Efficient FPGA Implementation of the SHA-3 Hash Function

Magnus Vik Sundal

Thesis to obtain the Master of Science Degree in

Electronics Engineering

Supervisor: Prof. Ricardo Jorge Fernandes Chaves

Co-supervisor: Prof. Leonel Augusto Pires Seabra de Sousa

Examination Committee

Chairperson: Prof. Pedro Miguel Pinto Ramos

Supervisor: Prof. Ricardo Jorge Fernandes Chaves

Members of the Committee: Prof. Fernando Manuel Duarte Gonçalves

October 2016

Acknowledgements

This thesis concludes my master degree course in Electronics Engineering at Instituto Superior Técnico in Lisboa in Portugal. It also concludes my just over two-years stay in this great country which is noticeably warmer and more humid than my own country, Norway. These two years have given me invaluable experiences together with new friends, my partner and her family who have open up their home and made me feel sincerely welcome. This course has met high expectations and I now feel ready to embark upon a professional career as an Electronics Engineer in a new location. During the course of this thesis I have had the pleasure of working with a supervisor with great technical and pedagogical skills and a highly motivating nature. I am therefore very grateful to Professor Ricardo Chaves for the valuable guidance along the way. I would like to thank my partner Silvia Barros for her endless patience, comfort, and general guidance and with whom I anticipate a multitude of adventures to come. Her family which I now consider a part of my own has far exceeded any expectation concerning hospitality for which I am forever grateful. Lastly I would like to thank my own family back home and especially my mother, Margit Vik Sundal, who is the main explanation for where I am today. Without her, I would most likely not have started an engineering degree in the first place. She remains a personal motivation and role model.

Resumo

O presente trabalho cobre as implementações em FPGA da nova função hash criptográfica padrão, SHA-3, com especial foco na sua eficiência. As funções hash criptográficas são uma parte essencial da criptografia moderna extensamente utilizadas em diversas aplicações, como a autenticação de mensagens e usuários. A SHA-3 representa a próxima geração de funções hash criptográficas e eventualmente assumirá o lugar da sua antecessora, SHA-2, que é uma das funções padrão mais usadas atualmente.

As funções hash estão preparadas para serem implementadas em hardware e são usadas como co-processadores para a realização de hashing de mensagens em larga escala, como é o caso dos pacotes Ethernet. A eficiência, definida como a taxa de transferência por área requerida, é um objetivo chave de performance para implementações de hardware.

O objetivo deste projeto tem sido alcançar uma solução para melhorar o estado da arte atual relativamente à eficiência, através da análise e exploração profundas da SHA-3 e da literatura existente.

O estado da arte mostra-se inconsistente relativamente a aplicações da função hash e objetivos de performance. Deste modo, neste trabalho são propostas soluções baseadas em considerações lógicas e objetivos de performance que excedem a performance do estado da arte. As funções estrutura funcionam como entidades independentes e a performance é baseada numa avaliação fiável. Para além disto, através de modelos teóricos baseados em fatores apropriados e realistas, a eficiência da SHA-3 mostra um claro limite superior que já foi alcançado. Um maior potencial de eficiência só existe sob pré-condições específicas, dependendo do tamanho das mensagens.

Abstract

This thesis presents the proposal of improved structures, supported on FPGAs, for the new cryptographic hash function standard, SHA-3, with a special focus on efficiency. Cryptographic hash functions are an essential part of modern cryptography and are used extensively in applications requiring message and user authentication and digital signatures. SHA-3 represents the next generation of cryptographic hash functions and is forecast to assume the position of its predecessor, SHA-2, which is one of the most prevalent standards today. Hash functions are well suited to be implemented in hardware as co-processors, performing large-scale hashing of messages such as Ethernet packages.

The goal of this project has been to devise structures and implementation approaches which are able to improve the existing state-of-the-art with respect to efficiency, where efficiency is defined as the achievable throughput per required area.

While several structures and optimizations have already been proposed in the existing state-of-the-art, these tend to be somewhat inconsistent and non-systematic regarding the applications and expected messages for the hash function and the targeted metrics. Herein, several solutions are proposed considering existing and novel optimization techniques, while also proposing an evaluation model to better evaluate the possible structural options and to define clear upper bounds to the achievable results.

Contents

Abbreviations

ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CLB	Configurable Logic Block
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FF	Flip Flop
FFT	Fast-Fourier-Transform
FIFO	First-in-First-Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
HDL	Hardware Description Language
IC	Integrated Circuit
IO	Input/Output
KAT	Known-Answer-Test
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit
LUT	Look Up Table
MAC	Message Authentication Code
MBM	Multi-Block-Message
MMH	Multi-Message Hashing
MSB	Most Significant Bit
MUX	Multiplexer
NIST	National Institute of Standards and Technology
OTP	One-Time-Programmable
PAR	Place and Route
PCB	Printed Circuit Board
PISO	Parallel In Serial Out
PLL	Phase Locked Loop
RAM	Random Access Memory
RC	Round Constant
ROM	Read only Memory
RTL	Real Time Logic
S-box	Substitution Box
SCA	Side-Channel Attack
SDP	Simple Dual Port
SMH	Single-Message Hashing

SRAM	Static RAM
SRL	Shift Register Lookup Table
V-5	Virtex-5 FPGA family
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WCS	Worst Case Scenario
WE	Write Enable

Nomenclature

χ	Chi - A step-mapping of the Keccak round function providing non-linear transformation
ι	Iota - A step-mapping of the Keccak round function providing round asymmetry
π	Pi - A step-mapping of the Keccak round function providing permutation
ρ	Rho - A step-mapping of the Keccak round function providing permutation
c	Capacity - size in bits of the inner part of the state which is initially empty
f	Frequency - MHz
r	Block size - size in bits of the outer part of the state where the message is injected
θ	Theta - A step-mapping of the Keccak round function providing diffusion
A	Area - FPGA slices
E	Efficiency - Gb/s*slice
FF	Folding factor - the degree in which the state is folded (state /internal data path)
L	Latency - the number of clock cycles required to process a message block of size r
PL	Pipeline registers - the quantity of pipeline registers incorporated in a structure
R	Rotation - Lane rotation cause by the rho step mapping
T	Throughput - Gb/s
UF	Unrolled factor - the quantity of round function instances in a structure

List of Figures

1 Introduction

Cryptographic hash functions are highly common in everyday life and provide integrity in authentication applications of users and messages. Bank transactions are an example of an application where it is crucial to know that the message transmitted is the same as the one which is received. As technology advances, a constant arms-race exists between threats and protection of matters concerning most aspects of our everyday life. There are many existing cryptographic hash functions, however, few currently bear the status of general acceptance [1], meaning that they are officially considered secure. We have already seen the dawning of the era of "Internet of Things" and with the forecast explosion of Internet-connected devices, the responsibility of engineers to provide implementations with an adequate level of security only increases.

In 2007, National Institute of Standards and Technology (NIST) concluded that there was a likelihood of a near-future discovery of a flaw in the current hash function standard, SHA-2. Already, papers had been published which proved a reduction in strength of the algorithm [2]. In practical terms SHA-2 is still considered secure to this day, but as a preventive measure, a public competition for the next standard called SHA-3 was initiated in 2007. Through several rounds of elimination, the contestants were narrowed down until Keccak was announced as the new standard in 2012 with four approved sub-variants of the algorithm.

Besides the need for being cryptographically secure, a hash function must be efficient, both in terms of area and throughput. For example, in digital signatures they must be able to hash potentially large quantities of data as quickly as possible. SHA-3 has proven to perform better than its predecessors in both software [3] and hardware [4]. Since the early stages of the SHA-3 competition, authors have presented increasingly optimized implementations of the hash function for a variety of objectives such as minimal area or high efficiency. The latter is defined as the throughput per required area. The former is useful in environments where resources are scarce, however, a larger design might be more efficient than multiple instances of a smaller one if sufficient resources are available. While many cryptographic functions are implemented in software, hash functions require minimal configuration and are therefore apt for hardware implementations, which benefits from improved parallelism and higher throughputs when compared with software implementations. Hardware implementations can be used as a co-processor which aids processors in off-loading highly demanding tasks such as large scale hashing of Ethernet packages.

The state-of-the-art regarding hardware implementation of SHA-3 has already matured throughout several years of development. Despite this, it was the belief of the author and supervisor that this relevant field was not fully explored and that room for improvement still existed.

While many advancements have been made, the existing literature is found to be seemingly non-consistent in both the utilization of optimization techniques and considerations regarding the applications of the hash function and the overall performance objective. The aim of this thesis is to clearly map the state-of-the-art with respect to these aspects and to arrive at new designs based on reasonable and logical considerations towards improving the efficiency and potentially the existing SHA-3 hardware implementations. Hardware implementations on Field Programmable Gate Arrays (FPGAs) have been the main focus of this project. While other technologies for programmable logic exists, FPGAs are convenient for prototyping as they are re-programmable and have a short development cycle. It is also the most common choice of technology among the state-of-the-art which ensues trivial assessment of the performance with respect to the existing literature. The focus is on structural optimization techniques and while the end result is presented as an FPGA implementation, the approaches and considerations should be valid for hardware in general.

1.1 Goals

The main objective of this work is to improve the state-of-the-art of efficient hardware implementations of the SHA-3 hash function standard. A special focus is made on FPGA implementations and the goal is to arrive at a proposed solution which exceeds the existing literature in efficiency. For the proposed solution to be efficient, it must achieve optimal throughput for rapid processing of messages, while requiring minimal area for implementation. A secondary goal is to define the upper bound efficiency of a SHA-3 FPGA implementation through theoretical models based on considerations of realistic conditions.

1.2 Requirements

A requirement for the absolute success of this work is to arrive at a SHA-3 FPGA implementation which exceeds the existing state-of-the-art with respect to efficiency. The proposed solution should be based on proper and realistic consideration with respect to hash function applications and the messages which are used for hashing. Performance results must be based on a reliable evaluation and the structure should be presented as a stand-alone-entity and not rely on other programmable logic components for complete functionality. This means that an interface should be incorporated in order to provide the necessary communication with the processor for hashing of messages and provision of the corresponding message digest following the algorithm specifications. While the targeted implementation technology is FPGAs, for easy prototyping and evaluation the proposed solution should be aimed at generic hardware implementations. As the performance objective is a function of the required area, the resource utilization should be adequately balanced, i.e. excessively utilizing one distinct type of FPGA resource compared with other types should be avoided. However, for comparative reasons, slices should be the main utilized FPGA resource and adoption of other resources should be done at a later stage with actual implementation in a physical environment.

1.3 Main contributions

The main contributions resulting from this work are:

- Design and implementation of proposed SHA-3 computation structures which exceed the existing state-of-the-art with respect to efficiency, by up to 28% on the same technology.
- Development of wrapping component, allowing for stand-alone components, allowing to evaluate the performances based on a reliable assessment.
- Development of theoretical models for the evaluation of the relevant optimization techniques, including the required preconditions related with their applicability.
- Demonstration of the theoretical upper bound efficiency for SHA-3 hardware implementations given reasonable considerations.

Additionally, a paper based on this work has been presented at the REC2016 conference in Vila Real in Portugal and a paper has been submitted to the 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) in Virginia, USA.

1.4 The document structure

Chapter 2 gives an introduction to the technologies which are involved in this project. This includes hash functions in general, hardware implementations particularly on FPGAs, SHA-3 and the underlying Keccak algorithm. The existing state-of-the-art is introduced in Chapter 3, where the relevant literature and existing implementations are analyzed based on their distinct features and contributions. The proposed solutions are presented in Chapter 4, including the description of the research and analysis beyond the existing state-of-the-art which shaped the resulting structures. Chapter 5 contains the implementation details of the several proposed SHA-3 structures and the specific factors which have been considered for each individual case. These structures are based on both the existing literature and the analyses and developed models proposed in the previous chapter. Chapter 6 presents an evaluation of the implemented structures and a comparison with the proposed models and the existing literature. Finally, concluding remarks are presented along with an outlook towards future work.

2 Background

Before covering the relevant aspects of the state-of-the-art and the developed work, it is important to have an understanding of the basic concepts of the technologies involved. This chapter starts by discussing the main concept of hash functions. Following this, an introduction of hardware designs and programmable logic is presented. The benefits of implementing algorithms in hardware are given along with some of the challenges involved in the design process. The chapter is concluded with the general construction of SHA-3 and the four sub-versions which are supported by the standard. The sub-versions are distinct by a set of parameters affecting the throughput and security of the underlying algorithm called Keccak.

2.1 Hash functions

Hash functions are an essential part of modern cryptography. Given a message of random length encoded as a binary string, a hash function will compute another binary string of fixed length which is conventionally called a digest, as depicted in Figure 2.1. A common use of hash functions is in message authentication where the digest works as a fingerprint for the original message and is transmitted along with it. The receiver of the message can therefore validate whether the message has been altered by hashing it and comparing this digest with the one that was received. If there are differences, it means that an active adversary has altered the message. Naturally, a set of preconditions are required for this to work. The hash function in itself should contain no secrets so that it can be distributed freely. Additionally, the hash function must be of sufficient quality so that the digest cannot be altered to correspond with a tampered message. A set of these qualifications are presented below.

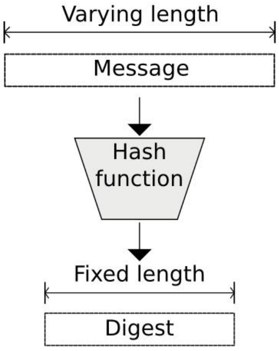


Figure 2.1: The basic concept of hash functions.

As with most cryptographic functions, the initial binary string or message is usually defined as an element or *message* m of a set M $(m \in M)$ and the digest similarly $c \in C$. There is a set of properties which defines a good hash function, namely it must be infeasible to calculate m given the digest. However, the inverse operation (from the message to the digest) should be easy. This property is called *preimage resistance* [5] and is mathematically described as a one-way function [6]. A one-way function from a set X to a set Y easily computes the image $y = h(x)$ for all $x \in X$. It is however *computationally infeasible* to go the other way, i.e for any arbitrary image $y \in Y$ to find x so that $h(x) = y$. The function is also defined as being non-secret, meaning that the security claim is not dependent on any structural components being confidential [7]. This does not correspond completely with the definition of a hash function which can rely on a key for operation. Computationally infeasible is loosely defined, but refers to the time required to map corresponding elements of the function sets with each other (all $x \in X$ to all images $y \in Y$ of the function h) until one arrives at the specific x for a given y. For the computation to be infeasible, the time required must be marginally higher than the lifetime of the cryptographic function. If it fails to be so, then the hash function is not preimage resistant. For most types of modern cryptographic hash functions the security claim is related with the length of the digest, i.e. the number of steps required to break the one-way function by brute-force-attack is in the order of $2^{c/2}$.

Another property of a strong cryptographic hash function is its *collision resistance*. The digest c has a fixed length and is usually smaller than the message m. This problem is explained by the pigeon hole principle [8]: If you have more pigeons than holes, there will be one or more holes with more than one pigeon. For a hash function, this inevitably means that there will be one or more messages m corresponding to the same digest c. If a hash function is collision resistant, it is computationally infeasible given two messages m_1 and m_2 to find a hash-text c so that $h(m_1) = h(m_2) = c$.

SHA-1 is a commonly used hash function still today, even though it is technically considered broken. It was discovered in 2005 that the previously considered difficulty of finding a collision, or the level of collision resistance, of the full 160-bit SHA-1 is in fact not 2^{80} (half the digest length) but instead 2^{69}. This new technique [9] is over 2000 times faster than the previous basic brute-force attack and is therefore within grasp of current computing technology [10].

2nd-preimage resistance is the last main property rating of a hash function. It is similar to the collision resistance, albeit much easier to obtain. Given a message m_1, it is computationally infeasible to find a second message m_2 which corresponds to the same digest c, i.e. $m_1 \neq m_2$ and $h(m_1) = h(m_2)$.

The rating of the different probabilities are based on how probable a successful attack is. The 2^{nd}-preimage resistance will not define the lowest bound of collision resistance and a hash function with this property is therefore inferior to a hash function being collision resistant. Fulfilling the collision resistance property automatically implies 2^{nd}-preimage resistance, but not preimage resistance.

2.2 Hardware co-processors

There are a couple of key differences between implementing cryptographic functions in software or hardware. The obvious advantages of software is accessibility with a short time to market and a low development and maintenance cost. This immediately reflects the downsides in that they can more easily be accessed and reconfigured by an adversary. Further more, programs are usually running in a shared memory space and on top of an operating system which creates much room for vulnerabilities. Hardware is less accessible and development is in general a costlier and longer process. The pay-off lies in factors such as speed - for specific applications, and security which in addition to lack of accessibility is also caused by improved random functions. Co-processors in hardware are advantageous for non-general applications such as cryptography and offer high parallel processing power compared to general purpose Central Processing Units (CPUs) [11]. While many cryptographic functions such as key-exchange for asymmetric encryption standards are conventionally implemented in software, cryptographic hash functions require a much lower level of reconfigurability and are very apt for implementation in hardware. SHA-3 was specifically designed with hardware implementations in mind. The algorithm does not utilize complex mathematical functions such as Fast-Fourier-Transforms (FFTs) or Substitution Boxes (S-boxs). Instead it relies on multiple simple sub-steps which allows for a short critical path and high system frequency [12].

The two main classes of hardware technologies exist, namely Application-Specific Integrated Circuits (ASICs) and FPGAs. ASICs are One-Time-Programmable (OTP), Integrated Circuits (ICs) which provide exceptionally efficient and self-contained implementations for specialized applications and have cheap unit-cost at large quantities. Many of the drawbacks with ASICs are avoided with FPGAs as they are re-programmable, have lower unit-cost at low quantities and with a shorter time to market. Hence, they are commonly used for prototyping designs which are later implemented on an ASIC. The ability of FPGAs to be re-programmed and to provide high parallel processing power makes them highly applicable as co-processors, aiding processors by offloading intensive computations involving signal processing [13] and cryptography [14]. Recent FPGA models also offer security providing concepts such as authenticated reconfiguring. Vulnerabilities which are more difficult to avoid are the ones exploited by Side-Channel Attacks (SCAs) based on information such as timing, power dissipation and electromagnetic emission. Restricted physical access to components dealing with sensitive data is necessary in virtually all scenarios. Still, FPGAs allow for easy adaptation of the cryptographic computation to more reliably resist SCAs.

The utilization of area on an FPGA can be assessed by the number of slices, Look Up Tables (LUTs) or Configurable Logic Blocks (CLBs) required by the design, as they constitute approximately the available logic on the IC at various granularities. The consensus is to count slices, but the specific definition of a slice is dependent on the FPGA family and must be taken into account. The relevancy of the solutions proposed in this work are intended to be independent of the FPGA model and also possibly the programmable logic technology. Still, the performance of a structure is dependent on the platform and to ease the task of comparing the various existing implementations, a standard FPGA model is chosen. As the most prevalent model among the relevant existing literature, the Virtex-5 FPGA by Xilinx is herein used as the default platform for performance evaluation. Of the available FPGA vendors on the market, Xilinx has steadily kept the largest market share for many years [15, 16]. While Virtex has been the vendor flagship, the Virtex-5 FPGA family (V-5) are presently affordable and future-rich devices and are therefore considered as good representatives for the market. A typical modern FPGA is made up of two variations of slices and the general functionality of a quarter of the main type, SLICEL, is depicted in Figure 2.2. Both types provide a 6-input LUT, multiplexer, flip flop and basic logic. A minority of the total number of slices are of type SLICEM where the LUT can be configured as a 64-bit distributed Random Access Memory (RAM) or 32-bit shift register [17]. Modern FPGAs also contain a rich supply of additional components such as Digital Signal Processor (DSP) slices, BRAM for storage of larger quantities of data, various clock sources and Input/Output (IO) drivers. Naturally, these should be proportionally used along with the general slices, but the applicability depends on the added routing delay.

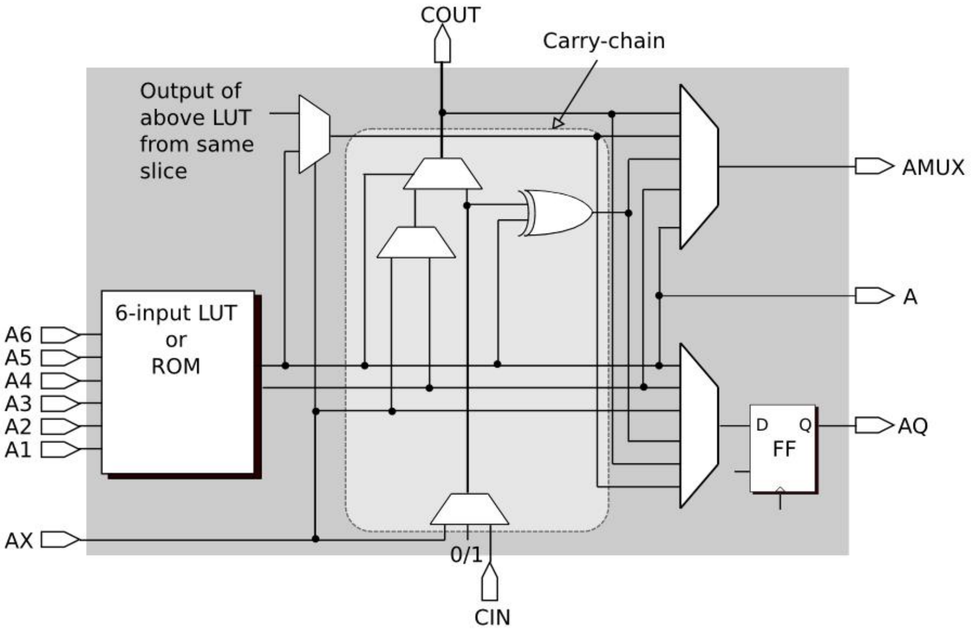


Figure 2.2: Simplified illustration of 1/4 of a slice of type SLICEL on a Virtex-5 FPGA.

Unlike most other markets, FPGAs do not have a similar Electronic Design Automation (EDA) tool diversity since the main two vendors, Xilinx and Altera, have pushed out all other competition of significance [18]. The design tools issued by these vendors are therefore usually involved in the design flow and the end result is highly dependent on the performance of the tool. The design flow consists of four main steps: Synthesis, Translate, Mapping and Place and Route (PAR) [19]. Synthesis translates the design into hardware components, whether the design is schematic or language based through a hardware description language such as Very High Speed Integrated Circuit Hardware Description Language (VHDL). Synthesis of programmable logic is different from software compiling as there are more variables with respect to which components the language-based design corresponds to. It is therefore a crucial step of the design flow. Translate converts the generic hardware components into vendor specific resources and Mapping allocates the design into actual slices on the FPGA which optionally includes the physical placement. PAR mainly handles the routing of the implementation. The steps of the design flow are more interconnected than it may seem. The process can be repeated in order to optimize certain aspects such as timing or area. Despite the importance of design flow, much responsibility is left to the designer. Additionally, the FPGA design can be carefully tweaked in order to be able to reach a performance close to the maximum throughput potential. The designer must have a thorough understanding of how the design flow works and how the description-language is translated into hardware. In order to avoid unnecessary and inefficient utilization of resources, careful typesetting is required to precisely communicate with the EDA tool which components should be used. For instance, knowing precisely the full potential of a slice is important in order to obtain better implementation results and FPGA usage.

2.3 Keccak algorithm

The name Keccak refers to a family of sponge functions [20] which are functions that can be used as stream ciphers, hash functions, Message Authentication Codes (MACs) and pseudo--random number generators depending on the configuration. In general, these functions are based on a fixed length transformation and a padding rule for the input which are capable of mapping a variable-length input to a variable length output. A sub-set of the Keccak sponge functions are currently included in the SHA-3 standard for cryptographic hash functions. Keccak is based on the earlier RadioGatun hash function [21] which distinguished itself from other hash functions by being based on a variable-length compression function rather than a fixed one, thus behaving more like a stream cipher than a block cipher. From this, the sponge function was developed. A sponge function simply means that a message is iteratively absorbed into the state with a fixed size throughout multiple rounds of a simple round function before the digest is squeezed out. The following presentation of Keccak is general, as it can have multiple variations. Nevertheless, the configurations related with SHA-3 will be underlined.

2.3.1 The sponge construction

The chassis, or skeleton of a sponge function is the sponge construction. The initial idea was for the sponge construction to be a reference to expressed security claims in hash functions, instead of using the conventional *random oracle model*. A random oracle is close to a perfect hash function, but returns a random infinite string instead of a truncated one which is not realistic. Originally, hash functions were assumed to be as strong as the length of the digest and were therefore using the random oracle model as proof of security. Contrary to a random oracle, a hash function consists of finite memory and a fixed-length truncation and is therefore prone to inevitable inner collisions and vulnerable to various attacks that exploit this [22]. A new model, the *random sponge function*, was therefore developed specifically for hash functions and stream ciphers, where the effects of finite memory is considered. The model takes into account a set of parameters, such as whether the function consists of a permutation or a transformation and most specifically the capacity c, which is the size of a partition of the state. Up until this point c has represented the message digest of general hash functions, but represents the security parameter c in a sponge construction. It is proven in [23] that if the permutations do not possess any structural distinguisher, the most successful attack on a random sponge function will be the generic one, where the resistance of the function is upper bound by $2^{c/2}$.

The sponge construction with an underlying random permutation or transformation f composes with this model a random sponge function, as illustrated in Figure 2.3.

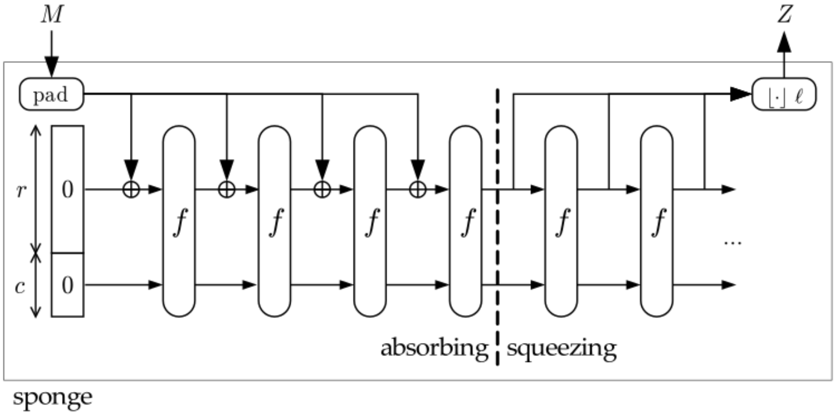


Figure 2.3: The sponge construction. / CC BY 3.0 @noekeon.org

The function f acts on a state, with width $b = r + c$ where r is the outer state and c is the inner state. At initialization, r is where the message is injected and c is empty, thus the size of r effectually determines the bit rate of the construction. The multiple iterations of the function f in the absorbing phase denotes the sponge nature of Keccak when the input message is larger than r. The Squeezing phase is similarly iterated for digests larger than r. The digest is at least half the size of c in all the sub-versions of SHA-3 and it is extracted from the outer state r after processing. The sponge construction works in three steps:

1. Initialization - The complete state is cleared and the message M is padded to multiples of r-bit blocks.

2. Absorbing phase - The padded r-bit message blocks are one-by-one XORed bit-wise into the r-part of the state and processed by the Keccak-*f* function. Keccak-*f* is therefore called iteratively if the message $M > r$ and each consecutive block is merged with the product of the previous blocks. When all message blocks have been XORed into the state and processed, the Squeezing phase begins.

3. Squeezing phase - The outer state is XORed to output blocks interleaved by the sponge function f, similarly to the absorbing phase. The output blocks are truncated to its first ℓ bits and added to Z. If $\ell = |r|$, which is always the case for SHA-3, there will be no iteration of the sqeezing phase.

2.3.2 The Keccak sponge functions

The benefits of the sponge construction is the security claim and a very high flexibility related with configurations. This flexibility is consistently seen in the underlying Keccak function

Keccak-*f*[*b*]. The Keccak functions can be configured into seven different transformations which are referred to as permutations. These permutations produce different digests from the same message and differ in the size of the state, the ratio between the size of the inner and outer state and the number of rounds the function is processed. The seven different permutations of Keccak, $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, are named after the size of the state which is organized as a 3-dimensional array. Figure 2.4 depicts the state which possible dimensions are $5 \times 5 \times w \in \{1, 2, 4, 8, 16, 32, 64\}$ where w is along the z-axis and is proportional to the width b of the permutation by $b = 25 \times w$. Each of the seven permutations have a distinct security claim where the largest types are the most secure. The number of rounds n_r for each permutation is given by $n_r = 12 + 2 \times log_2(w)$. Only the largest permutation with $b = 1600$ is supported for SHA-3. This means that the relevant Keccak function for this work is the Keccak-*f*[*1600*] which is made up of 24 rounds. In resource constrained environments, the smaller permutations can be used, though these have not been supported by NIST for use with cryptographic hash functions because of the lower security claim.

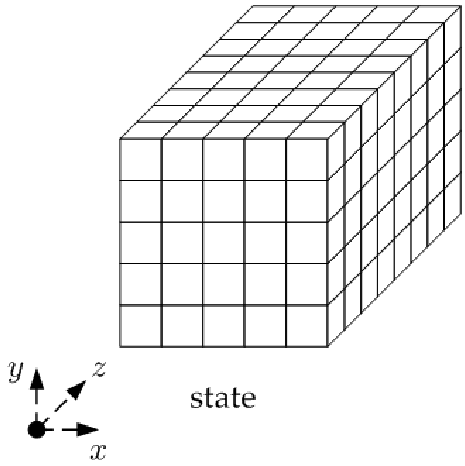


Figure 2.4: The Keccak state. / CC BY 3.0 @noekeon.org

The Keccak state is divided into smaller parts, as illustrated in Figure 2.5, where each square represents a bit. The inner and outer state r and c of the sponge function, depicted on the left side of Figure 2.3, are distributed lane-wise in the 3-dimensional Keccak state where the inner state c is located in the lower lanes.

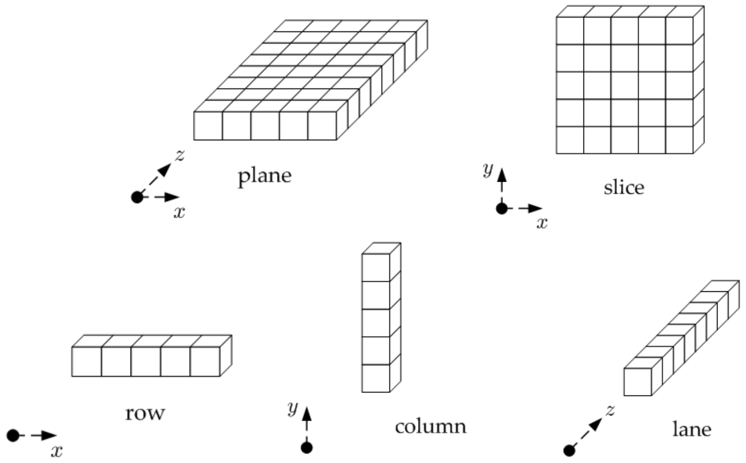


Figure 2.5: The structures withing the Keccak state. / CC BY 3.0 @noekeon.org

The input messages fill up the outer state by lanes. The relationship between the 3-dimensional state coordinates and the lanes of the inner and outer state is: $state(x)(y)(z) = r||c((64 \times 5 \times y) + (64 \times x) + z)$, where the padded r-bit input message is concatenated with c. The state is often referred to by the coordinates of each lane, as presented in Table 2.1. The left side coordinates are used in pseudo-codes explaining how the sub-steps of the Keccak function acts on the state. In programmable logic, 2-dimensional arrays are used and so the numbering of the right side is used in descriptions related with the implementation. The coordinates of the left-side matrix dictate the start and end of the lanes of the inner and outer state, i.e. the first lane of the outer state is located at [0,0], the second at [1,0] etc. and is concatenated by the inner state. In accordance with the left-side coordinates, the first lane of the message block is therefore number 12 and the following sequence continues the outer state: 13, 14, 10, 11, 17, 18 etc.

	X=3	**X=4**	**X=0**	**X=1**	**X=2**
Y=2	[3,2]	[4,2]	[0,2]	[1,2]	[2,2]
Y=1	[3,1]	[4,1]	[0,1]	[1,1]	[2,1]
Y=0	[3,0]	[4,0]	[0,0]	[1,0]	[2,0]
Y=4	[3,4]	[4,4]	[0,4]	[1,4]	[2,4]
Y=3	[3,3]	[4,3]	[0,3]	[1,3]	[2,3]

20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

Table 2.1: Keccak lane coordinates.

The Keccak round function Keccak-*f* consists of 5 unique step mappings [23] which are performed in sequence in each of the 24 rounds, as depicted in the pseudo-code of Figure 2.6. Theta (θ) provides diffusion by XORing columns into bits. Rho (ρ) and Pi (π) are pure permutations aimed at dispersion. Chi (χ) is the non-linear step mapping with an AND, NOT and XOR logic operator for each bit. The purpose of the last step mapping, Iota (ι), is to diversify each round by adding round dependent constants to the center lane $[0, 0]$. RC denotes the round constants of the ι step-mapping.

```
for i in 0 to 23{
    A'= roundfunction(A', RC(i))
} return A'

roundfunction(A', RC){
    B=$\theta$(A)
    C=$\rho$(B)
    D=$\pi$(C)
    E=$\chi$(D)
    A'=$\iota$(E)
} return A'
```

Figure 2.6: Pseudo-code of the Keccak sponge function acting on the state containing one padded message block.

For detailed explanations involving the implementation of Keccak in hardware, the state is presented in 2-dimensions instead of the 3-dimensional array, as depicted in 2.7. The x- and y-coordinates of the 3-dimensional array are mapped along the vertical LANE column and the z-coordinates along the SLICES row.

		SLICE																		
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	...	**55**	**56**	**57**	**58**	**59**	**60**	**61**	**62**	**63**
LANE	**24**	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	23	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	22	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	...					...					...					...				
	3	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	2	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	1	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63
	0	0	1	2	3	4	5	6	7	8	...	55	56	57	58	59	60	61	62	63

Figure 2.7: State representation in 2-dimensions. SLICES represent the z-axis while LANES the x and y axes.

Theta

The θ step mapping provides diffusion by XORing the parities of two adjacent columns with each bit of a third column, as depicted in Figure 2.8. The sizes of the state in the x and y axes are 5 bits and 64 bits in the z axis. The figure is simplified by shortening the lanes (z-axis). The three sub-steps are respectively:

1. θ_1: The parity of each column is generated, producing a plane of 320 bits.
2. θ_2: A second plane is generated where each [x][z] bit is a product of XORing every $[x-1][z]$ and $[x+1][z-1]$ bit of the first plane.
3. θ_3: Each bit of the second plane is XORed with each bit of the corresponding column of the original state matrix.

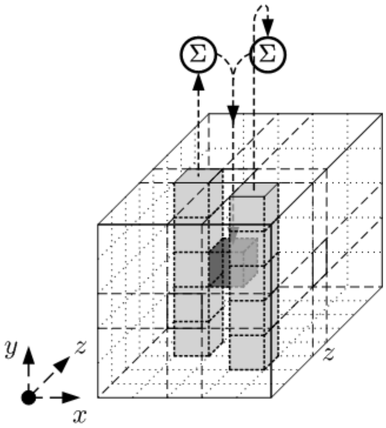


Figure 2.8: The θ step mapping illustrated. / CC BY 3.0 @noekeon.org

θ is the first step mapping since it efficiently mixes the inner and outer state which are separated across columns and rows. As depicted in Figure 2.9, each bit at the output depends on two columns at adjacent X and Z coordinates as well as the equivalent input bit. The colors indicate the corresponding input and output bits. In a straight forward hardware implementation, θ requires 1280+320+1600 XOR logic operators.

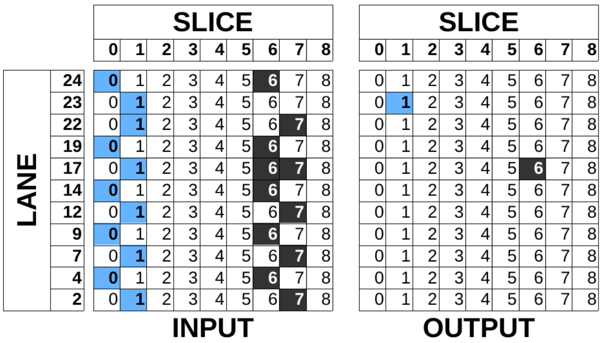


Figure 2.9: The input dependencies for every output of the θ step mapping. The output color matches the corresponding input.

Rho

The second step mapping provides inter-slice dispersion by rotating each individual lane by a fixed offset given by Table 2.2 as illustrated in Figure 2.10. As this is a permutation there are several potential solutions for implementing this in hardware.

	X=3	X=4	X=0	X=1	X=2
Y=2	25	39	3	10	43
Y=1	55	20	36	44	6
Y=0	28	27	0	1	62
Y=4	56	14	18	2	61
Y=3	21	8	41	45	15

Table 2.2: ρ rotational offsets for each lane.

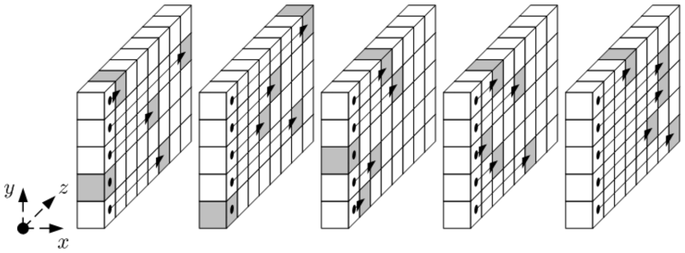


Figure 2.10: The ρ step mapping illustrated. / CC BY 3.0 @noekeon.org.

Pi

π disturbs horizontal and vertical alignment by shuffling the position of the lanes, turning rows into columns. The permutation is depicted in Equation 2.1 and in Figure 2.11. This step mapping is usually combined with ρ as they both are the only permutation steps of the Keccak round function and can be solved in hardware by wiring or addressing.

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} \tag{2.1}$$

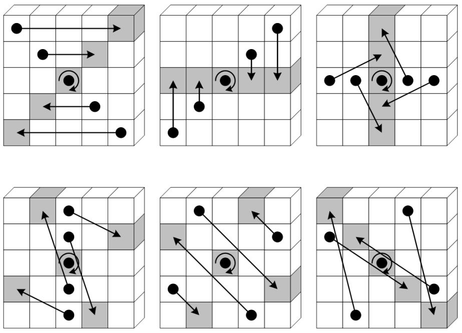


Figure 2.11: The π step mapping illustrated. / CC BY 3.0 @noekeon.org.

Chi

χ is the non-linear transformation step of the round function. The step mapping operates in a row-wise manner as illustrated in Figure 2.12. One NOT, AND and XOR is required per bit of the state and the dependencies are three input bits for each output bit.

Iota

The ι step-mapping XORs the center lane with round dependent constants in order to break symmetry along the z-axis and diversity the rounds. The round constants (RCs) for the complete center lane are listed in 2.1. Only 7 of the 64 bits are actual round constants and are the product of an 8-bit Linear Feedback Shift Register (LFSR) with polynomials of the form $P(x) = 1 + x^4 + x^5 + x^6 + x^8$. They can therefore be pre-generated and stored as a 7x24 bit Read only Memory (ROM) or generated on-the-fly by a LFSR.

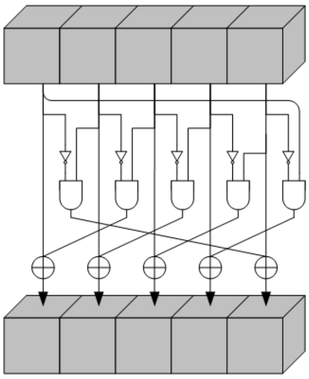


Figure 2.12: The Keccak state. / CC BY 3.0 @noekeon.org.

```
1: X"0000000000000001"  9: X"000000000000008A" 17: X"8000000000008002"
2: X"0000000000008082" 10: X"0000000000000088" 18: X"8000000000000080"
3: X"800000000000808A" 11: X"0000000080008009" 19: X"000000000000800A"
4: X"8000000080008000" 12: X"000000008000000A" 20: X"800000008000000A"
5: X"000000000000808B" 13: X"000000008000808B" 21: X"8000000080008081"
6: X"0000000080000001" 14: X"800000000000008B" 22: X"8000000000008080"
7: X"8000000080008081" 15: X"8000000000008089" 23: X"0000000080000001"
8: X"8000000000008009" 16: X"8000000000008003" 24: X"8000000080008008"
```

Listing 2.1: The 7 bits round constants for the Iota step-mapping distributed along the complete center lane of 64 bits.

Seven XORs are required for this step-mapping at coordinates Z=0, 1, 3, 7, 15, 31 and 63 at X=Y=0. This step can easily be combined with χ because of its simplicity. Figure 2.13 depicts the simple dependencies for this step-mapping.

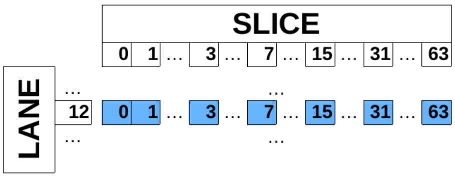


Figure 2.13: Input dependencies for the ι step mapping.

2.4 The SHA-3 standard

Currently, four versions of the Keccak sponge function family have been approved for the SHA-3 hash function standard: SHA3-224, SHA3-256, SHA3-384 and SHA3-512 [24]. The numbers indicate the length ℓ of the digest. All versions use the largest permutation of Keccak-*f*[*b*] where $b = 1600$. The security claim depends on the size of the inner state c and its relation to ℓ by $c \geq \ell \times 2$, making the $c - r$ relation different between the versions [25]:

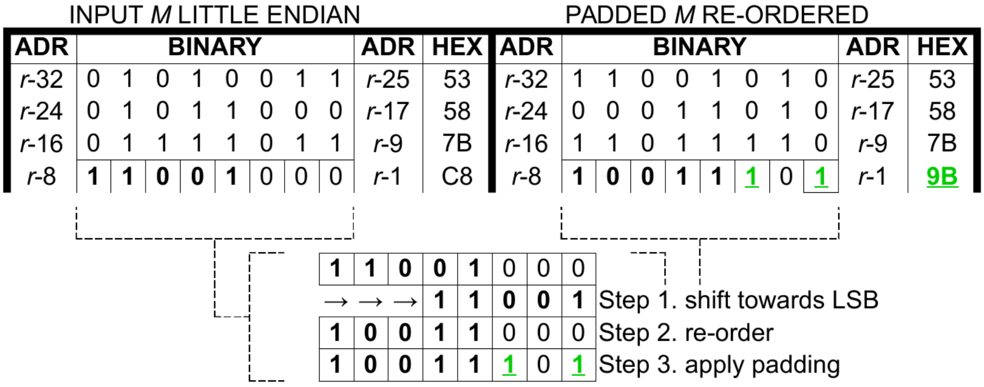
- $\ell = 224 : r = 1152, c = 448, \quad d = 28$
- $\ell = 256 : r = 1088, c = 512, \quad d = 32$
- $\ell = 384 : r = 832, \quad c = 768, \quad d = 48$
- $\ell = 512 : r = 576, \quad c = 1024, d = 64$

Any further reference to versions of SHA-3 and Keccak will concern these four specifications. In addition to the parameters already introduced, there is a diversifier d which exists to meet a NIST requirement stating that the output of a given hash function should not be the prefix of a larger version. d is simply the number of bytes to which the output is truncated to. Additionally, it allows for domain separation in the use of a specific SHA-3 version. For custom use of Keccak beyond the SHA-3 standard, users are free to set the diversifier for a given application as long as $d < \ell/8$. The default size of r is then 1024.

Message padding

The keccak sponge functions adopted a multi-rate padding scheme which practically means that the same padding rule works for all variations of the aforementioned parameters. The input message is shaped in order to fit within a block or multiples of it. The functionality of the padding rule meets another requirement related with the security claim which is that the last word (lane filled with data) within a block is not empty. The padding, which is denoted as **pad10*1**, appends a 1-bit after the last bit of the last byte containing data and a last 1-bit at the MSB of the last byte of the block.

The internal Keccak convention for bit ordering is little-endian, so that the LSBs within a byte are located at the lower address [26]. The NIST specified SHA-3 API [27] is different however, and uses a big-endian convention so that the MSB of each byte is located at the lower address. The consequence of this is that an incoming big-endian message, imposes a re-ordering of each byte. Figure 2.14 depicts the re-ordering and the padding for a case where $|M| = |r - 3|$ and the padding starts and ends at the last byte. As can be seen, the bits of the last byte are shifted towards the LSB of its big-endian order so that the zeros are contained towards the MSB of the little-endian internal Keccak order.



INPUT *M* LITTLE ENDIAN

ADR	BINARY								ADR	HEX
r-32	0	1	0	1	0	0	1	1	*r*-25	53
r-24	0	1	0	1	1	0	0	0	*r*-17	58
r-16	0	1	1	1	1	0	1	1	*r*-9	7B
r-8	**1**	**1**	**0**	**0**	**1**	0	0	0	*r*-1	C8

PADDED *M* RE-ORDERED

ADR	BINARY								ADR	HEX
r-32	1	1	0	0	1	0	1	0	*r*-25	53
r-24	0	0	0	1	1	0	1	0	*r*-17	58
r-16	1	1	0	1	1	1	1	0	*r*-9	7B
r-8	**1**	**0**	**0**	**1**	**1**	1	0	1	*r*-1	**9B**

1	**1**	**0**	**0**	**1**	0	0	0	
→	→	→	**1**	**1**	**0**	**0**	**1**	Step 1. shift towards LSB
1	**0**	**0**	**1**	**1**	0	0	0	Step 2. re-order
1	**0**	**0**	**1**	**1**	1	0	1	Step 3. apply padding

Figure 2.14: Padding and bit-wise reordering of a little-endian input message.

There are three possible scenarios related with the padding, where x is the absolute value of the difference between the message length, $|M|$, and the block size, r (i.e. $x = ||M| - r|$):

- $|M| = r + x$: The message will fill two blocks and the padding appends to the second block a 1-bit at the LSB followed by $r - x - 2$ 0-bits before a 1-bit at the MSB.
- $|M| = r$: The message will only fill one block and the padding appends a 1-bit to the MSB.
- $|M| = r - x$: The message will only fill one block and the padding appends a 1-bit at $r - x + 1$ followed by $x - 2$ before the last 1-bit at the MSB.

2.5 Summary

This chapter has presented the concept of cryptographic hash functions, the SHA-3 cryptographic hash function standard, and has also given an introduction to hardware co-processors and FPGAs. SHA-3 describes the next generation of hash functions and is expected to replace the now commonly used SHA-2 and SHA-1. The underlying algorithm is Keccak and four sub-versions are currently included in the standard. The algorithm consists of a function of 5 unique step-mappings which acts on a state over 24 rounds. The sub-versions of SHA-3 have no structural distinguishing features and differ by the ratio of the size between the inner and outer part of the state which further determines the size of the message block and the digest. The implementation technology of focus is the FPGA where the Xilinx Virtex-5 is selected as a performance standard. FPGAs are chosen because of their advantageous prototyping capabilities, good performance and prevalence among the state-of-the-art.

3 State of the art

From the early stages of the selection process for the new SHA-3 standard, several papers were published comparing implementations of the different candidates. These works assisted in demonstrating the varying performances between the potential winners as the competition progressed, highlighting their strengths and weaknesses related with various factors. As the competition ended and ensuing the announcement of the winner, increasingly improved works emerged, focusing more on structural optimization techniques and less on plain comparisons. These papers define the state of the art related with implementations of the four sub-versions of the SHA-3 hash function. The various approaches and implementation techniques are herein presented. There is much to learn from the mistakes and successes of these proposals.

There is a loose convention of classifying the implementations based on the internal data path, as presented in the Keccak Implementation Overview [12]. The three different classes are: compact, mid-range and high-speed designs. In this paper, the latter are simply referred to as unfolded designs while the others are sub-classes of folded designs with constrained data paths where only parts of the state are processed. The reasoning behind the classes is that the performances are distinctively different. As further explored in Chapter 4, smaller area footprints come with a greater cost in throughput. The choice of implementing a design within a class should be motivated by a specific objective, e.g. a compact folded-structure should be aimed at consuming an absolute minimum number of resources at the lowest loss in throughput. An unfolded structure on the other hand, should aim for maximum throughput and decreasing the area is a secondary objective.

A complete design conventionally involves a wrapper, the state, and the Keccak-*f*. The separation of these three components are naturally blurred among the existing literature and are presented here first and foremost for the benefit of distinguishing the functionalities. They remain equally separated during analysis and development in this thesis. It is uncertain whether other designs incorporate the same convention. The purpose of the wrapper is first of all to provide a hardware interface for managing the interconnection between the FPGA and the user/processor. For hardware implementations of SHA-3, this includes the reception and preparation of the message block so that it is compatible with Keccak-*f* as well as the transmission of the final digest. Additionally, the wrapper incorporates the functionality of the sponge construction, specifically the iterated calling of the sponge function (Keccak-*f*) for message with multiple blocks of length. A simplified schematic of a basic structured SHA-3 implementation is depicted in Figure 3.1. The color blue covers the state and the round function, Keccak-*f*, while the wrapper part is marked in red. The figure does not include how the state is initially composed of partly a message block and zeros at respective lanes, and that the output of the hash function is similarly composed of only a sub-set of lanes.

The round function for SHA-3 is composed of multiple identical rounds of relatively simple logic, and so every optimization which simplifies this component can easily have a solid impact on the overall performance. On the other hand, the relatively large state of 1600 bits results in a wide round function implying that routing is expected to be a big factor related with timing.

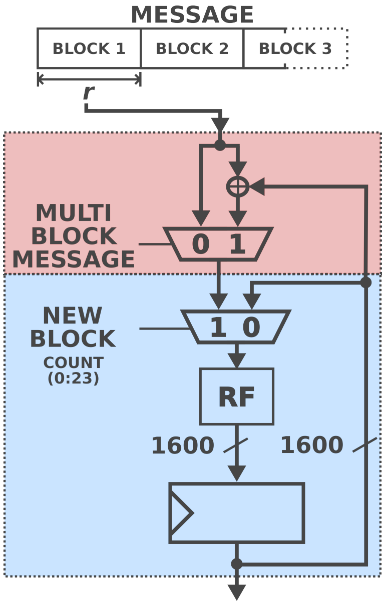


Figure 3.1: Simple illustration of SHA-3 with the sponge construction in red at the top and the round function Keccak-*f* and the state in blue below.

One of the main objectives of this thesis is to arrive at a design with improved efficiency (i.e. close to the maximum throughput per area). Throughput is defined as the rate at which messages are injected into the hash function and processed while area is the overall resource occupation. The usual trade-off between these two metrics is reflected in the structural design choices in the related state-of-the-art as herein discussed. It is apparent that economizing in resources results in decreased performance for most types of constructions and the challenge of a designer is to find the optimal trade-off. The concept of economizing in area is motivated by the potential for either combining the hash function with other components on the FPGA or simply cascading the structure for multiplied throughput. If the latter is relevant, it is crucial to maintain throughput. While there are many approaches towards optimizing hardware designs, low-level optimizations such as vendor-specific issues or manual component placement are herein not considered as these are out of the scope of this thesis. These low-level optimizations can improve the performance related with timing considerably. However, optimal Place and Route (PAR) is categorized as an NP-hard problem. Therefore, only high-level optimizations have been considered in this thesis. These comprise structural aspects of a design and are generally independent of the FPGA vendor and model, making the task of porting the design as transparent as possible. Each output of the round function depends on a set of input bits which dictates how many bits that can be processed in parallel.

The basis for the reported performance of individual designs is not equal. While area consumption remains mostly the same, synthesis results of clock frequency are vastly different from those which are obtained from PAR. It is therefore interesting to personally evaluate which performances that can be reached by replicating the literature. It is also uncertain how much effort has been put into the existing designs with respect to manually placing components on the FPGA and tool optimizations. This is a time consuming task, but one that can yield good results in frequency if done properly. San & At [28] mentions careful placing and routing of components and reports close to optimal frequency of their design. Others have reportedly utilized tools such as ATHENa [29] to iteratively search for the optimal synthesis and PAR options. It is up for discussion whether it is a reasonable approach to rely on manual methods or specialized tools instead of the standard vendor-provided software, when comparing proposed solutions. However, all affordable measures should be considered for a final and physical implementation of a hash function.

The main concepts of optimization techniques which are presented here have been considered and utilized by the existing literature. Other techniques exist which in general have been considered as irrelevant for SHA-3 because of the construction of the sponge function. Literature performing comparisons of the early SHA-3 candidates or covering hardware implementations of other cryptographic functions have served as helpful resources related with potential optimization techniques, such as Nachvatal *et al.* [30], Gaj *et al.* [4] and Lien [31].

3.1 Folding

To economize on the utilization of FPGA resources, the logic comprising the design can be re-used. This is achieved by folding the state so that only a sub-set of the 1600 bits are processed on each iteration. With respect to the standard structure of Keccak, this means that partitions of the state are processed in the round function. The concept of folding is depicted in Figure 3.2, where the folded structure, on the right, has an internal data path smaller by a factor equal to the folding factor (FF), at a proportional increase in the clock cycles.

While the state can be fragmented into any of its smaller components, processing too few bits in a combinational path is not beneficial. As explained in detail in Chapter 4, each output bit of the round function is dependent on multiple input bits and extra clock cycles and logic is required if these dependencies are not met. Thus, folds smaller than a slice or a lane are not considered to be effective.

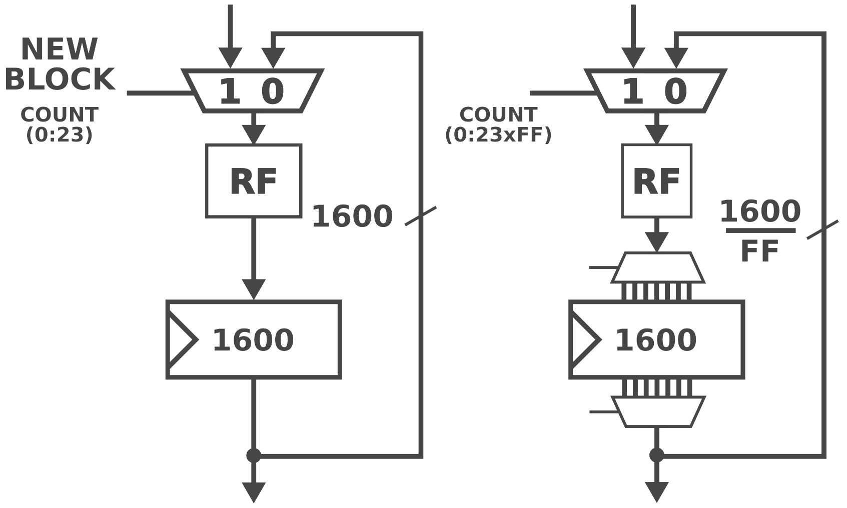


Figure 3.2: Simple schematic of a basic SHA-3 structure on the left and a folded structure on the right. FF=folding factor.

The relevant existing implementations have either incorporated a lane-wise or slice-wise folding-scheme. The first folded structure was suggested by Bertoni *et al.* [32] in the earliest version of the Keccak Implementation Overview in 2011. This structure incorporated a lane-wise folding-scheme so that the internal data-path is 64 bits wide. It relies on embedded memory for storage of the state. Each round of Keccak-*f* takes 215 cycles and contain 55 bubbles, which are clock cycles where no computation is performed. This results in a poor throughput and overall efficiency which represent both the specific downside of the folding-scheme and folding in general. E.g. the θ step-mapping can not be completed before 9 lanes have been read from memory. Intricate scheduling is also required in order to maintain a non-excessive latency and acceptable throughput.

Kerckhof *et al.* [33] improves upon the official compact lane-wise design through more efficient scheduling. The total latency is less than 50% of its predecessor with a total of 2154 clock cycles and the area is similarly reduced. They do not use any additional FPGA resources other than the slices, using distributed RAM to store both the state, round constants, and intermediate round function values The overall efficiency of this structure is 0.47 Mbps/slice on a Virtex-6 FPGA. San & At [28] improve upon the lane-wise architecture by introducing a fine-tuned instruction sequence which allows for high concurrency along a serialized round function. The latency is further reduced by 50% and along with an optimal frequency close to the upper limit of the Virtex-5 FPGA, this implementation yields a high throughput compared to the other compact folded designs and an efficiency of 1.66 Mbps/slice. BRAMs are utilized in this design to store the state and other necessary data. The authors in this paper do not include the BRAMs in the estimation of the efficiency which gives an improper view of the resource utilization.

Jungk & Apfelbeck [34] presents the first slice-wise structure which requires the re-scheduling of the round function to solve the dependencies of the ρ step-mapping. These dependencies are further explained in Chapter 4. The round re-scheduling ensures that the ρ and π step-mappings are the last steps in all but the first and last round. This results in 3 different rounds, but the dependency problem of the permutation steps is avoided. One remaining dependency issue is caused by θ as the column-parity of a slice at a lower address is required for each output. Slice number 0 is dependent on the processing of slice number 63. The paper does not address how this dependency is solved. A problem with slice-wise folding-schemes is related with how the message block is conventionally split into lanes and transmitted and received per 64 bits. These must be split up in a slice-wise folding-scheme. As a message lane is received, FF clock cycles are required to load 64 bits with the conventional interfacing. In this paper, they solve this with a Fast Simplex Link (FSL) which is a hardware interface developed by Xilinx [35]. The solution is to convert the standard interfacing to a slice-wise orientation by buffering the incoming message 1600 bit/FF per clock cycle. This interface is neglected from the area and throughput assessment, nevertheless, they have included a padding functionality which is presented and included in the assessment. Their structure has a folding factor of 8 and latency of 200 clock cycles. The state is stored in distributed RAM which can be read asynchronously. This is the first paper with an implementation in the mid-range class of folded structures, meaning that the throughput is improved compared with other compact designs at a smaller cost in area [33, 28, 12, 36]. The overall efficiency is 1.17 Mbps/slice. A later paper by Jungk *et al.* [37] perform a more elaborate exploration of various folding-factors of the slice-wise folding-scheme, thus demonstrating the flexibility of this approach. The first compact structure with a slice-wise folding-scheme is presented. This paper also addresses the intra-round dependency caused by θ which is solved by the addition of extra logic so that slice number 0 is processed along with slice number 63 during the last sub-round.

B. Jungk [38] performs a thorough analysis of the possible ways in which the state can be folded and the resulting challenges related with each case. Though most folded structures have either incorporated a slice-wise or lane-wise folding scheme, the former is suggested to be the optimal choice for efficiency and flexibility with respect to the folding factor.

Another compact slice-wise structure is proposed by Winderickx *et al.* [36] which utilizes Shift Register Lookup Tables (SRLs) of the SLICEM type of slices to store the state. The SRL is beneficial as it inherently solves the ρ dependency. This is similar to storing the state in distributed RAM, albeit more efficient for compact structures with a maximum FF. The authors mention that they do not solve the interfacing mismatch which is inherent for slice-wise folded structures. The overall efficiency is 1.16 Mbps/slice. Jungk & Stöttinger [39] proposes a compact slice-wise structure which stores the state in distributed RAM, incorporating an interface similar to the one used by Jungk & Apfelbeck.

Other approaches to folding exists. Gaj *et al.* [4] presents vertical and horizontal folding where the former is equal to what is herein considered. Horizontal folding can be described as a form of unrolling which is only relevant for a symmetric round function, i.e. a function consisting of two or more identical sub-functions. The complete function can then be reduced to only one of the sub-functions with a multiplication of the number of cycles. Throughput is therefore traded for area.

3.2 Pipelining

Pipelining is a common technique for adding parallelism to combinational logic by dividing the data path with intermediate registers. The most critical path of a design dictates the minimum clock period of the design. Incorporating pipeline registers along this data path will therefore result in an increased frequency of the overall design. The penalty of using this technique is a larger design because of the added registers, as depicted in Figure 3.3. Still, the overall throughput per area is generally improved with well-balanced employment of each pipeline. Latency is increased proportionally with each introduced pipeline register, but this is equally compensated by the added messages contained in each pipeline. A structure is herein considered to be pipelined if more than one iteration of the state register is incorporated in the design. The degree of pipelining is denoted by PL and the default pipelining factor is therefore PL=1.

A precondition for the benefit of this technique is that all pipelines are full and that no bubbles are contained in-between rounds. The aptitude of pipeline registers for folded structures relies on the data dependencies between the combinational logic, 4.2. For unfolded designs, pipelining is only relevant if the hash function is used to process multiple messages.

As described in 2.3.1, larger messages invoke an iterated absorbing phase of the sponge construction, i.e. each block constituting a part of a message is XORed into the state with the previously processed blocks of the same message. Each block must undergo 24 rounds of the round function before being merged with the subsequent block. Thus, two blocks of the same message cannot consecutively fill a pipeline.

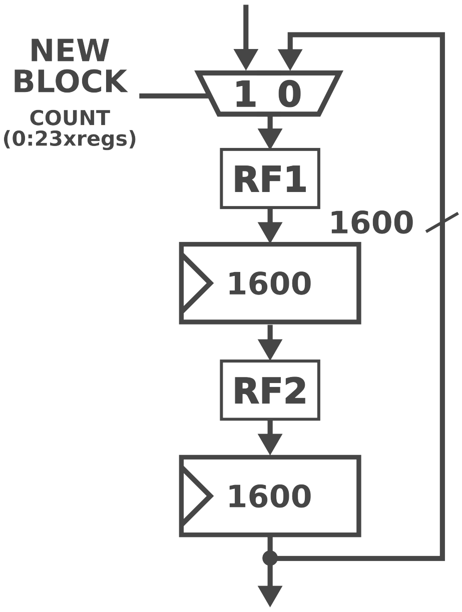


Figure 3.3: Simple illustration of the concept of pipelining. The round function RF is split into RF1 and RF2 while maintaining its width.

Some of the existing literature considers scenarios where multiple small messages are hashed, coining the term Multi-Message Hashing (MMH). Akin *et al.* [40] performed an early exploration of multiple pipelines within the round function. While they report an impressive operating frequency of 509 MHz and manage to increase the overall efficiency, the area is almost proportionally increased as they have incorporated five internal pipelines. What must also be noted is that their design is implemented on a Virtex-4 which has less powerful slices and a larger routing delay compared to V-5. Pereira *et al.* [41] made a thorough analysis of the total delay provided by each step mapping of the round function on a V-5. Based on this, they propose a different pipeline-scheme than Akin *et al.* [40] by distributing the step mappings in a different manner among three internal pipelines. The overall efficiency is increased as the reported frequency of 452 MHz is not much lower than the related state-of-the-art. Similar results were achieved by Ayusawa *et al.* [42]. The design by Athanasiou *et al.* [43] reports a frequency of 389 MHz with only one internal pipeline register, thus achieving a high efficiency.

The literature differs in the expectation of the messages. A common application for hash functions is the hashing of Ethernet packages and so messages should be expected to have sizes larger than the block size, i.e. multi-block-messages (MBM). For this reason, as is also pointed out by Ioannou *et al.* [44], pipelining the round function in unfolded structures should be avoided if this scenario is to be considered.

In most variations of folded designs, pipelining is not efficient as inter-round dependencies cause bubbles or cycles with empty pipeline registers at the transition of each round. The processing of the first folds of a round x depends on the last folds of round x-1. A technique which avoids the inter-round dependencies was suggested by B. Jungk [38] and explored by Jungk & Stöttinger [39]. This technique allows for up to 9 pipeline registers within the round function by re-scheduling the sequence of step mappings and exploiting the permutation of the ρ step. In the re-scheduled round, the ρ and π step mappings are located last and the structure is folded slice-wise, processing 1 slice in each combinational path. If the round function contains one internal pipeline register to process two slices concurrently, then slice 0 of round x+1 cannot enter the round function before it has been updated by slice 63 of round x. ρ only updates 25 different slices from each slice processed, leaving only $64 - 25 = 39$ slices that does not depend on the last slice. These dependency-free slices can therefore be the first slices of the next round. If they are consecutively located, then a larger quantity of pipeline registers can be incorporated in the round function. This is further explained in Chapter 4

Pipelines can be useful in other points of the design. In particular cases, control signals can impose the critical path. An example of this is if the round constants are stored in an embedded memory with significant routing delay from the location of the memory to where it is used in the ι step mapping. While BRAM in modern FPGAs contain built-in pipeline registers, it might not be sufficient in designs with higher working frequencies. For a selection of different structures with various folding-schemes, Ayuzawa *et al.* [42] have explored this optimization which they have named *Retiming of round constant preparation*. In their paper they mainly compare the design of Akin *et al.* [40] and their modifications of it with the unfolded Keccak reference design [12]. Their main conclusion is first off all that the five pipeline registers employed by in [40] within the round function are excessive, but also that pipelining the path of the round constants improves the overall performance for pipelined designs. They report a 52% improvement in efficiency over the pipelined structure by Akin *et al.* [40].

3.3 Unrolling

Loop unrolling trades lower latency for higher area requirements as multiple round functions are located in the combinational path and processed within a single clock cycle, as is depicted in Figure 3.4. For Keccak, the critical path in the basic structure is located through the round function. Unrolling will therefore decrease the frequency and throughput. Thus, the existing literature has mostly found this optimization technique to be relevant for other hash functions and not for Keccak. Bertoni *et al.* [12] performed an early analysis of the effect of purely unrolling the high-speed unfolded structure. The results from the simulations are presented in Table 3.1 with area measured in gates and a quantity of instances of the round function from one to six, given by the UF. Each incrementation of the unrolling factor (UF) decreases the latency by 50%, but it is not enough to compensate for the lower frequency and higher area requirements. As can be seen, the efficiency which is given in the right-most column denoted by T/A decreases as UF increases towards the lower rows.

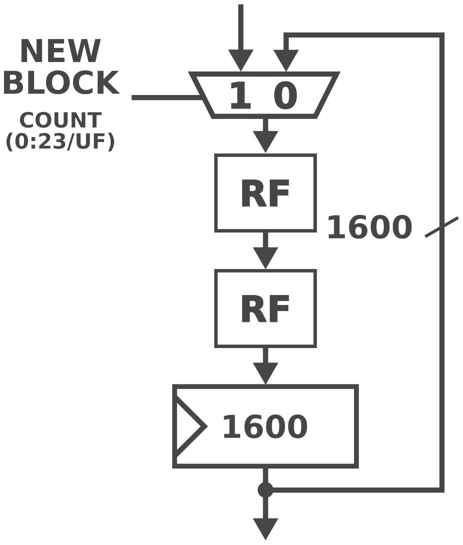


Figure 3.4: Simple illustration of the concept of unrolling. The round function is duplicated within a combinational path and the number of rounds reduced proportionally.

Unrolling factor	Area	Frequency	Critical path	Throughput	T/A
1	48 kgates	526 Mhz	1.9 ns	22.44 Gb/s	468 Kb/gs
2	67 kgates	333 Mhz	3.0 ns	28.44 Gb/s	425 Kb/gs
3	86 kgates	244 Mhz	4.1 ns	31.22 Gb/s	363 Kb/gs
4	105 kgates	192 Mhz	5.2 ns	32.82 Gb/s	313 Kb/gs
6	143 kgates	135 Mhz	6.3 ns	34.59 Gb/s	242 Kb/gs

Table 3.1: Unrolling results based on the reference architecture / The Keccak team [12].

Ioannou *et al.* [44] propose an unrolled architecture which is pipelined, however, only multiple independent messages are considered for hashing as the structure incorporates an external pipeline. Therefore, the round function remains in the critical path, but it is not increased as was seen in the analysis by Bertoni *et al.* [12]. One message block is processed 12 times in the first instance of the round function before proceeding to the second instance. The latency is reduced by 50%, the area increased by less than 100% and the frequency remains the same. The reported results are noticeably better than a basic structure, despite no internal pipelines. This is the only known consideration of an unrolled structure apart from the official Keccak literature.

3.4 Manual component instantiation

This technique borders low-level optimizations as it improves the non-optimal utilization of components on the FPGAs by the synthesis tool. Nevertheless, it is still herein included as it seems to be a relatively simple and efficient improvement that is relevant for the implementations of SHA-3. The downside of this optimization is that porting might not be straightforward between significantly different FPGA families such as across vendors.

Only one case is known among the existing literature where manual instantiation has been utilized to improve a SHA-3 implementation. It was suggested by Jungk *et al.* [37] that a manual LUT instantiation of the χ and ι step mappings would reduce the required number of LUTs and additionally save multiplexers. This specific optimization is relevant for implementations on V-5 FPGAs and newer families with 6-input LUTs. Multiplexers are saved as there are enough inputs for one of them to assert whether the step mappings should be bypassed, such as in round 0. This technique has been implemented by Jungk & Stöttinger [39], however the actual impact is not given and whether it is effective for other parts of the implementation is not clear. In general, it is not easy to determine precisely in which situations this optimization is relevant. No specific literature is found to be exploring the degree in which manual component instantiation is effective, depending on the given logic and synthesis tool.

There are many situations where knowing what the synthesis tool will implement is important in order to achieve the desired optimal performance. One example of this is when Multiplexers (MUXs) are described which are not a power of two [45], as is the case for many parts of a SHA-3 implementation. E.g. a 5-input MUX can be implemented by four 2-input MUXs or one with 5- or 6-inputs. As large MUXs are implemented with a combination of LUTs and 2:1 MUXs, the difference between the possible solutions is seen when the implementations are cascaded for large signals, as depicted in Figure 3.5.

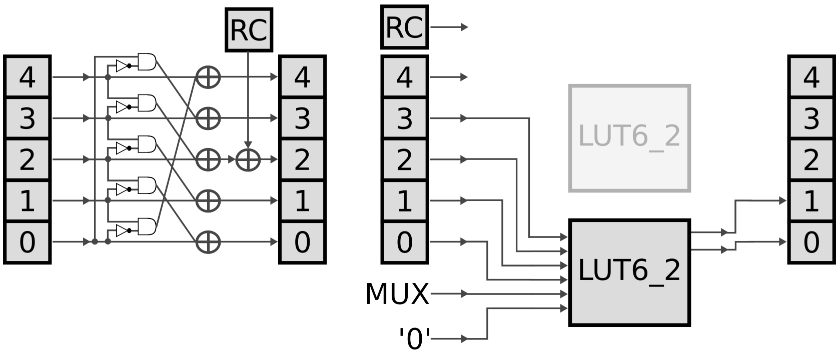


Figure 3.5: Illustration of manual LUT instantiation of the χ and ι step mapping acting on a row of the state matrix. Three Virtex-5 LUTs are required per row.

3.5 Accessory FPGA resources

Designs implemented on FPGAs should take advantage of all the resources available. However, the existing unfolded implementations rarely utilize more than slices. While the performance of components such as the DSP slices and BRAM have improved steadily throughout the years, the path between these blocks and the general logic often result in high routing delays. The challenge is to properly use these resources in order to either reduce the critical path and/or the total area requirements. The downside of using additional resources is the awkward comparability with other existing designs.

Xilinx FPGAs provide dedicated DSP slices for mathematical calculations. While only simple boolean operators are involved in Keccak-*f*, the DSP48E slice, provided by the V-5, supports 3-input 48-bit adders [46]. Without carry-in the adder works with GF(2), thus functioning as a boolean XOR operator. Ayuzawa *et a.l* [42] have explored the utilization of DSP slices for pipelined SHA-3 designs, by implementing two of the three sub-steps of θ, θ_2 and θ_3 with the DSP48E slices. They find that for certain pipeline schemes, DSP slices improve the overall performance of the design. No additional literature is found to be considering the usage of DSP slices in SHA-3, other than this paper.

θ_2 involves the XORing of each bit of a 320-bit plane, thus requiring seven DSP48E slices. Five times more DSP48E slices are required for the θ_3 sub-step. The number of available DSP slices vary greatly between models and as the V-5 provides between 32 and 1056 DSP slices, the total number of available slices can quickly be consumed by incorporating any of the wide step-mappings of Keccak-*f*. Additional usage of DSP slices can also be considered in the absorbing phase of the sponge function.

Embedded memory blocks which for FPGAs correspond to BRAM, can be used to store larger quantities of data than distributed RAM. The usage of BRAM to save slices emerges in folded designs because all bits are not accessed in the same cycle. Two aspects should be considered with the use of BRAM as these are dimensioned for larger quantities of data than the 1600 bits of the state and that read and write operations are synchronous. The former means that in case the FPGA is implemented with other functions in addition to the SHA-3 structure so that resources must be shared, large amounts of embedded memory is already consumed and unavailable for utilization by the other functions. This concept is herein referred to as *proportional resource utilization*. The issue with synchronous read and write operations can be bypassed by exploiting a collision-avoidance mechanism incorporated in these embedded memories in modern FPGAs.

Distributed RAM are more sorted for smaller quantities of data with asynchronous read operations. In each V-5 type SLICEM slice allow to implement 128 bits of distributed RAM. A V-5 FPGA with 4800 slices contains 320 kbits of distributed RAM. The ratio of SLICEM slices to the total number of slices is typically around 1:1.7.

San & At [28] have developed a coprocessor-based architecture for extremely resource constrained environments. In their design, two dual-port 36 Kbit BRAM units of Virtex-5 are used to store one instance of the state and additional temporary values. Each state is accessed lane-wise which results in a high number of cycles per round. This implementation achieves an optimal frequency of 520 MHz on the V-5 FPGA, but the high area constraint, results in a high latency of 1062 cycles, and a low throughput and overall efficiency.

The BRAM units in a V-5 can be cascaded to one 64 Kb RAM or configured as either one 36 Kb or two 18 Kb RAMs, both with Simple Dual Port (SDP) access [17]. They include built-in pipeline registers which are advantageous to reduce the slice consumption at a minimum and potentially to minimize the data path. With dual-port access the maximum data width for reading data is 72 bits. The challenging part of using BRAMs is to maintain parallelism so that a minimum of 5 SHA-3 lanes can be accessed simultaneously.

3.6 Wrapper

A wrapper is a common term for a component which handles the hardware interface of the design. Herein the additional control logic for the complete functionality of a design is also included in the wrapper. Since FPGAs contain a limited number of IO-ports, several cycles are involved in the reception and transmission of each message block and the digest. For compact designs, these cycles can be added to the 24 rounds of the round function in a trade-off for smaller area. The buffers are in that case removed and the message block is directly injected in and extracted from the state. For unfolded high-efficiency structures, on the other hand, this buffering is an important part of the functionality of the design.

A part of the existing literature have incorporated some sort of wrapping functionality which contains the IO-buffers, however, in some this is neglected. A selection of the state-of-the-art have also not included a wrapper at all and only the core functionality of Keccak is considered in the evaluation of the performance. Such structures can therefore be considered as incomplete and cannot function by themselves as a stand-alone entity.

A padding functionality can also be incorporated, but this is separated from the hardware design in most existing implementations and provided by assisting software. This is partly because of the earlier literature's focus on fair comparisons between the candidates where it was desired to avoid extra differentiating factors such as the individual padding rules. An exception to this is the paper by Baldwin *et al.* [47] which reports no decline in frequency while including a wrapper in the assessment of Keccak and only the area is distinctively larger. Ambarish Vyas [48] have studied hardware padders for several of the early SHA-3 candidates and reports that a relatively efficient padder component can be implemented for Keccak, based on priority encoders. His design is implemented on a V-5 and achieves a maximum frequency of 314 MHz with a footprint of 32 slices for the padder.

Athanasiou *et al.* [43] presents the first general SHA-3 implementation. Which SHA-3 version is supported is entirely dependent on the wrapper as it controls the block size and digest. Five XOR modules are used for version selection in this design. The round function is pipelined so that the achieved frequency is decent for scenarios where multiple small messages are hashed. No IO-buffer is included in the wrapper and the structure requires wide IO-ports. Another hardware interface titled GMU was developed by Gaj *et al.* [49] which is based on two Finite State Machines (FSMs) and passive First-In-First-Outs (FIFOs) for message input and digest output. This interface was developed during the early stages of the SHA-3 competition and is designed as a generic interface for hash functions. More recent implementations [33, 39] have considered this interface, some with slight modifications. In their approach, they assume that padding is performed externally by the user and only one of the SHA-3 sub-version is supported. Jungk & Apfelbeck [34] use the Fast Simplex Link (FSL) which is a 32 bit unidirectional link used between IP cores and microcontrollers [35].

3.7 Overall analysis

The relevant existing implementations are listed in Table 3.2, sorted by efficiency normalized for the SHA3-512 version. Because of the differences in the block size, the throughput and efficiency of a SHA3-256 structure is approximately 50% of an identical SHA3-512 structure. The presented structures in the state-of-the-art are implemented on Virtex-5 except those noted otherwise. UF(1-9) indicates the unrolling factor, FF(1-9) the folding factor, and PL(1-9)X/PL(1-9) indicates the number of pipeline stages and whether these are internal or e(x)ternal. A structure noted with PL2 incorporates one internal pipeline register in addition to the main state register. B (buffer) and NB (no buffer) implies whether an IO-buffer has been included in the assessment.

The existing structures report vastly different results and the comparison between them is not straight forward given the many factors involved. Moreover, some authors are presenting results obtained from synthesis only, which is highly erroneous with respect to timing. Frequency results which are based on synthesis are denoted by a † and frequencies where the authors fail to mention the source are denoted by ††. Akin *et al.* [40], Pereira *et al.* [41], Gaj *et al.* [4] and Jararweh *et al.* [50] seem to have based their performance on synthesis only. It is not clear which of the four sub-versions have been implemented by the work of Pereira [41]. Ayazuwa *et al.* [42] fail to present numbers and the paper only contains figures indicating their performance.

The structure presented by Akin *et al.* [40] is implemented on a Virtex-4 FPGA. The performance is expected to be better on a V-5 with smaller footprint and better timing. Kerckhof *et al.* [33] presents a Virtex-6 implementation and the performance should therefore be considered lower than what is presented when comparing with the other structures.

IO-buffers are usually not included. These buffers consume resources and the performance can be affected by whether these are included. For unfolded designs, minimal latency is key and so the round function should be able to run without interruption. An IO-buffer is therefore necessary to deliver the data in parallel to the round function and transmit the digest. Athanasiou *et al.*. presents a general wrapper, but without a buffer. This can explain the larger area compared with Ioannou *et al.* and other basic structures.

A general source of the deviation in the performance for all the structures is the frequency. As mentioned earlier, The NP-hard problem of optimal timing can be improved if sufficiently prioritized and many of the listed structures could potentially achieve better performances. For example, the basic structure of Ioannou *et al.* [44] achieves a comparable frequency to the internally pipelined structure by Athanasiou *et al.*. It is, on the other hand, clear that the performance of the latter is based on PAR while the basis of the former is unknown.

The added latencies of pipelined structures does not have an impact on the efficiency as these approaches are considering applications for the hash function where multiple small mes-

Paper	Lat. (clkc)	*f* (MHz)	A (slices)	T (Gbps)	T/A (Mbps/ slice)	Ver.	Note
Ioannou [44]	12	352 ††	2652	16.90	6.37	512	UF2.PL2X.NB
Ioannou [44]	24	382 ††	1581	9.17	5.79	512	NB
Athanasiou [43]	48	389	1702	18.70	10.98	224	PL2.NB
Gaj [4]	24	283 †	1272	12.82	10.08	256	B
Baldwin [47]	25	189	1117	8.50	4.32	512	NB
Jungk [37]	24	195	1305	8.49	3.87	256	NB
Pereira [41]	100	452 †	3117	7.70	2.47	?	PL4.B
Akin* [40]	121	509 †	4356	22.33	5.13	224	PL5.B
Baldwin [47]	25	189	1971	8.50	4.32	512	B
Jararweh [50]	24	271 †	2828	12.28	4.34	224	B
Akin* [40]	25	143 †	2024	6.07	3.00	224	B
San & At [28]	1062	520 ††	151	0.25	1.66	512	FF25.BRAM.B
Jungk [37]	50	144	914	3.13	2.04	256	FF2.NB
Jungk [37]	100	150	489	1.63	1.99	256	FF4.NB
Jungk [37]	200	166	301	0.90	1.79	256	FF8.NB
Jungk & Ap [34]	200	159	393	0.46	1.17	256	FF8.NB
Jungk & St [39]	1665	257	90	0.17	1.85	256	FF64.B
Winderickx [36]	1730	248 ††	134	0.25	1.16	256	FF64.B
Jungk [37]	1600	206	164	0.14	0.51	256	FF64.NB
Kerckhof** [33]	2154	250 ††	144	0.07	0.47	512	FF25.B
Bertoni [12]	5160	265	448	0.05	0.12	512	FF25.B

Table 3.2: Relevant existing unfolded (top) and folded (bottom) implementations sorted by efficiency adjusted for SHA3-512. FF(1-9)=folding factor, UF(1-9)=unrolling factor, PL(1-9)X/PL(1-9)= external/internal pipeline registers, NB/B=no buffer/buffer. *Virtex-4 implementation. **Virtex-6 implementation. † Results obtained from synthesis. †† Unknown source of results.

sages are processed and thus, no bubbles exist along the pipeline. Both 24 and 25 clock cycles are reported as the latency for unfolded structures and the latter is potentially caused by the extraction of the digest from the state register and not from the round function.

3.8 Summary

The relevant existing literature have been presented here and consists of SHA-3 structures with variable considerations and performance objectives. There are, however, a few clear tendencies with respect to the adoption of optimization techniques. Compact structures use folding and the most efficient cases among the state-of-the-art regarding compact implementations are folded slice-wise. Folded structures use RAM to implement both the state register and additional data such as the round constants. The most efficient structures are unfolded and use pipeline registers internally in the round function. The highest reported efficiency is seen in a structure which is both unfolded, pipelined, and unrolled so that the implementation includes multiple instances of the round function.

The solutions presented in the existing literature are seemingly lacking in the potential for working as stand-alone entities as many are missing a fully functioning wrapping component. The basis for the reported results are also not the same with many obtaining results from synthesis and only a few from PAR which are more reliable.

4 Proposed solution

This chapter presents the analyses and considerations which have been carried out in relation with this thesis. The existing state-of-the-art is already mature with multiple papers having advanced this scientific field with a variety of approaches. Still, some considerations and approaches are left unexplored. While not all of the concepts discussed here are completely unique or original, they are found worthwhile of presenting as they contribute to give a clear overview of this broad technical field. An exploration of the successful aspects found in the state-of-the-art and the analyses herein performed shaped the results achieved and presented in this chapter.

Folding is an optimization which improves the area consumption at a cost of higher latency and therefore usually a lower throughput. This is the main technique used to reduce the area which is related to the structure. Many considerations must be made when implementing a folded structure in order to meet input dependencies for step-mappings of the round function. These are explored subsequently along with the general efficiency cost of utilizing this technique. Other ways in which area can be reduced are by removing non-critical registers and to use asynchronous multiplexers for signal assignment. For stable timing performance, it is generally advised to maintain registers and use FSMs instead of asynchronous signal assignments. The removal of registers must therefore be done jointly with attentive analysis of the timing. Non-structural optimizations such as these are considered simply as rational and proper use of the syntax of the Hardware Description Language (HDL) and are therefore not discussed further. This is similar to how timing is affected by low-level optimizations such as bypassing the PAR and placing the Real Time Logic (RTL) components manually. Pipelining is the main optimization towards maximizing the timing and therefore throughput. A set of preconditions dictate the relevancy of its utilization, however. The final optimization considered in the proposed solution is focused on unrolling which enters the optimization domain of both area and throughput.

While some techniques are thoroughly covered by the existing literature, certain aspects of the implementations have not been discussed in depth. It is therefore an unavoidable risk that certain aspects of the analysis and the proposed solutions are subjects of *re-inventing the wheel*. This is mostly related with the memory mapping of the state, i.e. how the 1600 bit state is stored in RAM. Elaborately presenting the necessary steps in order to implement the proposed techniques are therefore a novel addition to the state-of-the-art and can be seen as another contribution of this thesis.

4.1 Folding

As seen in Chapter 3, most folded designs incorporate either a lane-wise or a slice-wise folding scheme. At first glance there seem to be several possibilities of partitioning the state. However, additional logic and clock cycles are required if each fold fails to contain all the necessary bits required to produce the output of a step mapping. These bits are referred to as dependencies and they are a bi-product of critical security features of a cryptographic hash function. The dependencies of each step mapping are rather simple, but many structural constraints arise when they are considered combined. They are discussed further subsequently (4.2). Ultimately, as also concluded in previous literature [38], the optimal folding scheme with respect to the combined dependencies of the round function is a slice-wise architecture. Therefore, other folded configurations are not considered in the further analyses.

There is a clear difference between the performances of the compact folded designs and the straight-forward unfolded designs. It is therefore trivial to observe that while there is a trade-off between the utilization of resources and the latency, the relationship between them is not symmetrical. There is a certainty that the latency will increase by at least a factor equal to the folding-factor, e.g. a minimum latency of 48 an 96 is achieved with a folding-factor of 2 and 4. The resource utilization, however, is more complicated to determine and while round function logic is reduced, additional components such as multiplexers and larger counters are introduced when a folding-scheme is employed.

The formula for calculating the throughput, T, is given in Equation 4.1, where r is the block-size, which is the size of the message block. The frequency, f, is the number of clock cycles the design can perform in a second and is determined by the critical path between registers. The latency, L, is the amount of clock cycles required for the processing of one message block. The efficiency, E, is obtained by dividing the throughput by the area, A, which is the number of slices consumed by the design, as depicted in Equation 4.2.

$$T = \frac{r \cdot f}{L} \tag{4.1}$$

$$E = \frac{r \cdot f}{L \cdot A} \tag{4.2}$$

When plotting the efficiency as a function of area for various folding schemes, the cost of the increased latency becomes apparent. This plot is illustrated in Figure 4.1, where 6 different folding approaches are plotted in addition to the unfolded scenario with 24 cycles latency. The frequency is fixed between 200 and 300 MHz with proportionally higher value as the folding factor increases. This is based on intermediate implementation results as well as the reported performances from the existing state-of-the-art [37]. The block size is set to 576 bits as this corresponds to the most secure and prevalent SHA-3 sub-version. The dotted horizontal line denotes roughly the maximum efficiency achieved so far in the existing state-of-the-art, by Ioannou *et al.* [44]. The dotted vertical line at 90 slices denotes the absolute smallest area requirements obtained in the existing state-of-the-art, by Jungk & Stöttinger [39]. Evidently, high folding factors impose great constraints in terms of slice utilization. It also suggests that an efficiency comparable to unfolded structures (with FF=1) seem to be unattainable with a solution with a high folding factor.

This is a simplified model which does not take into account possible optimizations such as pipelining and the actual achievable frequencies, which are not easily modeled. Nevertheless, this model helps to roughly illustrate the maximum footprint of a given folding scheme in order to achieve a given efficiency. For example, the obtained results suggests that an unfolded structure should be kept below 1000 slices for an efficiency metric (of about 5 Mbps/slice).

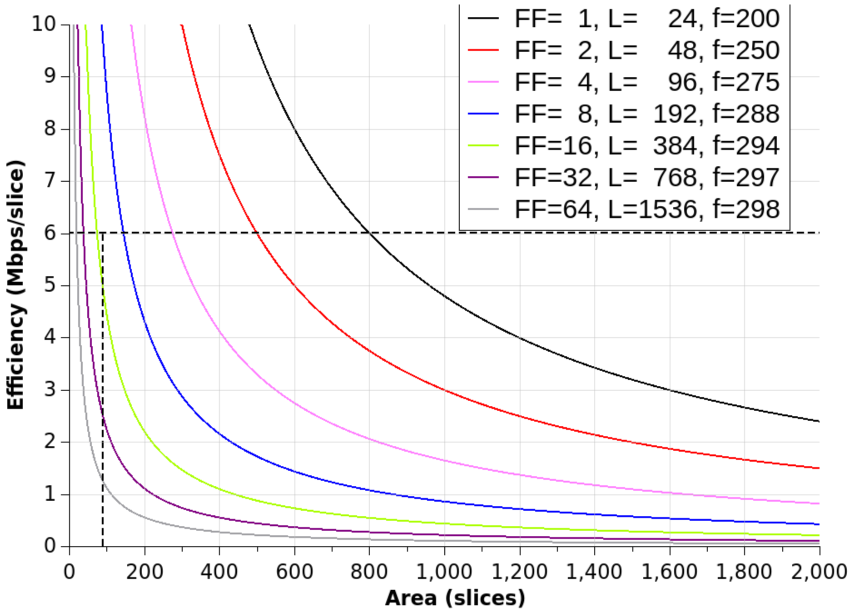


Figure 4.1: Efficiency model as a function of area for seven different folding factors for SHA3-512.

Three folded structures classified as compact and mid-range designs are found to be interesting and worth implementing. These correspond respectively to a folding factor of 64 , 8 and 4 for slice-wise structures and represent the best compromise of each class. A mid-range design should not be folded excessively if a throughput above 1 Gb/s is targeted with a relatively low area occupation. Both FF=8 and 4 minimize the area requirements, with acceptable throughputs. FF=8 is an interesting trade-off, given the possibility of pipelining which is further explored in section 4.3. The objective for a compact design is to maintain a highly conservative area consumption while minimizing the decrease in throughput. The compact designs have been thoroughly covered and recently optimized in [39].

Depending on the folding-scheme, there are various ways in which one can implement a wrapping component. A solution which trades larger area for lower latency involves detaching the wrapper so that it runs in parallel with the round function. This imposes a minimum limit to the size of the IO-ports, especially for designs with a low folding factor, so that transceiving the message and the digest occurs within the clock cycles needed for the round function. The minimum IO-port sizes depends on the SHA-3 sub-version and the folding factor, as illustrated in Table 4.1. When rounding up to the nearest factor-of-two number, for unfolded designs the port size should be kept above 64 bits, but there is no need for large ports in designs with a high folding factor. The alternative solution is to inject the message block directly into the state as the round function is halted. With a port size of 64 bits, the additional latencies introduced for version SHA3-224 to -512 are respectively 22, 21, 19, and 17 clock cycles including the extraction of the digest. This poses a significant decrease in throughput for unfolded structures, but can be a valuable trade-off for structures with high folding factors.

		Minimum I/O port size (bits)			
Folding	Latency (cycles)	SHA3-224	SHA3-256	SHA3-384	SHA3-512
1	24	58	56	51	46
2	48	29	28	26	24
4	96	15	14	13	12
8	192	8	7	7	6
16	384	4	4	4	3

Table 4.1: I/O port size restriction for minimal latency.

4.2 Dependencies

The separate dependencies of each step-mapping are detailed in Section 2.3.2. For an exhaustive exploration of the dependencies for all possible folding-schemes, readers are referred to [38]. The slice-wise folding-scheme fails to meet the dependency of the ρ and θ step-mappings. Each output of θ depends on the inputs bits of both the slice at the same location and at the lower z-coordinate and ρ provides a permutation across all folds of the state.

While the dependencies are quite simple individually, it is the combination of the step--mappings that result in the challenges from the implementation point-of-view. An example is given in Figure 4.2 where the combined dependencies for one output bit of the round function is depicted. To produce the bit at lane 12 slice 0 for the next round, a total of 33 input bits (depicted in blue) are required from several different slices and lanes. The combined dependencies of the standard round function therefore prohibit most types of folding-schemes without requiring additional logic and clock cycles.

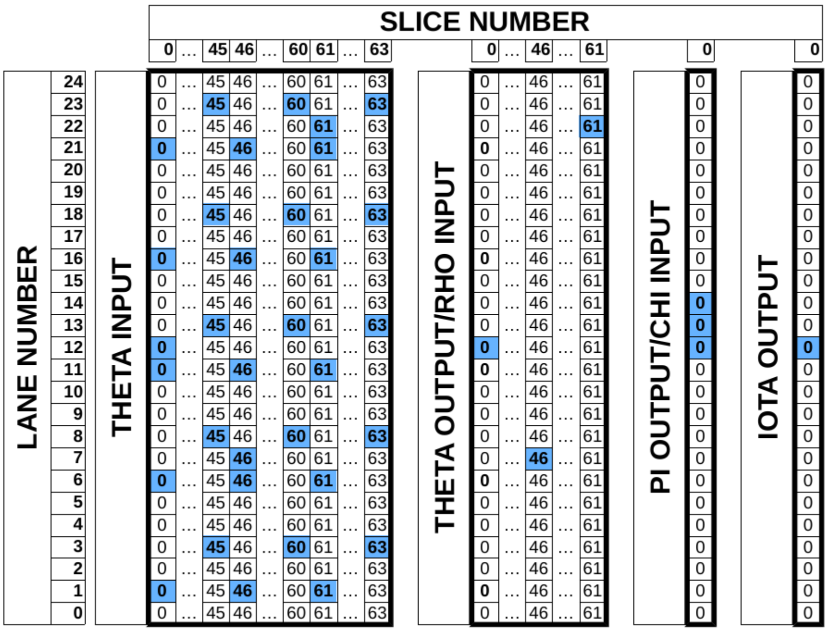


Figure 4.2: Dependency trace through the standard round function from start on the left side to finish on the right. 1 output bit depends on 33 input bits.

The slice-wise folding-scheme seems to be the most flexible solution with respect to a variable FF. It is clear from Figure 4.2 that bits are separated over more lanes than slices and handling the state per lane will therefore require a higher degree of parallel memory access or additional clock cycles. Implementing a slice-wise handling, on the other hand, allows for the processing of the complete round function with a maximum FF with little additional logic. A downside with slice-wise folding is related with the wrapper and the interface with the user. Consistently with the specification of SHA-3 by NIST, an input message is split into the lanes of the state, which are then referred to as words. For a slice-wise folding-scheme, it is therefore necessary to re-order the incoming words for compatibility with the internal data-path. Only then can the message be successfully XORed with the outer part of the state and further introduced in the Keccak round function. There are two possible solutions for solving this orientation-mismatch. The most common solution in the existing literature is to simply require that the incoming words are re-ordered. The loading and unloading from the IO-buffer is depicted in Figure 4.3. The alternative is to add more clock cycles to the latency so that only parts of the incoming word equal to the width of the fold are loaded in each clock cycle and vice versa.

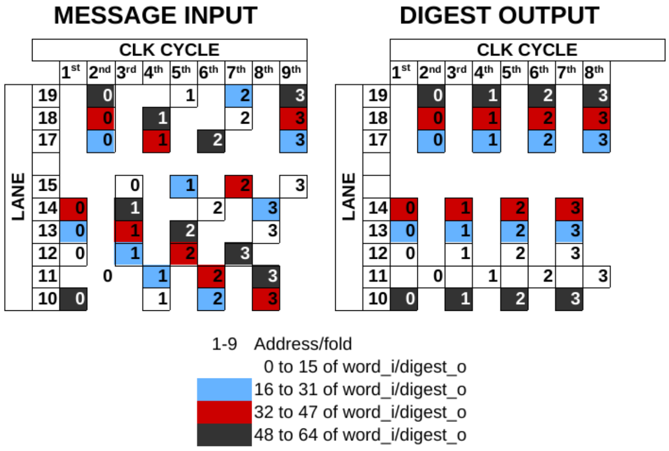

MESSAGE INPUT
DIGEST OUTPUT
CLK CYCLE
LANE
1-9 Address/fold
0 to 15 of word_i/digest_o
16 to 31 of word_i/digest_o
32 to 47 of word_i/digest_o
48 to 64 of word_i/digest_o

Figure 4.3: Loading procedure for the slide-wise folding-scheme if incoming words are already properly re-ordered.

In folded structures, the IO-buffer of the wrapper can be stored in RAM in an identical manner as the state and will therefore shrink proportionally with the folding factor. The logic of the wrapper will increase, however, the logic of the wrapper will increase as the counters are increased and as the intra-round dependencies must be resolved.

Pipelining requires another degree of dependency-free cycles, as briefly mentioned in the previous chapter discussing pipelines in Section 3.2. With the introduction of an internal pipeline registers, two folds are concurrently processed by the round function at two different stages. To avoid bubbles, a new fold must be able to enter the first stage of the round function before a previous fold is done processing in the last stage. In most cases this is hindered by inter-round dependencies as all folds are updated from the processing of each fold. Bits which correspond to a fold at the input are spread across all folds at the output because of the permutation step-mappings of the round function.

4.2.1 Rescheduling of the round-function

The dependencies of the ρ and π step mappings are collectively restricting any form of folding as the permutations act in all three dimensions of the state, providing a high level of mixing. A visualization of this mixing is depicted in Figure 4.5, where the top matrix illustrates the input and the bottom matrix the output. The main issue is caused by the order of the step mappings, with the permutations being located at the center of the round. Seemingly, no matter what kind of folding-scheme is used, the data at the output of the round function will belong to completely different fold(s) than the input. Even though the order of the step mappings within a round are critical for the correct functionality of the Keccak function, the sequence can be altered by rescheduling the round function. This concept was first presented by Jungk & Apfelbeck [34]. Grouping the θ, χ and ι step mappings together greatly simplifies the task of implementing a folded round-function as the permutation step-mappings of ρ and π are off-loaded from the round function itself. Two possible rescheduled round functions are shown in Figure 4.4. This adds one more round to the function as R_f1, composing round 0, R_f3 round 1 to 23 and R_f2 round 24. Additional multiplexers are also required to bypass R_f2 in round 0 and R_f1 in round

Standard round:
$R_f = \iota \circ \chi \circ \pi \circ \rho \circ \theta$ x 24

Re-scheduled round solution 1:
$R_f1 = \theta$ x 1
$R_f3 = \theta \circ \iota \circ \chi \circ \pi \circ \rho = R_f1 \circ R_f2$ x 23
$R_f2 = \iota \circ \chi \circ \pi \circ \rho$ x 1

Re-scheduled round solution 2:
$R_f1 = \pi \circ \rho \circ \theta$ x 1
$R_f3 = \pi \circ \rho \circ \theta \circ \iota \circ \chi = R_f1 \circ R_f2$ x 23
$R_f2 = \iota \circ \chi$ x 1

Figure 4.4: Listing of the two possible re-scheduled round functions compared with the default schedule.

24.

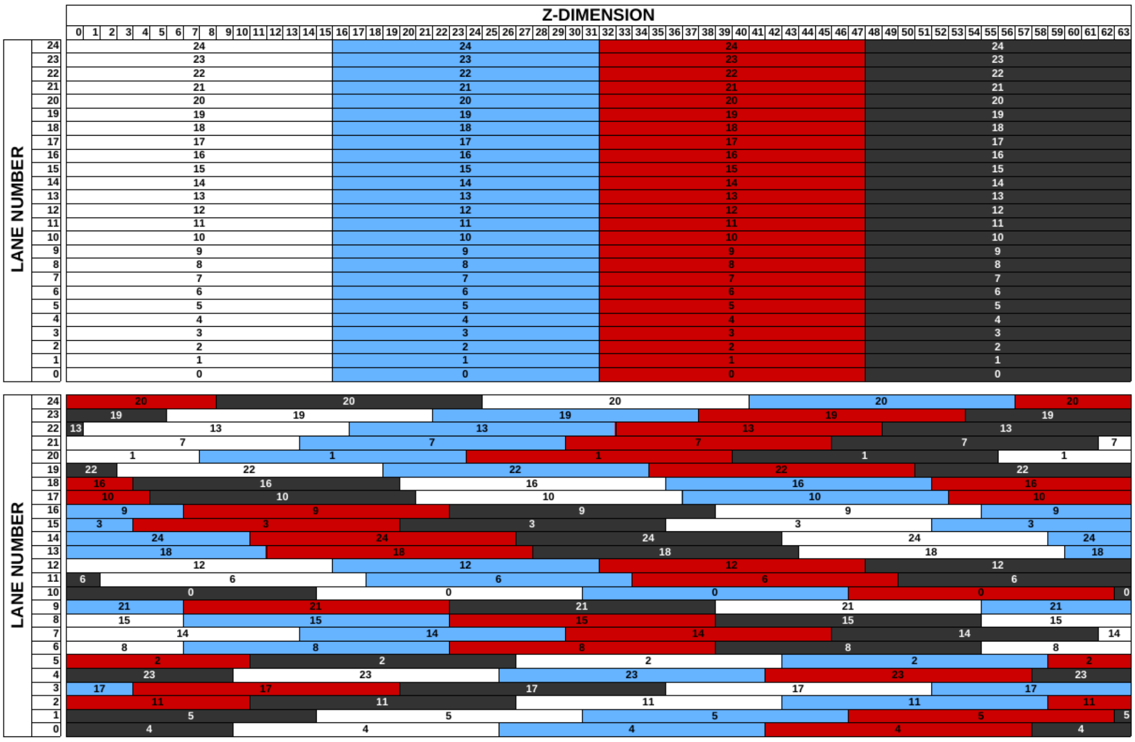


Figure 4.5: Simplified visualization of the combined Rho and PI step mapping. The top matrix constitutes the input and the bottom the output.

A dependency for solution 1 of the re-scheduled round function is depicted in Figure 4.6. For each output bit of this function, 16 bits are required at the input, which is less than half of the dependencies of the standard round function. The originally scheduled round function caused intra-round dependencies because of the centered permutation step-mappings. In the re-scheduled round function, the bits are spread across more slices, however, the bit dependencies are more predictable, i.e. the location of each bit is more consistent for the step-mappings with multi-input dependencies. A fold of consecutive bits at the input will correspond to a similar fold at the output, consistently as the folding factor increases. The intra-round dependencies are thus, traded for inter-round dependencies. The total cost is that an extra instantiation of the state is required in addition to a more complex addressing solution.

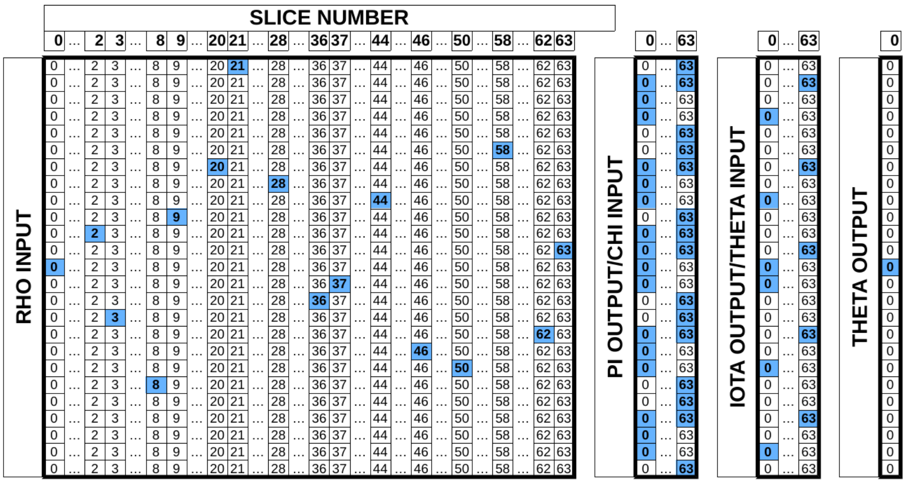


Figure 4.6: Dependency input on the left side, output on the right. 1 output bit depends on 16 input bits.

Two options are considered with respect to the re-scheduled round function. The standard solution is to inject the message block and state into the state register from the IO-buffer of the wrapper. This solution is depicted on the left side of Figure 4.7. R_f2 must then be bypassed during round 0 which can be solved by multiplexers at the input of R_f1. Round 24 involves the extracting of the state from the output of R_f2. This solution results in a latency of 25xFF clock cycles. A downside to this solution is the ρ step-mapping which is embedded in the state register so that any data which is written and read will go through this permutation. For round 0, this would result in an erroneous round function and any signaling and addressing which is not a straight forward mapping of signals must be compensated. The alternative solution is depicted on the right side of Figure 4.7. As this solution is bypassing the state register, or rather including the R_f1 block while being injected into the state register, the latency is reduced by FF clock cycles. The downside of this solution is that the combinational path may increase where the absorbing phase is iterated during hashing of large messages. The new message block can then not enter R_f1 without being XORed with the output of R_f2. This solution seems to be the standard approach in the existing state-of-the-art, with a latency of 25xFF.

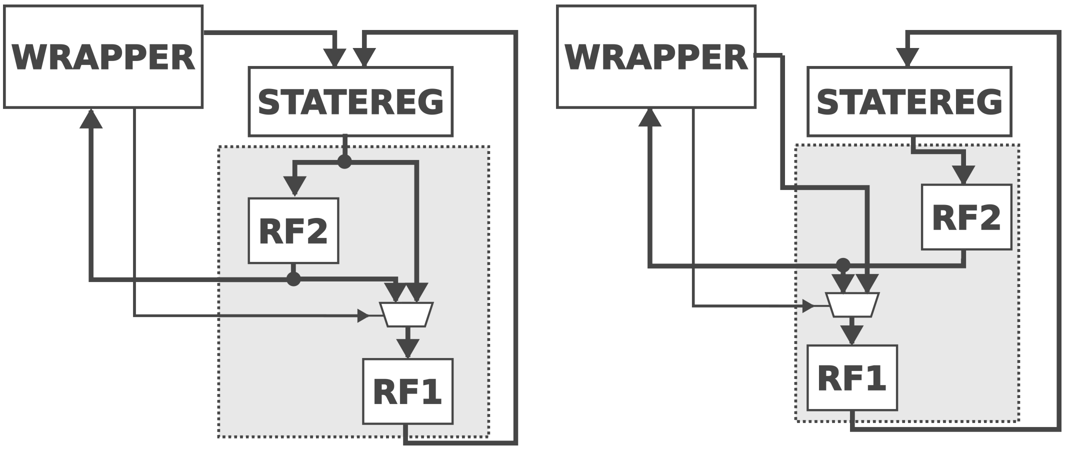


Figure 4.7: Alternatives for the implementation of the re-scheduled round function.

4.2.2 Theta intra-round dependency

With the established rescheduling of the round function and the resolved inter-fold ρ dependencies, the remaining issue concerning data dependencies is found in θ where the processing of every output slice depends on the same input slice and additionally the slice at the lower z-coordinate. For further analysis, a slice-wise folding-scheme with folding factor 4 is considered so that the state is partitioned into 4 folds of 16 slices. The consequence of the θ intra-round dependency when considering a slice-wise folding scheme, is that each fold requires the slice with the highest z-coordinate of the previous fold for processing of its first slice, i.e. θ generates F0S0' from F0S0 and F3S15, F1S0' from F1S0 and F0S15, etc. A straight forward method of solving this issue is to add another cycle for each round to complete the F0S0 slice processing. However with the cost of additional logic, 24 clock cycles can be saved. This is desired in a structure with a low folding factor as maintaining a minimal latency should be of priority. A small addition to the required area has a small impact of the overall efficiency.

The first approach, titled *F0S0 pre-processing*, involves providing the F3S15 slice for the F0S0 processing during sub-round 0. This is done by collecting the relevant future F3S15 bits from each sub-round of the previous round This solution is illustrated in Figure 4.8. The cost of this specific solution is that F3S15 is generated and stored twice. As the future F3S15 bits are collected, they must be processed by the χ and ι step-mappings in a "miniature round function" before the column parities are calculated. F3S15 for round 0 should be collected from the IO-buffer during the absorbing phase and from the output of the round function during the succeeding rounds. Latches fill the F3S15 during the four sub-rounds and it is constantly fed through the miniature round function. The column parities of the processed F3S15' is then provided to the θ step-mapping during the next sub-round 0 and XORed with the F0S0' slice. No other state-of-the-art work seem to have considered this solution.

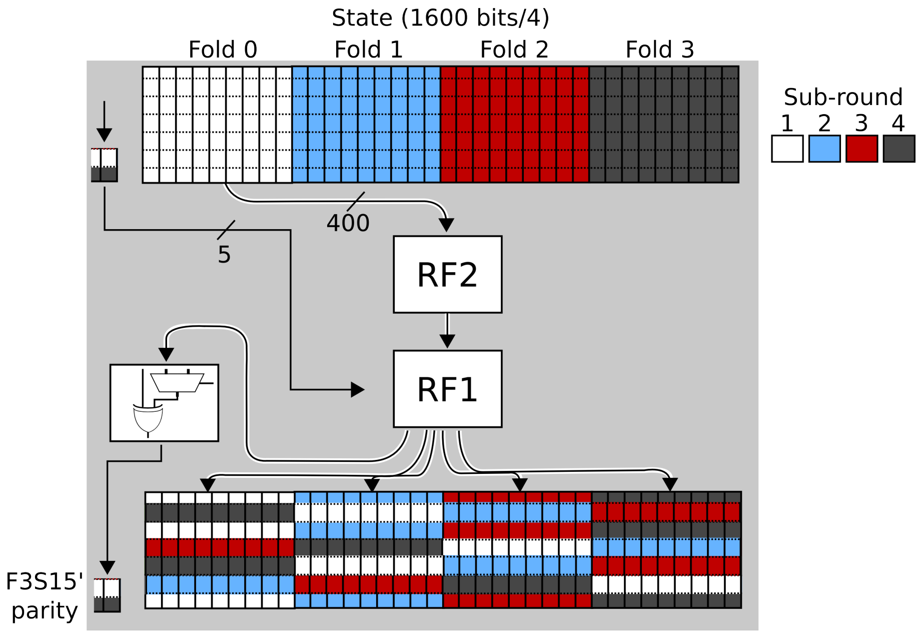


Figure 4.8: First approach to solving the intra-round dependency of θ: F0S0 pre-processing.

The second approach, called *F0S0 post-processing*, delays the computation of the F0S0 slice at the θ step-mapping of sub-round 0 and continues processing during sub-round 3 when F3S15 is processed. This solution is depicted in Figure 4.9. An identical solution is discussed in Jungk's thesis [38] and Jungk & Stöttinger [39]. This requires an additional register for storing the intermediate values for the F0S0 slice, its column parities and the eventual F0S0' end slice. The original location of the F0S0 slice in the RAM is only accessible during one of the sub-rounds because of the depth of the memory structure.

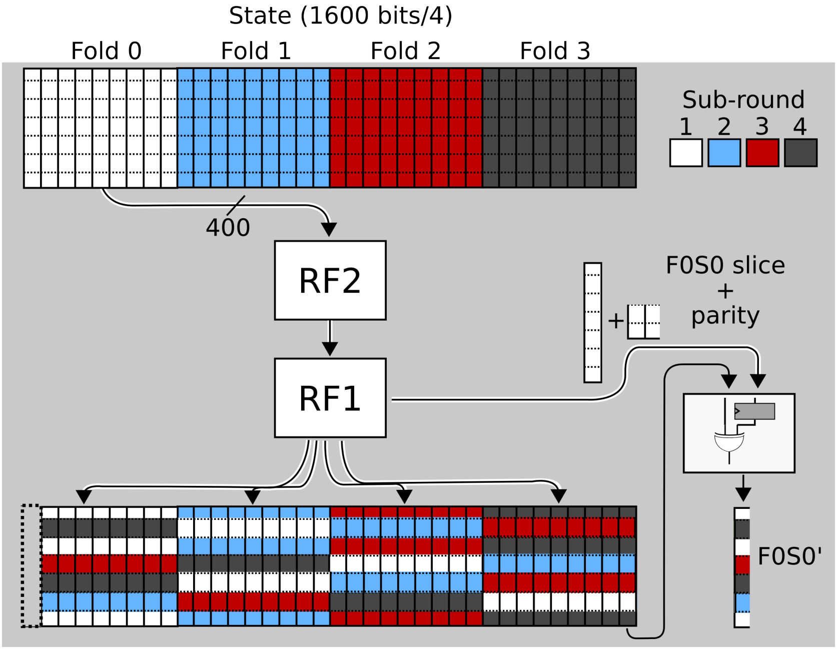


Figure 4.9: Second approach to solving the intra-round dependency of θ: F0S0 post-processing.

The down-side of the pre-processing solution herein proposed is that the F3S15 is processed twice. Nevertheless, the benefit is that the F0S0 slice is processed along with the rest of the slices of fold 0 and can therefore be stored in RAM along with the rest of the fold. The post-processing requires two versions of F0S0 along with the computation of its column parities. Regardless of how the processing of the F0S0 slice is processed, the bottom slices of each subsequent fold must still be provided with the column parity of the last slice of the previous fold. The colum parity of slice number 15 of each sub-round apart from sub-round 3 must be stored stored in a register for the θ step-mapping.

4.2.3 Embedded memory

The immediate benefit of folding is that less data needs to be accessible concurrently, allowing for the utilization of memory blocks as storage solutions. As the folding factor increases, the depth of memory blocks can be exploited further. Distributed RAM is often preferred to BRAM as the number of bits which are necessary to be stored are too few for a reasonable utilization of the blocks which are in the order of 10^4 bits. Additionally, distributed RAM still occupies slices and it is therefore more convenient to compare the area consumption of different structures. Consequentially with deep memory solutions, challenges arise with the limited availability of the state. Data can only be read and written into locations of the memory which are multiples of the width of the block. As illustrated in Figure 4.10, if the necessary bits are not aligned with the width, additional read cycles are necessary to obtain the data. With the suggested re-scheduled round function, the ρ and π step-mappings are either solved by wiring or integrating it with the memory addressing of the state. However, the intricate dependencies are still present. The ρ step-mapping is the main cause of the challenges related with the memory mapping as the rotations are not aligned with any factor-of-two number. The addressing challenges are illustrated in Figure 4.10. Independent of the folding-factor, the output of the round function in one sub-round are processed in different sub-rounds during the next round. The optimal method of storing the state in RAM is where the width of each block is utilized best and the smallest number of additional flip flops are required. This is a topic which has not been clearly mentioned in the existing literature. The solutions which are presented here are conceived in relation with this work unless otherwise noted and are therefore possibly novel additions to the state-of-the-art.

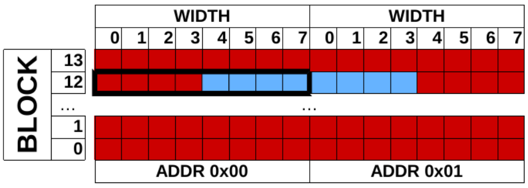


Figure 4.10: Illustration of the challenge related with data access in memory blocks.

lane-oriented memory mapping

For slice-wise structures, it is beneficial to treat the π step mapping as wiring between memory blocks while ρ is solved by addressing. The round-function is re-scheduled for this folding scheme and with both of the variations of the re-scheduled round, as listed in Figure 4.4, the permutation step-mappings are either addressed during reading or writing of the fold in the memory. In further analyses involving the re-scheduled round, the ρ step is solved during writing to memory, however there is no inherent difference between alternatives.

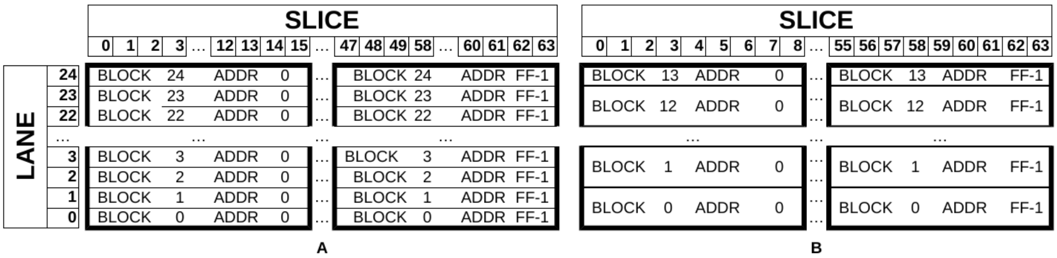


Figure 4.11: Solution A and B for memory mapping of the state for slice-wise structures.

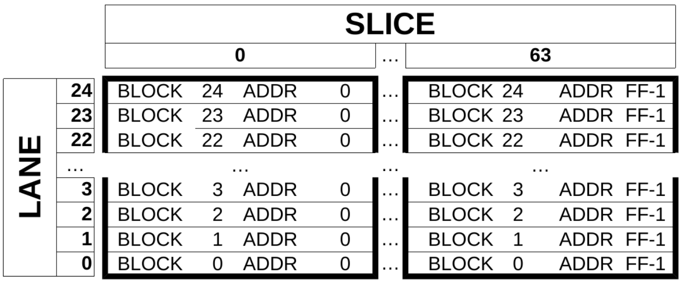
One solution is to assign a memory block to each lane, as depicted in Figure 4.11A. Depending on the folding factor and the specifications of the memory block, each fold is located at different depths/addresses and writing to memory is done by assigning an offset to the address for certain memory blocks. The addressing can be determined by Equation 4.3, where R denotes the rotation by ρ and $f(R)$ rounded down is the memory address of the RAM for the respective lane.

$$f(R) = \begin{cases} \frac{R*FF}{64} + 1, & ifR \bmod (\frac{64}{FF}) \geq \frac{64}{FF*2} \\ \frac{R*FF}{64}, & otherwise \end{cases} \tag{4.3}$$

As the inter-round dependencies produced by the permutations result in the mixing of bits from different folds, an additional state is required. This way, each sub-round will access the same state with no bits being overwritten. Comparing Figure 4.11 with the ρ and π step mappings illustrated in Figure 4.5, it becomes apparent that the misalignment of the ρ rotation with the width of the memory block causes some bits to be missed by the addressing. A larger figure is found in the Appendix 7.2.1, which depicts the full dependencies of a fold for slide-wise folding-schemes with various FF incorporating this memory mapping solution. This issue can be greatly reduced by packing together the bits from a fold into the memory blocks assigned to that fold/z-position.

As FF is increasing, the memory blocks and the addressing can remain the same so that the width of each block covers multiple folds. An alternative solution is to further exploit the depth and contain multiple lanes in a block, as depicted in Figure 4.11B. The number of memory blocks are then reduced, however, the number should remain equal or higher than the folding factor.

In order to better exploit the full depth of the embedded memory in structures with close to maximum folding factors, mapping solution C can be adapted, as depicted in Figure 4.12. Since only a small number of bits are accessed in parallel, the lane-oriented memory mapping can be replaced by a narrow bit-oriented approach. Large embedded memory blocks can then be changed for smaller memory solutions such as distributed RAM in FPGAs or Static RAM (SRAM) in ASICs. This solution is used by Jungk & Stöttinger [39].

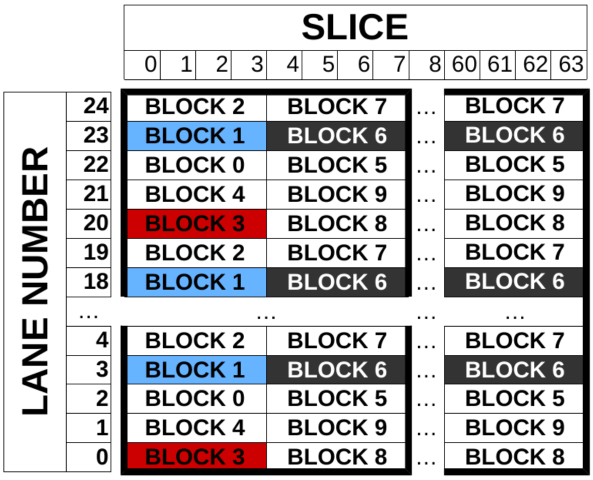


LANE	SLICE 0	...	SLICE 63
24	BLOCK 24 ADDR 0	...	BLOCK 24 ADDR FF-1
23	BLOCK 23 ADDR 0	...	BLOCK 23 ADDR FF-1
22	BLOCK 22 ADDR 0	...	BLOCK 22 ADDR FF-1
...	...	...	...
3	BLOCK 3 ADDR 0	...	BLOCK 3 ADDR FF-1
2	BLOCK 2 ADDR 0	...	BLOCK 2 ADDR FF-1
1	BLOCK 1 ADDR 0	...	BLOCK 1 ADDR FF-1
0	BLOCK 0 ADDR 0	...	BLOCK 0 ADDR FF-1

Figure 4.12: Solution C for memory mapping of the state for slice-wise structures with maximum folding factor.

Column-oriented memory mapping

The BRAM of modern FPGAs contain built-in parity calculation which can be used to substitute the θ_1 sub-step which involves the parity calculation of each column of the state. The round function will then contain one less level of logic as the first 5-input LUTs of a standard round function is reduced. This is a solution which trades-off smaller critical path of the round function with higher occupied embedded memory resources. Figure 4.13 depicts how bits are mapped into memory blocks. The port width is reduced as output bits are used for the parity bits. Additionally, the parities are generated for each output byte and only the five bits of a column are able to occupy this space. This is a very in-efficient mapping solution for low folding factors, but compact designs would occupy a low number of the available BRAM on an FPGA. E.g. FF=8 and 16 requires 10 and 5 RAMB36 blocks respectively. The default round function scheduling must be used so that the θ step-mapping is the first step of the round.



LANE NUMBER	SLICE 0 1 2 3	4 5 6 7	8	60 61 62 63
24	BLOCK 2	BLOCK 7	...	BLOCK 7
23	BLOCK 1	BLOCK 6	...	BLOCK 6
22	BLOCK 0	BLOCK 5	...	BLOCK 5
21	BLOCK 4	BLOCK 9	...	BLOCK 9
20	BLOCK 3	BLOCK 8	...	BLOCK 8
19	BLOCK 2	BLOCK 7	...	BLOCK 7
18	BLOCK 1	BLOCK 6	...	BLOCK 6
...	...		...	...
4	BLOCK 2	BLOCK 7	...	BLOCK 7
3	BLOCK 1	BLOCK 6	...	BLOCK 6
2	BLOCK 0	BLOCK 5	...	BLOCK 5
1	BLOCK 4	BLOCK 9	...	BLOCK 9
0	BLOCK 3	BLOCK 8	...	BLOCK 8

Figure 4.13: Illustration of a column-oriented approach to memory mapping using built-in parity calculation of Virtex-5 FPGA BRAM. This structure has a folding factor=8.

4.3 Pipelining

For pipelining to be an efficient technique, all the registers of the pipeline should be kept as full as possible, minimizing bubbles. At start-up and conclusion, bubbles are inevitable as the pipeline is filled and cleared. Because of the combination of the step-mappings, most SHA-3 structures will have inter-round dependencies so that each round will contain cycles where the pipeline is emptied. Each fold must wait to be processed until all previous folds have updated the state. This causes the efficiency of pipelining to be drastically reduced as it sets a high requirement for the increase in frequency to compensate the increased latency and area. In order to approximate the cost of pipelining, the relationship between the number of pipeline registers and the throughput and are requirements must be approximated. Equation 4.4 gives the total area requirement of an optimized structure as a function of the folding factor and the number of internal pipelines. An approximation regarding the area consumption is made so that the total area is divided equally between the three main components: the wrapper, the state and the round function. It is also considered that 10% of the basic structure occupies the same area, independently of the folding and number of pipeline registers (PL). The default for PL is one, as the state register is included in the factor.

$$A = (\frac{1}{10} + \frac{9}{10 \cdot FF}) \cdot A_0 + \frac{9}{10 \cdot FF \cdot 3} \cdot A_0 \cdot (PL - 1) \tag{4.4}$$

Similarly, the latency is increased as a function of FF and PL and the relationship for structures with and without inter-round dependencies is given by Equation 4.5 and 4.6.

$$L = FF \cdot L_0 + ((PL - 1) \cdot L_0) \tag{4.5}$$

$$L = FF \cdot L_0 + (PL - 1) \tag{4.6}$$

The Efficiency Equation 4.2 shows how the frequency must compensate for the increased latency and area. With the approximation of the area consumption ratio, it is possible to plot the necessary compensating frequency for the various cases of FF and PL. The cost of pipelining is approximated so that each increase in PL adds $3/10$ of the area for FF=1, but this is naturally decreasing as FF is increasing. These cost approximations are based on intermediate results obtained in this work and numbers reported by the existing state-of-the-art. Figure 4.14 depicts the relationship between the efficiency and the frequency for structures with various combinations of pipelining and folding factors. The left side presents situations where the inter-round dependencies have not been solved and the pipeline must be emptied before each new round. It is observable that even optimal frequencies do not result in a decent efficiency. However, pipelining can improve the efficiency of folded structures with bubbles to a larger degree than unfolded structures. This can be seen when comparing the red and black dotted lines, which denotes folded structures, with the solid lines of the same colors, which denotes unfolded structures. The angle difference is noticeably lower between the two folded structures compared with the unfolded.

The right side shows the more efficient structures with no bubbles in the pipeline. Latency is then given by Equation 4.6. The horizontal green line illustrates the top performance achieved by the state-of-the-art while the vertical green line the maximum frequency which is achievable with a V-5 FPGA. The right-side graph suggests that despite inter-round dependencies being met, the quantity of pipeline registers should be kept at a minimum. These theoretical models can possibly fail in accuracy where slices are initially utilized inefficiently. Pipelines can be purely or partly fitted into the already occupied slices if they contain unused Flip Flops (FFs). A more efficient slice utilization is then achieved and the implementation of the first pipeline registers will improve the efficiency significantly.

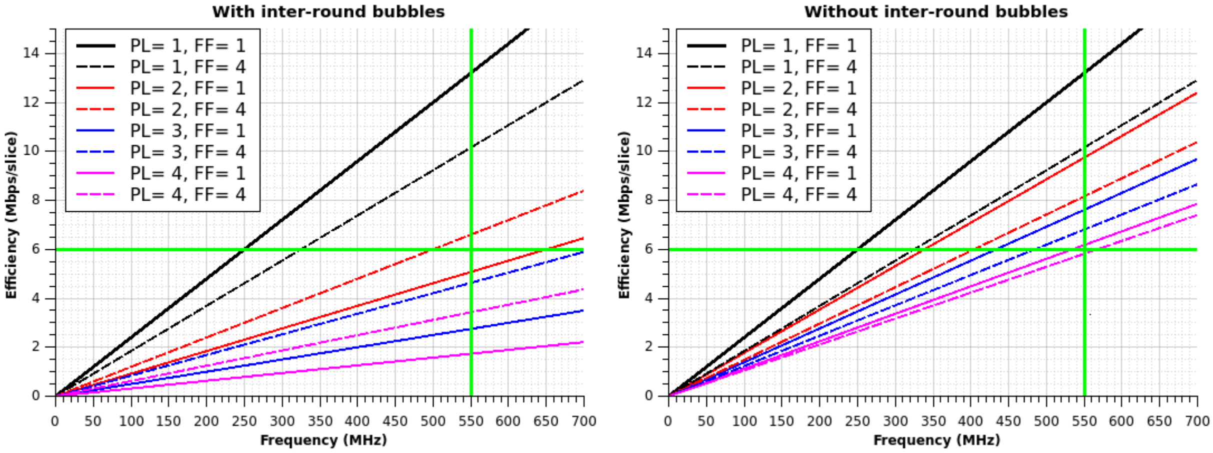


Figure 4.14: Graphs with the efficiency as a function of frequency for various combinations of pipelines and folding factors. Figure A (left side) presents cases where the pipeline contains bubbles While Figure B (right side) is without bubbles.

The general issue with pipelining for unfolded structures is the sponge construction characteristic stating that multi-block-messages must be processed by the sponge function before the next block of the message is merged with the result. For Keccak, the function involves 24 rounds of Keccak-*f*. Gaj *et al.* [4] presents a statistic of the cumulative distribution of packet size for a typical Ethernet node which is a common application for hash functions. The numbers they present show that roughly 50% of the packages processed are below the block size of SHA-3 versions. Furthermore, multiple packages are usually available in a queue for processing. From this it can be concluded that both Multi-Block-Messages (MBMs) and MMHs are realistic scenarios and should be considered, i.e. a SHA-3 design should support multiple small messages as well as larger ones causing an iterated absorbing phase.

Of the considered optimization techniques, pipelining and the PL is the largest factor impacting the frequency of a structure and therefore the throughput. Pipeline registers should be introduced into the round function if only small messages are considered for hashing, as this will greatly decrease the critical path, and thus determine the upper bound of the system frequency. A pipeline-scheme similar to the designs by Athanasiou *et al.* [43] could be adapted where only one or two registers are used. It is possible to design a wrapper which can support both multi-block-messages and small messages with internal pipelines, however this increases the complexity substantially. Consecutive blocks of a message must be absorbed with the correct state after 24xPL cycles and the protocol between the user and the wrapper must be modified. The existing literature with more elaborate analysis of various pipelining schemes [41, 40, 42] have implemented structures with a high number of pipelines. Athanasiou *et al.* [43]

have only implemented one pipeline and achieves the best performance off the structures with internal pipelines. This and the results depicted in Figure 4.14 suggest that one or two pipeline stages should be sufficient and optimal for an increase in efficiency within the boundaries of the potential system frequency.

Implementing pipelined folded structures involve the same issues as unfolded structures and additionally the inter- and intra-round dependencies. As was suggested by B. Jungk [38] and explored by Jungk & Stöttinger [39], the inter-round dependencies can be solved by exploiting the lane-rotations of the ρ step-mapping in slice-wise structures with high folding-factors. Figure 4.15 depicts the SHA-3 slices which contain the slice number 63 from the previous round in red. If the round function contains 1 pipeline register so that two slices are processed concurrently, no red slice can enter the round before slice 63 is done processing. However, all inter-round dependencies are met if the subsequent round is initiated at a white slice. The sequences of white slices dictate the maximum number of pipelines which can be incorporated in the round function as well as the required minimum folding factor. The relationship between these two is depicted in Figure 4.16. Multiple pipeline registers can be incorporated by exploiting a sequence of white slices such as between slice 45 and 53. Up to nine internal pipeline registers can be incorporated, however, this number seems excessive and the upper bound frequency is likely to be reached at a lower number of pipeline stages. A more efficient option would then be to decrease the folding factor and therefore increase the efficiency.

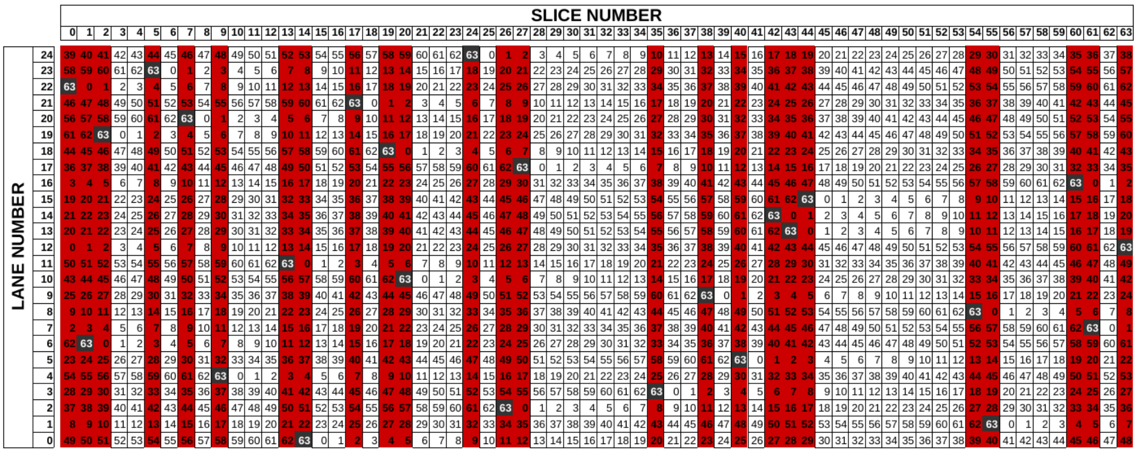


Figure 4.15: Visualization of slices with inter-round dependencies for pipelining of a slice-wise folding. Red highlights the slices which depend on the last cycle of the previous round.

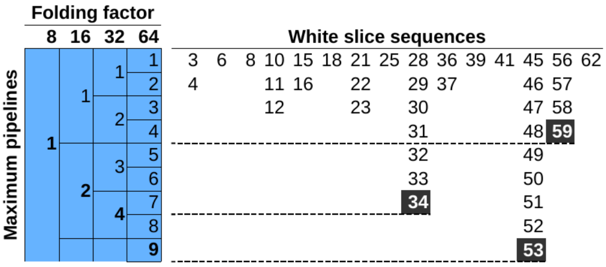


Figure 4.16: Sequences of white slices dictating the maximum number of pipeline registers and minimum folding factor for inter-slice dependencies.

4.4 Unrolling

Unrolling a structure can in certain cases increase the efficiency, but is dependent on whether or not pipelining is utilized and how it potentially is implemented. As was concluded by Bertoni *et al.* [12], unrolling without any added pipelines results in a decrease in efficiency. The latency is reduced proportionally with the unrolling factor, however, the area and the critical path is increased. Hence, unrolling is equally irrelevant as pipelining if multi-block-messages are considered and the wrapper is kept a relatively simple component. As Ioannou *et al.* [44] reports, if small messages are considered, then unrolling with external pipelines should attain good results. The area increases less than proportionally with the unrolling factor (UF) as the wrapper is kept in the design while the rest of the logic is multiplied. The structure presented by Ioannou *et al.* is unnecessarily in-efficient. Since each state is processed in the first instance of the round function for the first twelve rounds and the last instance for the rest, the scheduling ensues a start-up and end phase where only half of the round functions are utilized. This structure also requires additional multiplexers between the two round function instances. A conventional unrolled structure where each state is automatically processed in each round function during ever other clock cycle should be more efficient.

The efficiency as a function of the unrolling factor with and without pipelining can be illustrated graphically by a few approximations. The area is in a simplified manner divided equally between the three components of a SHA-3 structure: the wrapper, the state and the round function. Without pipelining, each increase in UF further increases the area by $1/3$. The Equation 4.2 is thus modified into Equation 4.7. For unrolled structures with an external pipeline, the area increases by $2/3$ instead of $1/3$, but the frequency should then remain constant. With internal pipelines, the area increases by a similar rate so that one and two internal pipelines increases the area by $3/4$ and $4/5$ respectively. These four different structures are plotted in Figure 4.17. The first unrolling-scheme is where the state register remains and only the round function is multiplied. The frequency is set by an exponential approximation similar to the numbers reported by Bertoni *et al.*, but with more realistic numbers for a structure with an undivided round function, i.e. 235 MHz for UF=1 down to 50 MHz for UF=6. The initial area is increased for the various structures so that the basic structure is set to 1000 slices and each additional component adds $1/3$ of the area. The green line represents roughly the maximum efficiency reached by the state-of-the-art.

$$E = \frac{r \cdot f}{\frac{L}{UF} \cdot (A + \frac{(A \cdot (UF-1))}{3}} \tag{4.7}$$

Naturally, only the internally pipelined structures differ at the initial UF=1, but they converge as the factor increases. The external pipeline structure achieves a good performance, but for a small UF, internal pipelining is better by over one Mbps/slice. According to the Equation 4.7 modified for pipelined throughput throughput, the throughput of the structure with two internal pipelines is 11.9 Gbps, which is more than twice that of the structure with an external pipeline. The uncertainties of this plot can be pointed to the un-proportional increase of slices during the

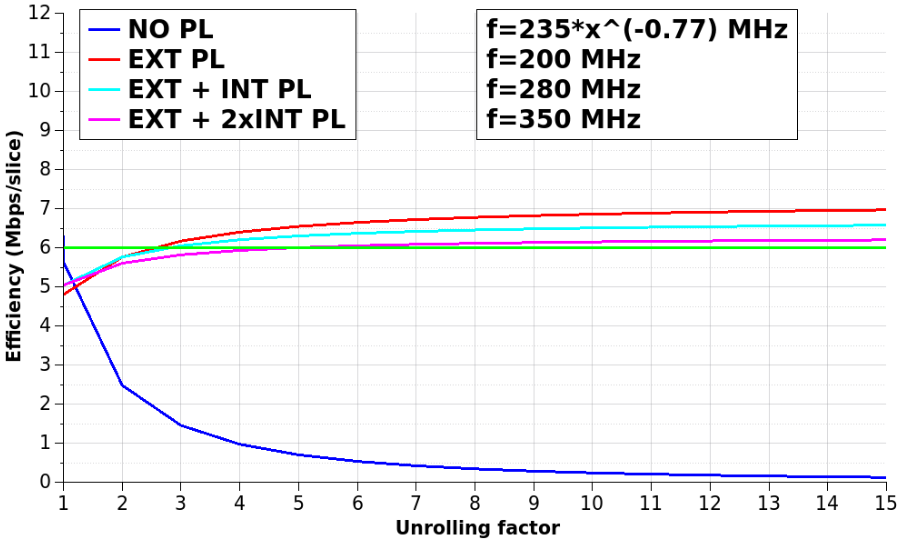


Figure 4.17: Efficiency as a function of the unrolling factor, UF, for four different pipelining-schemes for SHA3-512. Estimation is based Equation 4.7.

first UFs, the ratio between the area consumption of the three components and the frequencies. Only optimally efficient structures should be unrolled. A folded structure can be unrolled, but will achieve a lower efficiency than unfolded structures with lower UF.

A potential issue with unrolling of SHA-3 sub-versions other than SHA3-512 is related with the cycles required to load the words of a message block within the latency of the round function. If a SHA3-224 structure is unrolled with UF=2, 2x1152 bits must be loaded into the IO-buffer in the course of 24 clock cycles, thus requiring a minimum port width of 96 bits. With UF=10, SHA3-512 and -224 requires a minimum port width of 240 and 480 bits respectively with the latter being the upper limit of available IO-ports for many mid-range FPGA models.

4.5 Summary

Folding is the main technique for reducing area, but an increasing folding factor seemingly comes with a cost in throughput and potentially in efficiency. It is important to note the inherent trade-off with folding so that the utilization of this technique is motivated by a goal of minimal footprint at a cost in efficiency. Pipeline registers can be incorporated in the round function if the folding is done slice-wise and with a folding factor larger than 8, but analysis suggests that this is not enough to compensate for the increased latency which is caused by the folding.

Regarding unfolded structures, pipelining is the main technique for improving the timing and therefore the throughput. The efficiency can be distinctively improved if the number of pipelines, PL is kept reasonably low and the pipelining is implemented properly. The inherent issue with pipelining is that the relevancy of the adoption of this technique either depends on the size of the messages and therefore the specific application of the hash function, or a new type of complex wrapper design which is capable of handling both Multi-Block-Messages (MBMs) and small messages.

The ability to exceed the state-of-the-art with respect to efficiency seem to be dependent on the unrolling factor, UF. With a proportional amount of pipelines, the graph suggests that an increasing UF attains a good efficiency, though the plot is an asymptote with a clear upper bound. The analysis herein therefore points to the limitation of the efficiency of a SHA-3 implementation if all realistic applications for the hash function are considered. Hashing of MBMs constrains the adoption of the explored optimization techniques, or the complexity of the structure including the interface. The throughput is in this case upper bound by the critical path through the round function and the efficiency is upper bound by the minimum required resources for an unfolded structure.

The models which have been used in these analyses have their limitations and it remains to be seen how accurate they are by comparison with the actual performance of the implementations. Timing results and the achievable frequency is a metric which is especially uncertain.

5 Implementation

The existing state-if-the-art is by now plentiful and with varying degrees of novelty and factors considered. One negative aspect of the existing papers is that the performances are diverging heavily when compared to how similar many of the structures seem. On the other hand, reported results are based on different estimations. This makes comparing the existing literature a challenging task and drawing solid conclusions from these results are impractical. All the designs found in the more recent literature also propose some sort of optimization technique such as pipeline registers. Only the earlier literature has presented designs and results based on basic, straight forward structure. Also, many of the existing designs do not included a wrapper in their assessment. Therefore, the initial task of this thesis has been to arrive at a basic design which performance and structure can serve as a standard. Further modified structures and their performance can therefore be normalized to this basic structure for a clear and coherent analysis. Additionally, it is also desirable to examine the feasibility of matching or surpassing the claimed performance of the existing solutions. As already discussed in Chapter 3, the basis for the results vary among the literature as well as the extent at which NP-hard problems such as PAR have been prioritized. While area consumption is less challenging to estimate, the critical path is not.

The basic structure is therefore first presented as it serves as the foundation for further optimizations. This is the structure with the highest achievable efficiency when hashing of MBMs are considered. Thus, representing what is suggested as the structural limit of a SHA-3 FPGA implementation. Using the basic structure as a foundation, various implementations are derived where different trade-offs are considered and hence, various optimization techniques are proposed and adopted. A generic SHA-3 structure is developed in order to explore the feasibility of this functionality. Only one previous paper by Athanasiou *et al.* [43] has presented a generic SHA-3 implementation which makes this an interesting structure to explore. All other structures presented support SHA3-512, however configurations are easily done so that one of the other four sub-versions are supported.

A pipelined structure with and without unrolling has also been developed herein for hashing of multiple small messages. This is done both in order to further assess the existing structures which are relevant for this limited scenario and to improve the state-of-the-art. Especially the unrolled structure which has only been seriously explored by Ioannou *et al.* [44], albeit with an inefficient architecture and without internal pipelining. Only these two structures have been proposed, which disregard multi-block-messages.

A folded structure has been developed which belongs to the mid-range class of SHA-3 implementations with a folding factor of 4. This structure is noteworthy more complex than the ones previously introduced. This is mainly caused by the dependencies of the ρ and θ step-mappings and the re-scheduling of the round function. The two earlier papers by Jungk *et al.* and Jungk & Apfelbeck [34, 37] have presented a mid-range structure with a similar folding factor. Nonetheless, these structures have not included a complete wrapper in the assessment of the performance and the interface that is used is the Xilinx FSL IP core. Only one solution for the intra-round dependency of θ is presented in one of the papers and no clear solution is given with regard to how the ρ dependency is solved. The folded structure herein presented incorporates a new solution for the θ intra-round dependency and is assessed as a stand-alone entity. The wrapper is compatible with the standard interfacing where lanes are transmitted sequentially.

All the structures are named after their distinct feature, so that while some of the structures have incorporated multiple optimization techniques, these are not hinted to in the naming-scheme. All wrappers are optimized for latency and not for area, so that a detached IO-buffer off-loads loading and unloading of each message block and digest in parallel with the processing of the state by the round function.

The following section presents the developed basic structure which supports SHA3-512. Section 5.2 presents the generic structure which supports all four sub-versions of SHA-3. Then follows the pipelined structure in Secion 5.3 and the unrolled structure in Section 5.4. The folded structure is presented last in Section 5.5.

5.1 Basic structure

A straight-forward design was conceived with the main purpose of studying the achievable critical path of the round function for hashing of message of all sizes. Additionally, it was desired to set a performance reference for a vanilla structure to compare with various future approaches and implemented techniques. A basic structure here is informally defined as having one round of Keccak-*f* implemented in combinatorial logic, registers are made up of flip-flops and standard syntax for a hardware description language such as VHDL is used to infer all components. The basic structure supports one of the sub-versions of SHA-3 and is configured for SHA3-512 as this is the most prevalent version with the highest security claim. For comparison with other existing implementations, results are adjusted to the SHA3-512 version with the smallest block size if other versions are presented. As depicted in Figure 5.1, the top level of the design contains three RTL blocks: statereg, roundfunc and wrapper. In this form, the three components correspond respectively to their theoretical modules: the state, Keccak-*f* and the sponge construction. The roundfunc is passive and the input is always connected to the state in statereg. The complex logic is therefore kept in the wrapper so that any structural modifications to the basic design involve alterations of the wrapper while the two other components remain mostly the same. Both the state, partitions of it such as a plane which is used during the θ step-mapping and the IO-buffer are realized as two-dimensional arrays of $x * 64$ bits. The IO-ports have been set to 64 bits for this structure, however, as have been pointed out regarding unrolling in Section 4.4, a larger port size is required if UF and the block message size is sufficiently large.

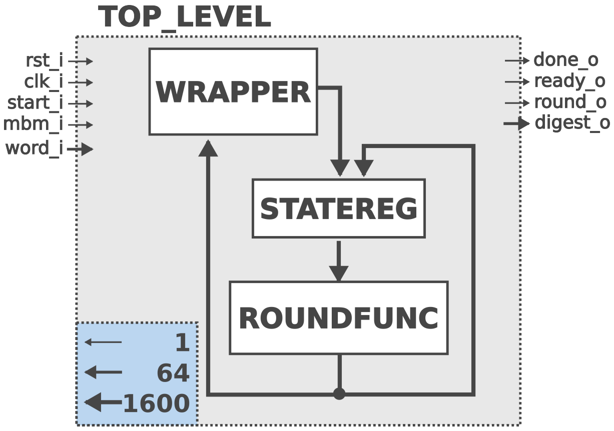


Figure 5.1: Simplified top level schematic of the basic structure.

5.1.1 Wrapper

The wrapper contains the IO-buffer for message block injection and digest extraction from the state, as depicted in Figure 5.2. The SHA3-512 sub-version which is supported is specified with the message block constituting 9 lanes and the digest 8 lanes. The wrapper is detached from the round function to keep the latency within the minimum 24 clock cycles. There is no padding functionality incorporated in the wrapper so the structure is dependent on the message being padded by the user in addition to signaling the start of a message and whether it is a Multi-Block-Message (MBM). In case of the latter, there is an iterated absorbing phase of the sponge construction so that the input message is XORed with the previous outer state. The inner state remains unchanged and is simply passed through to the state register. In this manner, no version specific logic is found in the other two components.

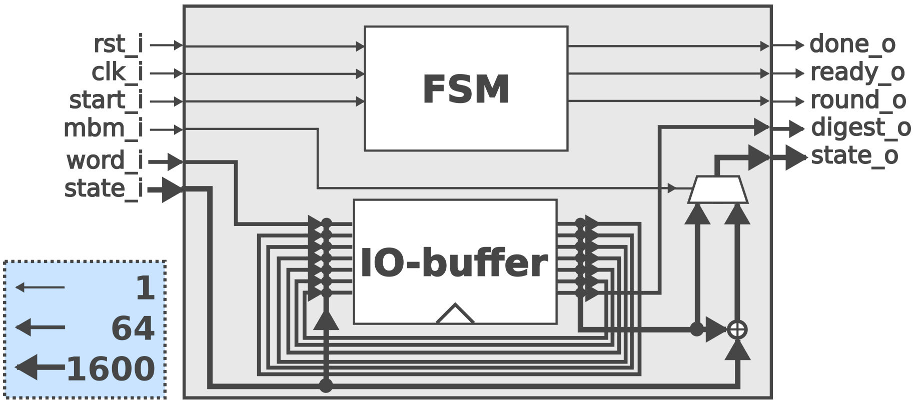


Figure 5.2: Schematic of the wrapper component of the basic structure.

The logic of the wrapper is kept as simple as possible with one Moore-type FSM, main_fsm, handling the round counter data in and out of the IO-buffer. The round counter decrements from 23 down to 0 and is reset if the buffer contains a new message. The schedule of the wrapper is depicted in the Appendix 7.4.1. Multiple wrapper designs were considered during the course of the project. Two FSMs could work in parallel where one handles the round counter and another the IO-buffer. A counter can be implemented in multiple ways and achieve a better working frequency by utilizing the fast carrier logic of a slice. For the basic structure, the counter is incorporated into each state of the main_fsm and controls the transitioning between states.

The buffer is realized as a FIFO which is accessed in parallel during the exchange with the round function and the state. The dimension of the buffer is a lane/word in depth and the length is determined by the largest block size and digest, r and ℓ. Since the message block of SHA-3 is 576 bits large and the digest is 512 bits, the former dictates the buffer size. The wrapper works in two main FSM states: user_com and state_com. In the former, the array is rotated downwards while words of the next message block are injected at the top address and the digest is extracted at the bottom. In the latter FSM state, the buffer is read in parallel and the complete message block is sent to the statereg while the outer part of the processed state is received from the roundfunc and written to the IO-buffer.

The IO-buffer can be decreased by one lane as the last word is fed directly from the input vector to the state. This should be done if the objective is to optimize an unfolded structure which is not unrolled or pipelined. However, since the main objective of this basic structure is to serve as a base for further development, this have been disregarded. Another solution for the IO-buffer and also the state register is to store them in distributed RAM. The slices of V-5 FPGAs and more modern models support distributed RAM solutions which are more efficient than storing with regular FFs even without using the depth. However, for the same reason as above and additionally for compatibility with other FPGA types, other solutions than the standard FFs-based registers have been disregarded for the basic structure.

The manner in which the wrapper is implemented allows for easy adaptation into more complex versions where the counter is multiplied and more multiplexers are required for output signal assignment.

5.1.2 The state

The main registers are in addition to the IO-buffer in the wrapper, the state of 1600 bits and the round constants. For the basic structure, nothing in particular is worth mentioning about these registers. Each slice contains four FFs so both registers occupy the FFs of 442 slices. Depending on the wrapper's start-off-round-flag, i.e. whether it is round 0 or 1-23, the state is either filled with data from the wrapper or the output of the round function. This is depicted in Figure 5.3.

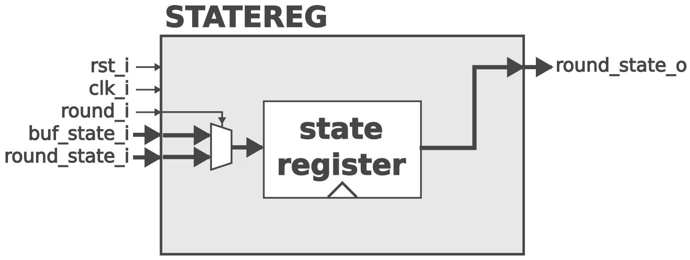


Figure 5.3: Schematic of the statereg component of the basic structure containing the state register.

5.1.3 Round function

The maximum possible resources are required for the implementation of the round function block in the basic structure, however the synthesis tool will automatically optimize the logic into the necessary number of LUTs. Figure 5.4 depicts Keccak-*f* acting on a column of the state and the corresponding logic for realizing the same functions, albeit on a fewer number of output bits.

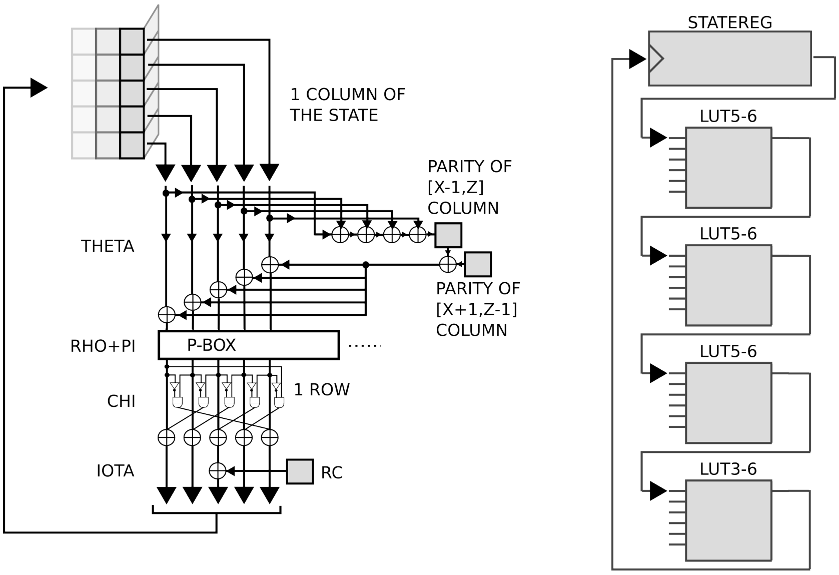


Figure 5.4: Illustration of Keccak-*f* acting on a column of the state on the left side and the same functionality realized with programmable logic.

The choice for the implementation of the ι step-mapping and the round constants have been based on the conventional approach from the existing literature. Athanasiou *et al.* [43] have considered both the options of pre-generating the Round Constants (RCs) and storing them in a distributed ROM and a circular buffer, or generating them on-the-fly with a LFSR. They conclude that the latter is the best alternative with respect to efficiency. Still, most other implementations have simply stored the pre-generated RCs in distributed ROM. For the basic structure, this have been the prioritized solution. The synthesis is efficient with the default settings as the round functions which are stored here are automatically optimized into the minimum required number of LUTs. The round constants are simply indexed by a round counter signal from the wrapper.

5.2 Generic structure

Implementing a generic wrapper which supports all four sub-versions of SHA-3 is uncomplicated for structures which are neither pipelined nor folded. The state and the round function are independent of the SHA-3 sub-version and the wrapper contains all components which must be modified in order to support them all. The IO-buffer is adapted from the basic structure so that it rotates and is simultaneously accessed in parallel. The size of the IO-buffer is dictated by the largest block size which is 1152 bits, or 18 lanes of 64 bits, for the SHA3-224 version

(Appendix 7.4.2). The IO-buffer stops rotating depending on the SHA-3 version selected. Unlike the approach by Athanasiou *et al.* [43] where multiple XOR modules make up the version control, only one XOR module is implemented here and for multi-block-messages, the complete IO-buffer is XORed with the outer part of the state. It is therefore important that the unused parts of the IO-buffer are empty when the other sub-versions are selected. A combination of the version selection input and the round counter selects whether the IO-buffer should work in FIFO or parallel mode. One of the four sub-versions are selected through a 2-bit input vector for the wrapper.

5.3 Pipelined structure

Using the basic structure as a foundation, one can proceed with implementing the explored optimization techniques. For the unfolded design, this means optimizing for efficiency. If the folding factor and small latency is to be maintained, then the number of relevant techniques are limited. As discussed in Chapter 4.3, the potential for increasing the efficiency relies on the possibility of utilizing pipelines. This again depends on the consideration of messages and support of the wrapper. Therefore, multi-block-messages are not considered here and only small messages are supported for hashing. This is simply a restriction caused by the sponge nature of Keccak and this consideration is valid for all implementations of SHA-3.

A structure with one internal pipeline has been considered which corresponds to PL=2. The manner in which it is incorporated and how the round function is partitioned is loosely based on the analysis performed by Pereira *et al.* [41]. While the number of pipelines incorporated in their proposed solution is found to be excessive, the study of propagation time of each step-mapping is still relevant and applicable. The top level of the pipelined structure is identical to the basic structure and the main modifications distinguishing these two are found in the implementation of the state register and the IO-buffer in distributed RAM and increased round counter of the wrapper. In the basic structure and in most designs in general, a portion of the occupied slices have unused FFs. This means that for the incorporation of the first pipeline registers, the area does not necessarily increase proportionally. The round function can therefore be split into multiple combinational paths and a more efficient slice utilization ensues. A precondition for this is that the slices are not saturated with control signals as this inhibits further utilization.

5.3.1 Wrapper

It is trivial to implement a pipelined structure which only supports small messages and is based on the basic structure. The XORs and multiplexers which are required for the multi-block--message support are removed and the additional latencies are compensated for by multiplying the round counter range of the basic structure with PL. The wrapper must also be able to handle the extra message blocks and digests. A new schedule (Appendix 7.4.3) is therefore constructed within the constraints of the 24 rounds and the width of the IO ports. With continuous use of the hash function without interrupted processing, the pipeline remains free of bubbles. As multiple blocks are contained by the IO-buffer, it becomes advantageous to utilize

RAM. No previous example of RAM use for a separate IO-buffer have been found in the existing literature. Each state must still be accessed in parallel so the number of occupied slices for the IO-buffer remains the same. For injection and extraction of words of the message and digest, the IO-buffer of the basic structure functioned as a rotating buffer. In distributed RAM, this is solved by multiplexers and demultiplexers which are implemented in LUTs.

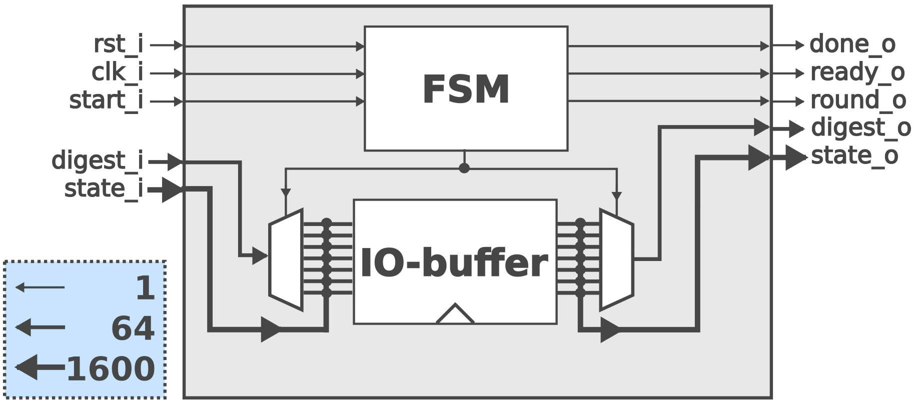


Figure 5.5: Schematic of the wrapper component of the pipelined structure.

5.3.2 Round function

With the incorporation of pipeline registers, the ι step-mapping must be modified as both messages need be processed by the same round constants. Since the round counter is doubled in range, the round constants are indexed by the counter divided by two. The structure with PL=2 is divided at the ρ and π step-mappings so that the ρ input is synchronized.

5.4 Unrolled structure

The unrolled structure is adapted from the pipelined structure, but with a further modification of the schedule (Appendix 7.4.4) with UF and PL=2, as depicted in Figure 5.6. A second state register is implemented between instance one and two of the round function and the internal pipelining scheme is identical to the pipelined structure with PL=2. According to the SHA3-512 sub-version specifications, four message blocks of $9 \cdot 64$ bits are injected into the IO-buffer within the clock cycles of the 24 rounds.

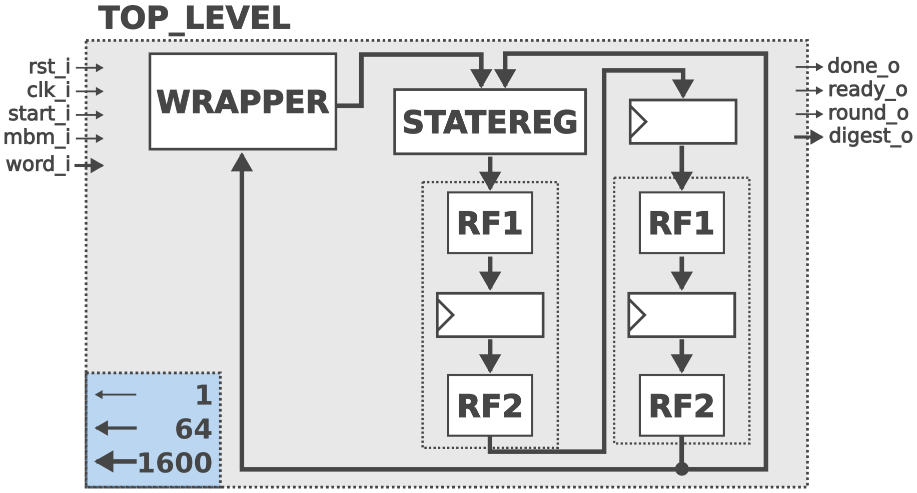


Figure 5.6: Simplified top level schematic of the unrolled structure with unrolled factor (UF)=2 and pipeline stages (PL)=2.

The ι step-mapping is more complicated here than in the pipelined structure as four messages must be processed by the same round constants during different clock cycles. This is solved by storing the precomputed round constants as two 7x47 ROMs with each 7-bit word replicated four times and divided between the roundfunc components. Each value of the round counter corresponds to a given round constant.

5.5 Folded structure

The folding factor affects all of the components of the design and the resulting complexity is noticeably higher than in the previously presented structures. With FF=4, the round function is reduced by 75% of its width and the IO-buffer and the state registers are reduced equally with the utilization of the depth of the RAM. With a latency of 96 clock cycles, the minimum IO-port size is limited to 12 bits and is rounded up to the next power of 2, thus corresponding to the number of bits of a lane fitting into each fold. In addition to solving the inter-round dependencies of ρ and the intra-round dependencies of θ, the implementation must consider the re-scheduled round function, the interfacing between the slice-wise structure and the standard of SHA-3, and the dependency of ι with the provision of the round constants. The intra-round dependencies caused by θ is solved either with extra logic or an extra clock cycle. Additionally, the memory mapping of the state must be done as efficiently as possible so that the least amount of slices are occupied. Each fold of a round x depends on processed bits from each of the sub-rounds of round $x - 1$.

Both solutions for the implementation of the re-scheduled round function have been considered. The standard solution on the left side of Figure 4.7 seems to be the most common alternative among the existing literature. It is not considered here as the best option because of a series of challenges related with its implementation. While both solutions need multiplexers to bypass R_f2, the embedded ρ step as well as the packing of bits into the memory blocks must be bypassed, thus requiring additional multiplexers. The alternative solution is therefore considered here and a simplified top level schematic of the folded structure with this re-scheduling solution is depicted in Figure 5.7. The latency is 24xFF as round 0 and 24 are processed during one clock cycle. This presents a trade-off between the size of the combinational path and the latency.

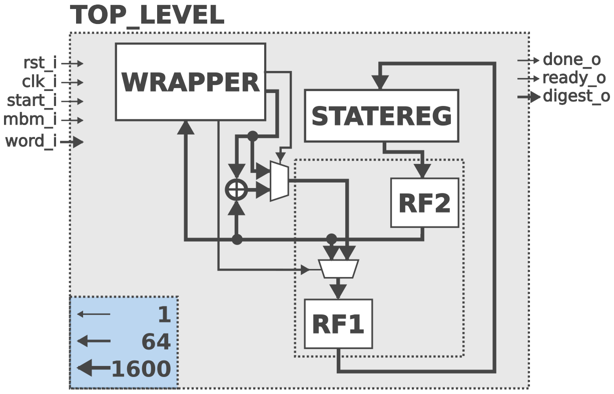


Figure 5.7: Simplified top level schematic of the folded structure with a re-scheduled round function.

No pipelining is utilized for this folded structure. The inter-round dependencies caused by ρ would further result in a bubble in the pipeline at the end of each round. The fold of the first sub-round must be updated with bits from the fold of the last sub-round. As Figure 4.14 suggests, this reduces the overall increase in efficiency because of the significant increase in latency. It is left for further exploration whether it is possible to attain a better efficiency despite the inter-round dependency.

5.5.1 Wrapper

The wrapper is adapted from the basic structure and sub-rounds are added by a sub-clock, counter_fold, which acts as an offset for the round counter. The modified schedule for the folded structure is given in 7.4.5. The performance objective of the wrapper is maximum efficiency so that the latency is kept at the minimum FFx24 at a cost in area requirements for the IO-buffer. This is distinctively different from the objective of the similar structure proposed by Jungk *et al.* [34, 37], which seem to be excessively focused on minimal area requirements. This objective is not optimal for the relevant folding-factor which should have a symmetric focus on both high throughput and area footprint.

Both the counter_fold and counter_round are used to control signals. The addressing of the IO-buffer is rotated for every sub-round while write enable and the other memory signals depend on the round counter. The MBM-support remains in the wrapper so that the input state is potentially XORed with the IO-buffer.

The round constants for the ι step-mapping are managed by the wrapper in the folded structure and modifications must be made so that the correct bits of the center lane of each fold is processed. ι depends on the center bits at z-coordinates 0, 1, 3, 7, 15, 31 and 63. This means that the processing of fold 0 requires the provision of bit 0 to 4 of each round constant, and fold 1 and 3 needs bit number 4 and 5 respectively. Therefore, the RCs are added to the ι step-mapping through a 5-bit word.

5.5.2 The state

The memory mapping is done with the lane-oriented solution (Section 4.2.3) as it is believed to be the best alternative for a slice-wise structure with a low folding-factor. The state is stored in distributed RAM and the slice utilization is therefore directly related with the folding-scheme. The utilized depth of the memory is equal to the FF and the width is equal to the total size of the state divided by FF. The lane-oriented memory mapping requires two instances of the state which are switched so that state A and B are read and written to, every other round. This solution is easily applied to both distributed RAM and BRAM. Despite distributed RAM being technically made up of fine-grained entities, they are treated as smaller memory blocks with two bytes width. The designation of memory blocks for each fold and the packing of lanes into each block is depicted in the Appendix 7.3.1. A large extra register is necessary if the bits are not packed together. This works in a roll-around manner so that for subsequent folds, the addressing of each memory block is simply incremented, as depicted in Table 5.1. Four different write addresses and one read address are used for the 24 RAM blocks depending on the sub-round and the state instance.

Cycle	0	1	2	3	4	5	6	7
Write	4	5	6	7	0	1	2	3
	5	6	7	4	1	2	3	0
	6	7	4	5	2	3	0	1
	7	4	5	6	3	0	1	2
Read	0	1	2	3	4	5	6	7

Table 5.1: Addressing of the state memory.

The FF=4 so that 400 bits of the state are accessed in parallel. Each SLICEM of a V-5 FPGA supports one 32x6bit Simple Dual Port (SDP) RAM. The LUTs of 67 slices should then be required to store both instances of the state. SDP RAM has one synchronous input port and one optionally synchronous output port. Both ports have individual addressing so that the concept of read and write collisions is admissible. If both ports have the same addressing, three options exist for the behavior of the read port: WRITE_FIRST, READ_FIRST and NO_CHANGE [17]. WRITE_FIRST means that whatever is written at a specific address will be available at the output as soon as possible. READ_FIRST means that the previous content at that address is shown at the output after data is written to memory. NO_CHANGE contains further latching behavior as the output remains the same as long as writing is enabled.

Write Enable (WE) is initialized to high as data is constantly written to the memory and the addressing is set by a multiplexer controlled by the fold counter from the wrapper. There are four folds and four sub-rounds of each round, but the two state instances requires a counter which distinguishes between two rounds and thus eight sub-rounds. A binary signal is therefore added to distinguish between which state to read and write from.

The state register is either filled with the output of the round function or from the wrapper, in a similar manner as the other structures. However, the folded structure is different as the state is packed together in order to fit into the correct fold dictated by the ρ permutation. The packing must be compensated for during the first round there must be a demultiplexer at the output in order to bypass the un-packing of the state.

5.5.3 Round function

The round function is narrowed down to 25% of its size in the basic structure and re-scheduled. Because of the re-scheduling, the round function is divided into RF1 and RF2, as presented in 4.2.1 and the two inputs of RF1, the buffer and RF2, are controlled by multiplexers.

The solution for the intra-round dependency of θ is contained in the round function. For the sake of novelty, the pre-processing solution is implemented first, where the F3S15 slice is obtained during the previous round and made available for the processing of F0S0 during sub-round 0. Round 0 is a special case where the parity of F3S15 is collected from the IO-buffer. For all subsequent rounds, the bits which belong to the future F3S15 are latched into a F3S15 register from the output of the round function. The separated F3S15 is therefore processed by a mini round function before the column parities are obtained and provided to the θ_2 sub-step-mapping. The ι step-mapping depends on the MSB of the round constants for processing of F3S15 which is separately provided by round constant function of the wrapper.

5.6 Implementation summary

Five distinct structures are presented here where one is folded and belongs to the mid-range class of SHA-3 implementations while the other four are unfolded. The folded structure is the implementation with the highest complexity, but achieves reduced area requirements. This structure is similar to one solution proposed by Jungk *et a.* [34, 37], but is unique in that it functions as a stand-alone entity with a wrapper component which includes an IO-buffer. The wrapper additionally converts the standard SHA-3 interfacing so that incoming message blocks and output digests are compatible with the internal slice-wise folding-scheme. The structure also incorporates a new solution for solving the intra-round dependency of θ.

The presented unfolded structures will have a higher throughput, but also a larger area requirement. Two of the unfolded structures are considered for hashing of messages of arbitrary size: a basic implementation which serves as a foundation for all of the presented structures and a generic implementation which supports all of the four sub-versions of SHA-3. The throughput of these structures are upper bound by the critical path through the un-partitioned round function. Based on the analysis of Chapter 4, these structures represent the theoretical limit for the achievable efficiency concerning the structural factors of a SHA-3 implementation. For applications where the hash function is limited to small messages with size equal or below the block size, two pipelined structures are presented with and without unrolling.

All structures incorporate a wrapper which handles the interface and control logic of the structure. This component is easily adaptable for modifications of the structure and is optimized for minimal latency and overall high efficiency. Combinations of optimization techniques beyond what is presented here are expected to increase the complexity of the wrapper as a multitude of circumstances and conditions must be considered. This is especially true with respect to the generic structure.

The theoretical models of Chapter 4 suggests that the folded structure obtains a lower throughput and overall efficiency than the unfolded structures. Furthermore, of the latter, highest efficiency should be achieved by the structures which have adopted pipelines. The unrolled structure is expected to obtain the best efficiency. While only a version with an internal pipeline is implemented, the analysis suggests that this can be disregarded without much reduction in efficiency.

6 Experimental evaluation

This chapter presents the experimental evaluation of the solutions proposed herein. Solutions are compared with both the relevant existing state-of-the-art and the theoretical models presented in CHapter 4. The proposed structures have been implemented on a xc5vlx50t Virtex-5 FPGA model and performances are based on these specifications. Each of the implemented structures have been validated by both behavioral and post-PAR simulation and the generated digests have been compared with Known-Answer-Tests (KATs) provided by the Keccak team [51]. The achieved performances herein presented are assessed with respect to both the theoretical models from Chapter 4 and the existing state-of-the-art. The presented results for the proposed structures were obtained using Xilinx ISE 14.7 considering the default parameters for the full implementation flow. Timing constraints have been used to arrive at the maximum obtainable frequency.

6.1 Results

The performances achieved for the structures in the related state-of-the-art and the ones herein proposed and highlighted in bold are listed in Table 6.1. The normal throughput and efficiency is presented as reported in the existing literature while the adjusted efficiency is obtained by the efficiency formula of Equation 4.2 for the SHA3-512 version. These two values can therefore differ for certain structures. Structures are sorted by the SHA3-512-adjusted efficiency and the structures at the top are all unfolded while the bottom structures have adopted some sort of folding. This table uses an identical notation as the one used in Table 3.2 in Chapter 3. All structures have been implemented on V-5, except where noted with asterisks. UF(1-9) indicates the unrolling factor, FF(1-9) the folding factor, and PL(1-9)X/PL(1-9) indicates the number of pipeline stages and whether these are internal or e(x)ternal. A structure noted with PL2 incorporates one internal pipeline register in addition to the main state register. B (buffer) and NB (no buffer) implies whether an IO-buffer has been included in the assessment. † and †† denote whether the results are based on synthesis or the source is unknown. Other results are confirmed to be based on PAR.

When comparing the existing state-of-the-art concerning basic unfolded structures [44, 47, 50, 40, 4] with the one herein proposed, named **Basic**, it is possible to note that different results are obtained. The basic structure achieves balanced area results compared with the similar existing structures which also includes a wrapper with an IO-buffer. Gaj *et al.* [4] is the structure with most structural similarities. While they present a similar area occupation, the frequency results are reasonably higher. Results are reportedly based on implementation and it is not certain whether this means PAR. What is certain is that the results have been obtained by ATHENa [29], which is an open source benchmarking environment which automatically determines the optimal options for implementation of a hardware design. This can explain the better timing results. The **Basic** structure herein presented can be used as a good basis of comparison for future implementations considering hashing of both small and large messages. The presented results are based on realistic assessment and the structure functions as a stand-alone entity.

The **Generic** structure differs from the **Basic** structure in that the wrapper supports all four sub-versions of the SHA-3 hash function. It can be seen that this structure requires slightly more area while the frequency is slightly lower compared with the **Basic** structure. However, the critical path is located through the round function in both structures and goes through the same levels of logic. The difference in frequency is therefore considered to be insignificant. The larger area is caused by the larger IO-buffer and additional XORs located in the wrapper for support of the SHA-3 sub-version with the largest block size. The **Generic** structure presented herein requires much less area than the structure by Athanasiou [43]. They seem to incorporate multiple XOR modules in their version control component which is excessive. Also, no IO-buffer seem to be included in their proposal. They have chosen to incorporate an internal pipeline stage in the round function but does not mention any consideration regarding the size of messages. The pipeline stage explains the higher frequency and the higher overall efficiency compared with the **Generic** structure presented herein.

The **Pipelined** structure is achieving a noticeably better frequency than the **Basic** structure, as expected. The expected additional cost in area is effectively none as the occupied slices of the **Basic** structure, which function as the foundation of the **Pipelined** structure, are utilized inefficiently with many unused flip flops. With this pipeline, a high throughput is achieved with no cost in area, resulting in a higher efficiency (of 6 Mbps/slice). It is worth noting that if the area had increased similarly to the model from 4.3, it would occupy 1550 slices and the efficiency would be inferior to the **Basic** structure. The frequency would have to exceed 390 MHz for a similar increase in efficiency with one internal pipeline. However, the practical results are expected to catch-up with the model as PL increases. While Gaj *et al.* [4] have explored both a pipelined and basic structure, they have failed to incorporate the pipeline registers internally. Akin *et al.* [40] presents both an internally pipelined and basic structure but these results are based on synthesis. The results presented here clearly demonstrate the cost and benefit of pipelining within realistic circumstances and the lower accuracy of the proposed model when dealing with an increase in registers.

The best results in efficiency and performance are achieved when unrolling the structure. The frequency obtained for the developed **Unrolled** structure are inferior to both structures by by Ioannou *et al.* [44] and comparable to Gaj *et al.* [4] and Jararweh *et al.* [50] which are structures without any internal pipelines. In one way, this emphasizes the highly deviating timing performances of each structure, despite the utilization of solid optimization techniques such as pipelining. The frequency of the **Unrolled** and the **Pipelined** structures are similar which is expected as the critical path should be identical. The performance and area results of the **Un-**

rolled structure reasonably matches the estimation by the proposed model (in Section 4.4). The cyan line of the graph of Figure 4.17 represents the specification of the implemented **Unrolled** structure and the efficiencies values satisfactory match the actual obtained values.

In regard to the proposed **Folded** structure, the obtained area results are lower than those presented by Jungk *et al.* [34, 37], the most efficient slice-wise folded structure in the related state-of-the-art. This is despite one of their structures having a lower folding-factor. Moreover, they do not include IO-buffers. The higher area occupation is likely caused by excessive amount of registers incorporated in their proposed solution. Delay-wise, the design herein proposed is also able to achieve higher frequencies, resulting in an overall efficiency gain of 28% (2.61/2.04). This improvement is partially due to the adopted solution for the intra-fold dependencies of θ and the careful scheduling adopted. The folded structure with the second best performance is the one proposed by San & At [28]. This structure presents a record low latency for a lane-wise folded structure. However, it is not clear what the timing results are based on and the frequency is very high for a Virtex-5 implementation. The performance of the **Folded** structure is lower than the estimation by the theoretical model for folding (Section 4.1). This is caused by a miss-approximation of the frequency. However, the frequency seem to match quite well for an unfolded structure when comparing with the performance of the **Basic** structure. As discussed, approximations of the frequency are challenging because of the many factors involved. The area consumption of the folded structure is quite comparable to the results attained by with their similarly folded structure. They do not include an IO-buffer and the deviation in the required area The timing performance is also comparable. A source of deviation is the different solutions adopted for solving the intra-fold dependencies of θ.

Two aspects separate the solutions proposed herein from a majority of the existing state-of--the-art. First of all, only a few of the existing papers report performances which are assured to be based on a reliable assessment such as Place and Route results. The proposed solutions presented here are evaluated by both behavioral and post-Place and Route simulation and the computed digest compared with Known-Answer-Tests provided by the Keccak team [51]. The timing performance and occupied area are also obtained from Place and Route. The last aspect is related with the complete nature of each implementation. Most of the relevant existing literature present solutions which does not function as stand-alone entities as many are missing a wrapping component which handles the interfacing with a user or processor. This further highlights the incomplete status of the performance reported by selections of the state-of-the-art. All solutions proposed herein include a wrapper component which handles the necessary interfacing and is easily modified in order to support various optimization techniques.

Paper	Latency (clk cycles)	*f* (MHz)	A (slices)	T (Gbps)	T/A (Mbps/slice)	SHA-3 version	T/A (adjusted) (Mbps/slice)	Note
Unrolled	12	**287**	1967	13.78	**7.00**	512	7.00	UF2.PL2.B
Ioannou [44]	12	352 ††	2652	16.90 ††	6.37	512	6.37	UF2.PL2X.NB
Pipelined	48	273	1163	**7.80**	6.06	512	6.06	PL2.B
Ioannou [44]	24	382 ††	1581	9.17 ††	5.79	512	5.79	NB
Athanasiou [43]	48	389	1702	18.70	10.98	224	5.48	PL2.NB
Gaj [4]	24	283 ††	1272	12.82 †	10.08	256	5.34	B
Basic	24	223	**1192**	5.35	4.49	512	4.49	B
Baldwin [47]	25	189	1117	8.50	4.32	512	4.06	NB
Generic	24	**201**	**1268**	4.83	3.80	512	3.80	B
Jungk *et al.* [37]	24	195	1305	8.49	3.87	256	3.59	NB
Pereira [41]	100	452 †	3117	7.70 †	2.47	?	3.34	PL4.B
Akin* [40]	121	509 †	4356	22.33 †	5.13	224	2.69	PL5.B
Baldwin [47]	25	189	1971	8.50	4.32	512	2.38	B
Jararweh [50]	24	271 †	2828	12.28 †	4.34	224	2.29	B
Akin* [40]	25	143 †	2024	6.07 †	3.00	224	1.63	B
Folded	96	**200**	**460**	1.20	**2.61**	512	2.61	FF4.B
San & At [28]	1062	520 ††	151	0.25	1.66	512	1.66	FF25.BRAM.B
Jungk *et al.* [37]	50	144	**914**	3.13	**2.04**	256	1.81	FF2.NB
Jungk *et al.* [37]	100	150	**489**	1.63	1.99	256	1.76	FF4.NB
Jungk *et al.* [37]	200	166	301	0.90	1.79	256	1.59	FF8.NB
Jungk & Ap [34]	200	159	393	0.46	1.17	256	1.17	FF8.NB
Jungk & St [39]	1665	257	90	0.17	1.85	256	0.99	FF64.B
Winderickx [36]	1730	248 ††	134	0.25	1.16	256	0.66	FF64.B
Jungk *et al.* [37]	1600	206	164	0.14	0.51	256	0.45	FF64.NB
Kerckhof** [33]	2154	250 ††	144	0.07	0.47	512	0.47	FF25.B
Bertoni [12]	5160	265	448	0.05	0.12	512	0.12	FF25.B

Table 6.1: Existing SHA-3 implementations sorted by efficiency adjusted for SHA3-512. Unfolded structures on top and folded structures below. FF(1-9)=folding factor, UF(1-9)=unrolling factor, PL(1-9)X/PL(1-9)= external/internal pipeline registers, NB/B=no buffer/buffer. *Virtex-4 implementation. **Virtex-6 implementation. † Results obtained from synthesis. ††Unknown source of results.

6.2 Further evaluation

The choice of FPGA family clearly impacts the performance, both in frequency and slice utilization. Smaller transistor size improves routing delay and the content of a slice differs depending on how many inputs each LUT contains. Much of the literature [44, 4, 50, 43] has compared the varying performances between both older and more modern FPGA families. For the Virtex families, when migrating from V-4 to V-5 the frequency tend to increase between 20-40% and the area decreases between 30-50%. From V-5 to V-6, the frequency increases between 10-20% and the area decreases between 3-30%. The tendencies are similar for V-7 and pipelined structures with many registers improve greatly with the newer technologies.

7 Conclusion

The main goal of this work has been to improve the existing state-of-the-art with respect to efficient hardware implementations of the SHA-3 hash function. While a special focus has been made on FPGAs, as the prototyping technology, the aim has been to keep the presented solutions as general as possible with respect to the supporting technology. Several solutions are herein proposed in the form of SHA-3 hardware structures with distinct performance objectives and considerations which are found to exceed the performance of the state-of-the-art.

The contributions of this work consists of a selection of proposed solutions evaluated for high-performance and theoretical models which through approximations demonstrate the upper bound efficiency of SHA-3 hardware implementations with respect to relevant optimization techniques. Additionally, the necessary preconditions and considerations for the utilization of the relevant optimization techniques are demonstrated.

Five structures were proposed in this thesis. A basic structure, representing the most straight forward implementation of the SHA-3 algorithm, is used as a basis of comparison for the existing state-of-the-art and the structures herein proposed; a generic structure, supporting all four sub-versions of SHA-3 which is a functionality that has been scarcely covered by the existing literature; an unrolled structure which contains two instances of the round function; a purely pipelined structure with an extra state register implemented in the round function; a folded structure which reduces the area requirements by processing parts of the state per clock cycle.

The generic and the basic structure are largely identical and differ only by way of the wrapper component which handles the loading of messages and digest to and from the core structure. The pipelined structure obtains a higher frequency than the basic structure and the area requirements is effectively the same. Despite containing one extra state register, it is merely implemented into already occupied slices of the basic structure. The unrolled structure is larger than the basic structure, but provides an effective decrease in latency by 50%. It also contains a pipeline stage in the round function similarly to the pipelined structure. The folded structure belongs to the mid-range class of structures as the internal path is reduced while a certain level of throughput is maintained. From these, two structures should be highlighted as they provide an improvement to the existing state-of-the-art and achieve the best efficiency metrics for the class of structures they represent. These are the unrolled structure with an internal pipeline stage and the folded structure with a folding factor of 4.

The unrolled structure allows for achieving a throughput of 13.8 Gbps with a cost of 1967 slices, on a Virtex-5 FPGA. This yields an efficiency of 7 Mbps/slice, which is 9% better than the best state-of-the-art to date [44]. This is achieved by multiplying all components of the structure apart from the wrapper containing the control logic. This results in the latency being reduced by 50% while the area is increased by less than 100% as the control logic is not multiplied. It is important to increase the number of state registers proportionally so that the frequency is not decreased. A combination with internal pipelining is proven to be effective.

Both the unrolled and the purely pipelined structure have internal pipeline stages in the round function. This is a key factor for the reduction of the critical path and increase of the efficiency. As such, pipelining and the combination of pipelining and unrolling are effective techniques for improving the efficiency. However, as is argued here, the relevancy of the adoption of these techniques is limited by the messages which are considered for hashing. A precondition for the success of pipelining is that bubbles are prevented so that all pipeline stages are full during each clock cycle. This will not be the case for messages which are larger than the block size. The sponge function nature of SHA-3 specifies that each block is processed by the round function for 24 rounds before being merged with the subsequent block of the same message. Therefore, only blocks of unrelated messages are able to fill consecutive pipeline stages. As unrolling is inherently inefficient without pipelining, these considerations are relevant for both optimization techniques. Distinctly higher efficiencies than the basic structure are only found to be achieved by structures which implement pipelining. Where hashing of arbitrarily sized messages is considered, the basic structure represents roughly the upper bound efficiency of SHA-3 hardware implementations.

The other proposed structure that significantly improves the state-of-the-art is the folded structure. This structure considers a folding factor of 4, meaning that 25% of the state is processed in each clock cycle. This is the only structure presented which economizes in area requirements. As herein explored, the ensuing trade-off between area requirements and throughput is distinctively asymmetric. This means that a compact structure will inevitably obtain a much lower throughput and efficiency than a fully unfolded one. This folded structure achieves a hashing throughput of 1.2 Gbps at a cost of 460 slices, on a Virtex-5 FPGA. This yields an efficiency of 2.6 Mbps/slice. Clearly, this achieved efficiency is significantly lower than the unrolled structure herein proposed. Nevertheless, it is the folded structure which presents better efficiency metrics when compared with the related folded state-of-the-art, being 28 % more efficient than the best slice-wise folded structure of the state-of-the-art [37]. This improved efficiency is achieved by an efficient structure which only contains the absolute necessary registers for a proper functionality. A memory mapping solution herein suggested, is adopted, which limits the required RAM units to the bare minimum of what is required with a small extra register. The intra-round dependencies caused by θ are solved by a novel solution which allows for slice 0 to be stored in RAM in contrast with the existing solution.

From this, it can be concluded that the use of the folding technique should be motivated by the particular objective of reducing area resources. However, the higher the folding factor, the lower the overall efficiency that can be reached.

7.1 Future work

Interesting future research directions involve the further exploration of combinations of the discussed optimization techniques for unfolded structures. Also, a highly unrolled structure with a generic SHA-3 sub-version support can be very useful. For optimal stand-alone functionality, a padder component could be incorporated in the wrapper.

The developed wrapper component also has room for improvement. The control logic can be implemented more efficiently with a faster counter and the IO-buffer can be reduced by one lane as this can be fed directly from the input port to the state register.

Compact structures have been thoroughly covered and no other specific techniques has been discovered which are worth implementing beyond what already exists in the state-of-the-art, regarding this class of designs. An exception would be to evaluate the use of DSP slices when considering the used multiple pipelines. This is not a very critical addition to the state-of-the-art, however, it could allow for a more balanced use of the resources available in modern FPGAs.

Structures with other folding factors are also worth evaluating. Specifically, a structure with FF=8 which could allow for an internal pipeline to be implemented in the round function. This may allow for improved performances.

References

[1] Valerie Aurora. "Life-cycle of cryptographic hash functions ", 2012. http://valerieaurora.org/hash.html.

[2] H. Gilbert and H. Handschuh. "Security Analysis of SHA-256 and Sisters", 2004. http://link.springer.com/chapter/10.1007/978-3-540-24654-1_13.

[3] D. J. Bernstein and T. Lange. "eBACS: ECRYPT Benchmarking of Cryptographic Systems ", 2012. http://bench.cr.yp.to/results-sha3.html.

[4] M. Rogawski E. Homsirikamol and K. Gaj. Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs. *Researchgate*, 2011.

[5] Erhan Kartaltepe. Properties of Secure Hash Functions - DenimGroup. http://www.denimgroup.com/know_artic_secure_hash_functions.html.

[6] A. J. Menezes, P. C. V. Oorschot and S. A. Vanstone. "Handbook of Applied Cryptography", 1996.

[7] Wolfram Mathworld. One-Way Function. http://mathworld.wolfram.com/One-WayFunction.html.

[8] Kristian Edlund. The pigeon hole principle. http://www.mathblog.dk/pigeon-hole-principle/.

[9] X. Wang, Y. L. Yin and H. Yu. Finding collisions in the full SHA-1, 2005. http://www.impic.org/papers/wang_sha1_v2.pdf.

[10] Bruce Schneier. Cryptoanalysis of SHA-1, 2005. http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html.

[11] L. Shannon and V. Cojocaru. "Technology Schaling in FPGAs - Trends in Applications and Architectures", 2015. http://fccm.org/2015/pdfs/M1_P1.pdf.

[12] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche and R. V. Keer. "Keccak implementation overview Version 3.3", 2012. http://keccak.noekeon.org/Keccak-implementation-3.2.pdf.

[13] Altera Robert Cottrell. "FPGA Coprocessors: Hardware IP for Software Engineers". http://www.design-reuse.com/articles/6733/fpga-coprocessors-hardware-ip-for-software-engineers.html.

[14] G. Steiner, K. Shenoy, D. Isaacs (Xilinx), and D. Pellerin (ImpulseC). "How to accelerate algorithms by automatically generating FPGA coprocessors". 2006. http://www.embedded.com/print/4014843.

[15] J. Edwards. No room for second place. *EDN*, 2006. http://www.edn.com/Home/PrintView?contentItemId=4320763.

[16] P. Tanner. Inside Xilinx's Lead over Altera in the 20- and 16-Nanometer Markets. *Market Realist*, 2016. http://marketrealist.com/2016/05/inside-xilinxs-lead-altera-20-nm-16-nm-market/.

[17] Xilinx. "Virtex-5 FPGA User Guide - UG190", 2012. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[18] Kevin Morris. "FPGA Synthesis Showdown The Big Game that Never Ends". 2016. http://www.fpgajournal.com/archives/articles/20160119-synthesis/.

[19] Xilinx. "Implementation Overview for FPGAs". http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_implement_fpga_design.htm.

[20] G. Bertoni, J. Daemen, M. Peeters and G. V. Assche. "Cryptographic sponge functions". 2011. http://sponge.noekeon.org/.

[21] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. "The RadioGatún Hash Function Family". . http://radiogatun.noekeon.org/.

[22] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche and R. V. Keer. "Keccak sponge function family main document ", 2008. http://keccak.noekeon.org/Keccak-main-1.0.pdf.

[23] M. Peeters G. Bertoni, J. Daemen and G. V. Assche. "the keccak reference version 3.0", 2011. http://keccak.noekeon.org/.

[24] National Institute of Standards and Technology - US Department of Commerce. "Approved hashing algorithms ", 2015. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[25] The Keccak team. "Note on Keccak parameters and usage ", 2015. http://keccak.noekeon.org/NoteOnKeccakParametersAndUsage.pdf.

[26] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche and R. V. Keer. "Keccak SHA-3 Submission, 6.1 Bit and byte numbering ", 1011. http://keccak.noekeon.org/Keccak-submission-3.pdf.

[27] National Institute of Standards and Technology. FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions , 2015. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[28] I. San and N. At. Compact Keccak Hardware Architecture for Data Integrity and Authenticaion on FPGAs. *Information Security Journal: A Global Perspective, 21*, pages 231–242, August 2012.

[29] "ATHENa - Automated Tool for Hardware EvaluatioN". `https://cryptography.gmu.edu/athena/index.php?id=about`.

[30] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. "Report on the Development of the Advanced Encryption Standard (AES)". 2000. `http://csrc.nist.gov/archive/aes/round2/r2report.pdf`.

[31] Roar Lien. "FPGA Implementations of SHA-1 Secure Hash Standard". 2003.

[32] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche and R. V. Keer. "Keccak implementation overview Version 3.0", 2011. `http://keccak.noekeon.org/Keccak-implementation-3.0.pdf`.

[33] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. M. de Dormale, and F. X. Standaert. Compact FPGA Implementations of the Five SHA-3 Finalists. *Researchgate*, January 2011.

[34] B. Jungk and J. Apfelbeck. Area-efficient FPGA Implementations of the SHA-3 Finalists. *IEEE*, 2011.

[35] Xilinx. "LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)". 2010. `http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf`.

[36] J. Winderickx, J. Daemen, and N. Mentens. "Exploring the Use of Shift Register Lookup Tables for Keccak Implementations on Xilinx FPGAs". *26th International Conference on Field-Programmable Logic and Applications, Lausanne*, 2016.

[37] B. Jungk, M. Stöttinger, and M. Harter. Among Slow Dwarfs and Fast Giants: A Systematic Design Space Exploration of KECCAK. 2013. `jungkshrink`.

[38] B. Jungk. "FPGA-based Evaluation of Cryptographic Algorithms - PhD Dissertation", 2016.

[39] Bernhard Jungk and Marc Stöttinger. Hobbit - Smaller But Faster Than A Dwarf: Revisiting Lightweight SHA-3 FPGA Implementations. *ReConFig 2016*, 2016.

[40] O. C. Ulusel A. Akin, A. Aysu and E. Savas. Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa, Blue Midnight WIsh for Single- and Multi-Message Hashing. *SINCONF, Taganrog, Russia*, pages 168–177, September 2010.

[41] F. Pereira, E. Ordonez, I. Sakai and A. Souza. "Exploiting Parallelism on Keccak: FPGA and GPU Comparison", 2013.

[42] Y. Ayuzawa, N. Fujieda, and S. Ichikawa. "Design Trade-offs in SHA-3 Multi-Message Hashing on FPGAs", 2014.

[43] G. P. Makkas G. S. Athanasiou and G. Theodoridis. High Throughput Pipelined FPGA Implementation of the New SHA-3 Cryptographic Hash Algorithm. *IEEE*, 2014.

[44] H. E. Michail L. Ioannou and A. G. Voyiatzis. High Performance Pipelined FPGA Implementation of the SHA-3 Hash Algorithm. *MECO*, pages 1–4, 2015.

[45] Ken Chapman. "Xilinx WP274 Multiplexer Selection, white paper". 2006. http://www.xilinx.com/support/documentation/white_papers/wp274.pdf.

[46] Xilinx. "XtremeDSP 48 Slice. http://www.xilinx.com/technology/dsp/xtremedsp.htm.

[47] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. O'Neill and W. P. Marnane. "FPGA Implementations of the Round Two SHA-3 Candidates". 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/BALDWIN_FPGA_SHA3.pdf.

[48] Ambarish Vyas. "Implementation and Benchmarking of Padding Units and HMAC for SHA-3 Candidates in FPGAs and ASICs", 2011. http://digilib.gmu.edu/jspui/bitstream/handle/1920/7512/Vyas_thesis_2011.pdf?sequence=1.

[49] M. Rogawski E. Homsirikamol and K. Gaj. "GMU hardware interface for cryptographic modules". 2010. https://cryptography.gmu.edu/athena/index.php?id=interfaces.

[50] H. Tawalbeh Y. Jararweh, L. Tawalbeh and A. Moh'd. Hardware Performance Evaluation of SHA-3 Candidate Algorithms. *Journal of Information Security*, pages 69–76, April 2012.

[51] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. "Known-Answer-Tests and Monte Carlo test results". . http://keccak.noekeon.org/KeccakKAT-3.zip.

Appendix

7.2 Dependencies

7.2.1 Rho dependencies for fold 0 of slice-wise folding-schemes

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

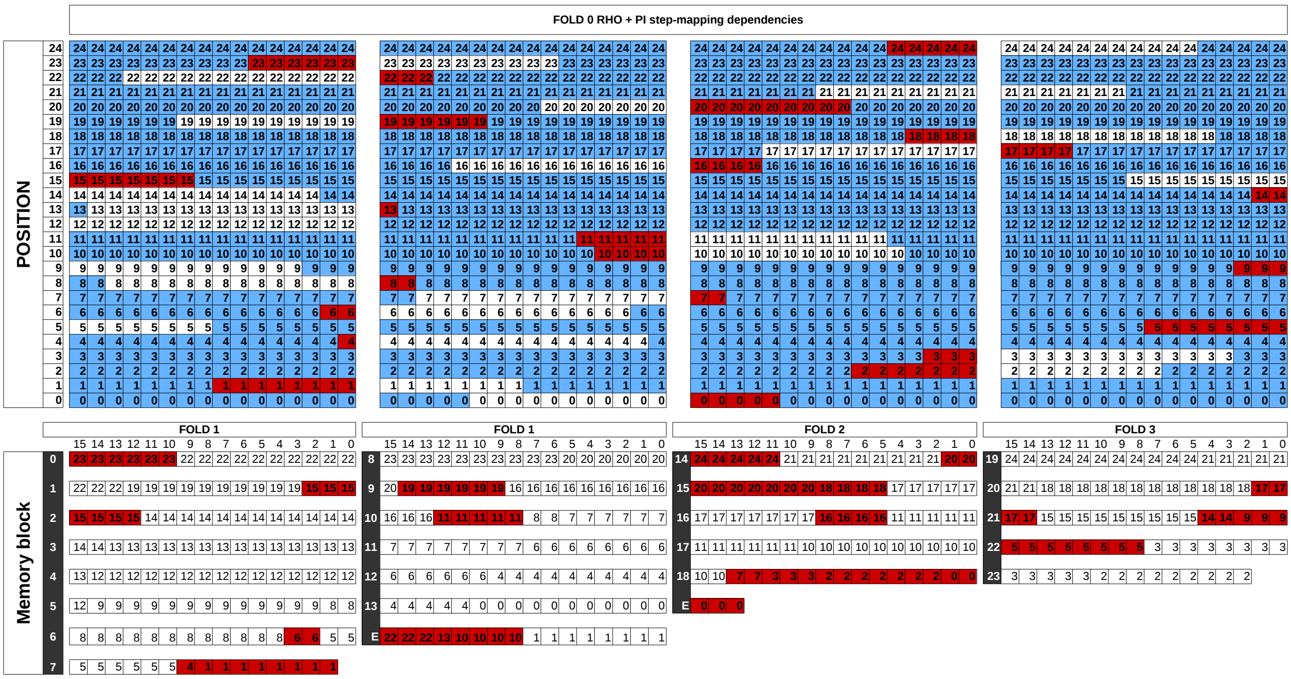
		SLICE NUMBER																																																													
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
LANE NUMBER	24	12	12	12	12	12	12	12	12	12	12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

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

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

7.3 Memory mapping

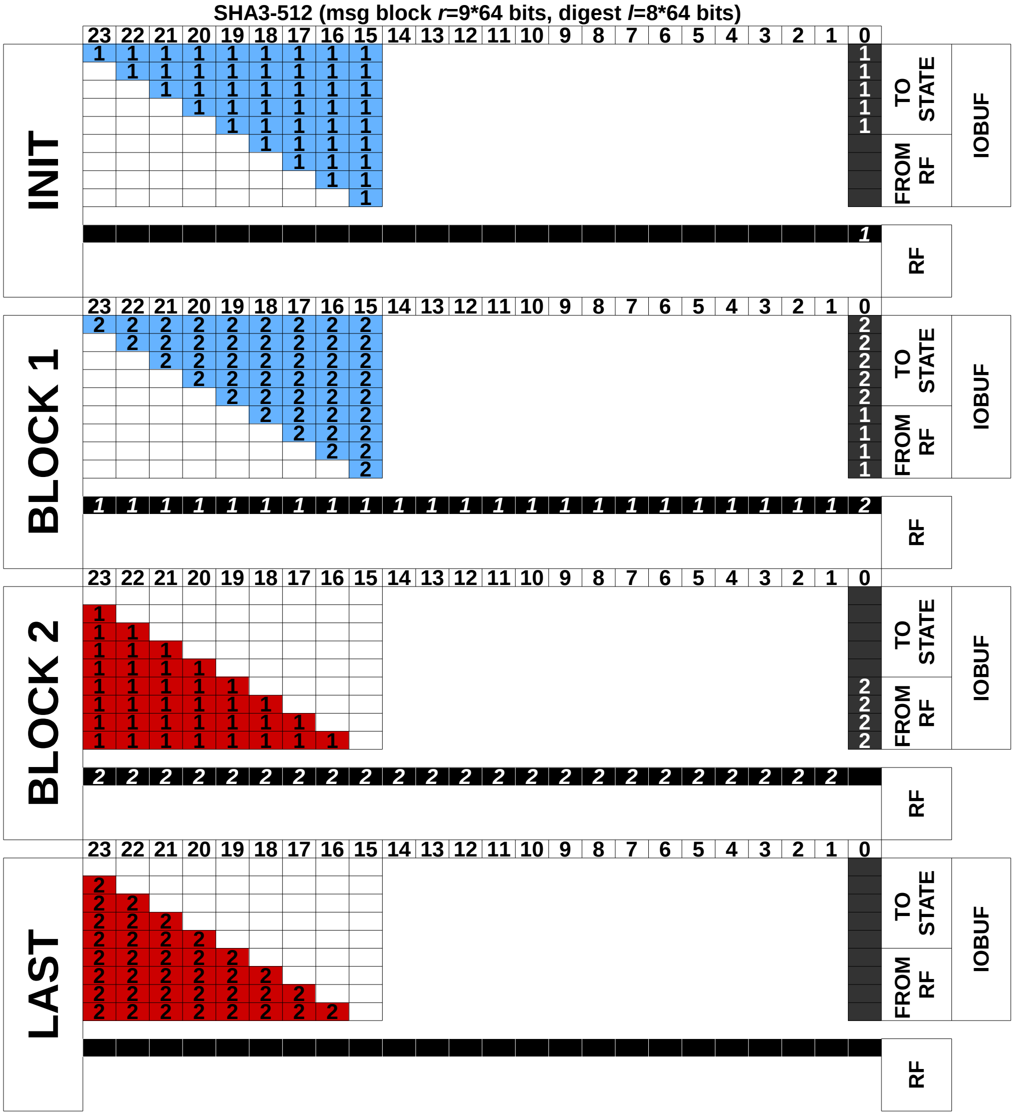
7.3.1 Memory mapping of state for the folded structure with FF=4



1-24	Lane number
1-23	Memory block number
E	Extra register
□	Standard dependency bits fitting in memory blocks designated for each fold
🟥	Dependency bits for Fold 0 which does not fit in straight forward memory mapping (only when packed)
🟦	Non-dependency bits for Fold 0

7.4 Scheduling

7.4.1 Basic structure SHA3-512



1-9 Indication of block number

⬛ Message+digest exchange with RF and statereg

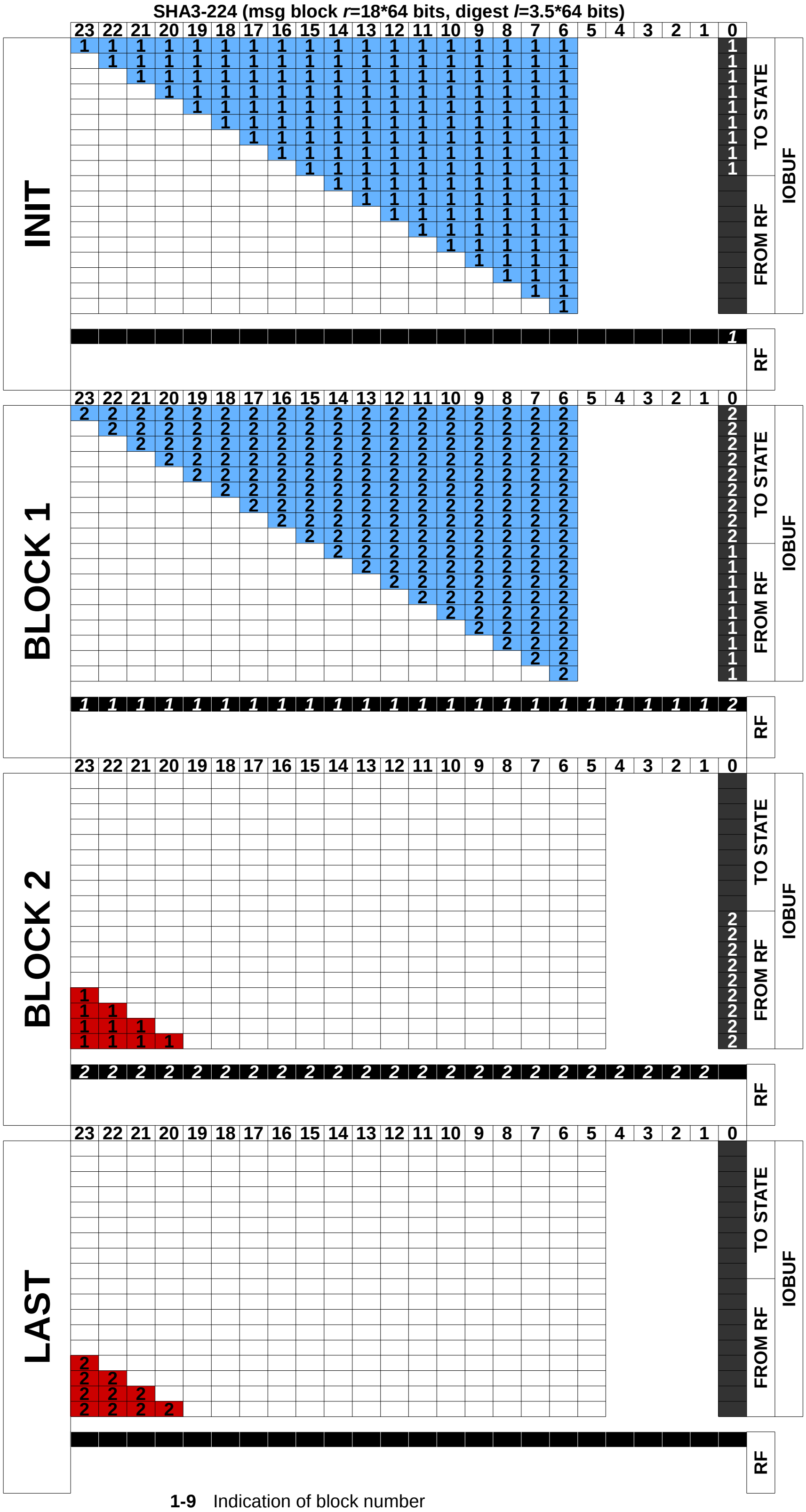
🟥 Digest output

🟦 Message input

1-9 Processing of numbered block

7.4.2 Basic structure SHA3-224

SHA3-224 (msg block *r*=18*64 bits, digest *l*=3.5*64 bits)



1-9 Indication of block number

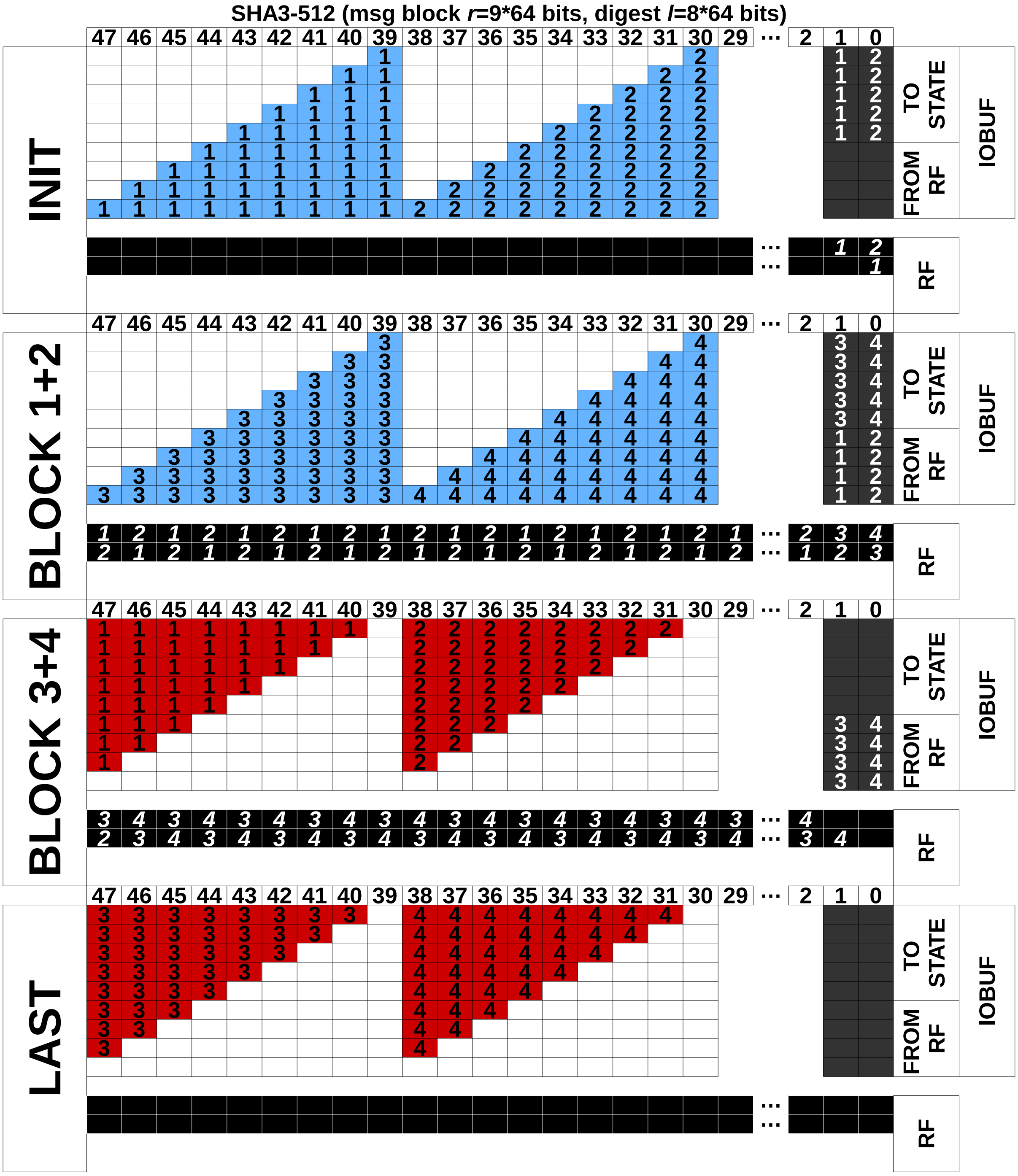
■ Message+digest exchange with RF and statereg

🟥 Digest output

🟦 Message input

1-9 Processing of numbered block

7.4.3 Pipelined structure SHA3-512



1-9 Indication of block number

■ Message+digest exchange with RF and statereg

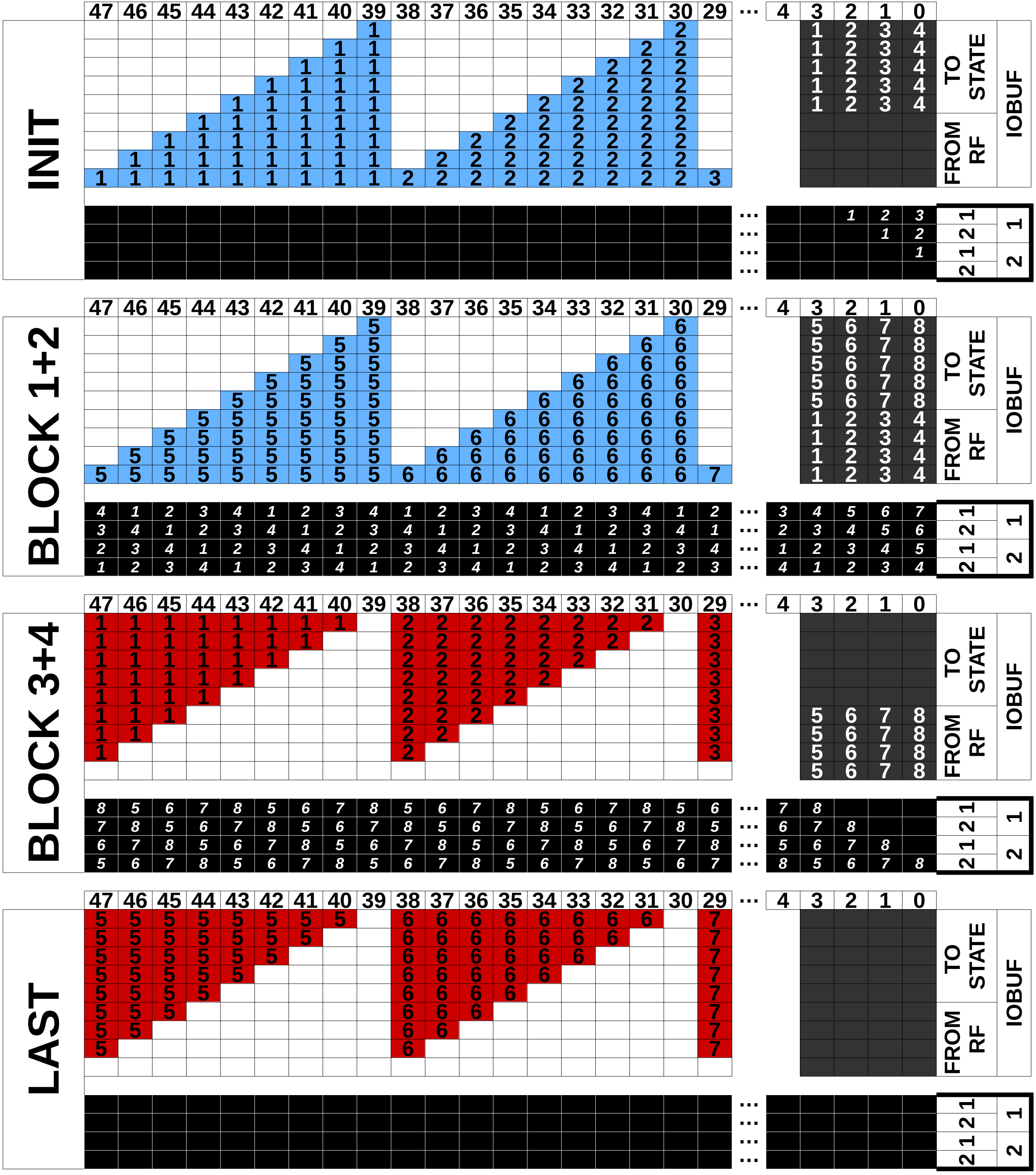
🟥 Digest output

🟦 Message input

1-9 Processing of numbered block

7.4.4 Unrolled pipelined structure SHA3-512

SHA3-512 (msg block r=9*64 bits, digest l=8*64 bits)



1-9 Indication of block number

■ Message+digest exchange with RF and statereg

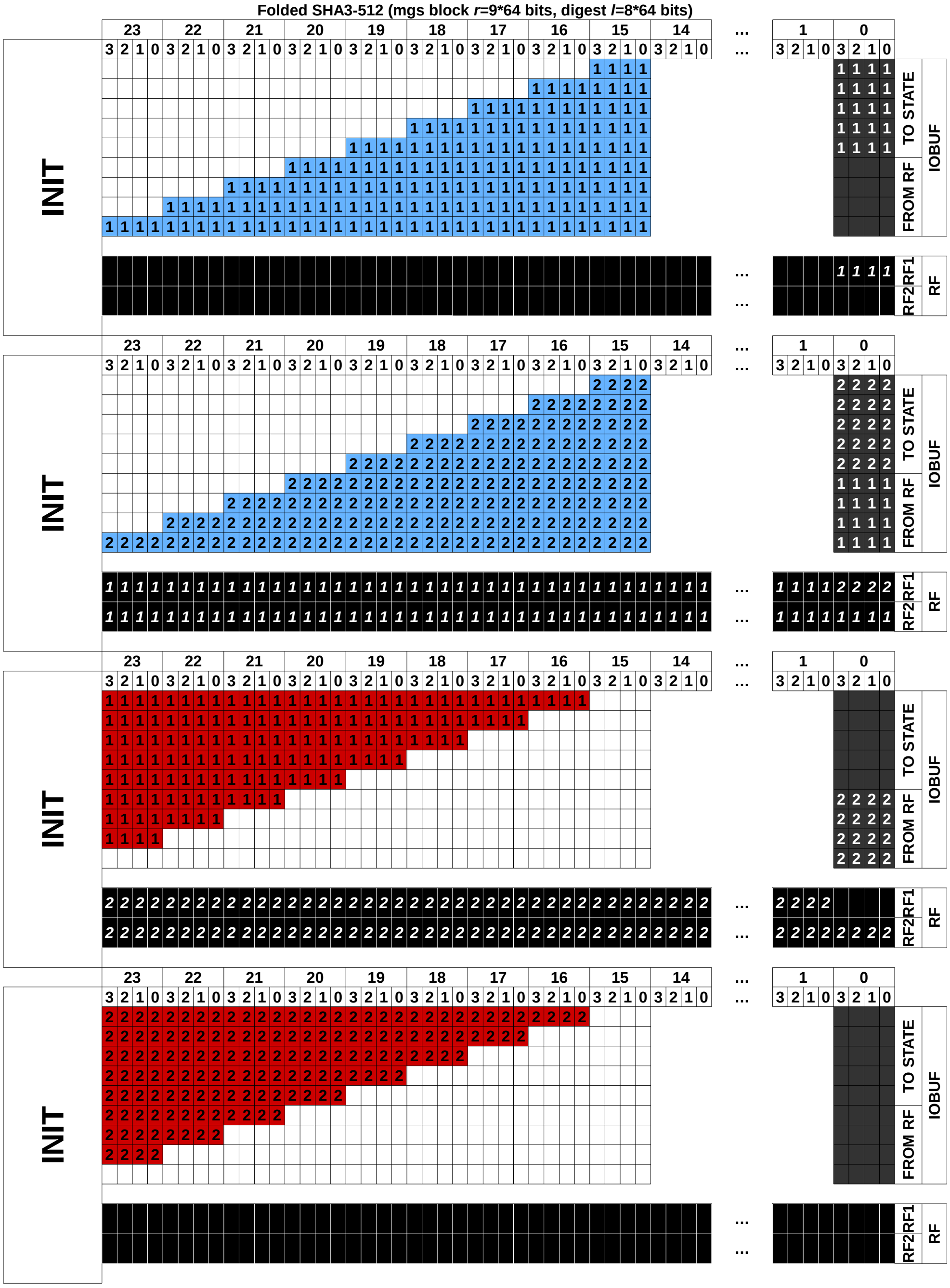
🟥 Digest output

🟦 Message input

1-9 Processing of numbered block

7.4.5 Folded structure SHA3-512

Folded SHA3-512 (mgs block *r*=9*64 bits, digest *l*=8*64 bits)



1-9 Indication of block number

⬛ Message+digest exchange with RF and statereg

🟥 Digest output

🟦 Message input

1-9 Processing of numbered block