for SIMD-256 and 8 for SIMD-512; *permute_size* is a size of permutation p (defined below), which is equal to *feistel_ladder_no*-1 (3 for SIMD-256 and 7 for SIMD-512); and $\text{mux}(x,y)(m)$ refers to the mux input m of the quarter step (x,y) .

Finally, the permutation p used in the above formulas is given below:

$$\begin{aligned}
 p^{(0)}(j) &= j \oplus 1 \\
 p^{(1)}(j) &= j \oplus 2 \\
 p^{(2)}(j) &= j \oplus 3.
 \end{aligned}$$

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:

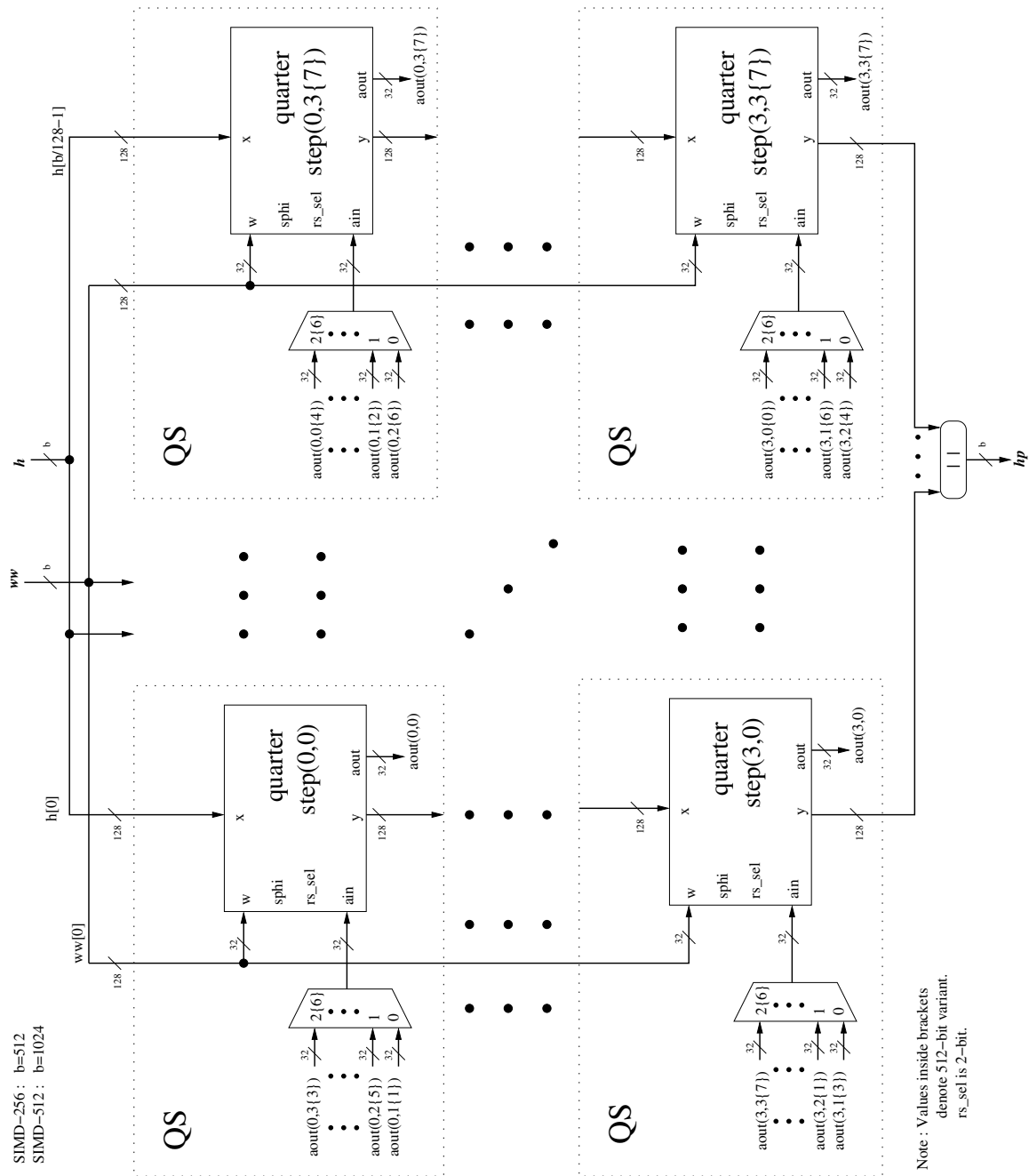


Figure 3.56: SIMD : Half Round

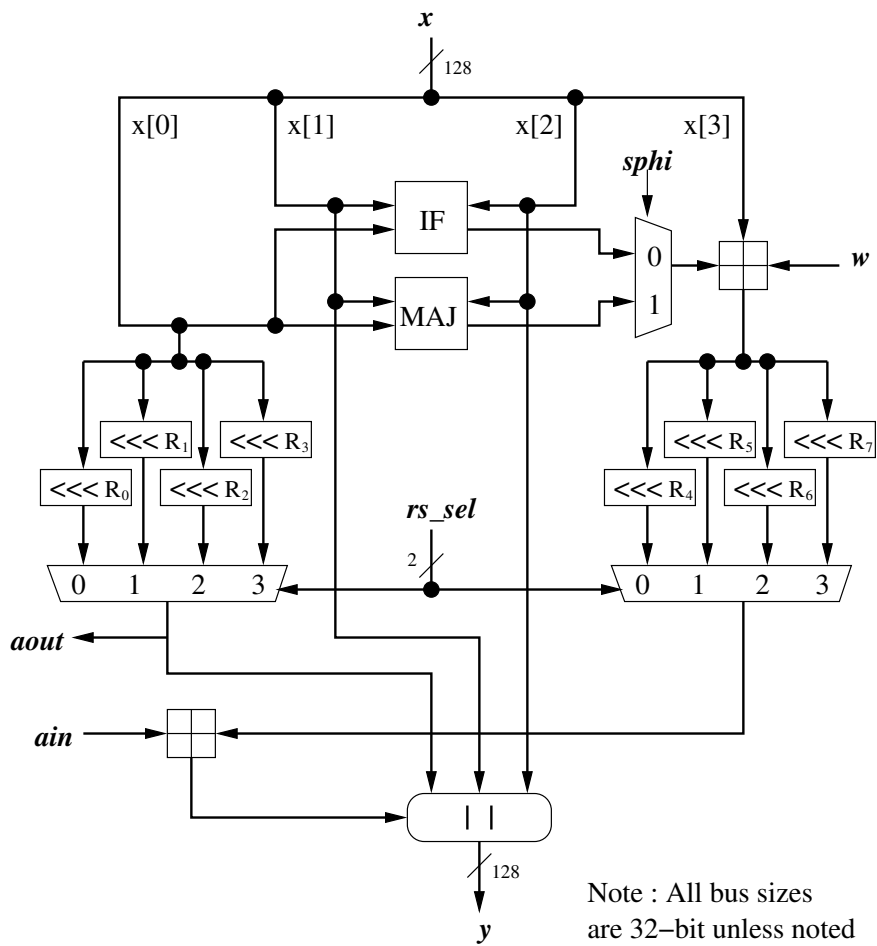


Figure 3.57: SIMD : Quarter Step

$$Z_j^i = \begin{cases} x[8i + 2j] \parallel x[8i + 2j + 1] & \text{when } 0 \leq i \leq 15 \\ x[8i + 2j - 256] \parallel x[8i + 2j - 128] & \text{when } 16 \leq i \leq 23 \\ x[8i + 2j - 383] \parallel x[8i + 2j - 255] & \text{when } 24 \leq i \leq 31 \end{cases}$$

with $0 \leq j \leq 7$

Additionally, the number of Feistel Ladders is increased from 4 to 8, and thus, the number of the QS units in Half Round increases from 16 to 32 (see Fig. 3.56). The inputs to the Ain muxes are defined by the same pseudocode as before, but the size and the definition of permutation p changes. The size of the permutation is now 7, and its definition is given below.

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

3.17 Skein

3.17.1 Block Diagram Description

The datapath of Skein is shown in Figure 3.58. 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.

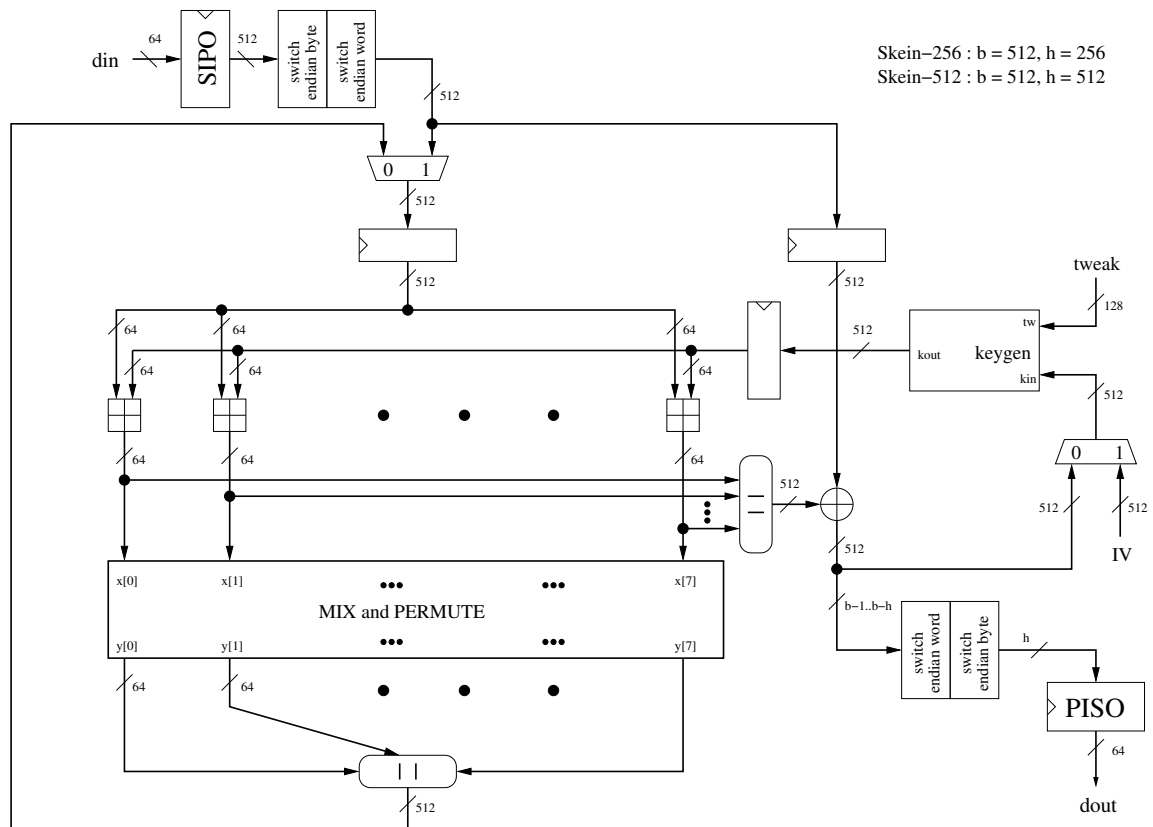


Figure 3.58: Skein : Datapath

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 [41]. In Figure 3.59, a key generation unit for our design is shown. s is the subkey counter. It gets reset for every new message block.

In Figure 3.60, 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 3.61. 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			
j	0	1	2	3
0	46	36	19	37
1	33	27	14	42
2	17	49	36	39
3	44	9	54	56
4	39	30	34	24
5	13	50	10	17
6	25	29	39	43
7	8	35	56	22

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.

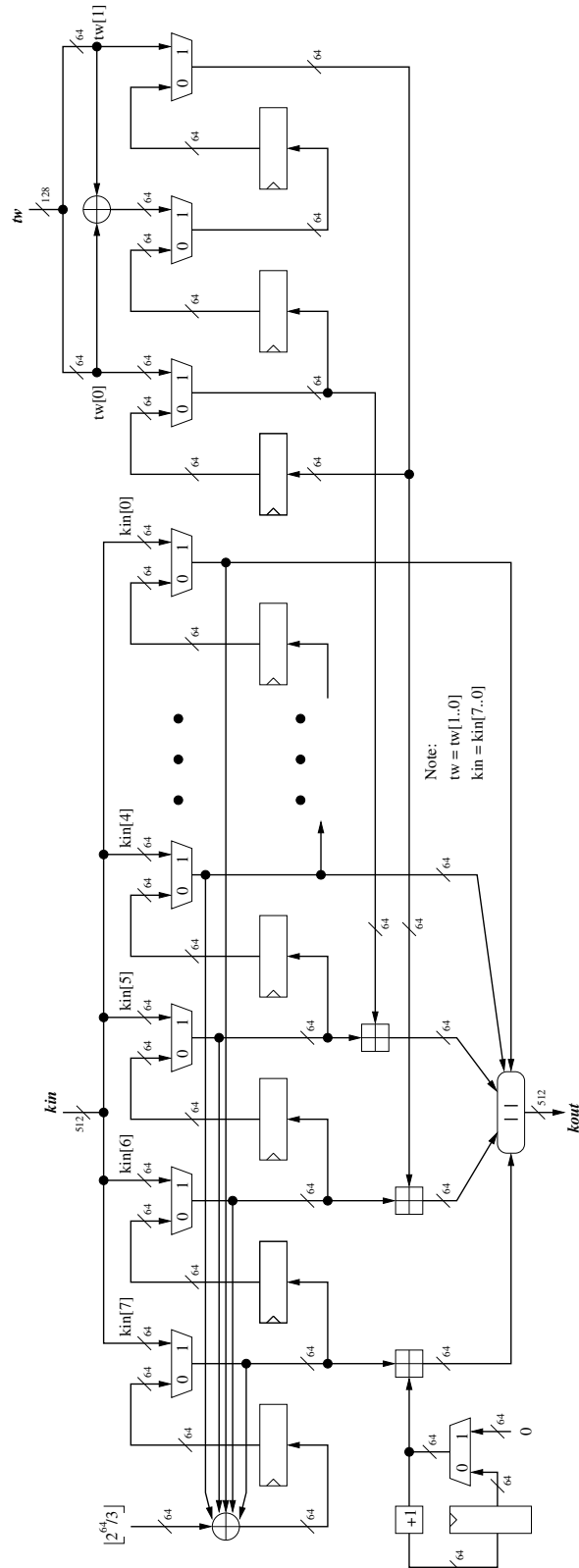


Figure 3.59: Skein : Key Generation

Note: All buses are 64 bits, except of r, where r is a single big line

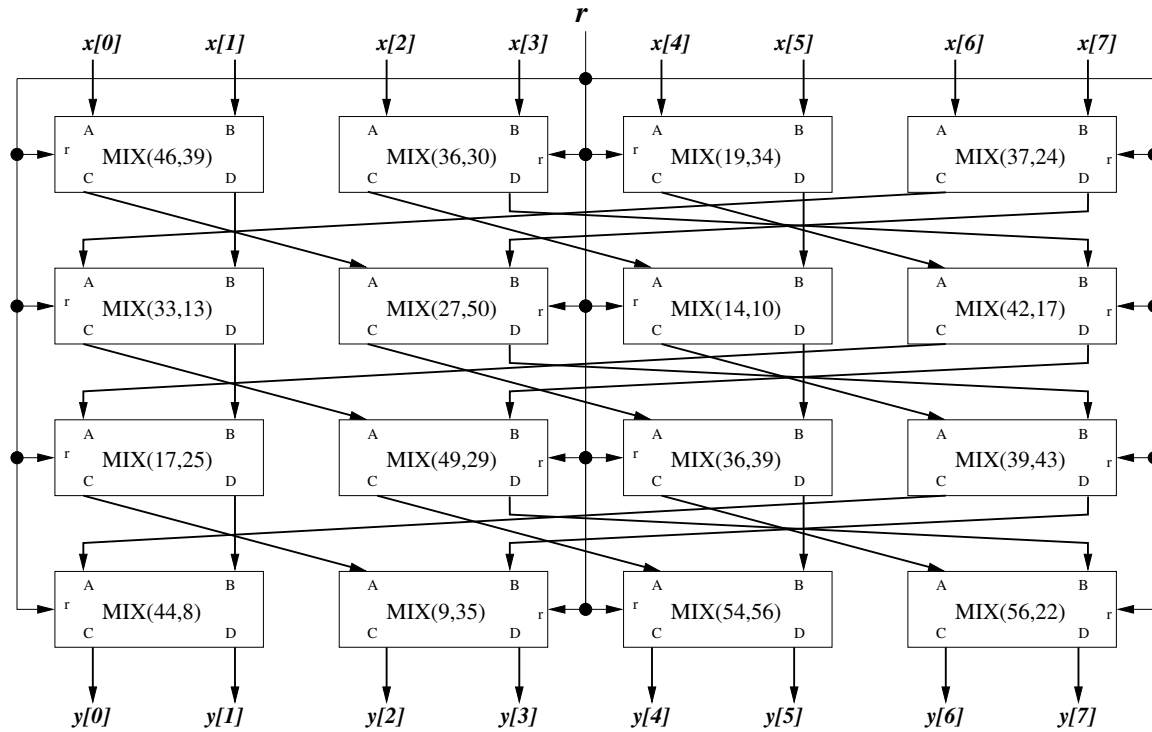


Figure 3.60: Skein : Round

Note: All buses are 64 bits, except of r, where r is a single bit line

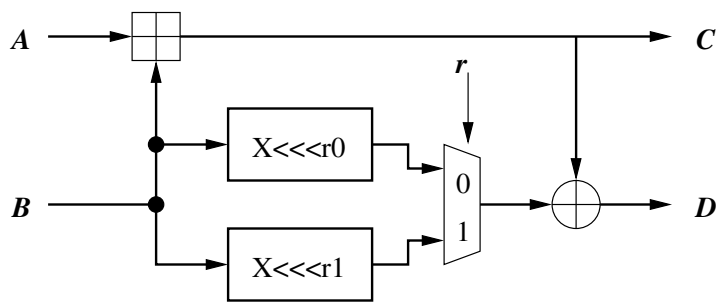


Figure 3.61: Skein : Mix

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 and SIMD-512, 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). For SIMD-512, this ratio was chosen to be 2, which was sufficient to assure that the time of loading the next message block was smaller than the time of processing of the current block.

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 Area Ratio	Predicted Thr Ratio
	state size	Block size	Round no	Word size	State size	Block size	Round no	Word size		
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

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 μ s. Values different between 256-bit and 512-bit variants are shown in bold.

Function	256-bit variants			512-bit variants		
	I/O Bus width	Hash Time [cycles]	Throughput [Mbit/s]	I/O Bus width	Hash Time [cycles]	Throughput [Mbit/s]
BLAKE	64	$2+8+21\cdot N+4$	$512/(21\cdot T)$	64	$2+\mathbf{16}+29\cdot N+8$	$\mathbf{1024}/(\mathbf{29}\cdot T)$
BMW	64	$2+8/8+N+1+4/8$	$512/T$	64	$2+\mathbf{16}/\mathbf{16}+N+1+\mathbf{8}/\mathbf{16}$	$\mathbf{1024}/T$
CubeHash	64	$2+4+16\cdot N+160+4$	$256/(16\cdot T)$	64	$2+4+16\cdot N+160+8$	$256/(16\cdot T)$
ECHO	64	$3+24+25\cdot N+1+4$	$1536/(27\cdot T)$	64	$3+\mathbf{16}+\mathbf{31}\cdot N+1+8$	$\mathbf{1024}/(\mathbf{31}\cdot T)$
Fugue	32	$2+2\cdot N+37+8$	$32/T$	32	$2+4\cdot N+\mathbf{85}+16$	$32/(4\cdot T)$
Groestl	64	$3+8+21\cdot N+4$	$512/(21\cdot T)$	64	$3+\mathbf{16}+29\cdot N+8$	$\mathbf{1024}/(\mathbf{29}\cdot T)$
Hamsi	32	$3+1+3\cdot(N-1)+6+8$	$32/(3\cdot T)$	64	$3+1+\mathbf{6}\cdot(N-1)+12+8$	$\mathbf{64}/(\mathbf{6}\cdot T)$
JH	64	$3+8+36\cdot N+4$	$512/(36\cdot T)$	64	$3+8+36\cdot N+8$	$512/(36\cdot T)$
Keccak	64	$3+17+24\cdot N+4$	$1088/(24\cdot T)$	64	$3+\mathbf{9}+24\cdot N+8$	$\mathbf{576}/(24\cdot T)$
Luffa	64	$3+4+9\cdot N+9+1+4$	$256/(9\cdot T)$	64	$3+4+9\cdot N+\mathbf{2}\cdot\mathbf{9}+1+8$	$256/(9\cdot T)$
Shabal	64	$3+8+1+25\cdot N+3\cdot 25+4$	$512/(25\cdot T)$	64	$3+8+1+25\cdot N+3\cdot 49+8$	$512/(25\cdot T)$
Shavite-3	64	$3+8+37\cdot N+4$	$512/(37\cdot T)$	64	$3+16+\mathbf{57}\cdot N+8$	$\mathbf{1024}/(\mathbf{57}\cdot T)$
SIMD	64	$3+8+8+9\cdot N+4$	$512/(9\cdot T)$	64	$3+\mathbf{16}/\mathbf{2}+\mathbf{9}+9\cdot N+\mathbf{8}/\mathbf{2}$	$\mathbf{1024}/(9\cdot T)$
Skein	64	$2+8+19\cdot N+4$	$512/(19\cdot T)$	64	$2+8+19\cdot N+8$	$512/(19\cdot T)$
SHA-256	32	$2+1+65\cdot N+8$	$512/(65\cdot T)$	64	$2+1+\mathbf{81}\cdot N+8$	$\mathbf{1024}/(\mathbf{81}\cdot T)$

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.

$$Predicted_Area_Ratio_{512/256} = \frac{State_size_{512}}{State_size_{256}}. \quad (4.1)$$

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.

$$\text{Predicted_Throughput_Ratio}_{512/256} = \frac{\text{Thr}_{512}}{\text{Thr}_{256}} = \frac{\frac{\text{Block_size}_{512}}{\text{Round_no}_{512}}}{\frac{\text{Block_size}_{256}}{\text{Round_no}_{256}}}. \quad (4.2)$$

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.

4.3 Results

In Tables 4.3 and 4.4, 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. The corresponding results for the remaining 8 families are provided in Appendix A.

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 ten selected FPGA families. The abbreviations used in this paper to denote 10 selected FPGA families from Xilinx and Altera are summarized in Table 4.5. These abbreviations will be used consistently in all tables included in this chapter. In terms of the design, an architecture

Table 4.3: 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	99.68	128.90	0.77	3,520	3,143	1.12	3,064	1,523	2.01	1.15	2.06	0.56
BMW	N/A	8.30	N/A	N/A	4,250	N/A	N/A	4,855	N/A	N/A	0.88	N/A
CubeHash	269.69	274.05	0.98	4,315	4,385	0.98	734	684	1.07	5.88	6.41	0.92
ECHO	235.50	184.30	1.28	7,779	11,323	0.69	5,044	4,982	1.01	1.54	2.27	0.68
Fugue	221.58	218.44	1.01	1,773	3,495	0.51	979	708	1.38	1.81	4.94	0.37
Groestl	292.10	323.40	0.90	10,314	7,885	1.31	3,138	1,597	1.96	3.29	4.94	0.67
Hamsi	182.10	285.88	0.64	1,942	3,049	0.64	1,900	720	2.64	1.02	4.24	0.24
JH	394.48	380.81	1.04	5,610	5,416	1.04	1,104	1,018	1.08	5.08	5.32	0.96
Keccak	285.23	282.73	1.01	6,845	12,817	0.53	1,257	1,272	0.99	5.45	10.08	0.54
Luffa	240.33	340.72	0.71	7,691	9,692	0.79	1,960	949	2.07	3.92	10.21	0.38
Shabal	214.92	214.92	1.00	1,719	1,719	1.00	283	283	1.00	6.08	6.08	1.00
SHAvite-3	213.80	235.10	0.91	3,841	3,253	1.18	2,090	1,076	1.94	1.84	3.02	0.61
SIMD	43.40	54.90	0.79	4,938	3,123	1.58	19,639	8,922	2.20	0.25	0.35	0.72
Skein	119.09	117.95	1.01	3,209	3,178	1.01	1,716	1,621	1.06	1.87	1.96	0.95
SHA-2	183.70	190.90	0.96	2,322	1,504	1.54	838	418	2.00	2.77	3.60	0.77

Table 4.4: 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	89.53	118.98	0.75	3,161	2,901	1.09	7,086	3,635	1.95	0.45	0.80	0.56
BMW	N/A	12.38	N/A	N/A	6,339	N/A	N/A	12,619	N/A	N/A	0.50	N/A
CubeHash	204.21	232.88	0.88	3,267	3,726	0.88	1,930	1,922	1.00	1.69	1.94	0.87
ECHO	247.40	233.32	1.06	8,172	14,335	0.57	21,187	20,723	1.02	0.39	0.69	0.56
Fugue	199.80	207.43	0.96	1,598	3,319	0.48	2,783	2,397	1.16	0.57	1.38	0.41
Groestl	202.27	220.65	0.92	7,142	5,380	1.33	12,355	6,350	1.95	0.58	0.85	0.68
Hamsi	187.55	280.98	0.67	2,001	2,997	0.67	6,401	2,308	2.77	0.31	1.30	0.24
JH	390.63	387.75	1.01	5,556	5,515	1.01	3,709	3,525	1.05	1.50	1.56	0.96
Keccak	304.60	273.37	1.11	7,310	12,393	0.59	3,979	4,213	0.94	1.84	2.94	0.62
Luffa	268.10	301.30	0.89	8,579	8,570	1.00	6,891	3,032	2.27	1.24	2.83	0.44
Shabal	109.66	109.66	1.00	877	877	1.00	1,744	1,744	1.00	0.50	0.50	1.00
SHAvite-3	226.60	245.52	0.92	4,071	3,397	1.20	5,619	3,042	1.85	0.72	1.12	0.65
SIMD	49.82	54.89	0.91	5,668	3,123	1.82	53,623	25,728	2.08	0.11	0.12	0.87
Skein	90.34	92.87	0.97	2,434	2,503	0.97	4,794	4,645	1.03	0.51	0.54	0.94
SHA-2	162.42	205.09	0.79	2,053	1,615	1.27	2,164	1,049	2.06	0.95	1.54	0.62

Table 4.5: Abbreviations used to denote Xilinx and Altera families in the subsequent tables.

Xilinx		Altera	
Family	Notation	Family	Notation
Spartan 3	S-3	Cyclone II	C-II
Virtex 4	V-4	Cyclone III	C-III
Virtex 5	V-5	Cyclone IV	C-IV
Virtex 6	V-6	Stratix II	S-II
		Stratix III	S-III
		Stratix IV	S-IV

Table 4.6: Results for the reference implementation of SHA-256 (architecture with rescheduling)

FPGA Family	Xilinx Families				Altera Families					
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV
Max. Clk Freq. [MHz]	79.96	157.88	190.90	270.56	108.68	118.75	126.50	150.31	205.09	217.63
Throughput [Mbit/s]	629.84	1243.61	1503.70	2131.18	856.06	935.38	996.43	1183.98	1615.48	1714.25
Area	857	861	418	335	1686	1685	1681	981	1049	1052
Throughput/Area Ratio	0.73	1.44	3.60	6.36	0.51	0.56	0.59	1.21	1.54	1.63

Table 4.7: Results for the reference implementation of SHA-512 (architecture with rescheduling)

FPGA Family	Xilinx Families				Altera Families					
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV
Max. Clk Freq. [MHz]	67.68	128.25	183.70	238.83	86.41	94.86	99.75	123.03	162.42	188.71
Throughput [Mbit/s]	855.61	1621.33	2322.33	3019.28	1092.39	1199.22	1261.04	1555.34	2053.31	2385.67
Area	1973	1952	838	538	3307	3315	3296	2033	2164	2165
Throughput/Area Ratio	0.43	0.83	2.77	5.61	0.33	0.36	0.38	0.77	0.95	1.10

by Chaves et al. [44], [46] 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.

Tables 4.8 and 4.9 summarize the clock frequencies of the implemented algorithms across ten selected FPGAs. For some combinations algorithm–FPGA family the implementation was not possible. This was because the amount of required resources exceeded the amount of resources available in the biggest device of a given family, or because the tools were not able to finish implementation within less than 48 hours. These cases are denoted by ‘N/A’ in the following tables. Specifically, BMW-256 did not fit in Cyclone II and Spartan3 due to the routing congestion, and BMW-512 was implemented successfully in only two out of ten FPGA families (Stratix III and Stratix IV). These problems were due to multi-operand additions exhausting all available routing resources. For SIMD in Spartan 3 and SIMD and ECHO in Cyclone II, resource utilization exceeded 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)

FPGA Family	Xilinx Families				Altera Families					
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV
BLAKE	44.43	84.93	128.90	136.24	53.67	61.52	61.70	100.67	118.98	126.23
BMW	N/A	5.00	12.00	15.21	N/A	6.79	7.06	10.35	12.38	12.08
CubeHash	94.76	196.04	274.05	328.19	114.05	137.70	137.50	177.21	232.88	250.06
ECHO	78.06	138.01	184.30	224.67	N/A	116.59	113.43	161.50	233.32	228.73
Fugue	79.56	137.63	218.44	272.41	95.63	111.22	109.03	142.84	207.43	125.99
Groestl	102.61	204.33	323.40	347.34	131.08	144.38	149.57	176.24	220.65	205.76
Hamsi	105.05	200.97	285.88	298.60	146.43	171.59	170.85	191.06	280.98	274.73
JH	142.47	276.93	380.81	415.46	172.92	227.07	225.38	294.90	387.75	410.17
Keccak	108.90	230.42	282.73	286.21	134.43	155.57	155.64	210.17	273.37	286.04
Luffa	138.10	289.35	340.72	322.17	171.59	199.20	199.24	215.80	301.30	316.16
Shabal	93.63	185.01	214.92	300.66	144.97	183.82	194.06	91.84	109.66	111.74
SHAvite-3	79.05	145.75	235.10	178.89	93.86	110.24	110.64	162.89	245.52	231.05
SIMD	N/A	40.13	54.90	60.23	23.02	26.37	26.77	42.62	54.89	57.77
Skein	37.86	68.16	117.95	132.31	46.35	54.90	57.13	73.94	92.87	98.82
SHA-512	79.96	157.88	227.58	270.56	108.68	118.75	126.50	150.31	212.27	217.63

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.

Table 4.9: Clock frequencies of all SHA-3 candidates (512-bit variants) and SHA-512 expressed in MHz (post placing and routing)

FPGA Family	Xilinx Families				Altera Families					
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV
BLAKE	34.76	68.52	99.68	103.47	38.57	45.77	47.06	71.27	89.53	106.07
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	9.59	9.89
CubeHash	98.74	202.63	269.69	317.36	110.02	137.46	137.70	158.60	204.21	220.70
ECHO	N/A	173.31	235.50	158.40	N/A	110.44	107.71	173.73	247.40	255.95
Fugue	74.58	137.46	221.58	234.69	91.68	104.64	104.20	137.70	199.80	65.64
Groestl	91.64	203.34	292.10	312.01	124.67	127.10	139.12	175.90	202.27	214.45
Hamsi	73.21	160.18	182.10	185.12	81.83	114.13	121.71	132.64	187.55	177.34
JH	123.21	256.64	394.48	412.54	172.71	220.85	224.37	294.81	390.63	388.65
Keccak	114.19	230.84	285.23	338.18	134.97	165.32	150.35	204.21	304.60	285.55
Luffa	94.51	202.88	240.33	195.73	141.08	167.03	172.35	189.74	268.10	264.62
Shabal	93.63	185.01	214.92	300.66	144.97	183.82	194.06	91.84	109.66	111.74
SHAvite-3	71.71	134.25	213.80	180.21	80.32	95.85	96.38	142.78	226.60	216.40
SIMD	N/A	36.73	43.40	46.30	N/A	22.04	22.74	34.33	49.82	51.93
Skein	37.46	63.94	119.09	138.52	46.22	54.90	55.98	70.55	90.34	99.09
SHA-512	67.68	128.25	183.70	238.83	86.41	94.86	99.75	123.03	162.42	188.71

In terms of throughput, the best performance is accomplished by ECHO, Keccak, and Luffa, followed by Groestl, BMW, and JH. Only Shabal is slightly slower than SHA-256 in terms of throughput. At the same time, it is the only candidate that outperforms SHA-256 in terms of area. The other candidates offering relatively low area include CubeHash, Hamsi, Fugue, 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 Luffa, Keccak, CubeHash, and Shabal, which are the only candidates outperforming SHA-256. The worst results in terms of this measure, belong to ECHO, BMW, and SIMD, which loose to SHA-256 by a factor of at least 3.

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 (>3.5), but their normalized area varies significantly from about 5 in case of Groestl, through 11 for BMW, up to over 23 in case of ECHO. SIMD is both relatively slow (less than 1.75 times faster than SHA-256) and big (more than 20 times bigger than SHA-256). The last group includes 8 candidates covering the range of the normalized throughputs from 0.99 to 3.2, and the normalized areas from 0.87 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. Five candidates, Keccak, CubeHash, Shabal, JH, and Luffa outperform SHA-512 in terms of the throughput to area ratio. Two more candidates, Groestl and Skein, have the overall

Table 4.10: Throughput of all SHA-3 candidates (256-bit variants) normalized to the throughput of SHA-256

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
ECHO	7.61	6.82	6.32	6.48	N/A	7.66	6.99	8.38	8.57	8.20	6.79	7.94	7.41
Keccak	7.84	8.40	7.15	6.09	7.12	7.54	7.08	8.05	7.41	7.56	7.32	7.45	7.40
Luffa	6.24	6.62	5.41	4.30	5.70	6.06	5.69	5.18	5.13	5.25	5.57	5.49	5.52
Groestl	3.97	4.01	4.40	3.97	3.73	3.76	3.66	3.63	3.22	2.93	4.08	3.47	3.71
BMW	N/A	2.06	3.43	3.65	N/A	3.72	3.63	4.48	3.79	3.61	2.95	3.83	3.48
JH	3.22	3.17	3.02	2.77	2.87	3.45	3.22	3.54	3.30	3.40	3.04	3.29	3.19
CubeHash	2.41	2.52	2.45	2.46	2.13	2.36	2.21	2.39	2.23	2.33	2.46	2.27	2.35
Fugue	2.02	1.77	1.95	2.05	1.79	1.90	1.75	1.93	1.98	1.18	1.94	1.73	1.81
Hamsi	1.78	1.72	1.70	1.49	1.82	1.96	1.83	1.72	1.79	1.71	1.67	1.80	1.75
SIMD	N/A	1.84	1.74	1.61	1.53	1.60	1.53	2.05	1.87	1.92	1.73	1.74	1.73
BLAKE	1.72	1.67	1.75	1.56	1.53	1.60	1.51	2.07	1.73	1.80	1.67	1.70	1.69
SHAvite-3	1.74	1.62	1.81	1.16	1.52	1.63	1.54	1.90	2.03	1.87	1.56	1.74	1.66
Skein	1.62	1.48	1.77	1.67	1.46	1.58	1.55	1.68	1.50	1.55	1.63	1.55	1.58
Shabal	1.19	1.19	0.96	1.13	1.35	1.57	1.56	0.62	0.52	0.52	1.11	0.91	0.99

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

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
Shabal	0.77	0.74	0.66	0.75	0.49	0.62	0.62	1.78	1.65	1.67	0.73	0.99	0.87
CubeHash	1.75	1.74	1.59	1.96	1.94	1.99	1.95	1.97	1.82	1.83	1.75	1.91	1.85
Hamsi	2.08	2.07	1.67	1.76	2.00	2.01	2.01	2.36	2.19	2.18	1.89	2.12	2.02
Fugue	2.91	2.90	1.65	2.17	3.59	3.60	3.59	2.44	2.27	3.43	2.34	3.10	2.77
Luffa	3.01	3.07	2.21	3.26	2.57	2.66	2.59	3.14	2.87	2.88	2.85	2.78	2.81
JH	4.54	4.34	2.37	2.86	3.81	3.85	3.86	3.58	3.34	3.36	3.40	3.63	3.53
Keccak	3.93	4.02	2.96	3.60	3.81	4.05	3.85	4.33	3.99	4.01	3.60	4.00	3.84
SHAvite-3	4.69	4.69	2.50	2.61	6.01	6.02	6.04	3.09	2.88	2.78	3.46	4.19	3.88
BLAKE	4.57	4.54	3.54	3.81	4.15	4.23	4.15	3.71	3.44	3.46	4.09	3.84	3.94
Skein	4.04	4.06	3.77	3.71	3.86	3.96	4.01	4.74	4.40	4.42	3.89	4.22	4.09
Groestl	8.97	8.93	3.71	4.16	3.47	3.51	3.69	3.76	6.01	5.93	5.93	4.26	4.87
BMW	N/A	11.39	10.12	9.20	N/A	11.61	11.64	12.88	11.95	11.99	10.20	12.00	11.29
SIMD	N/A	22.52	20.75	23.76	15.44	15.53	15.56	26.30	24.36	24.46	22.31	19.70	20.53
ECHO	30.47	30.19	11.59	15.55	N/A	41.20	41.21	21.14	19.62	19.75	20.18	26.83	23.64

Table 4.12: Throughput to Area Ratio of all SHA-3 candidates normalized to the throughput to area ratio of SHA-256

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
Luffa	2.07	2.16	2.45	1.32	2.22	2.28	2.19	1.65	1.79	1.82	1.95	1.98	1.97
Keccak	1.99	2.09	2.42	1.69	1.87	1.86	1.84	1.86	1.86	1.89	2.03	1.86	1.93
CubeHash	1.37	1.45	1.54	1.26	1.10	1.19	1.13	1.22	1.22	1.28	1.40	1.19	1.27
Shabal	1.53	1.62	1.46	1.51	2.76	2.53	2.50	0.35	0.32	0.31	1.53	0.92	1.13
JH	0.71	0.73	1.28	0.97	0.75	0.90	0.83	0.99	0.99	1.01	0.89	0.91	0.90
Hamsi	0.85	0.83	1.02	0.85	0.91	0.97	0.91	0.73	0.82	0.78	0.89	0.85	0.86
Groestl	0.44	0.45	1.18	0.96	1.08	1.07	0.99	0.97	0.54	0.49	0.69	0.81	0.76
Fugue	0.69	0.61	1.18	0.94	0.50	0.53	0.49	0.79	0.87	0.34	0.83	0.56	0.65
SHAvite-3	0.37	0.35	0.73	0.45	0.25	0.27	0.25	0.62	0.71	0.67	0.45	0.41	0.43
BLAKE	0.38	0.37	0.49	0.41	0.37	0.38	0.36	0.56	0.50	0.52	0.41	0.44	0.43
Skein	0.40	0.36	0.47	0.45	0.38	0.40	0.39	0.35	0.34	0.35	0.42	0.37	0.39
ECHO	0.25	0.23	0.55	0.42	N/A	0.19	0.17	0.40	0.44	0.42	0.34	0.30	0.31
BMW	N/A	0.18	0.34	0.40	N/A	0.32	0.31	0.35	0.32	0.30	0.29	0.32	0.31
SIMD	N/A	0.08	0.08	0.07	0.10	0.10	0.10	0.08	0.08	0.08	0.08	0.09	0.08

Table 4.13: Throughput of all SHA-3 candidates (512-bit variants) normalized to the throughput of SHA-512

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	4.78	4.25	N/A	4.51	4.51
Groestl	3.78	4.43	4.44	3.65	4.03	3.74	3.90	3.99	3.48	3.17	4.06	3.71	3.84
Luffa	3.53	4.00	3.31	2.07	4.13	4.46	4.37	3.90	4.18	3.55	3.14	4.09	3.68
ECHO	N/A	3.53	3.35	1.73	N/A	3.04	2.82	3.69	3.98	3.54	2.74	3.39	3.13
Keccak	3.20	3.42	2.95	2.69	2.97	3.31	2.86	3.15	3.56	2.87	3.05	3.11	3.09
JH	2.05	2.25	2.42	1.94	2.25	2.62	2.53	2.70	2.71	2.32	2.16	2.51	2.36
SIMD	N/A	2.58	2.13	1.74	N/A	2.09	2.05	2.51	2.76	2.48	2.12	2.36	2.27
CubeHash	1.85	2.00	1.86	1.68	1.61	1.83	1.75	1.63	1.59	1.48	1.84	1.65	1.72
SHAvite-3	1.51	1.49	1.65	1.07	1.32	1.44	1.37	1.65	1.98	1.63	1.41	1.55	1.49
BLAKE	1.43	1.49	1.52	1.21	1.25	1.35	1.32	1.62	1.54	1.57	1.41	1.43	1.42
Skein	1.18	1.06	1.38	1.24	1.14	1.23	1.20	1.22	1.19	1.12	1.21	1.18	1.19
Hamsi	0.91	1.05	0.84	0.65	0.80	1.02	1.03	0.91	0.97	0.79	0.85	0.91	0.89
Shabal	0.88	0.91	0.74	0.80	1.06	1.23	1.23	0.47	0.43	0.37	0.83	0.70	0.75
Fugue	0.70	0.68	0.76	0.62	0.67	0.70	0.66	0.71	0.78	0.22	0.69	0.58	0.62

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

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
Shabal	0.34	0.32	0.34	0.47	0.25	0.32	0.32	0.86	0.81	0.81	0.36	0.49	0.44
CubeHash	0.90	0.91	0.88	1.34	1.07	1.10	1.08	0.95	0.89	0.89	0.99	0.99	0.99
Keccak	1.52	1.55	1.50	2.20	1.71	1.71	1.73	1.96	1.84	1.84	1.67	1.80	1.74
Fugue	1.67	1.55	1.17	1.67	2.27	2.27	2.28	1.40	1.29	3.00	1.50	2.00	1.78
JH	1.92	1.94	1.32	2.00	2.04	2.06	2.07	1.82	1.71	1.72	1.77	1.90	1.85
Skein	1.83	1.88	2.05	2.52	2.06	2.08	2.12	2.37	2.22	2.21	2.05	2.17	2.12
Hamsi	2.21	2.21	2.27	2.81	2.39	2.39	2.40	3.20	2.96	2.95	2.36	2.70	2.56
Luffa	2.85	2.77	2.34	3.43	2.99	3.03	3.05	3.43	3.18	3.20	2.82	3.14	3.01
SHAvite-3	4.04	4.08	2.49	2.71	6.15	6.13	6.17	2.77	2.60	2.59	3.25	4.04	3.70
BLAKE	3.75	3.88	3.66	4.94	4.18	4.20	4.29	3.49	3.27	3.27	4.02	3.76	3.86
Groestl	7.71	7.84	3.74	5.42	4.91	5.85	4.00	3.50	5.71	5.73	5.92	4.86	5.26
ECHO	N/A	13.38	6.02	7.01	N/A	21.09	21.20	10.43	9.79	9.79	8.27	13.49	11.23
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	11.64	11.49	N/A	11.57	11.57
SIMD	N/A	20.63	23.44	31.14	N/A	16.57	16.62	26.44	24.78	24.66	24.69	21.36	22.55

Table 4.15: Throughput to Area Ratio of all SHA-3 candidates normalized to the throughput to area ratio of SHA-512

Family	Xilinx Families				Altera Families						Geometric Means		
	S-3	V-4	V-5	V-6	C-II	C-III	C-IV	S-II	S-III	S-IV	Xilinx	Altera	Overall
Keccak	2.11	2.20	1.97	1.22	1.73	1.93	1.66	1.61	1.94	1.56	1.83	1.73	1.77
CubeHash	2.04	2.21	2.12	1.25	1.51	1.66	1.62	1.72	1.78	1.66	1.86	1.66	1.73
Shabal	2.60	2.82	2.19	1.71	4.25	3.88	3.87	0.55	0.53	0.46	2.29	1.43	1.73
JH	1.07	1.16	1.83	0.97	1.10	1.27	1.22	1.48	1.58	1.35	1.22	1.32	1.28
Luffa	1.24	1.44	1.42	0.61	1.38	1.47	1.43	1.14	1.31	1.11	1.11	1.30	1.22
Groestl	0.49	0.56	1.19	0.67	0.82	0.64	0.97	1.14	0.61	0.55	0.69	0.76	0.73
Skein	0.64	0.57	0.67	0.49	0.55	0.59	0.56	0.52	0.54	0.51	0.59	0.54	0.56
SHAvite-3	0.37	0.36	0.66	0.40	0.21	0.23	0.22	0.60	0.76	0.63	0.43	0.38	0.40
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.41	0.37	N/A	0.39	0.39
BLAKE	0.38	0.38	0.41	0.24	0.30	0.32	0.31	0.46	0.47	0.48	0.35	0.38	0.37
Fugue	0.42	0.44	0.65	0.37	0.30	0.31	0.29	0.50	0.61	0.07	0.46	0.29	0.35
Hamsi	0.41	0.48	0.37	0.23	0.33	0.42	0.43	0.28	0.33	0.27	0.36	0.34	0.35
ECHO	N/A	0.26	0.56	0.25	N/A	0.14	0.13	0.35	0.41	0.36	0.33	0.25	0.28
SIMD	N/A	0.12	0.09	0.06	N/A	0.13	0.12	0.09	0.11	0.10	0.09	0.11	0.10

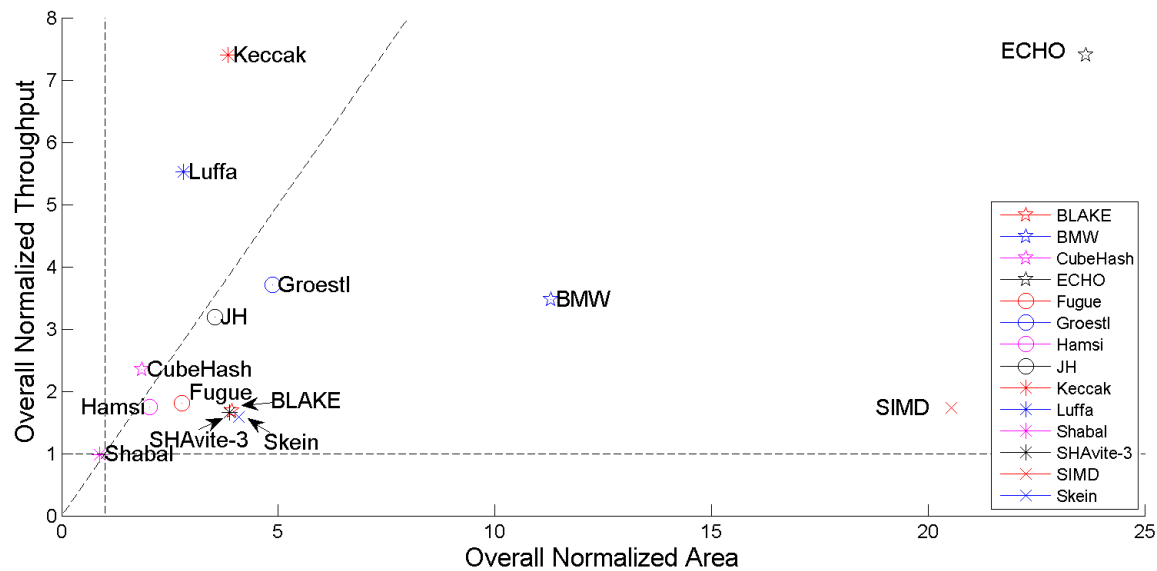


Figure 4.1: Relative performance of all Round 2 SHA-3 Candidates (256-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-256 used as a reference point.)

normalized ratio higher than 0.5. In terms of throughput, only five candidates, BMW, Groestl, Luffa, ECHO, and Keccak, outperform SHA-512 by a factor larger than 3. The additional six candidates have a normalized throughput in the range from 1 to 3. Three candidates, Hamsi, Shabal, and Fugue, are slower than SHA-512. In terms of area, only Shabal and CubeHash, in their 512-bit variants, are smaller than SHA-512. The spread of results is much larger than in the case of throughput, with the smallest SHA-3 candidate, Shabal, less than half the size as SHA-512, and the largest SIMD, lagging behind by a factor of over 22. The group following CubeHash in terms of area, including Keccak, Fugue, JH Skein, and Hamsi, covers the range between 1.7 and 2.6, 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. Five algorithms outperform SHA-512 in terms of the throughput to area ratio. Out of them Luffa is the fastest and most area consuming, while Shabal is the slowest and the smallest. SIMD is approximately 18 times worse than Keccak in terms of the throughput to area ratio, and ECHO is 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

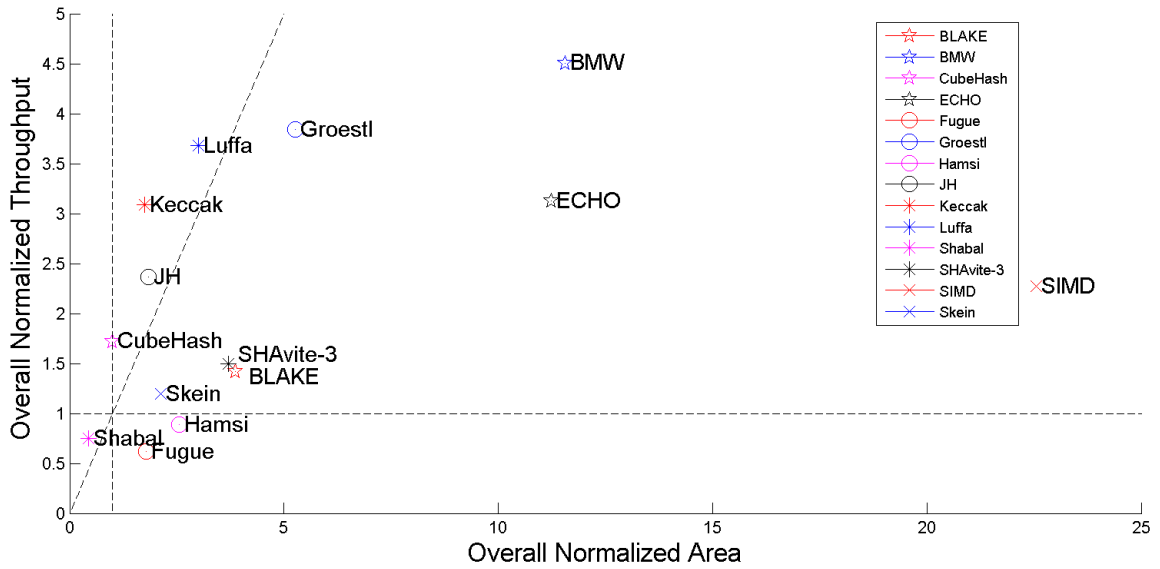


Figure 4.2: Relative performance of all Round 2 SHA-3 Candidates (512-bit variants) in terms of the overall normalized throughput and the overall normalized area (with SHA-512 used as a reference point.)

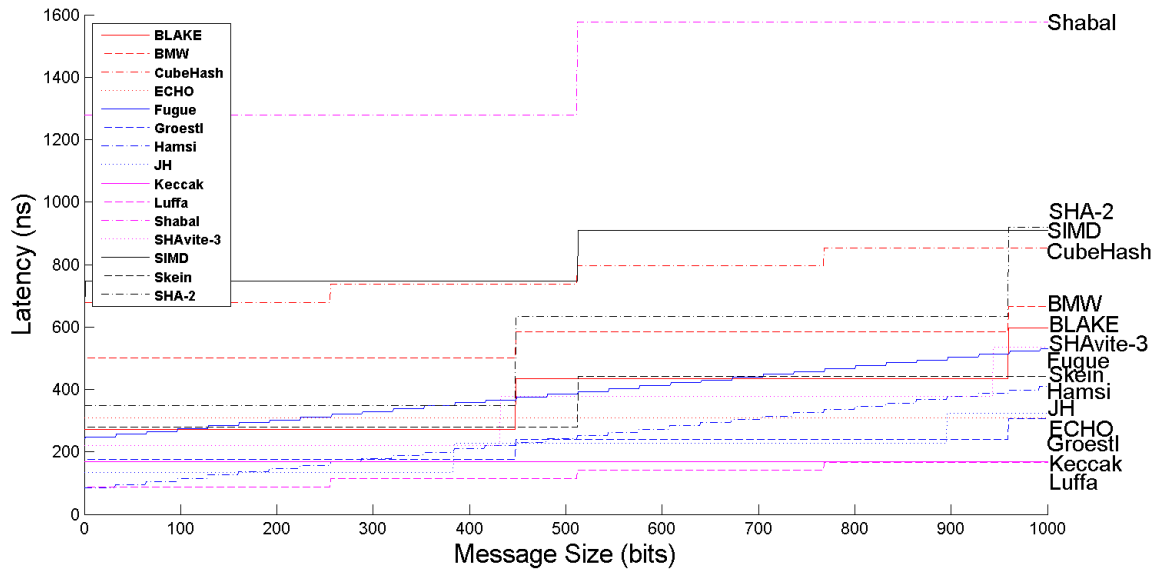


Figure 4.3: Execution time vs. message size for short messages up to 1,000 bits. 256-bit variants of all SHA-3 Candidates and SHA-256 in Virtex 5.

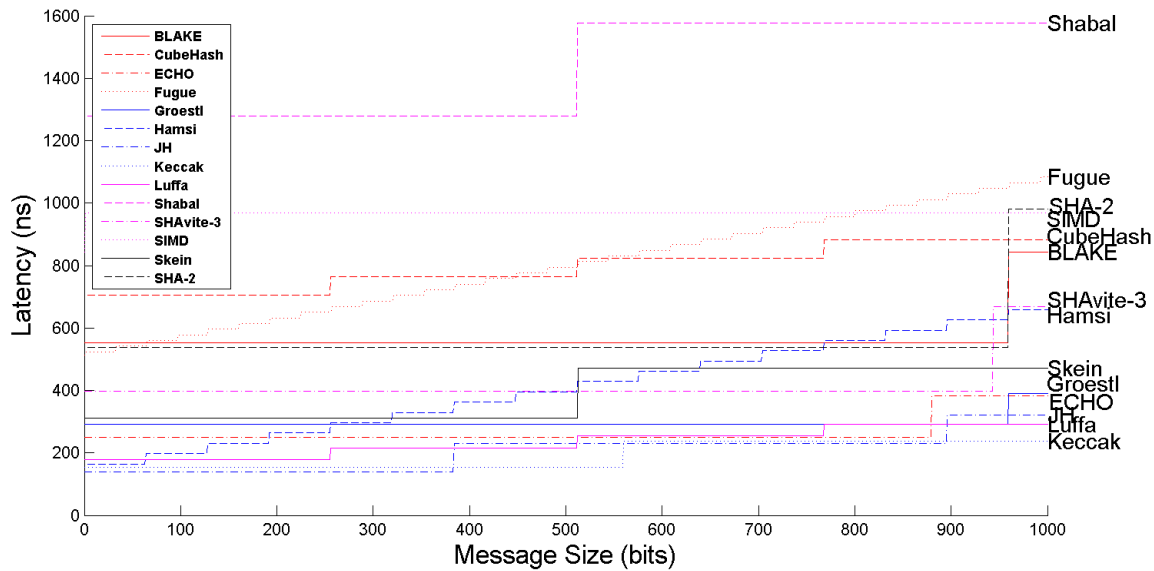


Figure 4.4: Execution time vs. message size for short messages up to 1,000 bits. 512-bit variants of all SHA-3 Candidates and SHA-512 in Virtex 5.

the behavior of the developed hash modules for short messages. For some applications, an 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 4.4. Message sizes used in these diagrams represent sizes before padding. Padding is assumed to be done outside of a hash core (e.g., in software), and its time is not included in the execution time.

For the 256-bit variants of all algorithms, the only one performing worse than SHA-256 for all message sizes in the investigated range is Shabal. BMW, SIMD, and CubeHash perform worse most of the time. The best performers are Luffa, Keccak, Groestl, ECHO, and JH. For the 512-bit variants, the worst performing algorithms are Shabal, Fugue, SIMD, CubeHash. BMW was not implemented in Virtex 5 because of the routing congestion. The best performers are the same as in the SHA-256 case, with the addition of Skein.

In Figures 4.5, 4.6, 4.7, 4.8, we demonstrate the influence of the message size on ranking of candidates in terms of the throughput and throughput to area ratio. We consider the following four message sizes: 40 bytes, 576 bytes, 1500 bytes, and “long”. 1500 bytes is the Maximum Transmission Unit (MTU) for Ethernet v2, 576 bytes is the Maximum Transmission Unit (MTU) for Internet IPv4 Path, 40 bytes is the smallest packet size popular in the computer-network protocols. “Long” messages are messages long enough, so that any initialization and finalization execution time overheads can be neglected compared

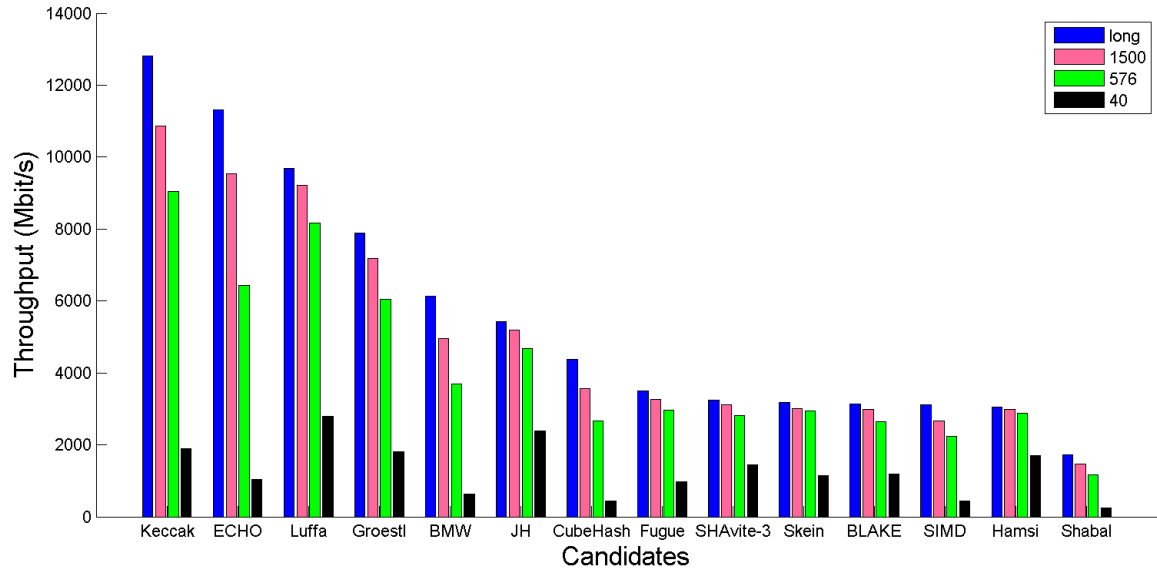


Figure 4.5: Throughput as a function of the message size for all SHA-3 Candidates (256-bit variants) and SHA-256 in Virtex 5. Investigated message sizes are: 40 bytes, 576 bytes, 1500 bytes, and “long”

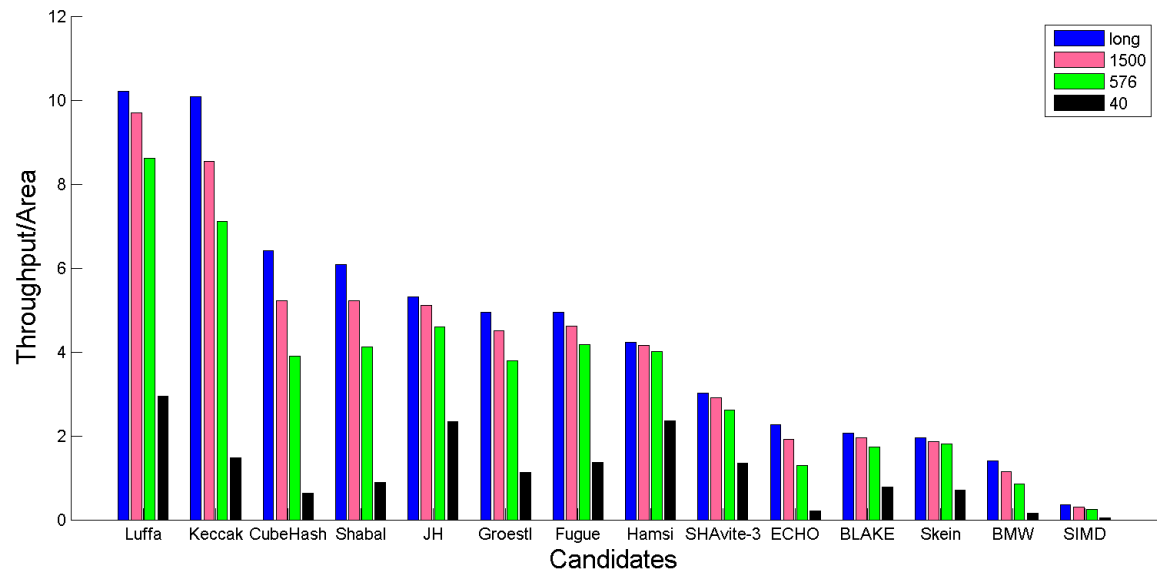


Figure 4.6: Throughput to area ratio as a function of the message size for all SHA-3 Candidates (256-bit variants) and SHA-256 in Virtex 5. Investigated message sizes are: 40 bytes, 576 bytes, 1500 bytes, and “long”

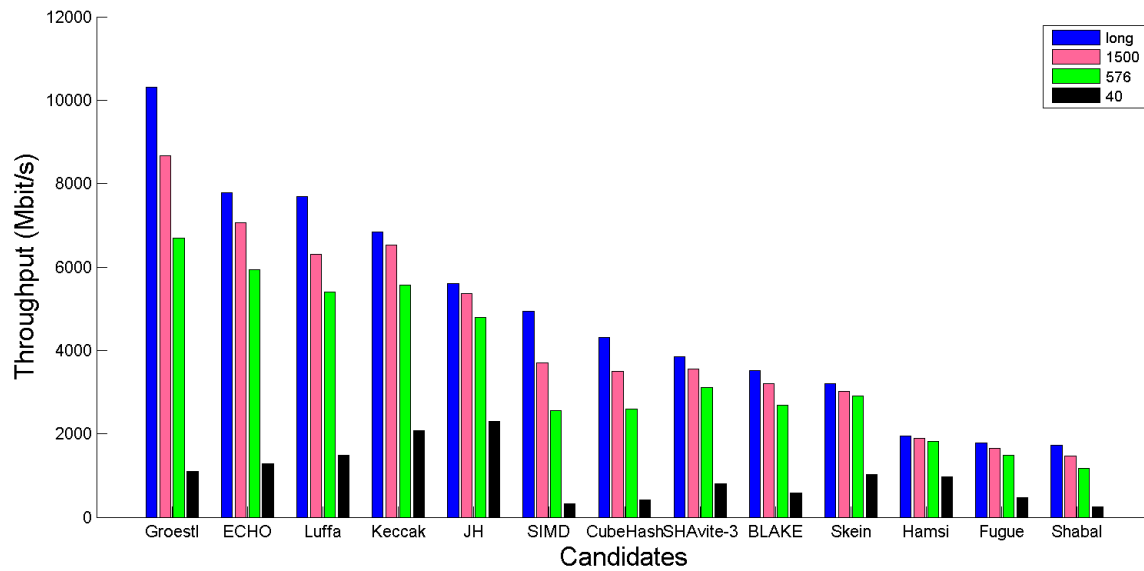


Figure 4.7: Throughput as a function of the message size for all SHA-3 Candidates (512-bit variants) and SHA-512 in Virtex 5. Investigated message sizes are: 40 bytes, 576 bytes, 1500 bytes, and “long”

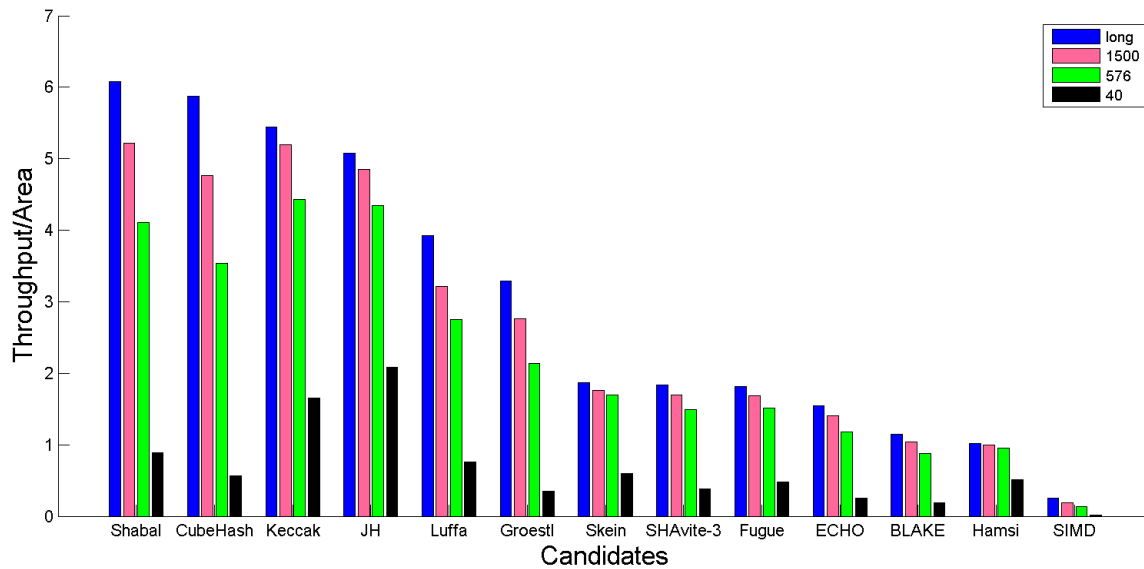
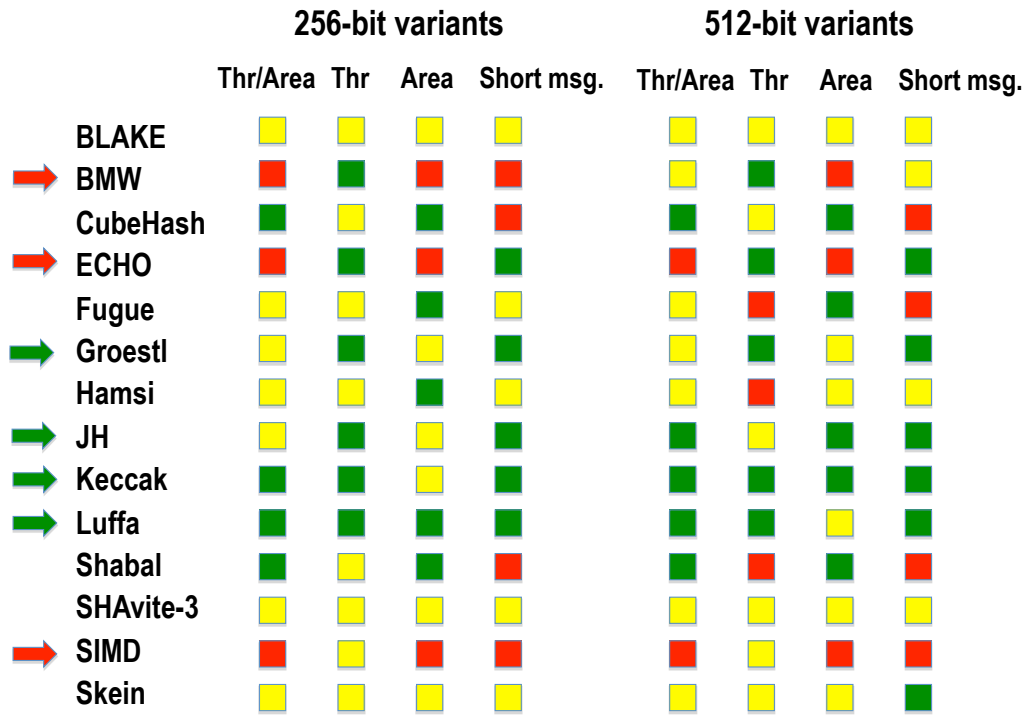


Figure 4.8: Throughput to area ratio as a function of the message size for all SHA-3 Candidates (512-bit variants) and SHA-512 in Virtex 5. Investigated message sizes are: 40 bytes, 576 bytes, 1500 bytes, and “long”

to the basic message processing time. Based on these figures, the influence of the message size on ranking of candidates, compared to ranking for ideal “long” messages is relatively small for 1500 byte packets. For 576 byte packets, already several algorithms change their positions. For 40-byte packets, the influence is already quite strong. For example, as shown in Figure 4.7, for 512-bit variants of all algorithm, the ranking of the first 5 candidates in terms of throughput is reversed for 40-byte packets compared to the ranking for long messages.

In Figure 4.9, 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 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 JH and Groestl, which excel in 5 and 4 categories, respectively, mostly 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 4 categories, mostly area and throughput to area ratio. Out of the remaining candidates, the ones with the highest potential are CubeHash and Shabal, excelling in terms of area and throughput to area ratio for both variants of the algorithm.



1

Figure 4.9: 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

The most comprehensive results published by other groups come from Baldwin et al. [15], Matsuo et al. [18], and Guo et al. [47]. All these groups have published results for all 14 Round 2 candidates. Majority of published results concern 256-bit variants of the candidates, implemented using Xilinx Virtex 5 FPGAs. Additionally, several other groups have published results regarding either individual candidates or a subset of candidates. Out of these publications, the most important ones are the papers by Detrey et al. [48] about Shabal, Gauravaram et al. [32] about Groestl, and Lu et al. [49] about ECHO.

Sometimes, results published by other groups are somewhat difficult to compare with our results because of different assumptions used. In particular, the basic designs by Baldwin et al. [15] include padding. In the comparison below, we use Baldwin's results obtained under the assumption that padding was done in software. These modified results have been included in the Baldwin's presentation at the Second SHA-3 Candidate Conference. At the same time, it should be noted that one of the main assumptions that limits the throughput and throughput to area ratio of the Baldwin's designs, is his assumption about the 32-bit input/output interface compared to our 64-bit input/output interface. Since the influence of the interface cannot be easily quantified without the detailed knowledge of the designs, in this comparison, we take into account only original results obtained by Baldwin using his 32-bit interface.

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, because majority of papers from other groups target this particular family. This table demonstrates that the GMU results are the best in terms of the throughput to area ratio for all candidates except BMW, Groestl, Shabal, and SHAvite-3. The exact reasons why our designs are inferior to other group's designs for these particular four algorithms will be

investigated in the future.

In Figure 5.1, the best results from other groups have been summarized using the throughput vs. area graph. Please note that no proper post-place-and-route results seem to be available for Skein-512-256, i.e., Skein with the 512-bit state and 256-bit output size. Instead, majority of groups published by mistake results for Skein-256-256.

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	1660	2676	1.61	Matsuo et al. [18]	1523	3143	2.06
BMW	4350	8704	2.00	Matsuo et al. [18]	4353	6141	1.41
CubeHash	590	2960	5.02	Matsuo et al. [18]	684	4385	6.41
ECHO	9333	14860	1.59	Lu et al. [49]	4982	11323	2.27
Fugue	1689	914	0.54	Baldwin et al. [15]	708	3495	4.94
Groestl	1722	10276	5.97	Gauravaram et al. [32]	1597	7885	4.94
Hamsi	718	1680	2.34	Matsuo et al. [18]	720	3049	4.24
JH	1291	3380	2.62	Baldwin et al. [15]	1018	5416	5.32
Keccak	1433	8397	5.86	Matsuo et al. [18]	1272	12817	10.08
Luffa	1048	7424	7.08	Matsuo et al. [18]	949	9692	10.21
Shabal	153	2051	13.41	Detrey et al. [48]	283	1719	6.08
SHAvite-3	1063	3382	3.18	Matsuo et al. [18]	1076	3253	3.02
SIMD	3987	835	0.21	Matsuo et al. [18]	8922	3123	0.35

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. In Figure 5.3, we rank all candidates in terms of the throughput to area ratio. The best results are obtained by Shabal, Luffa, and Keccak, followed by CubeHash, Groestl, and JH. In Figure 5.4, we take exactly the same implementations (selected because of the superior throughput to area ratio) and rank them according to the highest throughput. In Figure 5.5, we rank the same implementations in terms of minimum area. These graphs demonstrate that for Luffa, Keccak, Groestl, and JH, their high throughput to area ratio comes from high throughput and medium area. For CubeHash, the good ratio is the result of small area and medium throughput. Shabal is by far the best in terms of area and the worst in terms of throughput.

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, when the result by Detrey et al. is taken into account. Additionally, Shabal will most

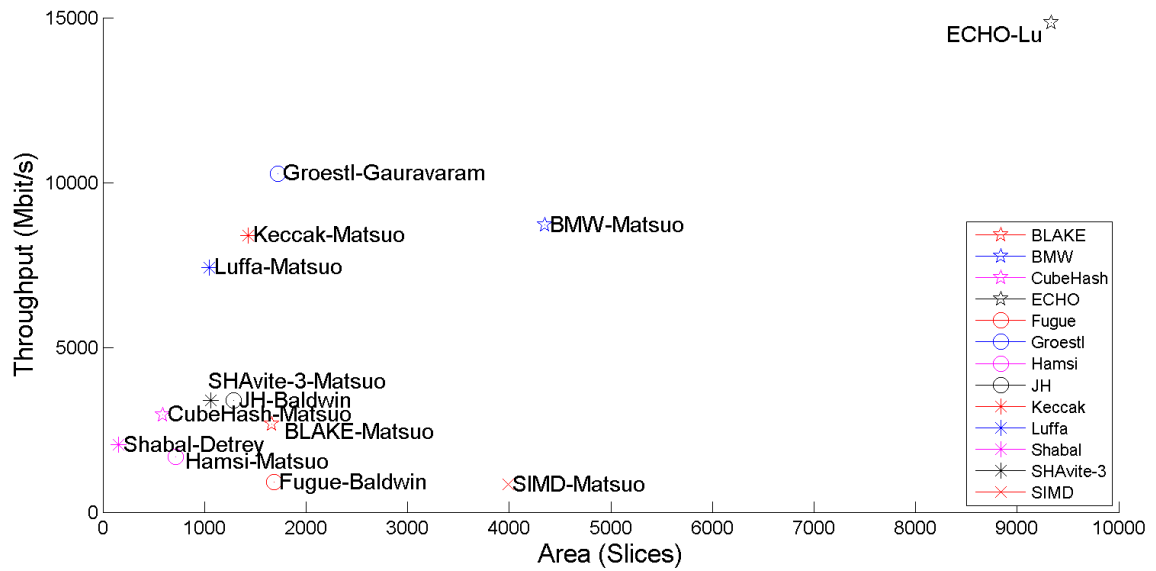


Figure 5.1: Best *published* results vs. GMU results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput to area ratio in Xilinx Virtex 5

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	1523	3142.7	2.06	GMU
BMW	4350	8704	2.00	Matsuo et al. [18]
CubeHash	684	4384.8	6.41	GMU
ECHO	4982	11323.39	2.27	GMU
Fugue	708	3494.98	4.94	GMU
Groestl	1722	10276	5.97	Gauravaram et al. [32]
Hamsi	720	3049.39	4.24	GMU
JH	1018	5415.96	5.32	GMU
Keccak	1272	12816.87	10.08	GMU
Luffa	949	9691.59	10.21	GMU
Shabal	153	2051	13.41	Detrey et al. [48]
SHAvite-3	1063	3382	3.18	Matsuo et al. [18]
SIMD	8922	3123.2	0.35	GMU
Skein	1621	3178.44	1.96	GMU

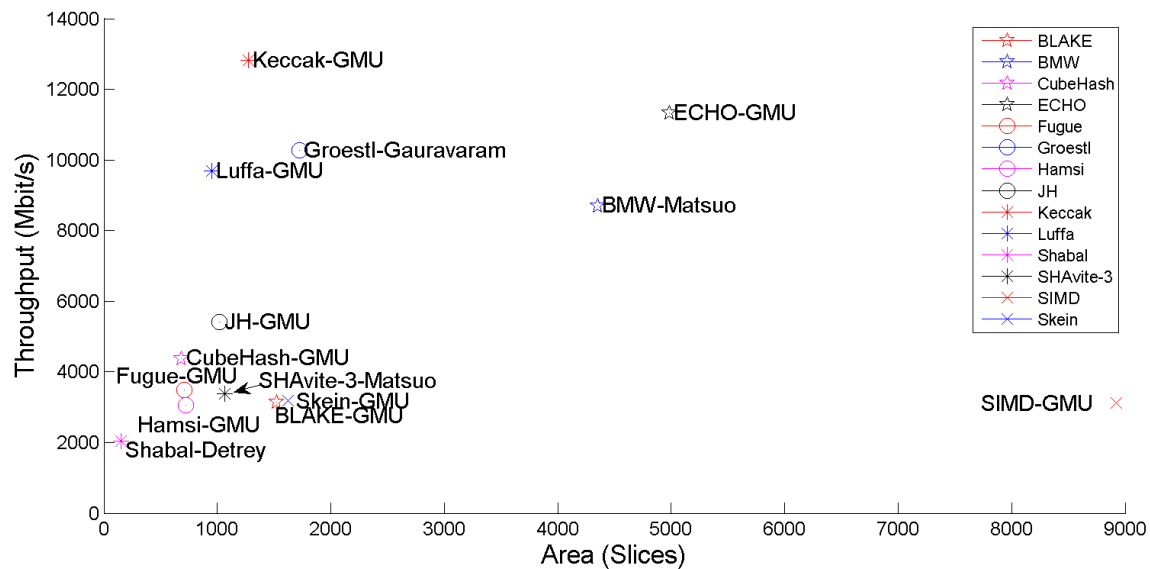


Figure 5.2: Best results for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput vs. area in Virtex 5

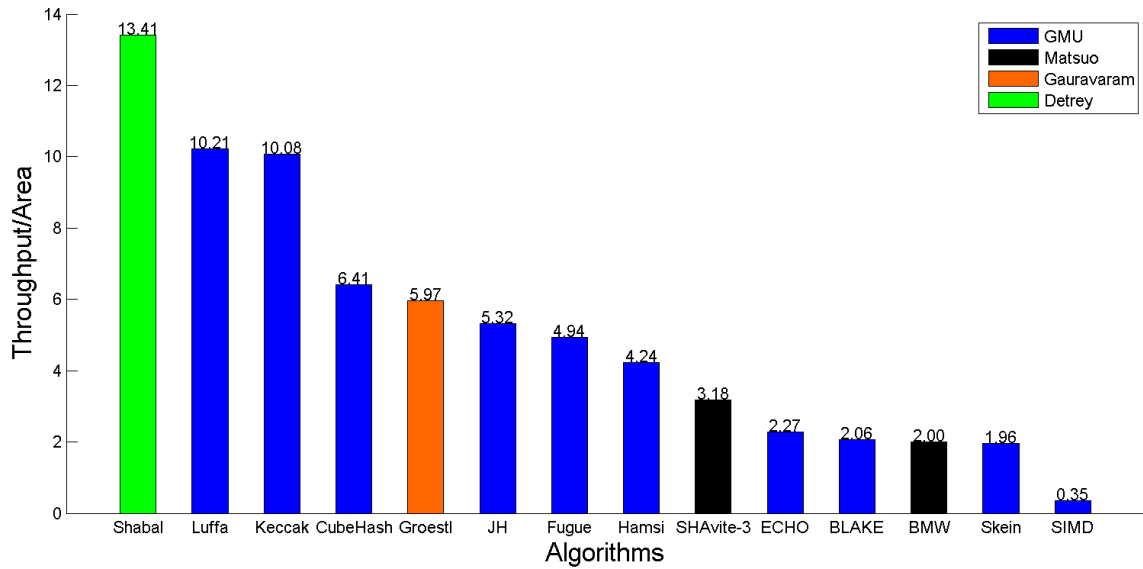


Figure 5.3: Best results from multiple groups for all Round 2 SHA-3 Candidates (256-bit variants) in terms of throughput to area ratio in Virtex 5

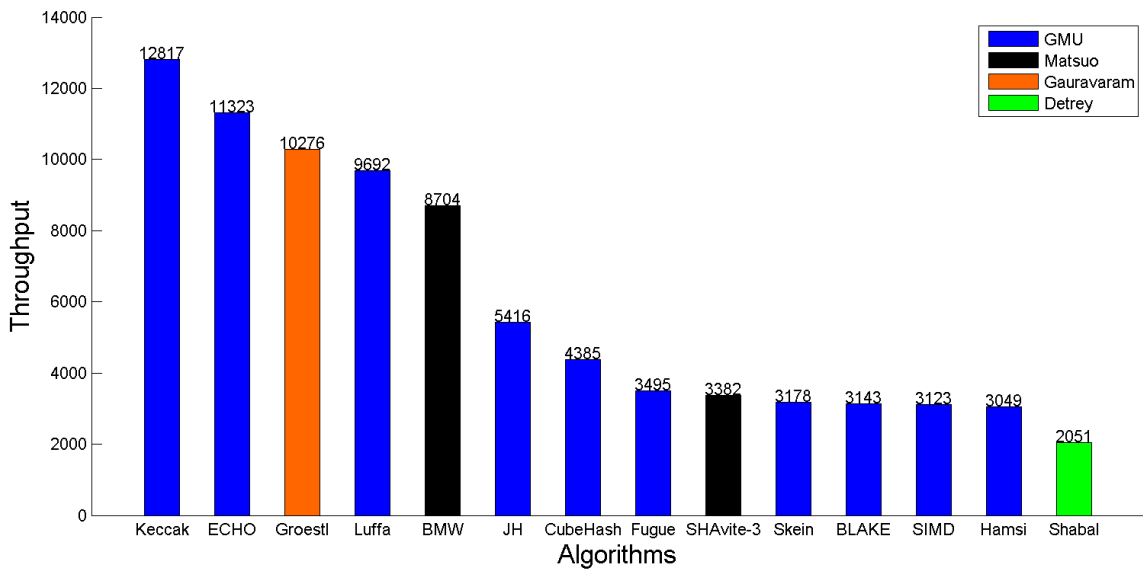


Figure 5.4: Throughput for the designs demonstrating the best throughput to area ratio (256-bit variants, Virtex 5)

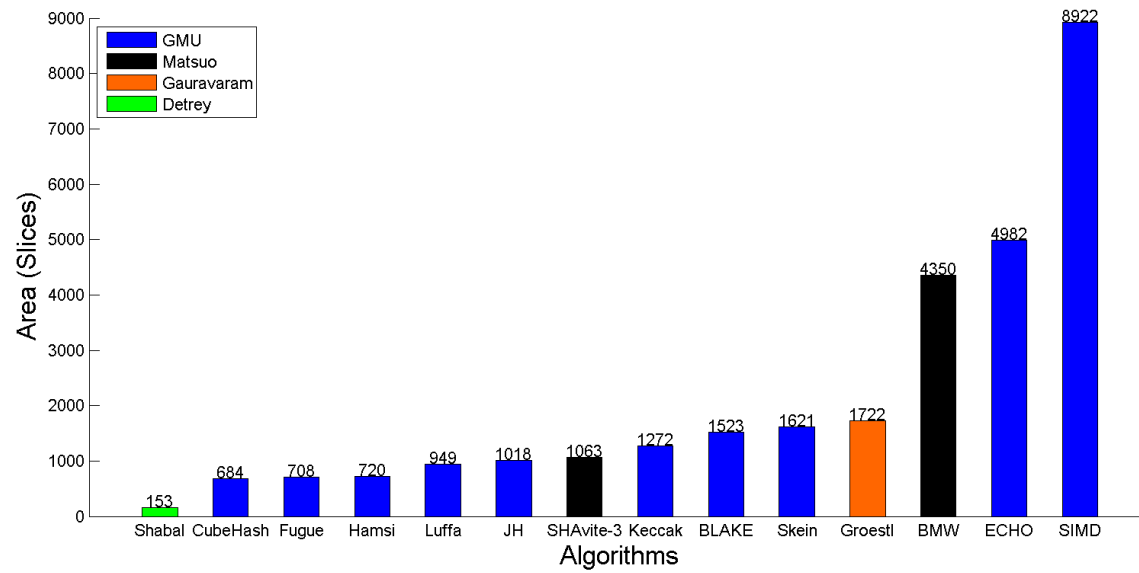


Figure 5.5: Area for the designs demonstrating the best throughput to area ratio (256-bit variants, Virtex 5)

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 7.5 in terms of the throughput (Keccak vs. Shabal), over 27 times in terms of area (Shabal vs. ECHO), and about 25 in terms of our primary optimization target, the throughput to area ratio (Luffa vs. SIMD). Only four candidates, Luffa, Keccak, CubeHash, and Shabal, have demonstrated the throughput to area ratio better than the current standard SHA-256. Out of these four algorithms, Keccak and Luffa have also demonstrated very high throughputs, while Shabal and CubeHash outperformed other candidates in terms of minimum area. Shabal is the only candidate that outperforms SHA-256 in terms of area, and at the same time the only one losing to SHA-256 in terms of throughput.

For the 512-bit variants, the ratio of the best result to the worst result was equal to about 7 in terms of the throughput (BMW vs. Fugue), over 50 in terms of area (Shabal vs. SIMD), and about 18 in terms of our primary optimization target, the throughput to area ratio (Keccak vs. SIMD). Five candidates, Keccak, CubeHash, Shabal, JH, and Luffa, have demonstrated the throughput to area ratio better than the current standard SHA-512. Out of these algorithms, Luffa, Keccak, and JH have also demonstrated high throughputs, while Shabal and CubeHash outperformed other candidates in terms of minimum area. Almost all candidates, except Fugue, Shabal, and Hamsi, outperform SHA-512 in terms of the throughput, but only two, Shabal and CubeHash, match 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. 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 and Fall 2010 editions 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. The authors would like also to thank Rajesh Velegalati from the Cryptographic Engineering Research Group at GMU for his tremendous help in generating comprehensive and optimized results presented in this paper using ATHENa.

Appendix A

Absolute Results for Ten FPGA Families

Table A.1: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Xilinx Spartan 3 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	34.76	44.43	0.78	1227	1083	1.13	7390	3913	1.89	0.17	0.28	0.60
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
CubeHash	98.74	94.76	1.04	1580	1516	1.04	1785	1502	1.19	0.89	1.01	0.88
ECHO	N/A	78.06	N/A	N/A	4796	N/A	N/A	26117	N/A	N/A	0.18	N/A
Fugue	74.58	79.56	0.94	597	1273	0.47	3286	2494	1.32	0.18	0.51	0.36
Groestl	91.64	102.61	0.89	3236	2502	1.29	15210	7686	1.98	0.21	0.33	0.65
Hamsi	73.21	105.05	0.70	781	1121	0.70	4353	1784	2.44	0.18	0.63	0.29
JH	123.21	142.47	0.86	1752	2026	0.86	3789	3894	0.97	0.46	0.52	0.89
Keccak	114.19	108.90	1.05	2741	4937	0.56	2991	3371	0.89	0.92	1.46	0.63
Luffa	94.51	138.10	0.68	3024	3928	0.77	5625	2576	2.18	0.54	1.52	0.35
Shabal	93.63	93.63	1.00	749	749	1.00	664	664	1.00	1.13	1.13	1.00
SHAvite-3	71.71	79.05	0.91	1288	1094	1.18	7965	4017	1.98	0.16	0.27	0.59
SIMD	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Skein	37.46	37.86	0.99	1009	1020	0.99	3618	3460	1.05	0.28	0.29	0.95
SHA-2	67.68	79.96	0.85	856	630	1.36	1973	857	2.30	0.43	0.73	0.59

Table A.2: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Xilinx Virtex 4 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	68.52	84.93	0.81	2419	2071	1.17	7566	3913	1.93	0.32	0.53	0.60
BMW	N/A	5.00	N/A	N/A	2560	N/A	N/A	9806	N/A	N/A	0.26	N/A
CubeHash	202.63	196.04	1.03	3242	3137	1.03	1769	1495	1.18	1.83	2.10	0.87
ECHO	173.31	138.01	1.26	5725	8479	0.68	26114	25990	1.00	0.22	0.33	0.67
Fugue	137.46	137.63	1.00	1100	2202	0.50	3023	2494	1.21	0.36	0.88	0.41
Groestl	203.34	204.33	1.00	7180	4982	1.44	15312	7691	1.99	0.47	0.65	0.72
Hamsi	160.18	200.97	0.80	1709	2144	0.80	4311	1785	2.42	0.40	1.20	0.33
JH	256.64	276.93	0.93	3650	3939	0.93	3789	3738	1.01	0.96	1.05	0.91
Keccak	230.84	230.42	1.00	5540	10445	0.53	3029	3457	0.88	1.83	3.02	0.61
Luffa	202.88	289.35	0.70	6492	8230	0.79	5412	2643	2.05	1.20	3.11	0.39
Shabal	185.01	185.01	1.00	1480	1480	1.00	633	633	1.00	2.34	2.34	1.00
SHAvite-3	134.25	145.75	0.92	2412	2017	1.20	7968	4042	1.97	0.30	0.50	0.61
SIMD	36.73	40.13	0.92	4179	2283	1.83	40261	19391	2.08	0.10	0.12	0.88
Skein	63.94	68.16	0.94	1723	1837	0.94	3662	3499	1.05	0.47	0.52	0.90
SHA-2	128.25	157.88	0.81	1621	1244	1.30	1952	861	2.27	0.83	1.44	0.58

Table A.3: 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	99.68	128.90	0.77	3,520	3,143	1.12	3,064	1,523	2.01	1.15	2.06	0.56
BMW	N/A	8.30	N/A	N/A	4,250	N/A	N/A	4,855	N/A	N/A	0.88	N/A
CubeHash	269.69	274.05	0.98	4,315	4,385	0.98	734	684	1.07	5.88	6.41	0.92
ECHO	235.50	184.30	1.28	7,779	11,323	0.69	5,044	4,982	1.01	1.54	2.27	0.68
Fugue	221.58	218.44	1.01	1,773	3,495	0.51	979	708	1.38	1.81	4.94	0.37
Groestl	292.10	323.40	0.90	10,314	7,885	1.31	3,138	1,597	1.96	3.29	4.94	0.67
Hamsi	182.10	285.88	0.64	1,942	3,049	0.64	1,900	720	2.64	1.02	4.24	0.24
JH	394.48	380.81	1.04	5,610	5,416	1.04	1,104	1,018	1.08	5.08	5.32	0.96
Keccak	285.23	282.73	1.01	6,845	12,817	0.53	1,257	1,272	0.99	5.45	10.08	0.54
Luffa	240.33	340.72	0.71	7,691	9,692	0.79	1,960	949	2.07	3.92	10.21	0.38
Shabal	214.92	214.92	1.00	1,719	1,719	1.00	283	283	1.00	6.08	6.08	1.00
SHAvite-3	213.80	235.10	0.91	3,841	3,253	1.18	2,090	1,076	1.94	1.84	3.02	0.61
SIMD	43.40	54.90	0.79	4,938	3,123	1.58	19,639	8,922	2.20	0.25	0.35	0.72
Skein	119.09	117.95	1.01	3,209	3,178	1.01	1,716	1,621	1.06	1.87	1.96	0.95
SHA-2	183.70	190.90	0.96	2,322	1,504	1.54	838	418	2.00	2.77	3.60	0.77

Table A.4: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Xilinx Virtex 6 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	103.47	136.24	0.76	3654	3322	1.10	2658	1275	2.08	1.37	2.61	0.53
BMW	N/A	15.21	N/A	N/A	7788	N/A	N/A	3081	N/A	N/A	2.53	N/A
CubeHash	317.36	328.19	0.97	5078	5251	0.97	723	655	1.10	7.02	8.02	0.88
ECHO	158.40	224.67	0.71	5232	13804	0.38	3774	5209	0.72	1.39	2.65	0.52
Fugue	234.69	272.41	0.86	1877	4358	0.43	901	726	1.24	2.08	6.00	0.35
Groestl	312.01	347.34	0.90	11017	8468	1.30	2914	1392	2.09	3.78	6.08	0.62
Hamsi	185.12	298.60	0.62	1975	3185	0.62	1513	588	2.57	1.31	5.42	0.24
JH	412.54	415.46	0.99	5867	5909	0.99	1075	959	1.12	5.46	6.16	0.89
Keccak	338.18	286.21	1.18	8116	12975	0.63	1185	1207	0.98	6.85	10.75	0.64
Luffa	195.73	322.17	0.61	6263	9164	0.68	1843	1091	1.69	3.40	8.40	0.40
Shabal	300.66	300.66	1.00	2405	2405	1.00	251	251	1.00	9.58	9.58	1.00
SHAvite-3	180.21	178.89	1.01	3237	2475	1.31	1456	873	1.67	2.22	2.84	0.78
SIMD	46.30	61.35	0.75	5268	3490	1.51	16754	7960	2.10	0.31	0.44	0.72
Skein	138.52	132.31	1.05	3733	3565	1.05	1355	1243	1.09	2.75	2.87	0.96
SHA-2	238.83	270.56	0.88	3019	2131	1.42	538	335	1.61	5.61	6.36	0.88

Table A.5: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Cyclone II FPGAs

	Max. Clk Freq. [MHz]			Throughput [Mbit/s]			Area [LEs]			Throughput/Area		
	512	256	ratio	512	256	ratio	512	256	ratio	512	256	ratio
BLAKE	34.76	53.67	0.65	1362	1309	1.04	13832	7001	1.98	0.10	0.19	0.53
BMW	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
CubeHash	98.74	114.05	0.87	1760	1825	0.96	3537	3276	1.08	0.50	0.56	0.89
ECHO	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Fugue	74.58	95.63	0.78	733	1530	0.48	7518	6057	1.24	0.10	0.25	0.39
Groestl	91.64	131.08	0.70	4402	3196	1.38	16229	5849	2.77	0.27	0.55	0.50
Hamsi	73.21	146.43	0.50	873	1562	0.56	7893	3371	2.34	0.11	0.46	0.24
JH	123.21	172.92	0.71	2456	2459	1.00	6756	6420	1.05	0.36	0.38	0.95
Keccak	114.19	134.43	0.85	3239	6094	0.53	5658	6419	0.88	0.57	0.95	0.60
Luffa	94.51	171.59	0.55	4515	4881	0.92	9873	4330	2.28	0.46	1.13	0.41
Shabal	93.63	144.97	0.65	1160	1160	1.00	827	827	1.00	1.40	1.40	1.00
SHAvite-3	71.71	93.86	0.76	1443	1299	1.11	20346	10134	2.01	0.07	0.13	0.55
SIMD	N/A	23.02	N/A	N/A	1310	N/A	N/A	26026	N/A	N/A	0.05	N/A
Skein	37.46	46.35	0.81	1246	1249	1.00	6798	6509	1.04	0.18	0.19	0.95
SHA-2	67.68	108.68	0.62	1092	856	1.28	3307	1686	1.96	0.33	0.51	0.65

Table A.6: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Cyclone III FPGAs

	Max. Clk Freq. [MHz]			Throughput [Mbit/s]			Area [LEs]			Throughput/Area		
	512	256	ratio	512	256	ratio	512	256	ratio	512	256	ratio
BLAKE	45.77	61.52	0.74	1616	1500	1.08	13932	7121	1.96	0.12	0.21	0.55
BMW	N/A	6.79	N/A	N/A	3476	N/A	N/A	19562	N/A	N/A	0.18	N/A
CubeHash	137.46	137.70	1.00	2199	2203	1.00	3653	3345	1.09	0.60	0.66	0.91
ECHO	110.44	116.59	0.95	3648	7163	0.51	69916	69423	1.01	0.05	0.10	0.51
Fugue	104.64	111.22	0.94	837	1780	0.47	7523	6059	1.24	0.11	0.29	0.38
Groestl	127.10	144.38	0.88	4488	3520	1.27	19389	5907	3.28	0.23	0.60	0.39
Hamsi	114.13	171.59	0.67	1217	1830	0.67	7932	3388	2.34	0.15	0.54	0.28
JH	220.85	227.07	0.97	3141	3229	0.97	6825	6481	1.05	0.46	0.50	0.92
Keccak	165.32	155.57	1.06	3968	7053	0.56	5680	6823	0.83	0.70	1.03	0.68
Luffa	167.03	199.20	0.84	5345	5666	0.94	10055	4482	2.24	0.53	1.26	0.42
Shabal	183.82	183.82	1.00	1471	1471	1.00	1049	1049	1.00	1.40	1.40	1.00
SHAvite-3	95.85	110.24	0.87	1722	1525	1.13	20326	10145	2.00	0.08	0.15	0.56
SIMD	22.04	26.37	0.84	2508	1500	1.67	54917	26171	2.10	0.05	0.06	0.80
Skein	54.90	54.90	1.00	1479	1479	1.00	6886	6667	1.03	0.21	0.22	0.97
SHA-2	94.86	118.75	0.80	1199	935	1.28	3315	1685	1.97	0.36	0.56	0.65

Table A.7: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Cyclone IV FPGAs

	Max. Clk Freq. [MHz]			Throughput [Mbit/s]			Area [LEs]			Throughput/Area		
	512	256	ratio	512	256	ratio	512	256	ratio	512	256	ratio
BLAKE	47.06	61.70	0.76	1662	1504	1.10	14138	6977	2.03	0.12	0.22	0.55
BMW	N/A	7.06	N/A	N/A	3615	N/A	N/A	19561	N/A	N/A	0.18	N/A
CubeHash	137.70	137.50	1.00	2203	2200	1.00	3550	3270	1.09	0.62	0.67	0.92
ECHO	107.71	113.43	0.95	3558	6969	0.51	69882	69269	1.01	0.05	0.10	0.51
Fugue	104.20	109.03	0.96	834	1744	0.48	7517	6033	1.25	0.11	0.29	0.38
Groestl	139.12	149.57	0.93	4912	3647	1.35	13183	6195	2.13	0.37	0.59	0.63
Hamsi	121.71	170.85	0.71	1298	1822	0.71	7920	3375	2.35	0.16	0.54	0.30
JH	224.37	225.38	1.00	3191	3205	1.00	6817	6496	1.05	0.47	0.49	0.95
Keccak	150.35	155.64	0.97	3608	7056	0.51	5696	6469	0.88	0.63	1.09	0.58
Luffa	172.35	199.24	0.87	5515	5667	0.97	10056	4362	2.31	0.55	1.30	0.42
Shabal	194.06	194.06	1.00	1552	1552	1.00	1048	1048	1.00	1.48	1.48	1.00
SHAvite-3	96.38	110.64	0.87	1731	1531	1.13	20340	10154	2.00	0.09	0.15	0.56
SIMD	22.74	26.77	0.85	2587	1523	1.70	54786	26157	2.09	0.05	0.06	0.81
Skein	55.98	57.13	0.98	1509	1540	0.98	7001	6741	1.04	0.22	0.23	0.94
SHA-2	99.75	126.50	0.79	1261	996	1.27	3296	1681	1.96	0.38	0.59	0.65

Table A.8: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Stratix II 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	71.27	100.67	0.71	2517	2454	1.03	7087	3635	1.95	0.36	0.68	0.53
BMW	N/A	10.35	N/A	N/A	5299	N/A	N/A	12636	N/A	N/A	0.42	N/A
CubeHash	158.60	177.21	0.89	2538	2835	0.89	1933	1929	1.00	1.31	1.47	0.89
ECHO	173.73	161.50	1.08	5739	9923	0.58	21206	20734	1.02	0.27	0.48	0.57
Fugue	137.70	142.84	0.96	1102	2285	0.48	2856	2398	1.19	0.39	0.95	0.40
Groestl	175.90	176.24	1.00	6211	4297	1.45	7121	3686	1.93	0.87	1.17	0.75
Hamsi	132.64	191.06	0.69	1415	2038	0.69	6507	2315	2.81	0.22	0.88	0.25
JH	294.81	294.90	1.00	4193	4194	1.00	3697	3516	1.05	1.13	1.19	0.95
Keccak	204.21	210.17	0.97	4901	9528	0.51	3990	4245	0.94	1.23	2.24	0.55
Luffa	189.74	215.80	0.88	6072	6138	0.99	6974	3078	2.27	0.87	1.99	0.44
Shabal	91.84	91.84	1.00	735	735	1.00	1749	1749	1.00	0.42	0.42	1.00
SHAvite-3	142.78	162.89	0.88	2565	2254	1.14	5625	3031	1.86	0.46	0.74	0.61
SIMD	34.33	42.62	0.81	3906	2425	1.61	53750	25798	2.08	0.07	0.09	0.77
Skein	70.55	73.94	0.95	1901	1992	0.95	4812	4654	1.03	0.40	0.43	0.92
SHA-2	123.03	150.31	0.82	1555	1184	1.31	2033	981	2.07	0.77	1.21	0.63

Table A.9: 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	89.53	118.98	0.75	3,161	2,901	1.09	7,086	3,635	1.95	0.45	0.80	0.56
BMW	N/A	12.38	N/A	N/A	6,339	N/A	N/A	12,619	N/A	N/A	0.50	N/A
CubeHash	204.21	232.88	0.88	3,267	3,726	0.88	1,930	1,922	1.00	1.69	1.94	0.87
ECHO	247.40	233.32	1.06	8,172	14,335	0.57	21,187	20,723	1.02	0.39	0.69	0.56
Fugue	199.80	207.43	0.96	1,598	3,319	0.48	2,783	2,397	1.16	0.57	1.38	0.41
Groestl	202.27	220.65	0.92	7,142	5,380	1.33	12,355	6,350	1.95	0.58	0.85	0.68
Hamsi	187.55	280.98	0.67	2,001	2,997	0.67	6,401	2,308	2.77	0.31	1.30	0.24
JH	390.63	387.75	1.01	5,556	5,515	1.01	3,709	3,525	1.05	1.50	1.56	0.96
Keccak	304.60	273.37	1.11	7,310	12,393	0.59	3,979	4,213	0.94	1.84	2.94	0.62
Luffa	268.10	301.30	0.89	8,579	8,570	1.00	6,891	3,032	2.27	1.24	2.83	0.44
Shabal	109.66	109.66	1.00	877	877	1.00	1,744	1,744	1.00	0.50	0.50	1.00
SHAvite-3	226.60	245.52	0.92	4,071	3,397	1.20	5,619	3,042	1.85	0.72	1.12	0.65
SIMD	49.82	54.89	0.91	5,668	3,123	1.82	53,623	25,728	2.08	0.11	0.12	0.87
Skein	90.34	92.87	0.97	2,434	2,503	0.97	4,794	4,645	1.03	0.51	0.54	0.94
SHA-2	162.42	205.09	0.79	2,053	1,615	1.27	2,164	1,049	2.06	0.95	1.54	0.62

Table A.10: Major performance measures of SHA-3 candidates (512-bit and 256-bit variants) when implemented in Altera Stratix IV 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	106.07	126.23	0.84	3745	3078	1.22	7084	3635	1.95	0.53	0.85	0.62
BMW	N/A	12.08	N/A	10127	6185	1.64	24882	12614	1.97	0.41	0.49	0.83
CubeHash	220.70	250.06	0.88	3531	4001	0.88	1936	1925	1.01	1.82	2.08	0.88
ECHO	255.95	228.73	1.12	8455	14053	0.60	21191	20781	1.02	0.40	0.68	0.59
Fugue	65.64	125.99	0.52	525	2016	0.26	6505	3606	1.80	0.08	0.56	0.14
Groestl	214.45	205.76	1.04	7572	5017	1.51	12416	6243	1.99	0.61	0.80	0.76
Hamsi	177.34	274.73	0.65	1892	2930	0.65	6391	2292	2.79	0.30	1.28	0.23
JH	388.65	410.17	0.95	5527	5834	0.95	3723	3533	1.05	1.48	1.65	0.90
Keccak	285.55	286.04	1.00	6853	12967	0.53	3983	4219	0.94	1.72	3.07	0.56
Luffa	264.62	316.16	0.84	8468	8993	0.94	6923	3034	2.28	1.22	2.96	0.41
Shabal	111.74	111.74	1.00	894	894	1.00	1752	1752	1.00	0.51	0.51	1.00
SHAvite-3	216.40	231.05	0.94	3888	3197	1.22	5600	2929	1.91	0.69	1.09	0.64
SIMD	51.93	57.77	0.90	5908	3286	1.80	53388	25736	2.07	0.11	0.13	0.87
Skein	99.09	98.82	1.00	2670	2663	1.00	4793	4645	1.03	0.56	0.57	0.97
SHA-2	188.71	217.63	0.87	2386	1714	1.39	2165	1052	2.06	1.10	1.63	0.68

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. Chodowicz, “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] K. Gaj, E. Homsirikamol, and M. Rogawski, “Fair and comprehensive methodology for comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs,” in *Cryptographic Hardware and Embedded Systems - CHES 2010*, vol. 6225 of *LNCS*, pp. 264–278, Springer, Aug. 2010.
- [14] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. K. Gurkaynak, “Developing a hardware evaluation method for SHA-3 candidates,” in *Cryptographic Hardware and Embedded Systems - CHES 2010*, vol. 6225 of *LNCS*, pp. 248–263, Springer, Aug. 2010.
- [15] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O'Neill, and W. P. Mar-nane, “FPGA implementations of the round two SHA-3 candidates,” in *Second SHA-3 Candidate Conference*, Aug. 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/BALDWIN_FPGA_SHA3.pdf.
- [16] K. Gaj, E. Homsirikamol, and M. Rogawski, “Comprehensive comparison of hardware performance of fourteen round 2 SHA-3 candidates with 512-bit outputs using Field Programmable Gate Arrays,” in *Second SHA-3 Candidate Conference*, Aug. 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/GAJ_SHA3_512.pdf.
- [17] X. Guo, S. Huang, L. Nazhandali, and P. Schaumont, “Fair and comprehensive performance evaluation of 14 second round SHA-3 ASIC implementations,” in *Second SHA-3 Candidate Conference*, Aug. 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/SCHAUMONT_SHA3.pdf.
- [18] S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama, and K. Ota, “How can we conduct “fair and consistent” hardware evaluation for SHA-3 candidate?,” in *Second SHA-3 Candidate Conference*, Aug. 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/MATSUO_SHA-3_Criteria_Hardware_revised.pdf.
- [19] “ECRYPT Benchmarking of Cryptographic Systems.” <http://bench.cr.yp.to>.

- [20] “Hardware Interface of a Secure Hash Algorithm (SHA).” <http://cryptography.gmu.edu/athena/index.php?id=interfaces>.
- [21] “ANSI C cryptographic API profile for SHA-3 candidate submissions.” http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html.
- [22] 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.
- [23] J. van Lint, *Introduction to Coding Theory*, vol. 86 of *Graduate Texts in Mathematics*. Springer, second ed., 1992.
- [24] K. Gaj and P. Chodowicz, *Cryptographic Engineering*, ch. Chapter 10, FPGA and ASIC Implementations of AES, pp. 235–294. Springer, 2009.
- [25] J.-P. Deschamps, G. J. Bioul, and G. D. Sutter, *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. Wiley-Interscience, 2006.
- [26] “ATHENa project website.” Online, 2010. <http://cryptography.gmu.edu/athena/>.
- [27] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, “SHA-3 proposal BLAKE.” Submission to NIST, 2008.
- [28] 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.
- [29] D. J. Bernstein, “CubeHash specification (2.b.1).” Submission to NIST (Round 2), 2009.
- [30] 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.
- [31] S. Halevi, W. E. Hall, and C. S. Jutla, “The hash function Fugue.” Submission to NIST (updated), 2009.
- [32] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlf-fer, and S. S. Thomsen, “Grøstl – a SHA-3 candidate.” Submission to NIST, 2008.
- [33] O. Kucuk, “The hash function Hamsi.” Submission to NIST (updated), 2009.
- [34] H. Wu, “The hash function JH.” Submission to NIST (updated), 2009.
- [35] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, “Keccak specifications.” Submission to NIST (Round 2), 2009.

- [36] C. D. Canniere, H. Sato, and D. Watanabe, “Hash function Luffa: Specification.” Submission to NIST (Round 2), 2009.
- [37] “Secure Hash Standard (SHS) FIPS publication 180-3,” Oct 2008.
- [38] 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.
- [39] E. Biham and O. Dunkelman, “The SHAvite-3 hash function.” Submission to NIST (Round 2), 2009.
- [40] G. Leurent, C. Boullaguet, and P.-A. Fouque, “SIMD is a message digest.” Submission to NIST (Round 2), 2009.
- [41] 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.
- [42] 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>.
- [43] J. Daemen and V. Rijmen, “AES proposal: Rijndael,” in *First Advanced Encryption Standard AES Conference*, (Ventura, California, USA), 1998. updated version from 1999.
- [44] 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.
- [45] 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/>.
- [46] 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.
- [47] X. Guo, S. Huang, L. Nazhandali, and P. Schaumont, “On the impact of target technology in SHA-3 hardware benchmark rankings.” Cryptology ePrint Archive, Report 2010/536, 2010. <http://eprint.iacr.org/>.
- [48] J. Detrey, P. Gaudry, and K. Khalfallah, “A low-area yet performant FPGA implementation of Shabal,” in *17th International Workshop on Selected Areas in Cryptography*

(*SAC'10*), Lecture Notes in Computer Science, (Waterloo, Canada), Springer-Verlag, Aug. 2010.

- [49] 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>.