Design of Multi-core Dataflow Cryptprocessor

by

Ali Saeed Alzahrani
B.Sc., Umm Alqura University, 2010
M.Sc., University of Victoria, 2015

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Design of Multi-core Dataflow Cryptprocessor

by

Ali Saeed Alzahrani
B.Sc., Umm Alqura University, 2010
M.Sc., University of Victoria, 2015

Supervisory Committee

Dr. Fayez Gebali, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Atef Ibrahim, Co-supervisor
(Department of Electrical and Computer Engineering)

Dr. Phalguni Mukhopadhyaya, Outside Member
(Department of Civil Engineering)

Supervisory Committee

Dr. Fayez Gebali, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Atef Ibrahim, Co-supervisor
(Department of Electrical and Computer Engineering)

Dr. Phalguni Mukhopadhyaya, Outside Member
(Department of Civil Engineering)

ABSTRACT

Embedded multi-core systems are implemented as systems-on-chip that rely on packet store-and-forward networks-on-chip for communications. These systems do not use buses nor global clock. Instead routers are used to move data between the cores and each core uses its own local clock. This implies concurrent asynchronous computing. Implementing algorithms in such systems is very much facilitated using dataflow concepts. In this work we propose a methodology for implementing algorithms on dataflow platforms. The methodology can be applied to multi-threaded, multi-core platforms or a combination of these platforms as well. This methodology is based on a novel dataflow graph representation of the algorithm.

We applied the proposed methodology to obtain a novel dataflow multi-core computing model for the secure hash algorithm-3. The resulting hardware was implemented in FPGA to verify the performance parameters. The proposed model of computation has advantages such as flexible I/O timing in term of scheduling policy, execution of tasks as soon as possible, and self timed event driven system. In other words, I/O timing and correctness of algorithm evaluation are dissociated in

this work. The main advantage of this proposal is ability to dynamically obfuscate algorithm evaluation to thwart side-channel attacks without having to redesign the system. This has important implications for cryptographic applications.

Also the dissertation proposes four countermeasure techniques against side channel attacks for SHA-3 hashing. The countermeasure techniques are based on choosing stochastic or deterministic input data scheduling strategies. Extensive simulations of the SHA-3 algorithm and the proposed countermeasures approaches were performed using object-oriented MATLAB models to verify and validate the effectiveness of the techniques. The design immunity for the proposed countermeasures is assessed.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

In the name of Allah, the Most Gracious and the Most Merciful

Alhamdulillah, all praises belongs to Allah the merciful for his blessing and guidance. He gave me the strength to reach what I desire. I would like to thank:

My parents, my family, for supporting me at all stages of my education and their unconditional love.

My Supervisor, Dr. Fayez Gebali, for all the support, encouragement, and encouragement he provided to me during my work under his supervision. It would not have been possible to finish my research without his invaluable help of constructive comments and suggestions.

My Committee, Dr. Atef Ibrahim, Dr. Phalguni Mukhopadhyaya, for their precious time and valuable suggestions for the work done in this dissertation.

Ali Alzahrani

DEDICATION

To my parents, **Saeed Alzahrani and Jumah Alzahrani** for their love, prayers, and encouragement.

To my lovely wife, **Reem Alzahrani** for always standing by me, and believing in me.

To my beautiful daughter and son **Aryam, and Muhammad**.

Chapter 1

Introduction

1.1 Motivation

Embedded multi-core systems are implemented as systems-on-chip (SoC) that rely on packet store-and-forward networks-on-chip (NoC) for communications [1] [27]. These systems do not use buses nor global clock. Instead routers are used to move data between the cores and each core uses its own local clock. This implies concurrent asynchronous computing. Implementing algorithms in such systems is very much facilitated using dataflow concepts. Some of these systems operate in Globally Asynchronous Locally Synchronous (GALS) mode [41].

Cryptographic applications running on the high-performance platforms include Secure Hash Algorithm-3 (SHA-3) and Advanced Encryption Standard (AES). Parallel implementations of these algorithms are cumbersome when using the classic control-flow; von Neumann processors. On the other hand, dataflow processing is more naturally suited to parallelize such algorithms [49].

Design for security is mandatory for cryptographic processors to provide immunity to attacks especially side-channel attacks [62]. Countermeasures employed for the classic control-flow processors included inserting dummy instructions, randomizing instruction set execution, clock randomization, and power consumption randomization. These countermeasures techniques require extra computing resources area, power, and time. The main advantage of using dataflow processing is the ability to frustrate side-channel attacks by randomizing the order of execution of the algo-

rithm tasks without requiring any modifications in the software or hardware of the cryptoprocessor.

1.2 Dissertation Contributions

1. Develop a new dataflow graph (DFG) description of an algorithm.
2. Define a novel three-step methodology to obtain a dataflow computing architecture of single- and multi-core systems.
3. Apply the DFG design methodology to some cryptographic algorithms such as SHA-3.
4. Develop obfuscation techniques through input data scheduling at the start of each iteration to counter side channel attacks.
5. Verify the correctness of the computation models through developing object-oriented programming using Matlab and generate an FPGA hardware implementation to validate the design and extract physical performance parameters.

1.3 Agenda

This section presents a map of the dissertation and a short summery of each chapter.

Chapter 1 presents the problem considered and the contributions of the dissertation.

Chapter 2 describes traditional approaches of parallel computer structures and parallel algorithm methods that been reported in the literature. Moreover, we explore the related topics to this work including dataflow computing, side channel attacks, and the cryptographic algorithm; SHA-3.

Chapter 3 gives a formal description of the new dataflow graph (DFG) scheme and a novel methodology to obtain a dataflow multi-core computing (DMC) architecture for a given algorithm.

Chapter 4 is where we applied the proposed methodology to obtain a novel DMC architecture for the secure hash algorithm-3 (SHA-3).

Chapter 5 discusses data access approaches and how to take advantage of dataflow computing platform. It also presents our object-oriented simulation and implementation results of the SHA-3 algorithm and highlight the importance of the outcomes.

Chapter 6 contains a summery of the dissertation contributions. It also enumerates avenues of future work for further.

Chapter 2

Background and Previous Work Review

The scope of this thesis is multidisciplinary in nature, in the sense it covers multiple topics such as parallel computing, parallel algorithm, dataflow computing, cryptographic algorithms, side channel attacks. Parallel computing can broadly be divided into parallel computing using von Neumann, and parallel computing using dataflow. Dataflow computing models provides a suitable alternative of von Neumann architecture models. Dataflow is more naturally suited to parallelize algorithm and frustrate side channel attacks.

2.1 Parallel computing and parallel algorithm

Typically, the traditional approach for parallel computing in the literature so far has been using control flow processors. All the methods that have been proposed have been targeting von Neumann machines. Parallel algorithms and parallel architecture are firmly tight together. Researchers proposed numerous methods to parallelize algorithms in different levels yet; we must consider the parallel hardware that will support it. Conversely, we also have to consider the parallel software that supports the parallel hardware. Increasing hardware resources utilization by exploiting parallelism can be implemented at different levels in a computing system [26]:

1. Data-level parallelism (DLP): Simultaneously apply a single operation on multi-

ple bits, independent data element. Examples of this are bit-parallel arithmetic operations of binary numbers, vector processor, and systolic arrays.

2. Instruction-level parallelism (ILP): Processors try to execute several instructions at the same time. Examples of this are the use of instruction pipelining, and superscalar execution.

3. Thread-level parallelism (TLP): A thread is a piece of a program that shares processor resources with other threads. TLP is applied by executing parallel software threads on a single processor or separate processing units.

4. Process-level parallelism: A process is a set of instructions that is running on a computer. A process allocates its required computing resources include: memory space and register. This level of parallelism is the classic multitasking computing where multiple tasks are executing simultaneously on one or more processing machines.

Flynn [25] introduced the most known classification of parallel computer systems. The four classifications are based on the number of concurrent instructions and the data streams. Flynn taxonomy is as follow:

1. Single instruction single data stream (SISD): This is the case of a sequential computer. Examples of SISD architecture are the uniprocessor machines.

2. Single instruction multiple data stream (SIMD): All processors perform the same operation on multiple data streams. The graphics processing unit (GPU) and video compression are applications of such category.

3. Multiple instruction single data stream (MISD): Multiple instructions operate on a single data stream. Examples of such approach are neural networks and dataflow machines.

4. Multiple instruction multiple data stream (MIMD): Multiple processors concurrently execute different instructions on the local data stream. Multi-core processors and multi-threaded multiprocessors belong to this category.

An alternative technique to parallelize applications is concurrency platform software tools that allow coordination, scheduling, and management of multi-core systems. Examples of concurrency platforms include clik++ [4], Open Multi-Processing (OpenMP) [55], or compute unified device architecture (CUDA) [54]. These tools allow the program developer to control the number of threads and workload distributed among threads. The above tools rely on the programmer or the developer ability to exploit parallelism and ensure proper program sequencing.

Moreover, several ad hoc techniques are used to implement parallel algorithms on parallel computers. Such techniques tackle what is called embarrassingly parallel algorithm [68] or trivial parallel algorithm [43]. Algorithms with complex data dependencies cannot be dealt with efficiently using these techniques. Loop spreading or unrolling, problem partitioning, and Divide-and-conquer are examples of ad-hoc techniques [26].

Gebali [26] introduced a simple yet powerful technique for regular iterative algorithms (RIAs). The technique discusses constructing the dependence graph of an iterative algorithm. The dependency graph will facilitate us schedule tasks, which will translate to software thread or hardware systolic processing elements. Gebali also discussed a formal, powerful, and simple approach for extracting parallelism from an nonserial-parallel algorithms(NSPAs) that cannot be characterized as serial, parallel, or serial-parallel algorithms. This method is suitable for general algorithms that are not parallel or exhibit a disorienting task dependence pattern.

2.2 Dataflow

The dataflow graph computation model is a radical alternative to the control flow; von Neumann computing model because all computation is data-driven. Dataflow model provides a powerful mechanism to explore possible parallel processing since it has no program counter nor global updateable memory. These two features of the classic von Neumann model that become barriers of parallelism.

The initial concepts of dataflow-like model were originated by Karp and Miller [33]. They introduced a graph-theoretic model for the description and analysis of parallel computations. In the early 1970s, two different dataflow models emerged.

The first dataflow graph model was developed by Dennis [18], his work was originally applied to computer architecture design. The dataflow graphs evolved rapidly which led to the first dataflow computing machain by Dennis and Misunas [19]. Kahn [28], developed the second dataflow model, concurrency theorist used Kahn work for modeling concurrent software. Based on these models, many areas of computer research has had been influenced by dataflow such as in programming language, processor design, reconfigurable computing, graphics processing, high-level logic design and digital signal processing.

A function can be represented as a dataflow graph which is a directed graph. The dataflow graph consists of two elements nodes and arcs. Nodes represents instructions and arcs represents data dependences among instructions [17] [34]. A node could be a single instruction or a series of instructions. In a packet format data are propagate along the arcs, called token. Two important features of dataflow graph: functionality and composability. Functionality implies that the outcomes of the execution of a given graph is equivalent to execution of the corresponding mathematical functions on the same given input value. Composability implies that multiple graphs can be combined to form a new graph [61].

Dataflow model is data-driven execution, the execution of instructions depends on the availability of the input data. A set of instructions of a dataflow graph are executed according to the instruction enabling and firing rules. Exactly, the enabling rule states that an instruction is enabled when all its associated operands are available to it. The firing rule states that an instruction is executed when it is enabled and all required hardware resources are available. While executing a graph, many instructions may become enabled to fire simultaneously, thus this simple fundamental would provides an opportunity to exploit massive parallel computations in several computing levels. Also, it is a self-schedule model since instruction sequencing is driven by data dependences among instructions.

Many *pure dataflow* computer architectures have emerged in the past, based on the way the data flow among graph entities. An implementation of a pure dataflow architecture can be classified as *static*, *dynamic* and *explicit token store*. The static (also called single-token-per-arc) dataflow architecture approach allows at most one token to reside on any one arcs. This is attained by modifying the original firing role

as follows: a node is executed when it is enabled and there is no token on any of its output arcs, and all required hardware resources are available [20]. The dynamic (also called multiple-tagged-token-per-arc) dataflow architecture tries to overcome a number of serious deficiencies of static dataflow. The performance of a dataflow machine increases by allowing the execution of the same instruction multiple times as a separate instance. Thus, achieving parallel invocations of loop iterations and subprogram. The explicit token store (ETS) dataflow architecture has been introduced to reduce the execution time overhead of matching tokens of tagged-token model [15].

Recently, dataflow computing is gaining some interest as an alternative way of computing. It gains it is importance for systems that are not tightly synchronous. Different authors approached the problem from different perspectives and goals. Some people works in developing programming languages, cloud computing, and parallel computing. This work is more towered parallel algorithm and security aspects using dataflow concepts.

2.3 Comparing Control-Flow vs. Dataflow Processing

In the classic control-flow; von Neumann processing, the data flows across buses. Any other information, such as its type or identity, is inferred in the design itself such as control and address buses or registers. The validity of the data is implied upon the arrival of a clock edge. The operations carried out on the data are specified in the instruction register and control signals. In dataflow processing, data and operation are combined together in a packet and no clock is necessary to synchronize the system components. The inclusion of token in the packet indicates that data is valid and ready to be processed.

In control-flow processing, the processors are always active as long as there is a clock and it is very difficult to detect when the processor is idle. In dataflow processing, the processor is idle by default until a packet arrives. Dataflow processing is more suitable to green computing because it prevents unnecessary computations.

In a traditional von Neumann processing, changes to I/O timing or inter-module

synchronization require complete redesign of hardware and software. This point can be illustrated by understanding the role of a system-wide clock in a traditional systems. The presence of a clock edge implies two things:

1. The **presence** of the edge indicates that the data is **valid**.
2. The **location** of the clock edge along the time axis indicates the **identity** of the data (i.e. which data sample is it).

Parallel implementations of algorithms are difficult and error prone when using the control flow processing. In dataflow processing as soon as any packet arrives with a token, processing could commence. The contents of the packet indicates the data and the operations to be done. Thus data dependencies are included in the packet and correct processing is guaranteed.

In control-flow processing, words must be propagated through the system in a predetermined sequence which makes the platform vulnerable to side-channel attacks. In dataflow processing, the randomization of the order of execution of the algorithm tasks by randomizing the order of feeding the incoming message packets will thwart such attacks.

A packet transmission mode that replaces system buses and lack of a system-wide clock results in a concurrent asynchronous computing are unique features of systems-on-chip using networks-on-chip for communication. Thus, dataflow processing is a natural extension to such systems. Table 5.1 summarize the comparison between the control flow and DFG processing.

2.4 Side channel attacks and countermeasures

Cryptographic algorithms provide crucial security services for computing systems. Data integrity, confidentiality, availability, and authenticity are what users expect from such algorithms. Cryptographic algorithms are targets to various methods of code breaking by cryptanalyst. The cryptanalysis methods could be classified into three categories [56]: classical cryptanalysis, implementation cryptanalysis (Side Channel Attacks), and social engineering attacks. Since the introduction of the SHA-3

Table 2.1: Comparing the DFG processing and the control flow (von Neumann)

Dataflow Processing	Control Flow Processing
Information is exchanged in the form of packets that contains the data and extra information.	Data is exchanged in the form of words that contain the actual data.
ID of data is specified in the packet.	ID of data is implied in the timing of the system.
Data is valid when packet contains a token.	Data is valid on the clock edge by assumption.
Order of processing and input timing are flexible.	Order of processing and input timing are pre-specified.
Changes in the scheduling (arrival and departure) does not require changes in the hardware or the software.	Changes in the scheduling require the modification of the whole hardware and the whole software.
Immune against side-channel attacks.	Vulnerable to side-channel attacks, countermeasures must be added to design.
Self-timed event-triggered system, no synchronization is required.	The system must be synchronized with a global clock.
Processors are in idle mode unless there is an event.	Processors are always active as long as there is a clock.

algorithm, various attacks targeting the mathematical structure (first category cryptanalysis methods) are presented in the literature [22], [16], [21], [52], [53], and [12]. The focus of this work is study of immunity of the SHA-3 algorithm to the side channel attacks.

Beside the efforts by cryptographers seeking a new cryptographic algorithms with high level of complexity that thwart cryptanalysis attacks, the implementation of the cryptosystem must be considered. Using a well concealed standard cryptographic algorithm is not sufficient to achieve security. A direct implementation of a cryptosystem could be subject to many cyber attacks that lead to leakage of sensitive information [38].

These attacks target the electrical activities of the device that implemented the cryptosystem instead of analyzing the mathematical structure and properties. Investigating electrical activities of a cryptosystem could reveal side channel information to the attacker such as power consumption, execution time, etc. The examination of the collected data eventually will unveil the valuable information such as secret key, or plaintext.

2.4.1 Side Channel Attacks (SCA)

Following are examples of SCAs categorized based on the type of side channel information investigated by attackers.

Timing attacks

Kocher *et al.* [39] introduced the first SCA attack which was timing related. This attack is based on measuring the execution time needed to complete different cryptographic operations.

Power Analysis attacks

The first power analysis attack was introduced in 1998 [38]. The power analysis attack is based on the observation of power consumptions by the device which is varies depending on the processed data and performed operations to retrieve valuable information. Power analysis attacks have multiple types, the main two are Simple Power

Analysis (SPA) and Differential Power Analysis (DPA). Software implementations of the SHA-3 algorithm were a target of power analysis attacks [64] and [63].

Electromagnetic Analysis attacks

Electromagnetic Analysis (EMA) attacks based on exploiting leaked electromagnetic fields due to current flows [58] and [2]. The first EMA was proposed by Quisquater and Samyde [58], which was inspired by the work of Kocher with timing and power measurements. There are two types of EMA attacks: Simple Electromagnetic Analysis (SEMA) and Differential Electromagnetic analysis (DEMA) attacks.

Fault Analysis Attacks

Intended and unintended faults of a cryptographic system are the core of fault analysis attacks (FAA). There are two required steps of a successful fault attack: Fault injection and fault exploitation. The first step, fault injection, where the faults could accidentally occur while computing the cryptographic algorithm or where an adversary insert a faulty input to the cryptosystem intentionally to generate faulty output. Simple Fault Analysis (SFA) and Differential Fault analysis (DFA) are the categories of fault analysis attacks [11] and [10]. Bagheri *et al.* presented the first differential fault analysis of the SHA-3. The works only focus on two of the six SHA-3 family hash functions (SHA3-512 and SHA3-384). It is the first step into a better understanding of the SHA-3 algorithm and the needs of applying adequate measurements to shield the implementation of the algorithm. More works releated to fault analysis were introduced in [44]

2.4.2 Countermeasures

Countermeasures against timing attacks include random delay and constant execution time [40] [45]. Many countermeasures have been proposed by researches to secure cryptosystem implementations against power analysis attacks. Double-and-add always and dummy instructions are examples of countermeasures against powers attacks [14] [50]. Countermeasures against EMA attacks include reduction of electromagnetic filed by metal, noise addition (masking) [42], or balancing power consumption [65]

and [66]. Error-detecting codes and spacial designed sensors are examples of countermeasures techniques to withstand fault analysis attacks [67] and [51]. In [46] and [44], presented a fault analysis countermeasures to secure the SHA-3 implementation.

2.5 Secure Hash Algorithm-3

This section is a brief review of the SHA-3 algorithm with the purpose of implementing it using a dataflow computer as per Section 4.1. Data integrity and authenticity is a crucial part of a secure system. Technology users become vulnerable to cyber attacks in many levels. Information integrity is a security concern for all involved parties. As a term, data integrity is used to describe the information accuracy and reliability. Exchanging a piece of information between two entities goes through many phases. This information could be altered in any phase such as processing, transforming or storing. Data alteration could be caused by malicious behaviour, system failure or errors by the user. To overcome these issues, cryptographers developed a data integrity vitrification mechanism. Cryptographic hash functions are developed to verify the data integrity. Secure Hash Algorithm-3 (SHA-3) is the latest verification algorithm. In 2004-2005 National Institute of Standard and Technology (NIST) held two hash workshops after cryptanalysis raise a series concerns about the security of the government approved hash function SHA-1. As a result of these workshops, NIST decides to build a new cryptographic hash algorithm for standardization. In 2007 NIST released a call for a new cryptographic hash algorithm SHA-3 family contest [35]. The competition runs from 2007-2012, in 2012 NIST announces the winner candidate who is Keccak [9].

In the SHA-3 (Keccak) [8] family there are four fixed hash functions and two expandable-output functions (XOFs). These six functions share a common structured function that is the sponge functions [6].

A hash function operates on a binary input and generates a fixed size output. The input to the hash function is called the *message* and the output called the *digest*. In a hash function, the digest is also called the hash value. The SHA-3 family consist of four hash functions SHA3-224, SHA3-256, SHA3-384, and SHA3-512. These four families represent the size of the output digest of the hash function, for instance,

the SHA3-384 output 384-bit hash value. The last two hash functions the XOFs are SHAKE128 and SHAKE256. The output hash of those functions is flexible to meet the desired length of the applications.

The SHA-3 hash functions are designed to provide resistance against collision, preimage, and second preimage attacks [23]. Hash functions are a crucial part in many information security applications, such as digital signature, key derivation, and pseudorandom bit generation. The SHA-3 functions perform the same permutation. In the SHA-3, the permutation serves as a mode of functions to provide some flexibility in term of security parameters and size for future development.

All six SHA-3 hash functions perform the same permutations. In fact, those functions are just a different mode of permutations to provide some flexibility for potential applications. The SHA-3 is based on a new cryptographic hash approach, sponge function family [6].

Two parameters are used in the KECCAK-p permutations. The first parameter is b which is the length of message in bits. The standard calls b the width of the permutation. The second parameter is R, which is the number of iterations, or rounds. The KECCAK-p permutation is denoted by KECCAK-p $[b, R]$. The bit strings b that are permuted form a *state*. The state has a fixed number of bits b. The state consists of two parts, *rate* λ and *capacity* c. The *rate* define the number of bits to be processed in each permutation block and the *capacity* is the remaining bits of the state. The width of the permutation is found by the summation of $\lambda + c$, which is restricted to predetermined seven values $\{25, 50, 100, 200, 400, 800, 1600\}$ [23]. In SHA-3 the desired size of the hash output, denoted by d, determines the values of λ and c. For instance, for 512 hash output: $b = 1600$ bits, $\lambda = 576$ bits, and $c = 1024$ bits, where $c = 2 \times d$ are selected. In SHA-3 the state consist of a maximum 1600 bits organized as $5 \times 5 \times w$ matrix, $w = 2^{\ell} = (b/25)$ and $\ell = \log_2(b/25)$. The seven possible values of these variables are predefined in the standard, Table 2.2 below shows all different values.

Table 2.2: KECCAK-p permutation variables

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

The SHA-3 inputs and outputs can be represented in two forms. The first form is to represent the data as a string S of b-bits indexed from 0 to $b-1$. The second form is to represent the data as a three-dimensional array $A[x, y, z]$ with three indices $0 \leq$ x,y < 5, and $0 \leq$ z $< w$. The mapping form S to A is given by:

$$A[x, y, z] = S[w(5y + x) + z] \tag{2.1}$$

Figure 2.1 shows a state matrix in three dimensions. The KECCAK also define a 2-D entity referred to as *plane*, and 1-D entity referred to as *lane* as in Fig. 2.2. The *plane* entity is a $5 \times w$ bits, and the *lane* entity is a w bits string.

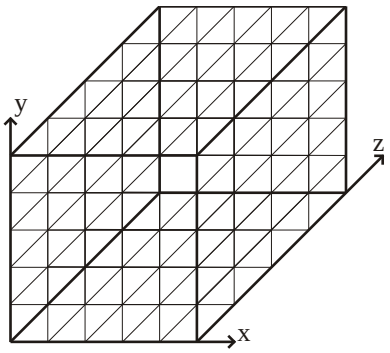
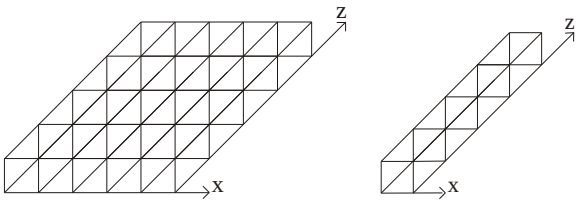


Figure 2.1: State of SHA-3



(a) plane (b) lane

Figure 2.2: Parts of state

2.6 SHA-3 Functions

The SHA-3 algorithm uses a function f, denoted by KECCAK-f, to process each input block. Each input block is processed once by the function f. The KECCAK-f function is iterative, and each iteration is referred to as a round. The function takes the input data block through rounds of processing. The width of the permutation b determines the total number of rounds R to be performed. $R = 12 + 2\ell$. Each round, denoted by r, updates the state matrix by five permutation or substitution operations. These five operations are denoted by $\theta, \rho, \pi, \chi, \iota$ and are explained in the following subsections.

2.6.1 Theta (θ) step

The input data block to the θ step is the original message to be hashed. It accepts a three-dimensional array $A[x, y, z]$ and returns an updated state $A'[x, y, z]$. This step implements three equations that perform a simple $XOR(\oplus)$ and a bitwise cyclic shift ROT operations. The value of a single bit of the state is updated by 11 input bits. For all pairs (x,z) such that $0 \leq$ x ≤ 4 and $0 \leq$ z ≤ 63, let

$$U[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \\ \oplus A[x, 3, z] \oplus A[x, 4, z]. \tag{2.2}$$

For all pairs (x,z) such that $0 \leq$ x ≤ 4 and $0 \leq$ z ≤ 63, let

$$V[x, z] = U[x - 1, z] \oplus ROT(U[x + 1, 1]) \tag{2.3}$$

where the addition and subtraction operations in the above equation are done modulo 5. This applies to all addition and subtraction operations in the succeeding functions in the following subsections.

For all triples (x,y,z) such that $0 \leq$ x, y ≤ 4 and $0 \leq$ z ≤ 63, the output of the θ step is given by:

$$A'[x, y, z] = A[x, y, z] \oplus V[x, z]. \tag{2.4}$$

2.6.2 Rho (ρ) step

The output state $A'[x, y, z]$ of the θ step is an input state to ρ step. The step equation performs a bitwise cyclic left shift ROT in all lanes. Note that the lane with coordinates $x, y = 0$ is unchanged. The amount of the a bitwise cyclic left shift is referred to as *offset* denoted by δ. For all pairs (x,y) such that $0 \leq$ x, y ≤ 4, the output of the ρ step is given by:

$$A'[x, y, z] = ROT(A[x, y, z], \delta[x, y]) \tag{2.5}$$

where the value of δ associated with the indices x and y can be found in Table 2.3.

Table 2.3: Offsets of ρ [7].

y = 4	92	276	231	136	78
y = 3	28	55	153	21	120
y = 2	190	6	171	15	253
y = 1	1	300	10	45	66
y = 0	0	36	3	105	210
	x = 0	x = 1	x = 2	x = 3	x = 4

2.6.3 Pi (π) step

The input state to the π step is the outputs of ρ step. All the lanes positions in the state are rearranged except the lane with coordinates $x, y = 0$. For all triples (x,y,z) such that $0 \leq$ x, y ≤ 4 and $0 \leq$ z ≤ 63, the output of the π step is given by:

$$B[y, (2x + 3y), z] = A[x, y, z]. \tag{2.6}$$

2.6.4 Chi (χ) step

The χ step accepts the output state A of the π step. Each bit of the lane is combined with neighbouring bits along the x-axis using AND, XOR, and NOT operations. For all triples (x,y,z) such that $0 \leq$ x, y ≤ 4 and $0 \leq$ z ≤ 63, the output of the χ step is given by:

$$A'[x,y,z] = B[x,y,z] \oplus (NOT(B[(x+1),y,z]) \\ AND(B[(x+2),y,z])). \tag{2.7}$$

2.6.5 Iota (ι) step

The output state of the χ step is input to the ι step. For all values of z such that 0 $\leq$ z ≤ 63, the output of the ι step is given by:

$$A'[0,0,z] = A[0,0,z] \oplus RC[z] \tag{2.8}$$

where RC is the round constant whose value changes for each round as explained in the standard document [23].

The ι step is the last step in the round, the output of the ι step is fed back as an input to the θ step until its reach the final round. The five steps mappings are repeated 24 times over the state matrix A, $r = 0 : R-1$. Figure 2.3 shows the 24 rounds of the KECCAK functions. The θ step is broken into three steps $\theta_{1a}, \theta_{1b}, \theta_{1c}$ based on the step equations (1), (2), and (3).

2.7 Implementations of SHA-3

Several states of the art hardware architectures were developed to implement the SHA-3 algorithm [32, 59, 30, 31, 36, 69, 57, 48, 47, 29, 60, 5]. Hardware implementations are designed on Application Specific Integrated Circuits (ASICs) or Field-Programmable Gate Array (FPGA) to get a real-time results. FPGA based designs are preferable since their performance is approaching that of ASICs but more flexible and less costly.

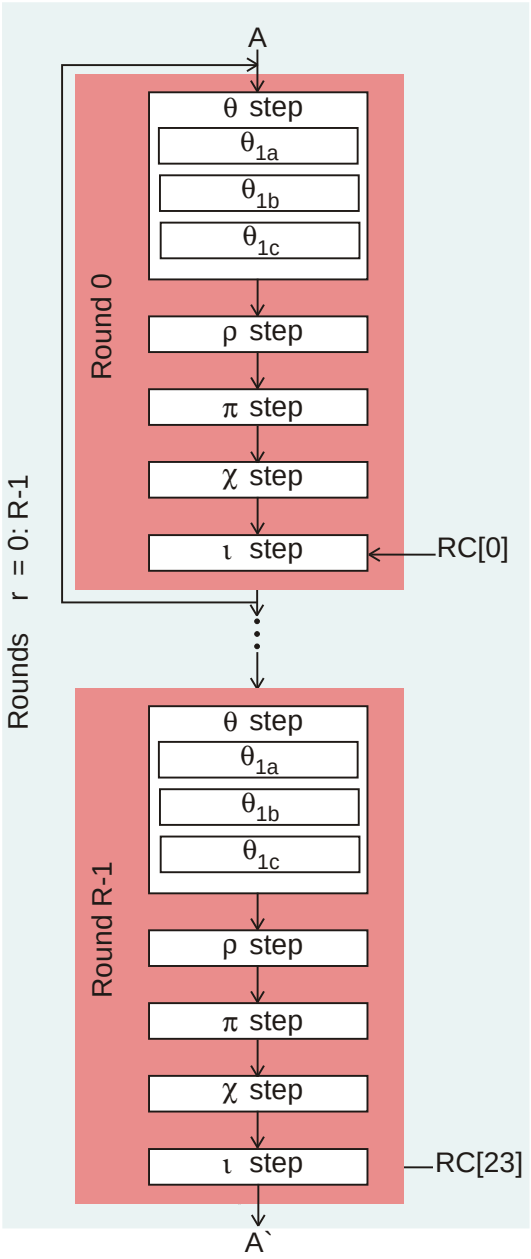


Figure 2.3: SHA-3 rounds

Despite the lack in the literature of existing similar architecture techniques for implementing the SHA-3 algorithm, we provided a comparison between this work and previously reported implementations. The SHA-3 3-D representation of a state allows hardware designers to use different approaches to carry out the algorithm computations. Some of those implementations are lane-wise such in [32] [59]. An alternative technique to perform the computation is slice-wise which was introduced by [30] then further efforts were made by other researchers to improve the throughput and reduce the area in [31], [36], [69].

Another approach is combining both lane and slice-wise computations into a unified design [57]. Other implementations approaches were also reported in the recent literature with focuses on a high throughput [48] [47], or utilizing embedded hardware resources of FPGA such as Look-Up-Table (LUT) [59], [29], Block RAM (BRAM) [32],[60] and Digital Signal Processing (DSP) slices [5]. Due to the flexibility of our computational model, our design could be implemented in slice-wise, lane-wise, or both slice and lane-wise computations based on the scheduling function.

Chapter 3

Develop of DFG Description

3.1 Dataflow Graph Computing model

We introduce in this section a dataflow graph computational model that is more suitable to describe, simulate, and design asynchronous concurrent systems.

3.1.1 Dataflow Graph (DFG) Construction

The data dependency among different tasks comprising the algorithm can always be represented by a directed graph (DG). A directed graph is a collection of *nodes* representing the algorithm variables and *directed arcs* representing the dependencies among the variables. The graph can be expressed as the pair $\mathbb{G} = (\mathbb{N}, \mathbb{A})$ [26]. The operations on the variables are implied. We propose in this work a novel representation of an algorithm as a dataflow graph (DFG) which is composed of three sets *variables* $\mathbb{V}$, *functions* $\mathbb{F}$, and *directed arcs* $\mathbb{A}$ instead of two like in the usual DG.

The proposed DFG is the tuple:

$$\mathbb{G} = (\mathbb{V}, \mathbb{F}, \mathbb{A}) \tag{3.1}$$

The set of *variables* $\mathbb{V} = (v_0, \ldots, v_{n-1})$, which stands for memories in hardware, is a finite set representing the algorithm variables, where $n > 0$. There are three unique types of variables, input, internal, and output. The three variables types were classified based on locations in the algorithm and the number of incoming and

outgoing arcs. The set of *functions* $\mathbb{F} = (f_0, \ldots, f_{m-1})$ which stand for transition in hardware jargon, is a finite set representing the operations and transformation to be carried out on the algorithm variables, where $m > 0$. The set of *directed arcs* $\mathbb{A} = (a_{i,j})$ which represent a communication conduit for data exchange, is a set of directed arcs representing the dependencies among the variables and the functions. An arc directed from a variable v_i to a function f_i defines the variable as an input to the function. An arc directed from a function f_i to a variable v_i defines the variable as an output of the function.

In the DFG of Fig. 3.1, a variable node is represented by a *circle* ◯, and a function is represented by a square □. Notice that our DFG is a directed acyclic graph (DAG) because most algorithms we are interested in are causal [26].

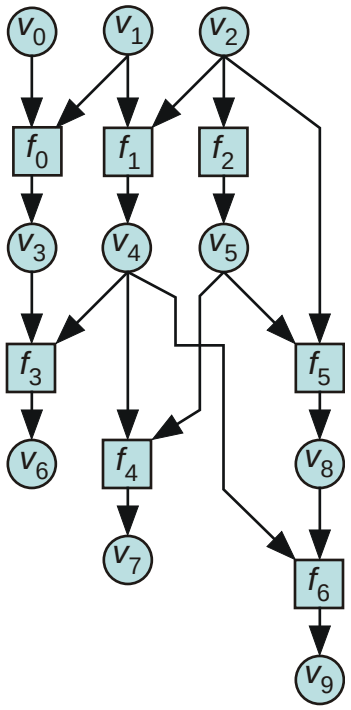


Figure 3.1: Dataflow graph (DFG) for an algorithm

The arcs connect a variable to a function or a function to a variable. The arcs do not connect a variable to a variable or a function to a function. The start of an arc is an output of variable or function, and the end of an arc is the input of a variable or a function. The number of arcs that are leaving or coming to a variable or a function is based on the following rules:

1. The variable could have one or more output arcs and must have only one input arc. The output arcs represent sending copies of the variable to different functions that use that variable. A single input arc from a function to the variable implies that this variable is produced by the function.

2. The function could have one or more input and output arcs. Multiple input arcs imply arguments to that function. Multiple output arcs imply the function produces more than output variable. An example of this is the division function where the quotient and the remainder are produced. Another example could be the addition where the sum and carry out or overflow flag is produced.

Figure. 3.1 shows a DFG example of an algorithm composed of 10 variables and 7 functions. The DFG graph illustrates dependencies among the variables and the functions of the algorithm. No information is indicated by the DFG Fig. 3.1 regarding:

1. Allocation of functions to hardware processors.

2. Association of variables with memories or registers.

3. The timing of availability of variables or execution of functions.

To add the notion of time to the construction of the DFG we use *tokens* as in Fig. 3.2. On the graph *tokens* are represented by black circles ●. Tokens are assigned to the variables when they are valid and could be used. Function f_i is ready to be evaluated when all its input variables v_i's have tokens.

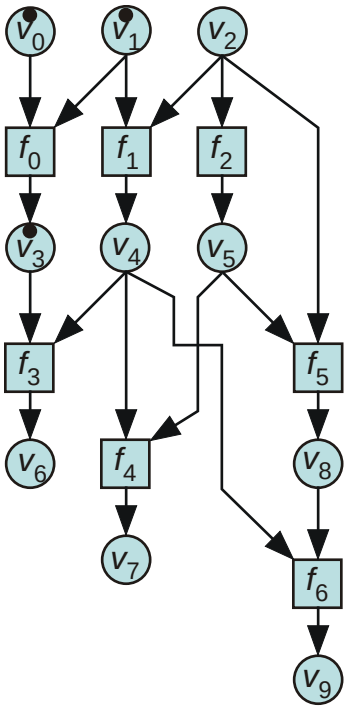


Figure 3.2: State of dataflow graph (DFG) for an algorithm at a given time instance

Referring to Fig. 3.2 we note that at a given time instance t a set of input variables v_0 and v_1 have tokens which indicate that function f_0 has fired. As a result when a function has been fired a token will be placed at the variable node associated with its output to indicate the availability of this variable. This can be seen by an internal variable v_3.

3.1.2 Useful Definitions

In this section, we define some useful terms.

Definition 3.1.1. A variable is an ***input variable*** if it has no incoming arcs. It represents one of the algorithm input variables. Figure 3.2 shows that the algorithm has three input variables v_0, v_1, and v_2.

Definition 3.1.2. A variable is an ***output variable*** if it has no outgoing arcs. It represents one of the algorithm output variables. Figure 3.2 shows that the algorithm has three output variables that represent the output v_6, v_7, and v_9.

Definition 3.1.3. A variable is an ***internal variable*** if it has incoming and outgoing arcs. It represents one of the algorithm intermediate variables. Figure 3.2 shows that variables v_3-v_5 and v_8 represent internal variables.

Definition 3.1.4. A function f_i is a ***target function*** to a variable v_j if a directed arc starts at v_j and terminate at f_i. The variable v_j is one of the input variables for function f_i. Figure 3.2 shows that function f_0 represents the target function to variables v_0 and v_1 .

Definition 3.1.5. A ***parent*** variable v_i of a variable v_j when variable v_i is an input argument to the function associated with variable v_j. Figure 3.2 shows that v_1 is parent to v_4.

Definition 3.1.6. The ***parent set*** of variable v_i is the set of all variables that are parents of v_i. Figure 3.2 shows that v_0 and v_1 are the parent set of v_3.

Definition 3.1.7. A ***child*** variable v_i of a variable v_j when variable v_j is an input argument to the function associated with variable v_i. Figure 3.2 shows that v_3 variable has tow source variables v_0 and v_1, which makes v_3 a *child* of two variables.

Definition 3.1.8. The ***child set*** of variable v_i is the set of all variables that are childes of v_i. Figure 3.2 shows that v_6, v_7 and v_9 variables are a *child set* of v_4.

3.2 Design Space Exploration Methodology for Dataflow Multi-core Computing Architecture

In this section, we discuss how to transform a given algorithm into a Dataflow Multi-core (DMC) architecture. We follow a systematic design space exploration methodology to obtain the DMC architecture. The methodology is divided into three steps:

1. Obtain the DFG associated with the given algorithm. This step is explained in Section 3.1, Subsections 3.2.1 and 4.1.1.

2. Define a memory architecture (distributed/shared) and a strategy for mapping the algorithm variables to the memory modules. This step is explained in Subsections 3.2.2 and 4.1.2 .

3. Define a multicore processor array architecture and a strategy for mapping the algorithm functions to the cores. This step explained in Subsections 3.2.3 and 4.1.3.

3.2.1 Deriving the DFG of an Algorithm

We indicated in Sec. 3.1 that an algorithm is defined through sets of functions, variables and the dependencies between the pairs of variables and functions. Deriving the DFG of an algorithm starts with identifying and classifying the algorithm variables then examination of the dependencies among the variables. The transformations on the variables define the algorithm dependencies and functions with their associated input and output variables. These dependencies produce the DFG discussed in detail in Section 3.1. From the DFG one can infer the algorithm properties such as workload, depth, degree of parallelism and presence of cycles, as discussed in more detail in [26]. The DFG reveals the types of variables as input, internal and output. This classification helps deciding the scheduling of input data, identifying critical paths and determining the delay of producing the outputs. In Fig. 3.3, the DFG graph is modified into sequential equitemporal domains or stages of execution.

The figure is obtained after making several idealized assumptions such as:

1. All inputs are available at time $t = 0$.

2. There are no constraints in memory and I/O bandwidths.

The functions at each domain are evaluated at the same time. For example the functions f_0-f_2 can be evaluated concurrently when all inputs v_0-v_2 are available and can be read simultaneously by all the functions.

Figure 3.3 is useful in determining the algorithm properties such as depth and degree of parallelism. The depth of the algorithm is the number of sequential stages, which is three in our case. This implies that the fastest completion time under ideal conditions is three stage delays.

The degree of parallelism is defined as the maximum number of functions associated with each stage. This defines the maximum number of cores that can operate simultaneously under ideal conditions. From the figure we determine that three cores can operate in parallel.

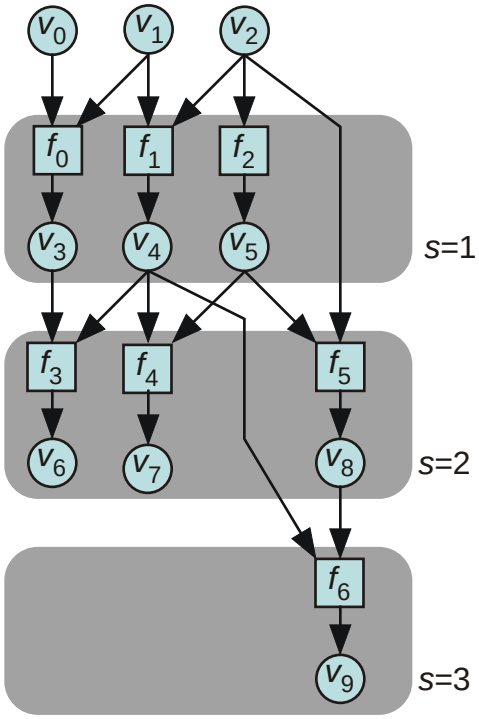


Figure 3.3: Allocation of functions and variables to different equitemporal domains.

3.2.2 Mapping Variables to Memory

We have two cross related mapping problems: map variables to memory modules and map functions to processors. There is a correlation between those two mapping problems. Functions depend on variables and variables are produced by functions. It is a circular relationship between variables and functions, both of them must be optimized taking into account hardware constrains and memory bandwidth. This is akin to the placement and routing problem in VLSI chips.

Communication capability of memory is limited by the number of I/O ports. A variable must be stored in memory but it must be accessed by one or more processors. Memories have large storage capabilities so many-to-one mapping between a set of variables and a single hardware memory is a suitable option. There are three memory architecture design options:

1. Allocate a single shared memory to store all the algorithm variables. This is an all-to-one mapping. This is not an attractive option, since it has the lowest memory bandwidth. Hence, parallelism will be difficult to achieve in such design, since only one variable could be accessed at a given time.

2. Allocate a single memory module to each stage and map the output variables of each stage to the memory assigned to that stage. This design option is classified as globally-distributed/locally-shared memory architecture. This is a many-to-one mapping. It is a suitable option that increases the memory bandwidth for the system. This option allows for parallelism among the stages. However, parallelism within the stage is limited due to the use of shared memory in each stage.

3. Allocate a memory module to each output variable in each stage. Since the stage output variables are associated with a function block, this design option is tantamount to a distributed memory architecture. This is a one-to-one mapping. This is the best option in term of memory bandwidth. Such design permits full parallelism.

3.2.3 Mapping Functions to Processor

Each function in Fig. 3.3 will be executed only once during the execution of the algorithm. Hence, one-to-one mapping of a function into a single hardware processor is not practical in term of area and power needed to implement the processor. The many-to-one mapping between a set of functions and one hardware processor is more suitable for our DMC architecture.

There are three design options for mapping functions to processors:

1. Associate a single processor core to map all the algorithm functions. Functions of all stages will be executed sequentially. This is an all-to-one mapping. It is very efficient for hardware utilization but does not allow parallelism.

2. Associate a processor core to each stage and map all functions of each stage to the processor assigned to that stage. This option allow multiple processor cores to execute functions in parallel at a given time. This is a many-to-one mapping. It shows good hardware utilization and also allows for parallelism.

3. Associate a processor core to each function of the algorithm stages. Functions that belong to a stage will be distributed among available processing units to be executed. This is a one-to-one mapping. It shows a low degree of hardware utilization but offers the most possible parallelism.

The degree of parallelism exhibited by each processor depends on the design of that processor, e.g. whether it is superscalar or not. However, in this work we assume our processor to be capable of executing a single function at a time.

Chapter 4

Applying the DFG to SHA-3 algorithm

4.1 Case Study: DMC Architecture for SHA-3 Algorithm

In this section, we discuss how to transform the SHA-3 algorithm operations described by equations (2.2)-(2.8) and Fig. 2.3 into a DMC architecture. We followed the methodology that was presented in Sections 3.1 and 3.2 to obtain the DMC architecture. It starts with deriving the algorithm graph components using the DFG principles. Then, mapping algorithm variables and functions to modules and processing cores, respectively.

4.1.1 Obtaining the SHA-3 DFG

The SHA-3 3-D state formation could be accessed in a variety of ways. SHA-3 is a multiple rounds algorithm and each round is a collection of five hash functions. The five functions are sequential. Applying the DFG methodology on the SHA-3 algorithm divide the five main functions into seven stages, where the θ-stage is represented by three sub-stages $\theta_{1a}, \theta_{1b}, \theta_{1c}$. Figure 4.1 illustrate the θ-stage three sub-stages functions dataflow graph for calculation of θ-effect.

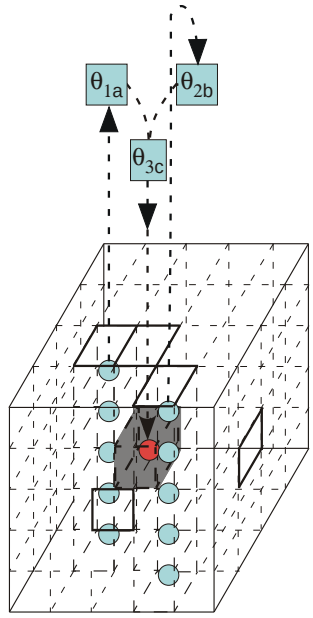


Figure 4.1: θ-stage three sub-functions

Figure 4.2 shows the impact of performing the DFG principles on one of the SHA-3 algorithm state; cube. The SHA-3 algorithm deals with data in the form of a cube along the x-, y- and z-axes of size $C = 5 \times 5 \times w$.

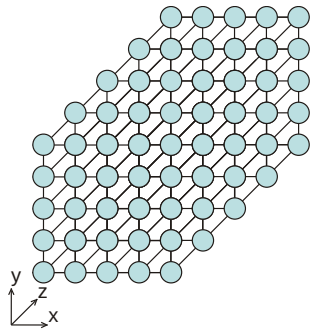


Figure 4.2: 3D Dataflow graph cube

Figure 4.3 shows the rectangle state form of the SHA-3 algorithm after applying the DFG methodology. The algorithm also deals with the data in the form of a rectangle in the x-z-axes of size $P = 5 \times w$. The value of w is determined from Table 2.2.

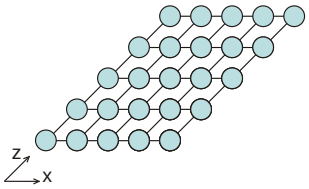


Figure 4.3: 2D Dataflow graph rectangle

We assume the dataflow processors use a word size g bits. The number of input or output variables per-stage depends whether the data comes from a cube or a rectangle. This number can be found using the following equations:

$$n = C/g \qquad \text{for a cube} \tag{4.1}$$

$$m = P/g \qquad \text{for a rectangle} \tag{4.2}$$

As an example, Fig. 4.4 indicates that θ_{1c} stage deals with two forms of s state, a rectangle as its input and a cube as its output. Hence the input variables are $v_{2,j}$ and the output variables are $v_{3,i}$ and the functions at that stage are $f_{3,i}$ with $1 \leq i \leq n$ and $1 \leq j \leq m$.

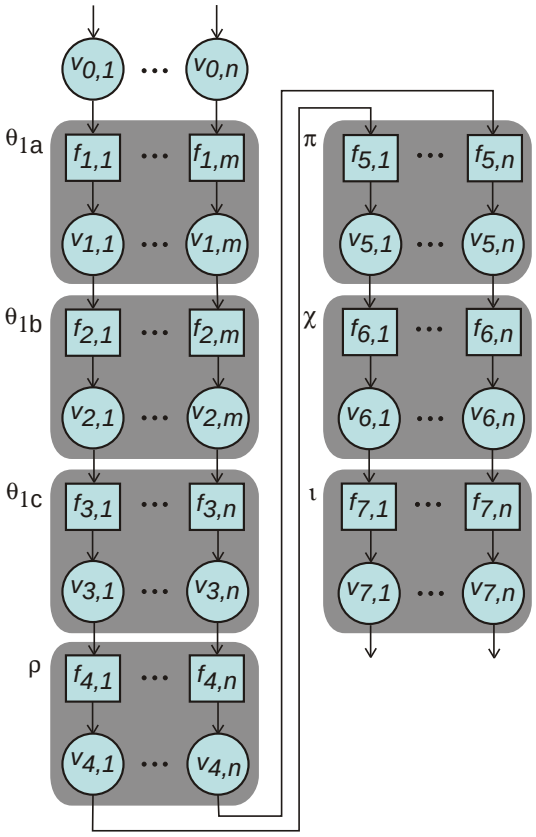


Figure 4.4: DFG of SHA-3 algorithm modeling [3]

4.1.2 Mapping SHA-3 Variables to Memory

Since the SHA-3 algorithm is a multi-stages, we shall follow the second alternative of mapping variables to memory modules. Distributed globally-shared locally mapping allows parallelism globally but does not allow it per-stage, locally it is sequential.

To map the SHA-3 algorithm variables we used heuristics. One of the heuristics is the input variables, all n or m variables of a single stage will be mapped in one memory, which means a single output port. As a result, we will have $M_{1a}, M_{1b}, M_{1c}, M_2, M_3, M_4$ and M_5 memories to store the SHA-3 states of each stage. The first mapping step is to map all input data into a single memory block m_5. The data will be transmitted to the first processor on a single port, a packet at a time. Based on this decision, parallelization of θ_{1a} state is precluded. The data has to be accessed in packet serial format, even though it is all available at t_0. Variables will be fed to the processor based on the scheduling policy. So the output will be packet serial from memory. No

parallel outputs are permitted. The following lemmas result as a consequence of the procedure that all variables of one stage will be mapped in a single memory and the packet serial transmission format of all variables in memory.

Lemma 4.1.1. One variable can be read from input memory of a stage. No state parallelism or throughput at each stage.

Proof. The best known available memory is a dual-ported RAM that allows one read and write operations simultaneously. Despite the large storage capabilities of memory at best one read and write operations can be done simultaneously. □

4.1.3 Mapping SHA-3 Functions to Processor

We adopted the second mapping option. A processor per-stage that does not allow parallelism within stage but allow it globally between stages.

In term of the processing capability, based on the mapping schema that we applied over the variables, all n or m functions of each stage will be mapped in a single processor, which means a single operation at a time. As a result we will have $P_{1a}, P_{1b}, P_{1c}, P_2, P_3, P_4$ and P_5 processors to implement the SHA-3. The first mapping step is to map all input functions of stage θ_{1a} into a single processor P_{1a}. The functions will be executed by the processor sequentially. Functions will operate based on the scheduling of input data. In term of communication capabilities, the I/O limitations imply single input and a single output at a time t. The following lemmas result as a consequence of the assumption that the processor is a simple ALU processor and no parallel operations will occur while operating.

Lemma 4.1.2. The processor of every stage will produce a variable every x clock cycles where x is the number of input variables.

Proof. Our processor is a simple single processor at every stage. Thus, the processor can execute one function out of n or m at a given time t that will only produce one variable. Also, the limitation on memory bandwidth implies a single input variable at a time will be fed to the processor. □

Functions that are associated with any of the seven SHA-3 stages will be mapped in a single processor. The output of those functions will be mapped into a single

memory. Functions operation on one stage is identical but with a different set of input arguments.

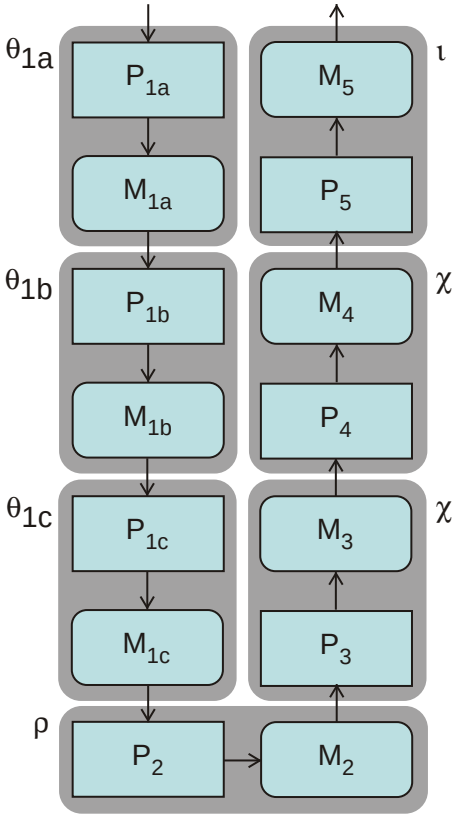


Figure 4.5: DMC seven stages mapping

Fig. 4.5 shows the SHA-3 seven stages mapping. Each stage composes of a hardware processor and a memory block. The output arguments of θ_{1a} stage will be input arguments of the θ_{1b} stage and so forth.

4.1.4 SHA-3 Operations of DMC Architecture

The system consists of self-timed event-triggered operations. As we mentioned earlier, the process will start after the completion of writing the input arguments in the receiver memory. All the inputs arguments are available at $t = 0$ so the scheduling policy of reading the inputs is free of restrictions. Then the processor will start reading the memory based on a specific scheduling policy. We note in this system that the memory has three operations two reads and one write. A processor that write in memory can read it also, and the other processor can only read.

According to Section 2.3, a packet is used to represent each algorithm variable. The representation includes the value of the variable, its unique ID, as well as its target functions (cf. Definition 3.1.4).

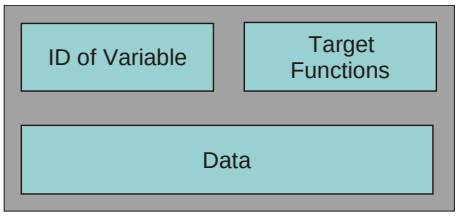


Figure 4.6: The unit of data exchange for the DMC architecture

These fields are illustrated in Fig. 4.6. Packets are propagated throughout the system between nodes. A node that output a packet is a parent of this packet and the generated packet is a child of that node. The system will read the packet and extract the identity of the variable.

The extracted identity then will be used to generate the child set identities of that variable. The source variable become a parent packet and the destination variable is a child packet. The child set will be determined on-the-fly while processing. The added set of destinations and identity bits will increase the size of the data in an acceptable ratio. We minimize the header size to maintain a small packet size by only adding the necessary fields.

Based on the generated child ID, the processor has to check the token counter that counts the maximum number of required counts before firing the variable. The token counter is a mechanism developed to keep a record of the operations of the system. One of the requirements of the memory in this system is to serve as a temporary storage for the variables so every time a processor wants to update a variable in a memory it can retrieve it and operate on it. When a processor checks the tokens counts of a variable and finds it reach threshold token count, it fires it by writing the packet in a memory and indicating to the adjacent receiving processor that the variable here is ready to be read. The designing criteria of our processor form a special purpose processor.

4.1.5 SHA-3 Proposed DMC Architecture

The best way to implement this design on System-on-Chip (SoC) is to build the cryptoprocessor based on network and memory architecture. Figure 4.7 shows a high-level diagram of our SHA-3 DMC architecture. The diagram consists of seven processors, seven memories, and seven routers. We note by the double arrows in the architecture that the memory is being written in only by one processor and being read by a pair of processors. So a memory does not belong to a single processor but belongs to a pair of adjacent processors. However, we need to communicate between processors, so the implementation we do on a SoC is to implement it in the form of a router in the following shape Fig. 4.7, ring shape network-on-chip (NoC). We implemented this architecture, and it is a single direction routing. We adopt a ring

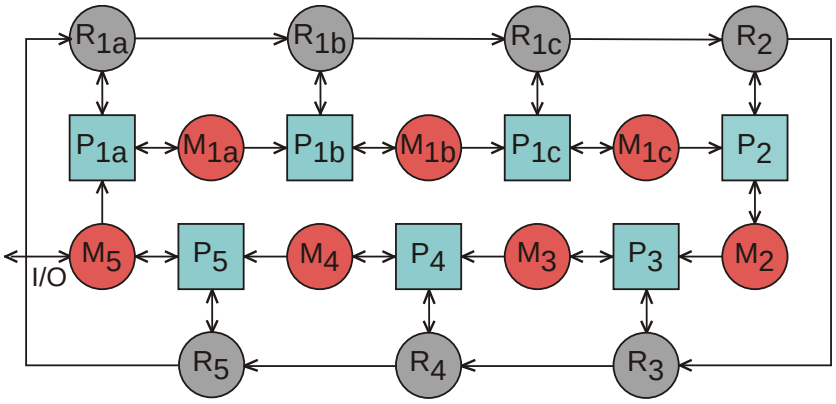


Figure 4.7: SHA-3 ring architecture

architecture because it is suitable for an algorithm with round functions. In a round based algorithm, the last function output will be circled as an input to the algorithm. The circulation makes a ring shape suitable architecture option for a round-based algorithm.

Since communication is packet based, at this stage, we have two alternatives in our design implementation. Whether to add the children ID's in the packet and increase the size of the packet or since these processors have specific functions, then they know how to build the children on the fly, and we chose the second option to reduce the packet size. That will minimize the exchanged packet size; it will be a single transmission versus multi-words serial transmission.

Although it looks like a pipelined system, it is not a pipelined system, since data

exchange is through routers. Pipelined system does not have a router, and the data is propagated without routing. The memory reading sequence of each round could be executed with different scheduling policy. This strategy means each round must be fully processed and stored in memory then the second round will start, which makes it a somewhat round pipelined system.

4.2 Implementation Results and Related work

Table 4.1 shows the results of proposed DMC using VHDL language and synthesized with Xilinx ISE v14.3 tool. The targeted FPGA devices are from Virtex-6 and Kintex-7 families [70] [71]. The throughput in this work is estimated according to the following equation:

$$Th = w \times f \tag{4.3}$$

where w is the processor word size and f is the operating frequency. In our implementation we used $w = 24$ bits and $f = 200$ MHz for the chosen FPGA device. This gives throughput of 4.8 Gbps.

The table also compares our results with published results using conventional implementations of the SHA-3 algorithm. Our implementation uses seven BRAMs and adequate resources of logical slices. The design gives decent results comparing to the previously reported implementations results. Although this initial study indicated that hardware and clock speed can be optimized. However, our choice of design has a significant advantage; it can randomize the execution of the operation without the requirement of retiming, redesigning, and reprogramming.

Table 4.1: Results comparison of FPGA based SHA-3 implementations

Desgin	Device	Slice, BRAM, DSP	Frequency (MHz)	Throughput (Gbps)
This work	Virtex-6	532, 7, 0	200	4.8
	Kintex-7	434, 7, 0	220	5.2
[60]	Virtex-5	151, 3, 0	520	0.251
[29]	Virtex-6	1181, 0, 0	251	4.3
	Kintex-7	1426, 0, 0	309	5.4
[36]	Spartan-6	216, 0, 0	166	0.045
[30]	Virtex-5	188, 0, 0	159	0.864
[47]	Virtex-5	4793, 0, 0	317	12.68
[31]	Virtex-6	116, 0, 0	267	0.108
[5]	Virtex-6	208, 0, 58	451	4.1
	Kintex-7	205, 0, 58	463	4.2
[59]	Kintax-7	1185, 0, 0	629	9.6
[48]	Virtex-6	1115, 0, 0	412	9.8

This hardware implementation relies on dataflow concepts to compute concurrent asynchronous applications. We have applied this computing model to implement a SHA-3 cryptographic algorithm. Beside the hardware implementation, we developed object-oriented MATLAB models to verify and validate the correctness of this computing model, that is being prepared at this time for publication.

Chapter 5

Securing the SHA-3 algorithm

5.1 Data Access Approaches

The main idea in our proposed countermeasure approaches is to use dataflow computing platforms and dynamically vary the order of execution of the algorithm computations. We take advantage of the ability of dataflow computing to recognize when a computation is ready to be executed to produce a valid output. Dataflow computing is able to correctly execute the algorithm operations regardless of the order of feeding of the data. Changing the order of operations execution in dataflow computing mode is cost efficient since no modifications in the software or hardware implementations are needed. The asynchronous execution feature of dataflow computing allows randomizing execution of the operations. This servers as an effective countermeasure to frustrate side channel attacks without incurring extra costs of delay or hardware resources compared to the classic countermeasures using control-flow computing. Hence, our strategy will be to vary the order of operation execution of the algorithm taking advantage of the multiple rounds and the multiple functions of the SHA-3 algorithm. We have two possible data access approaches deterministic and stochastic, as will be discussed in more detail in Sec. 5.1.1 and 5.1.2, respectively.

The transformation of the SHA-3 state format from a three-dimensional array of b-bits to a set of variables arranged in a 3D cube give us a large number of options for accessing and processing the data.

Accessing the SHA-3 variables is based on assigning IDs to the algorithm variables.

The ID of a variable depends on the arrangement of data in a cube or a rectangle, refer to Fig. 4.2 and 4.3, respectively. The IDs for variables in a cube or rectangle are given by:

$$ID_C = y + Y(z-1) + YZ(x-1) \tag{5.1}$$

$$ID_R = x + X(z-1) \tag{5.2}$$

where X, Y and Z denote the number of variables (words) along the x-, y- and z-axes. The values of X and Y are equal to 5, while the value of Z is given by:

$$Z = w/g \tag{5.3}$$

where w is the number of bits per lane (cf. Table 2.2) and g is the word size (cf. Sec. 4.1.1). Table 5.1 shows the effect of the word size g on the value of Z as used in this work.

Table 5.1: Effect of the word size g on the value of Z as for the case when $b = 1600$ and $w = 64$ bits.

g	1	2	4	8	16	32
Z	64	32	16	8	4	2

5.1.1 Deterministic Data Access Scheduling Strategies

There are six deterministic scheduling strategies. The first strategy starts by iterating over the x then y then z directions. We call this scheme XYZ scheduling. The second strategy iterates over the x then z then y directions. We call this XZY scheduling. There are four more scheduling strategies: YXZ, YZX, ZXY, and ZYX.

Algorithm 1 is the pseudo code for generating the access order of the data according to the six scheduling schemes. The algorithm output is a vector specifying the sequence of accessing the variables.

Line 1: specifies the address of the first element of the output vector $V(0)$.

Line 2: selects the desired schedule according to the input value $1 \leq i \leq 6$.

Lines 3, 11, and 19: represent the first three deterministic schedules.

Lines 4 – 6: show the first scheduling scheme XYZ iteration mechanism over the algorithm variables. The ordering of the three nested FOR loops specifies the order of data access as per the XYZ schedule.

Line 7: generates the value of $V(j)$ according to Eq. (5.1).

Algorithm 1 Pseudo code to assign execution sequences of Figure 4.2 state variables using six deterministic scheduling.

function: *get_schedule*(X, Y, Z, i)

1: **initialize:** $j = 0$
2: **switch** i
3: **case** 1: % XYZ Scheduling**:**
4: **for** $z = 0 : Z - 1$ **do**
5: **for** $y = 0 : Y - 1$ **do**
6: **for** $x = 0 : X - 1$ **do**
7: $V(j) = \text{get.ID}(x, y, z, Y, Z);$ $j = j + 1;$
8: **end for**
9: **end for**
10: **end for**
11: **case** 2: % XZY Scheduling**:**
12: **for** $y = 0 : Y - 1$ **do**
13: **for** $z = 0 : Z - 1$ **do**
14: **for** $x = 0 : X - 1$ **do**
15: $V(j) = \text{get.ID}(x, y, z, Y, Z);$ $j = j + 1;$
16: **end for**
17: **end for**
18: **end for**
19: **case** 3: % YXZ Scheduling**:**
20: $\vdots$
21: **end switch**
22: **return** V

5.1.2 Stochastic Data Access Scheduling Approach

The second approach is to access the data using a random permutation among the data address space using the Fisher-Yates shuffle algorithm [24] [37], as shown in Algorithm 2.

Algorithm 2 Pseudo code of the Fisher-Yates shuffle algorithm.

function: $rand\_permute(V)$

1: $L = get\_length(V)$
2: **for** $i = 0$ **:** $L - 2$ **do**
3: $\quad n \leftarrow : i \leq n < L$
4: $\quad$ exchange $V(i) \leftrightarrow V(n)$
5: **end for**
6: **return** V

Algorithm 2 requires the user to supply a vector V which contains all the IDs of the variables.

Line 1: determines the length of the input data vector V.

Lines 3 − 5: implement the Fisher-Yates shuffle algorithm.

5.2 Proposed Countermeasure Approaches

This section details the four SCA countermeasure approaches that are based on the two scheduling schemes discussed in Sec. 5.1. Figure 5.1 shows the different options for assigning input data schedules to the R rounds of the SHA-3 algorithm. In the figure `schd` refers to the deterministic scheduling strategies discussed in Algorithm 1 and `rand` refers to the deterministic scheduling strategies discussed in Algorithm 2.

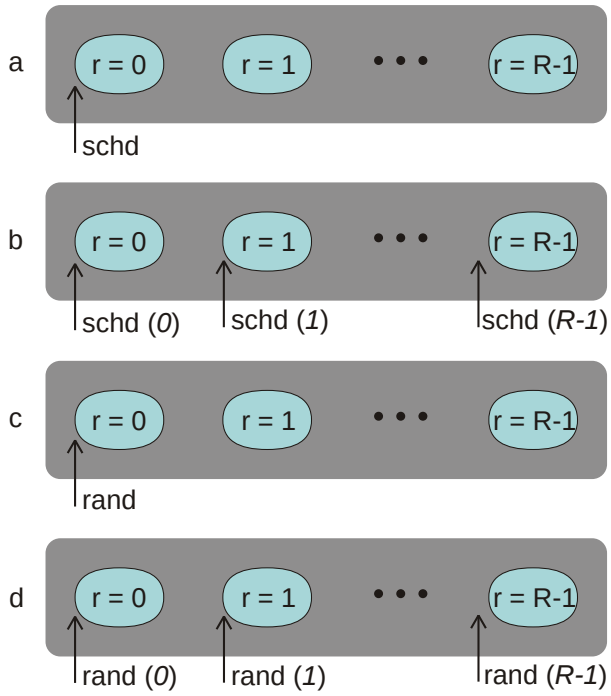


Figure 5.1: SCA countermeasures options. (a) When one deterministic schedule is used for all rounds. (b) When different deterministic schedules are used for different rounds. (c) When one stochastic schedule is used for all rounds. (d) When different stochastic schedules are used for different rounds.

5.2.1 First Countermeasure Approach

Figure 5.1a shows the first countermeasure approach where all the algorithm rounds use the same input data schedule. Algorithm 3 is the pseudo code for the first countermeasure approach.

Algorithm 3 Pseudo code of the 1*st* countermeasure approach using a deterministic scheme introduced in Algorithm 1.

1: $i = randi(1:6)$
2: $V = get\_schedule(X, Y, Z, i)$
3: $A_0 = A$;
4: **for** $r = 0$ **:** $R - 1$ **do**
5: $\quad A_{r+1} = \text{SHA-3}\ (V, A_r)$;
6: **end for**
7: **return** A_R

Line 1 randomly selects an integer between 1 and 6 to decide on which schedule to use among the six possible deterministic schedules discussed in Sec. 5.1.1. **Line** 2 returns the corresponding access order vector V generated by Algorithm 1. **Line** 3 starts the iterations, where A is defined in Eq. (2.1). Then the computation will proceed using the chosen scheduling scheme for all rounds. Finally, **Line** 7 returns the hash value A_R.

5.2.2 Second Countermeasure Approach

Figure 5.1(b) shows the second countermeasure approach where every algorithm round uses a different input data schedule. Algorithm 4 is the pseudo code for the second countermeasure approach. **Line** 1 starts the iterations. At the beginning of each

Algorithm 4 Pseudo code of the 2*nd* countermeasure approach using deterministic scheduling introduced in Algorithm 1.

1: $A_0 = A$;
2: **for** $r = 0$ **:** $R - 1$ **do**
3: $\quad i = randi(1 : 6)$
4: $\quad V = get\_schedule(X, Y, Z, i)$
5: $\quad A_{r+1} =$ SHA-3 (V, A_r);
6: **end for**
7: **return** A_R

round, a random schedule is chosen.

5.2.3 Third Countermeasure Approach

Figure 5.1c shows the third countermeasure approach where a random order of choosing the input data is used at the first round. Algorithm 5 is the pseudo code for the third countermeasure approach.

Algorithm 5 Pseudo code of the $3rd$ countermeasure approach using stochastic scheduling introduced in Algorithm 2.

1: $i = randi(1:6)$
2: $V = get\_schedule(X, Y, Z, i)$
3: $V = rand\_permute(V)$
4: $A_0 = A$;
5: **for** $r = 0$ **:** $R - 1$ **do**
6: $\quad A_{r+1} = \text{SHA-3}\ (V, A_r)$;
7: **end for**
8: **return** A_R

Line 3 randomly shuffles the input data vector. This random schedule vector will be used for all the SHA-3 rounds.

5.2.4 Fourth Countermeasure Approach

Figure 5.1d shows the fourth countermeasure approach where a random order of choosing the input data is used at every round of the SHA-3 algorithm. Algorithm 6 is the pseudo code for the fourth countermeasure approach.

Algorithm 6 Pseudo code of the $4th$ countermeasure approach using stochastic scheduling introduced in Algorithm 2.

1: $i = randi(1:6)$
2: $V = get\_schedule(X, Y, Z, i)$
3: $A_0 = A$;
4: **for** $r = 0$ **:** $R - 1$ **do**
5: $\quad V = rand\_permute(V)$
6: $\quad A_{r+1} = \text{SHA-3}\ (V, A_r)$;
7: **end for**
8: **return** A_R

Line 5 generates a new random shuffle of the input data at the start of each round.

5.3 Implementation Results & discussion

The SHA-3 DFG for all 24 rounds was simulated using MATLAB and abject-oriented programming paradigm. The algorithm consisted of 24 rounds. Each round consisted of seven different functions, as shown in Fig. 4.4. Five of these functions dealt with data in the form of cubes and two functions dealt with data in the form of rectangles, as shown in Fig. 4.2 and Fig. 4.3, respectively. The number of objects in a cube or a rectangle is given in general by Eq. (4.1) or (4.2), respectivley. For our simulations a cube had 50 objects and a rectangle had 10 objects and the word size was 32 bits.

We used all four countermeasure scheduling strategies discussed in Sec. 5.2.

5.3.1 First Countermeasure Approach Results

Figure 5.2 shows a Gantt chart for the processors of Fig. 4.4 during the first three rounds of the SHA-3 algorithm and all rounds use the same XYZ schedule. The

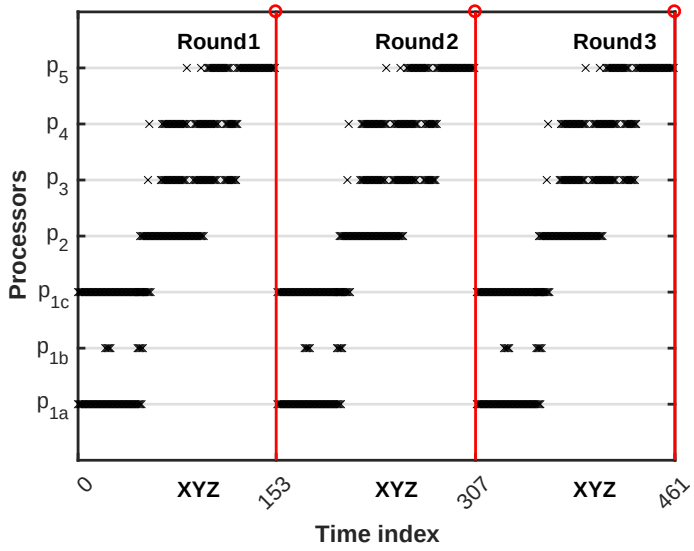


Figure 5.2: The first three round processors activities using 1st countermeasure approach.

vertical axis shows the seven processors and the horizontal axis shows the clock cycle to finish the first three rounds. An 'x' is used to represent when a processor is active. The activity pattern for each processor shows periodicity since the same schedule is used for each round. Table 5.2 shows the association between the SHA-3 functions of Fig. 4.4 and the processors of Fig. 5.2.

Table 5.2: Association between the SHA-3 functions of Fig. 4.4 and the processors of Fig. 5.2.

Function	θ_{1a}	θ_{1b}	θ_{1c}	ρ	π	χ	ι
Processor	P_{1a}	P_{1b}	P_{1c}	P_2	P_3	P_4	P_5

Figure 5.2 clearly shows the processors activity status at any given time during each round. The distribution of activity of each processor during each round is revealed. For example P_{1a} shows concentrated activity approximately during the first third of each round. On the other hand, P_3 shows lesser activity concentration and is active later in the round. Similar observations can be made about the other processors.

Figure 5.3 shows number of active processors during each clock cycle of operation. We notice that at least one processor is active at a given clock cycle. Also at most

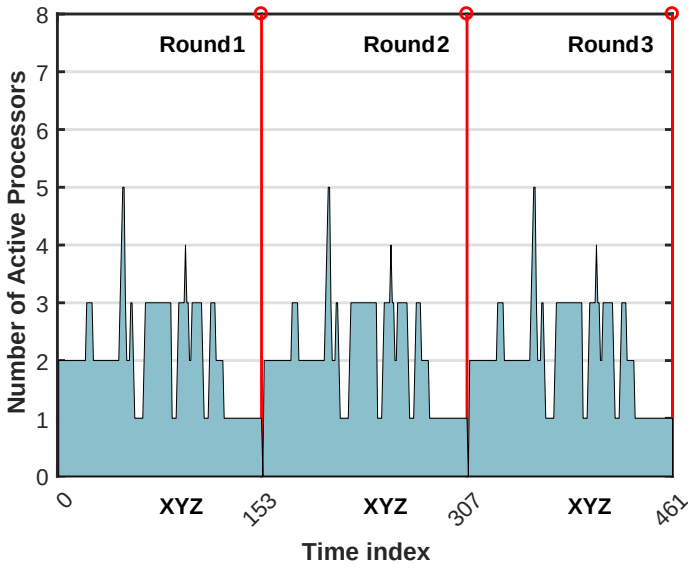


Figure 5.3: Activity profile for the the first three rounds using the 1*st* countermeasure approach.

five processors are active at a given clock cycle. The variation in number of active processors is dictated by two factors: the data dependency of the SHA-3 algorithm and the data scheduling function used. Although Fig. 5.3 shows periodic behaviour, it is very difficult to determine the identity of the variables being processed at a given time.

5.3.2 Second Countermeasure Approach Results

Figure 5.4 shows a Gantt chart for the processors of Fig. 4.4 during the first three rounds of the SHA-3 algorithm and every round uses a different input data schedule.

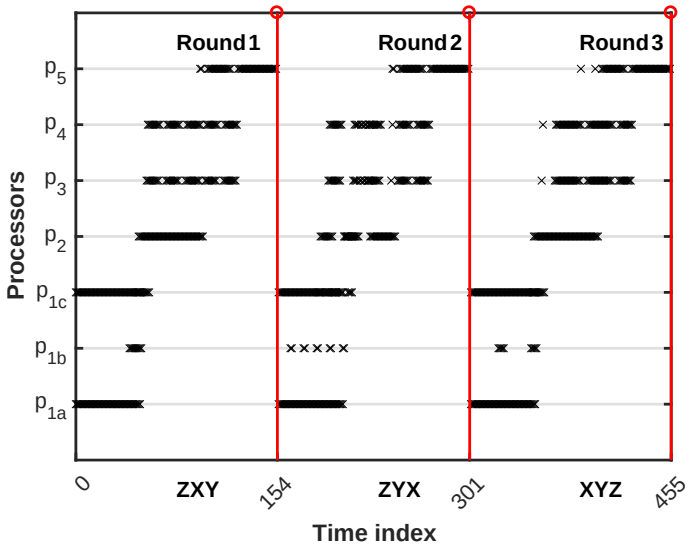


Figure 5.4: The first three round processors activities using $2nd$ countermeasure approach.

The activity patterns for some processors show periodicity despite different schedules are used for each round such as P_{1a}. However it should be pointed out that the *order* of the data being processed is different. On the other hand the activity patterns of some processors, such as processor P_{1b}, show variation among the rounds.

Figure 5.5 shows the number of active processors during each clock cycle of operation. At most six processors are active at a given clock cycle.

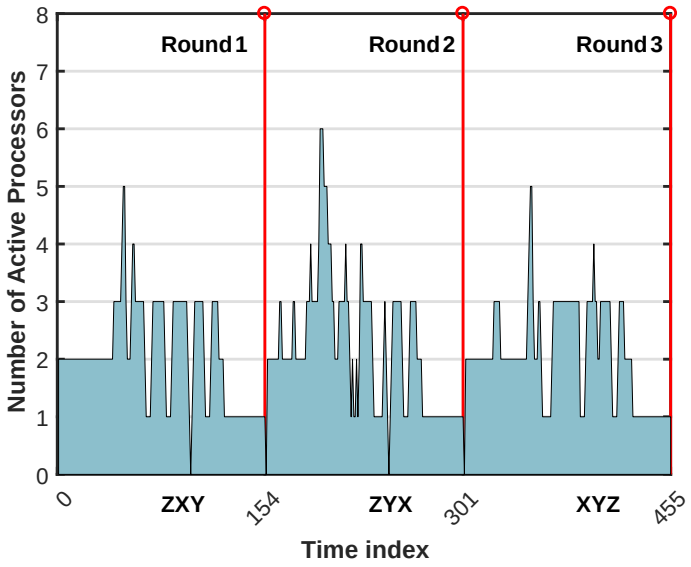


Figure 5.5: Activity profile for the the first three rounds using the 2*nd* countermeasure approach.

A very important observation is that the time needed to complete a round varies according to the schedule being used. The first round takes 154 clock cycle, the second round takes 147 clock cycle, and the third round takes 154 clock cycle. We note the both the first and the third rounds take the same time to complete even though they use two different schedules. This could be used to advantage to thwart delay SCA.

5.3.3 Third Countermeasure Approach Results

Figure 5.6 shows a Gantt chart for the processors of Fig. 4.4 during the first three rounds of the SHA-3 algorithm and all rounds use the same `rand` schedule.

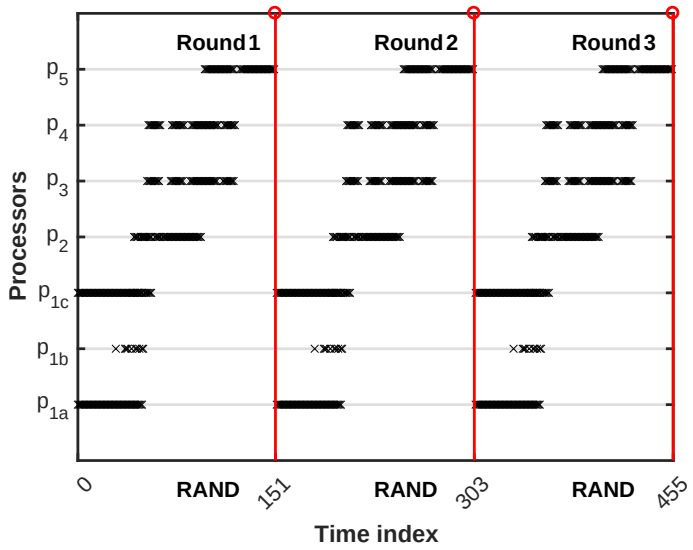


Figure 5.6: The first three round processors activities using $3rd$ countermeasure approach.

Figure 5.7 shows number of active processors during each clock cycle of operation.

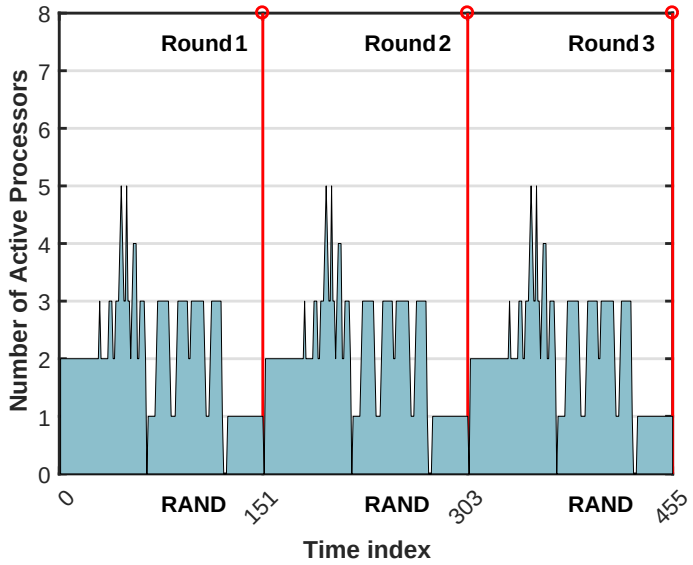


Figure 5.7: Activity profile for the the first three rounds using the $3rd$ countermeasure approach.

At most six processors are active at a given clock cycle. Figure 5.7, as well as Fig. 5.3, Fig. 5.5 and Fig. 5.10 show that some of the processors are not active at every clock cycle. The inactive processors could be put in hibernation mode to reduce the overall power consumption. This clearly shows that dataflow processing is very much

suited to green computing compared to the traditional von Neumann machines.

5.3.4 Fourth Countermeasure Approach Results

Figure 5.8 shows a Gantt chart for the processors of Fig. 4.4 during the first three rounds of the SHA-3 algorithm and every round uses a different `rand` schedules. We notice that the time needed to complete a round varies since each round uses a

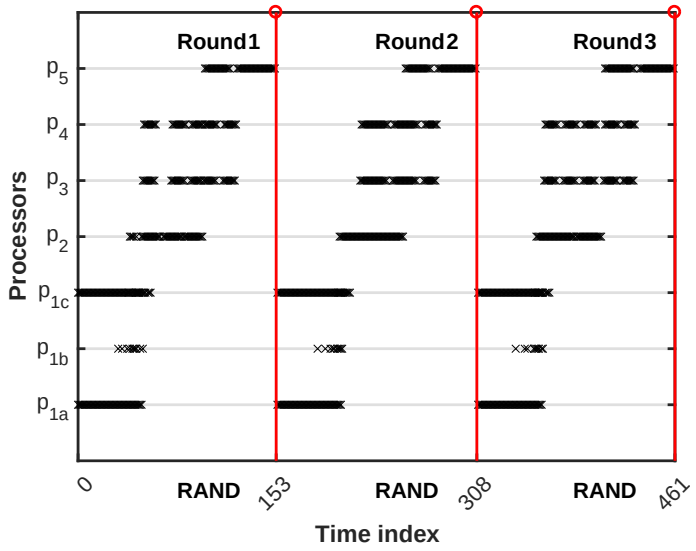


Figure 5.8: The first three round processors activities using $4th$ countermeasure approach.

different random schedule. Processors P_{1a} and P_{1c} activities appear almost identical, however, the variables being processed are different.

Figure 5.9 shows the IDs of the variables being processed by P_{1a}, P_{1b}, and P_{1c} for the first three rounds. The veiticale axis represent the variables' IDs.

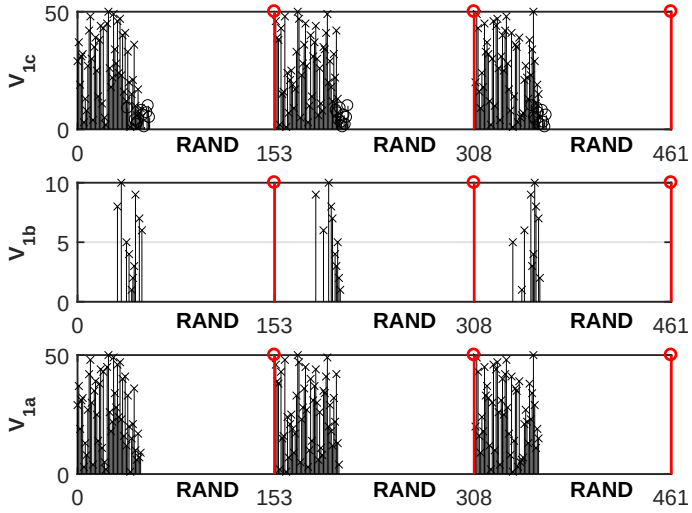


Figure 5.9: The first three round time traces of processed variables using $4th$ countermeasure approach.

The bottom trace of Fig. 5.9 shows the ID traces of variables V_{1a} that are being processed by P_{1a}. The middle trace shows of the figure is for the case of variables V_{1b} associated with P_{1b}, and similarly for the top trace for variables V_{1c}. The figure clearly indicates that the IDs profile are different for each round. So the attackers will not be able to distinguish the variables IDs. As a result, by using the proposed countermeasure, this multi-core cryptosystem exhibits better protection against SCAs. Figure 5.10 shows number of active processors during each clock cycle of operation

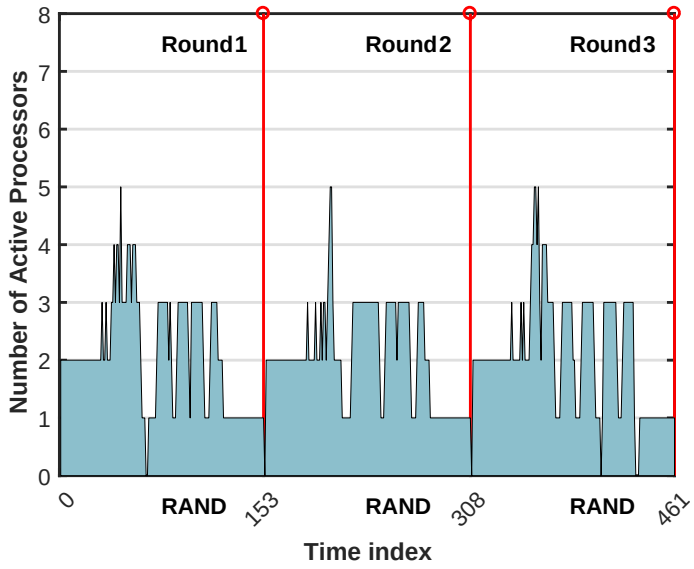


Figure 5.10: Activity profile for the the first three rounds using the $4th$ countermeasure approach.

Table 5.3 shows the time needed to complete the first three rounds of the SAH-3 algorithm using the four countermeasures. The delay for each round in the first approach is the same, as expected, due to using the same systematic schedule for all rounds. The delay for each round in the second approach is variable due to use of different systematic schedules for each round. The delay for each round in the third approach is the same due to using the same random schedule for all rounds. The delay for each round in the fourth approach is different due to use of different random schedules for each round.

Table 5.3: Comparison between the four countermeasures clock cycles of the the first three rounds.

Countermeasure	**Round** 1	**Round** 2	**Round** 3
$1st$ approach	154	154	154
$2nd$ approach	155	147	154
$3rd$ approach	152	152	152
$4th$ approach	154	155	153

As expected, the delay for Round 3 in the second approach is identical to the delay in the first approach since both use the same systematic schedule. This can be ascertained from Fig. 5.3 and 5.5.

A very interesting observation is that some of the round delays in the fourth approach, when a random schedule is used, are identical to the round delays in the first and second approaches where systematic schedules were used. This proves that mere analysis of round delays will not reveal the nature or ID of the data being processed. This helps in further frustrating side-channel attacks.

5.3.5 Immunity to Attacks Assessment

Assuming an adversary is observing the operation of the SHA-3 algorithm. The immunity of proposed countermeasures to SCAs is assessed in term of the computational complexity of the scheduling policy used. The complixity of the proposed four levels countermeasures techniques can be estimated as follow:

$1st$ Approach

According to Fig. 5.1a and Algorithm 3, the computation complexity is $\approx 6!$.

$2nd$ Approach

According to Fig. 5.1b and Algorithm 4, the computation complexity is $\approx 6! \times R$, where R is the number of SHA-3 algorithm rounds.

$3rd$ Approach

According to Fig. 5.1c and Algorithm 5, the computation complexity is $\approx XYZ!$, where X and Y are equal to 5 while the value of Z can be found in Table 5.1. As the word size g increases, the value of Z is decreases. The complexity is decreased.

$4th$ Approach

According to Fig. 5.1d and Algorithm 6, the computation complexity is $\approx XYZ! \times R$. Likewise, the effects of the size of g apply to this approach.

The complexity for this case can be approximated using Stirling's formula [13]:

$$\alpha! \approx \sqrt{2\pi\alpha}(\frac{\alpha}{e})^{\alpha} \tag{5.4}$$

where

$$\alpha = XYZ! \times R \tag{5.5}$$

Chapter 6

Contributions and Future Work

6.1 Contributions

The dissertations contributions are given as follows:

1. Data-Flow Implementation of Concurrent Asynchronous Systems. (*Published*)

   we proposed a dataflow computational model that is more suitable to describe, simulate, and design concurrent asynchronous systems.

2. Multi-Core Dataflow Design and Implementation of Secure Hash Algorithm-3. (*Published*)

   In this work, we proposed a new dataflow graph (DFG) scheme that is more suitable to describe, simulate, and design concurrent asynchronous systems. We proposed a novel methodology to obtain a dataflow multi-core computing (DMC) architecture for a given algorithm. This is a three-step methodology that starts with applying the DFG construction principles to the algorithm. The next two steps involve mapping the algorithm variables to memory modules and mapping the algorithm operations to the processing cores. We applied the proposed methodology to obtain a novel DMC architecture for the secure hash algorithm-3 (SHA-3). An application specific embedded multi-core system implementation of the SHA-3 algorithm on FPGA is presented.

3. Secure and parallel Dataflow Design of the SHA-3 Algorithm.

We proposed several countermeasures approaches for side channel attacks. We used dataflow computing paradigm to implement the SHA-3 hashing algorithm. We proposed four countermeasure techniques against side channel attacks for SHA-3 hashing. The countermeasure techniques were based on choosing stochastic or deterministic input data scheduling techniques. Extensive simulations of the the SHA-3 algorithm and the proposed countermeasures approaches were performed using object-oriented MATLAB models to verify and validate the effectiveness of the techniques. The computation complexity of the proposed countermeasures approaches was assessed.

6.2 Future work

This dissertation proposed ideas and methodology that could be expanded in the future in different directions:

1. Assess the use of different word sizes on the scheme performance and study the tradeoff between design flexibility and speed.

2. Apply the proposed methodology to parallelize other cryptographic algorithms such as AES.

3. Formulate the mapping of algorithm variables and operations as an optimization problem, and solve it using optimization techniques such as genetic algorithms, ant colony, or simulated annealing. The cost function of the optimization problem can be based on number of cores, memory modules, and most importantly communication between the functions and variables to prevent memory collisions.

4. Implement the proposed three steps methodology on graphics processing unit (GPU). The problems that might be encountered would be ensuring the threads access variables associated with the memory of their processors.

5. Develop a general purpose multi-core system that support different operations to accommodate the acceleration of several algorithms.

6. Modify routers architecture to support dynamic routing based on functionality and availability of the processing cores.

Bibliography

[1] Abderazek Ben Abdallah. *Multicore systems on-chip practical software/hardware design*. Atlantis Press, Paris, 2010.

[2] Dakshi Agrawal, Brucel Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side—Channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 29–45. Springer Berlin Heidelberg, 2003.

[3] Ali Alzahrani and Fayez Gebali. Multi-core dataflow design and implementation of secure hash algorithm-3. *IEEE Access*, 6:6092– 6102, 2018.

[4] Cilk Arts. Smoth path to multicore. *URL: http://www.cilkplus.org/*, 2018.

[5] Arshad Aziz et al. A low-power SHA-3 designs using embedded digital signal processing slice on FPGA. *Computers & Electrical Engineering*, 55:138–152, 2016.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions. *Submission to NIST (Round 3)*, 2011.

[7] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. *URL: http://keccak. neokeon. org/Keccak-implementation-3.2. pdf*, 2012.

[8] Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. The KECCAK SHA-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.

[9] Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. KECCAK. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 313–314. Springer, 2013.

[10] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.

[11] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

[12] Christina Boura and Anne Canteaut. A zero-sum property for the Keccak-f permutation with 18 rounds. In *IEEE International Symposium on Information Theory Proceedings (ISIT)*, pages 2488–2492. IEEE, 2010.

[13] Keith Conrad. Stirlings formula. *Available in http://www. math. uconn. edu/˜ kconrad/blu rbs/analysis/stirling. pdf*, 2016.

[14] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems*, pages 725–725. Springer, 1999.

[15] David E Culler and Gregory M Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[16] Sourav Das and Willi Meier. Differential biases in reduced-round Keccak. In *International Conference on Cryptology in Africa*, pages 69–87. Springer, 2014.

[17] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb 1982.

[18] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[19] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.

[20] JB DENNIS. Data flow supercomputers. *IEEE Computer*, 13(11):93–100, 1980.

[21] Itai Dinur, Orr Dunkelman, and Adi Shamir. Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials. In *International Workshop on Fast Software Encryption*, pages 219–240. Springer, 2013.

[22] Itai Dinur, Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michał Straus. Cube attacks and cube-attack-like cryptanalysis on the round-reduced Keccak sponge function. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 733–761. Springer, 2015.

[23] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds.(NIST FIPS)-202*, August 2015.

[24] Ronald Aylmer Fisher, Frank Yates, et al. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, (Ed. 3.), 1949.

[25] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[26] Fayez Gebali. *Algorithms and Parallel Computing*. John Wiley, New York, 2011.

[27] Fayez Gebali, H. Elmiligi, and M. W. El-Kharashi. *Networks-on-Chips: Theory and Practice*. CRC Press, Boca Raton, FL, 2009.

[28] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[29] Tatsuya Honda, Hendra Guntur, and Akashi Satoh. FPGA implementation of new standard hash function keccak. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pages 275–279. IEEE, 2014.

[30] Bernhard Jungk and Jurgen Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 235–241. IEEE, 2011.

[31] Bernhard Jungk and Marc Stöttinger. HobbitSmaller but faster than a dwarf: Revisiting lightweight SHA-3 FPGA implementations. In *International Confer-*

ence on ReConFigurable Computing and FPGAs (ReConFig), pages 1–7. IEEE, 2016.

[32] Jens-Peter Kaps, Panasayya Yalla, Kishore Surapathi, Bilal Habib, Susheel Vadlamudi, Smriti Gurung, and John Pham. Lightweight implementations of SHA-3 candidates on FPGAs. *Progress in Cryptology–INDOCRYPT*, pages 270–289, 2011.

[33] Richard M Karp and Rayamond E Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[34] K. M. Kavi, B. P. Buckles, and U. N. Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, C-35(11):940–948, Nov 1986.

[35] Richard F Kayser. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62, 2007.

[36] Stéphanie Kerckhof, François Durvaux, Nicolas Veyrat-Charvillon, Francesco Regazzoni, Guerric Meurice de Dormale, and François-Xavier Standaert. Compact FPGA Implementations of the Five SHA-3 Finalists. In *Cardis*, volume 7079, pages 217–233. Springer, 2011.

[37] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[38] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in cryptology*, pages 789–789. Springer, 1999.

[39] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[40] Paul C Kocher, Joshua M Jaffe, and Benjamin C Jun. Using unpredictable information to minimize leakage from smartcards and other cryptosystems, December 4 2001. US Patent 6,327,661.

[41] Milos Krstic, Eckhard Grass, Frank K Gürkaynak, and Pascal Vivet. Globally asynchronous, locally synchronous circuits: Overview and outlook. *IEEE Design & Test of Computers*, 24(5):430–441, 2007.

[42] Andrew J Leiserson, Mark E Marson, and Megan A Wachs. Gate-level masking under a path-based leakage metric. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 580–597. Springer, 2014.

[43] Ted G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[44] Pei Luo, Yunsi Fei, Liwei Zhang, and A Adam Ding. Differential fault analysis of sha3-224 and sha3-256. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 4–15. IEEE, 2016.

[45] D May, H Muller, and N Smart. Random Register Renaming to Foil DPA. In *Cryptographic Hardware and Embedded SystemsCHES 2001*, pages 28–38. Springer, 2001.

[46] H. Mestiri, F. Kahri, B. Bouallegue, M. Marzougui, and M. Machhout. Efficient countermeasure for reliable KECCAK architecture against fault attacks. In *2017 2nd International Conference on Anti-Cyber Crimes (ICACC)*, pages 55–59, March 2017.

[47] Hassen Mestiri, Fatma Kahri, Mouna Bedoui, Belgacem Bouallegue, and Mohsen Machhout. High throughput pipelined hardware implementation of the KECCAK hash function. In *International Symposium on Signal, Image, Video and Communications (ISIVC)*, pages 282–286. IEEE, 2016.

[48] Harris E Michail, Lenos Ioannou, and Artemios G Voyiatzis. Pipelined SHA-3 implementations on FPGA: Architecture and performance analysis. In *Proceedings of the Second Workshop on Cryptography and Security in Computing Systems*, page 13. ACM, 2015.

[49] Veljko Milutinović, Jakob alom, Nemanja Trifunovic, and Roberto Giorgi. *Guide to DataFlow Supercomputing*. Springer, 2015.

[50] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[51] Simon Moore, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor. Improving smart card security using self-timed circuits. In *Eighth International Symposium on Asynchronous Circuits and Systems*, pages 211–218. IEEE, 2002.

[52] Paweł Morawiecki, Josef Pieprzyk, and Marian Srebrny. Rotational cryptanalysis of round-reduced keccak. In *International Workshop on Fast Software Encryption*, pages 241–262. Springer, 2013.

[53] María Naya-Plasencia, Andrea Röck, and Willi Meier. Practical analysis of reduced-round keccak. In *International Conference on Cryptology in India*, pages 236–254. Springer, 2011.

[54] NVIDIA. A parallel computing platform and programming model. *URL: https://developer.nvidia.com/cuda-zone/*, 2018.

[55] OpenMP. OpenMP : The OpenMP API specification for parallel programming . *URL: http://http://www.openmp.org//*, 2018.

[56] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[57] Peter Pessl and Michael Hutter. Pushing the limits of SHA-3 hardware implementations to fit on RFID. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 126–141. Springer, 2013.

[58] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. *Smart Card Programming and Security*, pages 200–210, 2001.

[59] Muzaffar Rao, Thomas Newe, and Ian Grout. Secure hash algorithm-3 (SHA-3) implementation on Xilinx FPGAs, suitable for IoT applications. In *8th International Conference on Sensing Technology (ICST 2014), Liverpool John Moores University, Liverpool, United Kingdom, 2nd-4th September*, 2014.

[60] Ismail San and Nuray At. Compact KECCAK hardware architecture for data integrity and authentication on FPGAs. *Information Security Journal: A Global Perspective*, 21(5):231–242, 2012.

[61] Jurij Silc, Borut Robič, and Theo Ungerer. Asynchrony in parallel computing: from dataflow to multithreading. In *Progress in computer research*, pages 1–33. Nova Science Publishers, Inc., 2001.

[62] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure Integrated Circuits and Systems*, pages 27–42. Springer, 2010.

[63] Mostafa Taha and Patrick Schaumont. Differential power analysis of MAC-Keccak at any key-length. In *International Workshop on Security*, pages 68–82. Springer, 2013.

[64] Taha, Mostafa and Schaumont, Patrick. Side-channel analysis of MAC-Keccak. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 125–130. IEEE, 2013.

[65] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *2002 Proceedings of the 28th European Solid-State Circuits Conference , ESSCIRC 2002.*, pages 403–406. IEEE, 2002.

[66] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10246. IEEE Computer Society, 2004.

[67] Lih-Yang Wang, Chi-Sung Laih, Hang-Geng Tsai, and Nern-Min Huang. On the hardware design for DES cipher in tamper resistant devices against differential fault analysis. In *The 2000 IEEE International Symposium on Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva.*, volume 2, pages 697–700. IEEE, 2000.

[68] Barry Wilkinson and Michael Allen. *(Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2Nd Edition)*.

[69] Jori Winderickx, Joan Daemen, and Nele Mentens. Exploring the use of shift register lookup tables for KECCAK implementations on Xilinx FPGAs. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2016.

[70] Xilinx. Virtex-6 Family Overview. Product Specification, DS180 (v2.5), Xilinx, aug 2015.

[71] Xilinx. 7 Series FPGAs Data Sheet: Overview. Product Specification, DS150 (v2.5), Xilinx, aug 2017.