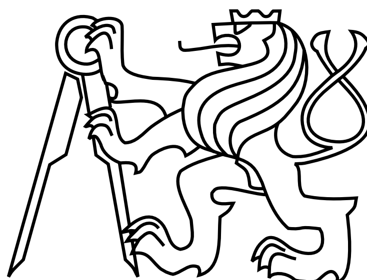
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Microelectronics



Diploma Thesis

Design of the HW accelerator of the KECCAK hash function

Nikita Litvishko

Supervisor: Prof. Ing. Pavel Hazdra, CSc.
Second Supervisor: Ing. Leoš Kafka, Ph.D.

Study Program: Electronics and Communication, Master

Field of Study: Electronics

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Litvishko** Jméno: **Nikita** Osobní číslo: **456867**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra mikroelektroniky**

Studijní program: **Elektronika a komunikace**

Specializace: **Elektronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Návrh HW akcelerátoru Keccak hashovacího algoritmu pro SoC platformu

Název diplomové práce anglicky:

Design of the HW accelerator of the Keccak hash function.

Pokyny pro vypracování:

1. Navrhněte akcelerátor pro Keccak hashovací algoritmus. Vyberte optimální interface pro integraci akcelerátoru v digitálním SoC (System-on-Chip).
2. Implementujte akcelerátor na úrovni RTL v jazyce Verilog2001. Ověřte funkci akcelerátoru simulací na RTL úrovni a validací na FPGA desce.
3. Porovnejte výkon akcelerátoru se softwarovou implementací tohoto hashovacího algoritmu ve vestavěném procesoru v FPGA. K softwarové implementaci použijte jádro Xilinx MicroBlaze.

Seznam doporučené literatury:

[1] The KECCAK reference, https://keccak.team/files/Keccak-reference-3.0.pdf
[2] KECCAK implementation overview, https://keccak.team/files/Keccak-implementation-3.2.pdf
[3] KECCAK in C, https://keccak.team/software.html https://keccak.team/obsolete/KeccakReferenceAndOptimized-3.2.zip

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Pavel Hazdra, CSc., katedra mikroelektroniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Ing. Leoš Kafka, Ph.D., Adesto Technologies

Datum zadání diplomové práce: **28.01.2020** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **30.09.2021**

prof. Ing. Pavel Hazdra, CSc. podpis vedoucí(ho) práce	prof. Ing. Pavel Hazdra, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Aknowledgements

I would like to thank my school supervisor Pavel Hazdra for valuable comments on the thesis. Also I would like to express my gratitude to the second supervisor, Leoš Kafka, for his excellent management of the thesis, weekly consultations, professional advices both in the field of digital design and in organizing time and work, as well as for help in difficult places.

I also wish to acknowledge the whole Adesto Technologies collective for a friendly atmosphere and nice working place.

I would also like to thank my parents and relatives for the opportunity to study abroad, for the endless support both during my studies at the university and throughout my life. I am also grateful to my friends for their moral support.

Last but not least, I would like to thank my girlfriend Daria Uslontceva for moral support in difficult times, for walking this path with me from beginning to end and for believing that I will succeed.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

signature

Abstract

The goal of this work is to propose an implementation of the KECCAK algorithm in FPGA and evaluate its effectiveness on the SoC platform. For this purpose we studied the theory behind SHA-3 hash function and KECCAK algorithm. We also chose the suitable interface to connect the implemented unit (accelerator) to the processor and wrote the firmware to access the accelerator from the processor. We evaluated the effectiveness of the implementation by comparing it with existing software implementation of SHA-3 hash function.

Keywords: hash function, hardware acceleration, digital design, FPGA, processor, communication interface, SoC

Abstrakt

Cílem této práce je návrh implementace Keccak algoritmu v FPGA a ověření jeho efektivity na SoC platformě. Pro tento účel jsme nastudovali teorii, která se týká SHA-3 hashovací funkce a Keccak algoritmu. Také jsme vybrali vhodný interface pro připojení implementované jednotky (akcelerátoru) k procesoru a napsali firmware pro přístup k akcelerátoru z procesoru. Efektivitu implementace jsme ověřili pomocí porovnání této implementace s existující softwarovou implementací SHA-3 hashovací funkce.

Klíčová slova: hash funkce, hardwarová akcelerace, digitální návrh, FPGA, procesor, komunikační interface, SoC

Contents

1 Introduction **1**

1.1 Thesis Structure . . . 2
1.2 Hash Function . . . 3
1.3 Common Hash Algorithms . . . 4
1.3.1 MD5 Message-digest Algorithm . . . 4
1.3.2 SHA-1 and SHA-2 Secure Hash Algorithms . . . 4
1.3.3 SHA-3 Secure Hash Algorithm . . . 5

2 SHA-3/Keccak Algorithm Description **6**

2.1 Sponge Construction and State Variables . . . 6
2.1.1 Absorbing Phase . . . 7
2.1.2 Squeezing Phase . . . 8
2.1.3 Long Messages Processing . . . 8
2.1.4 State Variables . . . 8
2.2 Algorithm Steps . . . 10
2.2.1 Specification of θ . . . 11
2.2.2 Specification of ρ . . . 12
2.2.3 Specification of π . . . 13
2.2.4 Specification of χ . . . 14
2.2.5 Specification of ι . . . 15

3 Algorithm Analysis **16**

3.1 Key Parameters of the Keccak for Hardware Implementation . . . 17
3.2 Estimation of Resources . . . 18
3.3 Conclusion . . . 20

4 HW Accelerator Description **21**

4.1 Block Diagram . . . 21
4.2 List of Ports . . . 22
4.3 HW Accelerator Hierarchy . . . 24
4.3.1 Main Control Unit . . . 25
4.3.2 Computational Core . . . 27
4.3.3 Internal Input and Output Memories . . . 28
4.3.4 Keccak Cell Unit . . . 29
4.3.5 XOR5 Auxiliary Block . . . 31

4.3.6 Output Multiplexer . . . 32
4.4 Clock Domains . . . 32

5 Design Flow **33**
5.1 Tools Used for Development . . . 35
5.2 Verification Environment . . . 36

6 Implementation Results **38**

7 SoC Design **40**
7.1 Block Design for Interface Selection . . . 40
7.1.1 Clock and Reset Generation . . . 41
7.1.2 MicroBlaze Processor . . . 42
7.1.3 AXI Interconnect . . . 43
7.1.4 AXI Central Direct Memory Access . . . 43
7.1.5 Local Memory Bus and AXI BRAM Controllers . . . 44
7.1.6 AXI4-Stream Data FIFO . . . 44
7.2 Address Space . . . 45
7.3 Interface Selection . . . 46
7.4 Interfaces Comparison Results . . . 47
7.5 Block Design for HW and SW Comparison . . . 48
7.5.1 SHA-3 Accelerator . . . 49
7.5.2 AXI to APB Bridge . . . 53
7.5.3 AXI Timer . . . 53
7.5.4 AXI UART . . . 54

8 Analysis of Firmware Function **55**
8.1 Data Transferring . . . 58
8.2 Data Processing . . . 60
8.3 Remaining Operations . . . 61
8.4 Influence of Each Aspect on the Total Execution Time . . . 62

9 Comparison of the SW and HW Approaches **65**
9.1 FPGA Development Board . . . 66
9.2 Data Set for Measurements . . . 67
9.3 Measurements Results . . . 68
9.4 Acceleration Dependence on the Message Length and SHA-3 Version . . . 73

10 Conclusion **76**

List of Figures

2.1 The sponge construction [5] 7
2.2 Parts of the KECCAK-f state [6] 9
2.3 $\boldsymbol{\theta}$ applied to a single bit [6] 12
2.4 $\boldsymbol{\rho}$ applied to the lanes [6] 13
2.5 $\boldsymbol{\pi}$ applied to a slice [6] 14
2.6 $\boldsymbol{\chi}$ applied to a single row [6] 15

4.1 Block diagram of the accelerator 22
4.2 `kctrl-fifo` states 26
4.3 `kctrl-fifo` operation 26
4.4 `kctrl-fifo` operation in the beginning of computation 27
4.5 `kctrl-fifo` operation in the end of computation 27
4.6 Block diagram of `keccak-cell` unit 30

5.1 Development steps 34

7.1 Block design for verification and measurements of the suitable interface 41
7.2 Block design for HW and SW comparison 49
7.3 Control register structure 50
7.4 Status register structure 51
7.5 Config register structure 51
7.6 `keccak-axis` block diagram 53

8.1 Message structure 55
8.2 Standard `putfsl` function assembly code 59
8.3 Optimized `putfsl` function assembly code 59
8.4 Ideal case of digest computation 62

9.1 USB/UART bridge [18] 67
9.2 Graph illustrating SHA-3 computation with accelerator 72
9.3 Graph illustrating SHA-3 computation without accelerator 73
9.4 The acceleration dependence on the message length and SHA-3 version 74

List of Tables

2.1 Round constants for the lane size 64 [9] . . . 11
2.2 $\boldsymbol{\rho}$ step offsets [6] . . . 13

3.1 Estimation of resources . . . 19

4.1 `keccak-fifo` signals description . . . 23
4.2 `kctrl-fifo` signals description . . . 25
4.3 `keccak-core` signals description . . . 28
4.4 `fifo-wrap` signals description . . . 29
4.5 `keccak-cell` parameters description . . . 30
4.6 `keccak-cell` signals description . . . 31
4.7 `xor5` signals description . . . 31
4.8 `outmux` signals description . . . 32

6.1 Number of blocks used in implementation . . . 38
6.2 FPGA resources utilization . . . 39

7.1 MicroBlaze address space . . . 45
7.2 CDMA address space . . . 46
7.3 Interfaces comparison . . . 48
7.4 Using DMA with different message lengths . . . 48
7.5 Control register bit definitions . . . 50
7.6 Status register bit definitions . . . 51
7.7 Config register bit definitions . . . 51

8.1 Data processing . . . 60
8.2 Ideal vs real firmware execution . . . 62
8.3 The influence of each category on the total execution time in clock cycles . . . 63
8.4 The percentage influence of each category on the total execution time . . . 64

9.1 SHA3-224 results . . . 68
9.2 SHA3-256 results . . . 69
9.3 SHA3-384 results . . . 70
9.4 SHA3-512 results . . . 71

Acronyms

AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CC	Clock Cycles
CDMA	Central Direct Memory Access
CRG	Clock and Reset Generation
DFF	D Flip-Flop
DLMB	Data Local Memory Bus
DMA	Direct Memory Access
DUT	Device Under Test
FIFO	First In First Out Memory
FIPS	Federal Information Processing Standards
FPGA	Field-programmable Gate Array
GF	Galois Field
HDL	Hardware Description Language
HW	Hardware
I/O	Input/Output
IDE	Integrated Development Environment
ILMB	Instruction Local Memory Bus
IP	Intellectual Property
LFSR	Linear-feedback Shift Register

LMB	Local Memory Bus
LUT	Lookup Table
MD	Message-Digest Algorithm
MUX	Multiplexer
NIST	National Institute of Standards and Technology
PC	Personal Computer
RC	Round Constant
RDEN	Read Enable
RTL	Register-transfer Level
SDK	Software Development Kit
SHA	Secure Hash Algorithm
SoC	System on a Chip
SW	Software
UART	Universal Asynchronous Receiver-transmitter
USB	Universal Serial Bus
WREN	Write Enable

Chapter 1

Introduction

The main purpose of general-purpose processors is to execute any given task. Due to the fact that these processors are universal, their architecture is also universal and is not oriented on the execution of specific tasks. If there is a need to execute the specific task more effectively, then we can use the processor with different architecture or we can use a hardware acceleration. The hardware acceleration is the execution of a certain task on the hardware that is specifically designed to execute this task effectively. One example of the hardware acceleration that closely relates to this work is cryptographic accelerator, which is the unit that can execute all cryptographic operations more efficiently than the general-purpose processor.

The hardware acceleration takes place when the task requires the processing of large amount of data and complex mathematical operations. The excellent candidates for the hardware acceleration are the hash functions that are described in Section 1.2 in detail. Briefly, hash function takes an input message of an arbitrary length, applies complex mathematical algorithm and provides the output with fixed length. The most demanding part of every hash function is the data processing part that is realized with some complex algorithm. In this work we will concentrate ourselves on the SHA-3 hash function and the KECCAK algorithm that it uses [12], [6]. As we will see, the KECCAK algorithm is very time-consuming, because it requires the processing of large amount of data.

The KECCAK algorithm has a lot of software implementations in many different programming languages that can be found at [32]. One of the software implementations in C language will be chosen in this work for comparison with our hardware implementation. Beside the software implementations, there is one hardware implementation of the KECCAK algorithm in VHDL language [29]. In this work we concentrate ourselves on the KECCAK algorithm implementation in Verilog-2001 hardware description language.

Beside the effectively implemented data processing part of the accelerator, it is very important to optimally integrate the accelerator in the system, otherwise the acceleration will not make any sense. Our main goal is to create an effective and fast hardware implementation of the algorithm that can later be used to accelerate the processor's execution of the SHA-3 hash function, which uses the KECCAK algorithm. As a result of this work we want to have an IP core that can be used in any FPGA design with the processor.

1.1 Thesis Structure

In this work we first provide a brief introduction to the world of hash functions in Chapter 1, their purpose and different types of hash functions. The universal knowledge about hash functions help us to better understand the theory behind the SHA-3 function. In Chapter 2 we concentrate on the description of the SHA-3 hash function and the KECCAK algorithm that it uses. The theory described in that chapter helps us to understand how the KECCAK algorithm works and can be implemented effectively.

Chapter 3 contains an analysis of the KECCAK algorithm from the point of view of its implementation in FPGA. The analysis provided in this chapter is crucial to finding the optimal way to design the accelerator. The analysis of the algorithm is closely connected to the theory knowledge obtained in Chapter 2. It helps us to propose the parallel implementation of the algorithm where all bits of the internal state of KECCAK algorithm are processed in parallel. Chapter 3 focuses more on ideas about how the accelerator should be implemented and in Chapter 4 we realize these ideas and show the RTL design of the accelerator. We describe the hierarchy of the designed accelerator and the structure of the distinct modules. Also we show how the parallel approach has been implemented and how the accelerator operates. As a result of this chapter we have the computational core of the KECCAK algorithm, which yet does not contain any interface to communicate with the processor.

In Chapter 5 we talk about different development tools that we used throughout this work and what steps we followed during implementation. We also provide a detailed description about how the designed accelerator was verified during all development steps. Chapter 6 provides the results of logic implementation of the designed accelerator for a specific FPGA, where we analyze the utilization of the chip resources. We also describe the frequency at which it can operate and how the assumption about the number of logic cells from Table 3.1 differs from the implementation results.

In Chapter 7 we describe two SoC designs. The main task of the first SoC design was to help us to indicate which communication interface better fits our application. We describe different types of interfaces, their advantages and disadvantages, and the simulation results. After the suitable interface has been chosen, we move to the second SoC design that helps us to evaluate the effectiveness of the designed accelerator in FPGA. In Chapter 8 we provide a firmware function that allows the processor to use the accelerator. We also analyze implemented function and describe its weak and strong sides. We divide the operation of the function into several categories and talk about each category in detail along with suggestions about how these categories can be improved and optimized. Firmware function from Chapter 8 also helps us to compare the time needed to execute the SHA-3 function with and without the accelerator. Chapter 9 contains the result of this comparison and provides the data that evaluate the effectiveness of the accelerator. The chapter contains the tables with measurements results and according graphs that visualize the comparison.

In Chapter 10 we summarize the results of this work and describe several difficulties that we faced throughout this work. We also provide a suggestion on how this work can be improved in future.

1.2 Hash Function

A hash function, or a hash algorithm, takes an input message of arbitrary length, mixes it in a certain way and gives a digest of the required length to the output. The input of the hash algorithm is called a "message", and the output is called a "digest" or simply a "hash value". Hash functions are ubiquitous, starting with generating pseudo-random sequences, encrypting passwords, and ending with digitally signed documents. All this poses a serious challenge for developers and scientists when creating hash algorithms, since the algorithms must exhibit a high level of reliability, good computing speed, flexibility in setting parameters, and relative simplicity in implementation.

In this work we concentrate ourselves on the hardware implementation of the SHA-3 family hash functions. Each of the SHA-3 functions is based on an instance of the KECCAK algorithm that National Institute of Standards and Technology (NIST) selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition [12]. We chose this hash function because it is well documented, has a lot of applications and can be effectively implemented in hardware. Software implementations of the KECCAK algorithm in different programming languages can be found in [32].

One of the main parameters of a hash function is its security. NIST defines three main criteria by which it is possible to judge the strength of hash algorithm [8]:

1. *Collision resistance*: how computationally difficult it is to find two input values at the input of a hash function, for which the same output result will be produced. It is measured in the amount of work needed to find such a collision. If the function had poor collision resistance, then the algorithm would be vulnerable to collision attacks, the essence of which is that an attacker can replace the original document with a fake one, but which has the same hash value as the original. The recipient of the signed message is then not able to distinguish the original from the fake, until he opens a document. Thus, a malicious program can be passed off as something else [28].

2. *Preimage resistance*: how computationally difficult it is to find the input value of the hash function for a given value at the output. It is measured in the amount of work required to search for an input message. Suppose that an attacker has obtained a table that stores users' password hashes. If a hash function with a small preimage resistance has been used, then it will not be difficult for an attacker to find the inverse images of the hash values and thus obtain users' passwords. Often attackers use prepared tables with hashes of simple and frequently used types of passwords [27].

3. *Second preimage resistance*: for a given input value $\boldsymbol{x_1}$, how computationally difficult it is to find the value $\boldsymbol{x_2}$, the hash of which will coincide with the hash of $\boldsymbol{x_1}$. This is similar to collision resistance, but differs in that we know the value of $\boldsymbol{x_1}$ in advance. This allows the attacker to create fraudulent certificates at any time, not just at the time of certificate issuance [11].

There are a huge number of different hash functions and each function does not have to have each of the parameters listed above on the high. If the function does not show excellent

results in one of the parameters, this does not mean that it is unsuitable. Its certain qualities are important in certain tasks and the hash function can simply be used for another purpose. Since hash functions are used very often and are widely used, attacks against functions are also developing. Therefore, hashing algorithms are constantly being improved and existing ones are being replaced by new versions or are no longer used in some applications, if making a successful attack becomes an easy task.

1.3 Common Hash Algorithms

Now we will briefly look at several algorithms that were used in the past or are being used now. Each of these algorithms has a different internal structure, which is based on certain mathematical operations. Basically, these mathematical operations are cyclically repeated. Some of them, for example, determine the parity of a group of several bits, others differently shift (rotate) a section of a message, and so on. As we will see in Chapter 2, in the case of SHA-3 and, in particular, the used KECCAK algorithm, some operations remove the symmetry of individual iterations, while others are oriented on building a strong dependence of one bit on many other bits.

1.3.1 MD5 Message-digest Algorithm

MD5 [1] is a message-digest algorithm. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It was created in 1991 by Ronald Rivest and was documented in April 1992. Algorithm was primarily designed as an extension of the MD4 message-digest algorithm. Nowadays MD5 is considered to be weak against attacks, such as collision attacks.

1.3.2 SHA-1 and SHA-2 Secure Hash Algorithms

SHA-1 and SHA-2 are two hash algorithms published by NIST. They are specified in the FIPS 180-4 standard [10] and were first released in the 1993 and 2001 respectively. The difference between them is in the internal structure and the length of the digest they generate. Both algorithms use padding of the input message, so that it can be divided into blocks with constant length, which are then processed sequentially.

SHA-1 may be used to hash a message, $\boldsymbol{M}$, having a length of $\boldsymbol{l}$ bits, where $0 \leq \boldsymbol{l} < 2^{64}$. The algorithm uses:

- Four different functions that are applied to the internal state of the algorithm
- Four internal constants that are XORed with the internal state every round
- Five internal variables used to store the intermediate results
- Five 32-bit hash values used at the output and updated every processed 512-bit block

The operation of the algorithm is divided into 4 steps with 20 iterations. The four functions and constants correspond to these steps and their purpose is to differently mix the bits of the message. Result after applying the function is stored into 5 internal variables, and iteration repeats. SHA-1 is able to produce only a 160-bit digest.

SHA-2 has a similar to SHA-1 structure. It is able to process a message having a length of $\boldsymbol{l}$ bits, where $\boldsymbol{l}$ is $0 \leq \boldsymbol{l} < 2^{128}$. It also pads the message and divides it into the blocks of the length 512 bit. However, it is able to produce different output lengths, such as 224, 256, 384 and 512 bits. This is achieved by having different size hash values that store the output of the algorithm. For example, SHA-224 uses seven 32-bit values, so that in sum it gives exactly 224. SHA-512 uses eight 64-bit values. Similar to SHA-1, SHA-2 uses:

- Six different functions that are applied to the internal state of the algorithm
- Eight 32-bit internal variables used to store intermediate result
- The total of 64(80) internal constants each used in one function every round

The processing of one 512-bit block is divided into 64 or 80 rounds. SHA-224 and SHA-256 use 64 rounds; SHA-384 and SHA-512 both use 80 rounds. These rounds are not identical, since the different constant is used every time.

1.3.3 SHA-3 Secure Hash Algorithm

SHA-3 hash family functions were first published by NIST in 2015. They are specified in FIPS 202 standard [12] and use the KECCAK algorithm specified in [6]. SHA-3 is very different in structure from previous standards and has $\boldsymbol{n_r}$ rounds. The number $\boldsymbol{n_r}$ depends on the size $\boldsymbol{b}$ of the internal state of algorithm. SHA-3 lets us to choose such that $\boldsymbol{b} = 25 \times \boldsymbol{2^l}$, where $\boldsymbol{l}$ ranges from 0 to 6. Each round, further, consists of five steps that are being executed sequentially. Rounds are not identical due to the presence of the constants that differ for every round and are XORed with the part of the internal state. Internal state $\boldsymbol{s}$ that is $\boldsymbol{b}$ bits long is the only internal variable used to store the intermediate results of the algorithm. We provide the detailed description of the algorithm in Chapter 2.

Chapter 2

SHA-3/KECCAK Algorithm Description

SHA-3 hash function uses the KECCAK algorithm [6], developed by Guido Bertoni, Joan Daemen, Michäel Peeters and Gilles Van Assche. The basis of this algorithm is the sponge construction, which we will analyze in detail in Section 2.1. This construction allows to process input data of any length and create the digest of the required length. On one side, the function receives an input message, which is the only input to the hash function. Everything that happens with the message and the way it is processed is determined by the definition of a specific function. Inside the sponge, the input data are divided into blocks of the same length and the so-called permutation function is applied on them.

The permutation function, in the case of SHA-3, is specified by the KECCAK algorithm and called KECCAK-f permutation. Inside this function, the data are repeatedly mixed. We will analyze in detail each operation of the KECCAK algorithm in Section 2.2. When all input data has been processed, the required number of processed data bits is taken, and provided as an output of the hash function.

2.1 Sponge Construction and State Variables

SHA-3 family hash functions are based on the sponge construction that is described in this section. We will refer to [12] and [5]. The sponge construction has the structure illustrated in Figure 2.1.

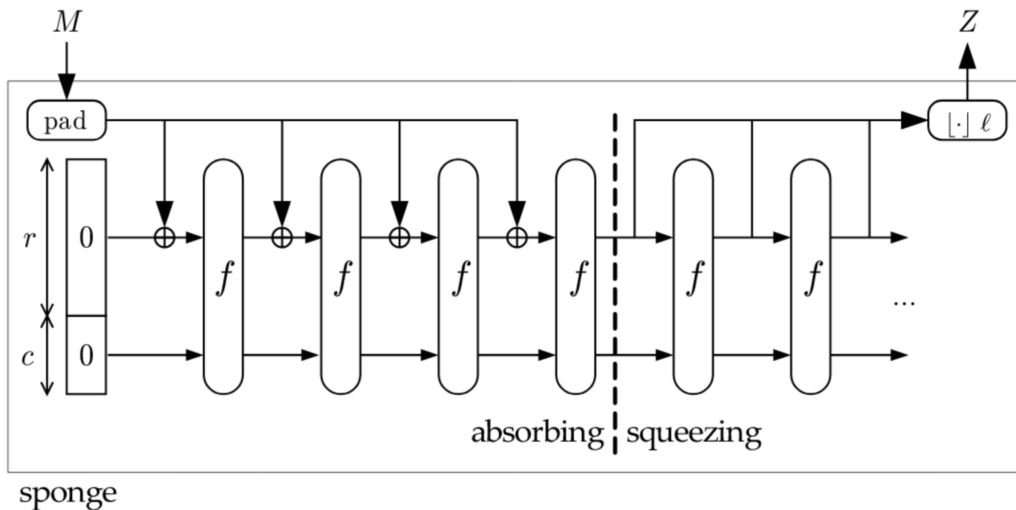


Figure 2.1: The sponge construction [5]

It has one input, which is the input message $\boldsymbol{M}$ and one output that is the output message $\boldsymbol{Z}$. The input sequence of bits can have an arbitrary length, but it has to be padded to the message $\boldsymbol{P}$ which length is divisible by bit rate $\boldsymbol{r}$. Before padding the two-bit suffix is appended to the message $\boldsymbol{M}$ to produce the input $\boldsymbol{N}$ to the pad function. In the case of SHA-3 hash functions the suffix is **01**.

Next, the multi-rate padding rule is applied to the new message $\boldsymbol{N}$. By [5] the multi-rate padding is defined according to Definition 2.1.1.

Definition 2.1.1. *Multi-rate padding, denoted by pad10*1, appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length.*

The block rate in the above definition is our bit rate $\boldsymbol{r}$. The padding is provided even if the length of the input message $\boldsymbol{M}$ is already divisible by $\boldsymbol{r}$. Let us call such message $\boldsymbol{M_1}$ and imagine that there is another message $\boldsymbol{M_2}$, which after padding will result in message $\boldsymbol{M_1}$ and these two **different** messages will produce same output. To avoid this error, the message $\boldsymbol{M_1}$ should be padded.

Sponge construction has two phases: **absorbing** and **squeezing**. Both phases are described in Subsections 2.1.1 and 2.1.2.

2.1.1 Absorbing Phase

First of all, the internal state $\boldsymbol{s}$ consisting of outer and inner part with lengths of $\boldsymbol{r}$ and $\boldsymbol{c}$ respectively, is initialized to zero. The message $\boldsymbol{P}$ is divided into blocks $\boldsymbol{P_i}$, each of the length $\boldsymbol{r}$. The $\boldsymbol{r}$-bit input message blocks are XORed into the outer part of the state, interleaved with applications of the function $\boldsymbol{f}$. When all message blocks are processed, the sponge construction switches to the squeezing phase. Function $\boldsymbol{f}$ is realized using KECCAK-f

permutations that we will describe in Section 2.2. Absorbing phase can be described using Algorithm 1.

Algorithm 1: Absorbing phase [6]

$s = 0^b$
for $i = 0$ **to** $|P|_r - 1$ **do**
  $s = s \oplus (P_i || 0^{b-r})$
  $s = f(s)$
end

2.1.2 Squeezing Phase

The second part of sponge construction is the squeezing phase. On the input side it receives the state $\boldsymbol{s}$ obtained in absorbing phase and on the output side it produces the bit sequence of desired length. First of all, the state $\boldsymbol{s}$ is truncated to its first $\boldsymbol{r}$ bits and assigned to $\boldsymbol{Z_0}$. If the required output length $\boldsymbol{l}$ is greater than $|\boldsymbol{Z_0}|$, then function $\boldsymbol{f}$ is applied to the state and new $\boldsymbol{Z_1}$ is taken and concatenated with $\boldsymbol{Z_0}$. These steps are repeated until $|\boldsymbol{Z}| \geq \boldsymbol{l}$. Then $\boldsymbol{Z}$ is truncated to its first $\boldsymbol{l}$ bits and is given as an output of the sponge construction. All of the above can be described using Algorithm 2.

Algorithm 2: Squeezing phase [6]

$Z = \lfloor s \rfloor_r$
while $|Z| < l$ **do**
  $s = f(s)$
  $Z = Z || \lfloor s \rfloor_r$
end
return $\lfloor Z \rfloor_l$

2.1.3 Long Messages Processing

Due to the design of the sponge absorbing stage, it can process messages of arbitrary length. The length of the message only affects the processing time, and does not affect the implementation of the hash function. All resources and blocks required to perform permutations can be reused, which is the benefit of using a sponge function.

2.1.4 State Variables

The state $\boldsymbol{s}$ is expressed and treated as a three-dimensional array with dimensions $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$. All the operations over $\boldsymbol{x}$ and $\boldsymbol{y}$ are taken modulo 5 and all the operations over $\boldsymbol{z}$ are taken modulo $\boldsymbol{w} = \boldsymbol{b}/25$ (in our case it is 64). We denote the three-dimensional state as $\boldsymbol{a}$ and one-dimensional state as $\boldsymbol{s}$. The mapping between $\boldsymbol{s}$ and $\boldsymbol{a}$ coordinate systems is $\boldsymbol{s}[\boldsymbol{w}(5\boldsymbol{y} + \boldsymbol{x}) + \boldsymbol{z}] = \boldsymbol{a}[\boldsymbol{x}][\boldsymbol{y}][\boldsymbol{z}]$ [6].

Since the state is a three-dimensional array, we can represent it as a cuboid, with sides $\boldsymbol{x}$ $\boldsymbol{y}$ and $\boldsymbol{z}$. Next, we divide it into the components shown in Figure 2.2.

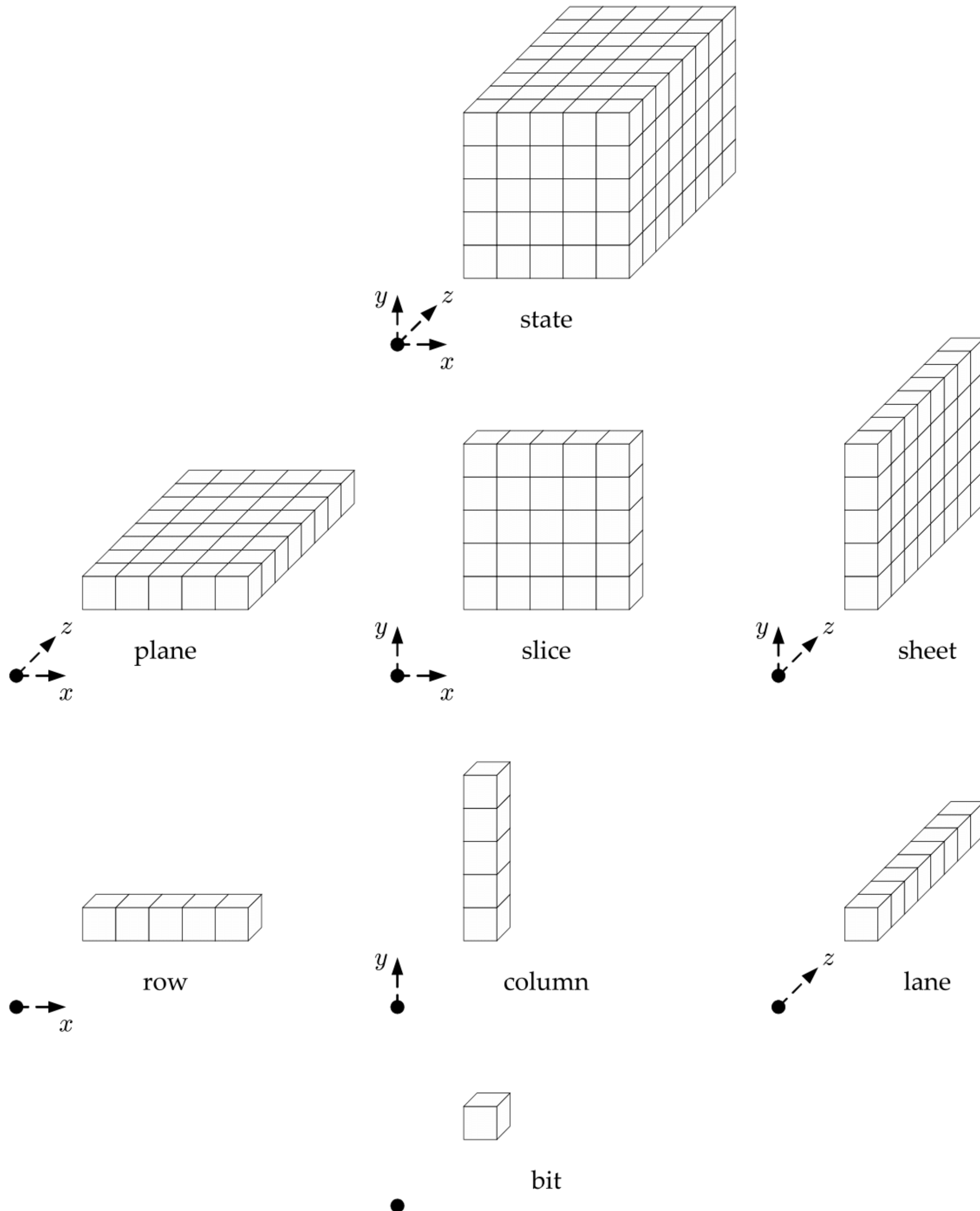


Figure 2.2: Parts of the KECCAK-f state [6]

The parts of the state have the following properties in the case of SHA-3 [6]:

- **plane** is a set of 5×64 bits with constant $\boldsymbol{y}$ coordinate
- **slice** is a set of 5×5 bits with constant $\boldsymbol{z}$ coordinate
- **sheet** is a set of 5×64 bits with constant $\boldsymbol{x}$ coordinate
- **row** is a set of 5 bits with constant $\boldsymbol{y}$ and $\boldsymbol{z}$ coordinates
- **column** is a set of 5 bits with constant $\boldsymbol{x}$ and $\boldsymbol{z}$ coordinates
- **lane** is a set of 64 bits with constant $\boldsymbol{x}$ and $\boldsymbol{y}$ coordinates

2.2 Algorithm Steps

SHA-3 hash function implements its permutation function using one of the KECCAK-f permutations. There are 7 permutations in total denoted as KECCAK-$f[b]$, where $\boldsymbol{b} = 25 \times 2^{\boldsymbol{l}}$ and $\boldsymbol{l}$ ranges from 0 to 6. The four SHA-3 hash functions (SHA3-224, SHA3-256, SHA3-384 and SHA3-512) uses the KECCAK-f[1600] permutation, which means that the length of the state is 1600 bits.

KECCAK-f[1600] is an iterated permutation, consisting of a sequence of $\boldsymbol{n_r} = 12+2l$ ($\boldsymbol{l} = 6$) rounds indexed with $\boldsymbol{i_r}$ from 0 to 23. Every round consists of five steps:

$$\mathrm{R} = \iota \circ \chi \circ \pi \circ \rho \circ \theta, \text{ with}$$

$$\theta: \; a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1],$$

$$\rho: \; a[x][y][z] \leftarrow a[x][y][z-(t+1)(t+2)/2],$$

$$\text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in GF}(5)2 \times 2$$

$$\text{or } t = -1 \text{ if } x = y = 0,$$

$$\pi: \quad a[x][y] \leftarrow a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},$$

$$\chi: \quad a[x] \leftarrow a[x] + (a[x+1]+1)a[x+2],$$

$$\iota: \quad a \leftarrow a + \mathrm{RC}[i_r].$$

The last step uses the 24 round constants $\mathbf{RC[\boldsymbol{i_r}]}$. These are $\boldsymbol{w}$-bit long and only applied to the first $\boldsymbol{w}$ bits of the state $\boldsymbol{s}$. The value of the RC constant is defined as the output of a binary linear feedback shift register (LFSR), the values for KECCAK-f[1600] are defined as follows:

Table 2.1: Round constants for the lane size 64 [9]

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

The algorithm for each step mapping takes a thee-dimensional state array, denoted by $\boldsymbol{a}$, as an input and returns an updated state array, denoted by $\boldsymbol{a'}$, as the output. The $\boldsymbol{\iota}$ step has a second input that is a round number. All step mappings require only bitwise Boolean operations and rotation. Now we will look at each of the five steps.

2.2.1 Specification of θ

The first step mapping is $\boldsymbol{\theta}$. This mapping is linear and translation-invariant in all directions. $\boldsymbol{\theta}$ applied to a single bit is illustrated in Figure 2.3. The primary goal of this mapping is to include the diffusion onto the hash function. Diffusion means that a single bit affects the value of many other bits and as a consequence a value of a single bit is affected by many other digits [2]. Again, its mathematical representation is the following:

$$\theta: \ a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y=0}^{4} a[x+1][y'][z-1]$$

The effect of $\boldsymbol{\theta}$ is to XOR each bit in the state with the parities of two columns in the array.

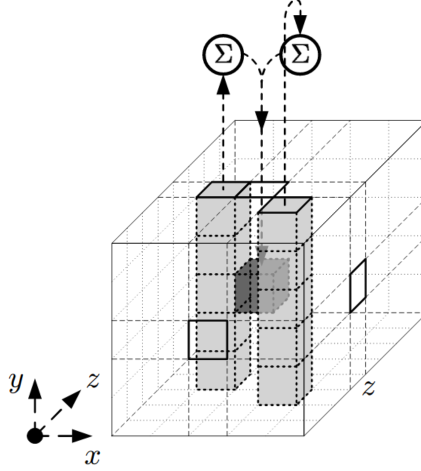


Figure 2.3: $\boldsymbol{\theta}$ applied to a single bit [6]

Symbol $\boldsymbol{\Sigma}$ indicates the XOR sum of all the bits in the column. As we can see, this step can be implemented using only XOR gates.

2.2.2 Specification of ρ

This step is intended to rotate each lane by some constant value that is calculated according to the definition of this step, which has the following mathematical representation:

$$\rho: \; a[x][y][z] \leftarrow a[x][y][z-(t+1)(t+2)/2]$$

$$\text{with } t \text{ satisfying } 0 \leq t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in GF}(5)2 \times 2$$

$$\text{or } t = -1 \text{ if } x = y = 0$$

The first equation shows us how the new bit position is calculated. The triangular numbers are used as the offsets. It can be seen with the equation $(\boldsymbol{t}+1)(\boldsymbol{t}+2)/2$ that represents triangular numbers. The second equation demonstrates how the coordinates of rotational constants are computed. The same matrix and principle is used here as in $\boldsymbol{\pi}$ step and the graphical representation can be found in Figure 2.5. The effect of $\boldsymbol{\rho}$ step mapping is illustrated in Figure 2.4.

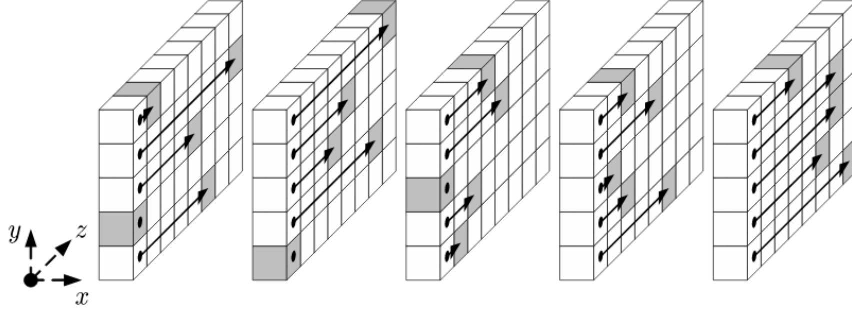


Figure 2.4: $\boldsymbol{\rho}$ applied to the lanes [6]

Rotation constants and the appropriate coordinates are given in Table 2.2.

Table 2.2: $\boldsymbol{\rho}$ step offsets [6]

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Since all operations over the $\boldsymbol{z}$ coordinate are taken modulo $\boldsymbol{w}$, these offsets can also be substituted with its value modulo 64. Without $\boldsymbol{\rho}$ step, the diffusion between slices would be very slow.

2.2.3 Specification of π

The third step mapping changes the $\boldsymbol{x}$ and $\boldsymbol{y}$ coordinates of every lane except for the first lane. New coordinates are calculated as follows:

$$\pi : \quad a[x][y] \ \leftarrow a[x'][y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

$\boldsymbol{\pi}$ step applied to a slice is shown in Figure 2.5.

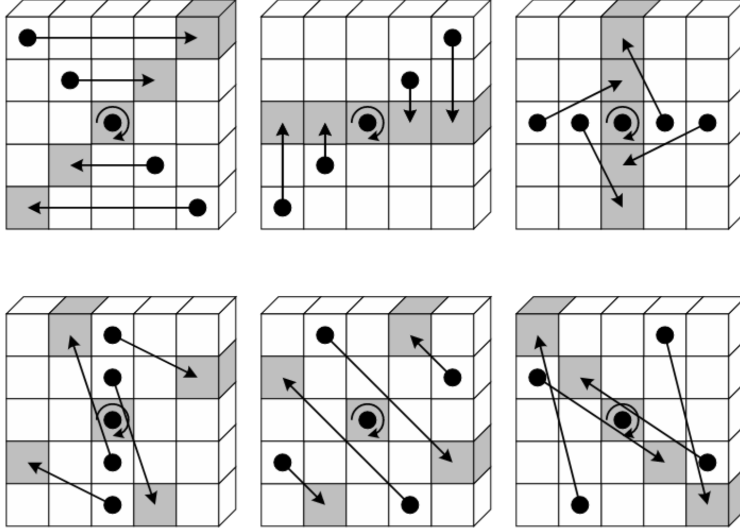


Figure 2.5: $\boldsymbol{\pi}$ applied to a slice [6]

We can see that the bit with coordinates $(0, 0, \boldsymbol{z})$ does not change its position. This step is translation-invariant in z-direction and can be implemented by addressing. Inspired by the software implementation [30] we combined the $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ steps into one for the optimization purpose.

2.2.4 Specification of χ

The fourth step called $\boldsymbol{\chi}$ is the only non-linear mapping. Without it, the Keccak-f round function would be linear, which means that the output of the round is the linear combination of the inputs and hence the input can be found by solving the system of linear equations. The mathematical representation of $\boldsymbol{\chi}$ step is the following:

$$\chi : \quad a[x] \quad \leftarrow a[x] + (a[x+1] + 1)a[x+2]$$

The $\boldsymbol{\chi}$ step is shown in Figure 2.6.

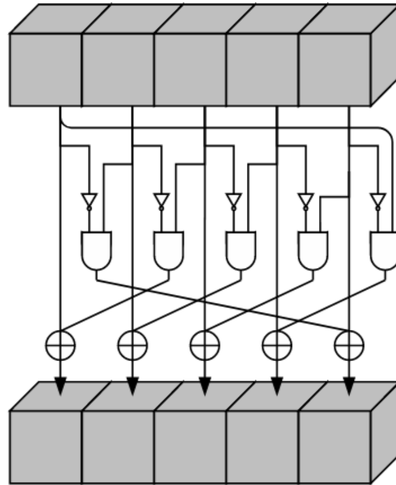


Figure 2.6: $\boldsymbol{\chi}$ applied to a single row [6]

Thanks to $\boldsymbol{\theta}$ and $\boldsymbol{\chi}$ steps, every bit at the input of the round affects 31 bits at its output, and each bit at the output depends on the 31 bits at the input.

2.2.5 Specification of ι

The last step in every round is the addition of a round constant which value is dependent on the current round. This mapping is aimed at disrupting symmetry between individual rounds. This function makes every round unique. The attacks, which are using symmetry (such as slide attacks) in hash functions, will not be effective against the hash function with asymmetrical rounds. This type of attacks is used against the ciphers and hash functions that use the identical permutation function every round [33]. All round constants can be found in Table 2.1. Every round constant is added only to the first lane.

Chapter 3

Algorithm Analysis

FPGA and custom digital circuits in general give us the ability to effectively process a large amount of data by splitting this data into parts and processing them in parallel. When examining the algorithm, it is very important to optimally break it into similar pieces, which can subsequently be processed simultaneously. This chapter is devoted to the analysis of the algorithm from the point of view of its implementation at the hardware level.

The digital designers, of course, have a completely different approach to solving the problem than the software engineers. Many things, such as logic operations, rotations and addressing, to name a few, are implemented quite easily and very efficiently at the hardware level. Efficiency is manifested both in the time required to perform these operations, and in the number of elements involved in its implementation. For example, the XOR operation in the FPGA technology is implemented by combinational logic using one logic cell (to be more precise, the lookup table), while in the processor (it may differ depending on the architecture), it is necessary to access the memory in which the variables are stored, pass them through the Arithmetic Logic Unit and save back to memory. The processing time is extended further due to memory latency and the pipelined nature of the processors.

Sequential execution of instructions allows the processor to process 32 or 64 bits at a time (depending on the architecture used), which for "load, compute, store" triple will require $(1600/32) \times 3 = 150$ iterations to process the whole state $\boldsymbol{s}$. It is possible that an effective solution can also be found for the software approach which allows the processing of the state quite efficiently. Refer to Chapter 2 in [9] that offers some implementation techniques, such as bit interleaving or plane-per-plane processing. However, using custom digital logic, we can create 1600 XOR cells that works in parallel, which will allow us to carry out the calculation as quickly as possible bounded just by the speed of the signal passing through a logic cell. Of course, it is logical to have a synchronous design implemented using sequential logic, so doing 1600 parallel XOR operations would take us one clock cycle.

It is also important to see the disadvantages that may be associated with the possible acceleration of some algorithms with the help of the hardware accelerator. Despite the fact that parallel computing can occur disproportionately faster than computing by the processor, it is necessary to evaluate whether it makes sense to use hardware acceleration. To clarify this

idea, we give an example of the accelerator that requires frequent communication with the processor for its operation. In this design, a lot of time will be spent on communication between the accelerator and the processor. Communication runs on some internal bus, for example, on the AMBA protocol, and this bus is used by other parts of the chip, too, which reduces its throughput from the point of view of the accelerator. This system will spend most of the time forwarding messages, and the advantage that the accelerator gives us may not justify itself.

3.1 Key Parameters of the Keccak for Hardware Implementation

In this section, we describe some aspects of the Keccak algorithm that make it an ideal candidate for implementation at the hardware level. The structure of this section will be as follows: we call the property of the algorithm, describe why it is an advantage, and how it can be used in its implementation.

The entire computational process can be implemented using only basic logic gates and one register (DFF) per bit. In Subsection 2.2.1, which describes the $\boldsymbol{\theta}$ step, we can see the use of only addition operations modulo two, which is the XOR function. The $\boldsymbol{\chi}$ operation described in Subsection 2.2.4 uses multiplication, addition, and inversion, which implies blocks such as AND, XOR, and NOT. The $\boldsymbol{\iota}$ operation (Subsection 2.2.5) simply XORs the first lane with the round constant, that is, we use XOR cells. Steps $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ do not use any logic cells. The mentioned DFF is necessary for storing the bit value for one clock cycle. Using only the basic blocks allows us to implement the algorithm in FPGA.

There is no need to use additional memory to store the state value. Here it is meant that the current state value is used only to calculate a new value in the next clock cycle and is not used later. This statement is justified by the structure of the algorithm itself and the absence of any feed-forward loop of the state with further calculations. This property simplifies the implementation of the algorithm and allows the use of only simple sequential logic consisting of DFFs that store the values of individual bits and the combinational logic necessary to calculate new bit values.

It is possible to operate with all 1600 bits of the state $\boldsymbol{s}$ concurrently. Thanks to the DFFs, the value of each bit remains unchanged for one clock cycle and a new value can be calculated during this cycle. The new value of the bits depends only on the previous state value. This allows a parallel approach to implementation, meaning that bit values are calculated and overwritten at the same time. Parallelism can be very effectively realized in custom digital circuits.

Every step mapping of the Keccak-f permutation function can be executed during one clock cycle. As noted earlier, the only element of the sequential logic is the DFF, which stores bit value, and everything else is implemented using combinational logic, allowing the calculation of the new bit values during one clock cycle. Moreover, inspired by software implementation[30], we combine $\boldsymbol{\pi}$ and $\boldsymbol{\rho}$ operations into one, which reduces the execution

time of each round by one clock cycle. One-clock cycle execution allows us to make an initial estimate of the complexity of our solution over time. To process each block $\boldsymbol{P_i}$ of message $\boldsymbol{P}$, 24×4, that is, 96 clock cycles, are needed.

All rounds are identical (except for the $\boldsymbol{\iota}$ step), implying reuse of the blocks and same wiring/addressing. Steps $\boldsymbol{\theta}$, $\boldsymbol{\chi}$, $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ are independent of the current round and are performed identically throughout the entire computational process. It means that the blocks used to calculate the value of the bits can be reused in each round. Despite the fact that the $\boldsymbol{\iota}$ step is dependent on the round, its principle remains the same, namely, XORing the first lane with the round constant. The XOR operation can be reused, but we need a multiplexer, which will switch one of the XOR inputs to various constants, depending on the current round. Since the constants are given by the algorithm itself and are independent of the input message, they can be calculated at the design stage (see Table 2.1) and, when implemented, they will be hard-wired to the relevant values. This also simplifies the implementation of the algorithm by the fact that there is no need to add LFSR to calculate the constants.

There is no need to use any logic to implement $\boldsymbol{\pi}$ and $\boldsymbol{\rho}$ steps, since each bit will always be assigned the value of the exact other bit. For example, bit 576, after completing both steps, will be assigned the value of the 1411 bit. This can be calculated by simply substituting the coordinates of bit 576, that is, $(4, 1, 0)$, in the formulas for $\boldsymbol{\pi}$ and $\boldsymbol{\rho}$, which give us the coordinates of the bit, the value of which is the new value of bit 576. In each round, these coordinates will be the same for bit 576, as for all other bits, which allows us to get rid of the use of any shift registers and logic for rearranging lanes and can be used a simple hard-wired assignment of new addresses.

Following the advantages of the algorithm described above, the best solution in terms of speed is to choose a parallel implementation in which we will develop a universal block, later called the `keccak-cell`, which will allow us to perform all algorithm operations for one bit. Using the versatility of the cell, the KECCAK core will consists of 1600 instances of the `keccak-cell`, which will be interconnected in a certain way.

3.2 Estimation of Resources

Before writing the RTL code, it is needed to estimate required number of logic gates. The theoretical introduction given in Chapter 2, along with the analysis from the previous section, gives us an idea of how individual steps can be implemented. Since we have already decided that the algorithm can be implemented using only the basic logic cells, and these cells can be used repeatedly during the calculations, it is enough to calculate how many elements are needed to implement one round. Each subsequent round will reuse the same blocks and only different round constants will be used (which, as we decided before, does not require any cells and is simply connected to zeroes and ones).

We will implement each step based on its mathematical notation and the pseudocode given

in [6], so that even before writing the code it is possible to evaluate its complexity. In order to further simplify our task, we will calculate how much logic is needed for one bit. Due to the parallel implementation that we have chosen, this number will be just multiplied by 1600 then. A preliminary estimate of the number of elements needed to implement individual steps may differ from the actual implementation, but even this gives us a good idea of whether it makes sense to try to implement the algorithm in this way.

According to the technical documentation of the Keccak algorithm [6], the following logic gates are needed to implement the single steps:

1. $\boldsymbol{\theta}$ step needs two XOR5 gates and one XOR3 gate
2. $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ steps do not require any combinational logic and can be implemented by wiring
3. $\boldsymbol{\chi}$ step uses one NOT, one AND and one XOR
4. $\boldsymbol{\iota}$ step uses constants which can be pre-calculated on implementation stage

All steps and rounds are repeated cyclically and a multiplexer will be used to switch between the steps of the permutation function. Multiplexer will have a minimum of four inputs plus one control input. Four inputs correspond to the new values of this bit after performing a certain step of the function ($\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ are combined into one step).

For the $\boldsymbol{\iota}$ step, we also need a multiplexer that switches between 24 round constants and changes its input every round. Due to the fact that the $\boldsymbol{\iota}$ step affects only the first lane, there is no need to have a multiplexer for each of the 1600 state bits. Since the lane length is 64 bits, we need 64 such multiplexers.

The presence of multiplexers also presupposes the presence of a control unit, which will gradually increment the control signal, responsible for changing the step, and also, every 4 clock cycles, change the number of rounds. We assume that the control unit does not take up much space in the design, compared with the computing core. The control unit can be implemented using several counters, multiplexers and a small number of standard cells.

After analyzing the individual parts, we can estimate the required number of elements. The implementation of the computational part of the algorithm will take approximately:

Table 3.1: Estimation of resources

Cell	**For 1 bit**	**Total**
AND	1	1600
XOR5	2	320
XOR3	1	1600
NOT	1	1600
DFF	1	1600
MUX6 (step select)	1	1600
MUX24 (round select)	1	64

In total we have only 5120 basic logic cells to implement every step, 1600 registers to store the bits values and 1664 multiplexers to implement the step and round selection. Modern FPGA chips can implement it without any problem.

3.3 Conclusion

In the previous two sections, we examined the properties of an algorithm implemented at the hardware level and explained why we chose a parallel implementation. In addition, we can say that the design of 1600 blocks can be automated, and how the state bits are interconnected can be calculated using the formulas given in Section 2.2.

Chapter 4

HW Accelerator Description

In this chapter, we present an implementation of the KECCAK algorithm in FPGA. The accelerator we developed can be connected to the processor using some interface for communication. In this chapter only the accelerator (without interface) is described. The accelerator supports all four SHA-3 hash functions (i.e. SHA3-224, SHA3-256, SHA3-384, SHA3-512). The internal architecture of the accelerator is the same for all four versions of the SHA-3 and only the maximum values of some counters in the control unit are changed. The version of SHA-3 can be dynamically changed after the processing of the message is completed.

The accelerator can be easily integrated into an existing system and connected to the processor via a communication bus. Integration and selection of a suitable environment for testing is described in Chapter 7.

Two clock domains are used: one frequency is used for the data processing part of the computing core itself, and the other for communication between the processor and the HW accelerator. Partitioning of the system into several domains is described in detail in Section 4.4.

4.1 Block Diagram

We will begin the analysis of the accelerator by considering its block diagram. Figure 4.1 shows a simplified block diagram of the accelerator. For readability, the names of individual signals and their width are not shown.

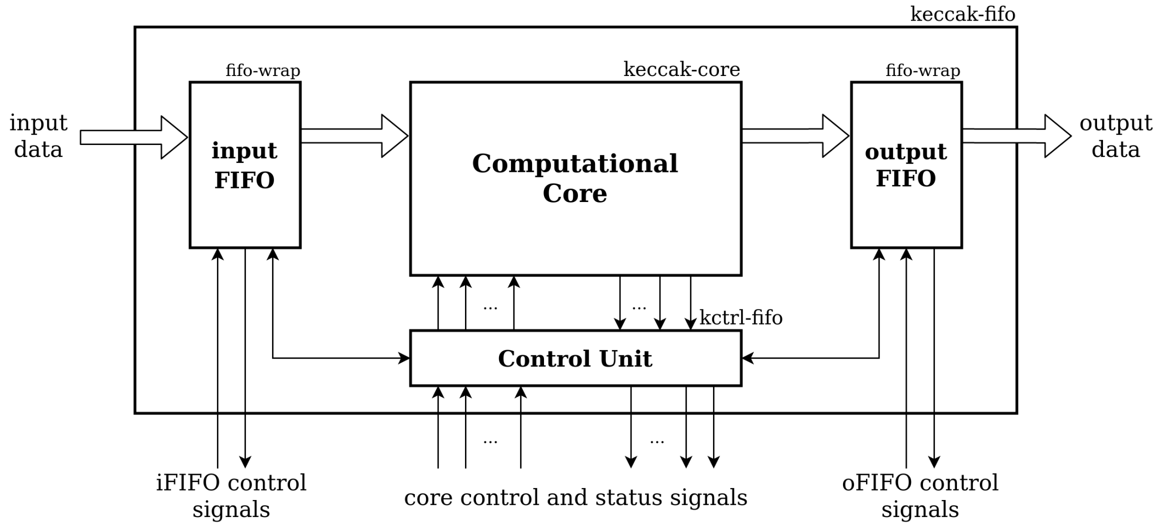


Figure 4.1: Block diagram of the accelerator

All signals of the accelerator can be divided into the following five categories:

- Input data
- Control input FIFO
- Core control and status signals
- Control output FIFO
- Output data

Each group of signals refers to one of the blocks of which the accelerator consists. Each block will be described in Section 4.3, where their structure is analyzed in detail and a description of all signals is given.

Now, in brevity, the principle of operation of this accelerator can be described as follows: the input message is loaded into the input FIFO memory, from where it is then read by the computing core. The core is automated using a control unit, which is responsible for switching the necessary steps and rounds, as well as for sending status messages to the processor about the state of calculations. Upon completion of the calculations, the result is loaded into the output FIFO memory, from where it can subsequently be read by the processor. Despite the fact that the computation is automated, external control is still necessary, which will start loading the message and export the result.

4.2 List of Ports

This section describes all the input and output signals used by the accelerator. Signals are shown in Table 4.1. In addition to the signals, one parameter is used to determine the width

of the data. *DATA_ WIDTH* parameter can be set to either 32 or 64, depending on the processor architecture or chosen communication interface.

Table 4.1: `keccak-fifo` signals description

Name	**Type**	**Width**	**Description**
msg_from_cpu_i	I	DATA_WIDTH	Input message from the processor
SHA3_version_i	I	2	Select the SHA-3 version
resetn_i	I	1	Asynchronous reset. Active low
clk_core_i	I	1	Clock signal in the core domain
clk_io_i	I	1	Clock signal in the I/O domain
run_comp_i	I	1	Starts the computation: load and compute states
run_export_i	I	1	Starts exporting of the digest
wr_msg_en_i	I	1	Enables writing message to input FIFO
rd_dig_en_i	I	1	Enables reading digest from output FIFO
clr_i	I	1	Resets the state $\boldsymbol{s}$ to 0s
compute_done_io_o	O	1	Indicates the end of computation
export_done_io_o	O	1	Indicates the end of exporting
ififo_empty_io_o	O	1	Indicates if input FIFO is empty
ififo_full_io_o	O	1	Indicates if input FIFO is full
ofifo_empty_io_o	O	1	Indicates if output FIFO is empty
ofifo_full_io_o	O	1	Indicates if output FIFO is full
dig_o	O	DATA_WIDTH	Computed digest

The *msg_from_ cpu_ i* signal, which is 32-bit data that will be connected to a certain external bus during accelerator integration, represents an input message for which the hash value must be calculated. This signal passes directly to the input FIFO memory.

The *SHA3_ version_ i* signal is a two-bit signal that allows choosing the version of the SHA-3 function. The version is chosen by the processor and may be changed only after the whole message is processed.

The *resetn_ i* signal is an asynchronous reset that is active low. This signal is used to initialize the whole system.

The signals *clk_ core_ i* and *clk_ io_ i* are clock signals in the computational and input/output domains, respectively. These domains will be described in detail in Section 4.4.

The *run_ comp_ i* signal starts the process of calculating the digest for one block of the input message. If the message has a length of more than one block, it is necessary to start the calculation several times using this command.

The *run_export_i* signal starts the process of loading the computed digest into the output FIFO memory. This command needs to be executed only once, when the whole message has already been processed. Later the processor can read the result from the output FIFO memory.

The signal *wr_msg_en_i* is connected to the internal input FIFO and allows writing to it the input message sent by the processor. After loading the message, the message will be read by the computing core and processed. Several different messages can be loaded into the memory, however, the accelerator itself is not able to distinguish which message the block belongs to, so it is necessary to control such situation with external signals.

The signal *rd_dig_en_i* is connected to the output FIFO memory and allows the processor to read the calculated digest. Thanks to the use of FIFO memory, accelerator can store several different digests, which belong to different input messages and these digests can be read regardless of the current state of the computing core.

The *clr_i* signal initializes the state $\boldsymbol{s}$ to zeroes so that it is possible to start the calculation of the digest of the new message. Also, this signal can be used to abort all pending operations and switch the accelerator to the initial state.

The signal *compute_done_io_o* indicates that the loading of the message into the computing core and its subsequent processing are completed and the core is ready for further work. Based on this signal, the processor decides whether it is necessary to start the calculation again (for example, if there are still unprocessed blocks of one message) or whether it is necessary to export the result to output FIFO memory.

The *export_done_io_o* signal indicates the end of the process of exporting the digest to the output FIFO memory, and therefore, the ability to read data from the memory by the processor using the connected bus.

Four signals *ififo_empty_o*, *ififo_full_o*, *ofifo_empty_o* and *ofifo_full_o* give us information about the state of two FIFO memories, namely, if the each memory is empty or full. These signals are designed to prevent situations such as trying to write to a full memory or reading from an empty memory.

Finally, the signal *dig_o*, which connects the output of the FIFO memory to an external bus connected to the processor. Using this signal, the calculated digest is transmitted to the processor.

4.3 HW Accelerator Hierarchy

Our implementation can be divided into several hierarchical levels and blocks, some of which are instantiated multiple times, forming a block standing in the hierarchy above.

4.3.1 Main Control Unit

One of the main blocks inside `keccak-fifo` is `kctrl-fifo`. It is the main control unit that controls the calculation process itself, input and output FIFO memories, and also receives and sends status signals outside the block. Based on these signals, the calculation process can be started, stopped, and internal state $\boldsymbol{s}$ initialized to zeroes.

The list of signals used to communicate with the `kctrl-fifo` block is shown in Table 4.2 below.

Table 4.2: `kctrl-fifo` signals description

Name	**Type**	**Width**	**Description**
SHA3_version_i	I	2	Select the SHA-3 version
resetn_i	I	1	Asynchronous reset with rising edge synchronized to core clock domain. Active low
clk_core_i	I	1	Clock signal in the core domain
run_comp_i	I	1	Starts the computation: load and compute states
run_export_i	I	1	Starts exporting of the digest
clr_i	I	1	Resets the state $\boldsymbol{s}$ to 0s
compute_done_o	O	1	Indicates the end of computation
export_done_o	O	1	Indicates the end of exporting
load_en_o	O	1	Enables reading from input FIFO
export_en_o	O	1	Enables writing to output FIFO
step_sel_o	O	3	Multiplexer control signal for step selection
round_sel_o	O	5	Multiplexer control signal for round selection
kcore_out_sel_o	O	3	Multiplexer control signal for digest part selection
en_o	O	1600	Enables `keccak-cell` DFFs

This block controls three tasks: loading a message, calculating the result and exporting the digest. To implement its operation we define the finite state machine with the total of 4 states. The first state is the IDLE(0) state, when the outputs of all cells are blocked and cannot be changed and all internal counters are stopped. As we can see in Figure 4.2, from this state system has the ability to go into the EXPORT(3) state and into the LOAD(1) followed by COMPUTE(2) states. After the command *run_comp* is asserted, the system goes into a LOAD state and reads the data block from the input FIFO memory for the required number of clock cycles (based on the length of the read block). The number of clock cycles required to read the message can be calculated in advance, since the length of the block is always fixed for a specific version of SHA-3 function. The counter responsible for controlling whether the message was read, counts up to a certain point and then generate the signal *load_done*. This signal changes the state machine from the LOAD state to the COMPUTE state.

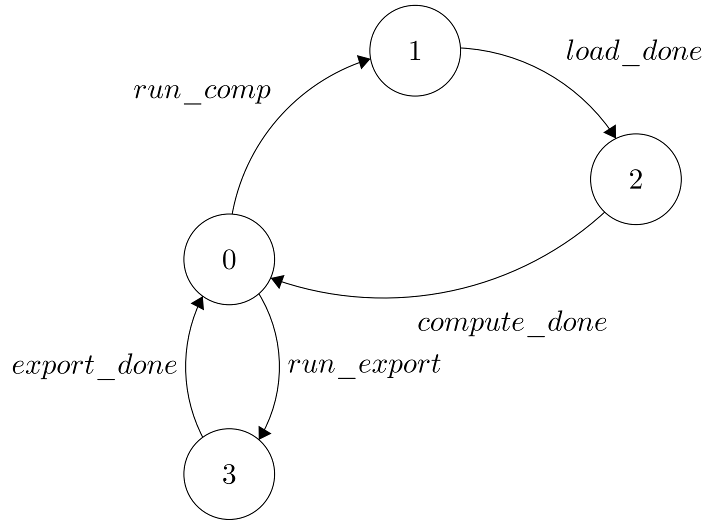


Figure 4.2: `kctrl-fifo` states

In this state, several counters are used, which are responsible for generating control signals for each `keccak-cell` (described in Subsection 4.3.4). One of the counters counts from 2 to 5, that is, it switches the multiplexer between $\boldsymbol{\theta}$, $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ (performed simultaneously), $\boldsymbol{\chi}$ and $\boldsymbol{\iota}$. When these steps are completed, the counter is cleared and begins to count again, moving to a new round. The second counter is used as a round counter, counting from 1 to 24. The counter value increases every four clock cycles, which are necessary to complete the four steps. The simultaneous overflow of both counters indicates the end of calculation of one message block. This generates a *compute_ done* signal, and the FSM returns back to the IDLE state. If there are still further data in the input FIFO memory related to the same message, the data-load and calculation process shall be started again. When the whole message was processed, the digest can be exported, using the *run_ export* signal, the state machine goes into the EXPORT state and writes the result to the output FIFO memory. This operation takes a fixed number of clock cycles, which depends on the digest length.

Figure 4.3 shows an approximate principle of operation of this state machine.

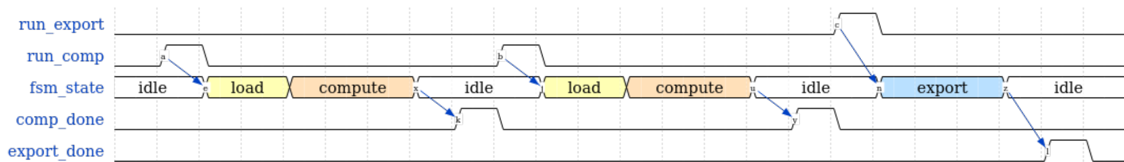


Figure 4.3: `kctrl-fifo` operation

As can be seen, the *run_ comp* signal switches the state machine to the message loading and starts the computational algorithm. At the end of the calculations, the *comp_ done* signal is generated, based on which the processor can either start the next block calculation again or start exporting the result. Figure 4.3 shows the operation of the control unit for a message consisting of two blocks. Despite the fact that the execution time for the calculation and export operations is shown as the same, calculations take up more time than exporting.

Figure 4.4 shows the control signals for multiplexers in `keccak-cells` at the beginning of calculations. We can see how the whole process is started by the *load_ done* command, after which the counters begin to increase.

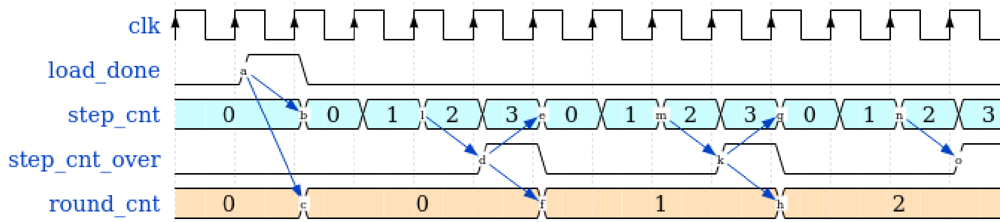


Figure 4.4: `kctrl-fifo` operation in the beginning of computation

Figure 4.5 shows the last few cycles of the calculation state. When both counters overflow (that is, the last step and the last round are executed), the *comp_ done* signal is generated and the counters are cleared.

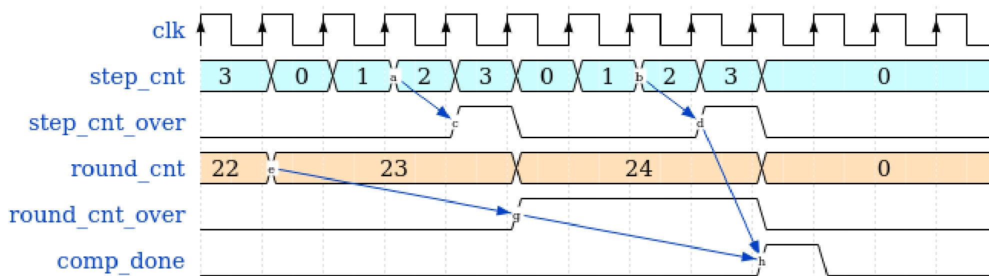


Figure 4.5: `kctrl-fifo` operation in the end of computation

Overflow occurs specifically two cycles earlier, so that as a result there is no delay in the whole system. That is, exactly one cycle is needed to change the values of the registers of overflow and one cycle to clear the counters.

4.3.2 Computational Core

The main control unit, the principle of its operation and the signals that it generates, were described in Subsection 4.3.1 . This section describes the computational core. In Section 3.1, we analyzed some positive aspects of the KECCAK algorithm for digital implementation and created an approximate idea of how it can be implemented. In the same place, we mentioned the creation of a universal cell that will implement one bit of the state $\boldsymbol{s}$ and we mentioned the creation of 1600 such instances (this cell will be described in detail in Subsection 4.3.4). The computational core block combines 1600 instances of the cell, as well as 320 instances of the XOR5 block (see Subsection 4.3.5). Based on the definition of the KECCAK algorithm, all these instances are interconnected in a certain way. In order not to manually create $1600 + 320$ instances, a Python script was written. Based on the formulas given in the KECCAK algorithm description, the script generates a Verilog code with the necessary instances and correct interconnections.

Table 4.3: `keccak-core` signals description

Name	Type	Width	Description
message_i	I	1152	Input message from FIFO memory
resetn_i	I	1	Asynchronous reset with rising edge synchronized to core clock domain. Active low
clk_i	I	1	Clock signal in the core domain
step_sel_i	I	3	Multiplexer control signal for step selection
round_sel_i	I	5	Multiplexer control signal for round selection
kcore_out_sel_i	I	4	Multiplexer control signal for digest part selection
en_i	I	1600	Enables `keccak-cell` DFFs
kcore_out_o	O	DATA_WIDTH	Output of the computational core connected to output FIFO

A new signal in Table 4.3 is *kcore_out_o*, which is connected to a multiplexer, the function of which is described in Subsection 4.3.6. This multiplexer is needed to take the first $\boldsymbol{X}$ bits by 32(64)-bit blocks sequentially (where $\boldsymbol{X}$ is the version of SHA-3, i.e. SHA3-$\boldsymbol{X}$) in order to export the digest to FIFO memory.

The width of the *message_i* signal is 1152, which is the longest message block $\boldsymbol{P_i}$. This length is used by the SHA3-224 function, while others applicable message sizes are smaller than 1152 (1088, 832, 576). Switching between SHA-3 versions will result in loading the input message into the first $\boldsymbol{r}$ cells, while the rest of the cells will remain unchanged.

4.3.3 Internal Input and Output Memories

Two FIFO memories used in this work have already been mentioned several times. In this subsection, we describe in detail why they are used and what functions they implement.

First, we describe the type of memory that we have chosen, namely, FIFO memory with two clock domains. This type of memory allows us to write with one frequency, and read from it with another. Thus, the processes of writing and reading are completely independent, which is beneficial for the following reasons:

- In our design, there are two clock domains and this type of memory implements a clock domain crossing scheme between these two domains
- The computing core can work regardless of whether the processor is currently writing to memory or reading the finished result, thus, the calculation and communication with the processor may overlap with each other

The signals used by the block are described in Table 4.4.

Table 4.4: `fifo-wrap` signals description

Name	**Type**	**Width**	**Description**
di_i	I	DATA_WIDTH	Input data
rd_clk_i	I	1	Read clock signal. For `ififo` is connected to core domain, for `ofifo` is connected to io domain
wr_clk_i	I	1	Write clock signal. For `ififo` is connected to io domain, for `ofifo` is connected to core domain
rd_en_i	I	1	Read enable
wr_en_i	I	1	Write enable
ext_reset_i	I	1	External reset connected to outside of the `keccak-fifo` block. Active low
fifo_empty_o	O	1	Indicates that FIFO is empty
fifo_full_o	O	1	Indicates that FIFO is full
do_o	O	DATA_WIDTH	Output data

To implement the memory, FIFO_DUALCLOCK_MACRO, available from Xilinx, was selected. To use it, a wrapper was created that removes some unnecessary signals and also inverts the reset signal, making it active low. The documentation for the Xilinx block inside the wrapper is available in [34].

4.3.4 Keccak Cell Unit

This subsection describes one of the most important parts of our design, namely the block responsible for a single state bit. This unit both stores the value of the bit and has logic for calculating its new value, that is, each of the five steps of the Keccak algorithm is implemented.

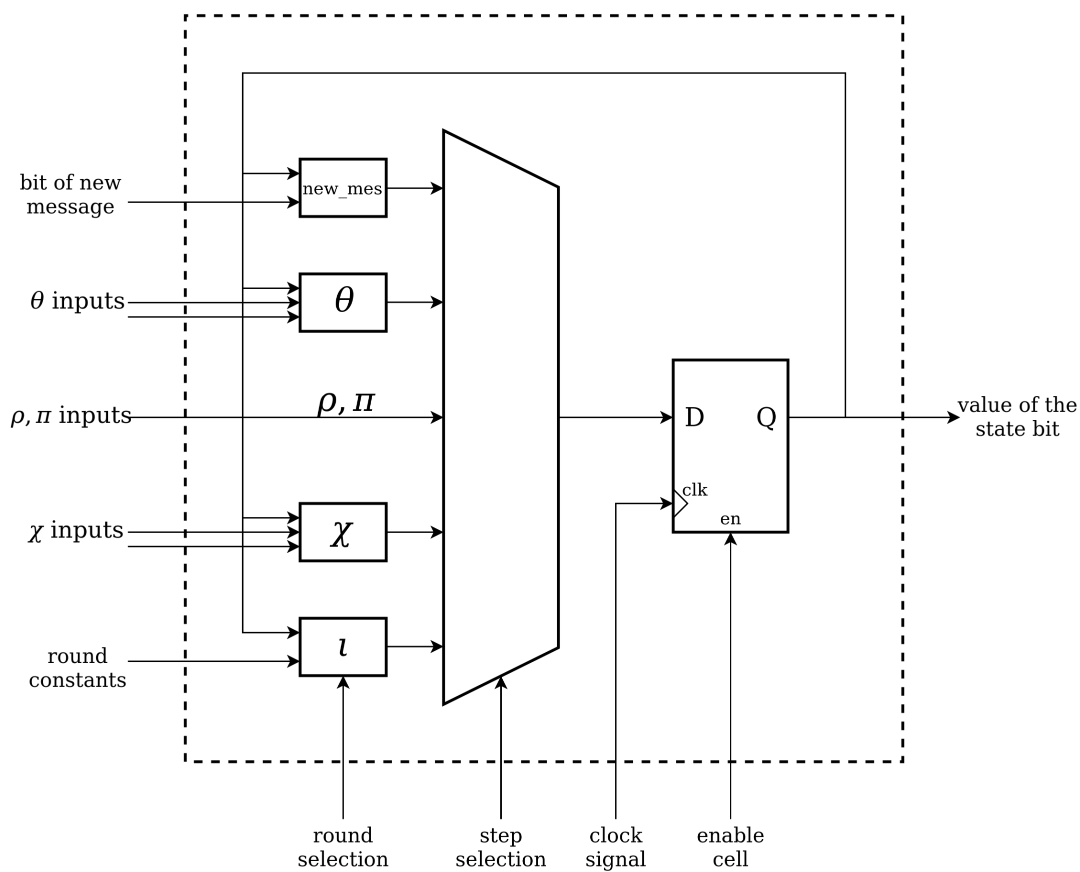


Figure 4.6: Block diagram of `keccak-cell` unit

Figure 4.6 shows the block diagram of the `keccak-cell`. The input signals are divided into data (shown on the left) and control signals (shown at the bottom). All signals are described in Table 4.6.

The key part of the block is the multiplexer, which is responsible for switching the steps. Four of its inputs are responsible for the new bit values and the value is supplied to them after the step is completed. With each new clock cycle, the multiplexer input switches and the new value is written to the DFF, which stores the value of the bit. An additional input for the multiplexer is the output from the `new_mes` block, responsible for the XOR of the bit of the input message with a current bit value. This input is active only when it is necessary to produce a XOR of a new message block with the internal state $\boldsymbol{s}$.

Table 4.5: `keccak-cell` parameters description

Name	Default	Width	Description
iota_param	1	1	Defines if `iota` block will be generated or not
new_bit_param	1	1	Defines if `new_bit` block will be generated or not

Table 4.5 describes two parameters of the `keccak-cell`. They are responsible for generating the block that XORs the input message with the state, and generating logic for the $\boldsymbol{\iota}$ step.

Since the maximum block size is 1152 bits, there is no need to create `new_mes` block for all 1600 bits of the state $\boldsymbol{s}$. It can be included only in the first 1152 cells, which will be combined with the input message block through the XOR function. Similarly, in the $\boldsymbol{\iota}$ step, only the first lane is XORed with the round constant, that is, with the first 64 bits, so it is not necessary to generate a XOR for all bits of the state $\boldsymbol{s}$.

Table 4.6: `keccak-cell` signals description

Name	**Type**	**Width**	**Description**
clk_i	I	1	Read clock signal. For input FIFO is connected to core domain, for output FIFO is connected to I/O domain
resetn_i	I	1	Asynchronous reset with rising edge synchronized to core clock domain. Active low
round_sel_i	I	5	Multiplexer control signal for round selection
step_sel_i	I	3	Multiplexer control signal for step selection
en_i	I	1	Enables `keccak-cell` DFFs
in_i	I	1	Input from FIFO
theta1_i	I	1	First input for $\boldsymbol{\theta}$ step
theta2_i	I	1	Second input for $\boldsymbol{\theta}$ step
rho_pi_i	I	1	Input for both $\boldsymbol{\rho}$ and $\boldsymbol{\pi}$ steps
chi1_i	I	1	First input for $\boldsymbol{\chi}$ step
chi2_i	I	1	Second input for $\boldsymbol{\chi}$ step
iota_i	I	1	Input for $\boldsymbol{\iota}$ step
s_o	O	1	Output bit

The `iota` block works in such a way that for bit 0, bits at zero position of all 24 round constants are taken and every 4 cycles the value at the input of the `iota` block changes, as does the round constant. For bit 1, the first bits of all 24 constants are taken, for bit 2, the second bits are taken, etc. Thus, in the `iota` block there is also a multiplexer, which is responsible for switching round constants.

4.3.5 XOR5 Auxiliary Block

This block is just five input XOR unit. Its function is to calculate the parity of a column, as shown in Figure 2.3.

Table 4.7: `xor5` signals description

Name	**Type**	**Width**	**Description**
xor1_i	I	1	First bit of column
xor2_i	I	1	Second bit of column
xor3_i	I	1	Third bit of column
xor4_i	I	1	Fourth bit of column
xor5_i	I	1	Fifth bit of column
xor_o	O	1	Output bit (parity of the column)

It was introduced into the design in order to simplify the whole implementation and make the `keccak-cell` less bulky, that is, instead of 10 (5 + 5) inputs, we only have 2 for the $\boldsymbol{\theta}$ step.

4.3.6 Output Multiplexer

The multiplexer at the output from the computational core allows to write the parallel output to the 32(64)-bit input of the output FIFO memory.

Table 4.8: `outmux` signals description

Name	Type	Width	Description
kcore_to_mux_i	I	512	First 512 bits of the state
kcore_out_sel_i	I	4	Multiplexer control signal to choose appropriate part
kcore_output_o	O	DATA_WIDTH	Output data to FIFO

Since the maximum digest length of all four SHA-3 functions is 512, the multiplexer consists of 16 32-bits width inputs (8 64-bits) and with the rising edges of the clocks, one of the inputs is sent to the output of the multiplexer. If the SHA-3 version with a shorter digest is used, the counter responsible for exporting the result simply counts in different range.

4.4 Clock Domains

In this section, we will in detail describe the two clock domains used in the accelerator. The first clock signal is used by the computing core, as well as reading from the input FIFO memory and writing to the output FIFO memory. The second clock signal is used by the processor itself and the communication interface. The division of the system into two clock domains allows us to execute communication and computing at two different frequencies. Two independent clock domains allow to further increase the efficiency of hardware acceleration.

One of the main problems that arise when dividing the system into several clock domains is the synchronization of signals moving from one domain to another. To solve it, synchronization blocks are used, at the output of which the signal changes its value with the rising edge of the new clock. When switching from a slow domain to a fast one, there are no problems, since a fast clock manages to capture a slow signal. However, if a signal moves from a fast domain to a slow one, then during one period of the slow clock its value can be changed several times, and, thus, the data would be lost. Since in our design there is only such a type of signal as the pulse (which, for example, means that the calculations have come to an end), we can use the stretching of the pulse for a minimum duration of one period of slow clocks. Thus, this pulse will always be registered during the transition of two clock domains. In addition to the module that stretches the pulses, the double-flop synchronizers are used on all scalar signals between the clock domains.

Chapter 5

Design Flow

The design flow of the accelerator presented in this work is shown in Figure 5.1. The development of the accelerator began with theoretical introduction into the world of hash functions and more precisely the SHA-3 hash function. To choose an effective approach in implementing the accelerator, it is required to deeply understand the theory, so that every aspect of the function is known. Saying this, the KECCAK algorithm, presented in Chapter 2, was studied in detail. In Chapter 3 we provided a deep analysis of the algorithm, so that the most efficient implementation can be chosen.

After the theory introduction we specified some key parameters that we wanted to achieve in our design. We can divide the characteristics of the implemented design into several categories, such as:

- Efficiency
- Power consumption
- Area

Since the main topic of this work is to accelerate the computational process of the hash function, we concentrated ourselves only on achieving the maximum efficiency and computation speed in terms of required clock cycles. This decision allowed us to create a parallel implementation that requires a lot of logic cells. However, as can be seen in Chapter 6, the parallel approach did not require the huge amount of logic cells. Modern FPGA chips can offer much more (over 100k) cells for custom logic implementation.

Our third step was to design the architecture of the system. Chapter 3 contains the reasons, why we chose the parallel implementation, and Chapter 4 describes the architecture of the accelerator. We started creating the architecture by drawing a block scheme, containing all the necessary inputs and outputs and considering in advance, how the accelerator will be connected to the processor. Next, we divided the top level block into several smaller blocks, such as the computational core (Subsection 4.3.2) and control unit (Subsection 4.3.1). The process of dividing the blocks into smaller parts continued until we have reached the lowest implementation level, meaning that only the implementation of single KECCAK step mappings left.

Having considered the architecture of the system, we moved on to describing individual steps using HDL language, namely Verilog. In our RTL design we followed the bottom-up strategy, meaning that we first designed all the low-level units and only after we were sure they function as expected, we moved to the next blocks.

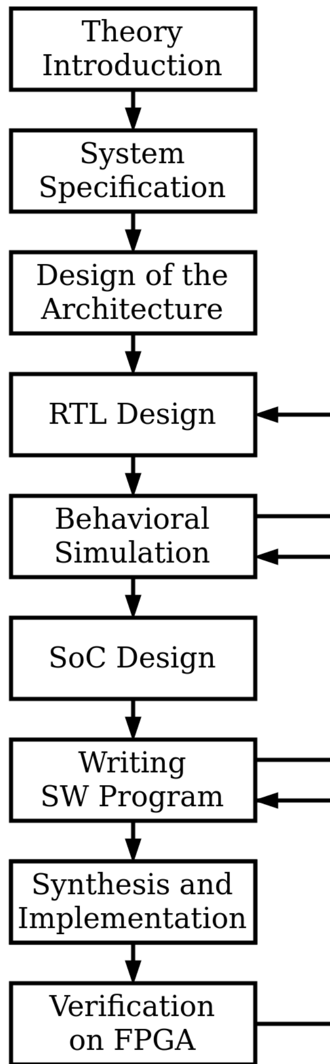


Figure 5.1: Development steps

Consequently with writing the RTL design, new blocks were verified using our custom testbenches. Testbench in digital design is a verification environment that contains the device under test (DUT), and the required signal generation blocks. Basically, testbench should imitate all the possible inputs to the system and be able to create some periodic signals, such as clock signal. For example, after we implemented the `keccak-core` block, we created the testbench called `keccak-core-tb` that contained designed computational core and also the necessary logic to control the functionality of the core. All the control signals from (during that time not-yet implemented control unit) and data signals were presented. This

step allowed us to make sure that the data inside the core are processed as expected. Next, we implemented the control unit `kctrl-fifo` and only after we verified that both blocks function separately, we connected them and corrected the RTL design if needed. The same method was applied to all other designed blocks. Thus, the behavioral simulation using testbenches gave us the feedback on what should be optimized and approved that the design works as expected.

After we verified that the accelerator works correctly, we started designing the complex block design containing the processor, memories and all required elements. This step is described in Chapter 7, which focuses on many aspects, such as choosing the appropriate interface, measuring different communication approaches and ways to compare implemented accelerator with the software function.

SoC design is closely related to writing the software program for the chosen processor that will be responsible for sending data and controlling the accelerator. The program was tested in the created block design using the testbench and behavioral simulation. Chapter 7 also describes the different software approaches for data transferring and the ways to solve potential problems. During the software development we discovered few places in our accelerator that should be optimized. For example, FIFO memories, described in Subsection 4.3.3 can be configured in two different ways: first way sets the data to the FIFO output before the signal Read Enable is active and the second way sets the data after the signal Read Enable was set to logic one. Having the FIFO memory configured to work in the second mode we encountered the problem that AXI4-Stream interface (see Section 7.3) reads one invalid block of data. To correct the error we just reconfigured the FIFO memory to another operating mode.

The last two steps are described in Chapters 6 and 9. After we have chosen the appropriate way to connect the accelerator to the processor, we synthesized and implemented the design and began the measurements of the accelerator's effectiveness on the real hardware.

5.1 Tools Used for Development

The entire development process took place using the Xilinx development tools. These tools include Vivado for creating and verification of the RTL code (as well as integrating the different IP cores) and SDK (Software Development Kit) used for writing the software program for the chosen processor. Xilinx environment was chosen due to the huge amount of available no-cost IP cores that greatly simplify the development process and allow an easy block-design creation. Also, Xilinx offers a lot of user guides and well-documented manuals.

As has been mentioned in Subsection 4.3.2, we used the Python script to create 1600 instances of the `keccak-cell` blocks. To be more precise, the Python 2.7 version and PyCharm IDE were used. The script generated a file of Verilog format, which contained the header, the declaration of the module, signals and 1600 instances of `keccak-cell` blocks. Python program contains all the necessary functions used to correctly interconnect all the instances.

5.2 Verification Environment

Before proceeding to bitstream generation and FPGA programming, the implemented design was verified. To verify every complex block of the designed accelerator we used testbenches that were briefly mentioned in the beginning of this chapter. This section will describe in detail different verification techniques and the ways we analyzed the implemented design. We devote a whole section to testbenches, since this is no less important part of the entire design than regular RTL code.

Every testbench contains the design under test and all the necessary logic that will generate input signals to the DUT. Every signal defined in the testbench can later be viewed in the built-in Vivado waveform analyzer after the simulation began. Furthermore, Vivado lets the designer to look inside the DUT and so every lower hierarchy level blocks can be inspected. This feature greatly simplifies the design process, because the one can have the overall representation of the design and follow the appropriate error to its origin.

There are some signals that were used in almost all testbenches that verified the sequential logic. These signals are the periodic clock signal and the asynchronous active low reset.

To verify that the computational core generates the digest correctly, we had to provide some input data. One way to do this was to create a long (minimum of 576 bits) vector inside the testbench that will be passed as the input to the computational core. The advantage of this method is its simplicity, because it does not require any additional functions or tools, and so it can be used to verify that the core functions as expected for the first time. However, since we wanted to test the core on many different input messages, it was more useful to create the distinct files that contained the input messages. After that, using the `readmemh` function in the testbench, we copied the content of the files into appropriate arrays that were later passed as the input to the core. This method allowed us to separate the input data and the testbench code, which made the code well-arranged and clear.

Since the SHA-3 hash function is officially specified in its standard [12], we had to compare produced digests with some reference official data. Such test data can be found in [31], where for every SHA-3 version there are five different input messages that varies in length and in its content. For example, for every SHA-3 function there is an empty message, as well as a one- or multi-block message. Test data are available as the pdf document which also contains the information about how the data have changed after a single step has been executed inside the Keccak algorithm. Without this information the verification and development process would take much more time, because it is nearly impossible to find where the error occurs due to the high randomness of the output, which excludes any prediction of what went wrong. Using these data and the opportunity to follow the waveforms of the signals inside of the computational core allowed us to quickly find all the errors we had in the design and to put the accelerator into operation.

After we ensured that the computational core, control unit and the input/output FIFO memories function together, we moved on to the creation of the SoC design, described in Chapter 7. Although we could still try to imitate the signals from the processor using the

testbench, we decided that it would we much easier to connect our accelerator to the real processor, because at this stage the signals and the data were at a relatively complex level and it would be hard to maintain them all. After creating the SoC design and the top-level module that combined the processor, peripherals and our accelerator, we wrote a small testbench that emulated the real FPGA development board. It means that this testbench had only the asynchronous reset, clock signal and UART rx/tx signals for further communication with PC (UART signals, were left unconnected in the testbench). Vivado simulations allowed us to follow every waveform of the signals, to find and fix the appropriate errors and to debug the software program using debug AXI4-Stream Data FIFO, described in Subsection 7.1.6.

The last verification step was to add one of the software implementations of SHA-3 function and to ensure that our accelerator and the software implementation provide the same result. This step was verified on the FPGA development board. Using UART for communication with PC we could exchange the data and follow the execution of the program. To compare the execution time of the software and hardware implementation of the SHA-3 function, we used the AXI Timer IP core (see Subsection 7.5.3) that returned the time needed to execute an appropriate function.

Chapter 6

Implementation Results

After we ensured that the accelerator works as expected, we moved on to the next design step, that is, implementation. The output of this step is the design placed on a specific chip. Before starting the implementation, it is necessary to create a timing constraints file. During the implementation, the design is optimized according to the given timing constraints file, so that the desired frequency can be achieved. The implementation result does rely on used FPGA chip, because different chips has different available logic elements.

The implementation was made for Xilinx Artix 7 FPGA `xc7a200tsbg484-1`. The implementation process is fully automated. The first step is the synthesis, when the RTL description is translated to the netlist of suitable FPGA cells. After the synthesis step, the cells are placed in the FPGA and all the nets are routed, using the existing routing resources in the FPGA. If provided placement meets the conditions from the timing constraints file, then the design is ready, otherwise Vivado tries to optimize the placement and routing steps.

Table 6.1: Number of blocks used in implementation

Ref Name	**Used**
LUT6	3322
DFF	1661
LUT4	1604
LUT5	912
MUXF7	121
LUT3	35
MUXF8	32
LUT2	10
LUT1	6
FIFO18E1	2

In Chapter 4 we described the accelerator that does not contain any communication and control interfaces. The interfaces we have chosen are described in Chapter 7 and despite the fact, they have not been yet introduced in this work, we provide the implementation results, that already contain both interfaces. The results of the implementation for required

frequency of 100MHz are shown in Table 6.1.

The implementation results match the expectations from Section 3.2, where we provided the estimation of the resources needed to implement the accelerator. Comparing Table 6.1 with Table 3.1, we can see that the number of DFFs is almost equal. Implementation output contains more DFFs due to the counters, used in the `kctrl-fifo` control unit and resynchronizing modules. The number of lookup tables is also almost equal to the number of logic elements provided in Table 3.1. Only the number of multiplexers greatly differs between the estimation and simulation results. This can be explained by the fact that the multiplexers in our RTL design was translated into built-in multiplexer blocks and combined in different ways by the implementation and optimization tools. Beside basic logic blocks, two FIFO memories were used by the implementation, which also matches the fact that we have two FIFOs in the accelerator.

Having analyzed the implemented design, we provide another Table 6.2 that shows us how many logic blocks we used relative to the available blocks on the chip.

Table 6.2: FPGA resources utilization

Type	**Used**	**Available**	**Utilization**[%]
LUT	5866	133800	4.38
DFF	1661	267600	0.62
MUXF7	121	66900	0.18
MUXF8	32	33450	0.10

Both tables above tell us that the implementation step has been executed according to our expectations and that designed accelerator does not take a lot of space on the chip. The timing analysis of the implemented design gave us the 0.874 ns Worst Negative Slack, which means that we can decrease the period by approximately 0.5 ns. Despite the fact, that the period can be further decreased, we tested the accelerator at 100MHz frequency, so that it has the same frequency with the remaining system, which is described in Chapter 7. Decreasing the frequency by half a nanosecond will give us only about a 10MHz rise, that, as we will see in Chapter 8 will not influence the overall effectiveness of the accelerator, because the most time is spent during data transferring between processor and accelerator and also during the firmware execution, which runs in *clk_io* domain.

Although we have chosen the `xc7a200t-1sbg484c` chip for our implementation and for measurements, it does not mean that the accelerator was designed only for this FPGA chip. We tried to make our accelerator to be as universal as possible and as can be seen in Subsection 4.3.2, the computational core itself does not use any FPGA-specific components. The computational core is implemented using only the basic logic elements, such as LUTs and DFFs, which are the part of every FPGA. The only components in our design that are specific for the chosen chip for the implementation are the two FIFO memories. FIFO memories are implemented using the Xilinx's macro that was specific for the Xilinx FPGAs.

Chapter 7

SoC Design

In order to evaluate how effective the developed accelerator is, it is necessary to create a validation environment. Since the module is designed to accelerate the processor computing SHA-3 function, the validation environment should contain at least a processor. Further, the processor and the accelerator must somehow communicate with each other, which implies the use of a communication protocol. For acceleration to make sense, the connection between the processor and the accelerator should be as fast as possible, otherwise the acceleration efficiency will be reduced.

Since there are a large number of different ways to send data, and each of them is superior to the others in specific cases, we need to determine which one suits us best. For this, we created a special block design that contains all the necessary components to evaluate the effectiveness of the individual communication protocols. In the beginning of this chapter we describe the block design used for evaluating which interface is the best for our application. Later we look at two different interfaces and talk about their advantages and disadvantages as well as provide the necessary measurements. After the suitable interface has been chosen we add this interface to the accelerator and create the new block design used for measurements on the real FPGA chip.

7.1 Block Design for Interface Selection

The block design used for choosing the suitable interface is presented in Figure 7.1. The description of the individual components is presented in the following subsections. Some signals in the design illustrated in Figure 7.1 are not shown, since we did not want the picture to be full of common signals, which can be conveniently grouped and referenced in the text. This is why there are no clock and reset signals. We have already described the clock domains in Section 4.4.

All bidirectional data connections are shown in Figure 7.1 as thin arrows pointing in two directions. We placed all master ports on the right side of components and slave ports on

the left side. The unidirectional arrows represent the AXI4-Stream interface described in Section 7.3.

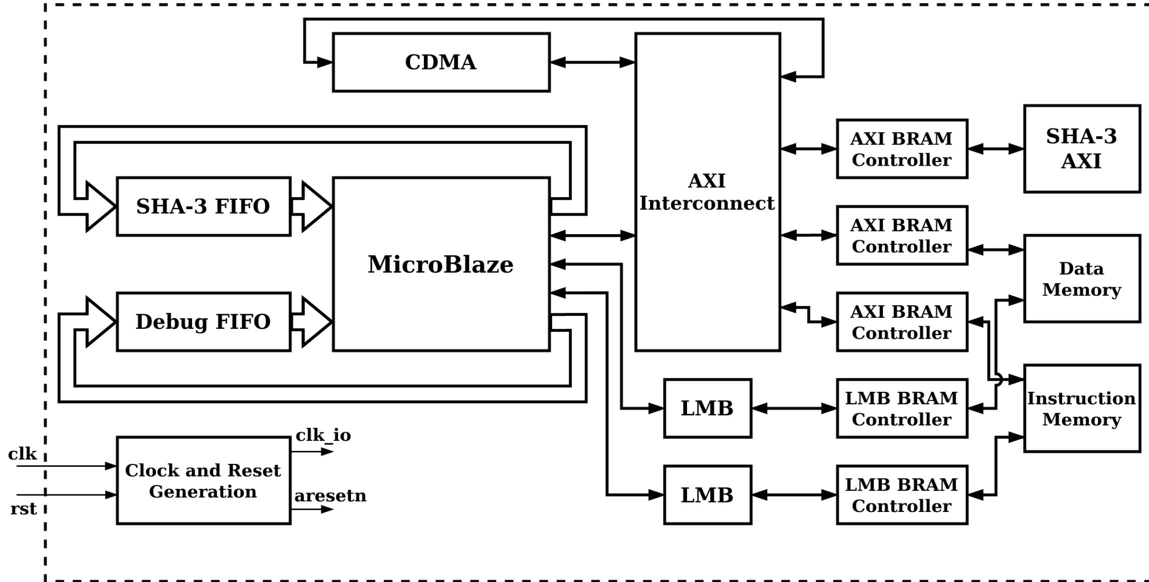


Figure 7.1: Block design for verification and measurements of the suitable interface

The block design presented above does have only two external signals, which are the clock and reset signal. These two signals are triggered in the appropriate testbench, since the measurements of the suitable interface took place only in the behavioral simulations.

7.1.1 Clock and Reset Generation

This section describes the Clock and Reset Generation (CRG) block that is responsible for generation of the global signals, that are later distributed throughout the design. CRG is made out of two IP cores, which are the necessary part of every design with embedded processor. These cores are:

- Clocking Wizard
- Processor System Reset

The main and the only task of the Clocking Wizard IP core [22] is to take an input clocks of some frequency and produce multiple clocks with different frequencies. In our case the input clocks are the external input and IP core produces clocks for two domains: *clk_ io* and *clk_ core*. The advantage of using the Clocking Wizard IP core is its simplicity and that there is no need to create complex frequency dividers in order to create required clock signals.

Processor System Reset IP core [14] is used to generate multiple different resets at the output, that can be independent on each other. This block is used in the design to create resets for the MicroBlaze processor, AXI4 and AXI4-Stream interfaces and also a reset signal for the SHA-3 accelerator (in the design described in Section 7.2). The advantage of

using this IP core is the high flexibility in setting the reset conditions and properties, so, for example, there is no need to create inverters in the design in order to change the reset polarity.

7.1.2 MicroBlaze Processor

The Xilinx MicroBlaze processor [26] was used as a processor core in the validation environment. Since the implementation of the project takes place in FPGA, we needed a processor that can be implemented using only logic blocks. Such processors are called soft processors and there are a large number of them from various manufacturers. One of such processors from Xilinx is the MicroBlaze processor, which is excellent for fulfilling our task and has a number of advantages, such as:

- It can be optimized for better power, frequency or area. Since we want it to work faster and do not care about area or power consumption, we configured it for maximum frequency
- Even when MicroBlaze is optimized for frequency, it does not take much FPGA resources on the chip
- Offers different types of communication interfaces, such as AMBA AXI4 [7] and AMBA AXI4-Stream link [4]
- MicroBlaze is available as a free IP core within the Vivado IDE
- There are a lot of manuals and user guides to MicroBlaze, so the development process was easier

When instantiating the processor core, other blocks are created along with it, responsible for synchronous reset and clock signal generation (see Subsection 7.1.1), as well as a block of the processor's local memory in which data and instructions are stored.

MicroBlaze is initially connected to its local memory through the dedicated bus available only for the processor. This bus is called a Local Memory Bus (LMB) and is designed for a fast data and instructions transferring between the MicroBlaze and memory. Local Memory Bus is connected to the processor through two ports, called DLMB and ILMB each dedicated to Data and Instructions respectively. The advantage of using this bus for accessing a local memory is that other buses, such as AXI4 are not busy by transferring the data and instructions. Another feature of LMB is that the data and instructions are available in one clock cycle, so there is a minimal latency.

7.1.3 AXI Interconnect

Although MicroBlaze has only one AXI4 port we still can connect several slave devices to it. In order to do so we have to use the AXI Interconnect IP core, which basically functions as a switch and allows connecting multiple masters with multiple slaves [19]. Since AXI4 interface uses addresses to indicate the transaction destination, switching between slave devices in AXI Interconnect is accomplished using their addresses. Thus, all slaves later produce a large address space which is described in Section 7.2. In our design, AXI Interconnect IP core allows us to connect all AXI4 slaves to the MicroBlaze processor and also include a DMA (see Subsection 7.1.4) into our design. Since the DMA needs an access to the same slave devices that correspond to MicroBlaze, it is required that DMA and MicroBlaze share the same AXI4 link.

Figures 7.1 and 7.2 illustrates that all AXI4 devices are connected to AXI Interconnect core meaning that they form a common network where the data can be transferred between every unit.

7.1.4 AXI Central Direct Memory Access

Transferring data between devices is a very time-consuming process. When the processor handles the data transfer by itself, it has to configure two devices: the one that wishes to send data and the one that will receive data. Beside that, processor needs to first copy the necessary data to its local memory and then write the copied data to the destination location. This is a very time-consuming process, especially when a large blocks of data have to be transferred. To resolve this problem and free the processor from doing these tasks, the Direct Memory Access unit can be used. The DMA handles the transfer of data and send an interrupt to the processor after completing.

In order to carry out the transfer, DMA has to be configured. Basically, the configuration process does not take a lot of time, because the processor only has to configure DMA source address, destination address and the length of the transfer. After the configuration has completed, the DMA handles the entire data transferring process. Then, there are two basic ways how the processor will be informed about the completion of the transfer. The processor can either periodically check the status register of DMA (polling method) or wait for the interrupt if DMA is configured to work in the interrupt mode. During the time when DMA handles the transfer, the processor is free to execute another tasks.

Xilinx environment allows us to use the free AXI Central Direct Memory Access IP core (CDMA [23]). This core provides data transferring between two devices using the AXI4 interface, which is one of the main interfaces in our design. CDMA has one master and one slave AXI4 port, both connected to AXI Interconnect, so that the processor has the access to CDMA and also CDMA can access all the processor's slave devices.

The huge advantage of the CDMA IP core is its ability to realize a burst AXI4 transfer, meaning that it can send up to 64 blocks of data with a single AXI4 transaction. Using

DMA we can greatly decrease the time required to transfer data even from the MicroBlaze's local memory, because MicroBlaze is not capable of burst transfers, so CDMA can help here.

7.1.5 Local Memory Bus and AXI BRAM Controllers

To access the memory we need some unit that serves as a bridge between the master device and memory, because the memory in FPGA is implemented with the BRAM blocks (Block RAM [21]). After the BRAM IP core is instantiated, it has only one port that can be used for reading and writing. This port further has to be connected to one of the memory controllers. The choice of controller depends on the type of the interface we use to access the memory.

As we have already said in Subsection 7.1.2, MicroBlaze processor uses its own Local Memory Bus due to its high throughput and one clock cycle access [17]. To make the memory accessible for DMA, we had to use the AXI BRAM Controller IP core [15]. If we would like to connect the BRAM memory to the Local Memory Bus, we have to use the LMB BRAM Controller IP core [25].

In our block design (see Figure 7.1) two controllers of each type are used because we have two physically distinct memories: data and instruction. The BRAM memory IP core was expanded to two-ports memory with each port connected via AXI4 or LMB. AXI BRAM Controllers are further connected to AXI Interconnect to include the memory in the common AXI network. AXI Controllers are used so that the DMA has the access to the memory, because the data for SHA-3 accelerator will be stored in processor's local memory and this allows the DMA to transfer data from the processor to accelerator and back.

Advantage of the BRAM memory and AXI Controller is that with their help we can simulate the communication between the processor and the accelerator as if we chose the AXI4 interface. In the block design we can see the SHA-3 AXI block, which is the BRAM memory connected to AXI4 interface through the AXI BRAM Controller. This instantiated block allows us to measure the time needed to transfer the message from the processor's local memory to the accelerator. The time can be measured for AXI4 interface with and without using the DMA unit.

7.1.6 AXI4-Stream Data FIFO

To measure the AXI4-Stream data transfer time we need some IP core that will be able to receive and send the interface signals. One of these IP cores is the AXI4-Stream Data FIFO [24]. The core acts like a FIFO memory with the AXI4-Stream interface, so it can be used as a temporary data storage. The block has two ports: master and slave that should be connected to slave and master ports of another device respectively. In the block design shown in Figure 7.2 two FIFO IP cores are instantiated. The first one (SHA-3 FIFO) is used to simulate the SHA-3 accelerator as if it is connected using the AXI4-Stream interface. The advantage of using that IP core for measurements is its simplicity in integration into the block design, similarly to the BRAM Memory used to simulate AXI4 interface.

The second AXI4-Stream Data FIFO IP core (Debug FIFO) is used for the debug purposes of the firmware program. Due to the fact that interfaces measurements were carried out in Vivado simulations, and not on the real hardware, we could not use `printf` function to debug our program, because it required a virtual serial port and the function would take a lot of time in relation to all other commands. A much simpler solution was to use the additional AXI4-Stream Data FIFO core, to which status messages indicating the progress of the program was sent. Unlike the `printf` function, writing to the FIFO memory with the help of AXI4-Stream interface is executed within one clock cycle and only the TDATA signal in the waveforms had to be tracked.

7.2 Address Space

There are two types of interfaces in the design presented in Figure 7.1. First type does distinguish between slaves using addresses that every slave is assigned and the second type does not use any addresses. Both LMB and AXI4 interfaces are a part of the first type and hence they require that every slave has its unique address or, more precisely, the address range. This range can be directly related to some of the parameters of used block. For example, if we want to access the 64KB BRAM memory using the AXI Controller, we will have a 64KB range in the address space corresponding to that memory. By referring to some address in that range we will directly refer to the BRAM's content. If the device is not a memory, then address range will mean device's register space. The important condition is that within one master device different address ranges can not overlap, because addresses function as a unique identifier for every slave device.

Block design for interfaces measurements has two master devices: MicroBlaze and CDMA, each of them having its own independent address space. Both address spaces are presented in Tables 7.1 and 7.2.

Table 7.1: MicroBlaze address space

Device	**Offset Address**	**High Address**
AXI SHA-3	`0xC000_0000`	`0xC000_FFFF`
AXI Data Memory	`0x4001_0000`	`0x4001_FFFF`
AXI Instr Memory	`0x4000_0000`	`0x4000_FFFF`
AXI CDMA	`0x44A1_0000`	`0x44A1_FFFF`
LMB Data Memory	`0x0001_0000`	`0x0001_FFFF`
LMB Instr Memory	`0x0000_0000`	`0x0000_FFFF`

We can see that address space of MicroBlaze contains two types of interfaces: AXI and LMB. After instantiating the block in the design, Vivado automatically creates offset address for each device. In order to easily access the local data memory, we remapped the default values for AXI and LMB data/instruction memories. Having them as shown in Table 7.1 we can use a simple mask `0x40000000` to access the same data through the AXI interface.

Table 7.2: CDMA address space

Device	Offset Address	High Address
AXI SHA-3	0xC000_0000	0xC000_FFFF
AXI Data Memory	0x4001_0000	0x4001_FFFF
AXI Instr Memory	0x4000_0000	0x4000_FFFF

As shown in the above tables, the same devices do have same addresses for CDMA and MicroBlaze. The reason for this is that if the processor wish to move data from address 0x41C00000 to 0x40600000, it can simply send these addresses to DMA and that will exactly define the source and destination. If the processor and DMA would use different ranges, then we would have to use masking or some complicated address conversion.

7.3 Interface Selection

The selection of a suitable mode of communication is one of the critical aspects. This is due to the fact that even if the developed accelerator is very effective, the advantages of using it can be meaningless if sending the message from the processor to the accelerator takes too long. Therefore, we need to look at several possible options, measure their throughput and subsequently choose the best one.

The MicroBlaze processor allows us to use two types of interfaces: AXI4 and AXI4-Stream. Each interface has its advantages and disadvantages and can give different results in different situations. This is due to the fact that they are implemented in different ways and, for example, the AXI4 uses a long handshake process, but allows to connect multiple devices to the same port at once. In contrast, AXI4-Stream provides a very fast connection, where data are sent within one clock cycle, but the connection is point-to-point and unidirectional. Now we will analyze each interface in more detail, and then summarize the analysis.

AXI4-Stream is a protocol from the AMBA family designed for fast unidirectional data transfer between two devices. Each device using the AXI4-Stream protocol can have two ports: a master port and a slave port, which are connected to the slave and master ports of another device respectively. The data bus is 32 bits wide, and reading/writing is executed in one clock cycle, which is the main advantage of this protocol. Also, the absence of a handshake process and complex procedures for confirming receipt makes the implementation of a slave device (accelerator in our case) a fairly simple task. The disadvantage of the AXI4-Stream interface is that the processor itself sends data, so if there are tasks with higher priority (that is, the processor will constantly be interrupted), then transferring a large amount of data can take a lot of time. Thus, this interface is advantageous to use if the main task of the processor is to transport data.

AXI4 is also a protocol of the AMBA family and is a high-speed method of transmitting data. High speed and excellent bandwidth are due to the fact that the AXI4 protocol data path is divided into several parts and has separate reading and writing channels. Thus, reading/writing can executed in parallel. The width of the data bus in this design is 32

bits, as in the case of AXI4-Stream. Despite the fact that simultaneous writing and reading are an advantage of the AXI4 protocol, this will affect the speed of data exchange with our accelerator only in certain cases. Since the calculation of the digest will take some time, processor will not be able to read the digest of the message it just sent immediately. AXI4 interface can also work in burst mode: when the hand-shaking process occurs only once and remains valid for a large amount of data. Relative to the burst mode mentioned above, we need to consider one important aspect related directly to the MicroBlaze processor. Its architecture and instruction set do not support burst mode. However, we can use DMA, which can transmit data in large blocks with a single transaction. Thus, even before any simulations, we can make an assumption that the connection between the processor and the accelerator will be more effective using AXI with DMA than without DMA. However, only the results of measurements carried out in simulations will give us the exact result.

The shortest possible message that would be sent to the accelerator is 576-bits long (in case of SHA3-512), that is divided into 18 32-bit words to be sent over some communication protocol. The AXI4-Stream can handle this transaction in 18 clock cycles. Even if DMA uses the burst type transfer, it still requires to be configured, which also consumes time and as a consequent, AXI4-Stream might be the best choice for our application.

7.4 Interfaces Comparison Results

The measurements were carried out in the following way: the value of the debug FIFO changed, the execution of the necessary program began, and upon its completion the value of debug FIFO changed again. Then, in simulation output, it was clear how many clock cycles had passed from the beginning of the function to its completion. Since writing to AXI4-Stream debug FIFO takes one clock cycle, the measurement result differs from the real value by only one cycle. This is explained by the fact that when the debug FIFO value changed for the first time, the processor has already begun to fulfill our function (one clock cycle of the new function has already passed). Then, when the function is completed, the processor spends one clock cycle writing a new value to the debug FIFO, and with a rising edge of the clocks, the debug FIFO value changes. However, to compare the SW and HW implementation we used AXI Timer, described in Subsection 7.5.3.

Tests were conducted for messages with different lengths and different methods of sending data were also tested (these methods are discussed in Section 8.1). For example, the different lengths of the messages sent made it possible to verify that the longer is the message, the more efficient the DMA unit is. Since DMA configuration always takes the same amount of time, the longer the message is, the less influence has the duration of DMA configuration on the total message sending time.

In our design we have a total of three different options how to transmit the data from the processor to the accelerator and back. These options include:

- AXI4-Stream
- AXI4 without DMA

- AXI4 with DMA

All three methods were tested under the same conditions and with the same messages. The results for a 1152-bits message are shown in Table 7.3. The third column indicates how many clock cycles (CC) is required for sending a single 32-bit word.

Table 7.3: Interfaces comparison

Method	**Duration in CC**	**CC/32-bit**
AXI4-Stream	75	2.08
AXI4 without DMA	363	10.08
AXI4 with DMA	158	4.39

Based on the measurement results, the most suitable option for us would be to use AXI4-Stream. In second place is AXI4 with DMA, which is expected, since DMA allows using burst mode when sending messages. The choice of AXI4-Stream is also due to the fact that we will send messages whose length depends on the version of SHA-3 functions and the maximum will be 1152 bits. By breaking the longest message into pieces of 32 bits, we get 36 words. DMA, however, begins to show its advantages when sending longer messages. To verify this, we conducted another test in which a message was sent longer than necessary. The results are shown in Table 7.4.

Table 7.4: Using DMA with different message lengths

Message length in bits	**Duration in CC**	**CC/32-bit**
1152	158	4.39
2304	259	3.59
4608	394	2.74

As we can see from the test results, the time spent for sending a message increases disproportionately with the increase in the length of the message, which implies that with increasing length, the efficiency of using DMA increases. We would also like to add that the main priority was the speed of the communication, and not the overall effectiveness of the program. For example, if the processor had a bunch of other tasks, besides communicating with the accelerator, the use of DMA could justify itself. We would have to sacrifice the speed of data transfer in favor of the processor having more time to perform other tasks.

7.5 Block Design for HW and SW Comparison

After all the necessary measurements described in Section 7.4 has been completed and the suitable interface has been chosen, we created the new block design that will help us to evaluate the effectiveness of designed accelerator. The new block design is shown in Figure 7.2.

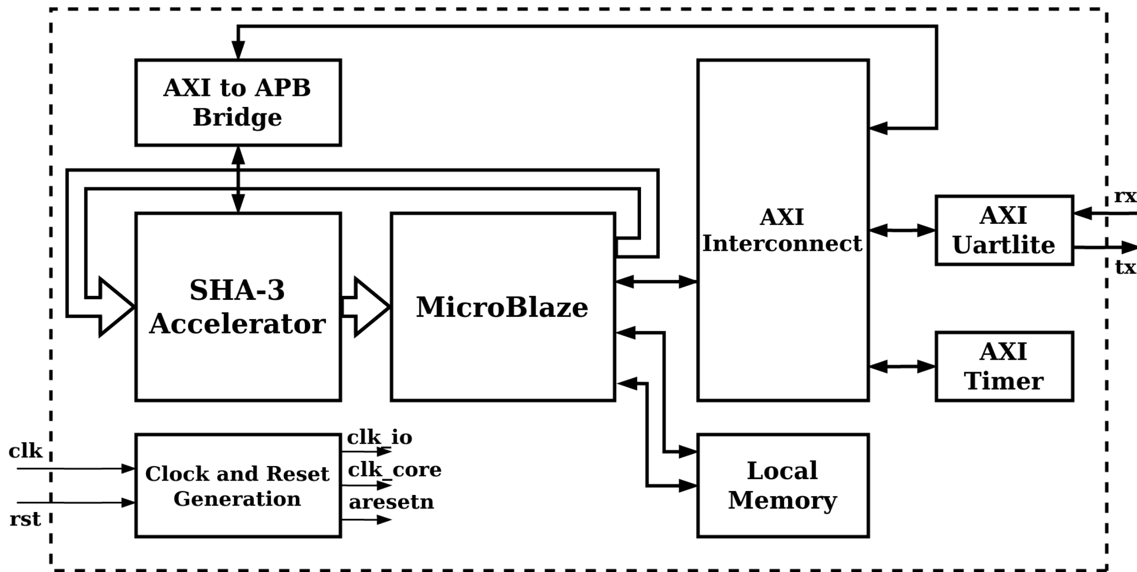


Figure 7.2: Block design for HW and SW comparison

Some of the blocks in the above design are already known to us from Section 7.1. However we can also see new components used in this design, such as the SHA-3 accelerator, AXI Timer, AXI UART and AXI to APB Bridge. Ale new components are described later in this section. Also, the new block design has additional two ports: rx and tx, which are related to the UART communication. These ports are connected directly to the FPGA chips pins. Another difference from the block design in Section 7.1 is that now we use two different clock domains: one for communication and one for the computational core. It means that Clock and Reset Generation block has the Clocking Wizard set to generate two clock signals with two frequencies.

In Chapter 6 the implementation result gave us the frequency at which the accelerator can operate, which is 100MHz. That means that the *clk_ core* domain will run at 100MHz frequency. The MicroBlaze processor and all the peripherals also run at 100MHz, meaning that *clk_ io* domain has 100MHz frequency. From all of the above follows that both domains will be the same in our design. However, the accelerator is still capable to run at the independent from the I/O domain frequency and it can run either at slower or at higher frequency, depending on the situation.

7.5.1 SHA-3 Accelerator

Chapter 4 describes the internal structure of the SHA-3 accelerator. In order to connect the accelerator to MicroBlaze processor we have to upgrade our accelerator and add the interface for communication. The explanation on why we chose AXI4-Stream interface is given in Section 7.3. Since the designed accelerator can not work on its own and should be started by some master device, we need an interface that will be responsible for sending control/status messages.

Sending status and control signals does not require the high throughput of the communication protocol, so we could choose the protocol that is easy to implement and can be connected to the master system (which is the MicroBlaze processor in our case). Xilinx's IP core library offers us the AXI to APB bridge IP core which will be described in Subsection 7.5.2. The information about AMBA APB Protocol can be found in its specification [3]. In order to translate the APB interface signals into control signals for the accelerator, new block named `keccak-fifo-regs` was created. This block serves as the bridge between the APB interface and the accelerator and has three registers.

Control Register

The structure of the control register is presented in Figure 7.3. This register is available at address `8'h00` and is a Write-Only register. Due to the fact, that `keccak-fifo` unit does not require a lot of control signals, the register is 8-bit long and only first 3 bits are used.

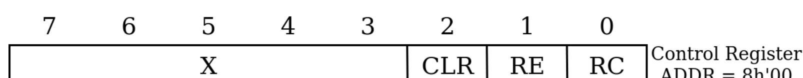


Figure 7.3: Control register structure

The definition of the appropriate bits of control register is given in Table 7.5. Setting the bits to one will generate the signal transmitted to the `keccak-fifo` block and the relative action will begin. The created signal is set to HIGH only for one clock cycle which is sufficient to initiate an action. Bits values can be set only if the WREN signal is HIGH. WREN signal is connected to the APB interface and transmits to logic one automatically whenever the write transaction takes place. One clock cycle after setting the bit, all bits in the control register are cleared until new write operation arrives.

Table 7.5: Control register bit definitions

Bit	Name	Description
0	RC	RUN_COMPUTE bit. Setting this bit to 1 will initialize the COMPUTE operation
1	RE	RUN_EXPORT bit. Setting this bit to 1 will initialize the EXPORT operation
2	CLR	CLEAR bit. Setting this bit to 1 will trigger the CLEAR signal, which, in turn, clears the internal state of `keccak-core` unit
3-7	X	Not used

Status Register

The structure of the status register is presented in Figure 7.4. This is a Read-Only register and is available at address `8'h01`.

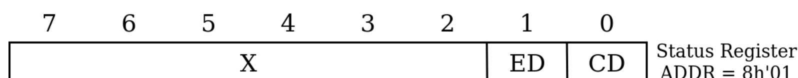


Figure 7.4: Status register structure

Status register is 8-bit long and contains the information about finished actions. The appropriate bits of this register are set when either computation or export has finished. Bits remain HIGH until the action started again or until the register value has not been read by the processor. Register value is read only when the RDEN signal is active, which is automatically active whenever there is a reading operation on the APB interface. The definition of the single status register's bits is provided in Table 7.6.

Table 7.6: Status register bit definitions

Bit	Name	Description
0	CD	COMPUTE_DONE bit. Indicates if the computation has finished
1	ED	EXPORT_DONE bit. Indicates if the exporting to output FIFO has finished
2-7	X	Not used

This register is used by the processor in order to evaluate if the computation is done and as follows, if the processor can initiate a next computation or start exporting the data to the output FIFO.

Config Register

The structure of the config register is presented in Figure 7.5. This is a Read-Write register, so that the version of the SHA-3 function can be changed and also the processor can check which version is running now.

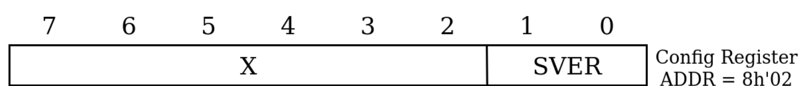


Figure 7.5: Config register structure

The definition of the config register's bits is in Table 7.7.

Table 7.7: Config register bit definitions

Bit	Name	Description
0-1	SVER	SHA-3 VERSION bit. Changing the value of these bits will cause the accelerator to change its actual version
1-7	X	Not used

The lower 2 bits of the config register are connected to the *SHA3_version_i* input of the keccak-fifo unit. After the register value has changed, the accelerator is ready to operate according to the new set value.

In order to receive or transmit any data through the AXI4-Stream interface, designed accelerator should contain the logic that will be responsible for generating all the signals required by the AXI4-Stream standard. That is why we created two additional blocks each responsible for transforming master/slave signals of the accelerator. These blocks are called axis-master-interface/axis-slave-interface and are placed at the master/slave AXI4-Stream ports respectively. These units are implemented only with combinational logic, so that there is no unnecessary delay in transactions. The minimum requirement for the AXI4-Stream interface are the three signals: TREADY, TVALID and TDATA. Signal TDATA is a 32-bit bus used for data transferring. Signals TREADY and TVALID are the 1-bit control signals used in the simple handshake process. TREADY is always generated by the slave device and indicates that the device is ready to obtain the data. TVALID signal indicates that the master is driving a valid transfer. The data on the link are valid when both TVALID and TREADY signals are high.

Since the processor is the unit, which initiates both reading and writing from accelerator, we had to implement the interface in the way that the accelerator's data will be ready as soon as possible. For example, axis-slave-interface block has the TREADY signal active when the input FIFO is not full. This means that the accelerator can immediately receive any data from the processor if there is a free space for them. Another task of axis-slave-interface unit is to send a Write Enable signal to input FIFO. The WREN signal is active when TVALID is high and the input FIFO is not empty.

On the output side of the accelerator axis-master-interface is placed. Its role is similar to axis-slave-interface unit with the difference, that it is responsible for TVALID signal generation and does receive TREADY from the processor. The working principle is similar to its slave analog in the way that it sets the TVALID signal high when the output FIFO is not empty, meaning that there are still some data ready for transferring. This device is also responsible for allowing reading from output FIFO. The Read Enable signal is active when the output FIFO is not empty and TREADY signal is high. The unit also changes the byte order inside the 32-bit word to little-endian format in order to later effectively read the data by the software function, which is described in Chapter 8.

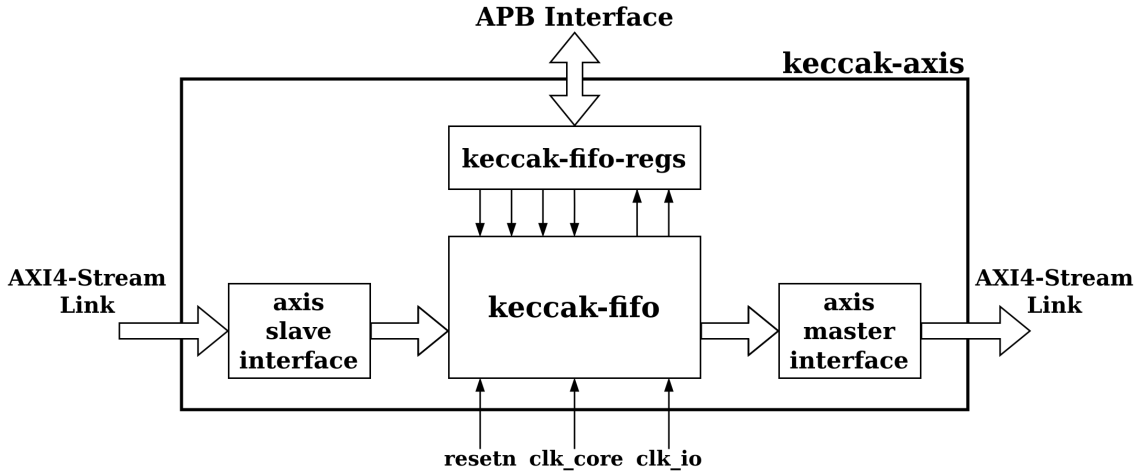


Figure 7.6: `keccak-axis` block diagram

Adding the blocks described above in this Subsection results in the accelerator wrapper, which we called `keccak-axis` to reflect that this accelerator uses the AXI4-Stream interface. Figure 7.6 illustrates the structure of the complete SHA-3 accelerator that can be integrated in the larger system and has two different interfaces each following its own purpose.

7.5.2 AXI to APB Bridge

The AXI4 to APB Bridge IP by Xilinx [13] is used as a bridge between the AXI4 interface and the APB bus, which is used as a control interface of the accelerator.

7.5.3 AXI Timer

In order to properly measure the program execution time, some external timer has to be used. One possibility was to use AXI Time IP core [16]. The single core integrates two independent 32-bit timers. Provided IP core has several advantages including a very simple register space and support of AXI4 interface.

To use the timer, we first had to reset its value to zero. The next step was to start the timer right before the execution of an appropriate part of the program (that has to be measured), stop the timer after program finished and read the timer value. All the control and data signals are transferred over the AXI4 interface. The timer is connected to AXI Interconnect to allow the MicroBlaze access its register space.

The timer IP core has a clock signal input that later is a reference during conversion of the timer output to time units. We connected the timer to *clk_io* domain, because this is the domain of the processor and we want to know how fast the accelerator is relative to the processor computation speed.

7.5.4 AXI UART

The usage of the AXI4-Stream Data FIFO as a debug instrument is bounded to the simulations in Vivado. We can not use FIFO for the debug purposes while running the program on the real hardware, because we do not have the opportunity to follow every signal as we do in the Vivado waveforms. To be able to debug our program while running it on FPGA, we used the AXI UART IP core [20]. The core translates AXI4 transactions into serial communication and do the reverse. UART ports are further connected to USB/UART unit on the development board, which is further connected to PC's serial port. The usage of the UART core allows us to send to PC the data containing, for example, the results of comparing SW and HW implementation of SHA-3 hash function. Another reason why we chose the UART IP core is that having the core in the design makes the MicroBlaze to automatically choose this interface as a standard input/output for `xil_printf()` function (this is a special Xilinx's light version of `printf()` function).

Chapter 8

Analysis of Firmware Function

In order for the developed accelerator to be conveniently used, it is necessary to create a firmware function to which the message will be passed and which will return the digest. Also, this function will help us to easily compare the efficiency of computing a hash function with and without the accelerator.

To compare our firmware function with the chosen software implementation of the SHA-3 function, we need our firmware function to have the same parameters. The parameters that are passed to our function are:

- Input message as an array of bytes
- The length of the input message
- Array of bytes for storing the digest value
- The digest length (which also indicates the SHA-3 version)

The message received by the function can be divided into several parts that we often refer to in this chapter. The structure of the input message is shown in Figure 8.1.

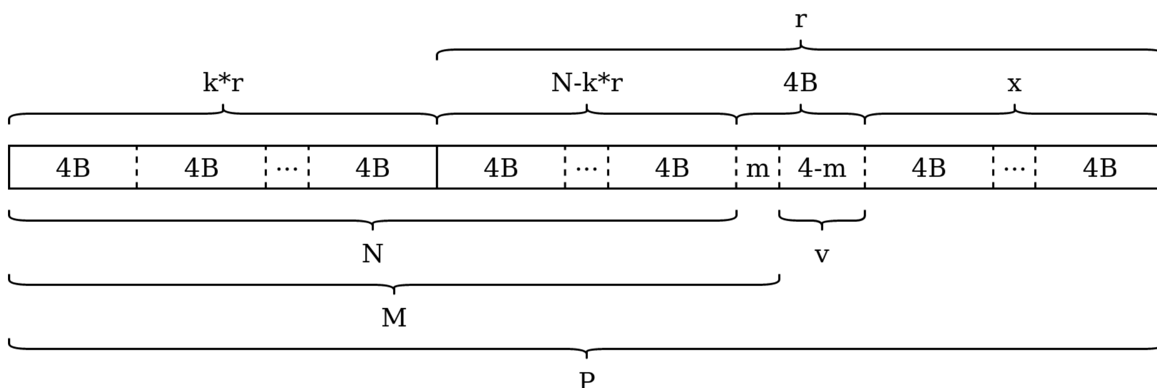


Figure 8.1: Message structure

The parts shown in Figure 8.1 have the following meaning:

- **P** is the length of the message that has to be sent to the accelerator
- **N** is the part of the received message that can be divided into the real number of the 32-bit words
- **M** is the length of the message received by the function
- **r** is the rate size given by the SHA-3 version
- **m** is the number of bytes of the message that remained after subtracting **N** and that does not consist of 32-bit words. The variable **m** can be either 0, 1, 2 or 3 and hence is always modulo 4
- **v** is the number of bytes that should be added to the received message in order to make the message divisible by 32. Variable **v** is equal to $4 - \mathbf{m}$
- **x** is the number of 32-bit words that should be added to form the new message, which new length **M** would be divisible by the rate size **r**

After we have familiarized with the input message structure and the variables that we use, we can move to describing how the firmware function is executed. The task of the firmware function is to receive the message as a byte array, add the suffix (see Section 2.1), pad the message, send the padded message to the accelerator, run the compute and export states, receive the digest and return it in the function. Basically, our firmware function does follow this structure and pattern, however, we tried to optimize few steps and now the function is executed in the following way:

1. Clear the computational core of the accelerator and prepare the accelerator for the new input message. This is executed by setting the CLR bit of the control register.
2. Create the function variables and set them according to the required SHA-3 version that has been passed as the argument to the function. These variables include setting the rate size **r**, the digest length in words and configuring the SHA-3 version of the accelerator. Configuring the version is achieved by writing to the SVER bits of the config register.
3. Since the designed accelerator is capable to process the data and receive the new message block simultaneously, the next function step is to try to send the whole message block and start the computation. This step can be described using Algorithm 3. Variable **N_temp** is assigned the value of **N** so that it is not changed by the following program and can be used later. The number **P** indicates how many blocks have been pipelined this way. Algorithm 3 tells us that pipelining is used only when the message length is higher than rate size **r**. Also, we wait for compute state to finish only if we already started the computation once.
4. After the possible pipelining is executed, the function sends the remaining $\mathbf{N} - \mathbf{r}$ 32-bit words.

5. Next step is to compute all the necessary variables for adding the suffix and applying the padding rule to the received message. These variables include **v** and **x**. If the message should be padded by multiple 32-bit words, then these words (made of zeroes according to the KECCAK padding rule) are directly sent over the AXI4-Stream interface.

6. After all the data has been transferred to the accelerator, the last computation process is initiated by setting the RC bit of the accelerator's control register. All other blocks and computation steps were initiated in point 3 of this list. The function uses the polling method to wait until the computation is not finished and checks the CD bit of the status register.

7. After the computation is over, the program starts the export of data by setting the RE bit of the accelerator's control register. Using polling method program waits until the export to the output FIFO is finished and constantly checks the ED bit of the accelerator's status register.

8. The last step is to read the digest value from the accelerator using AXI4-Stream interface.

The pipelining algorithm is shown below:

Algorithm 3: Pipelining the data transfer and computation

```
P = 0;
N_temp = N
while N >= r do
    send r/32 words;
    N = N − r;
    if P = 0 then
        compute;
    else
        wait for compute to finish;
        compute;
    end
    P = P + 1;
end
```

The time spent during the execution of the firmware function can be divided into three categories:

- Data transfer between processor and accelerator
- Data processing
- Other software operations related to calling functions, padding, etc.

Each category is described in this chapter in Sections 8.1, 8.2 and 8.3. These sections also provide some thoughts on how each process could be optimized in the future. The influence of every process on the total execution speed is described in Section 8.4.

8.1 Data Transferring

There are only few memory accesses that take its place in the beginning and in the end of the computation. To provide a hash value of the message, the message should be loaded into the accelerator and then the produced digest should be exported back to the processor. For more efficient communication between the processor and the computing core, we included two FIFO memories into the accelerator. The first FIFO memory serves to load the input message from the processor and then read it by the core, and the second FIFO saves the digest value so that processor can later read the result. For example, thanks to the FIFO memories, the processor can send multiple messages, execute other tasks (not related to SHA-3) and read multiple digest when they are ready. Hence, the communication between the processor and the accelerator is more efficient in the way, that the accelerator is always ready to receive the data (until the input FIFO is full, of course) and does not have to somehow interrupt its computational process. In the similar manner the processor can freely read the ready digests regardless of what the accelerator is currently doing.

We can divide the communication into data transferring and controlling the loading/exporting phases of the accelerator. Data transferring takes place only during sending the message to the accelerator and exporting the result to the processor. Control communication occurs every time the accelerator should perform the computation or export the digest to the output FIFO. However, the computational process (i.e. executing the 24 rounds of the KECCAK-f permutation function) is absolutely autonomous.

In Section 7.3 we looked at different interfaces and compared them between each other. During the comparison we tried the different ways of transmitting the message and saw how they affect the overall effectiveness of the data transfer. In Section 7.3 we looked only on the results of the measurements and did not describe the problems that we encountered. Since this section is dedicated to the firmware function and the data transferring process, we decided to look closer to the ways we sent the messages.

Different ways of transmitting a message do not only mean using various communication protocols, but it also means the different software approach to a given problem, i.e. changing the way program executes writing to peripherals and reading from them. For example, the standard compiler settings do not use the most efficient code optimization, as a result of which many functions are translated into large sequences of assembly instructions that take too long to execute, which in turn affects the overall duration of the program. Because of this we used the maximum possible compiler optimization, that is, `-o3` (optimize most). Also, despite the fact that theoretically the AXI4-Stream protocol is capable to send data in one clock cycle, problems appear during reading from memory and consequent writing to peripheral. The problem is associated with the processor's pipeline architecture. Namely, looking at the assembly code of the function responsible for writing data to the AXI4-Stream port, shown in Figure 8.2, we can see that it consists of two instructions: `lwi` and `put`.

```
888:   e861052c    lwi r3, r1, 1324
88c:   6c038001    put r3, rfsl1
890:   e8610530    lwi r3, r1, 1328
894:   6c038001    put r3, rfsl1
898:   e8610534    lwi r3, r1, 1332
89c:   6c038001    put r3, rfsl1
8a0:   e8610538    lwi r3, r1, 1336
8a4:   6c038001    put r3, rfsl1
```

Figure 8.2: Standard `putfsl` function assembly code

The first instruction loads the data from the memory into one of the general-purpose registers. The second instruction is intended to put the value of the register to the AXI4-Stream peripheral port. Since MicroBlaze uses a pipelined architecture, the data hazard is created, because the value of the register `r3` (in case shown in Figure 8.2) will change only in the Write Back pipeline stage, and the pipeline is stalled for two clock cycles, so that the Instruction Decode of the `put` instruction will work with correct register value. As the result we need four clock cycles to send one word. This problem, however, could be solved by rearranging the instructions, so that first there will be four `lwi` instructions that would change registers `r8, r7, r5, r4` and then there will be four `put` instructions that will use these four registers in the same order. After such reordering we give each register a time of three clock cycles to be written back in the memory, so that `put` instructions will read the correct value without the need to stall the processor. Reordered instructions are shown in Figure 8.3.

Changing the order of instructions, along with `-o3` level optimization, allows us to send four 32-bits data words in 12 clock cycles (4 extra cycles to get to the beginning of the `for` loop).

```
a2c:   e901052c    lwi r8, r1, 1324
a30:   e8e10530    lwi r7, r1, 1328
a34:   e8a10534    lwi r5, r1, 1332
a38:   e8810538    lwi r4, r1, 1336
a3c:   6c088001    put r8, rfsl1
a40:   6c078001    put r7, rfsl1
a44:   6c058001    put r5, rfsl1
a48:   6c048001    put r4, rfsl1
```

Figure 8.3: Optimized `putfsl` function assembly code

We use the method shown in Figure 8.3 to send the data to the accelerator and to receive the data from the accelerator. The data transferring is realized in a way, that we try to send the data by four 32-bit words per single `for` loop iteration until it is possible and then send the rest of the words by one word per `for` loop iteration. This approach optimizes the data transferring process for long messages, however has a disadvantage for a messages where we have to send three words, because these words are sent inefficiently. We could divide our program into more specific cases, where we will try to implement the effective data transferring for all message types, however this program would have so many `if` conditions, that it will waste a lot of time during these `if` checks. Using the approach of sending data by four 32-bit words we sacrificed the effectiveness for the short messages, however gain effectiveness for long messages.

Receiving the digest does slightly vary from sending the message, since we know in advance how long the received block of data would be. Three of four SHA-3 function versions does have the digest length divisible by four, which led us to the approach shown in Figure 8.3. Functions used for communication through the AXI4-Stream interface are `putfsl` and `getfsl`. The approach from Figure 8.3 can be applied to both functions, although we showed it only for `putfsl` function. The fact that three versions of SHA-3 function can use the approach from Figure 8.3 lead us to insert an `if` condition, which divides the SHA-3 versions into two categories: SHA3-224 and SHA3-256, SHA3-384, SHA3-512. The first category (SHA3-224) receives the digest without `for` loop using a simple list of seven `getfsl` functions and buffers between them to get rid of the data hazard. The second category receives the digest using `for` loop that gets the digest by four 32-bit words.

8.2 Data Processing

After all the necessary data has been transferred to the accelerator, the data processing part can begin. Data processing means the execution of the accelerator's computational core. It runs in parallel with the processor and hence is independent on the processor. The fact that the core is independent and can not be interrupted tells us that the necessary amount of clock cycles, required to process the data, is always the same for the same SHA-3 function and message length.

In Chapter 4 we described how the accelerator work and told that the execution of the single round takes only four clock cycles. Further, for every SHA-3 function there are 24 rounds to be executed for a single message block. Since the number of rounds and amount of clock cycles per round is independent on the SHA-3 version, we can say that the computational process (COMPUTE state in Subsection 4.3.1) is the same for all SHA-3 versions. The difference between SHA-3 versions is in the block size and hence changes the duration of the LOAD state, where every 32-bit word is processed in one clock cycle. After all the digest has been computed and the command to start exporting the digest arrived from the processor, the accelerator begins exporting the digest into its internal output FIFO. If we do not consider now the time that accelerator has been waiting for the commands form the processor, then we can exactly define how many clock cycles is necessary to execute the processing of one message block.

Table 8.1: Data processing

Action	**One block**	**Two blocks**
LOAD	$\mathbf{r}/32$	$2\times\mathbf{r}/32$
COMPUTE	96	2×96
EXPORT	$\mathbf{l}/32$	$\mathbf{l}/32$

Table 8.1 shows how many clock cycles is necessary to process a single and two-block input

message. Two variables **r** and **l** are the rate size and the digest length. We can see from the table that the EXPORT state does take the same amount of time with increasing message length. This is due to the fact, that the digest has to be exported only once for a single message. As we can also see from Table 8.1, the processing of data depends only on the version of SHA-3 function and on the length of the message.

If we wish to optimize the data processing step, we could only optimize the LOAD and EXPORT states of the accelerator, because the COMPUTE state is given by the accelerator's architecture and its parallel approach and does not depend on the SHA-3 version. The LOAD and EXPORT states can be optimized by expanding the data width of the internal FIFOs to, for example 64 bits and, hence, loading the message into the accelerator twice as faster. Exporting will also require twice less time to complete. However, if the data width of the FIFOs will be incremented, the width of the AXI4-Stream interface should also be incremented to the same width as FIFO.

8.3 Remaining Operations

Beside the data processing and data transferring steps remain all the other firmware operations, which have to be executed in order to make the function work. These operations include function calling, creating and assigning variables, running through `if` and `for` loops, controlling the accelerator and etc. As we will see in Section 8.4 this category takes most time of the three categories, to which we divided our firmware function.

We divide the firmware operations into several categories and suggest the ways these categories can be improved in the future work.

1. Controlling the accelerator through the AXI4/APB protocol. During the execution of the firmware function we use AXI4/APB protocol at least four times: to clear the accelerator, to configure its version, to start computation and to start export. All these commands require the communication through the AXI4 protocol, which does have a long hand-shake process and so to send the single CLEAR command takes about 20 clock cycles. This problem can be eliminated by creation of the control unit in FPGA for the accelerator that will by itself start the computation and exporting. However, this control unit will still require the control by the processor, because the control unit by itself doesn't know how long the message is and when the processor wishes to read the digest. So, that will not really solve the problem. Another option would be to create some unit similar to DMA that will be once configured by the processor in the way, that the processor will say where the data for SHA-3 are, how many messages are there and where to store the digest. After configuration, the unit will load the data into input FIFO, execute the computational state, store digest into required place and send the interrupt to the processor.
2. Variables assignment. The firmware function requires some internal variables in order to work correctly, so this step can not be eliminated in our designed function, neither somehow optimized, because we want our function to be universal and work for every SHA-3 version. This means that every time we need to define some internal variables.

3. Padding of the message. This step is executed with the help of `v` and `x` variables shown in Figure 8.1. Using several `if` loops, firmware function checks how many bytes should be added to form a 32-bit word (which later can be send via AXI4-Stream) and how the padding should be applied. Theoretically, the padding can be executed by the control unit that we described in the point 1. However, the control unit would have to know the length of the required output message and then execute the padding.

4. Passing the arguments and variables. The firmware function takes two arrays of bytes. The first one is the input message and the second one is the digest. Inside our firmware function we just create a 32-bit pointer to these arrays and thanks to the data alignment in the processor's memory, we access the byte array through the word pointer. This is an already optimized approach, because before that we transposed the byte array into word array using shifting of the byte variables. The disadvantage of non-optimized version was that shifting takes a lot of time and hence the execution of the firmware function is not so effective, especially for the longer messages. Optimized version uses shifting only for the `v` bytes to form a 32-bit word.

Basically, the above list presents the main operations that require the most time of the "other operations" category. Another, less time-consuming operations are returning back to the `for` loop and executing `if` conditions. Their influence is smaller, because we made our code universal and tried to use as few `if` conditions as possible.

8.4 Influence of Each Aspect on the Total Execution Time

This section is dedicated to the analysis of how each category from the beginning of the chapter influences the overall time required to execute the firmware function. First, we want to know, how many clock cycles is required to execute the whole algorithm and what is the minimum required amount of clock cycles, which is not possible to further optimize using our accelerator. The ideal case of digest computation for a single block message is shown in Figure 8.4.

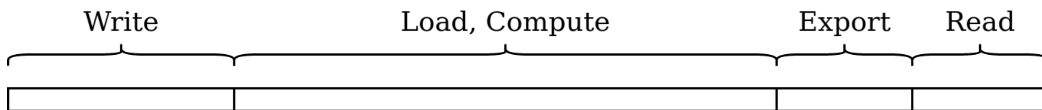


Figure 8.4: Ideal case of digest computation

To present some concrete values of clock cycles needed to execute single steps, we will look at the SHA3-256 hash function. The difference between the ideal and real cases is shown in Table 8.2.

Table 8.2: Ideal vs real firmware execution

Case	**Write**	**Load**	**Compute**	**Export**	**Read**	**Total**
Ideal	34	34	96	8	8	180
Real	106	34	96	8	24	268

The numbers in the **real** row of the table are created using the assumption about the AXI4-Stream interface from previous chapters and interface analysis. The write operation in Table 8.2 means writing the message into the accelerator. This step is 106 clock cycles because we need to transfer the 34 words to the accelerator and we said, that we send them by four words, meaning that one `for` loop execution (responsible for sending these data) takes 12 clock cycles: eight to send four words and four to return to the beginning of the `for` loop. Sending the message this way gives us 8 `for` loop iterations which sends by four words and two `for` loop iterations to send remaining two words. All this in sum gives us 106 $(96 + 10)$. The load, compute and export states are not affected by anything as has been stated in Section 8.2. The reading of the digest is executed as has been described in Section 8.1 and hence requires only 24 clock cycles. From the comparison results provided in Table 8.2 we can see that the data transferring step has to be optimized in the case if we would like to have higher accelerator efficiency. The optimization could be done by expanding the AXI4-Stream data width, however, the MicroBlaze processor is capable to use only a 32-bit AXI4-Stream interface. So, expanding the data width of the AXI4-Stream interface will not solve the problem if we wish to use the MicroBlaze processor.

The results in Table 8.2 do not contain the firmware operations beside the data transferring and processing. To find out, how many clock cycles are spent during other operations, we need to subtract the "Total" value in Table 8.2 from the value that we obtained in the real measurements described in Chapter 9. We provide the result of subtraction for three different message length of SHA3-256 function in Table 8.3 and Table 8.4.

- **Length** is the length of the input message
- **DP** is the data processing category
- **DT** is the data transfer category
- **OO** is other operations category

If we take the SHA3-256 case for a zero-length input message from Table 9.2, we get 461 clock cycles required to execute the whole firmware function. Subtraction of 268 from 461 gives us 193 $(193 = 461 - 268)$ clock cycles. This amount of clock cycles was spent by the firmware function on the other operations than data transfers and data processing.

Table 8.3: The influence of each category on the total execution time in clock cycles

Length	**DP**[CC]	**DT**[CC]	**OO**[CC]
0	138	130	193
21	138	130	381
168	268	236	443

Table 8.4: The percentage influence of each category on the total execution time

Length	**DP**[%]	**DT**[%]	**OO**[%]
0	30	28	42
21	21	20	59
168	28	25	47

The single AXI4 transaction takes about 20 clock cycles and as we have said in Section 8.3 we make a minimum of four transactions, implying that is consumes 80 clock cycles for all of them for a single-block message. After subtracting this number from 193 we obtain 113 ($113 = 193 - 80$) clock cycles, which remained for other operations. The large number of remained clock cycles can be explained by the polling method of checking the status register of the accelerator. Every time the firmware wish to check the accelerator status register, it initialize a read AXI4 transaction, which also requires 20 clock cycles. The status register reading occurs at least twice giving us another 40 clock cycles. It is important to note that 20 clock cycles for checking the status register is the ideal case, because the register value can be checked a clock cycle before it is asserted, meaning that this reading will tell the processor, that computation/exporting is not done yet and the processor should read the status register value again, wasting 20 more cycles. Subtracting 40 clock cycles from 113 we obtain 73 ($73 = 113 - 40$) clock cycles. They are required to run through all the `if` conditions that are in the firmware function and to create/assign different variables. We would also like to note that for a zero-length message the data are sent in the following way:

1. Send a suffix `0x00000006`
2. Send a necessary amount of words with zero value, which is 32 in the case of SHA3-256. These words represent the padding zeroes
3. Send the end of the padded message `0x80000000`

The above sequence means that not all 34 words are transferred using the `for` loop, which could also add few clock cycles to the remained 73 in the firmware function.

The growth of execution time with increasing message length can be explained in larger data transfers and more complicated padding cases, where the shifting can be required in order to form a 32-bit word from byte input data.

Chapter 9

Comparison of the SW and HW Approaches

After the analysis provided in Chapter 8 we move on to the measurements of the accelerator's effectiveness on the real hardware. To get a meaningful results of the acceleration, we need a software implementation of the SHA-3 function. Having both hardware (accelerator) and software approaches, we can compare them and provide the conclusion for the accelerator presented in this work. Both implementations should be tested under the same conditions, such as the same SoC design, same processor and the same data set.

This chapter provides the results of measurements of both SHA-3 executions: with and without accelerator. To measure the execution time without accelerator, we needed a software implementation of SHA-3. For this purpose we have chosen the `tiny_sha3` implementation by Markku-Juhani O. Saarinen [30]. This implementation was chosen for the following reasons:

- As the title of the implementation states, this is a small and readable implementation
- The SHA-3 function is implemented using only two files: one source and one header
- Small implementation does not only mean few lines of code, but also that it can easily fit onto the embedded processor.
- Author of the implementation also provides a user guide on how the function should be used and some useful functions for testing, such as `test_readhex` which converts the `char` array into `uint8_t` array. This function has later been used in our measurements.

After the suitable software implementation has been chosen and we ensured that our firmware function works as expected, we proceeded to comparison of these two functions. All the measurements were provided using the following algorithm:

Algorithm 4: Measurements flow

```
for i = 0; i < 4; i = i + 1 do
    testdata = "";
    set digest_length according to i;
    digest = "";
    for j = 0; j < 255; j = j + 1 do
        message = test_readhex(testdata);
        message_length = length(message);
        timer.start();
        sha3_hw(message, message_length, digest, digest_length);
        timer.stop();
        timer.start();
        sha3_sw(message, message_length, digest, digest_length);
        timer.stop();
        compare if the results are the same;
        print hw and sw results;
        testdata = testdata + "12";
    end
end
```

Algorithm 4 gives us an overview of how the measurements were conducted. We can see that there is an outside `for` loop, that goes over all four SHA-3 hash functions. Inside is the second `for` loop that is intended to create a data set and call both firmware and software functions. The execution time in terms of clock cycles is measured for both functions, the results are compared and printed to the PC console.

9.1 FPGA Development Board

We have chosen the Digilent Nexys Video FPGA Board [18]. The development board contains the Artix-7 FPGA `xc7a200t-1sbg484c` from Xilinx.

Beside the huge number of logic gates, there are some other advantages of using this board, such as the presence of the USB/UART bridge. Subsection 7.5.4 describes the AXI UART IP core that allows us to use the UART protocol with the MicroBlaze processor. To later connect the output of the AXI UART IP core (rx and tx signals) to the PC, we needed the USB/UART bridge that would allow to connect the PC via the micro-USB cabel. Figure 9.1 illustrates how the bridge is connected on the board and which pins of the FPGA chip are used for the UART. The V18 and AA19 pins are later mapped to the rx and tx signals of our design in the constraints file.

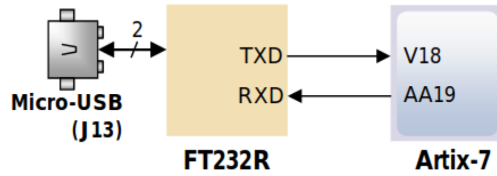


Figure 9.1: USB/UART bridge [18]

By connecting the micro-USB cabel to connector J13 of the Nexys development board we can monitor how the program is executed on the processor and get the results of the measurements using `xil_printf()` function.

The Nexys development board includes an on-board 100MHz oscillator that provides a clock input to the FPGA chip. As has been said in Subsection 7.1.1, the Clocking Wizard IP core can later be used to create clock signals with different frequencies, which will be created using the 100MHz input clock signal. The pin connected to the oscillator (R4 pin) is mapped into our design using the constraints file. The reset signal of our design is connected to the CPU reset button of the Nexys development board.

9.2 Data Set for Measurements

Our aim was to test the effectiveness of the accelerator on the different messages. For this purpose we had to create a suitable data set that will cover all the possible cases. As has been said in the beginning of this chapter, we conducted the tests using the structure shown in Algorithm 4. First `for` loop allows us to test all the SHA-3 functions and the second `for` loop is used to generate the input message for firmware and software functions.

The input message to both functions starts from zero length and continuous until the message length is not equal to 255. We did not conduct the test with larger input message, because 255 is enough to see the relation between SHA-3 execution with and without accelerator. With every iteration of the second `for` loop the length of the message increases, which can later give us an overview how the acceleration changes with increasing message length. Every `for` loop iteration the same two chars "12" were added to the message and resulted in new message, which always had the structure as "1212...12". Using `for` loop and the same appendix is an easy-to-implement approach.

The fact that input message is periodical and always consists of "12" sequence does not have any impact, because SHA-3 function does not distinguish between the messages of the same length, so it does not matter, if the message is "12" or "f3" or anything else. Analogously, our accelerator only takes the sequence of bits and also does not see the difference between two messages of the same length.

9.3 Measurements Results

Now that we have described the FPGA board and the data set we can move on to the results of the measurements. The amount of clock cycles required to execute the firmware and software function was measured using the AXI Timer described in Subsection 7.5.3. As has been described in Algorithm 4 we start the timer right before the function begins and stop the timer after the function finishes its execution. To provide the exact results, we measured how many clock cycles takes the configuration of timer, meaning how many clock cycles should we subtract from every measurement result in order to get only the clock cycles required to execute the function. We measured it in such a way, that we started the timer and stopped it with subsequent reading of the result. The test gave us nine clock cycles.

Table 9.1: SHA3-224 results

Length	Blocks	HW	SW	Acceleration
0	1	442	320256	724
21	1	630	320576	508
42	1	674	320891	476
63	1	692	321206	464
71	1	692	321326	464
72	1	693	321341	463
84	1	688	321521	467
103	1	718	321806	448
104	1	719	321821	447
105	1	729	321836	441
126	1	750	322151	429
135	1	752	322286	428
136	1	736	322301	437
143	1	753	322406	428
144	2	891	642010	720
147	2	908	642058	707
168	2	928	642373	692
189	2	954	642688	673
207	2	974	642958	660
208	2	960	642973	669
210	2	975	643003	659
215	2	966	643078	665
216	2	967	643093	665
231	2	979	643018	657
252	2	996	643633	646

In this section we provide four different tables with the measurements results. Each table corresponds to one of the SHA-3 functions.

In Section 9.2 we said that the message length was incremented by one, but in tables from this section we see that the length of the message almost always increases by 21. We did so because leaving all 255 measured values would result in very large table, so we selected only some values from a 0 to 255 range. Incremental number 21 was chosen on purpose and not randomly. Using an odd number we get all the cases for **m** and **v** variables shown in Figure 8.1.

Table 9.2: SHA3-256 results

Length	**Blocks**	**HW**	**SW**	**Acceleration**
0	1	461	320300	694
21	1	649	320620	494
42	1	678	320935	473
63	1	696	321250	461
71	1	711	321370	451
72	1	697	321385	461
84	1	707	321565	454
103	1	737	321850	436
104	1	723	321865	445
105	1	733	321880	439
126	1	754	322195	427
135	1	751	322330	429
136	2	904	641934	710
143	2	924	642042	694
144	2	910	642057	705
147	2	927	642102	692
168	2	947	642417	678
189	2	959	642732	670
207	2	979	643002	656
208	2	965	643017	666
210	2	980	643047	656
215	2	985	643122	652
216	2	986	643137	652
231	2	998	643362	644
252	2	1001	643677	643

The first column of Tables 9.1, 9.2, 9.3 and 9.4 shows the message length displayed in bytes. The second column tells us how many message blocks are formed after the input to the function is padded to the required length. The amount of message blocks depends on the version of SHA-3 function and can be calculated as $(\mathbf{M}/\mathbf{r}) + 1$, where $\mathbf{M}$ is the message's length and $\mathbf{r}$ is the rate size of the appropriate SHA-3 function. This column is included to clarify why there are leap changes in the computation time.

The third and fourth columns provide the amount of clock cycles which were required for the SHA-3 function be executed with and without accelerator respectively. These columns are also illustrated with the help of two Figures 9.2 and 9.3 which will be described later in this section. Each figure represents the graph and refers either to HW or SW approach and both results are not shown in the same graph due to the different range of measured values.

Table 9.3: SHA3-384 results

Length	**Blocks**	**HW**	**SW**	**Acceleration**
0	1	444	320476	721
21	1	632	320796	507
42	1	661	321111	485
63	1	679	321426	473
71	1	694	321546	463
72	1	680	321561	472
84	1	690	321741	466
103	1	700	322026	460
104	2	831	641630	772
105	2	841	641648	762
126	2	865	641963	742
135	2	881	642098	728
136	2	882	642113	728
143	2	888	642218	723
144	2	874	642233	734
147	2	891	642278	720
168	2	908	642593	707
189	2	920	642908	698
207	2	920	643178	699
208	3	1072	962782	898
210	3	1087	962815	885
215	3	1092	962890	881
216	3	1078	962905	893
231	3	1108	963130	869
252	3	1110	963445	867

The fifth column of the tables shows the acceleration of SHA-3 function computation. We defined the acceleration as the relation between the amount of clock cycles required to execute the SHA-3 function without accelerator to the amount of clock cycles required for SHA-3 function with accelerator. Simply said, the numbers in the fifth column represent the ratio between the fourth and the third columns of the tables. The acceleration is different for every SHA-3 version and also varies with the length message. The reasons for that are given in Section 9.4 together with Figure 9.4 that graphically illustrates the acceleration dependence.

As can be seen in every table, the message length is not always incremented by 21, but sometimes there is an increase by one. The reason is that we wanted to capture the changes in the computational speed at the moment when padded messages begins to consist of one more block than before. For example, for SHA3-224, shown in Table 9.1, we see the leap change of the required clock cycles appeared at 144 message length. The second column helps to clarify why the leap change occurred and indicates that starting from 144 the message is composed of two blocks.

Table 9.4: SHA3-512 results

Length	**Blocks**	**HW**	**SW**	**Acceleration**
0	1	437	320652	733
21	1	625	320972	513
42	1	654	321287	491
63	1	672	321602	478
71	1	667	321722	482
72	2	776	641326	826
84	2	785	641509	817
103	2	818	641794	784
104	2	819	641809	783
105	2	829	641824	774
126	2	836	642139	768
135	2	852	642274	753
136	2	836	642289	768
143	2	839	642394	765
144	3	969	961998	992
147	3	986	962046	975
168	3	991	962361	971
189	3	1017	962676	946
207	3	1037	962946	928
208	3	1021	962961	943
210	3	1034	962991	931
215	3	1024	963066	940
216	4	1155	1282670	1110
231	4	1181	1282898	1086
252	4	1187	1283213	1081

The values right before and after the changes in block amount were taken not only when the message composes of two blocks instead of one, but at every increase in the amount of blocks for the message. This is well illustrated in both Figures 9.2 and 9.3 for SHA3-512 version, where in the range from 0 to 255, message is composed of 1, 2, 3 and even 4 blocks.

Using the first and the third columns of the tables from this chapter we created a graph illustrated in Figure 9.2. The X axis of the graph represents the message length in bytes and the Y axis represents the amount of clock cycles required to execute the appropriate SHA-3

function. We can see how the amount of clock cycles for a zero-length message varies from non-zero length message. This is due to the fact, that zero-length message does require only simple padding and does not contain any shifting described in Section 8.3. The difference between non-zero-length and zero-length message case nicely illustrates how the padding and presence of additional `for` loops and `if` conditions influences the execution.

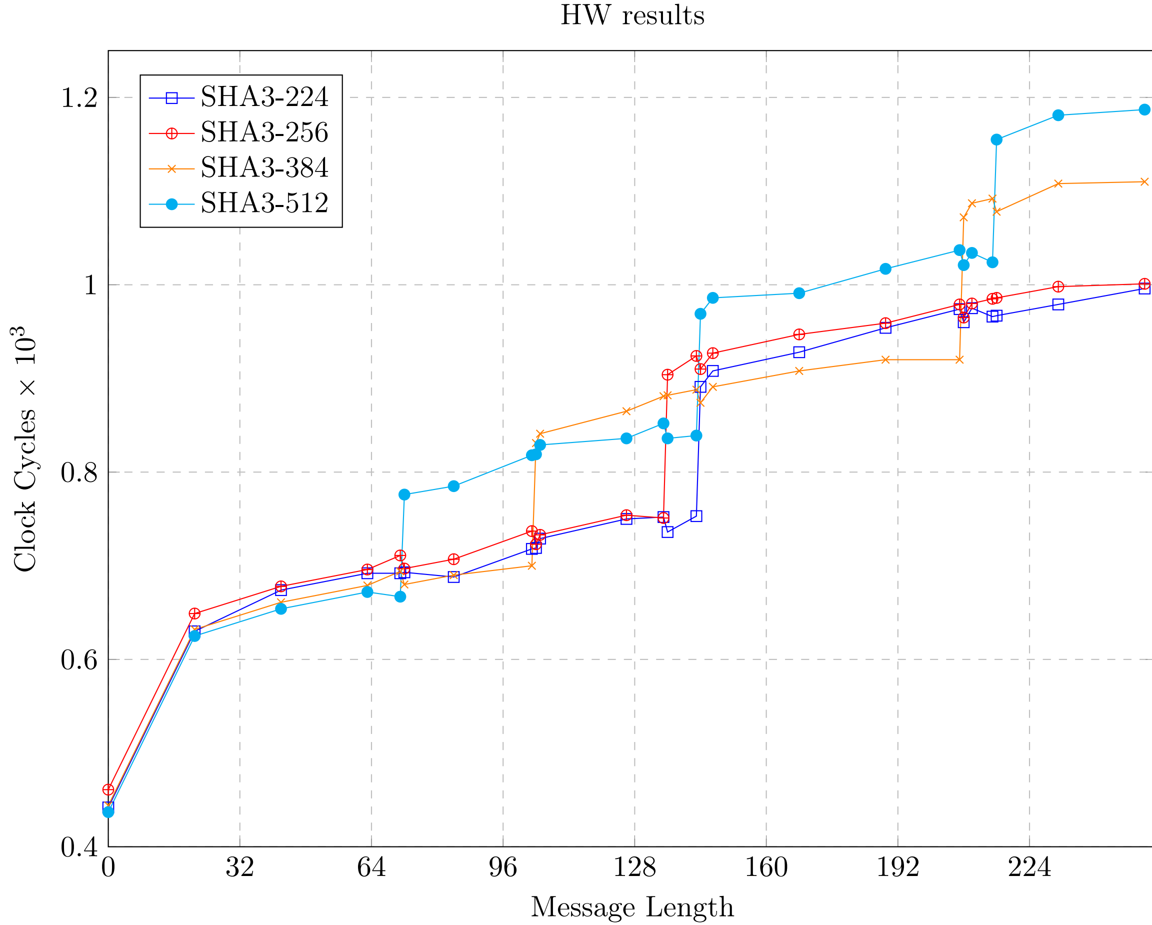


Figure 9.2: Graph illustrating SHA-3 computation with accelerator

We can see from Figure 9.2 that the execution time does grow with the increasing message length. As we described in Section 8.4, the time required to execute the LOAD, COMPUTE and EXPORT states is not influenced on the message length and is given only by the SHA-3 version. The clock cycles growth in Figure 9.2 shows us how the firmware function execution is affected by the length of the message. Sometimes we can see a decrease in clock cycles with increasing message length. This can be explained by the fact, that if we have **m** from Figure 8.1 equals to one, then we have to XOR this byte with `0x00000600`. If **m** equals two, then one byte should be shifted left by 8 and subsequently XORed. This byte shift increases the amount of clock cycles required to execute the firmware function. However, since the **m** is modulo four, after the message length increased from, for example 7 to 8, the **m** value is zero and no shifting is required meaning the amount of clock cycles will decrease from the

previous one.

Figure 9.3 represents the graph created from the values of the first and fourth columns of the tables from this chapter. Similarly to graph in Figure 9.2, the X axis represents the message length and Y axis represents the amount of clock cycles required to compute the digest of the appropriate SHA-3 function.

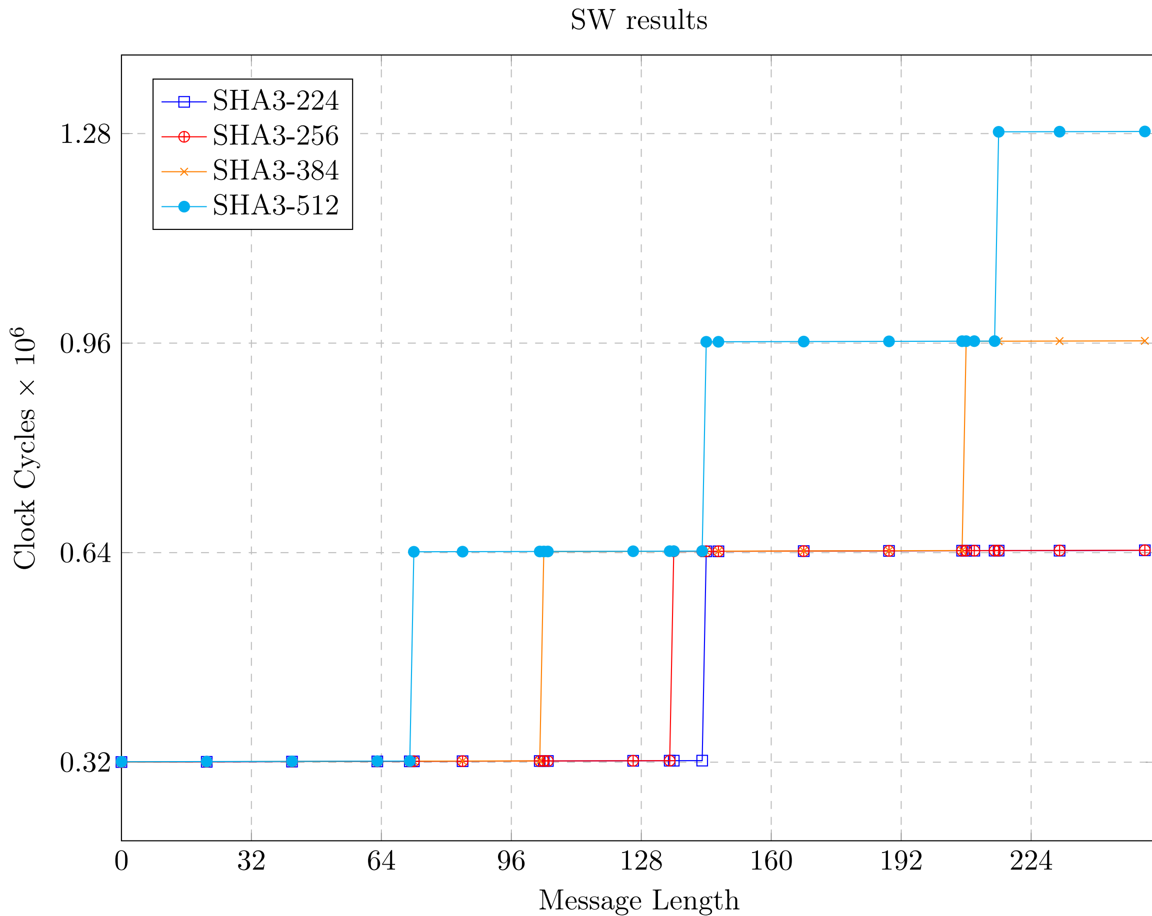


Figure 9.3: Graph illustrating SHA-3 computation without accelerator

At first sight it might seem, that the amount of clock cycles does not grow with the increasing message length (until the huge leap change), but this is only due to the fact, that the growing is very small regarding to the total number of clock cycles and is not visible in the graph.

9.4 Acceleration Dependence on the Message Length and SHA-3 Version

The most important data are located in the fifth columns of the tables in this chapter. As we said, the fifth column contains the ration between the approach without and with the

accelerator. In other words, it tells us how effective the accelerator is for a given SHA-3 version and message length. Data from each fifth column forms a graph illustrated in Figure 9.4. We have already explained, why the zero-length message is processed faster than other single-block messages, so the high effectiveness of the accelerator for a zero-length message does not cause the misunderstanding.

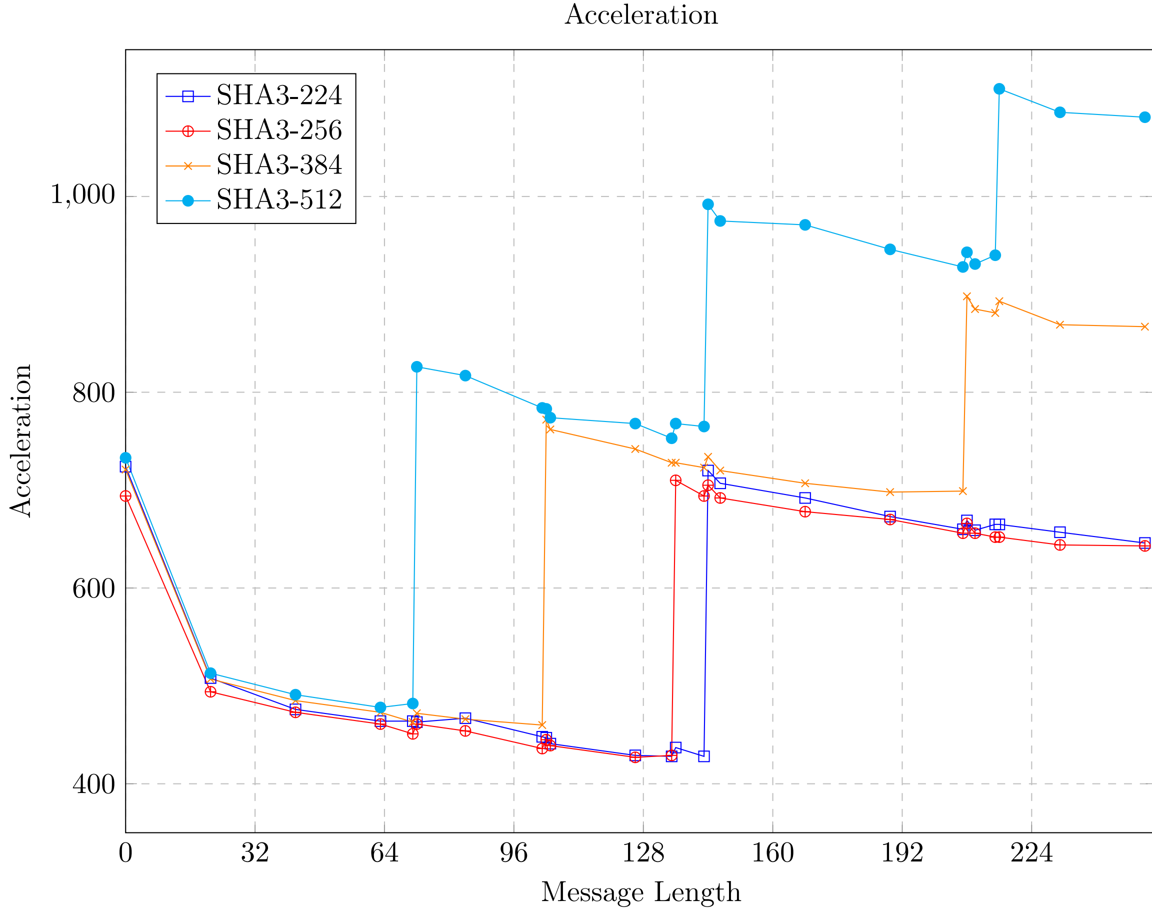


Figure 9.4: The acceleration dependence on the message length and SHA-3 version

With the increasing message length (within the same blocks amount that form the message) the effectiveness of the acceleration decreases. This is caused by the fact, that with increasing the length of the message by 21, the amount of clock cycles for the software approach changes only by 0.098%. This was calculated for SHA3-512 version and message lengths 42 and 21. In contrast, the number of clock cycles for the approach with accelerator changes by 4.64%, which explains why we see the decreasing line in the accelerator's effectiveness in Figure 9.4.

The accelerator shows its real advantages when used for the messages composed of multiple blocks. We can see in Figure 9.4 and Table 9.4 that for SHA3-512 the acceleration minimum and maximum values are 478 and 1110. Another important and interesting observation is

that with increasing amount of blocks the acceleration does not grow proportionally. We will describe it on the SHA3-512 version. When the amount of blocks in the second column of Table 9.4 changes from 1 to 2 the acceleration changes by 344. The the amount of blocks changes from 2 to 3, the acceleration changes by 227; when blocks change from 3 to 4, the acceleration changes by 170. The decreasing acceleration growth can be explained by the fact, that data transferring and other firmware operations (both described in Sections 8.1 and 8.3 respectively) influence the overall effectiveness of the accelerator more and more. The same can be observed for the SHA3-384 version and explained in the same manner. Other two versions of SHA-3 function will demonstrate the same dependence, however it cannot be seen in the provided graphs, because we conducted the tests only for 0-255 message length.

Chapter 10

Conclusion

The main goal of this work was to propose an implementation of the KECCAK algorithm in FPGA and verify it using an SoC design. The implementation is described in Chapter 4 and the SoC design used for verification is described in Chapter 7. During the development process we used the advantages of the programmable logic to propose the effective and fast implementation. As a result, we get the functional component that can effectively execute the SHA-3 hash function. This component, or accelerator, can be connected to the processor via one interface for communication and another interface for control. The processor sends the data for the SHA-3 function, starts the accelerator and receives the provided result. The accelerator is packaged as an IP core and is ready to be used in any design that contains the desired processor.

Comparison results of the software and hardware implementations are given in Chapter 9. Graph 9.4 shows the ratio between the execution time of software and hardware implementations. More precise values can be found in Tables 9.1, 9.2, 9.3 and 9.4. We denoted this ratio as the acceleration and in Sections 9.3, 9.4 we describe in detail the reasons why the acceleration depends on the input message's length and the version of the SHA-3 function. From the graphs provided in Section 9.3 we can see that for the given data set the minimum value for the acceleration is 427 and the maximum value is 1110. According to these values we can conclude, that the acceleration makes sense and the designed accelerator can execute the KECCAK algorithm very effectively.

During this work we faced several difficulties. Most of the difficulties were related to writing an optimal firmware function for the accelerator. As we saw in Section 8.4, the firmware has about 40-60% impact on the total SHA-3 execution time and that is why we devoted a lot of time to writing the firmware function and a whole chapter to analyze it. Other difficulties include writing the effective control unit for the computational core, solving the clock domain crossing problem and selection of a suitable communication interface for which we had to create a good validation environment in order to provide the correct measurements results.

The future work would consist of optimization of accelerator control tasks. For this purpose, the special unit can be designed which would control the operation of the accelerator and handle all the data transfers between the processor and the accelerator.

Appendix A
Contents of the attached CD

/

- Diploma-pdf .... Folder with PDF document
  - Diploma-thesis.pdf .... Diploma PDF document
- SHA3-accelerator .... Verilog files for the accelerator
  - keccak-axis .... Verilog files for keccak-axis
    - RTL .... RTL files for keccak-axis block
    - TESTBENCH .... Testbench files for keccak-axis block
  - keccak-core .... Verilog files for keccak-core
    - RTL .... RTL files for keccak-core block
    - TESTBENCH .... Testbench files for keccak-core block
  - keccak-fifo .... Verilog files for keccak-fifo
    - RTL .... RTL files for keccak-fifo block
    - TESTBENCH .... Testbench files for keccak-fifo block
  - s3 .... Verilog files for S3 sources
    - global-cell .... Folder with global files
      - RTL .... RTL S3 files
  - memory-files .... Folder with memory files for testbenches
- Python .... Folder with python project to generate keccak-core.v file
- Vivado .... Folder with vivado projects
  - SHA3-accelerator .... Folder with Vivado project for SHA-3 accelerator
  - SoC-interface-selection ..... Folder with Vivado project for interface-selection SoC design
  - SoC-sw-hw-comparison .... Folder with Vivado project for sw-hw-comparison SoC design

Bibliography

[1] Ronald Rivest. “The MD5 Message-Digest Algorithm”. *MIT Laboratory for Computer Science and RSA Data Security, Inc.* 1992. URL: https://tools.ietf.org/html/rfc1321.

[2] William Stallings. *Cryptography and Network Security Principles and Practices, Fourth Edition*. Prentice Hall, 2005.

[3] ARM. “AMBA APB Protocol”, 2010. URL: https://static.docs.arm.com/ihi0024/c/IHI0024C_amba_apb_protocol_spec.pdf?_ga=2.102145668.467062489.1588411576-1017671257.1588411576.

[4] ARM. “AMBA4 AXI4-Stream Protocol v1.0”, 2010. URL: https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf.

[5] G. Bertoni J. Daemen M. Peeters G. V. Assche. “Cryptographic sponge functions”, 2011. URL: https://keccak.team/files/CSF-0.1.pdf.

[6] G. Bertoni J. Daemen M. Peeters G. V. Assche. “The KECCAK reference”, 2011. URL: https://keccak.team/files/Keccak-reference-3.0.pdf.

[7] Xilinx. “AXI Reference Guide”, 2011. URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[8] Quynh Dang. “Recommendation for Applications Using Approved Hash Algorithms”. *National Institute of Standards and Technology*, 2012. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf.

[9] G. Bertoni J. Daemen M. Peeters G. V. Assche R.V. Keer. “KECCAK implementation overview”, 2012. URL: https://keccak.team/files/Keccak-implementation-3.2.pdf.

[10] “Secure Hash Standard (SHS)”. *National Institute of Standards and Technology*, 2012. URL: https://csrc.nist.gov/csrc/media/publications/fips/180/4/final/documents/fips180-4-draft-aug2014.pdf.

[11] Bruce Morton and Clayton Smith. “Why We Need to Move to SHA-2”, 2014. URL: https://casecurity.org/2014/01/30/why-we-need-to-move-to-sha-2/.

[12] Federal Information Processing Standards Publication. “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”, 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[13] Xilinx. “AXI to APB Bridge v3.0”, 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_apb_bridge/v3_0/pg073-axi-apb-bridge.pdf.

[14] Xilinx. “Processor System Reset Module v5.0”, 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf.

[15] Xilinx. “AXI Block RAM (BRAM) Controller v4.0”, 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_0/pg078-axi-bram-ctrl.pdf.

[16] Xilinx. “AXI Timer v2.0”, 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf.

[17] Xilinx. “Local Memory Bus (LMB) v3.0”, 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/lmb_v10/v3_0/pg113-lmb-v10.pdf.

[18] Digilent. “Nexys Video FPGA Board Reference Manual”, 2017. URL: https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-video/nexysvideo_rm.pdf.

[19] Xilinx. “AXI Interconnect v2.1”, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf.

[20] Xilinx. “AXI UART Lite v2.0”, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf.

[21] Xilinx. “Block Memory Generator v8.3”, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf.

[22] Xilinx. “Clocking Wizard v5.4”, 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_4/pg065-clk-wiz.pdf.

[23] Xilinx. “AXI Central Direct Memory Access v4.1”, 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf.

[24] Xilinx. “AXI4-Stream Infrastructure IP Suite v3.0”, 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf.

[25] Xilinx. “LMB BRAM Interface Controller v4.0”, 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/lmb_bram_if_cntlr/v4_0/pg112-lmb-bram-if-cntlr.pdf.

[26] Xilinx. “MicroBlaze Processor Reference Guide”, 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug984-vivado-microblaze-ref.pdf.

[27] Defuse Security. “Salted Password Hashing - Doing it Right”, Last modified: June, 2019. URL: https://crackstation.net/hashing-security.htm.

[28] Magnus Daum and Stefan Lucks. “Hash Collisions (The Poisoned Message Attack). "The Story of Alice and her Boss"”. URL: https://web.archive.org/web/20100327141611/http://th.informatik.uni-mannheim.de/people/lucks/HashCollisions/.

[29] Keccak *in VHDL*. URL: https://keccak.team/hardware.html.

[30] Markku-Juhani O. Saarinen. *Very small, readable implementation of the SHA3 hash function*. URL: https://github.com/mjosaarinen/tiny_sha3.

[31] National Institute of Standards and Technology. *Examples with Intermediate Values*. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values.

[32] *Third-party implementations*. URL: https://keccak.team/software.html.

[33] Chalermpong Worawannotai and Isabelle Stanton. “A Tutorial on Slide Attacks”. URL: http://docplayer.net/45146691-A-tutorial-on-slide-attacks.html.

[34] Xilinx. *Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL Designs*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_hdl.pdf.