# Putting Together What Fits Together - GrÆStl

A Combined Hardware Architecture for AES and Grøstl

Markus Pelnar, BSc.
m.pelnar@student.tugraz.at
pelnarm@student.ethz.ch

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Integrated Systems Laboratory (IIS)
Swiss Federal Institute of Technology
Gloriastrasse 35
CH-8092 Zurich, Switzerland

Master Thesis

Supervisors: Dipl. Ing. Michael Mühlberghuber, ETH Zürich
Dipl. Ing. Beat Muheim, ETH Zürich
Dr. Michael Hutter, TU Graz
Assessor: Dr. Karl-Christian Posch, TU Graz
Dr. Norbert Felber, ETH Zürich

May, 2012

Graz University of Technology

**Senat**

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

## EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

## STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

# Acknowledgements

First of all I would like to thank my supervisors Michael Hutter from Graz University of Technology (TUG), Michael Mühlberghuber and Beat Muheim from Swiss Federal Institute of Technology (SFIT) Zurich and all members of the Microelectronics Design Center from the Department of Information Technology and Electrical Engineering (D-ITET) for their outstanding support during the last months I was working on this thesis and the possibility to gain experience on taping out a functional chip. I would particularly like to thank Michael Hutter for the excellent advices on writing scientific papers. A special thanks also to Frank K. Gürkaynak for giving me tips for the future and sharing pieces of his knowledge regarding Application-Specific Integrated Circuit (ASIC) design. I am sure some tips will help me out in the future. Christian Pendl a former tutor of mine and from the master studies on a steady companion receives a special thanks too. Working with him on projects was always a pleasure whilst (at the same time) a challenge as targets were always set very ambitious. Two highlights from this collaboration must be highlighted. First, a scientific paper which got accepted and also presented at the RFIDSec 2011. Second, going together abroad to write the master thesis at the SFIT Zürich. Last but not least the biggest thanks goes to my family and my girlfriend for the patience and support during the whole studies, especially during the time of being abroad to write the master thesis. A special thanks to my big brother who was and will always be a shining example. I know it was not always easy to understand my decisions and acting but they got always behind me. Because of this I want to dedicate this thesis to my family.

# Abstract

This thesis comprises of two constructive parts. First, GrÆStl, a combined hardware architecture for the Advanced Encryption Standard (AES) and Grøstl, developed for low-resource devices and aiming for high flexibility by targeting both Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) platforms. The former is the most important encryption specification known so far, which was announced by the National Institute of Standards and Technology (NIST) in November 2001. Grøstl on the other hand is one of the final candidates of the cryptographic hash-algorithm competition initiated by NIST. The winner will augment the Federal Information Processing Standard (FIPS) 180-3, Secure Hash Standard (SHA) and become therefore maybe as important as AES. Combining these two primitives looks promising for the future as the integration of AES-128 into Grøstl-224 can be achieved with an area overhead of only 10 % when aiming for ASIC platforms and furthermore might be used for authenticated encryption in resource-constrained environments. Using a 0.18 µm fabrication process while targeting a maximum frequency of 125 MHz delivered a complexity for GrÆStl of around 17.1 kGE after the backend design. The stand-alone versions for AES-128 and Grøstl-224 from which GrÆStl was build upon require 15.5 kGE and 5 kGE , respectively. All designs were realized with standard cells only and no technology or platform-dependent components such as Random Access Memory (RAM) macros, Digital Signal Processors (DSPs), or Block RAMs were used. On FPGA platforms even better results can be reported as the stand-alone versions for both AES and Grøstl outperform all existing work by a factor of 4.8 and about 1.6. This impressive results were achieved through exploiting the Shift Register Logic (SRL) functionality of Xlinix FPGAs. AES-128 and Grøstl-224 occupy therefore only 442 and 488 slices on a Xilinx Spartan-3. The combined version requires 956 slices. With respect to timing the results on the two platforms are equal. An AES encryption can be fulfilled within 652 clock cycles. Decryption on the other hand requires by default 1,269 clock cylces whereas a reduction to 955 clock cycles occurs when an AES operation with the same master key has been applied immediately before. Grøstl requires 3,061 clock cycles to process one message block.

Second, an FPGA system encompassing a programmable microcontroller, namely the openMSP430 and the dedicated hardware components from the first part designed to evaluate the impact of the highly flexible four-phase handshaking protocol and to enable a fair hardware/software comparison. The FPGA system was evaluated by increasing the frequency of the dedicated hardware components from 5 to 20 MHz while leaving the microcontroller constantly at 5 MHz , which resulted in a speedup reduction from 4 to only 1.3 up to 1.6. On the other hand, outsourcing of software routines on dedicated hardware accelerates the computation by a factor of 50 up to 360.

**Keywords:** AES, Grøstl, GrÆStl, SRL-16, ASIC, FPGA

# Kurzfassung

Diese Abschlussarbeit besteht aus 2 aufeinander aufbauenden Teilen. Erstens, GrÆStl, eine kombinierte Hardware-Architektur für den Advanced Encryption Standard (AES) und Grøstl, die für Geräte mit stark beschränkten Ressourcen entwickelt wurde und weiters auf eine hohe Flexibilität bezüglich dem Einsatz auf Application-Specific Integrated Circuit (ASIC) und Field Programmable Gate Array (FPGA) Plattformen abzielt. Ersteres ist die bis jetzt wichtigste Verschlüsselungsspezifikation welche im November 2001 vom National Institute of Standards and Technology (NIST) angekündigt wurde. Grøstl ist auf der anderen Seite einer der Finalisten eines vom NIST gestarteten Wettbewerbes für kryptographische Hash-Algorithmen, wo der Gewinner den Federal Information Processing Standard (FIPS) 180-3, auch Secure Hash Standard (SHA) genannt, erweitern und deswegen womöglich auch gleichbedeutend wie AES wird. Eine Kombination dieser beiden Primitive schaut für die Zukunft vielversprechend aus, da eine Integration von AES-128 in Grøstl-224 auf ASIC Plattformen mit einem zusätzlichen Flächenmehraufwand von nur 10 % möglich ist. Des weiteren könnte dadurch eine authentifizierte Verschlüsselung in Umgebungen mit beschränkten Ressourcen ermöglicht werden. Bei einem 0,18 µm Fabrikationsprozess und einer Auslegung auf maximal 125 MHz weißt GrÆStl nach dem Backend-Design eine Komplexität von rund 17,1 kGE auf. Die autarken Versionen für AES-128 und Grøstl-224, auf welchen GrÆStl aufbaut, benötigen 15,5 kGE beziehungsweise 5 kGE . Alle Designs wurden nur mit Standardzellen und ohne die Benutzung von Technologie oder plattformabhängigen Komponenten wie zum Beispiel Random Access Memory (RAM) Makros, Digital Signal Processors (DSPs) oder Block-RAMs realisiert. Auf FPGA-Plattformen können sogar noch bessere Resultate erzielt werden, da die autarken Versionen für AES und Grøstl bestehende Arbeiten um den Faktor 4,8 bis 1,6 übertreffen. Diese eindrucksvollen Resultate konnten durch das Ausnutzen der Shift Register Logic (SRL) Funktionalität von Xlinx FPGAs erreicht werden. AES-128 und Grøstl-224 verbrauchen hierbei auf einem Xlinx Spartan-3 nur 442 beziehungsweise 488 Slices. Die kombinierte Version benötigt 956 Slices. Bezüglich des zeitlichen Verhaltens sind die Resultate auf den beiden Plattformen identisch. Eine AES-Verschlüsselung kann mit 652 Taktzyklen erfüllt werden. Eine Entschlüsselung benötigt standardmäßig 1.269 Taktzyklen, wobei eine Reduktion auf 955 Taktyzklen erfolgt, sofern eine AES-Operation mit dem selben Hauptschlüssel unmittelbar davor durchgeführt wurde. Grøstl benötigt 3.061 Taktzyklen um einen Nachrichten-Block zu verarbeiten.

Zweitens, ein FPGA-System welches einen programmierbaren Mikrocontroller, den openMSP430, und die dedizierte Hardware aus dem ersten Teil enthält. Es wurde dazu entwickelt um die Auswirkung des höchst flexiblen 4-Phasen Handshake-Protokolls aufzuzeigen und einen fairen Hardware/Software vergleich zu ermöglichen. Das FPGA-System wurde durch das Erhöhen der Frequenz der dedizierten Hardwarekomponenten von 5 auf 20 MHz evaluiert, während der Mikrocontroller konstant mit 5 MHz betrieben wurde und lieferte als Resultat eine Beschleunigungsreduktion von 4 auf nur 1,3 bis 1,6. Auf der anderen Seite konnte eine Beschleunigung durch das Auslagern der Software-Routinen auf die dedizierte Hardware im Bereich von 50 bis 360 erreicht werden.

**Stichwörter:** AES, Grøstl, GrÆStl, SRL-16, ASIC, FPGA

# Contents

**1 Introduction 1**

**2 Selected Chapters of Cryptography 3**

2.1 Symmetric-Key vs. Public-Key Cryptography . . . 3

2.1.1 Unsecured Communication and Its Risks . . . 3

2.1.2 Requirements for Secure Communication . . . 4

2.1.3 Symmetric-Key Cryptography . . . 5

2.1.4 Public-Key Cryptography . . . 5

2.2 Block Ciphers . . . 6

2.2.1 Electronic Code Book (ECB) Mode . . . 7

2.2.2 Cipher Block Chaining (CBC) Mode . . . 8

2.3 Stream Ciphers . . . 10

2.3.1 Cipher Feedback (CFB) Mode . . . 10

2.3.2 Output Feedback (OFB) Mode . . . 10

2.3.3 Counter (CRT) Mode . . . 11

2.4 Message Authentication Codes (MAC) . . . 12

2.5 Hash Functions . . . 14

2.6 Hash-Based Message Authentication Code (HMAC) . . . 16

**3 Rijndael - Winner of the NIST AES Competition 17**

3.1 Round Transformation . . . 18

3.1.1 SubBytes/InvSubBytes . . . 19

3.1.2 ShiftRows/InvShiftRows . . . 19

3.1.3 MixColumns/InvMixColumns . . . 20

3.1.4 AddRoundKey . . . 20

3.2 Round-Key Generation . . . 21

**4 Grøstl - A NIST SHA-3 Competition Finalist 23**

4.1 Compression Function . . . 24

4.2 Output Transformation . . . 24

4.3 Permutations $P$ and $Q$ in Detail . . . 25

4.3.1 AddRoundConstant . . . 25

4.3.2 SubBytes . . . 26

4.3.3 ShiftBytes . . . 26

4.3.4 MixBytes . . . 26

**5 Related Work 29**

**6 Asynchronous Interfaces** **33**
6.1 Synchronous Digital Circuits . . . 33
6.2 Asynchronous Interfaces for Synchronous Digital Circuits . . . 36
6.2.1 Inconsistent Data . . . 36
6.2.2 Measures Against Inconsistent Data . . . 37

**7 Basics on Xilinx FPGAs** **41**
7.1 General Structure . . . 41
7.1.1 Configurable Logic Block (CLB) . . . 42
7.1.2 Slice . . . 42
7.2 Memory Variants for Spartan-3 FPGAs . . . 43
7.2.1 Block RAM (BRAM) . . . 43
7.2.2 Distributed RAM (LUT RAM) . . . 44
7.2.3 Shift-Register Logic (SRL16) . . . 45

**8 GrÆStl - A Combined AES/Grøstl HW Architecture** **49**
8.1 On the Search for Optimal Resource Management . . . 49
8.2 Hardware Architecture . . . 55
8.2.1 Top Layer . . . 55
8.2.2 Common Datapath . . . 58
8.3 Results . . . 62
8.3.1 Application-Specific Integrated Circuit (ASIC) . . . 62
8.3.2 Field Programmable Gate Array (FPGA) . . . 66
8.4 Comparison with Related Work . . . 68
8.4.1 Comparison of ASIC Results . . . 68
8.4.2 Comparison of FPGA Results . . . 69

**9 FPGA System for HW/SW Evaluation** **71**
9.1 General System Requirements . . . 71
9.2 System Architecture . . . 72
9.2.1 Xilinx Spartan 3 . . . 72
9.2.2 openMSP430 . . . 73
9.2.3 Cryptographic Modules . . . 76
9.2.4 Cryptographic Library . . . 80
9.3 Results . . . 83
9.3.1 Influence of External Data Transfer on Overall Timing . . . 83
9.3.2 Software/Hardware Evaluation . . . 85

**10 Conclusions** **87**

**A Definitions** **91**
A.1 Abbreviations . . . 91

**B Chameleon - ASIC** **93**
B.1 General Features . . . 93
B.2 Pinout . . . 95
B.3 Pad Description . . . 96
B.4 Interface Description . . . 97

**C Chameleon - FPGA** **99**
C.1 Xilinx Spartan-3 and Spartan-6 Results . . . . . . . . . . . . . . . . . . . . . . 99
C.2 Xilinx Virtex-5 and Virtex-6 Results . . . . . . . . . . . . . . . . . . . . . . . . 99

**D FPGA System - AddOn** **101**
D.1 Basic Linker Script for openMSP430 MCUs . . . . . . . . . . . . . . . . . . . . 101
D.2 Installation Script for MSPGCC Toolchain . . . . . . . . . . . . . . . . . . . . . 105
D.3 Test Program for Verifying Cryptographic Modules . . . . . . . . . . . . . . . 114

**E Original Assignment** **123**

**Bibliography** **131**

# Chapter 1

# Introduction

Among the most commonly used cryptographic primitives in classical communication protocols are block ciphers and hash functions. The Advanced Encryption Standard (AES) [51] is by far the most widely spread block cipher since its standardization in 2001 by the National Institute of Standards and Technology (NIST). Grøstl [15] on the other hand is one of the final round candidates of the Secure Hash-Algorithm (SHA) competition [50], which will announce its winner in late of 2012. Therefore, AES is already and Grøstl could become an algorithm widely used to achieve data confidentiality, integrity and authenticity. In short, these three data properties can be fulfilled through authenticated encryption which plays a major role in the world of communication systems. In combination with the trend for ultra-mobile devices—containing evermore confidential information—equipped with an arbitrary communication interface there is the need for compact and power-efficient implementations. Due to the fact that AES and Grøstl feature several similarities such as a common S-box or similar diffusion layers an integration into one module looks promising for the future. In addition to the cost savings regarding area occupation, authenticated encryption could be fulfilled within one step. The consequential drawback that a parallel computation of an encryption/decryption and a hash computation is not supported anymore is mitigated by the circumstance that the most limiting factor in such ultra-mobile devices is area and low energy and not performance.

This work has been split into two constructive parts. First, GrÆStl a hardware architecture combining the functionality of AES-128 and Grøstl-224 in one piece of silicon. The design aims for high flexibility supporting both Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) platforms without using technology-dependent components such as Random Access Memory (RAM) macros, Digital Signal Processors (DSPs), or Block RAMs. Various optimization techniques were exploited to reduce the area footprint, for example, by sharing registers and a common datapath. The ASIC version of the design with the name Chameleon has been fabricated using the 0.18 µm Complementary Metal Oxide Semiconductor (CMOS) process technology from UMC and represents the first taped-out version of a combined AES/Grøstl architecture in literature. It requires only 17.1 kGE —backend results including eight parallel scanchains and clock gating—in total and needs 652/1,269 clock cycles for AES encryption/decryption and 3,061 clock cycles for hashing. The stand-alone implementations of AES and Grøstl consume around 5 and 15.5 kGE . Due to the chosen design were all large register banks are based on shift registers, backend results differ in only about 3 % to the synthesis results. This nearly negligible overhead is reasonable as the storage elements are already connected together in a favourable manner. The small area requirements and also the low power consumption of about 20 µW at 100 kHz make the design applicable to resource-constrained devices. Porting the implementation on an FPGA was easy as no platform-dependent components were used. Due to a recommendation from Xilinx regarding the removal of the reset functionality from shift registers, c.f. [7], the design was altered in a suitable manner to ease the usage of the SRL-16 mode where the Look-Up Tables (LUTs) of slices can be reconfigured to form an 16-bit shift register. As the design perfectly fits this mode, it shows that the implementations outperform existing FPGA solutions in terms of low area. They require up to 79 % less resources on a Spartan-3 compared to existing implementations. In addition, it shows that for FPGA platforms it is recommended to prefer the single versions as they are in sum smaller as the shared one. The reason for this is the state matrix which had to be adapted in order to integrate the AES state as well as also the master key and the actual round key, respectively.

In the second part an FPGA system was designed to enable a hardware/software comparison of the stand-alone implementations and the combined implementation. For this a microcontroller was required which was taken from OpenCores, c.f. [16]. The chosen Microcontroller Unit (MCU) is named openMSP430—RAM and ROM are located externally and connected to the core over two buses—and is compatible to the MSP430 of Texas Instruments, c.f. [61]. It is supported by the port of the GCC toolchain for the Texas Instruments MSP430 family (MSPGCC) which is important for setting up the design flow. Additionally, an interface to the cryptographic modules described in part one was required which was realized over the peripheral bus and a piece of software. For verification of the system, two test ROMs including testvectors for AES-128 and Grøstl-224 were attached to the design. As different clock domains were used for the MCU and the cryptographic modules a four-phase handshaking was incorporated to handle the communication. The clock of the cryptographic modules was designed for 5 and 20 MHz , respectively and is switchable over software. The expected speedup due to the increase of the clock frequency was reduced from 4 to about 1.3 up to 1.6 depending on the operation applied. Comparing software implementations targeting a low-memory footprint with the low-area implementations from part one resulted in a speedup between 50 and 360 depending on the mode of operation and the clock frequency used for the cryptographic modules.

The remainder of this thesis is organized as follows. In Chapter 2, an overview of symmetric-key and public-key cryptography is given. Furthermore, basic cryptographic primitives used in classical cryptography like block ciphers, secure hash functions and message authentication codes are explained. Chapter 3 presents the NIST AES competition and describes in detail its winner, the Advanced Encryption Standard (AES) followed by the NIST SHA-3 competition in Chapter 4 with a detailed explanation of Grøstl, one of the five finalists. Afterwards in Chapter 5 related work targeting low-area ASIC and FPGA implementations for AES and Grøstl are summarized. As an FPGA system with different clock domains is required in the second part of the thesis problems with interfaces between two independent subsystems—clock signals not synchronized—are addressed in Chapter 6 before in Chapter 7 basics on Xilinx FPGAs are stated. In Chapter 8 details on the taped-out chip, named Chameleon are given. Results and a comparison with related work is given for the ASIC as well as for the version ported on various Xilinx FPGAs. Afterwards, Chapter 9 presents the developed FPGA system targeting a Xilinx Spartan-3 FPGA, which contains an MSP430 MCU and the previously developed cryptographic modules. Finally, in Chapter 10 a summary of the results combined with drawn conclusions is given.

# Chapter 2

# Selected Chapters of Cryptography

This chapter gives a short introduction into the field of cryptography, especially targeting symmetric-key cryptography and public-key cryptography. Furthermore, it explains in detail general cryptographic primitives like block ciphers, hash functions and (hash-based) message authentication codes.

## 2.1 Symmetric-Key vs. Public-Key Cryptography

Symmetric-key and public-key cryptography are both used in various fields of application. In order to understand their need this section first introduces the risks of an unsecured communication. Afterwards, a list of requirements for secured communication is given. These requirements on the one hand can be fulfilled partly by symmetric-key cryptography and on the other hand fully by public-key cryptography. The reason why still both schemes are used should be clear after the subsections targeting these primitives.

### 2.1.1 Unsecured Communication and Its Risks

Before going into details of symmetric-key cryptography and public-key cryptography one must understand the basic communication scenario all further explanations will be based on. This scenario is given in Figure 2.1. Alice and Bob represent the two parties aiming to exchange data with each other without getting eavesdropped by a third party named Eve. All data exchange is established over an unsecure channel which means that it is not fully under control of the communication partners and therefore at risk that an adversary gets access to it. State-of-the-art mobile phones featuring several interfaces like Wi-Fi, bluetooth, infrared and so on are the best example for an infinitely seeming connectivity. This convenient way of exchanging data features also a drawback as all communication paths are not fully under control of the user because no one can tell on which path data is transmitted from Alice to Bob and vice-versa. Therefore a smart adversary could modify one stage of the path to eavesdrop the communication without the knowledge of either Alice or Bob, modify parts of the data or even exchange the whole. Impersonating Alice or Bob is in addition possible. These serious threats in combination with keeping a message secret—data confidentiality—were the reason cryptography was developed for.

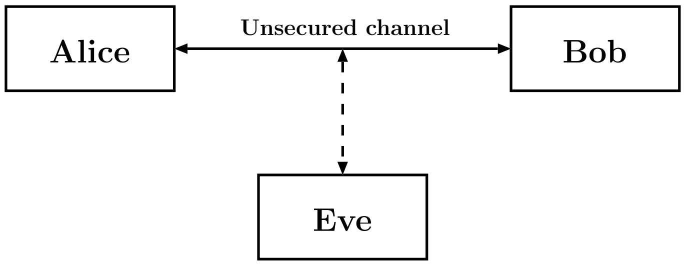


Figure 2.1: Basic communication model.

### 2.1.2 Requirements for Secure Communication

Through the mentioned attack scenarios the following fundamental objectives for a secure communication were revealed and stated by several authors, e.g., Hankerson et. al. [20]:

- **Confidentiality**
  Data should be only readable by parties to which access was granted to. Therefore, messages sent from Alice to Bob should not be readable by Eve.
- **Data integrity**
  Third parties like Eve should not be able to modify the message sent from Alice to Bob without getting detected by Bob.
- **Data origin authentication**
  Bob should be able to verify that data purportedly coming from Alice was indeed originated from Alice.
- **Entity authentication**
  Alice and Bob should be convinced of the identity of each other.
- **Non-repudiation**
  If Alice sends a message to Bob, Alice cannot deny of having sent this message and Bob can prove this to others. Furthermore, Bob is able to recognize if the message was originated from Alice.

In the early days of cryptography secured communication was all done with symmetric-key cryptography as it was the only scheme known so far. As the technical aids and the mathematical knowledge were not as sophisticated as today, the efforts were all on the side of finding more efficient and more secure symmetric-key primitives than as to find an alternative to it. With the technical improvement and the better mathematical background, the key-distribution problem and the unreachable **non-repudiation** were not negligible any more. Therefore, industry and cryptographic community focused more on finding an alternative which solves all the shortcomings of symmetric-key cryptography than improving the efficiency of it. In 1975, such a variant named public-key cryptography was introduced by Diffie, Hellman and Merkle, c.f. [38]. Today, around 37 years later, still both schemes are widely spread which seems awkward when remembering the mentioned

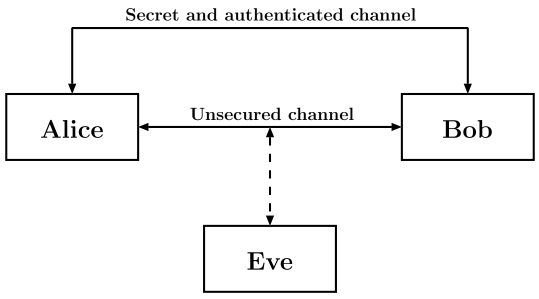


Figure 2.2: Symmetric-key communication model.

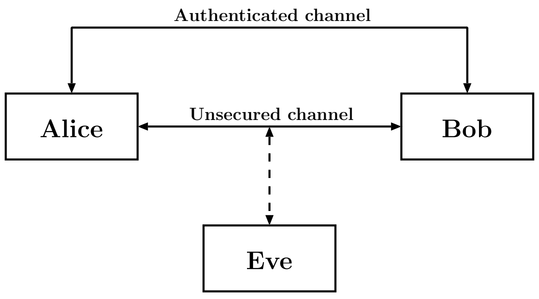


Figure 2.3: Public-key communication model.

problems. The reason for this is the much higher efficiency of symmetric-key cryptography compared to public-key cryptography. Therefore, most communication protocols are based on both primitives. As example public-key cryptography could be used for exchanging the secret key required for symmetric-key cryptography. Both primitives are described in Subsection 2.1.3 and Subsection 2.1.4.

### 2.1.3 Symmetric-Key Cryptography

Symmetric-key cryptography is based on a shared secret over which encryption/decryption of data is being enabled. The basic scheme is shown in Figure 2.2. Alice and Bob want to communicate in a secure way over an unsecure channel. In order to establish this behavior, in addition a secret and authenticated channel is required to distribute the secret key. This additional channel is one of the shortcomings. The reason for this is the difficult and expansive way of establishing such a channel. The other drawback is the key-management problem. In the basic model where only Alice and Bob—two persons—are communicating with each other, only one secret key is required. Adding another person which should be able to communicate with the other two implicates that already three secret keys are necessary. An alternative would be an on-line trusted third-party which creates and distributes the keying material on demand to prevent storing of unnecessary keys. Finally, as communication between multiple persons is still possible—sharing all the same secret key—the possibility of impersonating communication partners is given. Therefore non-repudiation would be desirable which in fact cannot be established with this primitive.

### 2.1.4 Public-Key Cryptography

Public-key cryptography uses a key pair—a public and a private key—instead of a single secret key. It was developed to eliminate all the shortcomings of symmetric-key cryptography and fulfills therefore all the security goals mentioned in Subsection 2.1.2. The basic scheme is shown in Figure 2.3. Alice and Bob want to communicate again in a secure way over an unsecure channel. The only difference in the communication model between the symmetric-key and the public-key scheme is in the additional channel between the two communication partners. For the public-key cryptography the channel must only be authentic and not secure. Therefore, it is possible for everyone to read the content but not to modify it in an unperceived manner. The authenticity is highly important as otherwise an adversary would be able to distribute wrong public keys, namely ones from which the corresponding private key is known. In such a scenario the encrypted message would not be safe anymore. If the channel is authentic then the instance which receives the public key can be sure that the corresponding private key is in possession of the right instance. Therefore, sending an encrypted message only intended to be readable by the receiver is easy as the encryption step can be fulfilled through the exposable public key of the receiver. The encrypted message can be afterwards decrypted with the corresponding private key which must be in possesion of the receiver only.

Public-key cryptography is much more complex as symmetric-key cryptography. As public-key cryptography eliminates all the shortcomings of symmetric-key cryptography and furthermore requires only an additional authenticated but not secure channel it makes sense to use this primitive to distribute a secret key between two parties that want to communicate in a secure way. Through this combination the high efficiency of symmetric-key cryptography can be used while avoiding the need for a secret and authenticated channel.

## 2.2 Block Ciphers

Block ciphers are widely spread throughout various communication protocols which are based on symmetric-key cryptography. The reason for is the high efficiency for encrypting/decrypting data with a secret key, from now on called $k$. Basic idea is that the transformation step operates on a full data block and not on single bits. Therefore, it must be guaranteed that the input data can be split up in a multiple of the data-block size. This is done through a padding function which is specially defined for each block-cipher algorithm. The output of a block cipher is determined by the input data, the secret key $k$ and the chosen algorithm. As block ciphers are deterministic algorithms—no randomness included—two instances stimulated with the same data set consisting of the input data and the secret key $k$ deliver the same output.

Noteable algorithms are the Data Encryption Standard (DES) [47], Triple DES [47], Rivest Cipher 5 (RC5) [55], Blowfish [58] and Rijndael [10] which became in 2000 the Advanced Encryption Standard (AES) [51]. DES, triple DES and AES are algorithms standardized by the National Institute for Standards and Technology (NIST). DES was introduced first followed by the Triple DES variant which is simply an improved version of DES to mitigate weaknesses found during cryptanalysis. Triple DES, as the name implies, operates on three single DES units whereas either two or three independent secret keys can be used. The three-key variant has the advantage of a higher bit-security compared to the two-key version and should therefore be preferred. The cryptographic community still was not satisfied with these algorithms and their corresponding bit-security levels. Hence a competition was started to find a successor of these two variants. In 2000, NIST announced the Rijndael algorithm [10] as winner which became after some small modifications—e.g. block size fixed to 128 bits, key size selectable between 128/192/ 256 bits—the most valuable and widespread block cipher AES [51].

Still the question remains how to act on block ciphers in detail. Figure 2.4 represents the easiest way to encrypt data through the aid of a block cipher. The data to be encrypted is first padded up to a multiple of the block length before it gets split up in blocks of same size. These represent in combination with the secret key $k$—remains the same for each block to be encrypted—the input data to the block cipher instance . As the transformations are independent from each other either one instance or multiple instances can be used. The maximum speedup can be achieved when as many block cipher instances

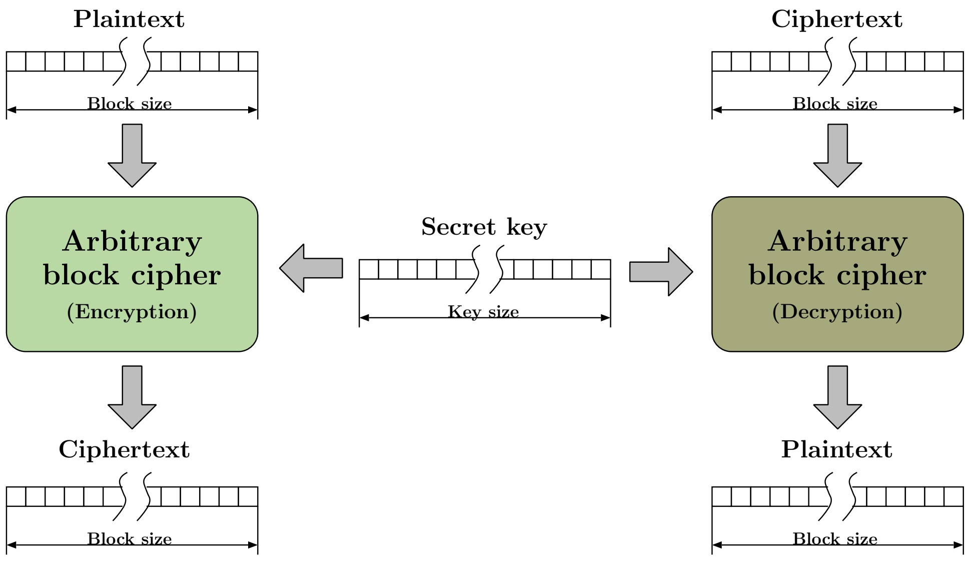


Figure 2.4: Encryption and decryption with an arbitrary block cipher.

as input blocks exist as then the encrypted data is available after the computation time of one block. The full ciphertext is received through a concatenation of the block-cipher outputs.

As this variant—called the Electronic Code Book (ECB) mode—exhibited some security flaws during the last years, modes of operation were introduced in order to increase the security level. Two modes during which the instances are still operated as block ciphers are explained in detail in the next two subsections. For more explanations on Block Ciphers, c.f. [35, 39].

### 2.2.1 Electronic Code Book (ECB) Mode

How to encrypt/decrypt data regarding the Electronic Code Book (ECB) mode is shown in Figure 2.5 and Figure 2.6. It represents the native mode as the data to be encrypted/decrypted is first padded up to a multiple of the block length—not explicitly shown—before it gets processed by the chosen block cipher instance in combination with the secret key $k$. As this key remains the same for all encryption/decryption procedures, the security level decreases as more plaintext/ciphertext pairs—generated with the same secret key $k$—are available to an attacker from which information on the used secret key can be revealed. Furthermore, as block ciphers are deterministic algorithms identical plaintext blocks are being transformed to identical ciphertext blocks and vice versa if the pair consisting of the block-cipher algorithm and the secret key $k$ remains the same. Therefore, the message confidentiality is not unconditionally given as if a plaintext/ciphertext pair once is being revealed it gets easily recognized if no change in the secret key $k$ or the block-cipher algorithm has been applied. Last but not least it hides poorly data patterns. For example, a change in the first data block gets isolated as the change cannot propagate into other blocks as well [35, 39].

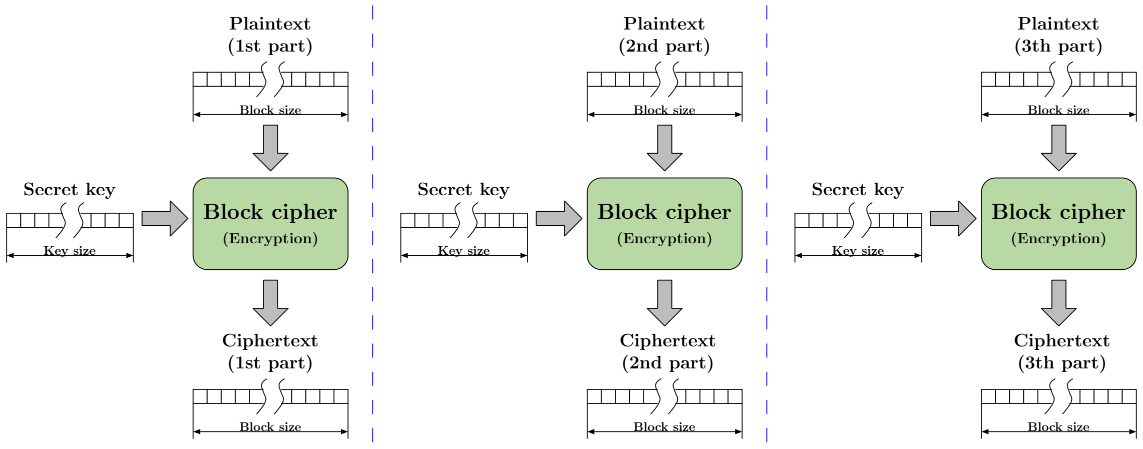


Figure 2.5: Encryption according Electronic Code Book (ECB) mode.

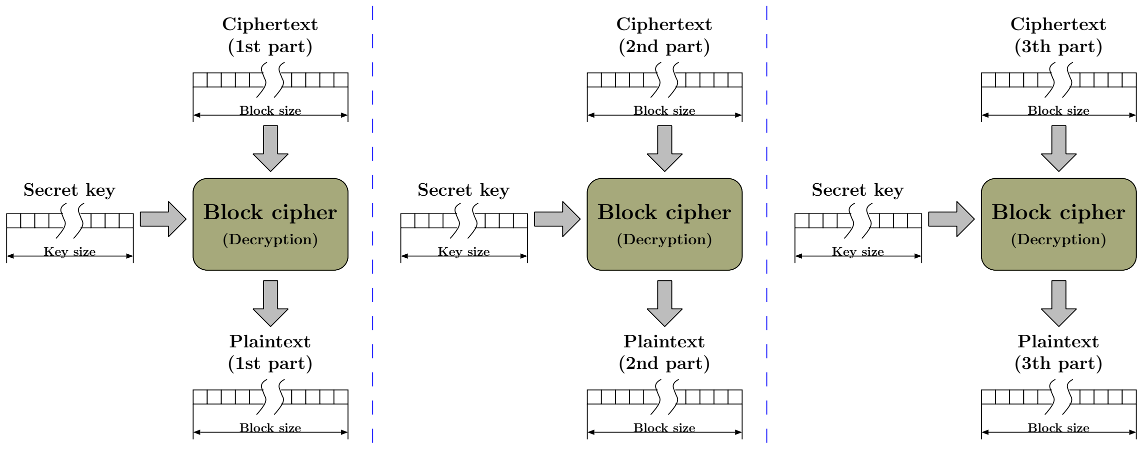


Figure 2.6: Decryption according Electronic Code Book (ECB) mode.

### 2.2.2 Cipher Block Chaining (CBC) Mode

In order to mitigate the security flaws introduced through the ECB mode, the Cipher Block Chaining (CBC) mode was invented. In that mode of operation the plain text is never directly applied to the block cipher as shown in Figure 2.7. Instead, an initial vector—ideally changes each time—gets XORed with the first input data block. The output of the block cipher represents afterwards on the one hand a part of the ciphertext and on the other hand the value which gets XORed with the second input block. Therefore, a change in the initial vector propagates through all steps. If this initial vector changes each time it is guaranteed that the same input data never corresponds to the same output data. Hence one of the security flaws of the ECB mode namely the confidentiality has been resolved. The other problem regarding the pattern hiding was improved but not fully removed. A change of one bit in the initialization vector will affect the whole ciphertext. Changes in the plaintext instead affect only following ciphertext blocks but not the already processed ones. If an attacker somehow achieves to apply multiple times the same initial vector it would be able to identify patterns through changing bits at various locations and observing the corresponding output. This improvement related to the ECB mode

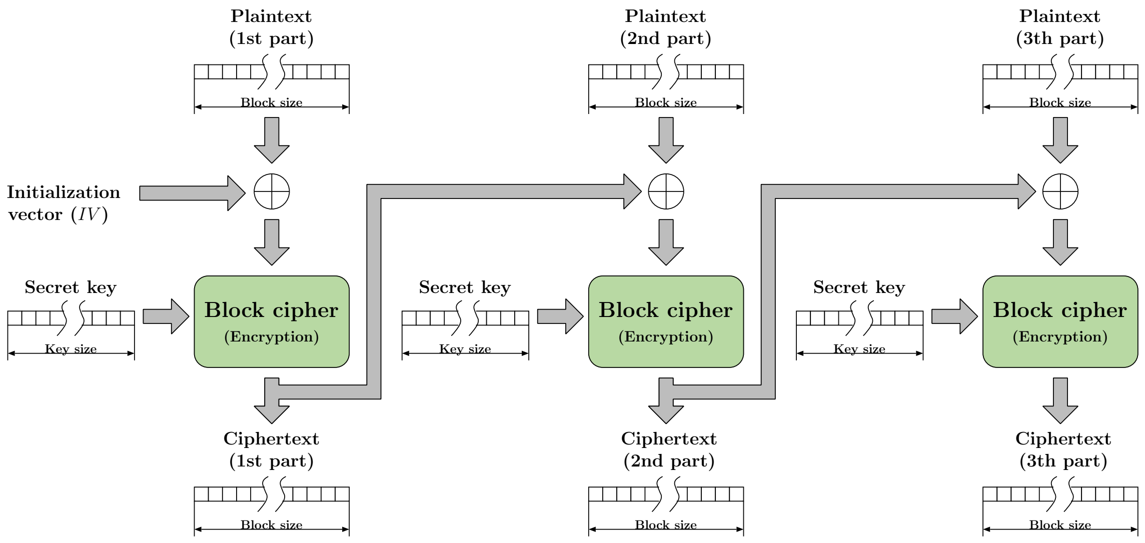


Figure 2.7: Encryption according Cipher Block Chaining (CBC) mode.

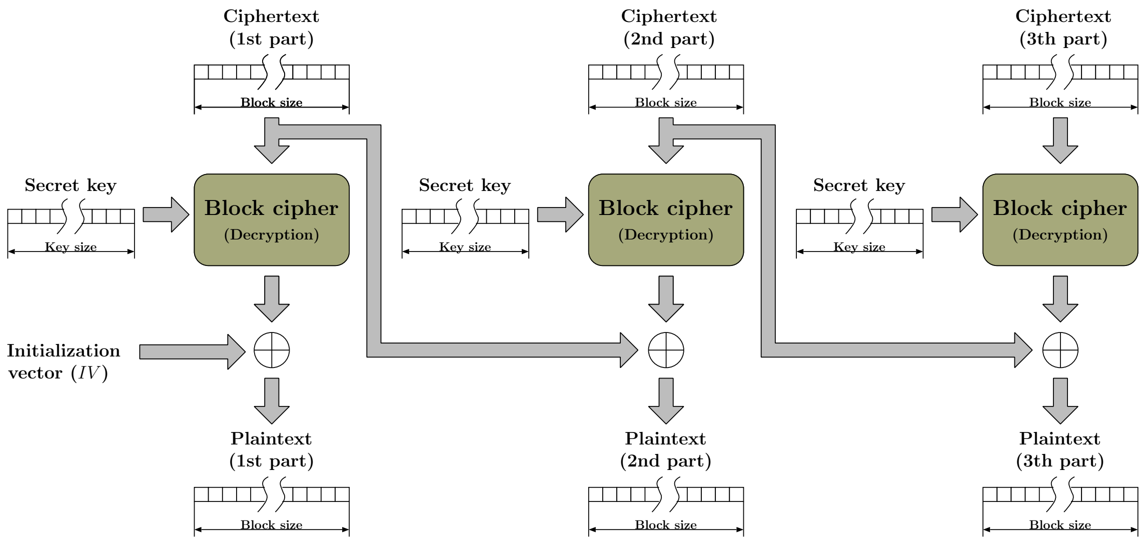


Figure 2.8: Decryption according Cipher Block Chaining (CBC) mode.

has also a drawback regarding performance as a parallelization of the encryption is not possible anymore as the various steps are depending on their predecessors. This changes for the decryption shown in Figure 2.8 as there a plaintext can be computed out of two ciphertext blocks and so a parallelization is again feasible. The computation of the first plaintext block differs from the rest as there is only one ciphertext block and the initial vector. Therefore, if a wrong initial vector is applied only the first plaintext block is invalid. This differs from the encryption process where the whole ciphertext would be wrong. Furthermore, a change of one bit in a ciphertext block would make the whole corresponding plaintext block invalid whereas only one bit of the following plaintext block will be flipped. All the other following plaintext blocks will be valid again [35, 39].

## 2.3 Stream Ciphers

Block ciphers are algorithms which take as input the plaintext and a secure key $k$ and deliver as result the corresponding ciphertext. As these operations are block based a parallelization generally can be applied through multiple instances of the same algorithm. Stream ciphers instead compute a pseudorandom keystream which is combined bit-by-bit with the plaintext. This combination step itself represents the encryption of the plaintext and delivers therefore the corresponding ciphertext. As stream ciphers are also used in symmetric-key cryptography both the transmitter and receiver must be able to compute the same keystream. This is done over an initial value from which the keystream can be derived, for instance, through digital shift registers. Therefore, the initial value can be seen as secret key $k$. Furthermore, both sides must have agreed before the transaction on the algorithm used to derive the pseudorandom keystream. Another difference is the fact that stream ciphers require for both encryption and decryption an encryption-only unit whereas block ciphers need both. The keystream is generally computed over a digital shift-register circuit like for example with a Linear Feedback Shift Register (LFSR). Another possibility is to use block ciphers as explained in the next three subsections. For more explanations on Stream Ciphers, c.f. [35, 40].

### 2.3.1 Cipher Feedback (CFB) Mode

Figure 2.9 and Figure 2.10 present encryption and decryption for the Cipher Feedback (CFB) mode which uses an arbitrary block cipher for the keystream generation. The stimulation of the first instance takes place with the initial vector $IV$ which must be known to both the transmitter and receiver and therefore represents the secret key $k$. The ouput of the first block cipher is afterwards combined with the plaintext and forms the corresponding ciphertext. Furthermore, this result is the input data to the next instance in order to create a keystream depending on the previous result. As the encryption step itself is simply an XOR operation of the block-cipher output with the plaintext, it can be easily reversed by exchanging the plaintext with the ciphertext. In order to perform a complete decryption the input to the next instance is again the ciphertext as the keystream must be exactly the same for both the encryption and the decryption procedure. An advantage of this scheme is the fact that it contains a self-synchronization as only the initial value must be known to both the transmitter and the receiver. The internal state doesn't matter as both sides are able to encrypt/decrypt the data independently from each other. Furthermore, a parallelization of the decryption can be applied as the keystream is derived from the different already known ciphertext parts. A drawback is that a loss of data permanently throws off decryption. Regarding pattern hiding flipping one bit of the plaintext during data encryption produces a flipped bit in the corresponding ciphertext. Furthermore, this modification propagates along as the ciphertext represents the data input to the next block cipher unit and therefore modifies up from this position the derived keystream. Flipping one bit of the ciphertext during the decryption effects at maximum two plaintexts and stays therefore locally restricted as the modification doesn't propagate further [35, 40].

### 2.3.2 Output Feedback (OFB) Mode

The Output Feedback (OFB) mode is very similar to the CFB mode and is shown in Figure 2.11 and Figure 2.12. Difference is that the keystream modification is achieved

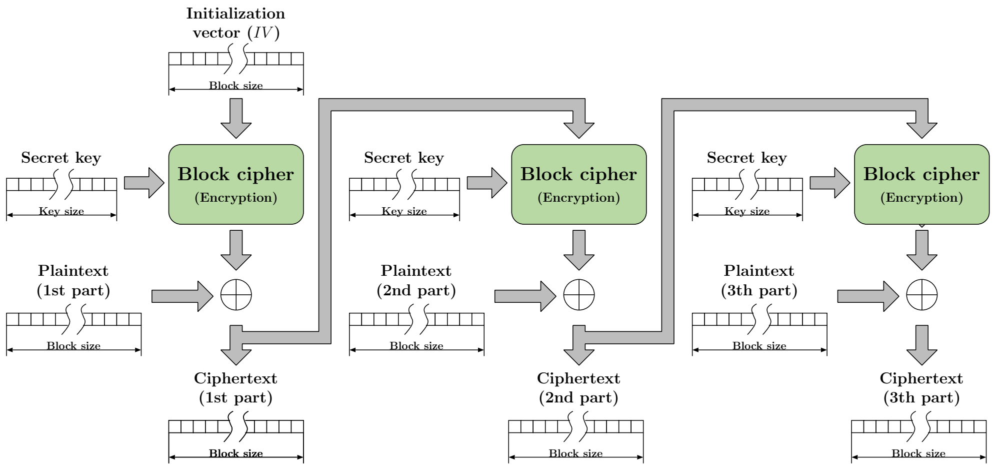


Figure 2.9: Encryption according Cipher Feedback (CFB) mode.

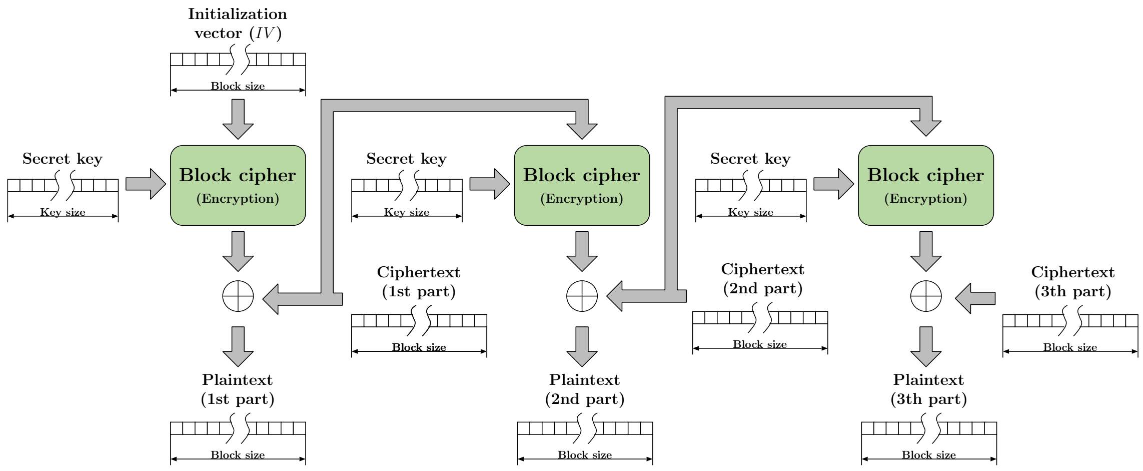


Figure 2.10: Decryption according Cipher Feedback (CFB) mode.

without taking the plaintext or ciphertext into account. On the first sight this looks like an improvement regarding performance but it isn't as the decryption cannot be parallelized. The reason for is that the input to a block-cipher unit is originated from the previous one which has to be computed first. Furthermore, pattern hiding is also lagging behind as flipping one bit of the plaintext during encryption or one bit of the ciphertext during decryption has only a locally restricted impact [35, 40].

### 2.3.3 Counter (CRT) Mode

Last but not least the Counter (CRT) mode gets described where the input to the various block-cipher units originates from a counter function as shown in Figure 2.13 and Figure 2.14. A requirement for this is that it produces a long sequence which is allowed to repeat afterwards. The keystream can therefore be computed in parallel as only the initial counter value and the corresponding function must be known. As both the plaintext for encryption or the ciphertext for decryption are simply combined with the pseudorandom keystream in order to achieve the desired output, the pattern hiding lags in the same way

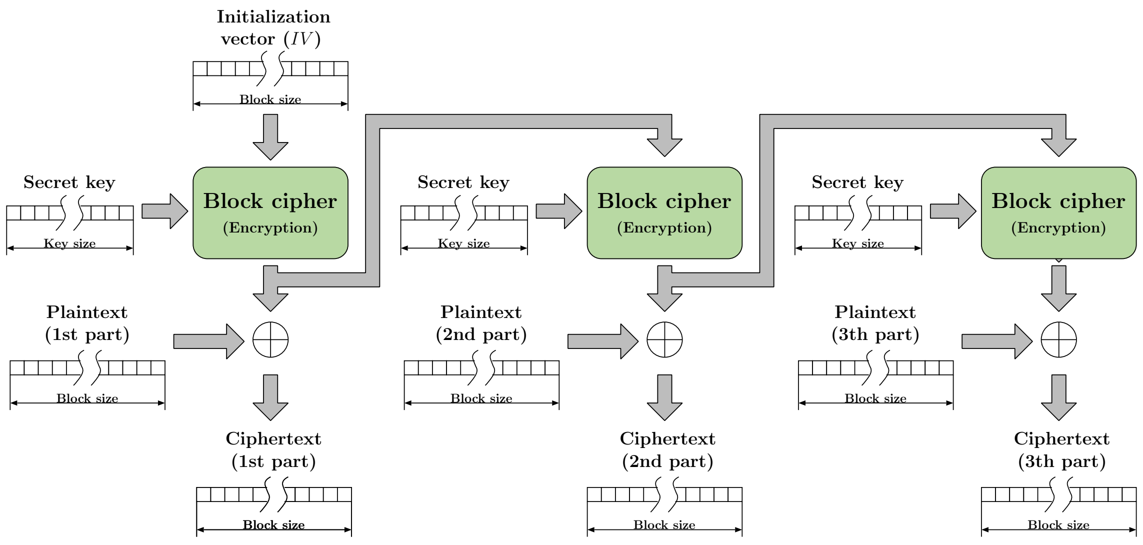


Figure 2.11: Encryption according Output Feedback (OFB) mode.

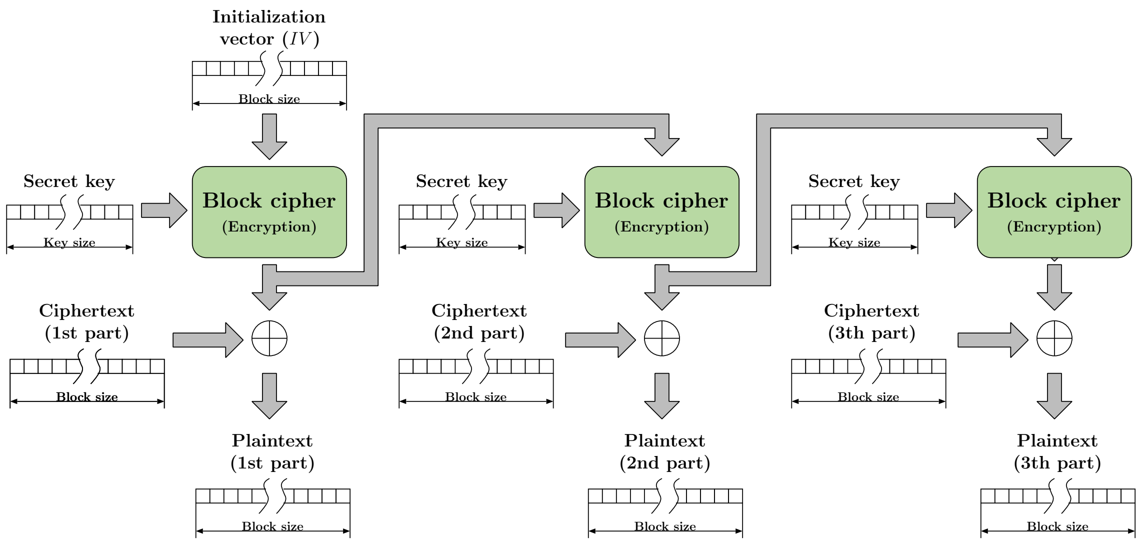


Figure 2.12: Decryption according Output Feedback (OFB) mode.

behind as described in the modes of operation before.[35, 40]

## 2.4 Message Authentication Codes (MAC)

Message authentication codes can be used to achieve data integrity by computing a checksum over the message while using a secret key $k$. As only the transmitter and the receiver are aware of this key it is not possible to modify the transmitted message by a third party without getting recognized. In order to achieve this, the transmitter computes a checksum over the message using the secret key $k$. Afterwards the message and also the computed checksum is transmitted to the receiver which computes the checksum on its own and compares it to the received one. It they match no modification of the message has been taken place.

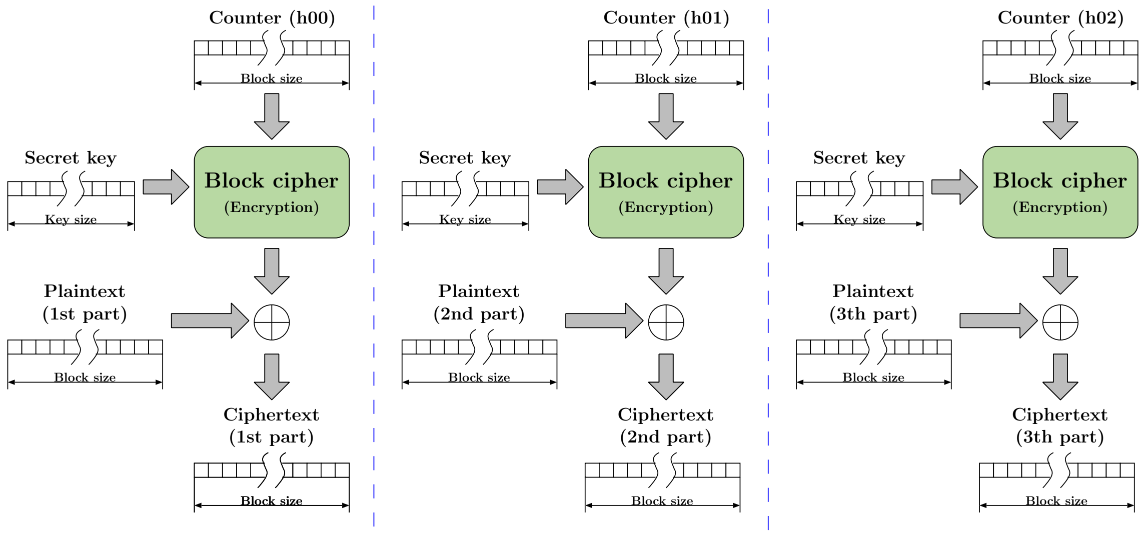


Figure 2.13: Encryption according Counter (CTR) mode.

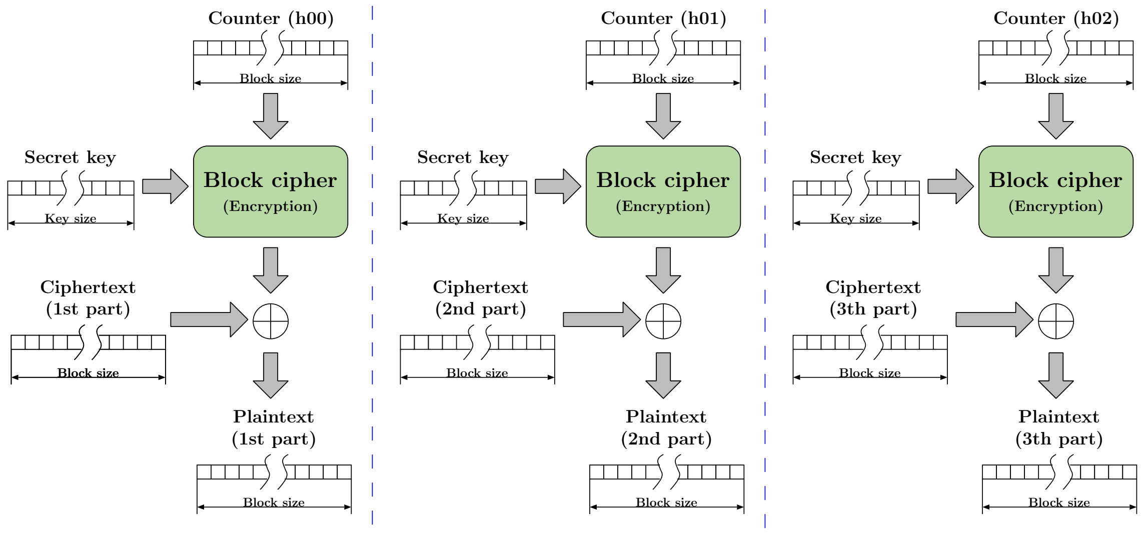


Figure 2.14: Decryption according Counter (CTR) mode.

Block ciphers for instance can be used to achieve this behaviour like for example with the Cipher Block Chaining Message Authentication mode (CBC-MAC) shown in Figure 2.15. This mode of operation is very similar to the CBC mode with the difference that the initial vector is set to zero and that only the last cipherblock is used. An available block cipher in CBC mode can therefore be easily reconfigured for MAC computation. Regarding security only fixed-length messages should be allowed for the standard CBC-MAC mode as an attack exists during which an attacker is able to append an arbitrary message which delivers as result the same MAC as before. For this, two messages and their corresponding MACs are required. A remedy would be to encode the data length into the first block. With this modification also variable lengths can be supported [35, 41].

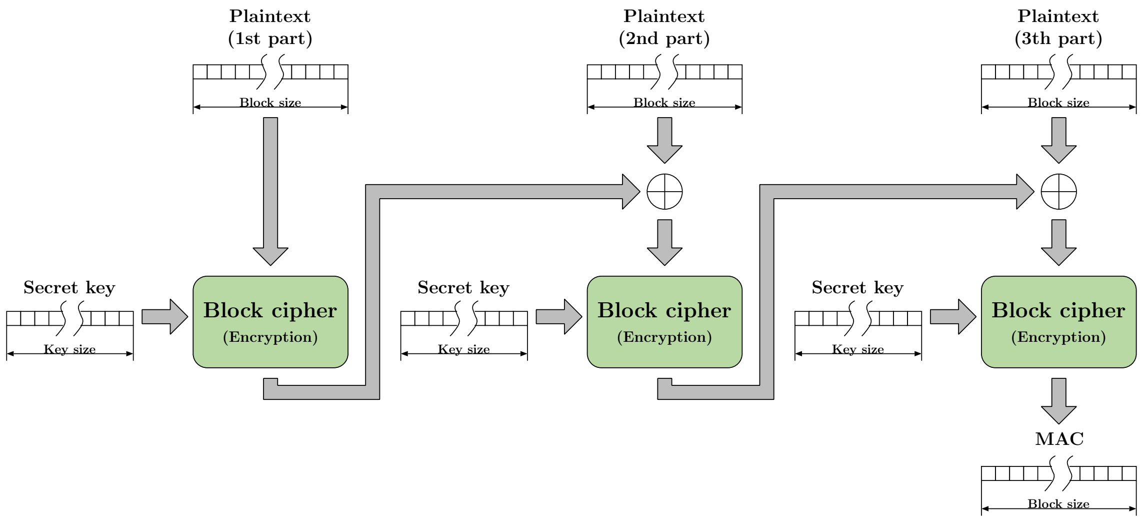


Figure 2.15: Computing the Message Authentication Code (MAC) for an arbitrary message using the CBC-MAC mode.

## 2.5 Hash Functions

Cryptographic hash functions compute of a given message $m$ the corresponding hash $h = hash(m)$ which is referred to in literature as message digest or simply digest. The length of it depends only on the algorithm used and not on the length of the given message $m$. Through this functionality data integrity of a message can be assured.

The cryptographic hash functions Message-Digest 4 (MD4) [56], MD5 [54], Secure Hash-Algorithm 1 (SHA-1) [48], SHA-2 [48] and also some candidates of the SHA-3 competition like Grøstl [15], Skein [12] or Blake [2] operate similarly to block ciphers on blocks of data which implies the need for a padding function. For the sake of completeness it must be stated that also other approaches exist, like for example, JH or Keccak which are also both final candidates for the SHA-3 competition.

Secure cryptographic hash functions must fulfill the following three requirements: **preimage resistance**, **second-preimage resistance** and **collision resistance**. More details on each of these requirements are given in the following list.

- **Preimage resistance**
  Given a message digest $h$ it should be difficult to find any message $m$ such that $h = hash(m)$ is fulfilled (similar to one-way functions).

- **Second-preimage resistance**
  Given a message $m_1$ it should be difficult to find any message $m_2$—$m_2$ must be not equal to $m_1$—such that $hash(m_1) = hash(m_2)$.

- **Collision resistance**
  It should be difficult to find any two messages $m_1$ and $m_2$—$m_2$ must be not equal to $m_1$—such that $hash(m_1) = hash(m_2)$.

The simplest scheme for achieving data integrity would be to transmit the message $m_1$ and also its message digest $h_1$ to the receiver, which can recompute the message digest and obtains therefore $h_2$. If these digests coincide, no modification of the message has

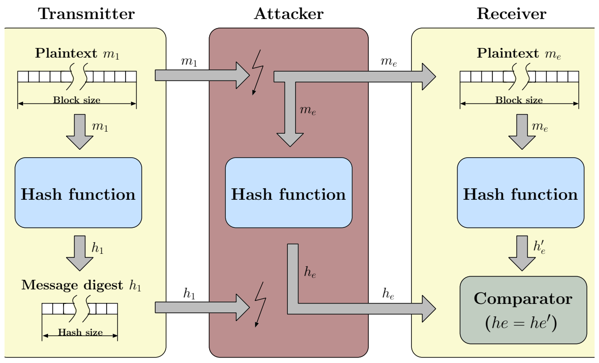


Figure 2.16: Man-in-the-middle attack on a communication scheme using only hash functions to obtain data integrity.

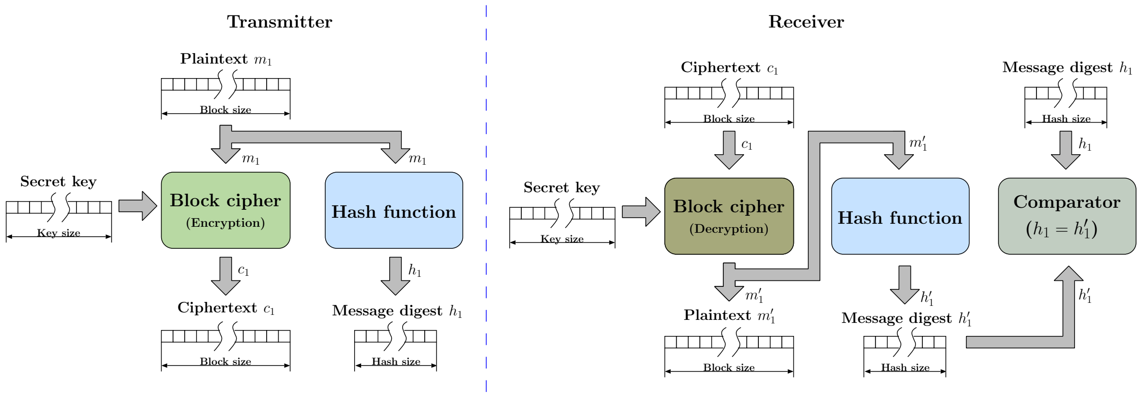


Figure 2.17: Communication scheme using cryptographic hash function in combination with symmetric-key encryption in order to obtain data integrity and data origin authentication (if only two instances knowing the secret key $k$).

occurred. The only drawback is the fact that a man-in-the-middle attack as shown in Figure 2.16 is possible where a third party exchanges both the original message $m_1$ and the corresponding digest $h_1$ with it's own message $m_e$ and the corresponding digest $h_e$. The receiver is only in the position of checking message modifications but not from whom the message was originated.

A better approach is given in Figure 2.17 where first the message $m_1$ is encrypted using the secret key $k$ which is only known to the transmitter and receiver. The plaintext on the other hand gets hashed and forms therefore the message digest $h_1$. Both the encrypted message $c_1$ and the digest $h_1$ get transferred. After receiving both data blocks the receiver first decrypts the ciphertext $c_1$ to obtain $m'_1$ which should be the original message sent by the transmitter. In order to proof the validity of the data the receiver computes out of the decrypted message $m'_1$ the hash value $h'_1$ and compares it to the received one. If they match it is guaranteed that no modification of the data has occurred [36, 42].

## 2.6 Hash-Based Message Authentication Code (HMAC)

Hash-based message authentication codes are another variant how to achieve data integrity and data origin authentication. The last aspect only applies if the instances knowing the secret key $k$ is limited to two. General idea behind is that the keying material $k$ is combined (e.g., using XOR operation) with the message $m$. The so gained message digest $h = Hash(m||k)$ and the original message $m$ get transferred to the receiver. In contrast to the scheme in Figure 2.17, the data is transmitted in a readable manner. After receiving the message $m$ it gets again combined with the keying material $k$ over the same function used at transmitter side. If the received digest and the computed one match, no corruption of the message has occurred [36, 43].

# Chapter 3

# Rijndael - Winner of the NIST AES Competition

In the early 1990s the U.S. Government stated its interest in a more secure way of encrypting data. Therefore, the National Institute of Standards and Technology (NIST) in combination with industry and the cryptographic community started the search for a standardized symmetric-key scheme in January 1997. Their goal was a Federal Information Processing Standard (FIPS) for an unclassified, publicly disclosed and worldwide royalty-free block cipher with a minimum supported block size of 128 bits and key sizes of 128, 192 and 256 bits. In August 1998, the First AES Candidate Conference (AES1) was held with the outcome of 15 potential candidates going for a detailed review. The results of the analysis conducted by the global cryptographic community were presented in March 1999 at the AES2 followed by a public comment period. NIST used both the analysis and the comments to select the five finalists, namely **MARS** [4], **Rivest Cipher 6 (RC6)** [57], **Rijndael** [10], **Serpent** [1] and **Twofish** [59]. This selection was also the end of round one and the starting shot for round two. The first initiative in the second round was a detailed analysis of the five finalists followed by a search for public comments including cryptanalysis, intellectual property and much more. For the last conference, the AES3, held on April 13-14, 2000 all submitters were invited to take a stand on comments of their algorithms. With the gained information from the last conference NIST announced in October 2000 Rijndael as winner of the competition [52].

As last step NIST published in February 2001 the Federal Information Processing Standard (FIPS) 197 [51] describing the algorithm nowadays known under the name Advanced Encryption Standard which restricts the block size to 128 bits and the key size to 128, 192 and 256 bits. Dependent on the key size used the algorithm is referred to as AES-128, AES-192 or AES-256. The original Rijndael algorithm [9] alternatively allows the independent choice of the key and block size out of the set of 128, 160, 192, 224 and 256 bits.

The AES algorithm was developed for symmetric-key cryptography and is a block cipher which encrypts blocks of data independent from each other. As this block cipher requires always 128 bits of data for encryption/decryption a padding function which fills up the data to a multiple of 128 bits is inevitable. Afterwards the data is split up into blocks of 128-bits size and iteratively or parallel processed by AES implementations consisting of the round transformation described in Section 3.1 and the round-key generation described in Section 3.2. For more details see [51].

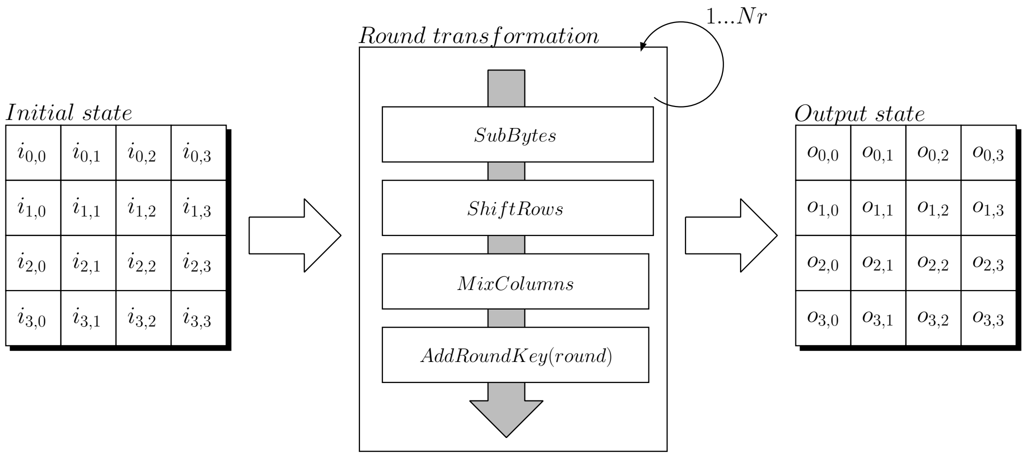

Initial state
$i_{0,0}$ $i_{0,1}$ $i_{0,2}$ $i_{0,3}$
$i_{1,0}$ $i_{1,1}$ $i_{1,2}$ $i_{1,3}$
$i_{2,0}$ $i_{2,1}$ $i_{2,2}$ $i_{2,3}$
$i_{3,0}$ $i_{3,1}$ $i_{3,2}$ $i_{3,3}$
Round transformation
1...Nr
SubBytes
ShiftRows
MixColumns
AddRoundKey(round)
Output state
$o_{0,0}$ $o_{0,1}$ $o_{0,2}$ $o_{0,3}$
$o_{1,0}$ $o_{1,1}$ $o_{1,2}$ $o_{1,3}$
$o_{2,0}$ $o_{2,1}$ $o_{2,2}$ $o_{2,3}$
$o_{3,0}$ $o_{3,1}$ $o_{3,2}$ $o_{3,3}$

Figure 3.1: Scheme of the AES encryption.

## 3.1 Round Transformation

The round transformation is build upon four operations: *SubBytes*, *MixBytes*, *MixColumns* and *ShiftRows*. As both encryption and decryption are supported the inverse operations are required in addition. Figure 3.1 and Figure 3.2 show the AES encryption and decryption scheme. Both are slightly simplified as the initial key addition, the round-key generation and also the last round (*MixColumns*/*InvMixColumns* operation skipped) are not illustrated.

The initial state matrix for encryption is obtained through the initial *AddRoundKey* operation where the master key is added (XORed) with the input message. Afterwards, the round transformation starts executing iteratively the *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey* operations as shown in Figure 3.1. For the *AddRoundKey* operation the appropriate round key is required which can be computed on-the-fly or precomputed and stored in memory. The on-the-fly computation is described in Section 3.2. The number of rounds to be executed depends on the key size. A key size of 128 bits implies 10 rounds, 192 bits 12 rounds and 256 bits 14 rounds. The last round is slightly different as the *MixColumns* operation is bypassed. The final state matrix represents the encrypted message.

Decryption is the inverse operation of encryption and is used to obtain the plaintext from a ciphertext. In order to achieve this behaviour a few modifications are necessary. First of all, the initial *AddRoundKey* operation requires the last round-key instead of the master key as for the decryption the round keys must be applied in an inverse manner to the encryption. Consequently an on-the-fly generation requires a longer computation time. At least as much longer as the time required to derive the last round key from the master key. The order of the round keys can be seen in Figure 3.9. Next, instead of the operations used during encryption their counterparts *InvShiftRows*, *InvSubBytes*, *AddRoundKey*, *InvMixColumn* must be used. The *AddRoundKey* transformation stays the same as only the input changes according to the previously described round-key generation. Last but not least the sequence of the round transformation must be adapted as shown in Figure 3.2. Similar to the encryption step the *InvMixColumns* operation is bypassed during the last round. The final state matrix represents the decrypted message. In the following, the components of the round transformation are described in more detail.

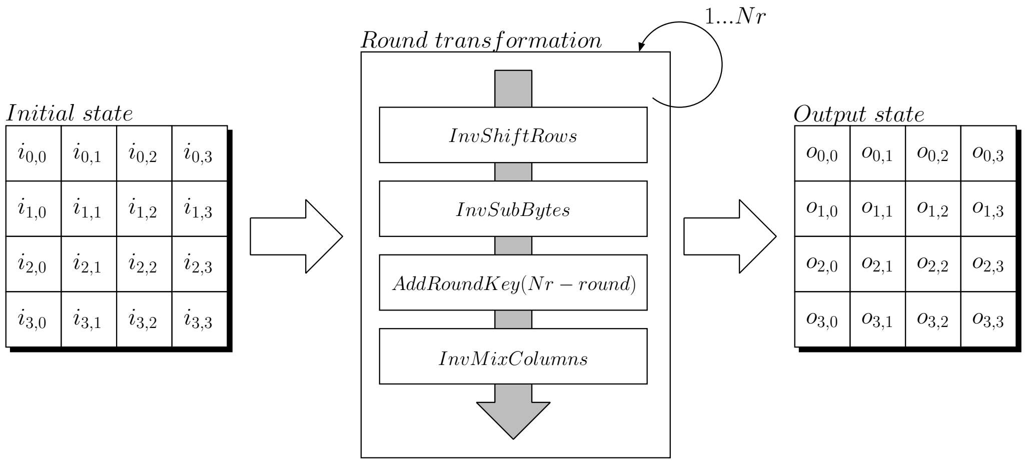

Initial state
$i_{0,0}$ $i_{0,1}$ $i_{0,2}$ $i_{0,3}$
$i_{1,0}$ $i_{1,1}$ $i_{1,2}$ $i_{1,3}$
$i_{2,0}$ $i_{2,1}$ $i_{2,2}$ $i_{2,3}$
$i_{3,0}$ $i_{3,1}$ $i_{3,2}$ $i_{3,3}$
Round transformation
1...Nr
InvShiftRows
InvSubBytes
AddRoundKey(Nr − round)
InvMixColumns
Output state
$o_{0,0}$ $o_{0,1}$ $o_{0,2}$ $o_{0,3}$
$o_{1,0}$ $o_{1,1}$ $o_{1,2}$ $o_{1,3}$
$o_{2,0}$ $o_{2,1}$ $o_{2,2}$ $o_{2,3}$
$o_{3,0}$ $o_{3,1}$ $o_{3,2}$ $o_{3,3}$

Figure 3.2: Scheme of the AES decryption.

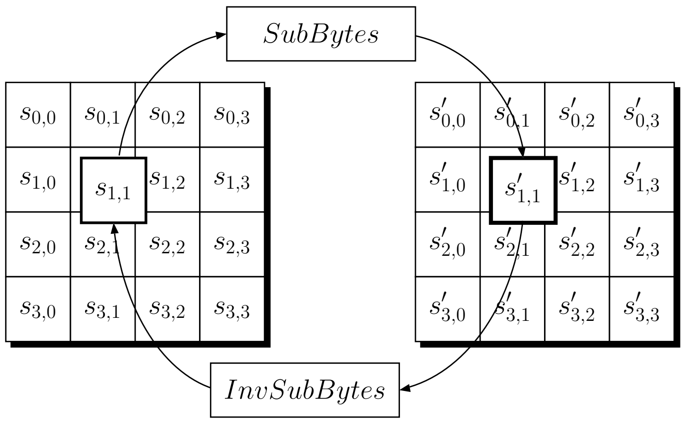

SubBytes
InvSubBytes
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s'_{0,0}$ $s'_{0,1}$ $s'_{0,2}$ $s'_{0,3}$
$s'_{1,0}$ $s'_{1,1}$ $s'_{1,2}$ $s'_{1,3}$
$s'_{2,0}$ $s'_{2,1}$ $s'_{2,2}$ $s'_{2,3}$
$s'_{3,0}$ $s'_{3,1}$ $s'_{3,2}$ $s'_{3,3}$

Figure 3.3: An *InvSubBytes* transformation is the inverse of the *SubBytes* transformation.

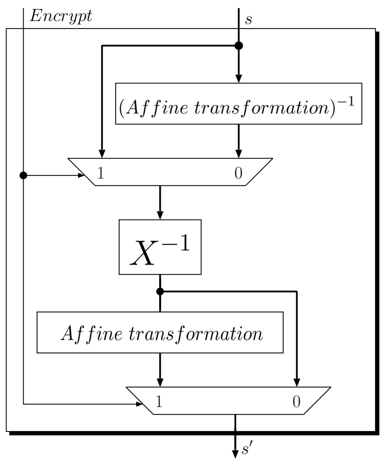


Figure 3.4: Scheme of the *SubBytes/InvSubBytes* transformation.

### 3.1.1 SubBytes/InvSubBytes

The *SubBytes* and *InvSubBytes* transformation, respectively is a non-linear byte substitution that substitutes each byte of the state as shown in Figure 3.3. They can be realized either over a Look-Up Table (LUT) or by an on-the-fly computation. The latter is shown in Figure 3.4. In order to get the *SubBytes* functionality the encryption signal must be set to one which implies that first the multiplicative inverse in the finite field $GF(2^8)$ is taken followed by a subsequent affine transformation. In order to get the *InvSubBytes* functionality this sequence must be reversed. For more details see [51].

### 3.1.2 ShiftRows/InvShiftRows

Both transformations are rotate operations and equal regarding the offset of each row. The only difference is the rotate direction because it is a left rotate for the *ShiftRows* and a right rotate for the *InvShiftRows* transformation. The impact of the *ShiftRows*

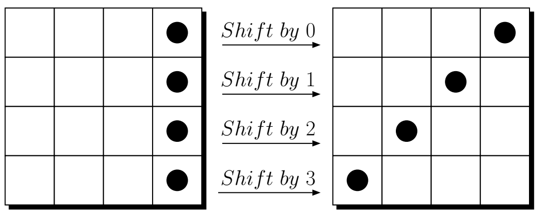


Figure 3.5: The effect of the *ShiftRows* transformation on the state.

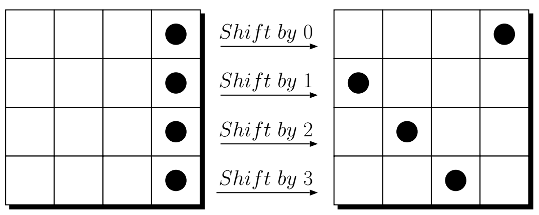


Figure 3.6: The effect of the *InvShiftRows* transformation on the state.

transformation and the *InvShiftRows* transformation on the state matrix is shown in Figure 3.5 and Figure 3.6.

### 3.1.3 MixColumns/InvMixColumns

These transformations can be seen as a matrix multiplication and operates on a complete column instead of a single byte. The matrices used for both transformations are circulant meaning that each row has the same content but is shifted to the right by one with respect to the row above. In short, we may write for the *MixColumn* matrix

$$B = circ(02, 03, 01, 01) \tag{3.1}$$

and for the *InvMixColumn* matrix

$$B = circ(0e, 0b, 0d, 09). \tag{3.2}$$

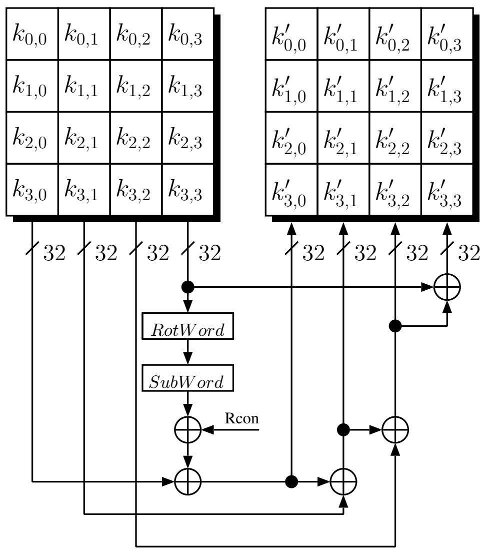
The detailed notation for the *MixColumns* multiplications can be seen in Formula 3.3 and for the *InvMixColumns* multiplication in Formula 3.4.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{3.3}$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{3.4}$$

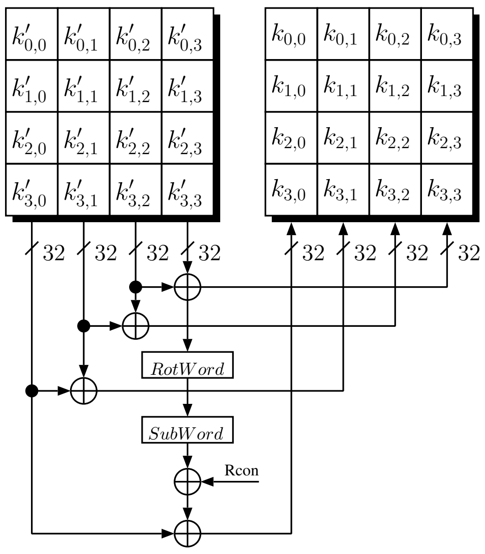
### 3.1.4 AddRoundKey

The *AddRoundKey* transformation XORes each byte of the round key—either available over an on-the-fly computation or through a lookup in the memory in case of precomputation—with the appropriate byte of the state. Details on the generation can be found in Section 3.2.



$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
$k'_{0,0}$ $k'_{0,1}$ $k'_{0,2}$ $k'_{0,3}$
$k'_{1,0}$ $k'_{1,1}$ $k'_{1,2}$ $k'_{1,3}$
$k'_{2,0}$ $k'_{2,1}$ $k'_{2,2}$ $k'_{2,3}$
$k'_{3,0}$ $k'_{3,1}$ $k'_{3,2}$ $k'_{3,3}$
32
RotWord
SubWord
Rcon

Figure 3.7: Scheme of the forward round-key generation.



$k'_{0,0}$ $k'_{0,1}$ $k'_{0,2}$ $k'_{0,3}$
$k'_{1,0}$ $k'_{1,1}$ $k'_{1,2}$ $k'_{1,3}$
$k'_{2,0}$ $k'_{2,1}$ $k'_{2,2}$ $k'_{2,3}$
$k'_{3,0}$ $k'_{3,1}$ $k'_{3,2}$ $k'_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
32
RotWord
SubWord
Rcon

Figure 3.8: Scheme of the backward roundkey generation.

## 3.2 Round-Key Generation

All round keys can be computed with the aid of a *SubWord* transformation that applies four times the *SubBytes* function, a *RotWord* transformation that rotates four bytes to the left by one and a small Look-Up Table (LUT) named Rcon containing the round constants (one for each round).Details on the round constants can be found in FIPS-197 [51]. The scheme for the forward round-key generation and also its inverse are shown in Figure 3.7 and Figure 3.8. Both can be computed on-the-fly.

For the forward round-key generation it is the easiest to start with the first column of the key matrix which can be updated through an XOR operation with the modified values–*RotWord*, *SubWord* and Rcon applied—of the last column. Afterwards, the rest is straight forward as always the previously gained result is taken and XORed with the next column until all are updated.

For the backward round-key generation the last column must be saved in a temporary place as it is required later on. Afterwards, it is updated through an XOR operation with the column before. This is repeated another two times (third column is updated with second column and afterwards the second column is updated with the first column). The final step is achieved by taking the temporary values, applying the *RotWord*, *SubWord* and Rcon and XORing it afterwards to the first column. In that way the predecessor from a specific round key can be computed.

Restricted environments typically claim for an on-the-fly computation of the round keys as the storage is simply often not available. Hence, it is important to know that a decryption has a big drawback regarding computation time. The reason for this is that the order of the round keys must be applied in a reverse manner. As most of the time an AES implementation obtains the master key and a ciphertext to be decrypted, it is necessary to first compute all the round keys down to the last one before the real computation can start. This fact is also illustrated in Figure 3.9.

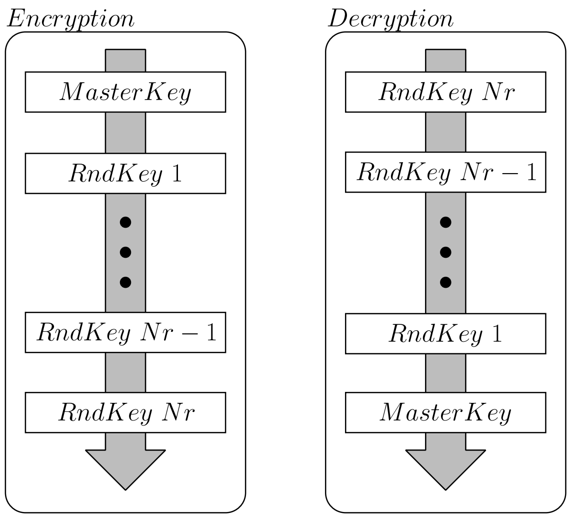


Figure 3.9: Order of the round keys depending on the mode of operation.

# Chapter 4

# Grøstl - One of the Finalists of the NIST SHA-3 Competition

The Federal Information Processing Standard (FIPS) 180-3 [48] specifies cryptographic hash algorithms converting a variable-length message into a fixed-length message digest. The algorithms specified can be grouped into SHA-1 and SHA-2. SHA-0 is no longer included as it contains a weakness which has been fixed in SHA-1 that is now its replacement. SHA-2 encompasses the functions SHA-224, SHA-256, SHA-384 and SHA-512 and is significantly different to SHA-1, which is the most widely used secure hash algorithm. In 2005, a possible mathematical weakness of SHA-1 was identified. As SHA-2 is algorithmically similar, it is assumed that this security flaw might occur in the group of algorithms summarized as SHA-2 too. Until today no successful attacks on SHA-2 have been reported. Because of the advances in cryptanalysis and the disclosed weaknesses of SHA-1 and probably SHA-2, NIST started a public competition searching for a successor of these algorithms in November 2007. The winner will be named SHA-3 and will extend the algorithms currently specified in FIPS 180-3. 64 candidates were submitted by October 2008 but only 51 of them were selected for the official first round which started in December 2008. 14 out of 51 candidates made it to the second round with the starting shot in July 2009. NIST allocated a year for public comments and internal reviews on the 14 chosen candidates. Based on the information gained, 5 finalists were selected for the third and also final round, namely **BLAKE**, **Grøstl**, **JH**, **Keccak** and **Skein**. This last round started in December 2010 and is still ongoing. From the beginning of round three until January 2011 submitters of the final candidates were allowed to make minor changes on their algorithms to incorporate improvements gained through public comments and internal reviews. NIST planned afterwards again an additional year for public comments. The final SHA-3 conference was held in March 2012. Submitters of the final candidates were invited to take a stand on comments to their algorithms. The winner of the competition is planned to be announced in late 2012 [44, 48, 53].

Grøstl is one of the finalists of the SHA-3 competition and is used to compute a fixed-length message digest out of a message with variable length. During the SHA-3 competition all the authors were allowed to make once minor changes on their algorithms in order to incorporate improvements triggered by public comments and internal reviews. Due to that, the specification from the 31st of October, 2008 was adapted to the newer one from the 2nd of March, 2011. The former one is since then known as Grøstl-0 specification. The following algorithmic description is targeting the newer specification from the 2nd of

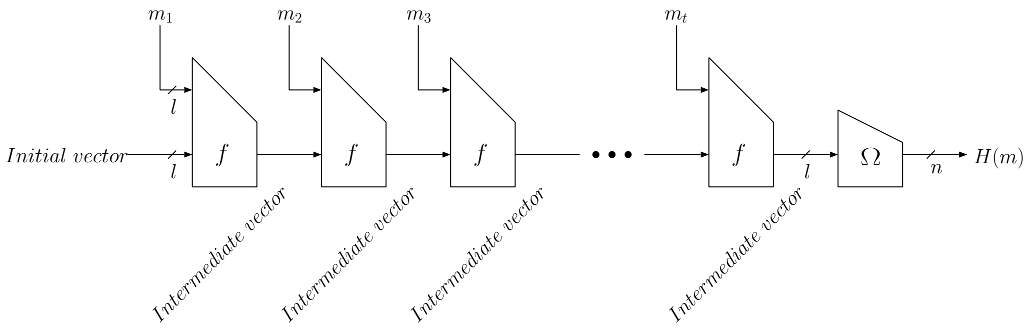


Figure 4.1: The Grøstl hash function.

March 2011. Differences to Grøstl-0 will be highlighted through an information field.

## 4.1 Compression Function

Grøstl supports the variants Grøstl-224, Grøstl-256, Grøstl-512 and Grøstl-1024. The appended number represents the bit length of the message digest from now on called $n$. For Grøstl-224 and Grøstl-256 the input width represented by $l$ is 512 bits whereas it is increased to 1024 bits for Grøstl-512 and Grøstl-1024. The basic concept of the hash function is shown in Figure 4.1 and reminds on a block cipher in cipher-block chaining mode (CBC). A padding function is inevitable for the same reason as for block ciphers (input width must be always of same size - either 512 bits or 1024 bits). Figure 4.1 assumes that the padding has been already applied. The message is furthermore split up in blocks of size 512 bits and 1024 bits, respectively. As no intermediate hash value is available for the first compression function $f$ a well defined initial vector is used.

**Info:** In order to improve the overall security the initial vectors differ to the ones from the Grøstl-0 specification.

The output of the compression function represents an intermediate hash value acting in combination with the next message block as input to the subsequent compression function. After processing of the last message part an output transformation is applied to reduce the amount of bits to the desired digest size.

The compression function is shown in detail in Figure 4.2 and consists mainly of two blocks $P$ and $Q$, both $l$-bit permutations. For the input to the $Q$-block no computational effort is required as the message part goes straight in. This differs to the $P$-block as there the input is the result of the intermediate hash value XORed with the input message. Both outputs are XORed to the intermediate hash value and form so the new one.

## 4.2 Output Transformation

Figure 4.3 presents the output transformation $\Omega$. The intermediate hash value—output of the last compression function—is applied to the $P$-block and the result is afterwards XORed to it. Last but not least a truncation step is required as the message digest must have a size of 224, 256, 512 or 1024 bits. Therefore, as many leading bits as to form the desired digest size are cut off. For more information see the Grøstl specification [15].

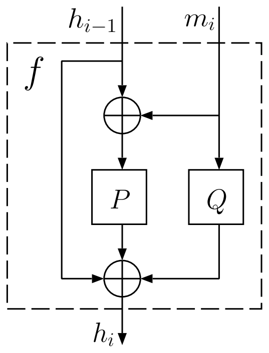


Figure 4.2: The compression function $f$ build upon a pair of $l$-bit permutations $(P/Q)$.

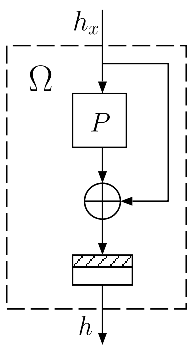


Figure 4.3: The output transformation computes $P(h_x) \oplus h_x$ and cuts off the leading bits to obtain the $n$-bit message digest.

## 4.3 Permutations $P$ and $Q$ in Detail

The $l$-bit permutations $P$ and $Q$ come each in two variants from now on named $P_{512}$ and $P_{1024}$ as well as $Q_{512}$ and $Q_{1024}$. The reason for is the input size $l$ which can be either 512 bits or 1024 bits depending on the variant used. For Grøstl-224 and Grøstl-256 an input size of 512 bits is mandatory whereas for the other two variants the size must be extended to 1024 bits. The inner structure of a permutation is shown in Figure 4.4 and is equal for $P$ and $Q$. It consists of four round transformations: *AddRoundConstant*, *SubBytes*, *ShiftBytes* and *MixBytes*. The execution order is also presented in Figure 4.4. Beside, note the similarity between the permutation structure and the AES round transformation. The difference between the two variants of each permutation is the number of rounds and the behaviour of the four round transformations. For the small variants including Grøstl-224 and Grøstl-256 10 rounds are mandatory which increases to 14 rounds for the other two variants. The behaviour of the round transformations will be explained for Grøstl-224 and Grøstl-256 only. Details for the others can be found in the Grøstl specification, cf. [15].

### 4.3.1 AddRoundConstant

The *AddRoundConstant* transformation adds a round-dependent constant to the state matrix. Equation 4.1 presents the round constant used for a $P$ permutation whereas Equation 4.2 targets the $Q$ permutation.

**Info:** In order to improve the overall security the round constants differ to the ones from the Grøstl-0 specification. In Grøstl-0, only a single byte gets modified for both permutations.

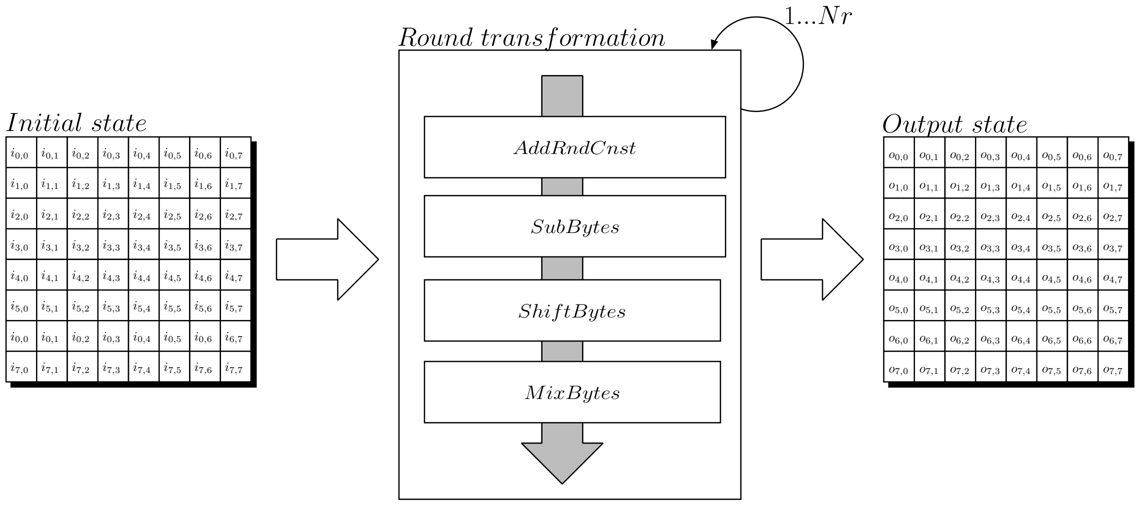


Figure 4.4: Structure of the $P$ and $Q$ $l$-bit permutation.

$$C_{P_{512}}[\ i\ ] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix} \tag{4.1}$$

$$C_{Q_{512}}[\ i\ ] = \begin{bmatrix} ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff & ff & ff & ff & ff & ff & ff & ff \\ ff \oplus i & ef \oplus i & df \oplus i & cf \oplus i & bf \oplus i & af \oplus i & 9f \oplus i & 8f \oplus i \end{bmatrix} \tag{4.2}$$

### 4.3.2 SubBytes

The *SubBytes* transformation of Grøstl substitutes each byte of the state matrix in the same manner as in AES. See Section 3.1.1 for more details.

### 4.3.3 ShiftBytes

The *ShiftBytes* transformation rotates in a similar manner as AES each row of the state matrix. Simply the matrix is four times bigger and the offsets—see Figure 4.5 and Figure 4.6—are different.

### 4.3.4 MixBytes

This transformation can be seen as a matrix multiplication and operates therefore on a complete column instead of a single byte. The matrix used is circulant meaning that each

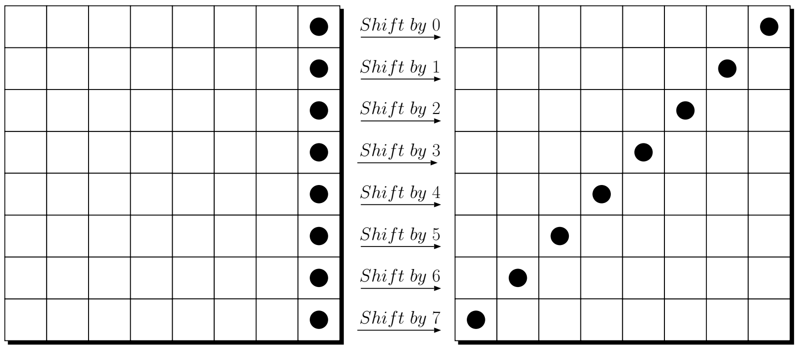


Figure 4.5: Offsets for the *ShiftBytes* transformation in the $P$ permutation.

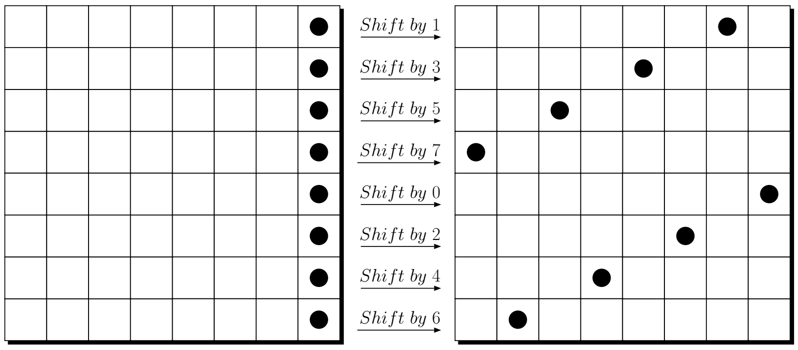


Figure 4.6: Offsets for the *ShiftBytes* transformation in the $Q$ permutation.

row has the same content but it is shifted to the right by one with respect to the row above. In short, we may write for the *MixColumn* matrix

$$B = circ(02, 02, 03, 04, 05, 03, 05, 07). \tag{4.3}$$

The detailed notation for a *MixBytes* column multiplication can be seen in Formula 4.4.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \\ s'_{4,c} \\ s'_{5,c} \\ s'_{6,c} \\ s'_{7,c} \end{bmatrix} = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \\ s_{4,c} \\ s_{5,c} \\ s_{6,c} \\ s_{7,c} \end{bmatrix} \tag{4.4}$$

# Chapter 5

# Related Work

There exist many papers in literature that present low-resource hardware implementations of AES or Grøstl. Since publication of the algorithms in 1998 and 2008, several optimization techniques have been proposed that reduce the area requirements for both ASIC and FPGA platforms. One example is the optimized AES S-box implementation of Canright [5] that has been also used by Feldhofer et al. [11] to realize a very compact version of AES-128.

Feldhofer et al. reported in 2005 a design requiring only 3.4 kGE for both encryption and decryption. The results stated are after synthesis to the gate level using a 0.35 µm CMOS process from Philips Semiconductors and are using standard cells only. The taped-out version including clock tree, filler cells and other layout overhead has a complexity of around 4.4 kGE. Performance measures of the produced chip showed that it works correctly with a supply voltage larger than 0.65 V. The gained frequency is thus reduced to around 2 MHz. When supplied with the full supply voltage of 3.3 V a maximum frequency of 80 MHz is reached. The design based on an 8-bit datapath requires 1,032 clock cycles to encrypt one 128-bit block. Decryption requires 1,165 clock cycles. The throughput for encryption at the maximum supply voltage—maximum frequency of 80 MHz is reached—amounts to 9.9 Mbps. Reducing the power consumption was a big aspect of the authors to enable AES in environments not imaginable at the time the paper was written. With the incorporated power-reduction measures the power consumption of the chip operated at a frequency of 100 kHz amounts to only 4.5 µW.

Similar results have been reported by Hämäläinen et al. [19] in 2006. They presented an encryption-only design which occupies 3.1 kGE of chip area. In contrast to Feldhofer et al. [11] no backend design was applied as the target was not a chip for production. Therefore all numbers stated are gained after synthesis to the gate level using a 0.13 µm standard-cell CMOS technology. Their design was based on an 8-bit datapath and requires only 160 clock cycles per block. With the maximum frequency of 152 MHz a throughput of 121 Mbps was achieved. Furthermore they performed a gate-level power analysis based on switching activities when stimulated with random test vectors resulting in a power estimation of 37 µW/MHz . Supporting AES decryption was estimated with an additional chip area of 25 % resulting in 3.9 kGE.

Kaps et al. [29] and Kim et al. [34] reported encryption-only designs requiring around 4 kGE in total after synthesis to the gate level. Kaps et al. used a 0.13 µm ASIC library from TSMC, specially characterized for low power. The design based on an 8-bit datapath and supporting CBC mode is capable of encrypting one block of data (128 bits) within 534 clock cycles consuming a total power of around 24 µW when operated at 500 kHz. Kim et al. on the other hand used a 0.25 µm standard-cell CMOS process from Hynix Corp. and Samsung Electronics. Their design encrypts one 128-bit block of data in 870 clock cycles. For the power estimation Synopsys PowerCompiler was used resulting in 21.4 µW for the Samsung Electronics technology and 4.85 µW for the Hynix Corp. technology. For both estimations 2.5 V supply voltage were applied in combination with a clock frequency of 100 kHz.

At EUROCRYPT 2011, Moradi et al. [45] presented an area-optimized implementation of encryption-only AES which needs about 2.4 kGE after synthesis to the gate level using the standard cell library UMCL18G212T3 which is based on the UMC L180 0.18 µm process technology operating with a typical supply voltage of 1.8 V. For the power estimation Synopsys Power Compiler version A-2007.12-SP1 was used. Both synthesis and power estimation were applied with a target frequency of 100 kHz. Their work was split in two parts targeting different goals, namely first solely low area and second to protect the previous developed design against side-channel attacks (first order DPA). The unprotected version occupies 2.4 kGE of chip area and requires only 226 clock cycles to encrypt one data block, which marks the lowest level of state-of-the-art AES implementations. The throughput and the power consumption amount to 57 Kbps and 7 µW, when operated with 100 kHz. The protected version on the other hand requires around 11 kGE. An encryption is performed within 266 clock cycles. The achieved throughput for 100 kHz operating frequency amounts so to 48 Kbps and the power consumption to around 24 µW.

As opposed to AES, there exist only a few publications so far that describe low-area optimizations for Grøstl on ASIC devices. Tillich et al. [62] have been the first who presented an implementation requiring 14.6 kGE after synthesis to the gate level using a 0.35 µm technology from Austriamicrosystems. They acquired their results with the Cadence PKS shell with enabled low-power option. Their design requires 196 clock cycles to compute the message digest out of one block of data with size of 512 bits. With the maximum frequency of around 55 MHz a throughput of around 146 Mbps is capable. The power consumption was estimated with around 221 µW when operated with a clock frequency of 1 MHz. This achieved result also corresponds well to the area estimations given in the Grøstl specification from Gauravaram et al. [15] which reported a size of less than 15 kGE. Further implementations have been presented by Katashita et al. [31], Guo et al. [18] and Henzen et al. [21] which require between 34.8 and 72 kGE after synthesis to the gate level. The most recent work done by Kavun et al. [32] was presented in the scope of the third SHA-3 conference in March 2012. Their design bases on an 8-bit datapath has been synthesized to the gate level using a 90 nm technology and occupies 9.2 kGE of chip area. For hashing a single block with size of 512 bits 1,280 clock cycles are required. The throughput of 40 Kbps was achieved with a clock frequency of 100 kHz.

In view of FPGA platforms, large effort has been made to reduce the complexity by reusing existing hardware components, e.g., Block RAMs, LUTs and DSPs. One of the first low-resource AES implementations on FPGAs has been presented by Chodowiec and Gaj [8] in 2003. Their AES-128 implementation needs 222 slices and 3 Block RAMs on a low-cost Xilinx Spartan II XC2S30 FPGA supporting both encryption and decryption with a throughput of 150 Mbps. The maximum frequency varies between 50 up to 60 MHz depending on the speed grade of the FPGA. Besides other optimization techniques, they made use of Look-Up Tables (LUTs) to efficiently implement shift registers such that intermediate values can be easily shifted without generating additional address logic.

In the upcoming years, the design has been improved by several authors, e.g., by Good et al. [17] who presented a design needing only 124 slices and 2 Block RAMs of size 4 Kb on a Xilinx Spartan II XC2S15-6. Their design was based on an 8-bit datapath and achieved an average throughput of 2.2 Mbps. The maximum frequency was limited to 68 MHz. Other implementations have been reported by Chi-Wu et al. [22] and Bulens et al. [3]. The former reported a design with an 32-bit datapath occupying 148 slices and 11 Block RAMs on a Spartan-3 XC3S200. As througput and maximum operating frequency 647 Mbps and 287 MHz were stated. The latter reported a design which uses no Block RAMs and stated results for a Virtex-5 and a Spartan-3 FPGA. The design supporting both encryption and decryption occupies on a Virtex-5 550 slices in contrast to 2,150 slices on a Spartan-3. As maximum frequency 350 MHz for the Virtex-5 and 150 MHz for the Spartan-3 were presented. The througput is limited to 4.1 and 1.7 Gbps, respectively.

A very compact FPGA implementation of Grøstl has been presented by Jungk et al. [25, 26, 27] in 2010, 2011 and 2012. The latest work has been presented at the third SHA-3 conference. They applied several optimization techniques on Grøstl that have been previously applied on AES. Their design based on an 64-bit datapath needs 1,125 slices on a Spartan-3 FPGA (note that the design needs only 967 slices without Fast Simplex Link (FSL) interface) and 470 slices on a Virtex-5 FPGA (355 slices without Fast Simplex Link (FSL) interface) requiring no Block RAMs. The interface affects the throughput only minimal. For the Spartan-3 FPGA the throughput is limited to around 580 Mbps and the maximum frequency to around 180 MHz. Exchanging the Spartan-3 through an Virtex-5 FPGA roughly doubles both the throughput and the maximum frequency.

Sharif et al. [60] reported results for Grøstl on four different FPGA types in 2011. For the Virtex-5 they reported 1,627 slices (without Block RAMs) and 1,141 slices (using 18 Block RAMs). In the same year, Kerckhof et al. [33] presented an implementation that needs only 343 slices on a Spartan-6 and 260 slices on a Virtex-6 FPGA (without using any Block RAMs or DSPs). This design exhibits a maximum frequency of 240 and 280 MHz, respectively depending on the FPGA used. Same applies for the throughput as it features either 548 Mbps or 640 Mbps. In 2012, Kashif et al. [37] presented an efficient hardware design for Grøstl providing a good trade off between area and throughput. The area occupied on a Virtex-5/Virtex-6 FPGA was reported with 1,419/1,467 slices. Regarding throughput 6.20/9.62 Gbps were achieved.

The state-of-the-art regarding low-area implementations of Grøstl targeting FPGAs was presented at the third and last SHA-3 conference. Kaps et al. [30] reported two designs, one using Block RAMs and one without where all Block RAMs were exchanged through distributed RAM. The latter variant needs 357 clock cycles to compute the message digest for a message consisting of 512 bits of data. They furthermore listed results for Xilinx Spartan-3 and Spartan-6 as well as for Xilinx Virtex-5 and Virtex-6 FPGAs. The logic-only version occupies on a Spartan-3 xc3s50-5 766 slices and achieves a maximum throughput of 97.9 Mbps. On a Spartan-6, Virtex-5 and Virtex-6 FPGA 230, 313 and 263 slices are required.

While there exist several papers that analyze the combination of different block ciphers and hash functions (mostly combining MD5 with SHA-1, e.g., [6, 14, 24, 63]), there exist only one publication that focuses on the combination of AES and Grøstl on FPGA platforms. Järvinen [23] analyzed various resource-sharing techniques to reduce the area requirements for an Altera Cylcone III. Their smallest design needs 12,387 Logic Cells (LCs) whereas AES takes an overhead of about 2.5 %, i.e., 300 LCs.

A combined ASIC version of AES and Grøstl has been not reported so far. To the best of our knowledge this master thesis presents the first taped-out combination of AES and Grøstl.

# Chapter 6

# Asynchronous Interfaces

Complex systems—e.g., a system on chip (SOC)—can no longer be designed in one step. Therefore, a modular achitecture is inevitable. As the various subsystems do not necessarily have to be operated with the same clock frequency, problems with data exchange can occur. The reason for is that the possibility of synchronizing the clock is not always given. In order to understand the arising problems, an introduction to synchronous digital circuits and basic terms used within is given. Afterwards, the threats of data exchange between subsystems where no clock information or synchronization is given will be addressed.

## 6.1 Synchronous Digital Circuits

In the field of digital circuits two different logic variants exist. The first variant named combinational logic contains only components where the output changes immediately with the input. Examples for are XORs, ANDs and the like which take the input and deliver the appropriate output. Sequential logic instead is built upon elements which are able to store information, e.g., flip flops or latches. The output of such storage elements changes therefore only at the positive clock edge in case of a positive edge-triggered one-phase clocking scheme. Nowadays, many of such schemes exist. Chapter 6 of the book "Digital Integrated Circuit Design" [28] gives a deeper insight into the various possibilities of operating a digital circuit. This chapter will concentrate on the edge-triggered one-phase clocking scheme only as shown in Figure 6.1. This means that the output of a flip flop changes only at the positive edge of the clock signal.

The clock distribution in such systems is difficult to establish as the clock signal should arrive at each component at the same time. Due to physical and process-technical reasons this cannot be guaranteed. Also the humidity, ambient temperature, local on-chip temperature trend, constant supply voltage and the like influence this behaviour. Therefore, tools have been developed to hold the difference between point-of-time arrivals at various components as low as possible. This aspect is also called **clock skew** and strongly influences the maximum clocking rate. Another fact regarding the clock arrival times is called **clock jitter** and describes the local clock deviation at a single component from cycle to cycle.

- **Clock skew** - Deviation of clock arrival times at various components.
- **Clock jitter** - Deviation of clock arrival times at a single component (deviation between consecutive clock cycles).

Storage elements like flip flops, latches and the like must be provided with a stable and constant clock in order to gain a reliable operation. In addition, a few more terms must be introduced. The most important ones for storage elements are the **setup time** and the **hold time**. Through these times a window named data-call window is being stretched around the clock cycle. During that interval no change of the input value at the receiver flip flops is allowed. Otherwise, a malfunction of the storage element—unpredictable output or extended time until output has been settled—could occur. This input value is further provided by the output of the transmitter flip flop. As both are triggered with the same clock it is important that the hold-time condition of the receiver is fulfilled. Therefore, two more terms must be introduced, namely the **contamination delay** and the **propagation delay**. Both times target the signal propagation in combinational circuits only.

Figure 6.3 presents, for example, such a combinational network. The functionality is a two-input AND function built upon NOR gates. Both inputs can change either at an arbitrary moment in time or at the positive clock edge if coming from the output of a storage element like a flip flop. The assumption for the following explanation is that both inputs to the combinational network are gained through outputs of storage elements triggered with the same clock. Depending on the clock skew, clock jitter, length of the signal line and further the specific properties of each storage element, changes in the storage elements output very probably do not arrive at the same moment of time at the inputs to the combinatorial network. The components of the combinatorial network exhibit also different properties. Further, the length of the signal lines must be not equal. As already the inputs to the combinatorial network do not exactly change at the same moment in time the signal variation is enhanced. As signal lines can be combined over combinatorial logic elements and the fact that changes to their inputs can occur at different moments in time an output of the combinatorial network can change more than once per clock cycle.

With this knowledge the terms contamination delay and propagation delay are finally ready to be explained. The contamination delay is the time from the positive clock edge until the input of the receiver's flip flop changes the first time. This time period depends on the network between the transmitter and the receiver (each element on a network's signal line contributes to the contamination delay). A change in the output of the transmitter's flip flop shows therefore a delayed impact on the input of the receiver's flip flop. This time is exactly what is understood as contamination delay. The propagation delay is the time until the output of the combinatorial network—input of the receiver's flip flop—has settled. The time in between the first change of the input to the receiver's flip flop and until it has settled is critical as there changes of the signal are allowed to occur which implies that the data-call window is not allowed to overlay with this area. The inverse time on the other hand is called data-valid window as no changes at the receiver's input occur. These terms are also presented in detail in the Anceau diagram in Figure 6.2.

- Setup time ($t_{su}$)
  Time before the positive clock edge the input is not allowed to change.
- Hold time ($t_{ho}$)
  Time after the positive clock edge the input is not allowed to change.
- Contamination delay ($t_{cd}$)
  Time from the positive clock edge to the first change of the receiver's input.
- Propagation delay ($t_{pd}$)
  Time from the positive clock edge to the last change of the receiver's input.

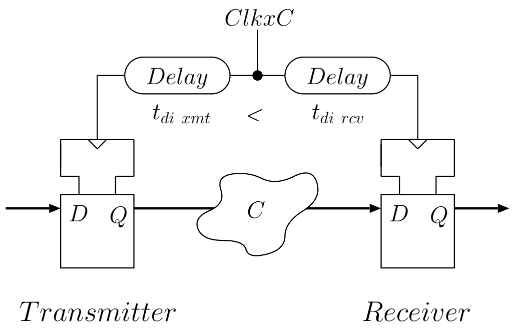


Figure 6.1: Edge-triggered one-phase clocking scheme with included clock skew.

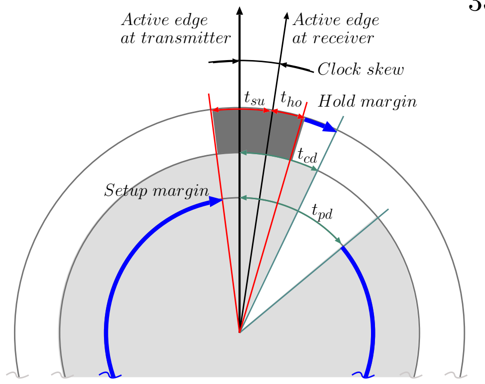


Figure 6.2: Anceau diagram of the edge-triggered one-phase clocking scheme.

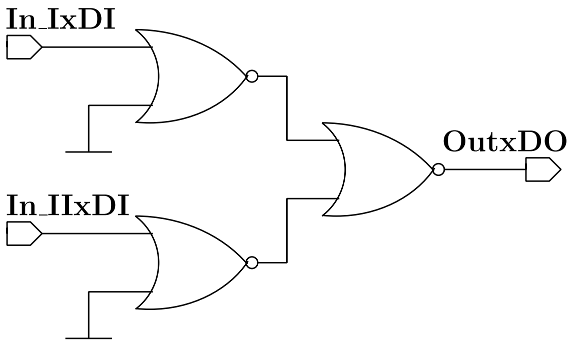


Figure 6.3: Combinatorial logic for a two-input AND function built upon NOR gates.

- Data-call window
  Time interval spanned up through the setup time and the hold-time condition (time where the receiver's input must be constant).

- Data-valid window
  Time interval spanned up through the contamination delay and the propagation delay (time where the receiver's input is constant).

Each digital circuit must be designed with keeping the timing analysis in mind which can be fulfilled with an Anceau diagram. For a better understanding a closer look at the Anceau diagram of Figure 6.2 is given. The time is propagating clockwise with the circumference as reference to one clock cycle. As first step both clock triggers—one for the transmitter and one for the receiver flip flop—must be registered. Afterwards, the setup time and hold time—values given through the characterization of the manufacturer—of the receiver's flip flop must be entered around its active clock edge. Therefore, the data-call window is implicitly stretched represented through the dark-gray segment of the Anceau diagram. Afterwards, the contamination delay—encompassing all delays given on the shortest way from the transmitter to the receiver—and the propagation delay—summed up delays on the longest path—are entered starting each at the active clock edge of the transmitter and forming so the data-valid window represented through the light-gray segment. The white segment in between forms the time interval where the data input of the receiver is not constant which implies that a valid operation can be guaranteed only if the data-call window is fully encompassed by the data-valid window as for instance in the example in Figure 6.2.

In Figure 6.1 a combinational network is included between the receiver and the transmitter flip flop. On the one hand it is a positive effect for the hold time as the contamination delay gets increased but on the other hand it reduces the maximum frequency as also the propagation delay increases. In the case of lag elements—various flip flops connected together in series—where no combinatorial networks are available the characteristics of the chosen flip flops must be inspected carefully as the contamination delay must be larger than the hold time.

## 6.2 Asynchronous Interfaces for Synchronous Digital Circuits

In Section 6.1 an introduction to synchronous digital circuits was given. A bigger problem depicts the communication between systems or subsystems where no information about the clock cycle is shared. This section dedicates to asynchronous interfaces for synchronous digital circuits and the problems coming with them. Additionally, it addresses how to resolve and mitigate these problems. For easing the description the clock-distribution network is seen as ideally. Therefore, neither clock jitter nor clock skew is considered.

### 6.2.1 Inconsistent Data

Figure 6.4 shows the basic hardware organization of a single-edge triggered one-phase system similar to the one in Section 6.1. For easing the analysis the delay components one to four combine all the delays—flip flop delay as well as connection-line delay—between the receiver and the transmitter flip flop. Furthermore, as no combinational network is included the contamination delay is equal to the propagation delay. With these simplifications and the single-clock domain the timing analysis can be applied very easily. In addition to the Anceau diagram presented in Figure 6.6 also the timing diagram is given in Figure 6.5. In both diagrams it is easily recognizable that the data is taken over in a correct manner.

Now the system is split up into two clock domains as shown in Figure 6.7. Both subsystems are single-edge triggered one-phase systems where the right unit (clock domain II) contains a plain bit-parallel synchronization. The difficulty for exchanging data is that no clock information is shared between the two subsystems and therefore successful transferring of data is not guaranteed. If this fact gets overlooked troubles regarding the validity of the data received and stored in subsystem two can occur as presented in Figure 6.8. Two times the data is taken over at moments where the data-call window is not fully encompassed in the data-valid window. The first time this condition gets violated the data stays constant all over the data-call window thus no data inconsistency can be observed. During the second time $DataxD[1]$ changes during the data-call window and therefore a crossover pattern is taken over. Furthermore, as the input to the flip flop stays not constant over the setup and hold time a metastable behaviour could occur manifesting in intermediate voltages or extensive delays for settling to a valid output value.

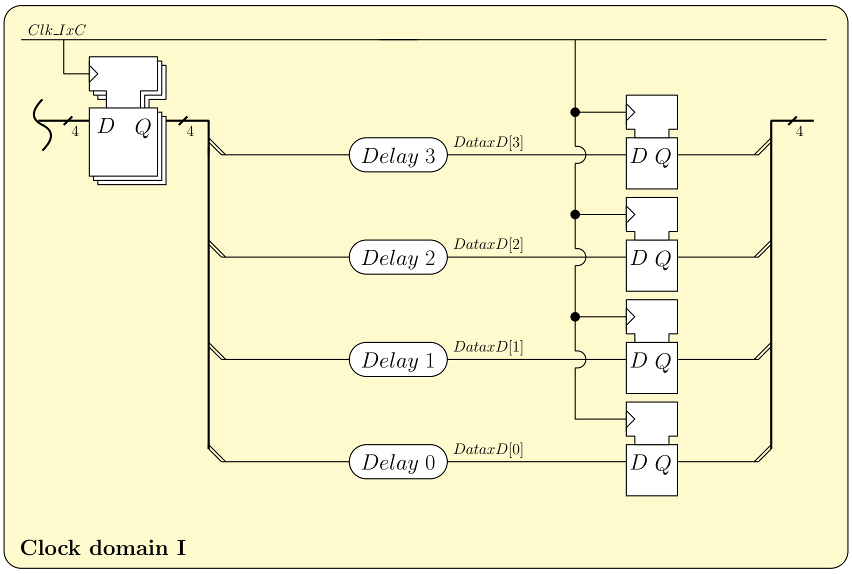


Figure 6.4: Basic hardware organization of a single-edge triggered one-phase system.

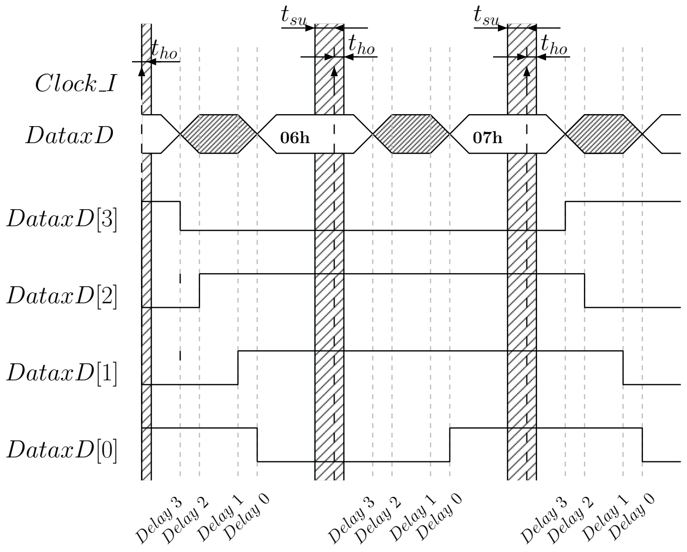


Figure 6.5: Timing diagram for a single-edge triggered one-phase system.

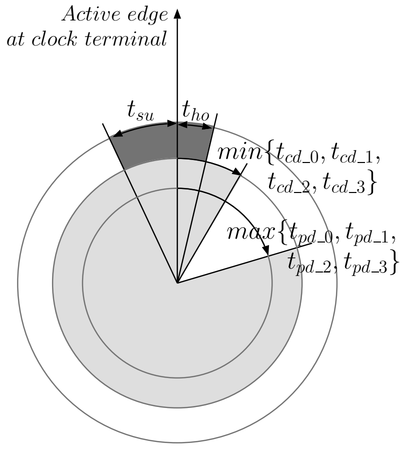

Active edge at clock terminal
$t_{su}$
$t_{ho}$
$min\{t_{cd\_0}, t_{cd\_1}, t_{cd\_2}, t_{cd\_3}\}$
$max\{t_{pd\_0}, t_{pd\_1}, t_{pd\_2}, t_{pd\_3}\}$

Figure 6.6: Anceau diagram of a single-edge triggered one-phase system.

### 6.2.2 Measures Against Inconsistent Data

In order to prevent taking over inconsistent data various methods have been developed. One is called unit-distance coding where each change of the data is at maximum one step in either direction (data value changes only by one), which limits the applicability to acquisition of position, angle encoder and such like. Furthermore, a data converter is necessary to transform the binary representation into a coding scheme where for each step only one bit changes which is equal to a hamming distance of one. Another possibility

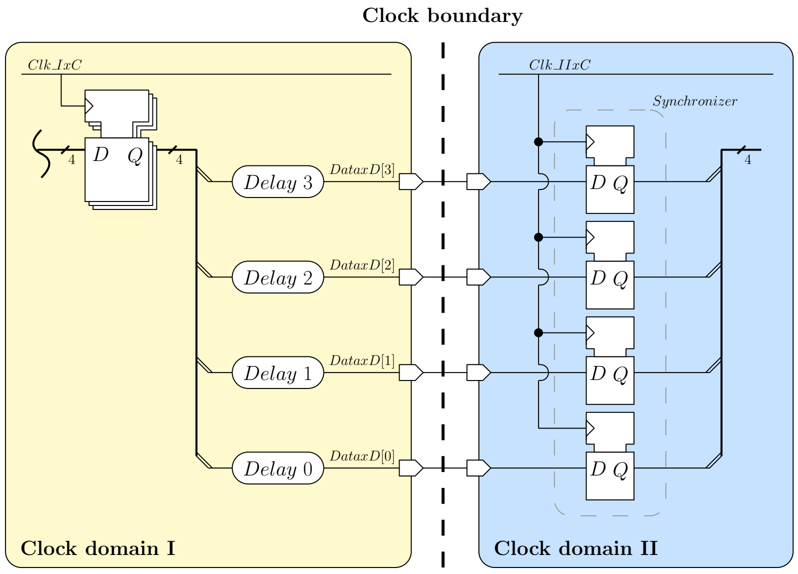


Figure 6.7: Basic hardware organization of two independent single-edge triggered one-phase systems.

would be the suppression of crossover patterns. Therefore, the data is only taken over if in two consecutive cycles the data stays constant. Advantage compared to the previous method is that the data must not be converted into another representation. Drawback on the other hand is the delayed data acquisition. Both approaches allow that inconsistent data enter the receiver's circuit. This contrasts with handshaking protocols where the updating—transmitter's circuit—and sampling of data—receiver's circuit—is regulated by control lines. For a full handshaking two of them are required: **request REQ** and **acknowledge ACK**. Partial handshaking instead requires only the request control line. This section targets only at full handshaking protocols, namely the two-phase and the four-phase handshaking. Further details about partial handshaking can be found in Chapter 7 of the book "Digital Integrated Circuit Design" [28].

For easing the description of the two/four-phase handshaking Figure 6.10 presents the basic hardware organization. Both circuits contain a finite state machine controlling the handshaking procedure. Furthermore, each of these circuits contains additionally a scalar synchronizer subcircuit which is required for storing the state of the control line and to prevent metastable signals entering the circuits.

Next, the sequence of the four-phase handshaking is described by referencing to Figure 6.11. In order to transfer valid data the transmitter must first rise the request line. The receiver on the other side listens as long as it recognizes this event which is represented by the first cycle of the request line. At the next positive clock edge of clock domain II—both systems are again single-edge triggered one-phase systems—the receiver takes over the data and additionally rises the acknowledge line to signal the transmitter that the data has been successfully taken over. Now the same applies for the transmitter as before for the receiver, namely that it listens as long as it recognizes that the acknowledge signal has changed represented by the first circle on the acknowledge line. At the next positive clock edge of the first clock domain the request line is set back to its original state. The receiver again waits for this event (second circle on the request line) and resets with the next clock of its clock domain the acknowledge signal. Now both control lines are back

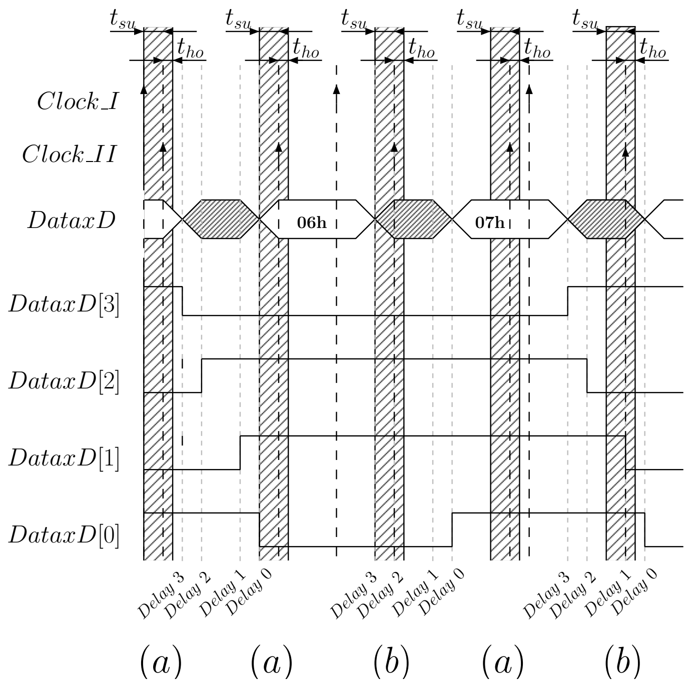


Figure 6.8: Timing diagram for the two independent single-edge triggered one-phase systems.

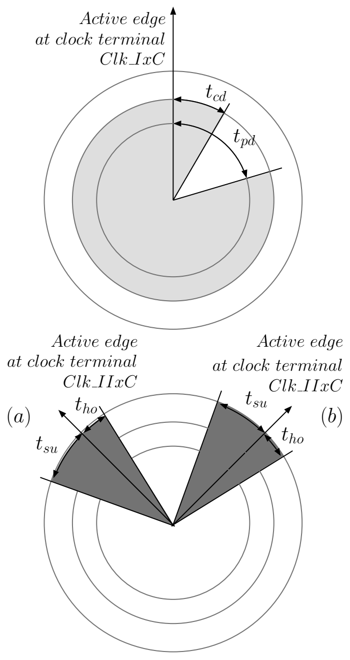

Active edge at clock terminal Clk_IxC
$t_{cd}$
$t_{pd}$
Active edge at clock terminal Clk_IIxC
Active edge at clock terminal Clk_IIxC
(a)
(b)
$t_{su}$
$t_{ho}$
$t_{su}$
$t_{ho}$

Figure 6.9: Anceau diagram for the two independent single-edge triggered one-phase systems.

in their original state. After the transmitter recognizes the original states of the control lines (second circle on the acknowledge line) it applies with the next clock cycle of clock domain I the next valid data to be transferred and starts the next transaction.

Taking a closer look at this scheme shows that it is not optimal regarding time consumption as both control lines must be set back to their original state after the data has been taken over. The two-phase handshaking protocol resolves this drawback—next valid data is applied after recognizing that the data has been successfully taken over—with the impact of a slightly higher area consumption as a more complex finite state machine is required. The time for one transaction therefore can be cut in half compared to the four-phase handshaking as the control signals must not be set back.

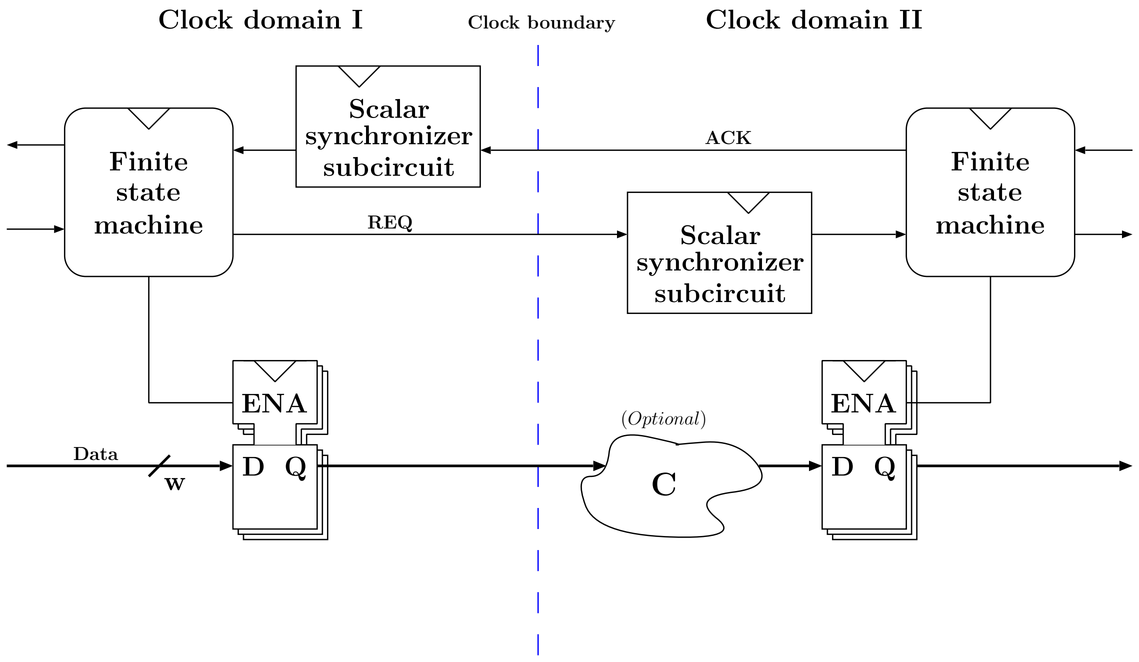


Figure 6.10: Basic hardware organization of four-phase handshaking.

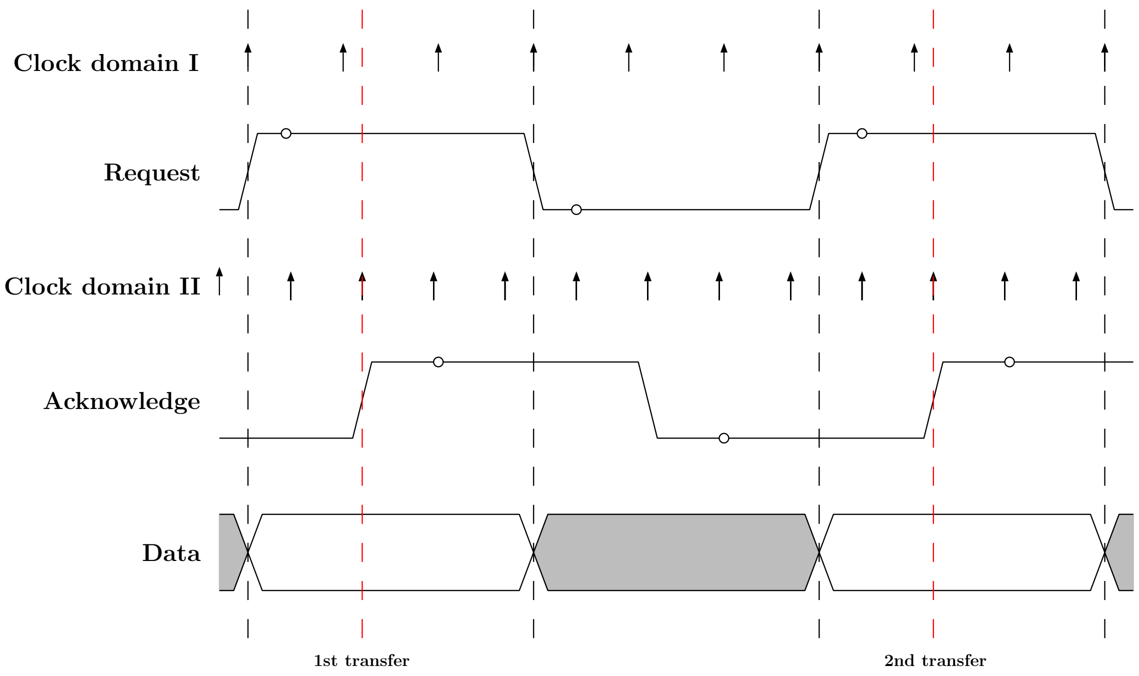


Figure 6.11: Representative timing diagram of four-phase handshaking.

# Chapter 7

# Basics on Xilinx FPGAs

FPGAs in general are devices that feature a very regular structure of various logic elements. The main components are flip flops with a combinational network in front in order to realize the desired functionality. Furthermore, these components can be linked together to build more complex functions. The input width of these LUTs depends on the FPGA's level of complexity and is therefore strongly addicted from the chosen FPGA family. In most of the cases the logical switches and storage elements are realized through Static Random Access Memory (SRAM) storage cells which lose their content when switching off the power. Devices built upon these cells require an initial phase—i.e., a configuration process—during which the LUTs and logical switches are configured. This information is stored typically in a special Flash ROM.

The first company that introduced this kind of programmable devices was Xilinx. Nowadays, Xilinx is the leader regarding FPGA devices. The second most important manufacturer is Altera. Both offer a broad spectrum of various FPGAs starting with low-cost devices and ending up with high-tech devices. Therefore, all areas of application can be covered. The differences between the various FPGA families of each manufacturer are mainly the number and complexity of available logic cells. In respect of the manufacturers peculiarities various minor and major differences exist. One example for a minor difference is the clock divider structure—creating from a reference (input) frequency a desired output frequency—which is done on the one hand via delay-locked loops (DLLs) for Xilinx FPGAs and on the other hand by phase-locked loops (PLLs) for Altera FPGAs. For the list of major differences the routing structure is noteworthy to be mentioned. As this work is partly based on a Xilinx Spartan-3 XC3S400 all further explanations target this special FPGA device [13].

## 7.1 General Structure

The general structure of a Xilinx Spartan-3 FPGA is shown in Figure 7.1. Around the internal logic all the Input/Output Blocks (IOBs) are located controlling the data flow between the I/O pins and the internal logic. Furthermore, the IOBs can be configured for bidirectional data flow and 3-state operation, respectively. Main part of the internal logic are the Configurable Logic Blocks (CLBs) which contain storage elements—can be used as flip flops or latches—and combinational logic networks in order to implement logical functions. These networks are built from RAM-based LUTs. The Xilinx Spartan-3 XC3S400 contains in sum 896 of these CLBs arranged in an array of 32 rows and

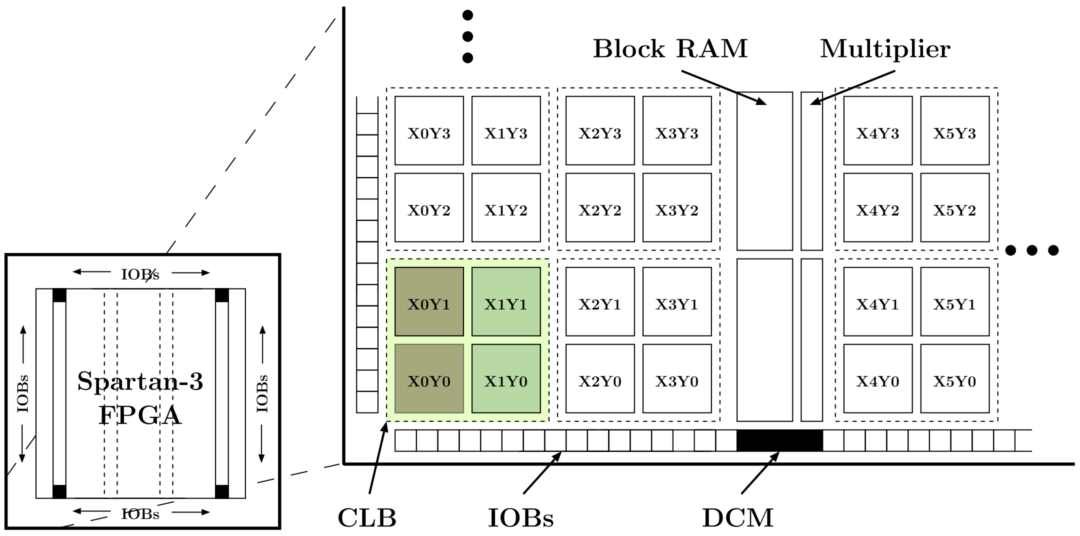


Figure 7.1: Arrangement of CLBs and Slices within a Xilinx Spartan-3 FPGA.

28 columns. Further details on CLBs are given in Subsection 7.1.1. In order to provide various clock signals—potentially required for the internal logic—generated from reference signals this special Xilinx FPGA features four Digital Clock Managers (DCMs). These are fully digital solutions for distributing, delaying, multiplying, dividing and phase shifting of clock signals. Furthermore, 16 dedicated multiplier blocks exist which accept two 18-bit binary numbers and deliver the product as output. Last but not least another variant of storing data namely Block RAMs (BRAMs) is provided. In addition to distributed RAM where the maximum storage space is 56 Kb—storage elements of the CLBs are connected together—another 288 Kb of data can be stored in BRAM. More details on both storage variants are given in Subsection 7.2.1 and Subsection 7.2.2. Numbers and details for the other devices of the Spartan-3 family can be looked up in the Spartan-3 Generation FPGA User Guide [68]. Same applies for all components as only CLBs and Slices will be explained in detail. The reason for this is that this thesis is concerned with low-area and low-memory footprints, respectively where knowledge about the storage possibilities of FPGAs is necessary.

### 7.1.1 Configurable Logic Block (CLB)

CLBs represent the major blocks for implementing sequential as well as combinational circuits. Figure 7.2 shows the structure for a CLB of the Spartan-3 XC3S400 in detail. It consists of four slices interconnected with each other. In addition these slices are grouped in pairs named **Left-Hand SLICEM** and **Right-Hand SLICEM**. Difference is that the former pair has two additional functions, one for storing data using distributed RAM and one for shifting data with 16-bit registers, or in short Shift-Register Logic (SRL). Both functions are explained in detail in Subsection 7.2.2 and Subsection 7.2.3.

### 7.1.2 Slice

Each slice has the following elements in common: two logic function generators, two storage elements, wide-function multiplexers, carry logic and arithmetic gates. The two

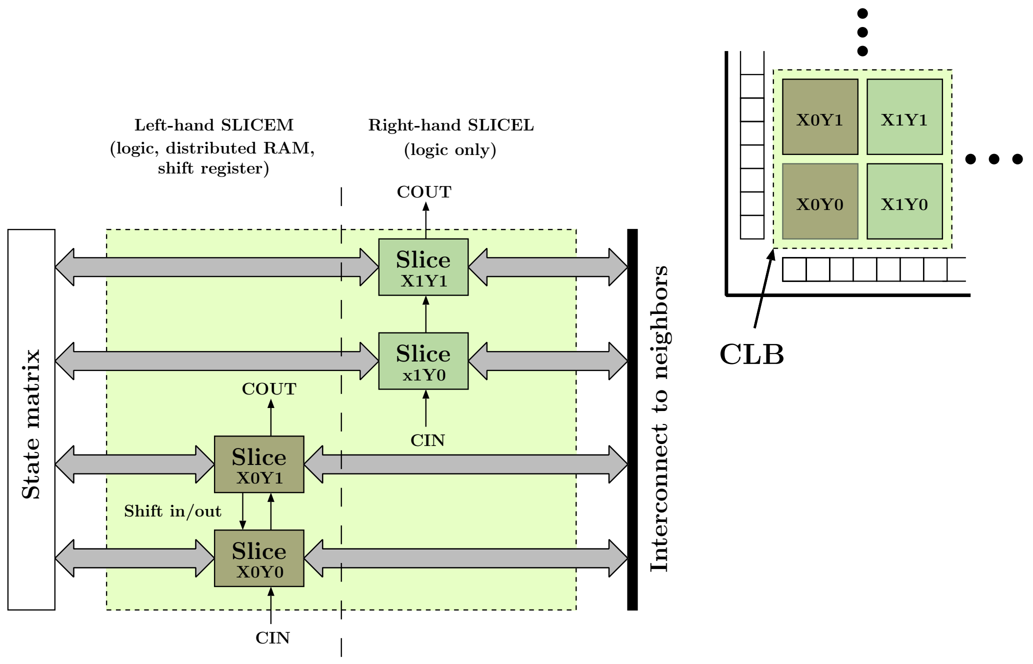


Figure 7.2: Arrangement of Slices within a CLB.

logic function generators are realized through LUTs and build the main resource for implementing logic functions. In some cases the functional complexity reached with these elements is not enough and must therefore be extendable. A solution is given over the wide-function multiplexer which combines LUTs to realize more complex logic functions. This is even more important on low-cost devices as there the complexity—size of LUTs—is quite limited. Sequential circuits can be realized with the two storage elements which can be configured as either D-type flip flop or level-sensitive latch. The carry chain, together with various dedicated arithmetic logic gates represent a powerful combination in order to support fast and efficient implementations of mathematical operations. Summarizing, with these components, logic, arithmetic and ROM functions can be realized.

## 7.2 Memory Variants for Spartan-3 FPGAs

Hardware designs are often limited by their required storage size. In order to overcome these limitations and to address further challenges like high-speed designs, low-area footprint and the like various possibilities are provided by the manufacturers. Three kinds of memory are therefore addressed in the next subsections: Block RAMs, distributed RAM and SRL16.

### 7.2.1 Block RAM (BRAM)

Generally, state-of-the-art FPGAs feature—independent of the manufacturer—Block RAMs which are simply large on-chip memories. Focusing on the Xilinx Spartan-3 FPGA family further configurations can be applied to use this storage as RAM, ROM, FIFOs, large look-up tables, shift registers, data width converters and circular buffers. Each of these special modes of operation additionally features various data widths and depths and can

Table 7.1: Details about the Block-RAM sizes for Xilinx Spartan-3 FPGAs.

| **Device** | **RAM Columns** | **RAM Blocks per Column** | **Total RAM Blocks** | **Total RAM Bits** |
|---|---|---|---|---|
| XC3S50 | 1 | 4 | 4 | 73,728 |
| XC3S200 | 2 | 6 | 12 | 221,184 |
| XC3S400 | 2 | 8 | 16 | 294,912 |
| XC3S2000 | 2 | 20 | 40 | 737,280 |
| XC3S5000 | 4 | 26 | 104 | 1,916,928 |

therefore be adapted for nearly every application. In order to ease accessibility and usage, Xilinx provides so-called Core Generators where the developer is able to generate the modules by a graphical wizard which leads in a convenient way through the whole configuration. Another possibility targeting experts only is the configuration over VHDL or Verilog instantiations. From experience it showed that it is more practical to use the graphical user interface as from time to time updates of the libraries, design tools (Xilinx ISE) and the Core Generator in special are provided. Newer versions of the Core Generator allow to implicitly update the components generated with an older version. Therefore, a straightforward way of updating the whole toolchain in an appropriate manner is given.

In order to get an overview of the available storage gained through the usage of BRAMs Table 7.1 summarizes the numbers for the various Xilinx Spartan-3 FPGA variants. These numbers are taken from the application note for the Spartan-3 FPGA family [65].

Figure 7.3 gives an overview of the BRAM locations for the Xilinx Spartan-3 XC3S400 device. As stated in Table 7.1 this special type features two columns with each having 8 BRAMs. Therefore, beside the CLBs in sum an additional storage size of 294,912 bits is available. Further details on the various modes of operation and the accessibility can be found in the application note for Xilinx BRAMs [65].

Summarizing, BRAMs are a welcome improvement regarding storage-size extensions and their accessibility. Nonetheless, also drawbacks exist as for example they are not recommended for high-speed designs. The reason for this is their fixed location which implies longer signal lines. This differs to distributed RAM where the storage can be placed directly beside the logic. More details on distributed RAM follow in the next subsection.

### 7.2.2 Distributed RAM (LUT RAM)

Beside Block RAMs Xilinx FPGAs feature another possibility of storing data called distributed RAM where the LUTs of slices are reused as 16 x 1-bit synchronous RAM. Limitation for this mode of operation is the fact that only slices in the SLICEM group are supported. A CLB of a Spartan-3 FPGA contains up to 64 bits of single-port RAM and 32 bits of dual-port RAM, respectively. Write operations are synchronous which contrasts to asynchronously read operations. Should a synchronous read operation be inevitable, the register associated with each LUT can be used to resolve this problem. Furthermore, each 16 x 1-bit RAM is combinable with other RAM types of same kind and is therefore cascadable for deeper and/or wider memory applications. The only drawback is a minimal timing penalty through specialized logic resources.

In order to create primitives such as single-port and dual-port RAMs, the Xilinx Core Generator is used. The advantage is that it outputs already optimized distributed RAMs

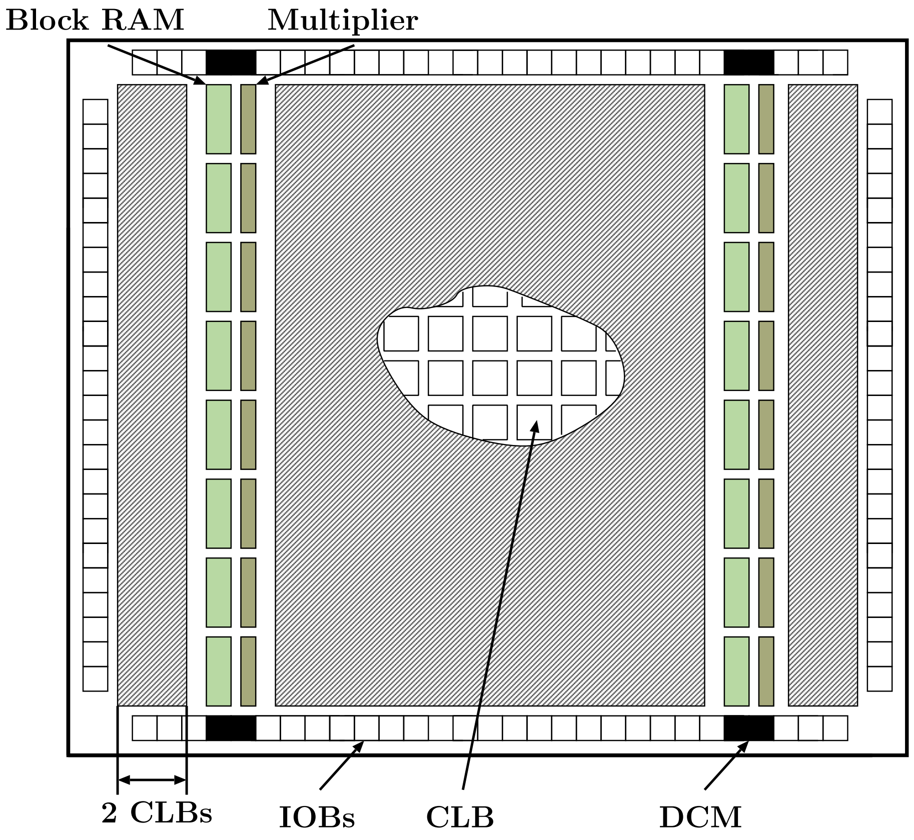


Figure 7.3: Arrangement of Block RAMs for Xilinx Spartan-3 XC3S400 FPGA.

targeting the desired FPGA structure. Summarizing, distributed RAM is fast, localized and ideal for small data buffers, FIFOs, or register files [66].

### Single-Port/Dual-Port RAMs

Dual-port RAMs are a special kind of RAMs where a read and a write operation can be applied simultaneously on the same storage space. Xilinx FPGAs handle this by allocating the double of the size as usually required. For example, if a dual-port RAM with 16 x 1-bit memory is desired, the Xilinx Core Generator allocates two LUTs both of size 16-bit. As Figure 7.4 shows two read outputs are available but only one write input. The reason for this is that the storage space gets mirrored and the write operation is applied on both of them. Hence, both address spaces contain always the same data which enables the functionality of an independent read operation.

## 7.2.3 Shift-Register Logic (SRL16)

Figure 7.5 shows the basic configuration of a slice's LUT which is realized through common flip flops. It represents an arbitrary logical function which is expressible through a truth table fitting in the size of the LUTs and their appropriate input width of four bits. An example is given in Figure 7.5 a.) where the truth table of a four-input AND function is stored in the LUT. The input variables to the logical function are given through the address input. The output $D$ represents the result of the four-input AND function with

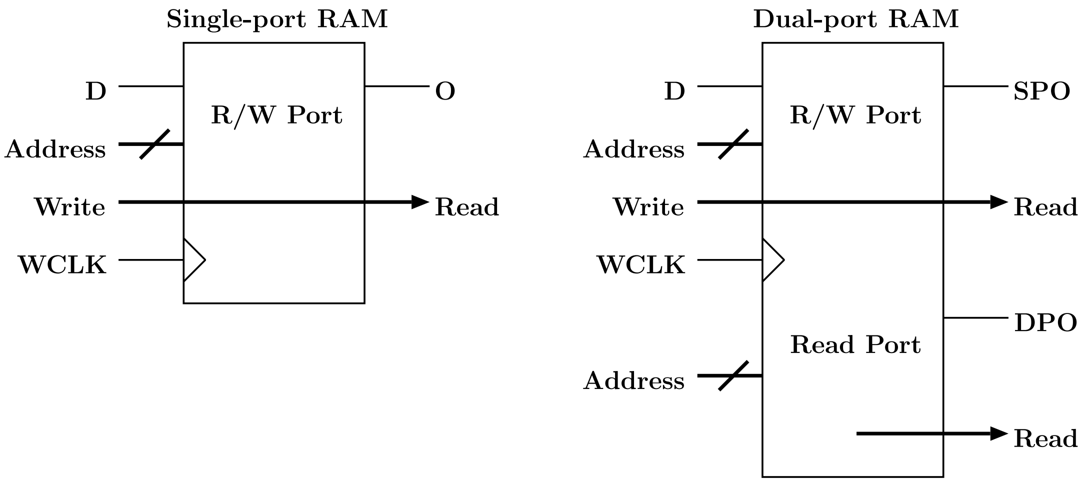


Figure 7.4: Single-port and dual-port distributed RAM.

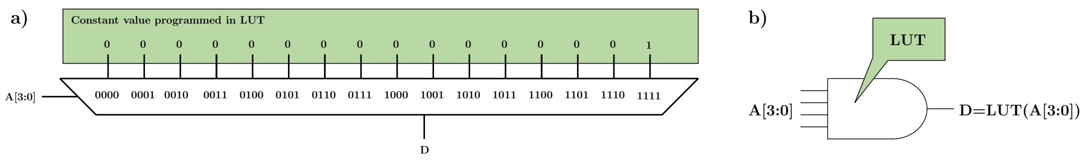


Figure 7.5: LUT modeled as 16:1 multiplexer. Schematic is shown in a) whereas the equivalent circuit diagram is presented in b).

address lines acting as input variables.

The SRL16 mode is similar to distributed RAM as it reuses the LUTs of SLICEMs as 16-bit shift registers. This mode of operation is shown in Figure 7.6. In contrast to the common mode the outputs of the flip flops are connected in addition to the input of the next flip flop. Therefore, a shift-register scheme is achieved. One LUT of the Spartan-3 XC3S400 can contain at most an 16-bit shift register. In this case the output is provided from the last flip flop (Q15). The size can be adapted through a dynamic length adjustment. For this purpose the 16:1 multiplexer can be used as each output of the flip flops is connected to it. In order to achieve an 8-bit shift register the output of the 8th flip flop is chosen through the address inputs. This adjustment is also shown in Figure 7.6.

The structure of such an SRLC16 cell is shown in Figure 7.7. Each LUT is associated with a further flip flop in order to create a synchronous output. If this behaviour is not required than the flip flop can be skipped ending up with a combinational output.

Sometimes it is required to build shift registers exceeding the maximum size of one LUT (16 bits). In that case it is possible to connect more shift registers together in order to achieve the desired size. Such a construction is shown in Figure 7.8. In detail one CLB contains in sum four slices grouped in pairs of two. These are called SLICEM and SLICEL. Only slices in the SLICEM group can be reconfigured to operate in shift-register mode of operation. The others are designated for logic only. Therefore, the maximum shift-register width for one CLB is limited to 64 bits as only four of the eight LUTs can be reconfigured. [67]

With this mode of operation, cost savings of an order of magnitude can be achieved. As the SRL16 mode is mainly automatically inferred by the software tools it is important to

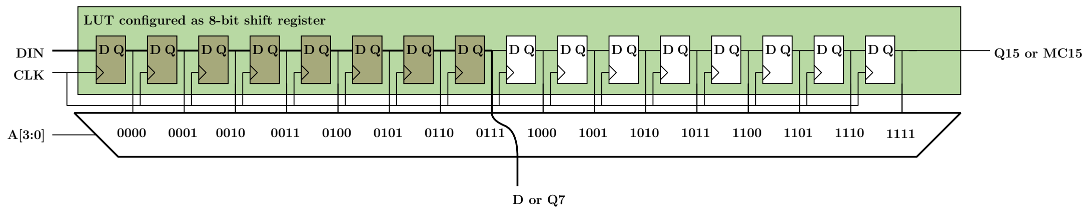


Figure 7.6: LUT configured as 8-bit shift register.

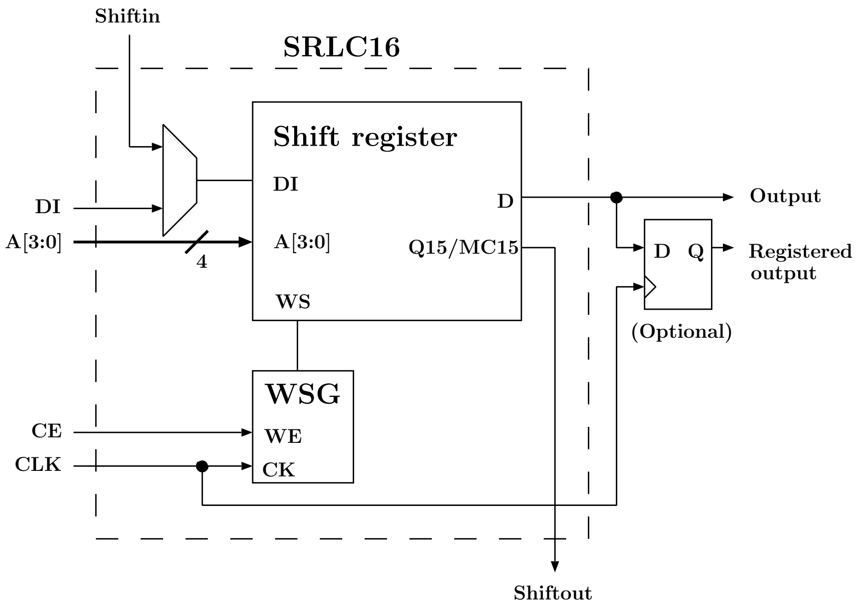


Figure 7.7: Structure of the SRLC16 cell.

write the code in an appropriate manner. For example, the flip flops used to build the slice LUTs have no reset functionality. If code for shift registers is written for ASIC designs and overtaken for FPGA design flows targeting the SRL16 mode, the efficiency of the automatically inferred design is mitigated. The reason for this is the reset functionality as the software tool is not able to decide if it is required out of design reasons or not. Adapting the code snippet can lead to additional cost savings as therefore the software tools should be able to recognize more possibilities to instantiate SRL16 cells. Further details on writing appropriate code for the SRL16 mode can be found in the white paper “Saving Costs with the SRL16E” [7].

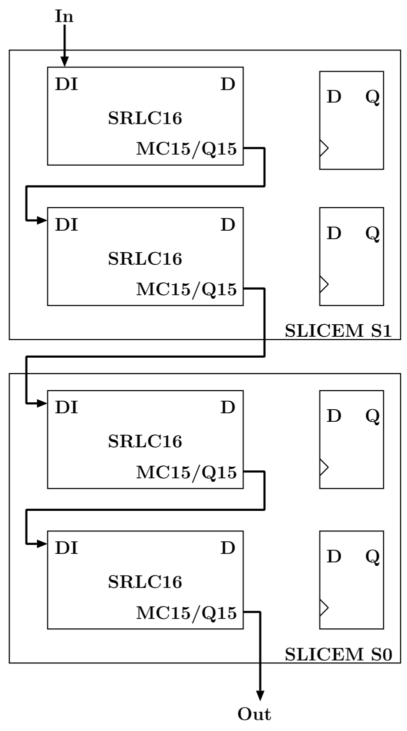


Figure 7.8: Cascading SRLC16 cells inside a CLB in order to extend the shift-register width up to 64 bits.

# Chapter 8

# GrÆStl - a Combined AES/Grøstl Hardware Architecture

This chapter concentrates on the development of a combined AES/Grøstl hardware architecture targeting ASICs and FPGAs. For easing the design flow no technology or platform-dependent components such as RAM macros, DSPs, or Block RAMs are used. All design decisions were affected by keeping the target of a small area footprint—occupied die area for ASICs and number of occupied slices for FPGAs—in mind.

This chapter is organized as follows. In Section 8.1 an examination of various AES and Grøstl variants is presented in order to gain information about the best way to combine AES and Grøstl. In Section 8.2 the taped-out chip named Chameleon containing the cryptographic modules AES-128, Grøstl-224 and GrÆStl is described. The starting point is the top-level module of the chip which handles the communication with the external world over a four-phase handshaking and additionally the data distribution to the various modules. As the focus of this work was the development of GrÆStl and the fact that the single versions for AES-128 and Grøstl-224 are contained in the common architecture, a detailed description is only given for the shared variant. Finally Section 8.3 presents the results for both platforms—ASIC and FPGA—before a comparison with related work is given in Section 8.4.

## 8.1 On the Search for Optimal Resource Management

In order to gain an optimal resource sharing hardware architecture supporting AES-128 as well as Grøstl-224 both algorithms must be analysed regarding their resource requirements. This process is done on an abstract level as for a detailed analysis concrete implementations would be required. For time saving-reasons the architecture is simply reduced to the components and their size regarding the used datapath width and their corresponding cycle count. Details on both algorithms can be found in Section 3 and Section 4. As AES decryption uses similar hardware resources as its counterpart the AES encryption, only the latter is considered during this analysis phase. Noticeable differences would be the higher cycle count for the decryption in order to compute from the master key to the last round key before the actual decryption can be computed and also a slightly more complex $MixColumns$ operation.

To stay on an abstract level, values for the component sizes have to be assumed. Therefore first the size (complexity) for a flip flop was determined with respect to a 2-

input NAND gate. According to the standard-cell library and a 2-input NAND gate a single flip flop exhibits a size of 6.25 GE. Afterwards, the size of registers was computed through a linear approximation. An 128-bit register is therefore of size 800 GE (800 GE = 128×6.25 GE). The size of a *SubBytes* component is well known and was estimated according the paper from Wolkerstorfer et al. [64] to around 300 GE. The *MixColumns/MixBytes* component was evaluated through existing work and an example implementation. The size was so determined to around 200 GE and 400 GE, respectively. As the *MixColumns/MixBytes* component can be used with different output widths a possibility must be given to easily recompute the size in an appropriate manner (the input size must be either 32 bits for AES or 64 bits for Grøstl as it operates always on a full state column). In order to keep the analysis simple the factor required to get from the base width to the target width is used. For example, if the *MixColumns* component with an 32-bit output is required the size is computed to 800 GE (800 GE = 4×200 GE). The assumed values for the various components are given in the following list:

- 128-bit register $\rightarrow$ 800 GE
- 512-bit register $\rightarrow$ 3200 GE
- 8-bit *SubBytes* $\rightarrow$ 300 GE
- 8-bit *MixColumns* (AES) $\rightarrow$ 200 GE
- 8-bit *MixBytes* (Grøstl)$\rightarrow$ 400 GE (as resources can be shared, size is only roughly doubled compared to the AES variant)

Next, various hardware architectures for AES-128 (encryption only) and Grøstl-224 are examined regarding their required resources. AES decryption is not considered. The reason for is that an AES encryption/decryption features a similar structure. Only the sequence of operations is slightly modified and the components used are the inverse as for the encryption. The encryption-only consideration is therefore enough to determine the optimal resource sharing variant.

To stay compliant throughout all variants it is assumed that for AES both the message to be encrypted and the masterkey as well as for Grøstl only the message are already loaded into the system (not constantly applied from external!). Furthermore, for Grøstl only the $P/Q$ computation is considered as it represents the core of the system and the fact that the truncation is based on resources already contained in the $P$-block. Regarding only the hardware resources is not enough as these must be connected together in an appropriate manner. Due to this fact an additional overhead (20 percent of the hardware resources) is added to the overall size.

Starting point for the examination is an AES architecture with full-size datapath. In that case two 128-bit registers are used in order to store the message and the actual round key. The latter is computed in parallel on-the-fly. To avoid latencies caused by computation delays of the round key, the architecture must be carefully designed. Therefore, to the 16 *SubByte* units of the encryption module—substitution step is done in only one cycle—another four are added for the round-key generation. These are needed to deliver the next round key in time. Furthermore, as the datapath width is fully blown up, the *MixColumns* component receives 128 bits as input and must therefore compute immediately all column multiplications. As AES is based on 10 rounds the result is also available within 10 cycles. Next, the datapath width is reduced to 64 and 32 bits which effects in general only the amount of *SubByte* units and the size of the *MixColumns* component as well as the cycle count for one computation. A reduction of the datapath to only 16 bits width results in addition to the previous mentioned effects in another 32-bit register used to store the values of one column. This is required as the *MixColumns* component requires always a full column as input. Besides the additional hardware resource the cycle count is comparable higher to the previous versions. The reason for is that the 32-bit register has to be filled with valid data (cycle count required depends on the datapath used) before the *MixColumns* component is able to compute the corresponding output and updates then the state matrix in an appropriate manner. For one column four cycles (two cycles to fill the 32-bit register in front of the MixColumns unit and another two for updating the state matrix) are required which ends up in 16 cycles as the state matrix encompasses four columns. The smallest variant built around an 8-bit datapath features the same behaviour as the 16-bit version. More details on these variants can be found in Table 8.1.

Similar to the AES examination the Grøstl architecture started with a version that is based on the full datapath width of 512 bits and the two permutations $P$ and $Q$, respectively computed in parallel. This implies two separate instances of the permutations and the need for three 512-bit registers. Two of them are used to store the State of the $P/Q$-blocks whereas the other is used for the intermediate hash value. Due to the full datapath width and the separate instances for the permutations, one message block can be handled in only 10 cycles. Drawback is the immense hardware effort as for example the 64 *SubBytes* instances or the 512-bit *MixBytes* component have to be duplicated. Next step is the merging of the $P/Q$-block into one instance and therefore the need for a sequential computation. On the one hand the hardware effort is so nearly halved but on the other hand the cycle count gets doubled. It has to be noted that still three 512-bit registers are designated although one has been saved due to the fact that the parallel computation of the permutations was altered to a sequential one. Problem is that a sequential computation entails another 512-bit register as the message is required for both the $P$ and the $Q$ permutation (assumption is that the message is not externally applied during the computation itself). The other two 512-bit registers are used to store on the one hand the intermediate hash value and on the other hand the internal state of the permutation. Next the datapath width is reduced to 128 and 64 bits, respectively which effects in general in the same way as before for AES only the amount of *SubByte* units and the size of the *MixBytes* component as well as the cycle count for one computation. Smaller datapath widths require another 64-bit register in front of the *MixBytes* component. The reason for that is that this component operates always on full columns and therefore before a computation can be started the register in front must be filled with valid data. More details on these variants can be found in Table 8.2.

The AT-characteristics for all examined AES designs as well as for all Grøstl designs can be found in Figure 8.1 and Figure 8.2. These plots help to make a decision which versions should be combined in order to achieve the smallest hardware architecture—keeping the efficiency in mind—featuring AES and Grøstl functionality.

**Attention: Consider the different scaling of the axes in Figure 8.1 and Figure 8.2.**

The characteristics for the AES designs show that it is best to use an 8-bit datapath instead of an 16-bit datapath as around 1 kGE of area can be saved. This reduction from around 5 kGE to only 4 kGE represents an area saving of around 20 %. In respect of the Grøstl design a reduction of the datapath width from 16 bits to 8 bits results in an area reduction from around 14 kGE to 13 kGE which corresponds to an area saving of around 7 %. Both datapath width reductions would result additionally in an excessive cycle count penalty. Targeting only low-area consumption it is recommended to use an 8-bit datapath for the stand-alone versions of AES and Grøstl. In view of the combination step and in addition keeping efficiency in mind it makes sense to use for the combined version an 8-bit datapath for AES and an 16-bit datapath for Grøstl. This difference in the datapath width brings beside the faster computation of Grøstl another advantage regarding the usage of the *SubBytes* units. Grøstl with an 16-bit datapath width requires two *SubByte* components which coincides perfectly with an 8-bit AES architecture if the round keys are generated on-the-fly. Reason for is that one *SubByte* unit of the Grøstl design can be reused for the encryption and the other for the round-key generation.

Table 8.1: Resource analysis of various AES hardware architectures.

| **Datapath Width** [bits] | **# Reg.** [bits] | **# SubByte** [Units] | **# MixColumn** [bits] | **Cycle Count** [cycles] | **Size** [kGE] | **Throughput**[1] [Mbit/s] | **AT-product** Gate/(bit/µs) |
|---|---|---|---|---|---|---|---|
| 128 | 2 x 128 | 16(4)[2] | 1 x 128-bit | 10 = 10 x 1 | 13.0 | 12.8 | 1,015 |
| 64 | 2 x 128 | 8(4)[2] | 1 x 64-bit | 20 = 10 x 2 | 9.0 | 6.4 | 1,406 |
| 32 | 2 x 128 | 4(4)[2] | 1 x 32-bit | 40 = 10 x 4 | 6.5 | 3.2 | 2,032 |
| 16 | 2 x 128 + 1 x 32[3] | 2(2)[2] | 1 x 16-bit | 160 = 10 x 16 | 5.0 | 0.8 | 6,250 |
| 8 | 2 x 128 + 1 x 32[3] | 1(1)[2] | 1 x 8-bit | 320 = 10 x 32 | 4.0 | 0.4 | 10,000 |

[1] Throughput given for 1 MHz clock.
[2] The term in brackets represents the number of *SubByte* units for the round-key generation.
[3] *MixColumns* can only be computed from a full column.

Table 8.2: Resource analysis of various Grøstl hardware architectures.

| **Datapath Width** [bits] | **# Reg.** [bits] | **# SubByte** [Units] | **# MixByte** [bits] | **Cycle Count** [cycles] | **Size** [kGE] | **Throughput**[1] [Mbit/s] | **AT-product** Gate/(bit/µs) |
|---|---|---|---|---|---|---|---|
| 512 | 3 x 512 | 2 x 64 | 2 x (1 x 512-bit) | 10 = 10 x 1 | 120.0 | 51.2 | 2,343 |
| 512 | 3 x 512 | 64 | 1 x 512-bit | 20 = 2 x (10 x 1) | 65.0 | 25.6 | 2,544 |
| 128 | 3 x 512 | 16 | 1 x 128-bit | 80 = 2 x (10 x 4) | 25.0 | 6.4 | 3,906 |
| 64 | 3 x 512 | 8 | 1 x 64-bit | 160 = 2 x (10 x 8) | 19.0 | 3.2 | 5,937 |
| 32 | 3 x 512 + 1 x 64 [2] | 4 | 1 x 32-bit | 640 = 2 x (10 x 32) | 15.5 | 0.8 | 19,376 |
| 16 | 3 x 512 + 1 x 64 [2] | 2 | 1 x 16-bit | 1280 = 2 x (10 x 64) | 14.0 | 0.4 | 35,000 |
| 8 | 3 x 512 + 1 x 64 [2] | 1 | 1 x 8-bit | 2560 = 2 x (10 x 128) | 13.0 | 0.2 | 65,019 |

[1] Throughput given for 1 MHz clock.
[2] *MixBytes* can only be computed from a full column.

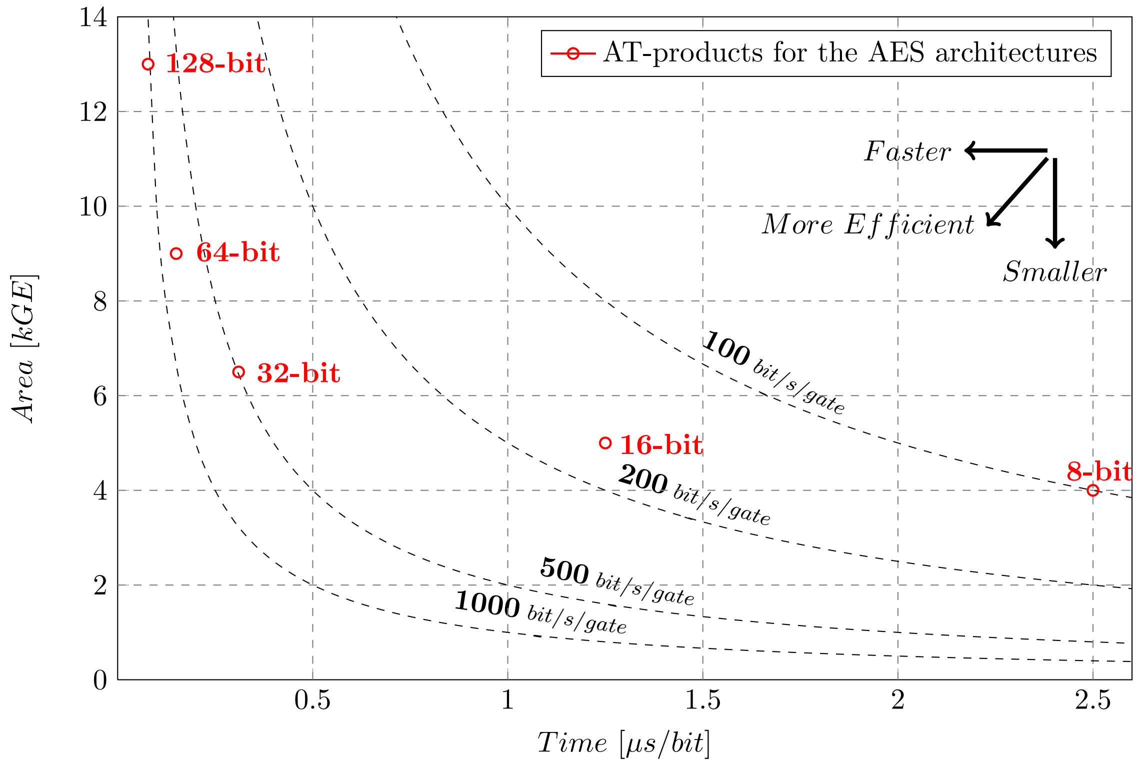


Figure 8.1: AT-characteristics of the various AES architectures.

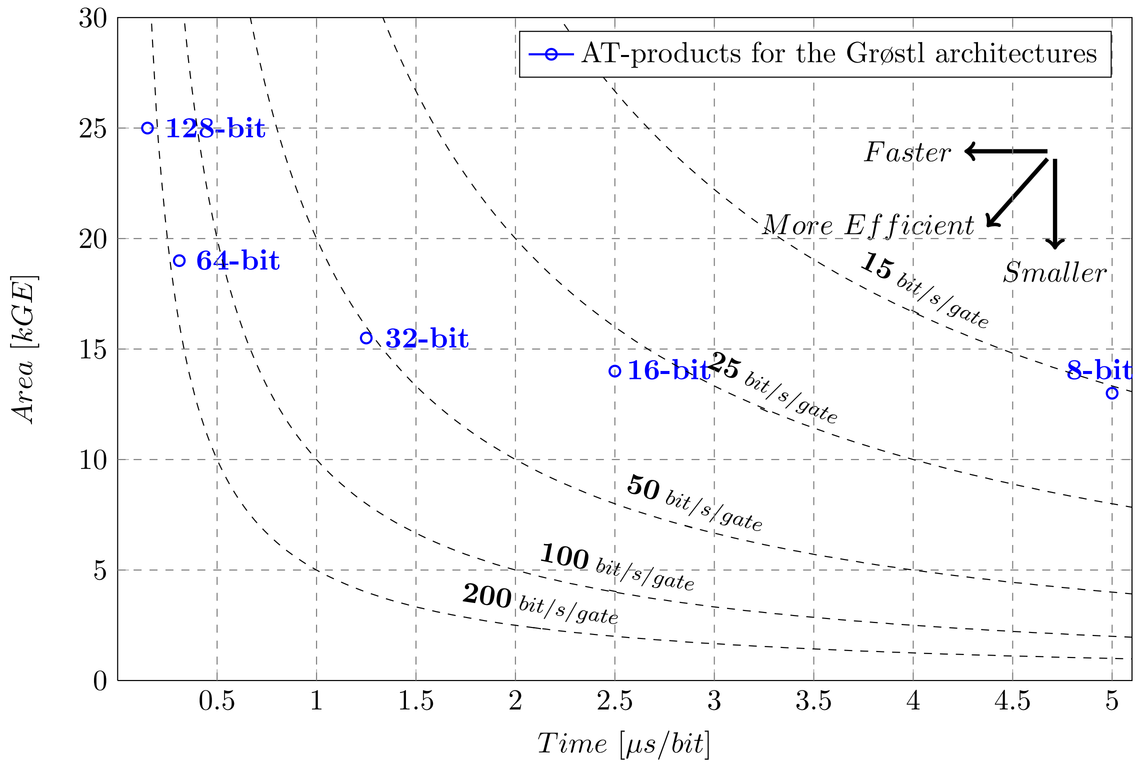


Figure 8.2: AT-characteristics of the various Grøstl architectures.

## 8.2 Hardware Architecture

The hardware architecture was designed to operate in environments containing more than one clock domain. In order to mitigate the problems of asynchronous interfaces described in detail in Chapter 6, a common four-phase handshaking was used as presented in Chapter 7 of the book "Digital Integrated Circuit Design" [28]. Details on the implementation itself and furthermore on an implementation vulnerability—jump to wrong state in Finite State Machine (FSM) makes the interface only operable in a test environment—can be found in Appendix B.4. Due to this implementation flaw the taped-out chip, named Chameleon is not recommended for use in different clock domains. This differs to the FPGA version as there the corrected interface is used enabling so each possible system arrangement. The I/O interface is shared by all cryptographic modules. Changes in the interface effect therefore all cryptographic modules that have it included.

Chameleon was designed as reference platform in order to evaluate in a fair manner the efficiency of combining AES-128 with Grøstl-224. The top layer of the architecture is shown in Figure 8.3. On the first sight it appeals suspicious that no I/O registers can be found. This is due to the fact that they are included in each cryptographic module itself in order to reuse them for the architecture design. Due to that more efficient implementations are possible. Drawback is the need for additional signals ($SelUnitxSI$ and $SelModexSI$) as the interface has no direct access to the I/O registers. The $SelUnitxSI$ signal enables only one cryptographic unit at a time. Due to power saving reasons all other modules are switched off over clock gating. The dedicated signals for this are $EnAESModulexS$, $EnGroestlModulexS$ and $EnSharedModulexS$. As only one component at a time is active the input data can be directly guided through. The $SelModexSI$ is responsible for the handshaking process itself as it controls the number of required data items. Grøstl needs to read 64 bytes whereas for AES in both modes only 32 bytes are required. Due to area-saving reasons, the sequence of input/output data is strictly defined in order to avoid the need for address lines. The I/O registers were therefore designed as shift-registers. Drawback is the need for another signal ($NewInDataxS$) informing the active cryptographic module to store a value applied from external and this only once per data item. After the computation, the result is stored in the output register from which it must be read out. Similar to reading data, a signal is required triggering the component to reveal a data item. This is fulfilled by the $NewOutDataxS$ signal. As more cryptographic modules may be instantiated, the possibility is given that values of the output port get overwritten. In order to avoid this, a multiplexer controlled by the $SelUnitxS$ signal is used, guiding through only values of the active component.

### 8.2.1 Top Layer

GrÆStl has been designed with the aim for a very compact solution that supports both AES-128 and Grøstl-224 in one piece of silicon. Target was a low-resource design (primarily area and power optimized) which features additionally high flexibility so that the design can be applied on both ASIC and FPGA platforms. In order to achieve high flexibility the use of process-dependent technologies like RAM macros or the use of Block RAMs or DSPs on FPGA architectures was avoided. The reason for this is the drawback of these technologies as they might not be available in all CMOS libraries and that they have to be recreated in case of a possible CMOS-process change. Moreover, in case of FPGAs, resources such as Block RAMs or DSPs might be already used by other system

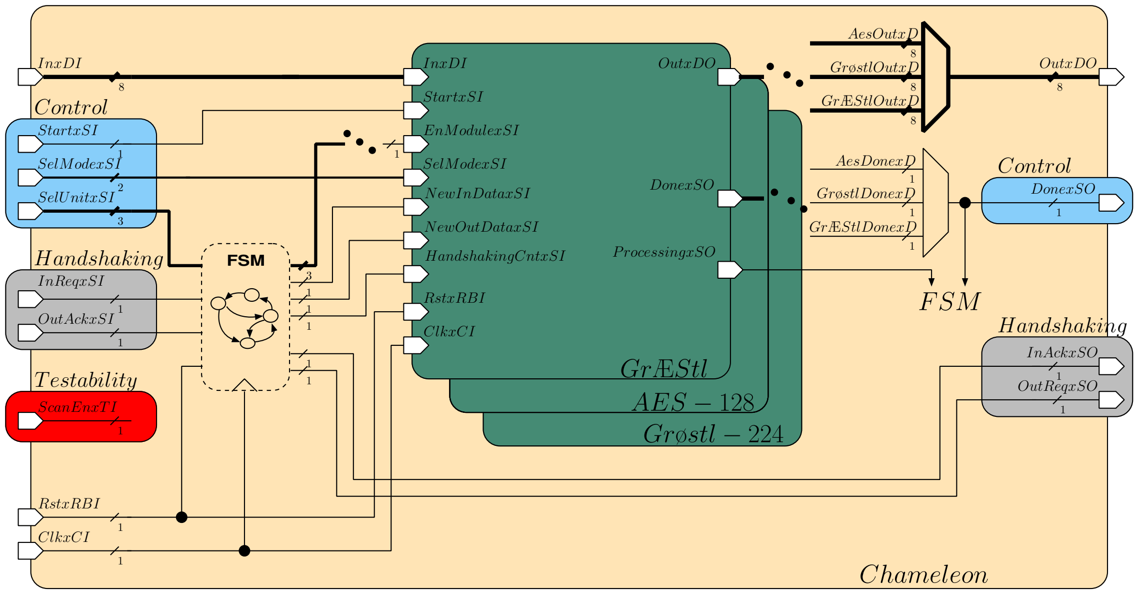


Figure 8.3: Top layer of the Chameleon chip managing the accessibility of the cryptographic modules AES, Grøstl and GrÆStl.

components such that the applicability of the design depends on their availability. In order to avoid those dependencies, the design was based on standard cells and generic hardware components in order to make it very flexible and portable to other platforms.

Since Grøstl needs per se more resources than AES, it is advisable to reuse existing Grøstl hardware components such that the overhead for providing AES functionalities can be kept as small as possible. Therefore, the idea was to efficiently integrate the AES datapath into the one of Grøstl. The 8-bit interface (four-phase handshaking) at the chip's top level that is used to exchange data with the external world was already described in Section 8.2. On that level the data is simply guided through as the I/O registers are directly located in the cryptographic modules.

An overview of the GrÆStl architecture is given in Figure 8.4. The main components are a common datapath (denoted as *Core* unit) that combines most of the round transformations for AES and Grøstl, the I/O shift registers both of size 512 bits and the logic required to achieve the functionality for either AES or Grøstl. In order to keep the area requirements low, the permutations $P$ and $Q$ are computed sequentially instead of computing them in parallel. This reduces the performance of hashing but allows to implement only one shared permutation instance in hardware. The need for the two 512-bit I/O shift registers is compensated by the fact that they are needed anyway. First to store the original message (needed by the second permutation $Q$) . Second to store the output of the first permutation $P$ and the intermediate hash value (in case of mesages not fitting one block), respectively. Additionally, AES decryption can be accelerated by storing the last round key—gained during an AES encryption/decryption operation—and the master key in the output register. At the beginning of an AES decryption a check is being applied if the stored master key fits the one which got applied. In that case the *Core* receives instead of the applied master key the already computed and stored last round key. Therefore the forward round-key generation can be skipped.

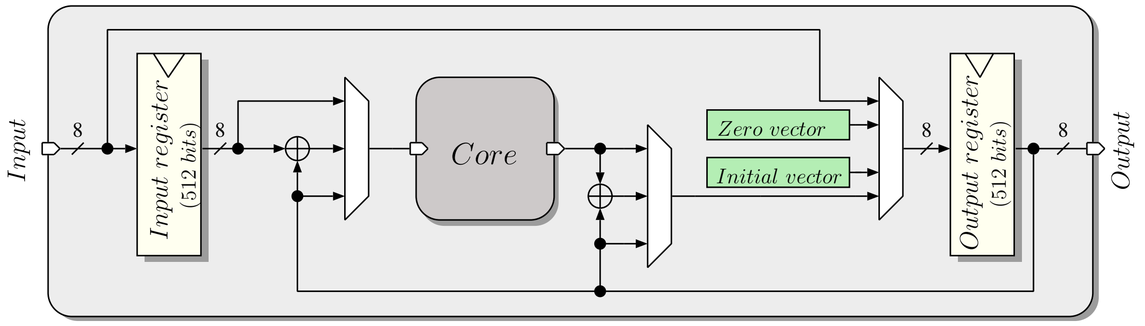


Figure 8.4: Overview of the GrÆStl architecture containing the I/O shift-registers, the Core implementing AES and the permutations $P$ and $Q$ as well as the logical connections required to achieve the desired functionality (AES or Grøstl).

In order to compute an AES encryption/decryption operation the data in big-endian representation—consisting of 16 bytes for the master key and 16 bytes for the data—must be read in which is handled by the interface located at on the chip's top level. The data is there forwarded to the active cryptographic unit containing the input register. In order to avoid additional address lines the I/O registers are implemented as shift-registers. Therefore, the data must be strictly submitted in the following manner. First comes the master key with 16 bytes—highest byte must be applied first (big endian)—followed by 16 bytes of data. As the read data is smaller than the register size—only 32 of 64 bytes are allocated—additional cycles are required to shift the data to the start position. This can be done in 32 cycles as exactly half of the input storage is occupied. Afterwards, the data is shifted into the *Core* component which has been configured for either AES encryption or decryption. The result is afterwards stored in the first 16 bytes of the output register which must be shifted by 48 cycles before it can be read out.

The drawback of an AES decryption optimized for low area is the fact that the round keys must be computed on-the-fly. This implies that first computation time must be offered to compute the last round key out of the master key as the round keys are required in the inverse manner as compared to the encryption process. Through the integration of AES in the Grøstl datapath additional storage space is available which can be used to improve the decryption performance. Therefore, each AES encryption/decryption reuses the output register to store besides the result also the master key and the computed last round key. A special restructuring of the register into four parts all of size 16 bytes has therefore been applied and is shown in Equation 8.1.

$$output\_register = master\_key|last\_round\_key|...|result \tag{8.1}$$

An AES decryption proceeds in the following way. First a check is applied if the last round key corresponding to the applied master key is known. This is fulfilled by comparing the read master key with the first 16 bytes of the output register. If they match the 16 bytes following the master-key entry in the output register represent the corresponding last round key. The *Core* unit obtains therefore instead of the master key the already known last round key which implies that the cycles otherwise required for computing the last round key can be already used for the decryption process itself.

A Grøstl computation on the other hand is much more complex regarding logical

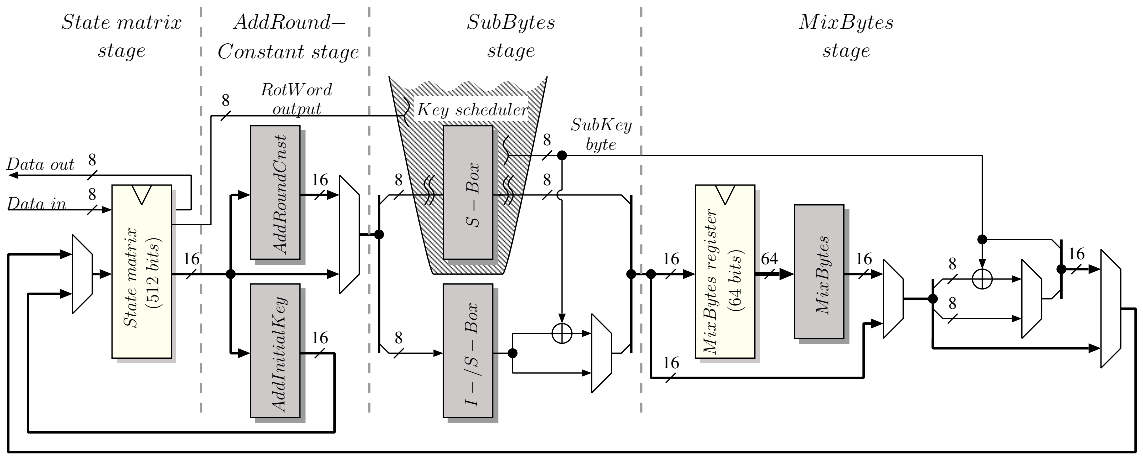


Figure 8.5: Core datapath w/o round-key generation and un-/loading of the state matrix.

connections on the top layer of GrÆStl and Grøstl, respectively. First of all the data to be hashed is again read in through the interface which saves the data in the cryptographic unit, in this case in the GrÆStl component. As the architecture was designed for a sequential computation of the $P/Q$-permutation first the original message contained in the input register is XORed with the output register containing the initial vector for Grøstl-224 before it gets applied to the *Core* unit which has been configured for $P$-permutation. Afterwards, the output register is updated through XORing its own content with the result from the *Core* computation. Next, the *Core* unit is reconfigured to $Q$-permutation and gets the original message as input. The result updates again the output register in the same way as before and represents now the new intermediate hash value. In case of a message fitting not into one block, the procedure starts from the beginning with the difference that the output register contains now instead of the initial vector the intermediate hash value. Otherwise the truncation step must be executed which is simply achieved through configuring the *Core* for $P$-permutation and applying the actual intermediate hash value to it. The result is afterwards again XORed with the output register. In order to finalize the truncation step a shift operation of the output register by 36 positions is applied. Afterwards, the result is ready to be read out.

**Note:** Grøstl-224 and Grøstl-256 can be easily exchanged through each other as their difference is simply another initial vector and the number of truncated bytes.

### 8.2.2 Common Datapath

Through the AT-plots for various AES and Grøstl architectures presented in Section 8.1, the decision was made to implement the common datapath based on 16/8 bits. The corresponding architecture is shown in Figure 8.5. It has been separated into four main components according to the round transformations of AES and Grøstl: a shared state implicitly performing the $ShiftBytes/ShiftRows$ operation, an $AddRoundConstant/AddInitialKey$ stage, a *SubBytes* stage and a *MixBytes* unit.

**Sharing the State.** One of the most obvious ways to share resources between AES and Grøstl is to share the memory resources for the state. The size of the state for AES

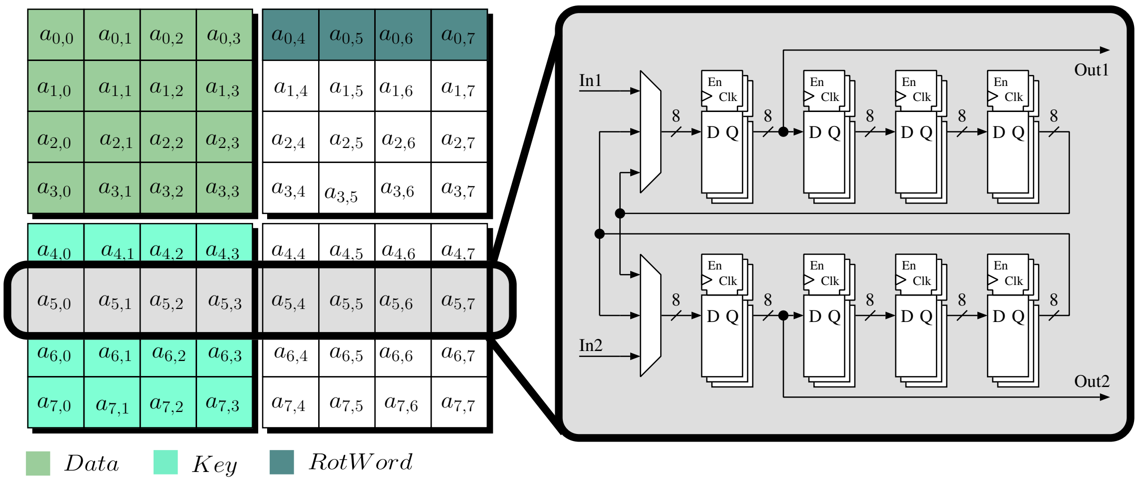

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{3,6}$ $a_{3,7}$
$a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{4,6}$ $a_{4,7}$
$a_{5,0}$ $a_{5,1}$ $a_{5,2}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$ $a_{5,6}$ $a_{5,7}$
$a_{6,0}$ $a_{6,1}$ $a_{6,2}$ $a_{6,3}$ $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ $a_{6,7}$
$a_{7,0}$ $a_{7,1}$ $a_{7,2}$ $a_{7,3}$ $a_{7,4}$ $a_{7,5}$ $a_{7,6}$ $a_{7,7}$
In1
In2
Out1
Out2
En
Clk
D Q
8
Data
Key
RotWord

Figure 8.6: Mapping the AES structure into the Grøstl structure and the construction of a single state-matrix row.

is 128 bits, i.e., a $4 \times 4$-byte matrix. Grøstl, in contrast, needs 512 bits (for variants returning a message digest of a size up to 256 bits), i.e., an $8 \times 8$-byte matrix. Thus, up to four AES States can fit into one Grøstl State which allows to integrate up to four AES encryption/decryption units in parallel to speed up the computation with minimal overhead. In order to keep the area requirements as low as possible parallel computations were strictly avoided. Thus, only one AES structure was mapped into the Grøstl state as illustrated in Figure 8.6, i.e., the data (requiring the upper left $4 \times 4$ byte) and the round key (requiring the lower left $4 \times 4$ byte). In addition to these memory locations, four bytes of the upper right $4 \times 4$ matrix were reused as temporary registers for the round-key generation, further on denoted as *RotWord* shift register. The round keys can therefore be computed on-the-fly without the need for a further memory bank.

The common state has been implemented using shift registers which has several advantages. First, they reduce the area requirements on common FPGA platforms (e.g., Xilinx FPGAs) since the Look-Up Table (LUT) in certain logic blocks can be configured as a shift register without using the flip flops available in each slice as also noticed by Chodowiec et al. [8]. Detailed informations on this mode called SRL-16 can be found in Section 7.2.3. Second, they are very flexible and can be used for both ASIC and FPGA designs as opposed to other memory architectures such as RAM macros or Block RAMs. Third, due to automatic shifts of intermediate values, additional address logic and multiplexer stages can be avoided. Thus, no *ShiftRows* or *ShiftBytes* units are needed because they are implicitly performed by the applied shift registers.

Each row in the State has been implemented as an 8-byte shift register that can be split into 4-byte shift registers with two independent inputs. Figure 8.6 shows one internal row composed of two 4-byte shift registers. When AES is performed, only 4-byte shift registers are used, 8-byte shift registers are used only during Grøstl computations. In order to reduce the power consumption during AES computation, an operand isolation technique was applied which switches off unused parts of the matrices, e.g., the lower right $4 \times 4$ byte of the state matrix. Furthermore, clock gating cells were applied to minimize toggling activity.

**Tweaked AddRoundConstant Stage.** For Grøstl, the state is modified through the *AddRoundConstant* function, which varies with the type of the required permutation. In case of the $P$ permutation, the first row gets modified through fixed constants whereas the rest stays untouched. This changes for the $Q$ permutation, where instead of the first row, the last row gets modified. Additionally, the rest of the state bits get flipped, cf. [15]. Note that in contrast to the Grøstl-0 specification (from the 31st of October 2008) where only a single byte gets modified (for both permutations), in the tweaked Grøstl version (from the 2nd of March 2011, version 2.0.1) multiple bytes get modified. Compared to existing work, which mostly presents solutions for Grøstl-0, e.g., in [62], this work presents an implementation that considers the tweaked variant including the modified round constants and initial vectors.

For AES, the state gets modified through the *AddRoundKey* function, which simply adds (XOR operation) the round key located in the lower left 4 x 4 bytes of the State matrix to the data in the upper left 4 x 4 bytes.

**Reusing SubBytes for AES Round-Key Generation.** The SubBytes transformation in Grøstl can be efficiently combined with the S-Box operation of AES, because both algorithms make use of the same S-box transformation. Minor effort has to be made in order to provide the inverse S-Box transformation required for AES decryption.

There exist several implementation optimizations for the AES S-box, e.g., given in [5], [49] or [64]. Most of the related work transformed the finite-field operations over $GF(2^8)$ into a composite of smaller fields, i.e., $GF((2^4)^2)$. This work uses the method proposed by Wolkerstorfer et al. [64], where a S-box is composed of two transformations, namely the calculation of a multiplicative inverse in the finite field $GF(2^8)$ and an affine transformation. An abstract view of the implementation is shown in Figure 8.7. For AES decryption, the affine transformation is exchanged with its counterpart and executed after computing the multiplicative inverse. As the Grøstl version is based on an 16-bit wide datapath, two S-boxes were implemented, one of them providing both transformation directions. AES is based on an 8-bit datapath, thus presence of an additional S-box was exploited in order to improve the performance of the AES round-key generation as described in the following.

Basically, there exist two possibilities to generate the round keys for AES encryption and decryption. First, the round keys are pre-computed and stored in non-volatile memory. Second, the round keys are computed on-the-fly. While the first option provides fast access to existing round keys, the second option is cheaper in terms of area requirements since no memory is needed to store the keys. Therefore, the second option was implemented.

While one S-box is used to perform the *SubBytes* operation of AES, the second S-box can be reused to calculate the round keys in parallel. For the round-key generation, one S-box, XOR operations and a small LUT that holds the round constants are required. Figure 8.8 and Figure 8.9 illustrate the general forward and backward round-key generation.

The forward round-key generation is done as follows. First, during the initialization of the common state, the last four bytes of the master key are loaded into the *RotWord* shift register (located in the upper right $4 \times 4$ matrix as shown in Figure 8.6). The output of the *RotWord* shift-register gets substituted by the shared S-box and modified with the round dependent constant *Rcon* before it gets added to the output of the first row of the key matrix (located in the lower left $4 \times 4$ matrix as shown in Figure 8.6). Afterwards, the result is loaded back into the *RotWord* shift-register and the first row of the key matrix before both get shifted. This is done for the first byte of each row of the key matrix in

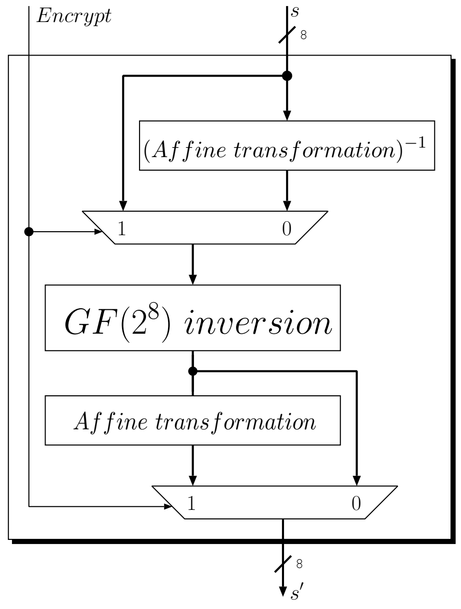
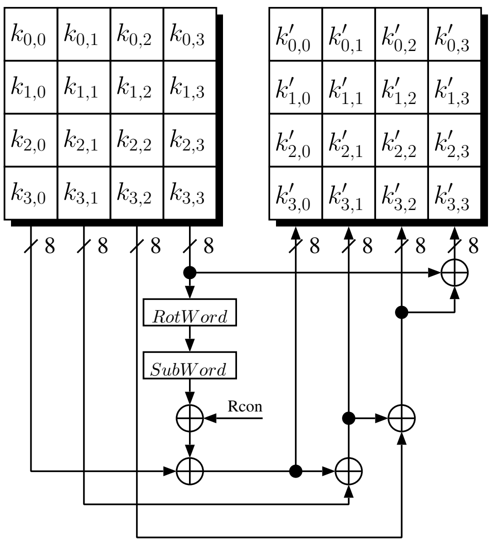


Figure 8.7: Architecture of the S-box supporting both transformation directions [64].



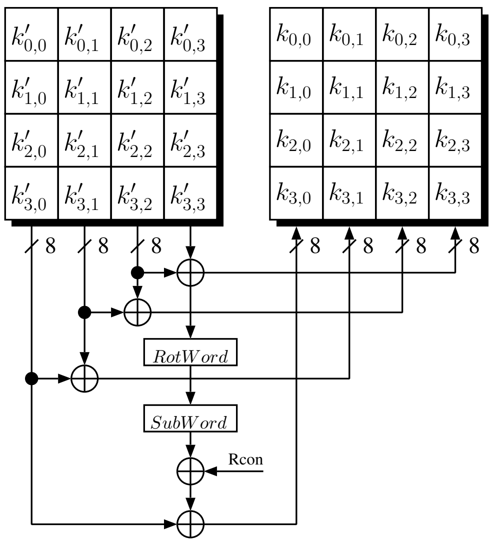


Figure 8.8: Scheme for the forward round-key generation.

Figure 8.9: Scheme for the backward round-key generation.

order to obtain the highest four bytes of the next round key. The following three columns of the next round key get calculated by applying the same procedure while bypassing the S-box and *Rcon* modifications. Due to the similarity of the forward and backward round-key generation, the latter will not be explained in detail.

**Combined MixColumns and MixBytes.** *MixColumns* and *MixBytes* have been combined to a common *MixBytes* function with 64 bits as input. As the datapath width was set to only 8 bits for AES and 16 bits for Grøstl, respectively an additional shift register with 16 bits as input and 64 bits as output was placed in front of the *MixBytes* function. This register is implemented in such a way as to shift not byte wise instead it shifts always two bytes per time. This special mode is required as during Grøstl operation *MixBytes* computes always an 16-bit output by internally shifting the data by one and the use of two separate 8-bit datapaths. Therefore, after one computation a shift by two bytes is required to avoid re-evaluating one value twice. Due to the datapath widths both algorithms require four clock cylces to load the *MixBytesReg* with the modified data from one state column. The *MixBytes* function stores the output into the state matrix needing four clock cylces in addition. Note that the *MixColumns* operation for AES encryption comes for free as it is implicitly computed during a Grøstl *MixBytes* operation. Only additional effort had to be made for AES decryption as there larger coefficients compared to the Grøstl version exist. Furthermore the power consumption of the *MixColumns* function can be lowered during AES operation due to the fact that only 32 bits of the 64 bits of the *MixBytesReg* are valid. Therefore, one of the two separate datapaths inside the *MixColumns* function can be switched off.

## 8.3 Results

Chameleon was implemented in VHDL and synthesized for both ASIC and FPGA platforms. In order to evaluate the efficiency of GrÆStl compared to separate implementations, results for stand-alone variants of AES and Grøstl are provided. However, it has to be noted that a fair comparison of the stand-alone variants with related work is largely infeasible since target was the combined version and therefore optimization techniques were integrated that do not affect the single-implementation variants. Nevertheless, still a statement regarding the efficiency of the combination can be given since all three components—the stand-alone variant for AES, the stand-alone variant for Grøstl as well as the combined version named GrÆStl—are contained on Chameleon. Results for the ASIC design are given in Subsection 8.3.1 whereas details on the FPGA version can be found in Subsection 8.3.2.

### 8.3.1 Application-Specific Integrated Circuit (ASIC)

Results for functional simulations (RTL and post-layout verification) and synthesis were achieved using *Mentor Graphics ModelSim* 6.5c and *Synopsys Design Compiler* 2010.03. The design was further mapped on a standard-cell library based on the $0.18\mu m$ CMOS process by UMC. One single Gate Equivalent (GE) corresponds to the area of a 2-input NAND gate, i.e., 9.3744 $\mu m^2$.

Table 8.3 lists the area occupation by AES, Grøstl and GrÆStl for a target frequency of 100 MHz. GrÆStl needs 16,550 GE of area not including an interface which would occupy another 200 GE. This is about 16% less than the sum of the two separate implementations. Furthermore, it shows that the overhead for AES is only 1.5 kGE which is in fact about the factor 2.3 smaller than the yet smallest known AES implementation reported by Feldhofer et al. [11] in 2005. Their design requires 3.4 kGE for both encryption and decryption. Comparing this work with the encryption-only designs presented by Hämäläinen et al. [19] in 2006 and the encryption-only design presented by Moradi et al. [45] in 2011 an area reduction by the factor 2 and 1.6 can be reported.

Figure 8.11 shows graphically the area distribution between the major components of the GrÆStl unit. As it was expected the biggest part is occupied by the I/O shift registers, both of size 512 bits and the *Core* containing the state matrix with another 512 bits of

Table 8.3: ASIC area results after synthesis.

| **Component** | **AES** [GE] | **Grøstl** [GE] | **GrÆStl** [GE] |
|---|---|---|---|
| Top Level Glue Logic | 50 | 460 | 800 |
| Input/Output Reg. | - | 8,250 | 8,250 |
| Core | 2,550 | 6,340 | 7,500 |
| State Matrix | 1,100 | 4,200 | 4,350 |
| AddRndCnst. | - | 100 | 100 |
| SubBytes | 330 | 540 | 600 |
| MixBytes | 490 | 900 | 1,100 |
| Core Glue Logic | 630 | 600 | 1,350 |
| Round-Key Gen. | 2,150 | - | - |
| Overall | 4,750 | 15,050 | 16,550 |

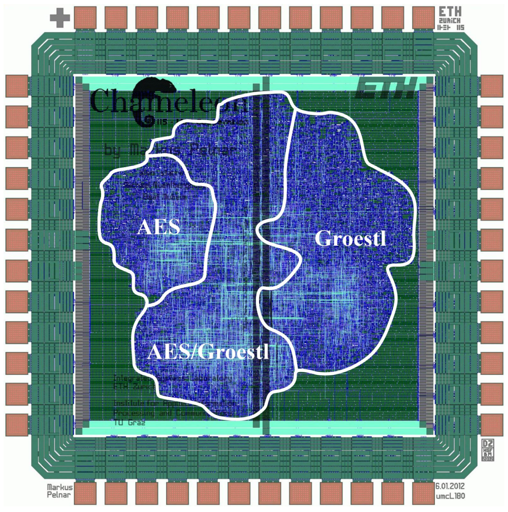


Figure 8.10: Chip layout of Chameleon.

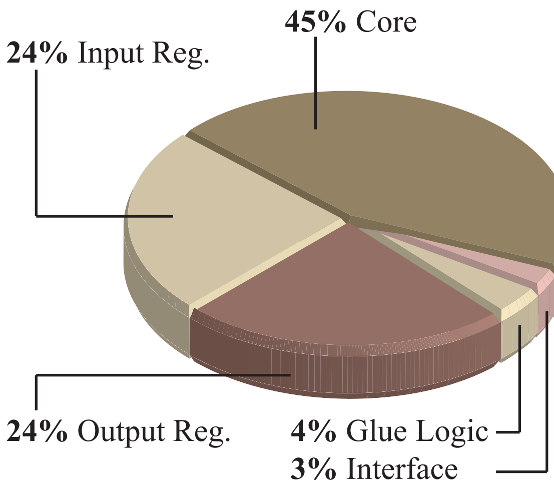


Figure 8.11: Synthesis results for the area consumption of the GrÆStl component.

data. In detail the complexity of the I/O shift registers is evaluated to 8,250 GE. The *Core* on the other hand occupies 6,340 GE whereas 4,200 GE are required for the state matrix. Therefore, the logic for the $P/Q$-permutation as well as for AES encryption/decryption (containing the on-the-fly computation of the round keys) are both realized with only 2,140 GE.

Execution times are given in Table 8.4, which presents only numbers for GrÆStl as their corresponding counterparts, the stand-alone versions for AES and Grøstl exhibit the same cycle counts. Therefore, a detailed description of the timings is only given for the combined hardware architecture.

AES encryption needs 940 clock cycles for processing one block of data. This includes an 8-bit four-phase handshaking that needs 288 clock cycles to load and unload the data and the key into the I/O registers of the GrÆStl component. As both of these registers are of size 64 bits and the data and the key for AES is only half of it, additional clock cycles are required in order to place the data in the shift registers on the right location. These cycles are already considered in the cycle count of the interface. Next, the data and the key must be loaded into the State matrix of the *Core* which takes another 32 time steps. Afterwards, the actual AES encryption starts with adding the initial key to the message. This is simply an XOR operation of the bottom left quarter of the state matrix with the top left quarter and takes another 33 clock cycles. Now the round computation starts by shifting the data area of the state matrix within 4 cycles according the AES Specifications, cf. [51]. As the datapath width is only 8-bit and due to the hardware architecture four sub rounds are required whereas each of these updates one column of the State. A sub round requires 13 clock cylces, whereas four clock cycles are required for updating the *MixBytes* register, 7 clock cycles are needed for updating the state and two further clock cycles for shifting the *MixBytes* register and the whole state by one. This results in a cycle count of 593 (encryption only not considering loading/unloading of the state, etc. ... .), but due to the fact that the last round of AES differs (*MixBytes* is skipped) it reduces to 588. Finally, 32 clock cycles are required to update the output register in an appropriate manner. 16 clock cycles for the result and 16 clock cycles for the last round key that is stored in order to increase the performance of a decryption operation.

Decryption on the other hand requires 1,558 clock cycles, whereas the behaviour of the interface stays the same. Due to two facts time consumption is increased. First, the decryption process requires (according to the AES specification, cf. [51]) the round keys in an inverse manner which implies that the last round key has to be first computed from the master key before the actual decryption can be started. Second, the on-the-fly computation of the round keys in an inverse manner is due to the given hardware resources slightly more time consuming. The procedure in detail is as follows. After loading the data from the input register into the state matrix of the *Core*, first the last round key is computed within 330 clock cycles followed by another 33 clock cycles to XOR the bottom left quarter of the state matrix with the top left quarter (initial *AddRoundKey* function). Afterwards, the round computation starts by shifting the state according to the AES specification, which takes 4 cycles. Again, due to the reduced datapath width and the chosen hardware architecture four sub rounds are required, whereas each of them updates one column of the state within 21 clock cycles. In detail, 13 clock cycles are needed for updating the *MixBytes* register, 7 clock cycles for updating the state and one clock cycle for shifting the *MixBytes* register. Furthermore, an additional cycle is required for each of the first nine sub rounds. This results in a cycle count of 1,164. The last round is again shorter and therefore only 57 cycles must be added, resulting in 1,221 cycles in total (decryption only not considering loading/unloading of the state, etc. ... .). Finally, the result has to be transferred to the output register which requires 16 clock cycles. An important fact not explained so far is that the last round key is written into the output register as soon as it gets available. Through this improvement the decryption process can be shortened if an encryption/decryption with the same master key has been executed

Table 8.4: Execution times for the different stages and modes of the GrÆStl unit.

| **Stage** | **AES Encryption** [Cycles] | **AES Decryption** [Cycles] | **Grøstl** [Cycles] |
|---|---|---|---|
| Interface | 288 | 288 | 404 |
| Load StateMatrix | 32 | 32 | $3 \times 64$ |
| Unload StateMatrix | 32 | 16[1] | $3 \times 64$ |
| Core | 588 | 907[2]/1221 | $3 \times 880$ |
| Overall | 940 | 1,243[2]/1,557 | 3,465[3] |

1) Last round key is immediately shifted out after it gets available. Different to encryption where it is shifted out at the end of the computation.
2) Last round key stored in output register. Therefore, forward round-key generation can be skipped.
3) Padded message fits one block. For each additional message block 2,277 clock cycles are necessary.

immediately before. In that case the 330 clock cycles required to compute the last round key can be avoided. Overall the cycle count reduces to 1,243 clock cycles only. For the stand-alone version this improvement is not an option as therefore additional storage space would be required (no I/O registers are available as the data is immediately transferred to the state matrix).

Hashing of a single block takes for both the stand-alone version of Grøstl as well as for GrÆStl 3,465 clock cycles. This cycle count results from the interface, additional I/O register shifts, loading/unloading of the state matrix and the *Core* itself. The four-phase handshaking occupies 404 clock cycles. As Grøstl-224 has been implemented the first 36 bytes of the output have to be truncated. Therefore, additionally 37 clock cycles are needed. The state matrix composed of shift registers to 8 rows has a size of 64 bytes and can be loaded and unloaded in 64 clock cycles. Due to the fact that first the $P$ permutation followed by the $Q$ permutation has to be computed before in the finalization step (since only one message block to hash) another $P$ permutation is required, the loading and unloading of the state is executed three times. Both $P$ and $Q$ permutation are based on an internal round function with a width of 10 rounds ($rnds = 10$) and require 880 clock cycles. At the beginning of each round, first the ShiftBytes function must be applied which takes 8 clock cycles ($shift_{bytes} = 8$). Afterwards, 8 sub rounds are applied whereas each is responsible for one column ($sub_{rnds} = 8$) . During the sub rounds for each row four clock cycles are required to load the 64-bit shift register in front of the *MixBytes* function ($reg_{update} = 4$). The next 4 clock cycles are responsible for updating the actual column of the state matrix with the result from the *MixBytes* function ($state_{update} = 4$) followed by another clock cycle for shifting the state matrix by one in order to gain the next column to be computed ($state_{shift} = 1$). Equation 8.2 presents the computation of the cycle count for the *Core* configured for either $P$ or $Q$ permutation. By replacing the variables with their assigned values the cycle count computes to 880 clock cycles.

$$Core_{P,Q} = ((tmp\_reg_{update} + state_{update}) * sub\_rnds + state_{shift}) + shift\_bytes) * rnds \quad (8.2)$$

Message padding is not supported at all as the assumption was taken that the padding is done externally. Therefore, all submitted data must already be padded.

Chameleon was taped-out successfully. For this, the target frequency was increased from 100 MHz to 125 MHz which represents the maximum possible frequency of the design (during synthesis 100 MHz were choosen for an easier comparability with results stated in literature). This in combination with clock gating and eight scan chains (not considered in the results stated before) required an additional amount of only 750 GE of chip area, ending up with around 17.1 kGE. Eight parallel scan chains were chosen as exactly the same amount of I/O pins is available. Reusing them for the scan paths itself has two advantages. First, another 16 pins on the chip can be avoided. Second, the test complexity can be reduced by orders of magnitude. More details on the test I/O pins can be found in Appendix B.3.

**Information:** Scan chains are a conecpt used in design for testability (DFT) to enable verification of a produced chip. The principle is that sequential storage elements and combinatorial logic are separated in order to ease the finding of appropriate testvectors (testvector = stimulus vector + expected response vector) Evaluating sequential circuits—containing storage elements and combinatorial logic—is much more complex as evaluating combinatorial circuits only!
In order to obtain this feature all the storage elements (full scan path) are replaced by special scan chain storage elements during the backend design. These feature two data inputs. The selection of which to use is done over the testmode signal. In order to create a scanpath—all the storage elements are connected in a row—the output of each storage element is additionally connected to the test input of the next storage element in the line. Verification of the system functionality can be reached by activating the testmode and loading the stimulus vector into the system. Afterwards, one cycle of normal operation (test mode disabled) is applied. The result is unloaded (shifted out) and compared with the expected response vector. If they match no failure has been detected. In order to reduce the complexity further multiple parallel scan paths can be used. Therefore, the test effort can be reduced by orders of magnitude. A good approach is to multiplex the data I/O pins and use them during test mode as I/O pins for the various scan paths.

The remarkable small difference between synthesis results and back-end results (in fact it is even negligible when comparing synthesis results and back-end results both targeting the same frequency) between the synthesis results and the back-end results are due to the design which is largely build upon shift registers. The layout of the chip, highlighting the floorplan of the three different modules, is illustrated in Figure 8.10. Regarding power consumption, GrÆStl requires for an AES encryption/decryption 130 µW/MHz and for a Grøstl computation 200 µW/MHz. These values were achieved through a power analysis with a value change dump (VCD) file containing the switching profile of a previously executed gate-level simulation for the various modes of operation. The produced chip on the other hand exhibits a maximum frequency of around 133 MHz and delivered for an AES encryption an average power consumption of 63 µW/MHz. To the best of our knowledge, there is no ASIC design for an AES and Grøstl combination available so far, which targets a low-resource implementation and has finally been taped-out.

### 8.3.2 Field Programmable Gate Array (FPGA)

All stated FPGA synthesis results have been achieved with *Xilinx ISE Design Suite* 12.1. In order to get comparable numbers the parameter set "Area Reduction with Physical Synthesis" was used. Results for the architectures with and without an interface included were acquired for Xilinx Spartan-3, Spartan-6, Virtex-5 and Virtex-6 FPGAs whereas

Table 8.5: Post place-and-route results for various Xilinx FPGAs not considering the interface (original design without adaption for SRL-16 mode).

| | Spartan-3 | | | Virtex-6 | | |
|---|---|---|---|---|---|---|
| | **AES** | **Grøstl** | **GrÆStl** | **AES** | **Grøstl** | **GrÆStl** |
| Number of slice LUTs | 806 | 976 | 1,798 | 505 | 650 | 1,247 |
| Number used as logic | 796 | 823 | 1,676 | 424 | 490 | 1,024 |
| Number used as shift reg. | 8 | 128 | 64 | 5 | 76 | 64 |
| Number used as a route-thru | 2 | 25 | 58 | 76 | 84 | 159 |
| Number of slice registers | 277 | 543 | 841 | 272 | 541 | 722 |
| Frequency (MHz) | 35 | 40 | 30 | 110 | 115 | 80 |
| Number of occupied slices | 493 | 643 | 1,166 | 169 | 228 | 371 |

only the results for the Spartan-3 as well as for the Virtex-6 are described in detail. Numbers for the others can be found in Appendix C.1 and Appendix C.2. As the overall architecture was designed in a way to simply add and remove the support for each of the cryptographic modules, the area numbers stated were always achieved with only one component instantiated.

Table 8.5 shows the place-and-route results of the original design (interface not included and no modification due to the SRL-16 mode) on a Spartan-3 (low-cost device) and a Virtex-6 (high-end device). GrÆStl occupies here on a Spartan-3 1,166 slices running with a maximum frequency of 35 MHz. As easily noticeable through the occupied slice registers not even a single 512-bit register is available (e.g., to store the Grøstl state). Hence, the information must be stored in a different way which is called SRL-16 (by default enabled). There, besides slice registers also shift registers are used to store information. The amount of used 16-bit shift registers can be found in Table 8.5. During synthesis of the design the Xilinx tools map shift registers onto reconfigured LUTs of slices which are not reserved for logic only. Such a reconfigured LUT can be used as an 16-bit-wide shift register (the width is variable). One CLB of a Spartan-3 contains for example four slices whereas two of them can be used for distributed RAM, SRL-16 or the like. A slice contains two LUTs. Hence, a shift register with a width up to 64 bits can be provided. Through a connection the LUTs of more than one CLB even wider shift registers can be provided. As the LUTs inside a slice do not support a reset functionality an improved automatic mapping can be achieved by removing the reset from shift registers where it is not absolutely required. In that way (normally more) shift registers get automatically recognized. Table 8.6 presents the results for the modified design where the reset was removed from the I/O registers and the state matrix. For the GrÆStl design so nearly 200 slices on a Spartan-3 and around 70 slices on a Virtex-6 can be saved.

On the first sight it looks like that the Spartan-3 is more efficient regarding this mode of operation. In fact it is not as the complexity of one CLB can greatly differ between various FPGA families (e.g., number and size of the available LUTs in a slice).

Another interesting point to mention is the fact that the stand-alone Grøstl version is much smaller as GrÆStl which differs greatly to the ASIC design where only about 16 percent overhead was required to embed AES into Grøstl. Also the relation between AES and Grøstl on ASIC and FPGA platforms is completely different. For example, AES requires only a third of the Grøstl resources if targeting an ASIC design flow whereas it requires about five sixths if targeting the Xilinx Spartan-3 FPGA. The reason for that is that a major difference exists between ASIC and FPGA designs. On FPGA platforms

Table 8.6: Post place-and-route results for various Xilinx FPGAs not considering the interface (modified design due to SRL-16 mode).

| | Spartan-3 | | | Virtex-6 | | |
|---|---|---|---|---|---|---|
| | **AES** | **Grøstl** | **GrÆStl** | **AES** | **Grøstl** | **GrÆStl** |
| Number of slice LUTs | 838 | 896 | 1,805 | 455 | 531 | 1,046 |
| Number used as logic | 788 | 743 | 1,554 | 419 | 443 | 927 |
| Number used as shift reg. | 48 | 128 | 192 | 24 | 64 | 96 |
| Number used as a route-thru | 2 | 25 | 59 | 12 | 24 | 23 |
| Number of slice registers | 48 | 293 | 407 | 169 | 279 | 360 |
| Frequency (MHz) | 35 | 40 | 30 | 110 | 115 | 80 |
| Number of occupied slices | 442 | 488 | 956 | 142 | 202 | 302 |

storage space comes nearly for free. For ASICs, storage space is always a limiting factor as it concurrently implies an increase in the overall area. Referring to the pie-chart in Figure 8.11 it is easily recognizable that GrÆStl is dominated by storage space and not by logic. Furthermore, if summing up the slices of the stand-alone versions of AES and Grøstl it shows that they are together smaller as GrÆStl. This can be explained through the fact that the state matrix of GrÆStl is split up in the half. Therefore, twice as much resources are required as it is not possible to reuse the unused space of a LUT reconfigured for a shift register. For example, if such a LUT is used as 4-bit shift register it is not possible to use the rest as a 10-bit shift register.

## 8.4 Comparison with Related Work

In this section the achieved results are set in contrast with publications targeting the same goal, namely low-area implementations of AES and Grøstl. Therefore no results for high-throughput variants or the like are included. Furthermore, no results are listed which are based on platform or technology-dependent components as this would be not a fair comparison to the design developed in this work (targeting low-area and high flexibility). A comparison with ASIC publications targeting low-area implementations of AES and Grøstl is given in Subsection 8.4.1. Last but not least, Subsection 8.4.2 lists implementations based on Xilinx FPGAs and low-area variants for AES and Grøstl and puts them in contrast with the designs developed in this work.

### 8.4.1 Comparison of ASIC Results

Table 8.7 gives a comparison of the ASIC results with related work. While the stand-alone implementation of AES is about 1 kGE larger than existing work [11, 19], only a small overhead is required for Grøstl to support also AES, i.e., 10 %. Moreover, the GrÆStl implementation is only slightly larger than the work of Tillich et al. [62] but supports AES and implements the tweaked version of Grøstl instead of Grøstl-0. The work from Kavun et al. [32] on the other hand outperforms both, the stand-alone version of Grøstl and also GrÆStl. As their design is very similar to this one and they do not explicitly state the number of used 512-bit registers it is assumed the results were achieved by applying constantly the message on the input as therefore a reduction from three 512-bit registers to only two is possible (in case of computing the $P/Q$-permutation sequentially). Regarding power consumption, it shows that the design meets most requirements even

Table 8.7: ASIC comparison of AES-128 (incl. decryption) and Grøstl-224.

| Source | Width | Techn. | $f_{\mathrm{max}}$ | Enc. | Dec. | Hash | Power | Area | |
|---|---|---|---|---|---|---|---|---|---|
| | [bits] | [$\mu m$] | [MHz] | [cycles] | [cycles] | [cycles] | [µW / MHz] | [kGE] | |
| Hämäläinen et al. [19] | 8 | 0.13 | 153 | 160 | n/a | - | 37 @ 1.2 V | 3.9[1] | |
| Feldhofer et al. [11] | 8 | 0.35 | 80 | 1,032 | 1,165 | - | 45 @ 1.5 V | 3.4 | AES |
| **This work - AES** | **8** | **0.18** | **100** | **742** | **1,025** | **-** | **130 @ 1.8 V** | **4.75** | |
| Tillich et al. [62] | 64 | 0.35 | 56 | - | - | 196 | 2,210 @ 3.3 V | 14.6 | |
| Kavun et al. [32] | 8 | 0.09 | n/a | - | - | 1,280 | n/a | 9.2 | Grøstl |
| **This work - Grøstl** | **16** | **0.18** | **100** | **-** | **-** | **3,061** | **200 @ 1.8 V** | **15.05** | |
| **This work - GrÆStl** | **8/16** | **0.18** | **100** | **742** | **1,025** | **3,061** | **200 @ 1.8 V** | **16.55** | GrÆStl |

[1] The area for the design, including the decryption has been estimated with additional 25 % of the original (encryption only) AES design.

for a contactless operation, e.g., for contactless smart cards. However, due to different fabrication technologies and supply voltages, a fair comparison to the given numbers is not possible.

### 8.4.2 Comparison of FPGA Results

Table 8.8 provides a comparison of the FPGA results with related work. It shows that the stand-alone AES implementation is the smallest on the Spartan-3 occupying only 442 slices. For the Virtex-6, the design needs only 142 slices. The Grøstl implementation needs 488 slices on the Spartan-3 and is therefore about 2 times smaller than the work of Jungk et al. [27] with 967 slices. The smallest state-of-the-art implementation so far was presented by Kaps et. al [30] at the final SHA-3 conference. Their implementation requires only 766 slices which is about 300 slices larger than the implementation presented in this work. Compared to the work of Kerckhof et al. [33], we only need 202 slices on the Virtex-6 instead of 260, i.e., a factor of about 1.3 smaller. The implementation of Kaps et al. [30] on a Virtex-6 is in the same region and therefore also considerable larger. Considering the area occupation only, this work presents the so far smallest stand-alone versions for AES and Grøstl without using Block RAMs. For a fair comparison it must be stated that the computation times of the implementations are much larger due to the reduced datapath width.

Table 8.8: FPGA comparison of AES-128 (incl. decryption) and Grøstl-224.

| **Source** | **Width** [bits] | **Digest** [bits] | **Device** [type] | $\mathbf{f_{max}}$ [MHz] | **Enc.** [cycles] | **Dec.** [cycles] | **Hash** [cycles] | **Area** [slices] | |
|---|---|---|---|---|---|---|---|---|---|
| Bulens et al. [3] | 128 | 128 | Spartan-3 | 150 | 12 | 12 | - | 2,150 | |
| **This work - AES** | **8** | **128** | **Spartan-3** | **35** | **742** | **1,025**[1] | **-** | **442** | AES |
| Bulens et al. [3] | 128 | 128 | Virtex-5 | 350 | 11 | 11 | - | 550 | |
| **This work - AES** | **8** | **128** | **Virtex-6** | **110** | **742** | **1,025**[1] | **-** | **142** | |
| Jungk et al. [27] | 64 | 224/256 | Spartan-3 | 182 | - | - | 160 | 967 | |
| Kaps et al. [30] | 32 | 256 | Spartan-3 | 150 | - | - | 357 | 766 | |
| **This work - Grøstl** | **8/16** | **224** | **Spartan-3** | **40** | **-** | **-** | **3,061** | **488** | |
| Kerck. et al. [33] | 64 | 256 | Virtex-6 | 280 | - | - | 450 | 260 | Grøstl |
| Kaps et al. [30] | 32 | 256 | Virtex-6 | 360 | - | - | 357 | 263 | |
| **This work - Grøstl** | **8/16** | **224** | **Virtex-6** | **115** | **-** | **-** | **3,061** | **202** | |
| **This work - GrÆStl** | **8/16** | **224** | **Spartan-3** | **30** | **742** | **1,025**[1] | **3,061** | **956** | GrÆStl |
| **This work - GrÆStl** | **8/16** | **224** | **Virtex-6** | **80** | **742** | **1,025**[1] | **3,061** | **302** | |

[1] Decryption with stored last round key.

# Chapter 9

# Building an FPGA System for Hardware/Software Evaluation

This chapter is concerned with a hardware/software comparison of the cryptographic primitives AES, Grøstl and GrÆStl. The hardware components already described in the previous chapter must be incorporated in the FPGA system. With the general goal of a low-area design combined with highly restricted environments a low-budget FPGA was chosen as target platform. Therefore, the decision was to implement the system on a Xilinx Spartan-3 XC3S400-4FTG256C.

The rest of this chapter is structured in the following way. An overview over the general system requirements for establishing a fair hardware/software comparison is given in Section 9.1 followed by a general description of the developed system in Section 9.2. Details for the target FPGA are given subsequently in Subsection 9.2.1. Subsection 9.2.2 presents details about the used microcontroller, the supported toolchain and the developed system toolchain. The used cryptographic modules and their interface to the microcontroller is described in Subsection 9.2.3. Furthermore, it contains details about the verification strategy, for both the simulation and the real device. An insight to the developed cryptographic library is given in Subsection 9.2.3. Last but not least the results in Section 9.3 are split up into two parts. The first part presents in Subsection 9.3.1 the influence of external data transfer on the overall timing. The second part concerns the hardware/software comparison and follows in Subsection 9.3.2

## 9.1 General System Requirements

In order to obtain a FPGA system which can be used for a hardware/software comparison a few system requirements are necessary. These are as follows:

- Synthesizable microcontroller
- Toolchain supporting chosen microcontroller
- Cryptographic modules for AES, Grøstl and GrÆStl (hardware and software)
  - Hardware components running with either 5/20 MHz switchable over software
- Verification of cryptographic modules
- Cryptographic library for accessing the three units in all modes of operation

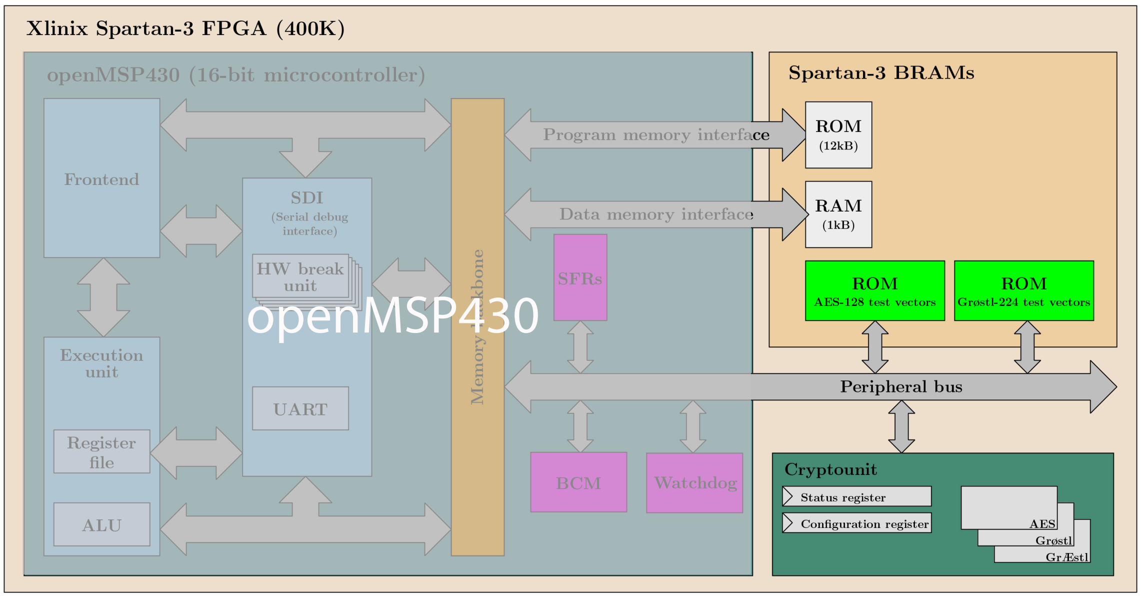


Figure 9.1: FPGA system for hardware/software comparison.

## 9.2 System Architecture

The developed system architecture is presented in Figure 9.1. As basis the Xilinx Spartan-3 XC3S400-4FTG256C FPGA is used. On top of it an 16-bit syntesizeable microcontroller taken from OpenCores, namely the openMSP430 [16] is working, which is compatible to the MSP430 family from Texas Instruments (TI) [61]. Both variants are supported from the toolchain “GCC 4.x toolchain for Texas Instruments MSP430 MCU”, or in short MSPGCC4 [46]. The newer version is named MSPGCC. In this work still the old toolchain, the MSPGCC4 was used as initial linker problems with the newer version occured. It turned out that the problems resulted from incorrect configurations which were solved at the end. Therefore, both versions can now be used for generating the ROM content (stores the program code). Setting up on these toolchains, a script was developed which builds the ROM background for the simulation and the real device as described in detail in Subsection 9.2.2. ROM and RAM for the synthesizeable microcontroller are located externally and can be accessed over two busses named *Program Memory Interface* and *Data Memory Interface* both of 16-bit size. The ROM further features 12 kB of memory and the RAM 1 kB. Therefore, enough space is available for implementing the software routines for AES and Grøstl. Both memories are outsourced into the BRAMs of the FPGA to reduce the utilized number of slices. The memory mapping for accessing the ROM/RAM and the peripherals is described in Subsection 9.2.2. Via this mapping, an additional hardware interface and the peripheral bus access to the cryptographic modules AES, Grøstl and GrÆStl was established. Same applies for the two ROMs containing information (test vectors) used for verifying the functionality of the system.

### 9.2.1 Xilinx Spartan 3

Detailed informations and explanations can be found in Chapter 7 and the Spartan-3 Generation FPGA User Guide [68].

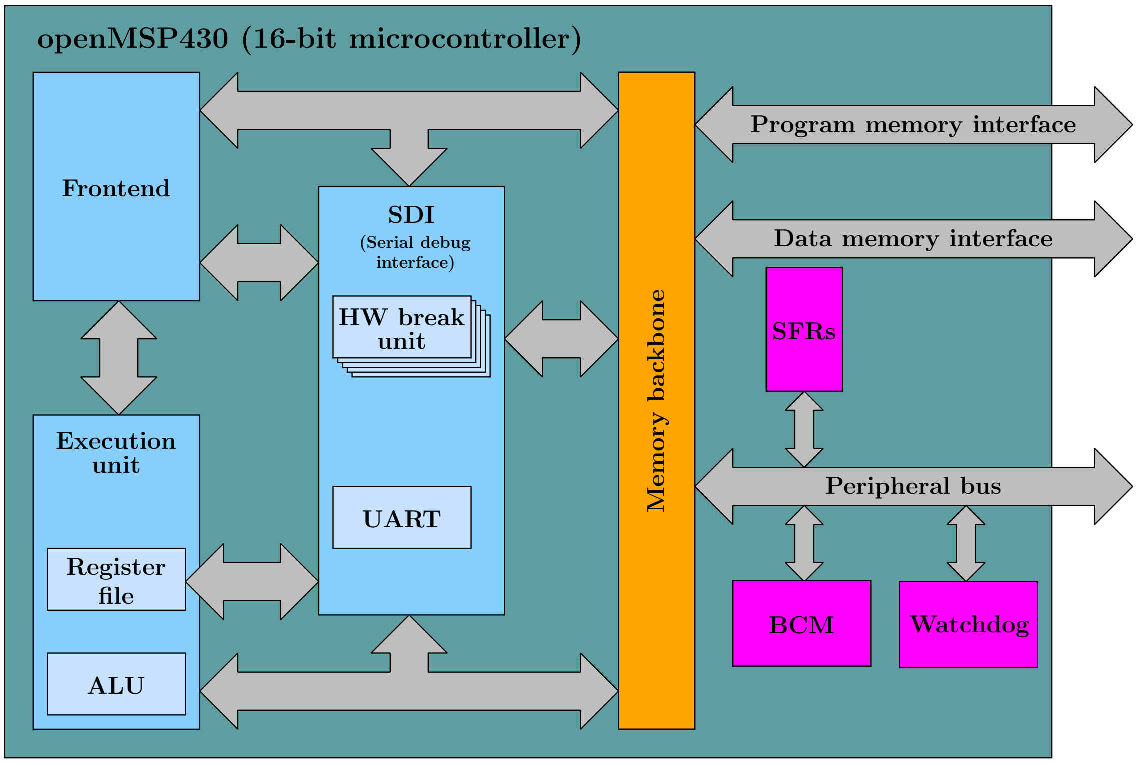


Figure 9.2: Structure of the openMSP430 from OpenCores [16].

### 9.2.2 openMSP430

Figure 9.2 shows the structure of the openMSP430 which is compatible to the MSP430 family of Texas Instruments [61]. It features a full instruction set which is supported by the old MSPGCC4 and the newer MSPGCC toolchain. Furthermore, interrupts that were originally available were removed in order to reduce the overall size of the microcontroller to fit the complete system within the target FPGA. Power-saving modes can be additionally addressed over software to save energy, for example, during times of low workload. As the memory size for both program and data is easily configurable it can be fitted to each application requirement. The microcontroller was designed in a modular way where peripherals like the "16x16 hardware multiplier" can be accessed over the peripheral bus. Same applies for the basic clock module, watchdog, timer A and GPIO (port 1 to 6). As many 8 and 16-bit peripherals can be connected as these fit into the peripheral address space. As the target FPGA is quite limited regarding the available resources, watchdog and timer A were removed. Also the GPIO ports were limited to only two. On the other hand the two test roms containing test vectors for AES and Grøstl as also the cryptographic modules were made accessible in this way.

The basic memory mapping is shown in Figure 9.3. It starts at the bottom with the peripheral registers followed by the data memory with a size of 1 kB. At the top a space is reserved for the interrupt vector table which is not occupied as interrupts are not enabled. Below, the program memory with a size of 12 kB follows. The stack of the system starts at the end of the RAM and grows in the direction of its beginning. Therefore, a careful programming style must be used in order to avoid an overwriting of return addresses through allocated memory or conversely an overwriting of data through the stack.

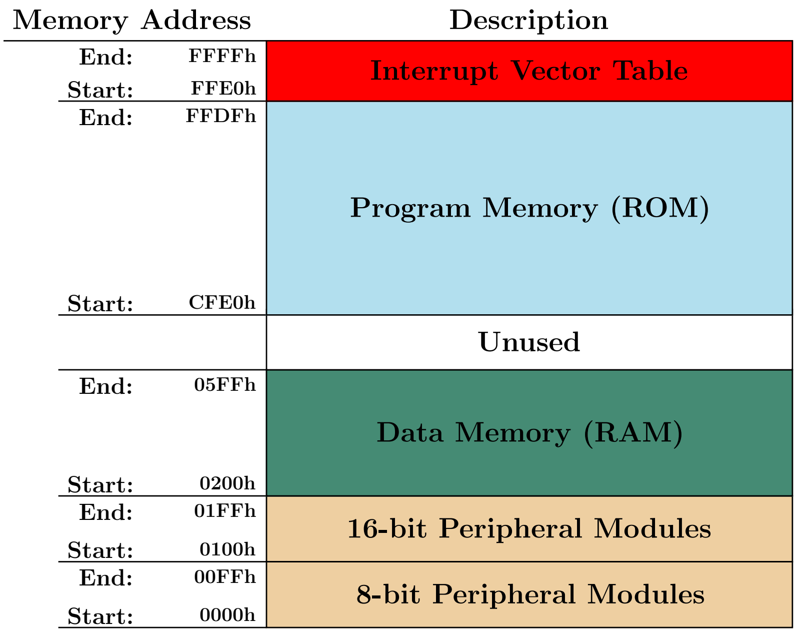


Figure 9.3: Memory mapping of openMSP430 with 12 kB of ROM and 1 kB of RAM.

The test roms and the cryptographic modules are both accessible over the 8-bit peripheral bus. The base address for the test roms and the cryptographic modules is set to 0080h and 0090h, respectively. Detailed information can be found in Subsection 9.2.3.

### MSPGCC Toolchain

The MSPGCC4 toolchain is the porting of the GCC toolchain for the Texas Instruments MSP430 family and is no longer supported as the contributions have been incorporated into the newer version named MSPGCC. During this work the old version, which is the MSPGCC4-20110312 was used. The package can be downloaded from Sourceforge.net [46] and includes an installation script named *buildgcc.pl*. The newer version is slightly more difficult to install as no explicit install script is delivered. Therefore, such a script was developed which is presented in Appendix D.2. Both scripts can be executed by a user as long as the access to the installation path is not restricted. The following libraries and files are available after installation:

- GNU Mulitple Precision Arithmetic Library (GMP)
- GNU MPFR Library (MPFR)
- Multiprecision Library (MPC)
- PPL
- CLoog
- GCC Toolchain for MSP430
  - BINUTILS
  - GCC Library

- GDB Library
- MSP430MCU Files
- MSP430-LIBC Files

**System Toolchain**

The system toolchain shown in Figure 9.4 and represented through the perl script "build_system.pl" generates the content for the ROM located in the BRAMs of the target FPGA. As both the simulation and the real device must be supported, two different file types are required namely *.mif* and *.coe*. The former represents the ROM initialization data for simulation and the latter for the FPGA itself. The procedure of the script is as follows. First, it obtains as input the desired program and data memory size, a linker file for the open-MSP430 as presented in Appendix D.1—contained in the package from openCores—and the sources themselves. With the input information the script adjusts three lines of the linker script: text area, data area and the stack offset.

- `text   (rx)  : ORIGIN = 0xF800,  LENGTH = 0x800`
- `data   (rwx) : ORIGIN = 0x0200,  LENGTH = 0x080`
- `PROVIDE (__stack = 0x280) ;`

The adjusted linker file and the sources themselves form the input for the msp430-gcc compiler. With the linker option of the compiler enabled both steps the compilation and the linking are performed at once resulting in the executable *sandbox.elf*, which gets subsequently transformed into a *.ihex* format. As the *create_msp430_rom.pl* script requires a *.mem* format an additional transformation step is necessary which is done through the *ihex2mem* tool. The output of the *create_msp430_rom.pl* represents afterwards the initialization data for the simulation (*.mif*) and also for the FPGA itself (*.coe*) and is automatically named after the size of the generated block-ram initialization files. For a 12 kB BRAM two output files with the names *rom_8x6kB_hi.mif* and *rom_8x6kB_lo.mif* are generated as this 16-bit microcontroller supports single-byte access. Therefore, both files together form the whole memory space where, for example, the first byte of each file are tied together and so on.

These files are only helpful in combination with the fitting framework for the BRAMs. In order to get these the Xilinx LogicCore IP Block Memory Generator v6.3 was used. The wizard requires only basic data like the size of the block memory or if an enable signal is desired. Additionally, it offers the possibility of initializing the BRAM with files of type *.coe* which in fact were generated from the *build_system.pl* script before. The output of the wizard are the files shown in Figure 9.4. They can be used for executing a simulation or for the back-end design to generate the bitstream.

For the FPGA target it is necessary to re-build the framework for the ROM each time another background should be used. For the simulation instead a trick was used to avoid these rebuild process. Therefore, the framework for the ROM was build once with the *.coe* files used as initialization data. The output files of the Core Generator can be seen in Figure 9.4. The *.mif* files must be located in the base Modelsim directory as they are loaded from there each time the simulation is executed. As no difference exists between the *.mif* files generated from the *build_system.pl* script and the Core Generator with the *.coe* files as input a symbolic link can be used to reference each time to the *.mif* files

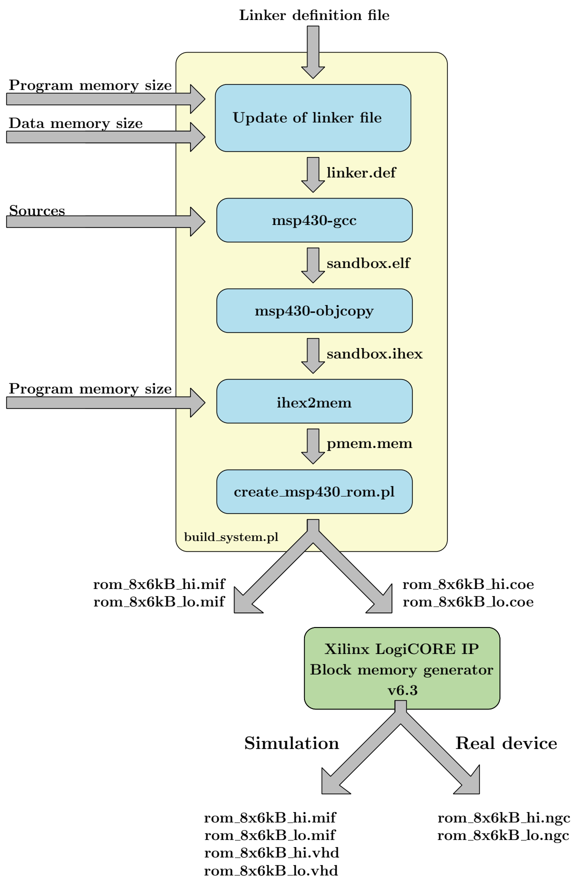


Figure 9.4: Creating the ROM files for the simulation and the real device.

generated by the *build_system.pl* script. In that way it is possible to avoid the rebuild process through the Core Generator.

### 9.2.3 Cryptographic Modules

The cryptographic modules have been taken over from Chapter 8 of this thesis. The work which had to be done was to develop an interface for establishing a communication between the openMSP430 and the cryptographic modules as well as the ROMs, containing test vectors for the AES/Grøstl verification.

#### Linkage to openMSP430

The connection between the openMSP430 and the cryptographic modules was established over a shared interface as presented in Figure 9.5. For that purpose four register banks, all with a size of 8 bits, are introduced which can be accessed on the one hand by the openMSP430 over the peripheral bus and on the other hand over general connections (wires) by the cryptounit itself. The memory mapping for the registers is presented in Figure 9.6. Through content modifications of memory location 0090h both the status of the cryptounit as well as the I/O handshaking can be managed. 0091h controls the behaviour of the cryptounit in terms of selecting the desired unit and mode of operation as well as the frequency at which the cryptounit should operate. Each I/O byte is further

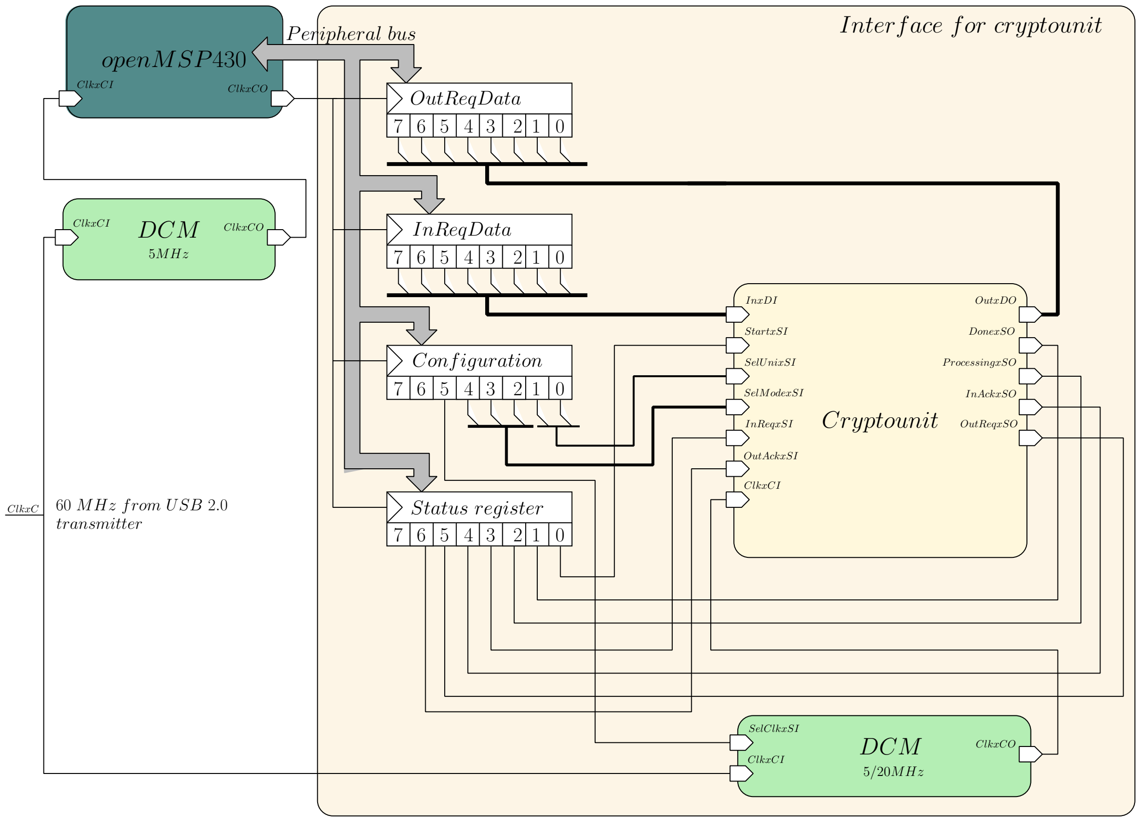


Figure 9.5: Peripheral bus used for connecting the cryptounit to the openMSP430 core.

Table 9.1: Description of the status register.

| **Bit** | **Name** | **Description** | **Access** |
|---|---|---|---|
| 7 | RFU | - | - |
| 6 | OutAck | Acknowledge signal for unloading of data | Software |
| 5 | OutReq | Request signal for unloading of data | Peripheral |
| 4 | InAck | Acknowledge signal for loading of data | Peripheral |
| 3 | InReq | Request signal for loading of data | Software |
| 2 | Processing | High if processing is going on | Peripheral |
| 1 | Done | High for duration of output handshaking | Peripheral |
| 0 | Start | Must be high for duration of input handshaking | Software |

buffered at memory location 0093h/0094h to establish a secure handshaking as the clock domains for the microcontroller and the cryptounit differ. The registers of the shared interface always operate with the same clock frequency as the microcontroller.

An arbitrary—AES/Grøstl/GrÆStl—computation can be started through software in the following way. First the content of the configuration register is written according Table 9.2. As an example, for an AES encryption with 20 MHz the content must be modified to 0024h. For the rest of the computation this value must not be changed.

Afterwards the *Status* register and the *InReqData* register control the input handshaking. Table 9.1 explains the meaning of the individual bits in the status register in detail. First, the *Start* bit of the *Status* register must be set to high to signal the be-

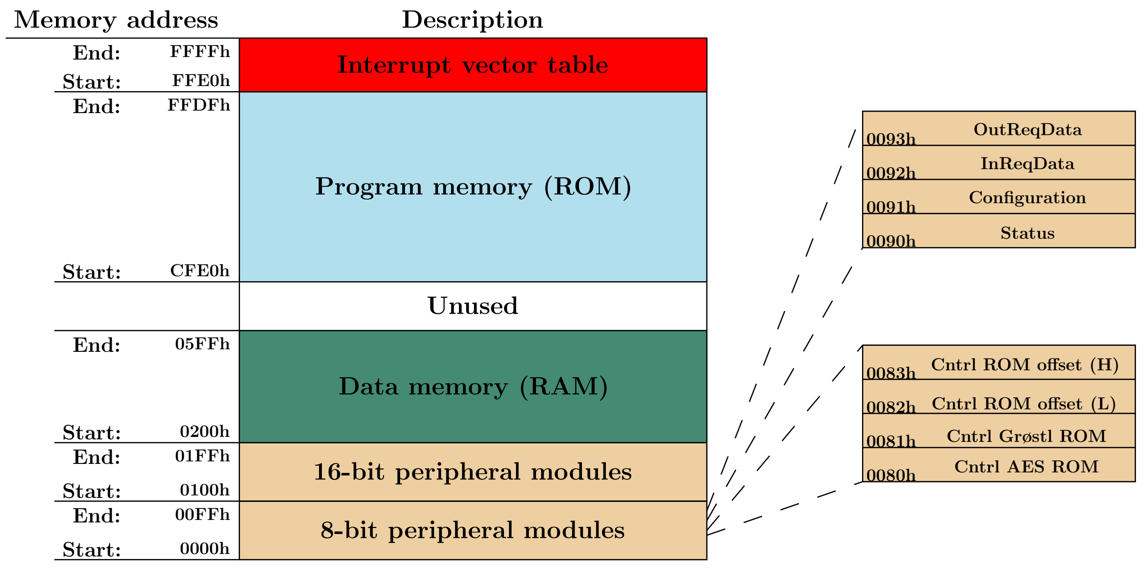


Figure 9.6: Memory mapping of the interface registers.

Table 9.2: Description of the configuration register.

| Bit(s) | Name | Description | Access |
|---|---|---|---|
| 7:6 | RFU | - | - |
| 5 | SelClock | Select Clock Frequency<br>1...20 MHz<br>0... 5 MHz | Software |
| 4:2 | SelUnit | Select cryptounit<br>000...Disable all modules<br>001...AES<br>010...Grøstl<br>011...GrÆStl in AES mode of operation<br>100...GrÆStl in Grøstl mode of operation | Software |
| 1:0 | SelMode | Select AES mode of operation<br>0X...Encryption<br>1X...Decryption<br>Select Grøstl mode of operation<br>00...Intermediate block<br>01...First block<br>10...Last block<br>11...First and last block | Software |

ginning of the input handshaking. Afterwards, this bit stays high until the end of the input handshaking. Next the *InReqData* register is written with the input byte for the computation before the *InReq* bit of the *Status* register is set in addition to the *Start* bit signalling the core that valid input data is available. Until this step all bits were modified through software which changes next as the core sets the *InAck* bit of the *Status* register to high after it has recognized that a request is pending and signals therefore that it has taken over the valid input data. This information is written to the *Status* register always

Table 9.3: Status of the FPGA on-board LED during the test phase.

| **Status of LED** | **Description** |
|---|---|
| Enabled | After a reset the LED is enabled for a few milliseconds before it gets disabled. At that moment of time the test procedure starts. Only two subsequent LED states are possible. Either the LED is again enabled (test procedure finished successful) or it starts blinking (failure occured). |
| Disabled | Test procedure active. |
| Blinking | Failure occured and test program aborted. |

at the positive clock edge of the microcontroller and not at the clock edge of the cryptounit. Additionally, the core sets the *Processing* bit of the *Status* register to signal that a computation is actually ongoing. As a four-phase handshaking is used the signals on the request and acknowledge line must be brought back into the original state. Therefore, the software checks periodically if the hardware has set the *InAck* bit to high. If so it resets the *InReq* bit. Same applies for the cryptounit which resets the *InAck* bit after recognizing the original state of the *InReq* bit. Afterwards, the *InReqData* is updated through software with the next input byte. This procedure takes place until all input bytes—either 32 for AES or 64 for Grøstl—have been taken over by the cryptounit. The lowest bit of the *Status* register is additionally reset to zero signalling the end of the input handshaking. Afterwards, it takes some time until the computation has finished which is signalled through the cryptounit by setting the *Done* bit to high. This bit stays set until the output handshaking has finished. Furthermore, the *OutReq* bit of the *Status* register is set to high by the cryptounit which represents the request for reading out data. The software polls in the meanwhile for this value. After recognizing this state the software saves the output byte and sets the *OutAck* bit to high to acknowledge the request of the cryptounit. The cryptounit recognizes the acknowledge and resets the *OutReq* bit according to the standard four-phase handshaking followed by resetting the *OutAck* bit through the software. This procedure takes place through the whole process of reading out the result. For finalizing the computation the *Done* bit is reset to zero.

### Verification of Modules

Functional verification of the cryptographic modules is a very important aspect as otherwise the achieved results are not reliable. As no debug interface is available another simple way had to be found. Therefore, the hardware resources of the FPGA board were analyzed with the outcome of available I/O ports, one LED and one button. Simplest solution regarding this resources is to use the LED for giving information on the state of execution. Table 9.3 shows the description for each LED state during the verification process.

Using the LED as status signal is only helpful with a test program and reliable test vectors. The former is presented in Appendix D.3. It tests all software and hardware modules in each permissible configuration by applying stimuli vectors and checking the results against the expected response vectors. The test vectors can either be stored in the same ROM as the program code or separately in explicit dedicated ROMs. In this work the test vectors are stored in dedicated ROMS in order to avoid congestion of the program space. For this two test ROMs each with a size of 8 kB were used to store test vectors

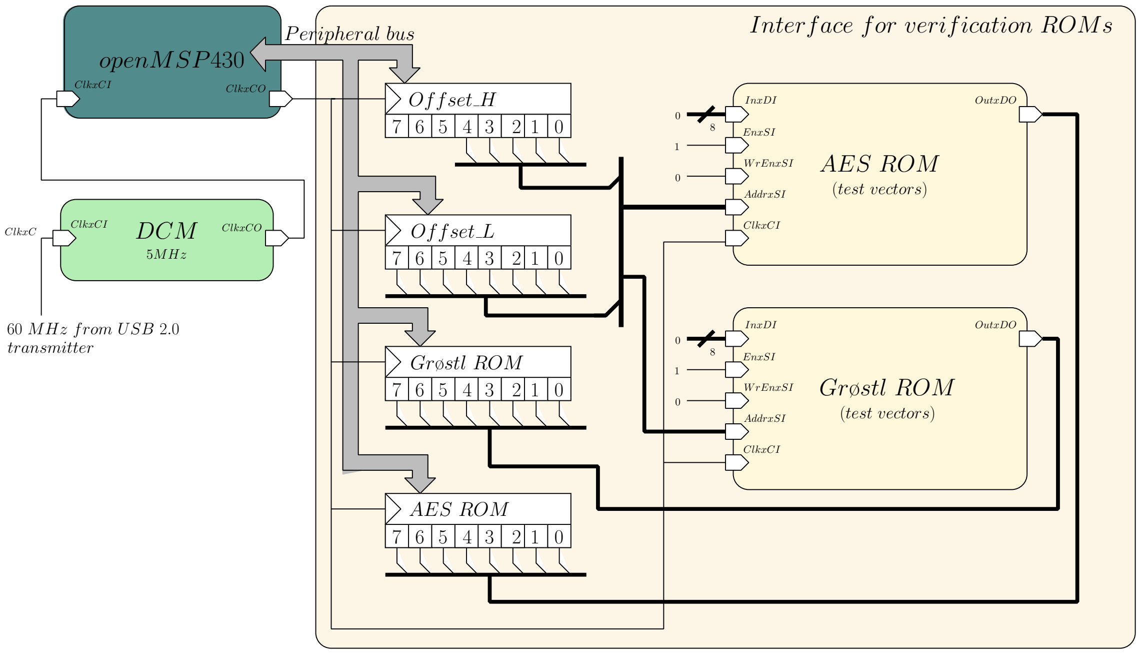


Figure 9.7: Peripheral bus used for connecting the verification ROMs to the openMSP430 core.

for AES-128 and Grøstl-224. The connection to the microcontroller was established again over the peripheral bus as presented in Figure 9.7.

Due to the limited peripheral address space of 512 bytes a direct access to the 8 kB address space of the two test ROMs was not possible. For bypassing this problem four registers were introduced. Two of them are dedicated for the address itself and get modified via software. The other two store the content of the AES or Grøstl ROM at the position represented by the address in the first two registers. These are written by hardware and read by software.

For accessing a test vector sequence the following procedure has to be applied. First, the start address of the test vector is written into the dedicated registers of the interface. Both test ROMs—AES and Grøstl—deliver the corresponding data to the applied address as both are enabled all the time. These values enter the two registers of the interface which must be in addition accessible over software. In order to ease the data access, each read access on these registers increases the address by one. Therefore, no explicit increment functionality is required. Caution has to be exercised due to the fact that the address registers for AES and Grøstl are shared (read access on AES test vector increases also the address for Grøstl).

### 9.2.4 Cryptographic Library

For establishing a fair hardware/software comparison a cryptographic library had to be developed in addition. Goal was to support the same cryptographic primitives in hardware and software.

### Cryptographic Hardware Wrappers

For the cryptographic hardware primitives various wrappers were introduced which handle the communication (handshaking) as well as the configuration (select unit, mode of operation and frequency) with/of the hardware modules. As for both AES and Grøstl no padding function is available, the data size must be either 16 bytes (AES) or a multiple of 64 bytes (Grøstl). The keysize for AES must be always 16 bytes. Furthermore, both variants overwrite the input data with the result. Each method is additionally available as slow and fast variant. The slow method configures the hardware unit for 5 MHz. The fast method on the other hand configures the hardware unit for 20 MHz. The available wrappers for computing AES-128 and Grøstl-224 in hardware are listed below:

- int hw_single_aes_encrypt_w_slow_clock(unsigned char *msg, unsigned char *key)
- int hw_single_aes_encrypt_w_fast_clock(unsigned char *msg, unsigned char *key)
- int hw_single_aes_decrypt_w_slow_clock(unsigned char *msg, unsigned char *key)
- int hw_single_aes_decrypt_w_fast_clock(unsigned char *msg, unsigned char *key)
- int hw_single_groestl_w_slow_clock(unsigned char *msg, int length)
- int hw_single_groestl_w_fast_clock(unsigned char *msg, int length)
- int hw_shared_aes_encrypt_w_slow_clock(unsigned char *msg, unsigned char *key)
- int hw_shared_aes_encrypt_w_fast_clock(unsigned char *msg, unsigned char *key)
- int hw_shared_aes_decrypt_w_slow_clock(unsigned char *msg, unsigned char *key)
- int hw_shared_aes_decrypt_w_fast_clock(unsigned char *msg, unsigned char *key)
- int hw_shared_groestl_w_slow_clock(unsigned char *msg, int length)
- int hw_shared_groestl_w_fast_clock(unsigned char *msg, int length)

### Cryptographic Software Primitives

The straightforward software variants were written completely in C with keeping a low-memory footprint in mind. Therefore, no execution speed optimizations like loop unrolling and the like were used. Padding again was excluded to be compatible with the hardware components. The AES specification [51] and the Grøstl specification [15] served as guidelines for the implementations.

AES encryption was realized in the following way. Message and key are both stored in an 16-byte unsigned char buffer to initialize the state matrix and the key matrix that consist of a two-dimensional unsigned char array (4×4 matrix). Afterwards, the state matrix is updated with the key matrix containing the master key for encryption. This step represents the initial add round key. As the master key no longer is needed the key matrix is updated through the key generation with the next round key. The initial phase has now been completed and is followed by the round transformation consisting of the *ShiftRows*, *SubBytes*, *AddRoundKey* and *MixColumns* steps, executed nine times in this order. *ShiftRows* modifies the state matrix by shifting each row to the left by the given offset as stated in the AES specification. Afterwards, each byte of the state matrix is substituted through the AES Sbox which was realized as Look Up Table (LUT). The *AddRoundKey* step followed next updates the state matrix with the actual stored round key which gets updated in the following as it is not needed anymore. The *MixColumns* operation transforms each column of the State matrix separately as this computation is equal to a matrix multiplication. In order to realize the various coefficient multiplications temporary values were computed representing either a multiplication by two of a previous value or a combination (XOR) of two previously computed values. The multiplication by two was chosen as it is easy to realize in hardware.

**Hardware Multiplication by Two:**
First, the highest bit is checked if it is set. If not only a shift operation by one to the left is performed. In the other case the shift operation is followed by an XOR operation with an irreducible polynomial of the finite field ($GF(2^8)$), represented by the value 0x1B.

After executing the round transformation nine times a final round follows whereas the *MixColumns* operation is skipped. Afterwards, the state matrix contains the encrypted message which gets written back into the input buffer and represents so the result of the computation.

The AES decryption is build quite similarly. Differences are that for the initial add round key step first the last round key has to be computed from the master key by which the state matrix gets updated next. Furthermore, the round keys are computed by the round-key generation in inverse order compared to the AES encryption operation. The round transformation consists of the same functions with a different order of execution, namely that the *MixColumns* is executed before the *AddRoundKey*. Furthermore, *ShiftRows* shifts the rows of the state matrix to the right. *SubBytes* additionally exhibits a different LUT representing the inverse operation for the forward Sbox. As mentioned before the *AddRoundKey* operation computes always the previous round key instead of the next one. *MixColumns* just uses different coefficients but is realized in the same way as for encryption. After the round transformation was applied again a final round follows where *MixColumns* is skipped.

As Grøstl must be capable of computing the hash for messages containing more than 512 bytes (block size), the first step is to compute the number of message blocks from the given length. Next, the intermediate hash-value buffer consisting of 64 unsigned chars is initialized according to the Grøstl specification. Afterwards, a for loop is required to compute subsequently each message block whereas always first a $P$ permutation is applied followed by a $Q$ permutation. These permutations have the following structure. First, the state matrix consisting of a two-dimensional unsigned char array ($8 \times 8$ matrix) is initialized either for a $P$ permutation by XORing the intermediate hash value with the message or simply with the message alone for the $Q$ permutation. Afterwards, a round transformation with 10 rounds is applied consisting of *AddRoundConstant*, *SubBytes*, *ShiftBytes* and *MixBytes* operations. The *AddRoundConstant* operation XORs different fixed values with the state matrix depending on the actual round and the kind of permutation. In order to perform *SubBytes* the same LUT as used for the AES encryption is deployed. *ShiftBytes* is similar to *ShiftRows* of the AES encryption operation as it shifts also each row by a given offset to the left. Differences are the offsets and furthermore that they are different for both permutations. The last operation of the round transformation is *MixBytes* which has its equivalent in the AES *MixColumns* operation. Only the matrix is bigger and contains different constants. The implementation on the other hand was handled in the same way. In order to finalize a permutation the intermediate hash value is XORed with state matrix. After processing each message block a truncation step follows. There, the $P$ permutation is executed again with a zero message block and the intermediate hash value as input. The output gets truncated so that the last 28 bytes remain. Moreover, the output is stored to the input message buffer.

The function calls for the previous described software variants are listed below:

- void aes_encrypt(unsigned char *msg, unsigned char *key);
- void aes_decrypt(unsigned char *msg, unsigned char *key);
- void groestl_hash(unsigned char *msg, unsigned int length);

## 9.3 Results

After the description of the system and its components we are now in a position for going into details of the acquired results. Therefore, first the influence of external data transfer on the overall timing will be analyzed in Subsection 9.3.1 followed by the Software/Hardware comparison in Subsection 9.3.2.

### 9.3.1 Influence of External Data Transfer on Overall Timing

Generally, each engineer should be aware of the timing drawbacks if going externally from a system (e.g., reading/writing from/to external memory) and the problems which can occur as mentioned in Chapter 6. It makes no sense to realize each small (simple) component as a separate chip and connect it to the system where it is needed. Furthermore, several ways exist to connect components with each other. Some offer a high flexibility and others are quite restricted depending on their field of application. In this work all external data transfer is handled over a four-phase handshaking as a simple integration into various environments—also including different clock domains—should be possible. This protocol is not the fastest but highly reliable (each data transfer must be requested and acknowledged) which is more important in the field of secure applications. The high time consumption is easy to explain as both the transmitter and the receiver have to change a signal twice in order to successfully transfer a single byte. Other protocols have for example only an initial phase after which the bytes are transferred cycle by cycle (in case of a synchronous system).

Table 9.4 and Table 9.5 present the timings for all hardware modules operated at a frequency of either 5 MHz or 20 MHz. The lower frequency is named from now on $f_L$ and the higher frequency $f_H$. As the shared version (GrÆStl) requires for all modes of operation exactly the same amount of cycles as the single versions, it will not be targeted separately anymore. Furthermore, for the numbers taking the interface (Ifc.) into account it is important to know that the microcontroller is operated at the lower frequency.

With the presented timings for both frequencies the speed-up ($S$) factors for all modes of operation and additionally for either taking the interface into account or leaving it out

Table 9.4: Timing for all modes of operation at **5 MHz** clock frequency.

| **Operation** | **w/o Ifc.** [$\mu s$] | **w/ Ifc.** [$\mu s$] |
|---|---:|---:|
| AES enc. | 147 | 487 |
| AES dec. | $270^1$ | $610^2$ |
| Grøstl | 613 | 1,268 |

$^{1,2}$ 205 $\mu s$ and 545 $\mu s$, respectively if last round key is stored in output register. Only available for GrÆStl unit.

Table 9.5: Timing for all modes of operation at **20 MHz** clock frequency.

| **Operation** | **w/o Ifc.** [$\mu s$] | **w/ Ifc.** [$\mu s$] |
|---|---:|---:|
| AES enc. | 36.75 | 375 |
| AES dec. | $67.5^1$ | $405^2$ |
| Grøstl | 153.25 | 805 |

$^{1,2}$ 51.25 $\mu s$ and 388.75 $\mu s$, respectively if last round key is stored in output register. Only available for GrÆStl unit.

Table 9.6: Speed-up comparison for all modes of operation with and without the interface taken into account.

| **Operation** | $S_{w/o\ Ifc.}$ | $S_{w/\ Ifc.}$ |
|---|---:|---:|
| AES encryption | 4 | ~1.3 |
| AES decryption | 4/4 | ~1.5/~1.4 |
| Grøstl | 4 | ~1.6 |

can be computed. Therefore, Equations 9.1 and 9.2 were used. The so gained results are presented in Table 9.6.

$$S_{w/o\ Ifc.} = \frac{T_{w/o\ Ifc.@f_L}}{T_{w/o\ Ifc.@f_H}} \tag{9.1}$$

$$S_{w/\ Ifc.} = \frac{T_{w/\ Ifc.@f_L}}{T_{w/\ Ifc.@f_H}} \tag{9.2}$$

For the variant not taking the interface into account a linear speed-up factor of four is obtained represented through the $S_{w/o\ Ifc.}$ column in Table 9.6 and represents also the natural influence one would expect through the given frequency increase from $f_L$ to $f_H$.

This speed-up factor is not informative at all when keeping a functional system in mind. Therefore, the handshaking procedure has to be taken into account which is represented through the $S_{w/\ Ifc.}$ column in Table 9.6. As the microcontroller is operated with the lower frequency heavy losses regarding the speed-up factor can be observed. The reason for that is that the increase of the frequency for the hardware units from $f_L$ to $f_H$ only shortens the time for the core computation but not for the handshaking. This circumstance entails that Grøstl with 3,061 core cycles exhibits a higher speed-up factor than AES in all modes of operation (652 core cycles for encryption and 955/1,269 core cycles for decryption).

Regarding this numbers a bigger difference between the speed-up factors of AES and Grøstl as presented in Table 9.6 would be expected. The explanation for this unexpected small difference is the handshaking procedure for AES and Grøstl which is negligible affected by the frequency increase of the hardware units. AES requires 32 input bytes (data + master key) and 16 output bytes whereas Grøstl needs 64 and 28 bytes ending up in 1,700 MCU clock cycles for the AES handshaking and 3,275 MCU clock cycles for Grøstl handshaking when operated with the lower frequency $f_L$. Switching from $f_L$ to $f_H$ negligibly influences the cycle counts as the bottleneck is always the microcontroller which is operated all over with the lower frequency $f_L$ and has additional work to do (polling I/O pins, interrupt handling, ...) reflected through short breaks in the processing

Table 9.7: Results for the AES-128 and Grøstl-224 software routines targeting a low-memory footprint. They were achieved on the dedicated target platform, the openMSP430 running at a frequency of 5 MHz.

| **Operation** | **Time** [$\mu s$] | **ROM Usage** [$Bytes$] |
|---|---|---|
| AES enc. | 30,000 | 2,200 |
| AES dec. | 32,000 | 2,200 |
| Grøstl | 290,000 | 1,200 |

of the peripheral bus. The hardware units instead react immediately on changes of the handshaking control lines. Therefore, increasing the frequency of the hardware units while leaving the frequency of the microcontroller constant entails no real profit as the reaction of the hardware units may occur earlier but gets processed by the microcontroller at the same moment as it would be with the lower frequency. Choosing a frequency for the hardware units which is lower than the one from the microcontroller increases for the same reason the number of required cycles of the handshaking. This fact explains why the speed-up for the Grøstl computation is nearly the same as for the AES computation.

### 9.3.2 Software/Hardware Evaluation

The software versus hardware evaluation is based on the straightforward implementation of AES-128 and Grøstl-224 on the openMSP430 with the focus on a low-memory footprint and the appertaining hardware modules both already described. The comparison is only concerned with the time required to compute AES/Grøstl from reading in the first byte until reading out the last byte. Table 9.7 presents the achieved results for the software routines.

Comparing the software routines with the hardware-accelerated computations shows that they need at least around 50 times longer which can also be seen in Table 9.8. With the computational complexity measured for example by the amount of cycles required by the hardware components to fulfil a desired computation also the efficiency of the hardware accelerated computation raises. The lowest computational effort is represented by AES whereas Grøstl exhibits the highest complexity. Regarding the AES algorithm encryption was expected to feature a lower hardware-acceleration efficiency than the decryption which is in fact in this software/hardware version not the case. The reason for that is the key scheduler which computes in the software variant always all round keys in advance which contrasts to the hardware version where each byte of the round key is computed consecutively. Therefore the software variant features the same computational effort for computing the next/previous round key. The hardware components on the other hand exhibit a higher complexity for computing the previous round key.

Table 9.8: Comparison of the AES-128 and Grøstl-224 software routines/hardware components executed/connected to the dedicated target platform, which is the openMSP430 running at a frequency of 5 MHz. The hardware components are operated with a speed of either 5 MHz or 20 MHz.

| **Operation** | **Speed-up through HW running at 5 MHz** | **Speed-up through HW running at 20 MHz** |
|---|---|---|
| AES enc. | 61 | 80 |
| AES dec. | 52[1] | 79[2] |
| Grøstl | 229 | 360 |

[1,2] 59/82 if last round key is stored in output register. Only available for GrÆStl unit.

# Chapter 10

# Conclusions

This work has been split into two constructive parts. First, GrÆStl a combined hardware architecture for the Advanced Encryption Standard (AES) and Grøstl. GrÆStl was developed for low-resource devices and aims for high flexibility by targeting both Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) platforms. In order to obtain an expressive statement about the benefit of combining AES and Grøstl, a chip named Chameleon was produced. Chameleon contains the designs of the stand-alone versions for AES and Grøstl as also the combined version GrÆStl. The chip was taped-out, produced and verified successfully. In the second part an FPGA system was developed containing the openMSP430 microcontroller and the ported cryptographic modules. Target was to establish a reliable system for making a fair hardware/software comparison and to show the influence of the used interface on the overall computation time.

In detail, the first part targets the benefits of combining AES and Grøstl. AES on the one hand is the by far most widely spread block cipher since its standardization in 2001 by the National Institue of Standards and Technology (NIST). Grøstl on the other hand is one of the final-round candidates of the SHA-3 hash-function competition, which will announce its winner in late of 2012. Therefore, AES is already and Grøstl could become an algorithm widely used to achieve data confidentiality, integrity and authenticity. Due to the fact that AES and Grøstl feature several similarities such as a common S-box or similar diffusion layers an integration into one module looks promising for the future. In addition to the cost savings regarding area occupation authenticated encryption could be fulfilled in one step with such a design.

The whole design is based on standard cells only and contains therefore no process-dependent technologies like Random Access Memory (RAM) macros, Block RAMs or Digital Signal Processors (DSPs) as they might not be available on all Complementary Metal-Oxide Semiconductor (CMOS) platforms and furthermore would have to be recreated in case CMOS-process is changed. Moreover, in case of FPGAs, resources such as Block RAMs or DSPs might be already used by other system components. Therefore, the design is highly flexible and portable to other platforms.

In order to lower the area requirements the following optimization techniques are integrated in the combined hardware architecture: (1) the AES datapath was integrated into the Grøstl one, starting with a mapping of the AES state into the Grøstl state matrix (one half of the Grøstl matrix is occupied by the AES state and the actual AES round key) to avoid the need of additional memory, (2) the I/O registers as well as the complete state matrix are made out of shift registers in order to provide high flexibility (for ASICs as well as FPGAs) and to avoid the implementation of *ShiftBytes* and *ShiftRows*, (3) the *AddRoundConstant* function was implemented according to the tweaked Grøstl specification, cf. [15], instead of the Grøstl-0 specification (from the 31st of October 2008) as given in most of the related work, (4) the S-boxes were reused for AES and Grøstl and further they were also reused to increase the performance of AES round-key generation, (5) *MixBytes* and *MixColumns* were combined and finally (6) the I/O registers are shared in order to avoid forward round-key generation during decryption reducing the overall number of clock cycles by 330.

As result, a chip named Chameleon was taped-out and produced on a 0.18 µm CMOS process technology from UMC. Chameleon contains GrÆStl, the first combined hardware implementation fabricated as ASIC, occupying 17.1 kGE in total. In order to establish a fair comparison, AES-128 and Grøstl-224 were also included as stand-alone versions. It showed that the integration of AES into Grøstl needs an overhead of only 10 % which corresponds to 1.5 kGE. The smallest AES version for both encryption and decryption was presented by Feldhofer et al. [11] in 2005. Their design requires 3.4 kGE after the synthesis and 4.4 kGE after the backend design. Moradi et al. [45] reported in 2011 an AES encryption-only version with a complexity of about 2.4 kGE after synthesis. The result of this work is a combined hardware architecture in which AES (encryption and decryption) was implemented with about 60 % and 40 % less resources than the designs from Feldhofer et al. and Moradi et al.. The small area requirements and the low power consumption (gate-level power analysis based on switching activities) of about 13 µW for AES encryption/decryption and 20 µW for Grøstl at 100 kHz make the design highly suitable for low-resource devices. Chameleon was successfully verified after production and features a maximum frequency of about 133 MHz. The power consumption for GrÆStl in AES encryption mode is 63 µW at 100 kHz.

The efficiency of Chameleon on FPGA platforms was evaluated through porting the design on various Xilinx FPGAs. The only modification was the removal of the reset functionality from the registers in order to lower the area requirements according to the SRL-16 [7, 67] mode. In particular, on a Spartan-3 FPGA, the stand-alone AES and Grøstl implementations outperform existing solutions by a factor of 4.8 and about 1.6. To the best of our knowledge the stand-alone versions are the smallest designs known so far. GrÆStl instead is in fact about 2 % larger than the sum of the stand-alone versions for AES and Grøstl. The reason for that is the state matrix which had to be split up in the half in order to establish the integration of AES into Grøstl. Therefore, the SRL-16 mode can not be fit to the design as optimal as otherwise.

In the second part of the thesis an FPGA system containing the openMSP430 microcontroller and the ported cryptographic modules was developed. The target was to show the influence of the used interface on the overall computation time using two different clock frequencies and to establish a fair hardware/software comparison. Therefore, the cryptographic modules were connected to the openMSP430 and made accessible through software routines. Moreover, the stand-alone software versions of AES-128 and Grøstl-224 were developed with keeping a low-memory footprint in mind. It showed that the four-phase handshaking makes the design highly flexible regarding various clock domains but reduces the efficiency of the outsourced computations substantially. The speedup through an increase in the clock frequency of the cryptographic modules from 5 to 20 MHz by holding the frequency of the openMSP430 constantly at 5 MHz reduced therefore from 4 (without interface) to only 1.6 in case of Grøstl and to only 1.3 for AES encryption. The software variants for AES encryption/decryption occupied 2,200 Bytes of ROM whereas

Grøstl needs only 1,200 Bytes due to the missing round-key generation. In view of execution time, AES encryption needs around 30,000 µs, AES decryption 32,000 µs and Grøstl 290,000 µs. The speedup through outsourcing the computation into dedicated hardware was therefore expected to be highest for Grøstl, which turned out to be true. A speedup of 229 for Grøstl and only 61/52 for AES encryption/decryption was achieved. If the last round key is already known at the beginning of a decryption (forward round-key computation can be skipped) the speedup is increased to 82.

Summarizing, this work achieved an integration of the AES-128 datapath into the Grøstl-224 datapath with an overhead of only 10 % which corresponds to 1.5 kGE. The benefit of the combination process is therefore the smallest AES encryption and decryption unit known so far. Regarding FPGA platforms the smallest stand-alone versions for AES-128 and Grøstl-224—not using distributed RAM or Block RAMs—have been created. Further, through the FPGA system of part two a hardware accelerated speedup for AES encryption, AES decryption and Grøstl of 61, 52 and 229 was achieved. The influence of the interface for the cryptographic modules—frequency of cryptographic modules was increased from 5 to 20 MHz while leaving the frequency of the microcontroller constantly at 5 MHz—showed a speedup reduction from 4 to only 1.6 in case of Grøstl and 1.3 for AES encryption.

# Appendix A

# Definitions

## A.1 Abbreviations

| | |
|---|---|
| **AES** | Advanced Encryption Standard |
| **ASIC** | Application-Specific Integrated Circuit |
| **BRAM** | Block RAM |
| **CBC** | Cipher-Block Chaining |
| **CFB** | Cipher Feedback |
| **CLB** | Configurable Logic Block |
| **CMOS** | Complementary Metal-Oxide Semiconductor |
| **CRT** | Counter |
| **DES** | Data Encryption Standard |
| **D-ITET** | Department of Information Technology and Electrical Engineering |
| **DSP** | Digital Signal Processor |
| **ECB** | Electronic Code Book |
| **FIPS** | Federal Information Processing Standard |
| **FF** | Flip Flop |
| **FPGA** | Field Programmable Gate Array |
| **FSM** | Finite State Machine |
| **GE** | Gate Equivalent |
| **HW** | Hardware |
| **MAC** | Message Authentication Codes |
| **MCU** | Microcontroller Unit |
| **MD** | Message Digest |
| **MSPGCC** | Port of the GCC Toolchain for the Texas Instruments MSP430 Family |
| **NIST** | National Institute of Standards and Technology |
| **OFB** | Output Feedback |
| **RAM** | Random-Access Memory |
| **RC** | Rivest Cipher |
| **RFID** | Radio-Frequency Identification |
| **ROM** | Read-Only Memory |
| **SFIT** | Swiss Federal Institute of Technology |
| **SHA** | Secure Hash Algorithm |
| **SLICEL** | SLICEL (L = logic) |
| **SLICEM** | SLICEM (M = memory) |
| **SOC** | System On Chip |

| | |
|---|---|
| **SRL** | Shift-Register Logic |
| **SRAM** | Static Random-Access Memory |
| **SW** | Software |
| **TUG** | Graz University of Technology |
| **UMC** | United Microelectronics Corporation |
| **VCD** | Value Change Dump |
| **VHDL** | Very High Speed Integrated Circuit Hardware Description Language |

# Appendix B

# Chameleon - ASIC

## B.1 General Features

- Package
  - QFN 56 (8x8), only 48 pins used
- Pads/Padframe[1]
  - 4-pad power-supply pins
  - 4-pad core power-supply pins
  - 40 I/O pins
- Chip size
  - 1.565 mm x 1.565 mm (including seal ring)
- Core size
  - 1.09926 mm x 1.09926 mm = 1.21 mm$^2$ (incl. global and power routing)
  - Conforms to logic complexity of 90,000 GE estimated for 70 % occupation of maximum core area.
- Supply Voltages
  - 3.3 V pad supply voltage
  - 1.8 V core supply voltage
- Max. operating speed
  - 125 MHz @ max. supply voltage for both pads and core.
- Core area occupation[2]
  - AES-128: 5,035 GE
  - Grøstl-224: 15,551 GE

[1]Spacings and the location of the power pads and the clock pad have been predefined through the used standard bonding diagram provided by the Microelectronics Design Center of ETH Zürich.

[2]Complexity of cryptographic modules exclusive interface (four-phase handshaking)

|   |   |
|---|---|
| – GrÆStl: | 17,137 GE |
| – Overall: | 38,254[3] GE |

- Cryptographic performance (number of cycles without interface)

|   |   |
|---|---|
| – AES-128 (enc./dec.): | 652/1,269 cycles |
| – Grøstl- 224 (hash): | 3,061 cycles |
| – GrÆStl (enc./dec./hash): | 652/1,269[4]/3,061 cycles |

- Estimated power consumption[5] for the cryptographic modules

|   |   |
|---|---|
| – AES-128: | 13.4 µW/MHz |
| – Grøstl- 224: | 20.3 µW/MHz |
| – GrÆStl (enc. and dec./hash): | 20.2/21.6 µW/MHz |

[3]Including the three cryptographic modules and their shared interface set on top of them.
[4]955 cycles if enc./dec. with same master key was carried out immediately before.
[5]Estimations based on toggling counts of the nodes gained through a gate-level simulation with random test vectors.

## B.2 Pinout

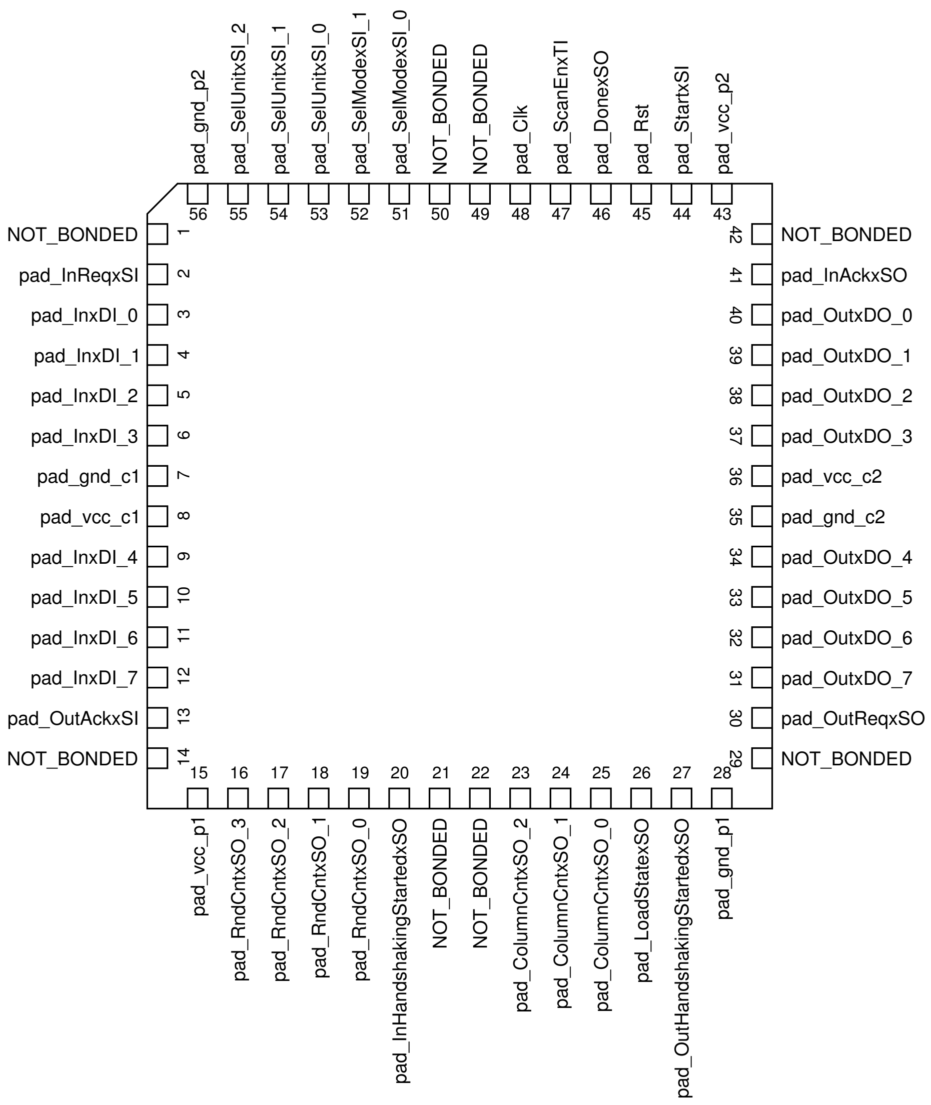


Figure B.1: Pinout of the taped-out chip named *Chameleon*.

## B.3 Pad Description

| Pad(s) | Description |
|---|---|
| pad_vcc_p1, pad_vcc_p2<br>pad_gnd_p1, pad_gnd_p2 | Power supply for the padframe (3.3 V). |
| pad_vcc_c1, pad_vcc_c2<br>pad_gnd_c1, pad_gnd_c2 | Power supply for the core (1.8 V). |
| pad_InReqxSI, pad_InAckxSO<br>pad_OutReqxSO, pad_OutAckxSI | I/O-handshaking signals for un-/loading data. |
| pad_InxDI_[7:0][1]<br>pad_OutxDO_[7:0] | 8-bit parallel data I/O (reused as scanchain I/O) |
| pad_ScanEnxTI | If set to high chip operates in test mode in which the functionality can be verified using eight parallel scanchains accessible over the data I/O pads. |
| pad_Clk<br>pad_Rst | Clock Input<br>System Reset (Asynchronous) |
| pad_SelUnitxSI_[2:0] | Select cryptounit<br>000...Disable All Units<br>001...AES<br>010...Grøstl<br>011...GrÆStl in AES mode of operation<br>100...GrÆStl in Grøstl mode of operation |
| pad_SelModexSI_[1:0] | Select AES mode of operation<br>0X...Encryption<br>1X...Decryption<br>Select Grøstl mode of operation<br>00...Intermediate block<br>01...First block<br>10...Last block<br>11...First and last block |
| pad_StartxSI<br>pad_DonexSO | Must be high for duration of input handshaking<br>High for duration of output handshaking |
| pad_RndCntxSO_[3:0]<br>pad_ColumnCntxSO_[2:0] | Computation progress in term of actual round<br>Detailed computation progress in term of actual column of actual round |
| pad_InHandshakingStartedxSO<br>pad_OutHandshakingStartedxSO | High if input handshaking active<br>High if output handshaking active |
| pad_LoadStatexSO | High during loading of internal state<br>→AES Unit: Internal state loaded **during** reading in of data<br>→Other Units: Internal state loaded **after** reading in of data into input registers |

## B.4 Interface Description

Chameleon was designed in a way offering high flexibility regarding various platforms as well as clock domains. Therefore, the I/O interface was implementing via a four-phase handshaking. This scheme is on the one hand highly flexible regarding various clock domains but on the other hand also slower as other interfaces like, for example, the two-phase handshaking. A detailed overview of the finite state machine representing the four-phase handshaking that is used in Chameleon is presented in Figure B.2.

The interface of the taped-out chip named Chameleon has three vulnerabilities regarding its reliability. These are highlighted by red lines and dashed lines in the detailed representation of the finite-state machine. Additionally, a number is placed beside in order to describe the vulnerabilities and their impact in an organized manner which can be found in the following enumeration.

- 1.) Missing $InAckxSO <=' 1'$ entry in state *innotreq*
  In the original four-phase protocol the acknowledge line is only set back to its original state after recognizing the same behaviour on the request line. The taped-out chip nevertheless exhibits the behaviour that the reset of the acknowledge line is independent from the request line and is therefore each time exactly one cycle at high level. This can cause an erroneous behaviour in case the transmitter is operated with a slower clock. If so, it is possible that the acknowledge line gets reset to its original value before the transmitter recognizes this. As a result the data exchange will end up in a deadlock as the receiver waits for an action of the transmitter and vice versa!

- 2.) Jump to wrong state
  After recognizing that the receiver has acknowledged the reception of the data the request is set back to its original state. Afterwards, an idle time occurs which is as long as the time until the receiver resets the acknowledge line. However, the implemented interface skips this idle time and signals directly afterwards, that the next data item is ready to be transferred.

- 3.) Missing state
  Due to the wrong jump mentioned above a state is omitted. The impact is the same as already described before.

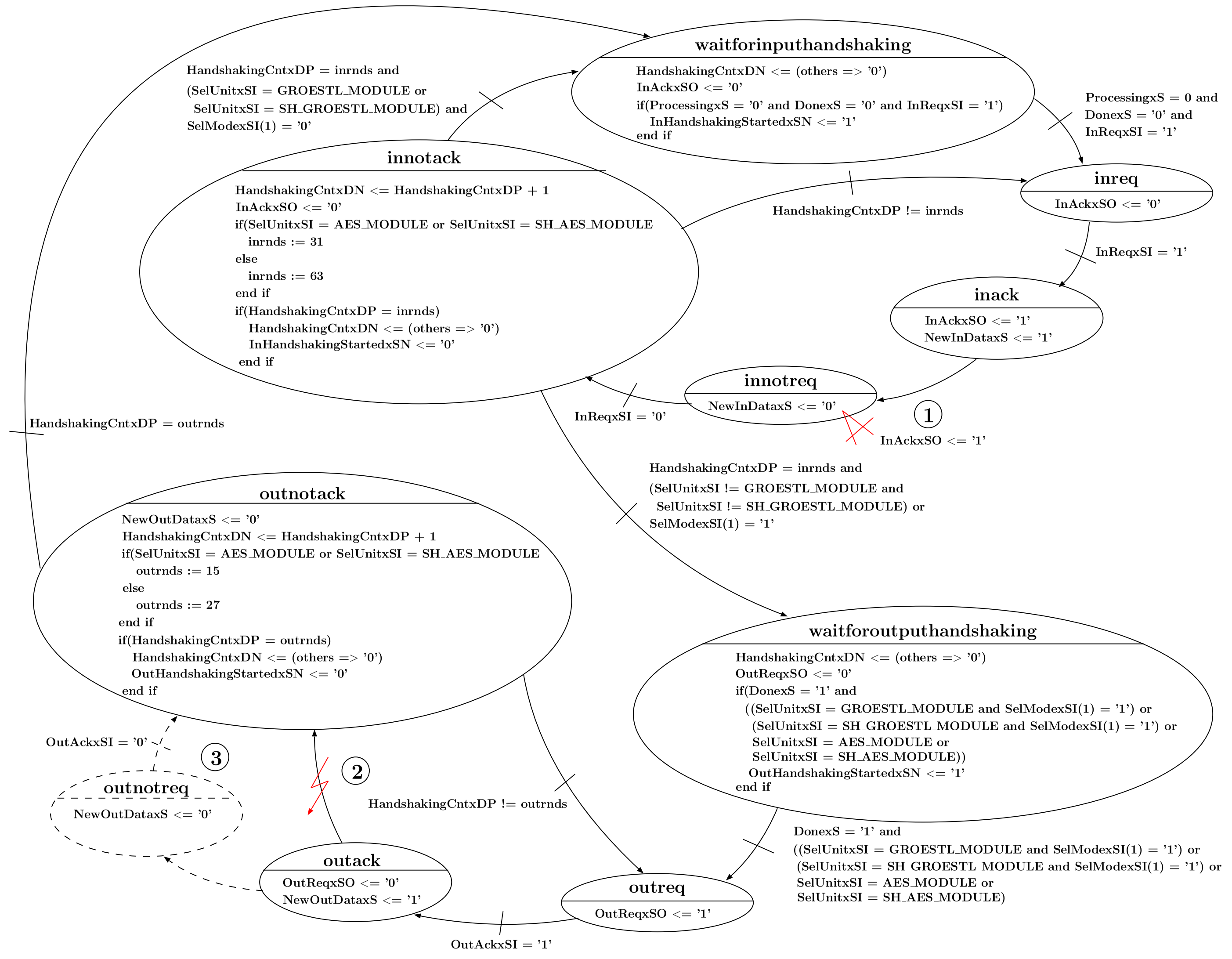

waitforinputhandshaking
HandshakingCntxDN <= (others => '0')
InAckxSO <= '0'
if(ProcessingxS = '0' and DonexS = '0' and InReqxSI = '1')
InHandshakingStartedxSN <= '1'
end if
ProcessingxS = 0 and
DonexS = '0' and
InReqxSI = '1'
inreq
InAckxSO <= '0'
InReqxSI = '1'
inack
InAckxSO <= '1'
NewInDataxS <= '1'
innotreq
NewInDataxS <= '0'
1
InAckxSO <= '1'
InReqxSI = '0'
innotack
HandshakingCntxDN <= HandshakingCntxDP + 1
InAckxSO <= '0'
if(SelUnitxSI = AES_MODULE or SelUnitxSI = SH_AES_MODULE
inrnds := 31
else
inrnds := 63
end if
if(HandshakingCntxDP = inrnds)
HandshakingCntxDN <= (others => '0')
InHandshakingStartedxSN <= '0'
end if
HandshakingCntxDP != inrnds
HandshakingCntxDP = inrnds and
(SelUnitxSI = GROESTL_MODULE or
SelUnitxSI = SH_GROESTL_MODULE) and
SelModexSI(1) = '0'
HandshakingCntxDP = inrnds and
(SelUnitxSI != GROESTL_MODULE and
SelUnitxSI != SH_GROESTL_MODULE) or
SelModexSI(1) = '1'
HandshakingCntxDP = outrnds
waitforoutputhandshaking
HandshakingCntxDN <= (others => '0')
OutReqxSO <= '0'
if(DonexS = '1' and
((SelUnitxSI = GROESTL_MODULE and SelModexSI(1) = '1') or
(SelUnitxSI = SH_GROESTL_MODULE and SelModexSI(1) = '1') or
SelUnitxSI = AES_MODULE or
SelUnitxSI = SH_AES_MODULE))
OutHandshakingStartedxSN <= '1'
end if
DonexS = '1' and
((SelUnitxSI = GROESTL_MODULE and SelModexSI(1) = '1') or
(SelUnitxSI = SH_GROESTL_MODULE and SelModexSI(1) = '1') or
SelUnitxSI = AES_MODULE or
SelUnitxSI = SH_AES_MODULE)
outnotack
NewOutDataxS <= '0'
HandshakingCntxDN <= HandshakingCntxDP + 1
if(SelUnitxSI = AES_MODULE or SelUnitxSI = SH_AES_MODULE
outrnds := 15
else
outrnds := 27
end if
if(HandshakingCntxDP = outrnds)
HandshakingCntxDN <= (others => '0')
OutHandshakingStartedxSN <= '0'
end if
HandshakingCntxDP != outrnds
outreq
OutReqxSO <= '1'
OutAckxSI = '1'
outack
OutReqxSO <= '0'
NewOutDataxS <= '1'
2
3
outnotreq
NewOutDataxS <= '0'
OutAckxSI = '0'

Figure B.2: Finite state machine handling the I/O communication for the ASIC and the FPGA implementation, respectively.

# Appendix C

# Chameleon - FPGA

## C.1 Xilinx Spartan-3 and Spartan-6 Results

Table C.1: Post place-and-route results for Xilinx Spartan-3 and Spartan-6 FPGAs considering the interface (original design - no adaptation due to SRL-16 mode).

| | Spartan-3 | | | Spartan-6 | | |
|---|---|---|---|---|---|---|
| | **AES** | **Grøstl** | **GrÆStl** | **AES** | **Grøstl** | **GrÆStl** |
| Number of slice LUTs | 871 | 1,045 | 1,876 | 560 | 709 | 1,426 |
| Number used as logic | 863 | 892 | 1,757 | 478 | 551 | 1,119 |
| Number used as shift reg. | 8 | 128 | 64 | 4 | 64 | 32 |
| Number used as a route-thru | 0 | 25 | 55 | 78 | 94 | 275 |
| Number of slice registers | 292 | 558 | 856 | 286 | 546 | 833 |
| Frequency(MHz) | 35 | 40 | 25 | 50 | 55 | 35 |
| Number of occupied slices | 525 | 682 | 1,203 | 187 | 232 | 400 |

## C.2 Xilinx Virtex-5 and Virtex-6 Results

Table C.2: Post place-and-route results for Xilinx Virtex-5 and Virtex-6 FPGAs considering the interface (original design - no adaptation due to SRL-16 mode).

| | Virtex-5 | | | Virtex-6 | | |
|---|---|---|---|---|---|---|
| | **AES** | **Grøstl** | **GrÆStl** | **AES** | **Grøstl** | **GrÆStl** |
| Number of slice LUTs | 467 | 657 | 1,005 | 508 | 704 | 1,285 |
| Number used as logic | 458 | 548 | 968 | 431 | 535 | 1,062 |
| Number used as shift reg. | 9 | 108 | 36 | 5 | 76 | 64 |
| Number used as a route-thru | 0 | 1 | 1 | 72 | 93 | 159 |
| Number of slice registers | 292 | 563 | 858 | 287 | 555 | 744 |
| Frequency(MHz) | 125 | 135 | 90 | 110 | 115 | 85 |
| Number of occupied slices | 160 | 259 | 369 | 149 | 225 | 363 |

# Appendix D

# FPGA System - AddOn

This chapter presents first in Section D.1 the basic linker script used for building a working ROM content by the system toolchain. Afterwards, in Section D.2, an installation script for installing the MSPGCC toolchain is given as installers for the major operating systems are only given for the old toolchain, the MSPGCC4. Finally, in Section D.3 the test program for verifying the correct functionality of the cryptographic modules is presented.

## D.1 Basic Linker Script for openMSP430 MCUs

This section presents the basic linker script [16] which forms one of the inputs to the system toolchain which is responsible for building a functional ROM content. According to the data memory size and the program memory size the system toolchain modifies the *text* origin and length, *data* origin and length and also the stack offset.

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-msp430","elf32-msp430","elf32-msp430")
OUTPUT_ARCH("msp430")
MEMORY
{
  text   (rx)     : ORIGIN = 0xF800,      LENGTH = 0x800
  data   (rwx)    : ORIGIN = 0x0200,      LENGTH = 0x080
  vectors (rw)    : ORIGIN = 0xffe0,      LENGTH = 0x20
}
/* INCLUDE periph.x */
SECTIONS
{
  /* Read-only sections, merged into text segment.  */
  .hash          : { *(.hash)             }
  .dynsym        : { *(.dynsym)           }
  .dynstr        : { *(.dynstr)           }
  .gnu.version   : { *(.gnu.version)      }
  .gnu.version_d   : { *(.gnu.version_d)  }
  .gnu.version_r   : { *(.gnu.version_r)  }
  .rel.init      : { *(.rel.init) }
  .rela.init     : { *(.rela.init) }
  .rel.text      :
    {
      *(.rel.text)
      *(.rel.text.*)
      *(.rel.gnu.linkonce.t*)
    }
  .rela.text     :
    {
      *(.rela.text)
      *(.rela.text.*)
      *(.rela.gnu.linkonce.t*)
    }
  .rel.fini      : { *(.rel.fini) }
  .rela.fini     : { *(.rela.fini) }
  .rel.rodata    :
    {
      *(.rel.rodata)
      *(.rel.rodata.*)
      *(.rel.gnu.linkonce.r*)
    }
  .rela.rodata   :
    {
      *(.rela.rodata)
      *(.rela.rodata.*)
      *(.rela.gnu.linkonce.r*)
    }
  .rel.data      :
    {
      *(.rel.data)
      *(.rel.data.*)
      *(.rel.gnu.linkonce.d*)
    }
  .rela.data     :
    {
      *(.rela.data)
      *(.rela.data.*)
      *(.rela.gnu.linkonce.d*)
    }
  .rel.ctors     : { *(.rel.ctors)        }
  .rela.ctors    : { *(.rela.ctors)       }
  .rel.dtors     : { *(.rel.dtors)        }
  .rela.dtors    : { *(.rela.dtors)       }
  .rel.got       : { *(.rel.got)          }
  .rela.got      : { *(.rela.got)         }
  .rel.bss       : { *(.rel.bss)          }
  .rela.bss      : { *(.rela.bss)         }
  .rel.plt       : { *(.rel.plt)          }
  .rela.plt      : { *(.rela.plt)         }
  /* Internal text space.  */
```

```
.text :
{
  . = ALIGN(2);
  *(.init)
  *(.init0)  /* Start here after reset.  */
  *(.init1)
  *(.init2)  /* Copy data loop  */
  *(.init3)
  *(.init4)  /* Clear bss  */
  *(.init5)
  *(.init6)  /* C++ constructors.  */
  *(.init7)
  *(.init8)
  *(.init9)  /* Call main().  */
   __ctors_start = . ;
   *(.ctors)
   __ctors_end = . ;
   __dtors_start = . ;
   *(.dtors)
   __dtors_end = . ;
  . = ALIGN(2);
  *(.text)
  . = ALIGN(2);
  *(.text.*)
  . = ALIGN(2);
  *(.fini9)  /*   */
  *(.fini8)
  *(.fini7)
  *(.fini6)  /* C++ destructors.  */
  *(.fini5)
  *(.fini4)
  *(.fini3)
  *(.fini2)
  *(.fini1)
  *(.fini0)  /* Infinite loop after program termination.  */
  *(.fini)
   _etext = . ;
}  > text
.data   : AT (ADDR (.text) + SIZEOF (.text))
{
   PROVIDE (__data_start = .) ;
  . = ALIGN(2);
  *(.data)
  . = ALIGN(2);
  *(.gnu.linkonce.d*)
  . = ALIGN(2);
   _edata = . ;
}  > data
```

```
    PROVIDE (__data_load_start = LOADADDR(.data) );
    PROVIDE (__data_size = SIZEOF(.data) );
  .bss  SIZEOF(.data) + ADDR(.data) :
  {
     PROVIDE (__bss_start = .) ;
    *(.bss)
    *(COMMON)
     PROVIDE (__bss_end = .) ;
     _end = . ;
  }  > data
    PROVIDE (__bss_size = SIZEOF(.bss) );
  .noinit  SIZEOF(.bss) + ADDR(.bss) :
  {
     PROVIDE (__noinit_start = .) ;
    *(.noinit)
    *(COMMON)
     PROVIDE (__noinit_end = .) ;
     _end = . ;
  }  > data
  .vectors  :
  {
     PROVIDE (__vectors_start = .) ;
    *(.vectors*)
     _vectors_end = . ;
  }  > vectors
  /* Stabs debugging sections.  */
  .stab 0 : { *(.stab) }
  .stabstr 0 : { *(.stabstr) }
  .stab.excl 0 : { *(.stab.excl) }
  .stab.exclstr 0 : { *(.stab.exclstr) }
  .stab.index 0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment 0 : { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 : { *(.debug) }
  .line           0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info) *(.gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  PROVIDE (__stack = 0x280) ;
  PROVIDE (__data_start_rom = _etext) ;
  PROVIDE (__data_end_rom   = _etext + SIZEOF (.data)) ;
  PROVIDE (__noinit_start_rom = _etext + SIZEOF (.data)) ;
  PROVIDE (__noinit_end_rom = _etext + SIZEOF (.data) + SIZEOF (.noinit)) ;
}
```

## D.2 Installation Script for MSPGCC Toolchain

The MSPGCC4 can be installed very easily over provided installers for the major operating systems. As such installers are missing for the MSPGCC a script has been developed which handles the download and installation of the binaries.

```
#!/bin/csh

#################################################################
## Set the installation log file
#################################################################
set INSTALL_LOG = "/scratch/msc11h1/mspgcc-files/install_log"
#################################################################
## Clear the file structure
#################################################################
echo "1.) CLEARING FILE STRUCTURE"
rm -r -f /scratch/msc11h1/msp*
#################################################################
## Create file structure
#################################################################
mkdir -p /scratch/msc11h1/mspgcc
mkdir -p /scratch/msc11h1/mspgcc-files
#################################################################
## Install the GNU Multiple Precision Arithmetic Library (GMP)
#################################################################
echo "2.) DOWNLOADING AND INSTALLING GNU MUTLIPLE PRECISION
ARITHMETIC LIBRARY (GMP)"
cd /scratch/msc11h1/mspgcc-files
wget ftp://ftp.gmplib.org/pub/gmp-5.0.2/gmp-5.0.2.tar.bz2 >&
 $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE GNU MULTIPLE PRECISION ARITHMETIC
    LIBRARY (GMP) FAILED!"
  exit -1
endif
tar xvfj gmp-5.0.2.tar.bz2 >& $INSTALL_LOG
```

```
if ($status != 0) then
  echo "EXTRACTING THE GNU MULTIPLE PRECISION ARITHMETIC
   LIBRARY (GMP) FAILED!"
  exit -1
endif
mkdir -p gmp
cd gmp
../gmp-5.0.2/configure --enable-cxx
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE GNU MULTIPLE PRECISION ARITHMETIC
    LIBRARY (GMP) FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE GNU MULTIPLE PRECISION ARITHMETIC
   LIBRARY (GMP) FAILED!"
  exit -1
endif
make check >& $INSTALL_LOG
if ($status != 0) then
  echo "CHECK MAKE PROCESS OF THE GNU MULTIPLE PRECISION
    ARITHMETIC LIBRARY (GMP) FAILED!"
#  exit -1
#endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE GNU MULTIPLE PRECISION
   ARITHMETIC LIBRARY (GMP) FAILED!"
  exit -1
endif
#################################################################
## Install the GNU MPFR Library (MPFR)
#################################################################
echo "3.) DOWNLOADING AND INSTALLING GNU MPFR LIBRARY (MPFR)"
cd /scratch/msc11h1/mspgcc-files
wget http://www.mpfr.org/mpfr-current/mpfr-3.1.0.tar.bz2 >&
 $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE GNU MPFR LIBRARY (MPFR) FAILED!"
  exit -1
endif
tar xvfj mpfr-3.1.0.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE GNU MPFR LIBRARY (MPFR) FAILED!"
  exit -1
endif
```

```
mkdir -p mpfr
cd mpfr
../mpfr-3.1.0/configure --with-gmp=/scratch/msc11h1/mspgcc
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE GNU MPFR Library (MPFR) FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE GNU MPFR Library (MPFR) FAILED!"
  exit -1
endif
make check >& $INSTALL_LOG
if ($status != 0) then
  echo "CHECK MAKE PROCESS OF THE GNU MPFR Library (MPFR)
   FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE GNU MPFR LIBRARY (MPFR) FAILED!"
  exit -1
endif
#################################################################
## Install the Multiprecision library (MPC)
#################################################################
echo "4.) DOWNLOADING AND INSTALLING MULTIPRECISION LIBRARY (MPC)"
cd /scratch/msc11h1/mspgcc-files
wget http://www.multiprecision.org/mpc/download/mpc-0.9.tar.gz >&
$INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE MULTIPRECISION LIBRARY (MPC) FAILED!"
  exit -1
endif
tar xvf mpc-0.9.tar.gz >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE MULTIPRECISION LIBRARY (MPC) FAILED!"
  exit -1
endif
mkdir -p mpc
cd mpc
../mpc-0.9/configure --with-gmp=/scratch/msc11h1/mspgcc
--with-mpfr=/scratch/msc11h1/mspgcc
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE MULTIPRECISION LIBRARY (MPC) FAILED!"
  exit -1
```

```
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE MULTIPRECISION LIBRARY (MPC) FAILED!"
  exit -1
endif
make check >& $INSTALL_LOG
if ($status != 0) then
  echo "CHECK MAKE PROCESS OF THE MULTIPRECISION LIBRARY (MPC)
   FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE MULTIPRECISION LIBRARY (MPC) FAILED!"
  exit -1
endif
#################################################################
## Install  PPL version 0.11
#################################################################
echo "5.) DOWNLOADING AND INSTALLING PPL VERSION 0.11"
cd /scratch/msc11h1/mspgcc-files
wget ftp://ftp.cs.unipr.it/pub/ppl/releases/0.11.2/
ppl-0.11.2.tar.gz >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE PPL LIBRARY VERSION 0.11 FAILED!"
  exit -1
endif
tar xvf ppl-0.11.2.tar.gz >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE PPL LIBRARY VERSION 0.11 FAILED!"
  exit -1
endif
mkdir -p ppl
cd ppl
../ppl-0.11.2/configure --with-gmp=/scratch/msc11h1/mspgcc
--with-mpfr=/scratch/msc11h1/mspgcc
--with-mpc=/scratch/msc11h1/mspgcc
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE PPL LIBRARY VERSION 0.11 FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING the PPL LIBRARY VERSION 0.11 FAILED!"
  exit -1
endif
```

```
make check >& $INSTALL_LOG
if ($status != 0) then
  echo "CHECK MAKE PROCESS OF THE PPL LIBRARY
   VERSION 0.11 FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE PPL LIBRARY VERSION
   0.11 FAILED!"
  exit -1
endif
#################################################################
## Install CLoog version 0.15
#################################################################
echo "6.) DOWNLOADING AND INSTALLING CLOOG VERSION 0.15"
cd /scratch/msc11h1/mspgcc-files
wget ftp://gcc.gnu.org/pub/gcc/infrastructure/
cloog-ppl-0.15.11.tar.gz >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE CLOOG LIBRARY VERSION 0.15 FAILED!"
  exit -1
endif
tar xvf cloog-ppl-0.15.11.tar.gz >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE CLOOG LIBRARY VERSION 0.15 FAILED!"
  exit -1
endif
mkdir -p cloog-ppl
cd cloog-ppl
../cloog-ppl-0.15.11/configure
--with-gmp=/scratch/msc11h1/mspgcc
--with-mpfr=/scratch/msc11h1/mspgcc
--with-mpc=/scratch/msc11h1/mspgcc
--with-ppl=/scratch/msc11h1/mspgcc
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE CLOOG LIBRARY VERSION 0.15 FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE CLOOG LIBRARY VERSION 0.15 FAILED!"
  exit -1
endif
make check >& $INSTALL_LOG
if ($status != 0) then
  echo "CHECK MAKE PROCESS OF THE CLOOG LIBRARY VERSION
```

```
    0.15 FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE CLOOG LIBRARY VERSION
   0.15 FAILED!"
  exit -1
endif
#################################################################
## DOWNLOAD AND EXTRACT THE GCC TOOLCHAIN FOR MSP430
#################################################################
echo "7.) DOWNLOADING THE GCC TOOLCHAIN FOR MSP430"
cd /scratch/msc11h1/mspgcc-files
wget http://sourceforge.net/projects/mspgcc/files/mspgcc/
mspgcc-20120119.tar.bz2/download >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE GCC TOOLCHAIN FOR MSP430 FAILED!"
  exit -1
endif
tar xvf mspgcc-20120119.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE GCC TOOLCHAIN FOR MSP430 FAILED!"
  exit -1
endif
#################################################################
## DOWNLOAD AND INSTALL THE BINUTILS LIBRARY
#################################################################
echo "8.) DOWNLOADING AND INSTALLING THE BINUTILS LIBRARY"
cd /scratch/msc11h1/mspgcc-files/mspgcc-20120119
wget ftp://ftp.gnu.org/pub/gnu/binutils/binutils-2.21.1a.tar.bz2
 >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
tar xvf binutils-2.21.1a.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
cd binutils-2.21.1
patch -p1 <../msp430-binutils-2.21.1a-20120119.patch
>& $INSTALL_LOG
if ($status != 0) then
  echo "PATCHING THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
```

```
cd ..
mkdir -p BUILD/binutils
cd BUILD/binutils
../../binutils-2.21.1/configure
--target=msp430
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE BINUTILS LIBRARY FAILED!"
  exit -1
endif
#################################################################
## DOWNLOAD AND INSTALL THE GCC LIBRARY
#################################################################
echo "9.) DOWNLOADING AND INSTALLING THE GCC LIBRARY"
cd /scratch/msc11h1/mspgcc-files/mspgcc-20120119
wget ftp://ftp.gnu.org/pub/gnu/gcc/gcc-4.6.1/gcc-4.6.1.tar.bz2
>& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE GCC LIBRARY FAILED!"
  exit -1
endif
tar xvf gcc-4.6.1.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE GCC LIBRARY FAILED!"
  exit -1
endif
cd gcc-4.6.1
patch -p1 <../msp430-gcc-4.6.1-20120119.patch >&
 $INSTALL_LOG
if ($status != 0) then
  echo "PATCHING THE GCC LIBRARY FAILED!"
  exit -1
endif
cd ..
mkdir -p BUILD/gcc
cd BUILD/gcc
../../gcc-4.6.1/configure
--target=msp430
```

```
--enable-languages=c,c++
--with-gmp=/scratch/msc11h1/mspgcc
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE GCC LIBRARY FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE GCC LIBRARY FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE GCC LIBRARY FAILED!"
  exit -1
endif
#################################################################
## DOWNLOAD AND INSTALL THE GDB LIBRARY
#################################################################
echo "10.) DOWNLOADING AND INSTALLING THE GDB LIBRARY"
cd /scratch/msc11h1/mspgcc-files/mspgcc-20120119
wget ftp://ftp.gnu.org/pub/gnu/gdb/gdb-7.2a.tar.bz2 >&
 $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE GDB LIBRARY FAILED!"
  exit -1
endif
tar xjf gdb-7.2a.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE GDB LIBRARY FAILED!"
  exit -1
endif
cd gdb-7.2
patch -p1 <../msp430-gdb-7.2a-20111205.patch >&
 $INSTALL_LOG
if ($status != 0) then
  echo "PATCHING THE GDB LIBRARY FAILED!"
  exit -1
endif
cd ..
mkdir -p BUILD/gdb
cd BUILD/gdb
../../gdb-7.2/configure
--target=msp430
--prefix=/scratch/msc11h1/mspgcc >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING THE GDB LIBRARY FAILED!"
  exit -1
endif
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING THE GDB LIBRARY FAILED!"
  exit -1
endif
make install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE GDB LIBRARY FAILED!"
  exit -1
endif
#################################################################
## DOWNLOAD AND INSTALL THE MSP430MCU FILES
#################################################################
echo "11.) DOWNLOADING AND INSTALLING THE MSP430MCU FILES"
cd /scratch/msc11h1/mspgcc-files/mspgcc-20120119
wget http://sourceforge.net/projects/mspgcc/files/msp430mcu/
msp430mcu-20111224.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE MSP430MCU FILES FAILED!"
  exit -1
endif
tar xjf msp430mcu-20111224.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE MSP430MCU FILES FAILED!"
  exit -1
endif
cd msp430mcu-20111224
scripts/install.sh /scratch/msc11h1/mspgcc
echo "WARNING: MSP430MCU_ROOT must be set to the location where the
msp430mcu release lives!"
#################################################################
## DOWNLOAD AND INSTALL THE MSP430-LIBC FILES
#################################################################
echo "12.) DOWNLOADING AND INSTALLING THE MSP430-LIBC FILES"
cd /scratch/msc11h1/mspgcc-files/mspgcc-20120119
wget https://sourceforge.net/projects/mspgcc/files/msp430-libc/
msp430-libc-20120119.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "DOWNLOADING THE MSP430-LIBC FILES FAILED!"
  exit -1
endif
tar xjf msp430-libc-20120119.tar.bz2 >& $INSTALL_LOG
if ($status != 0) then
  echo "EXTRACTING THE MSP430-LIBC FILES FAILED!"
  exit -1
endif
```

```
cd msp430-libc-20120119
./configure >& $INSTALL_LOG
if ($status != 0) then
  echo "CONFIGURING OF THE MSP430-LIBC FILES FAILED!"
  exit -1
endif
cd src
make >& $INSTALL_LOG
if ($status != 0) then
  echo "MAKING OF THE MSP430-LIBC FILES FAILED!"
  exit -1
endif
make PREFIX=/scratch/msc11h1/mspgcc install >& $INSTALL_LOG
if ($status != 0) then
  echo "INSTALLATION OF THE MSP430-LIBC FILES FAILED!"
  exit -1
endif
###############################################################
## EXTEND THE EXECUTABLE SEARCH PATH
###############################################################
PATH=/scratch/msc11h1/mspgcc/bin:$PATH
export PATH
```

## D.3 Test Program for Verifying Cryptographic Modules

The results from the developed FPGA system are only reliable after a passed functional system test which verifies all cryptographic modules (hardware as also software). For that test vectors (stimulus vectors and expected response vectors) are stored in ROM. The functionality of the cryptographic modules is verified by comparing the response to a stimulus vector which the corresponding expected response vector. The program code below represents the functional system test through which the developed FPGA system was verified.

```
#include <msp430x11x1.h>
#include <signal.h>    // Needed for using interrupts with msp430-gcc

//#include "TI_aes.h"
#include "aes.h"
#include "groestl.h"
#include "cryptolib.h"

#include <signal.h>
#include <iomacros.h>

typedef unsigned char INT8U;

// Comment the following line for real device
// Modifications for real device:
```

```
//        - Delay function is longer
//        - Full testrun instead of small testvector selection
//#define SIMULATION

#ifdef SIMULATION
  // AES TESTVECTORS
  #define AES_ENCRYPTION_TESTVECTORS        0x2
  #define AES_DECRYPTION_TESTVECTORS        0x2
  // GROESTL TESTVECTORS
  #define GROESTL_TESTVECTORS               0x3C
  #define GROESTL_SINGLE_BLOCK_TESTVECTORS  0x1E
  #define GROESTL_DOUBLE_BLOCK_TESTVECTORS  0x14
  #define GROESTL_TRIPLE_BLOCK_TESTVECTORS  0x0A
#else
  // AES TESTVECTORS
  #define AES_ENCRYPTION_TESTVECTORS        0x50
  #define AES_DECRYPTION_TESTVECTORS        0x50
  // GROESTL TESTVECTORS
#define GROESTL_TESTVECTORS                 0x3C
  #define GROESTL_SINGLE_BLOCK_TESTVECTORS  0x1E
  #define GROESTL_DOUBLE_BLOCK_TESTVECTORS  0x14
  #define GROESTL_TRIPLE_BLOCK_TESTVECTORS  0x0A
#endif

// TEST VARIANTS
//------------------------------------------
#define NUM_TEST_VARIANTS           0x03
//------------------------------------------
#define SOFTWARE                    0x00
#define HARDWARE_SLOW_CLOCK         0x01
#define HARDWARE_FAST_CLOCK         0x02
// UNIT
//------------------------------------------
#define NUM_UNITS                   0x04
//------------------------------------------
#define SINGLE_AES                  0x00
#define SHARED_AES                  0x01
#define SINGLE_GROESTL              0x02
#define SHARED_GROESTL              0x03
// MODE OF OPERATION
//------------------------------------------
#define AES_NUM_OPERATION_MODES     0x02
#define GROESTL_NUM_OPERATION_MODES 0x01
//------------------------------------------
#define ENCRYPTION                  0x00
#define DECRYPTION                  0x01

void wait(void);
void signal_finished(void);
void signal_failure(void);
void check_aes_result(unsigned char *data_1, unsigned char *data_2);
void check_groestl_result(unsigned char *data_1, unsigned char *data_2);

int main(void) {

  unsigned short i;
  unsigned int act_testvector;
  unsigned int num_testvectors;
  unsigned short testvector_offset;
  unsigned int groestl_msg_length;
  // Software or Hardware
  unsigned int test_variant;
  // Single or Shared Unit
  unsigned int unit;
  // 2 for AES (Encryption/Decryption) and 1 for Groestl
  unsigned int num_operation_modes;
  // Encryption or Decryption (AES)
  unsigned int mode_of_operation;
  // AES stimuli and expected response buffers
  unsigned char aes_stimuli[32];
  unsigned char aes_expected_response[16];
  // GROESTL stimuli and expected response buffers
  unsigned char groestl_stimuli[192];
  unsigned char groestl_expected_response[28];

  WDTCTL = WDTPW | WDTHOLD;          // Disable watchdog timer
  P1DIR = 0xff;
  P1OUT = 0xff;
  P2DIR = 0xff;
  P2OUT = 0xff;

  // Switch on the led to signal start of test
  P1OUT = 0x00;
  wait();
  // Switch off the led during the test. In case of an error led starts
  // blinking. If test finished without an error then the led is
  // constantly on!!!!
  P1OUT = 0x01;

  // SW | HW with slow clock |  HW with fast clock
  for(test_variant = 0; test_variant < NUM_TEST_VARIANTS;
   test_variant++) {
    P2OUT = test_variant;  // Debug information

    // SINGLE AES | SHARED AES | SINGLE GROESTL | SHARED GROESTL
    for(unit = 0; unit < NUM_UNITS; unit++) {
      P2OUT = test_variant | (unit << 2);  // Debug information

      // For the software variant only single versions are available
      // Skip the shared version tests!
      if(test_variant == SOFTWARE &&
        (unit == SHARED_AES || unit == SHARED_GROESTL))
        continue;

      // define number of operation modes
      if(unit == SINGLE_AES || unit == SHARED_AES)
        num_operation_modes = AES_NUM_OPERATION_MODES;
      else
        num_operation_modes = GROESTL_NUM_OPERATION_MODES;

      for(mode_of_operation = 0; mode_of_operation < num_operation_modes;
          mode_of_operation++) {
        // Debug information
        P2OUT = test_variant | (unit << 2) | (mode_of_operation << 4);

        // define number of testvectors and offset
        if(unit == SINGLE_AES || unit == SHARED_AES) {
          if(mode_of_operation == ENCRYPTION)
          {
            num_testvectors = AES_ENCRYPTION_TESTVECTORS;
            testvector_offset = 0x0000;
          }
          else {
            num_testvectors = AES_DECRYPTION_TESTVECTORS;
            testvector_offset = 0x0F00; //3840
           }
         } else {
           num_testvectors = GROESTL_TESTVECTORS;
           testvector_offset = 0x0000;
         }

         // Reset the address of the test roms -
         // both aes and groestl testrom
         WRITE_PERIPHERAL_REGISTER(ADDRESS_OFFSET_L,
           (unsigned char) testvector_offset);        // Low byte offset
         WRITE_PERIPHERAL_REGISTER(ADDRESS_OFFSET_H,
           (unsigned char) (testvector_offset >> 8)); // High byte offset

         // Iterate through the testvectors
         for(act_testvector = 0; act_testvector < num_testvectors;
             act_testvector++) {
           P2OUT = test_variant | (unit << 2) | (mode_of_operation << 4) |
               (act_testvector << 5);  // Debug information
```

```
// Read in stimuli and expected response vectors either from the
// AES testrom or the GROESTL testrom
if(unit == SINGLE_AES || unit == SHARED_AES) {
  for(i = 0; i < 32; i++)
  aes_stimuli[i] = READ_PERIPHERAL_REGISTER(AES_CNTRL);
  for(i = 0; i < 16; i++)
  aes_expected_response[i] = READ_PERIPHERAL_REGISTER(AES_CNTRL);
} else {
  // define number of bytes to read in
   if(act_testvector < GROESTL_SINGLE_BLOCK_TESTVECTORS)
     groestl_msg_length = 64;
   else if(act_testvector < GROESTL_SINGLE_BLOCK_TESTVECTORS +
       GROESTL_DOUBLE_BLOCK_TESTVECTORS)
     groestl_msg_length = 128;
   else
     groestl_msg_length = 192;

   for(i = 0; i < groestl_msg_length; i++)
     groestl_stimuli[i] = READ_PERIPHERAL_REGISTER(GROESTL_CNTRL);
   for(i = 0; i < 28; i++)
     groestl_expected_response[i] =
         READ_PERIPHERAL_REGISTER(GROESTL_CNTRL);
 }

 // Compute either AES or Groestl result
 if(unit == SINGLE_AES || unit == SHARED_AES) {
   // Software
   if(test_variant == SOFTWARE
       && mode_of_operation == ENCRYPTION)
     aes_encrypt(aes_stimuli+16, aes_stimuli);
   else if(test_variant == SOFTWARE
       && mode_of_operation == DECRYPTION)
     aes_decrypt(aes_stimuli+16, aes_stimuli);
   // Hardware with slow clock
   else if(test_variant == HARDWARE_SLOW_CLOCK &&
       mode_of_operation == ENCRYPTION && unit == SINGLE_AES)
     hw_single_aes_encrypt_w_slow_clock(aes_stimuli+16,
       aes_stimuli);
   else if(test_variant == HARDWARE_SLOW_CLOCK &&
       mode_of_operation == ENCRYPTION && unit == SHARED_AES)
     hw_shared_aes_encrypt_w_slow_clock(aes_stimuli+16,
       aes_stimuli);
   else if(test_variant == HARDWARE_SLOW_CLOCK &&
       mode_of_operation == DECRYPTION && unit == SINGLE_AES)
     hw_single_aes_decrypt_w_slow_clock(aes_stimuli+16,
       aes_stimuli);
   else if(test_variant == HARDWARE_SLOW_CLOCK &&
      mode_of_operation == DECRYPTION && unit == SHARED_AES)
```

```
        hw_shared_aes_decrypt_w_slow_clock(aes_stimuli+16,
          aes_stimuli);
      // Hardware with fast clock
      else if(test_variant == HARDWARE_FAST_CLOCK &&
          mode_of_operation == ENCRYPTION && unit == SINGLE_AES)
        hw_single_aes_encrypt_w_fast_clock(aes_stimuli+16,
          aes_stimuli);
      else if(test_variant == HARDWARE_FAST_CLOCK &&
          mode_of_operation == ENCRYPTION && unit == SHARED_AES)
        hw_shared_aes_encrypt_w_fast_clock(aes_stimuli+16,
          aes_stimuli);
      else if(test_variant == HARDWARE_FAST_CLOCK &&
          mode_of_operation == DECRYPTION && unit == SINGLE_AES)
        hw_single_aes_decrypt_w_fast_clock(aes_stimuli+16,
          aes_stimuli);
      else
        hw_shared_aes_decrypt_w_fast_clock(aes_stimuli+16,
          aes_stimuli);
    } else {
      // Software
      if(test_variant == SOFTWARE)
        groestl_hash(groestl_stimuli, groestl_msg_length);
      // Hardware with slow clock
      else if(test_variant == HARDWARE_SLOW_CLOCK &&
          unit == SINGLE_GROESTL)
        hw_single_groestl_w_slow_clock(groestl_stimuli,
          groestl_msg_length);
      else if(test_variant == HARDWARE_SLOW_CLOCK &&
          unit == SHARED_GROESTL)
        hw_shared_groestl_w_slow_clock(groestl_stimuli,
          groestl_msg_length);
      // Hardware with fast clock
      else if(test_variant == HARDWARE_FAST_CLOCK &&
          unit == SINGLE_GROESTL)
        hw_single_groestl_w_fast_clock(groestl_stimuli,
          groestl_msg_length);
      else if(test_variant == HARDWARE_FAST_CLOCK &&
          unit == SHARED_GROESTL)
        hw_shared_groestl_w_fast_clock(groestl_stimuli,
          groestl_msg_length);
    }

    // Check either AES or Groestl result
    if(unit == SINGLE_AES || unit == SHARED_AES)
      check_aes_result(aes_stimuli+16, aes_expected_response);
    else
      check_groestl_result(groestl_stimuli, groestl_expected_response);
}
```

```
      }
    }
  }

  signal_finished();

  return 0;
}

void wait(void)    //delay function
{
  // declare i as volatile int (do not remove "volatile" otherwise
  // delay function will not work anymore!)
  volatile unsigned int i;

  #ifdef SIMULATION
  for(i=0;i<10;i++) {
  };
  #else
  for(i=0;i<32000;i++) {
  };
  #endif
}

void signal_finished(void)
{
  P1OUT = 0x00;    // switch on led
  while(1) {}
}

void signal_failure(void)
{
  while(1)
  {
    wait();
    //toggle least significant bit - connected to on-board led
    P1OUT ^= 0x01;
  }
}

void check_aes_result(unsigned char *data_1, unsigned char *data_2)
{
  unsigned char i;

  for(i=0;i<16;i++)
  {
    if(data_1[i] != data_2[i])
      signal_failure();
  }
}

void check_groestl_result(unsigned char *data_1, unsigned char *data_2)
{
  unsigned char i;

  for(i=0;i<28;i++)
  {
    if(data_1[i] != data_2[i])
      signal_failure();
  }
}
```

Appendix E

# Original Assignment

The original assignment is appended in the next six pages. It encompasses a short introduction into the topic followed by the general project description which can be mainly split into two tasks. The first targets a low-area ASIC/FPGA implementation of a combined hardware architecture for AES and Grøstl. Second, an FPGA system has to be created containing the openMSP430 and in addition the cryptographic modules designed previously in order to establish a fair hardware/software comparison. After the project description the goals of the work are defined, which can be summarized to a low area and low power ASIC/FPGA implementation of a combined hardware architecture and a functional FPGA system containing the openMSP430 and the previously developed cryptographic modules for AES, Grøstl and GrÆStl. Next, milestones are set which have to be reached in a sequential manner. First, the separate implementations of AES and Grøstl should be implemented before designing the combined hardware architecture, named GrÆStl. Afterwards, a functional chip containing these three cryptographic modules and an interface to access them should be created. A full back-end design is required in order to tape-out the chip in an appropriate manner. After the tape-out the focus is put on FPGA platforms where a functional system is targeted containing a microcontroller and the previously developed ASIC modules which have to be ported on the FPGA platform before. Last step is to achieve results for a hardware/software comparison. Therefore, besides the hardware units also software routines for the cryptographic algorithms should be written. After defining the milestones to reach the project goals, the project realization is discussed encompassing a project plan, weekly meetings, the report to deliver, the design review for the ASIC to be taped-out and the presentation required to conclude the work. Finally, a listing is given presenting the deliverables in order to finish the work successfully.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Master Thesis at the Departement of
Information Technology and Electrical Engineering

Winter Term 2011

Markus Pelnar

# In the Footsteps of a Combined and Miniaturized AES/Grøstl Design

October 25, 2011

Advisors: Michael Muehlberghuber (IIS), ETZ J71.2, Tel. +41 44 632 57 45, `mbgh@iis.ee.ethz.ch`
Beat Muheim (DZ), ETZ J60.1, Tel. +41 44 632 66 75, `muheim@ee.ethz.ch`
Michael Hutter (IAIK), TU-Graz, Tel. +43 316 873 5541 `Michael.Hutter@iaik.tugraz.at`

Handout: September 19, 2011
Due: March 19, 2012

## 1 Introduction

Radio Frequency IDentification (RFID) systems have become an important part of everyday life. An RFID system consists of two parts: a reader and a tag. The reader can uniquely identify different tags by exchanging information through an RF communication protocol. Cryptographic primitives are used throughout this exchange to provide different services, such as authentication.

For most applications, the reader is not really resource constrained. It usually has its own power supply, can have significant computation power and a (comparatively) strong RF interface. RFID tags on the other hand, need to be cheap (small chip area) and in most cases do not have their own power source (these are called *passive RFID tags*). In such passive tags, the energy needed to process the reader communication request and provide an answer back, is harvested from the RF signal sent by the reader. Needless to say for such systems energy efficiency is paramount. Depending on the field of application, the number of required passive RFID tags being manufactured is increasing constantly (e.g. tagging of clothes, etc.). Therefore the price and thus the needed chip area is often the most constraining resource.

## 2 Project Description

This project can be subdivided into two different tasks. These tasks should, in general, be processed one after another, but can of course overlap somewhat.

**AES/Grøstl** Two of the most important cryptographic primitives which are required on an RFID tag are a block cipher and a hash function. Hence, the first task is represented by the design of an ASIC (Application-Specific Integrated Circuit), which enables to peform the calculation of a block cipher (AES - Advanced Encryption Standard) and a hash function (Grøstl). The design of this ASIC should be highly optimized towards low area in order to reduce manufacturing costs.

**MSP430** The second task is made up of the implementation of the MSP430 microcontroller and the previously designed AES/Grøstl block on a Spartan-3 FPGA (Field Programmable Gate Array) board. The MSP430 must be able to communicate with the AES/Grøstl block in order to provide cryptographic services which make use of the block cipher and the hash function.

### 2.1 AES/Grøstl ASIC

AES is the *de facto standard* among block ciphers for more than a decade right now and because it has already been properly analyzed towards it security, some cryptographic primitives appear which make use of the AES core functionalities. One of them is the Grøstl hash algorithm, which has been submitted to the SHA-3 competition [5] and has now become one of the five finalists.

At first two distinct designs of AES and Grøstl have to be implemented. Because the two cryptographic primitives partly make use of the same core functionalities, a combined design, highly optimized towards low area, is targeted afterwards. In order to demonstrate the area reduction, a comparison has to be made between the two distinct designs and the combined version. Due to the fact that both cryptographic primitives should share the same hardware circuits, it is not required to execute them simultaneously.

### 2.2 MSP430 Implementation

In order to be able to implement cryphtographic protocols which make use of a block cipher and a hash function, a TI-MSP430 compatible microprocessor will be implemented on a Spartan-3 FPGA board. The MSP430 HDL description is freely available on the internet and will be taken from the *opencores.org* webpage [6]. Furthermore the previously designed AES/Grøstl block has to be ported onto the Spartan-3 FPGA and an appropriate communication with the microprocessor has to be established. The MSP430 will require some program and data memory. The exact size of these memories will be determined during the project.

## 3 Goals

The main goal of the project is to implement a resource sharing AES/Grøstl design where area represents the major constraining resource. Furthermore this design, as well as the MSP430 microprocessor from opencores.org have to be ported to a Spartan-3 FPGA board.

What sets this work apart from others is the low-area implementation of the combined AES/Grøstl design. Currently no ASIC implementation of AES and Grøstl is known which makes use of the same core functionalities in order to reduce the required chip area.

### 3.1 Low Area

The selling point for RFID tags is that they are inexpensive. This directly translates to small circuit area. The smaller the circuit the better. Hence, the major goal is to decrease the required chip area for the AES/Grøstl design using any means possible.

### 3.2 Low Power

A passive RFID tag is extremely short on power as it has no independent power supply of its own, but relies on the energy collected from the RF transmission. Both the peak power and the total energy have strict limits. Hence, despite the fact that area is the most constraining resource, the system must be designed such that the power consumption stays within a reasonable amount.

### 3.3 FPGA System Design

In order to provide a system which allows the implementation of different cryptographic protocols, the MSP430 compatible microprocessor from opencores.org will be ported to a Spartan-3 FPGA board. The ASIC design of the AES/Grøstl block will be ported to the FPGA as well, which allows the implementation of hardware accelerated protocols based on a block cipher and/or a hash function.

## 4 Milestones

The following is a list of expected milestones in the project.

- **AES/Grøstl - Separated Implementations**
  Before a combined version of the AES/Grøstl block is designed, both cryptographic prim-

itives have to be implemented separately. The combined block can then be compared with these designs in order to present the area reduction which is due to the shared resources.

- **AES/Grøstl - Combined Implementation**
  The combined block of the AES/Grøstl design represents the second milestone and should be implemented such that as many resources can be shared among the block cipher and the hash algorithm. The major target hereby is a low-area implementation of the two cryptographic primitives.

- **AES/Grøstl - ASIC**
  The final result of the AES/Grøstl specific work will be one ASIC sent to manufacturing using a suitable technology (UMC180/UMC130 or others).

- **FPGA System Design**
  The MSP430 needs to be implemented on the Spartan-3 FPGA board with all the options (timers/interrupts/IO ports) that are deemed necessary for the project. This also includes the necessary RAM and ROM blocks for the processor. Furthermore the AES/Grøstl design has to be ported onto the FPGA board in order to realize hardware accelerated cryptographic protocols which make use of a block cipher and a hash algorithm.

- **Hardware/Software Comparison**
  In order to demonstrate the correctly working implementations of the MSP430 microprocessor and the AES/Grøstl block on the FPGA board using a *real world example*, AES and Grøstl have to be implemented on the Microprocessor at first. Afterwards the hardware accelerated version of the two cryptographic primitives should be used to demonstrate the improved performance.

# 5 Project Realization

## 5.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and set deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between the students and the advisors. Note that the project plan should be updated constantly depending on the project's status.

## 5.2 Meetings

Weekly meetings will be held between the student and the assistants every Tuesday at 09:00. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

## 5.3 Reports

Documentation is an important and often overlooked aspect of engineering. One short intermediate report and one final report (the Master Thesis) are to be completed within this study. Note that the intermediate report should be designed to be part of the final report.

The common language of engineering is de facto English. Therefore, the intermediate and final report of the work are preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif (for block diagrams) is strongly encouraged by the IIS staff.

**First Intermediate Report** This report should be written in such a way to become the first part of your final report. It should contain general information about the topic, a description of the problem, explanations of related terminology, and descriptions of similar approaches in literature (with corresponding references to books, papers etc.). So it should mainly contain the *theoretical part* of your Master Thesis.

**Final Report** The final report has to be presented at the end of the Master Thesis and a digital copy needs to be handed out and remains property of the IIS. This report is only accepted if the keys for the ETZ building as well as those for the student working room have been properly returned. Note that this task description is part of your thesis and has to be attached to your final report.

### 5.4 Design Review

Because the AES/Grøstl design is supposed to be manufactured using an appropriate semiconductor technology, a review of the chip design will be held during late November. The exact date of the review will be determined a few weeks in advance.

### 5.5 Presentation

There will be a presentation (20 min presentation and 5 min Q&A) at the end of this project to present your results to a wider audience. The exact date will be determined towards the end of the work.

## 6 Deliverables

Throughout the project, the following deliverables have to be submitted in order to finish the work successfully:

- Project plan
- Separated AES/Grøstl designs
- Combined AES/Grøstl design
- AES/Grøstl ASIC layout
- MSP430 FPGA implementation
- AES/Grøstl FPGA implementation
- Intermediate report
- Final report

## References

[1] H. Kaeslin, "Digital Integrated Circuit Design", Cambridge University Press, 2008

[2] Design Zentrum website: `http://www.dz.ee.ethz.ch` and VHDL naming conventions: `http://www.dz.ee.ethz.ch/en/information/hdl-help/vhdl-naming-conventions.html`

[3] FIPS PUB 197: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[4] Grøstl website: `http://www.groestl.info`

[5] SHA-3 Competition: `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`

[6] openMSP430 website: `http://opencores.org/project,openmsp430`

Zurich, December 19, 2011 Prof. Dr. Hubert Kaeslin

**The thesis will not be accepted without returning the keys!**

# Bibliography

[1] Anderson, R. and Biham, E. and Knudsen, L. Serpent: A Proposal for the Advanced Encryption Standard. In *Proceedings of the First AES Candidate Conference*, Ventura, CA, USA, jun 1998. National Institute of Standard and Technology.

[2] Aumasson, J. P. and Henzen, L. and Meier, W. and Phan, R. C. W. SHA-3 proposal BLAKE. Submission to NIST (Round 3), 2010.

[3] Bulens, P. and Standaert, F. X. and Quisquater, J. J. and Pellegrin, P. and Rouvroy, G. Implementation of the AES-128 on Virtex-5 FPGAs. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 16–26. Springer, 2008.

[4] Burwick, C. and Coppersmith, D. and D'Avignon, E. and Gennaro, R. and Halevi, S. and Jutla, C. and Matyas Jr, S. M. and O'Connor, L. and Peyravian, M. and Luke, Jr. and Peyravian, O. M. and Stafford, D. and Zunic, N. MARS - a candidate cipher for AES. *NIST AES Proposal*, 1999.

[5] Canright, D. A Very Compact S-Box for AES. In Rao, Josyula R. and Sunar, Berk, editor, *Cryptographic Hardware and Embedded Systems CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, chapter 32, pages 441–455. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[6] Cao, D. and Han, J. and Zeng, X. Y. A Reconfigurable and Ultra Low-Cost VLSI Implementation of SHA-1 and MD5 Functions. In *International Conference on ASIC Proceeding – ICASIC 2007, 7th International Conference, Guilin, China, October 25-29, 2007*, pages 862–865. IEEE, October 2007.

[7] Chapman, K. Saving Costs with the SRL16E. http://www.xilinx.com/support/documentation/white_papers/wp271.pdf, 05 2008.

[8] Chodowiec, P. and Gaj, K. Very Compact FPGA Implementation of the AES Algorithm. In Colin D. Walter and Çetin Kaya Koç and Christof Paar, editor, *Cryptographic Hardware and Embedded Systems – CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[9] Daemen, J and, Rijmen, V. *AES Proposal: Rijndael.* http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf, 2nd edition, September 1999.

[10] Daemen, J. and Rijmen, V. *The Design of Rijndael.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[11] Feldhofer, M. and Wolkerstorfer, J. and Rijmen, V. AES implementation on a grain of sand. *IEE Proceedings - Information Security*, 152(1):13–20, October 2005. .

[12] Ferguson, N. and Lucks, S. and Schneier, B. and Whiting, D. and Bellare, M. and Kohno, T. and Callas, J. and Walker, J. The Skein Hash Function Family. Submission to NIST (Round 3), 2010.

[13] FPGA. http://www.mikrocontroller.net/articles/FPGA, 05 2012. [Online; accessed 28-May-2012].

[14] Ganesh, T. S. and Sudarshan, T. S. B. ASIC Implementation of a Unified Hardware Architecture for Non-Key Based Cryptographic Hash Primitives. In *International Conference on Information Technology: Coding and Computing (ITCC 2005), April 4-6, 2005, Las Vegas, Nevada, USA, Proceedings*, volume 1, pages 580–585. IEEE Computer Society, April 2005. ISBN 0-7695-2315-3.

[15] Gauravaram, P. and Knudsen, L. R. and Matusiewicz, K. and Mendel, F. and Rechberger, C. and Schläffer, M. and Thomsen, S. S. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011.

[16] Girard, O. openmsp430. http://opencores.org/project,openmsp430, March 2012.

[17] Good, T. and Benaissa, M. AES on FPGA from the Fastest to the Smallest. In Rao, Josyula and Sunar, Berk, editor, *Cryptographic Hardware and Embedded Systems CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer Berlin / Heidelberg, 2005. 10.1007/11545262_31.

[18] Guo, X. and Huang, S. and Nazhandali, L. and Schaumont, P. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. In *Second SHA-3 Candidate Conference, 2010*, 2010.

[19] Hämäläinen, P. and Alho, T. and Hännikäinen, M. and Hämäläinen, T.D. Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 577 –583, 0-0 2006.

[20] Hankerson, D. and Menezes, A. J. and Vanstone, S. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[21] Henzen, L. and Gendotti, P. and Guillet, P. and Pargaetzi, E. and Martin Zoller and Frank K. Gürkaynak. Developing a Hardware Evaluation Method for SHA-3 Candidates. In *Cryptographic Hardware and Embedded Systems – CHES 2010 12th International Workshop, Santa Barbara, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Note in Computer Science*, pages 248–263, Santa Barbara, CA, 2010. Springer-Verlag.

[22] Huang, C. W. and Chang, C. J. and Lin, M. Y. and Tai, H. Y. Compact FPGA implementation of 32-bits AES algorithm using Block RAM. In *TENCON 2007 - 2007 IEEE Region 10 Conference*, pages 1 –4, 30 2007-nov. 2 2007.

[23] Järvinen, K. Sharing Resources Between AES and the SHA-3 Second Round Candidates Fugue and Groestl. In *Second SHA-3 Candidate Conference*, August 2010.

[24] Järvinen, K. U. and Tommiska, M. and Skyttä, J. A Compact MD5 and SHA-1 Co-Implementation Utilizing Algorithm Similarities. In *Engineering of Reconfigurable Systems and Algorithms – ERSA 2005, International Conference, Las Vegas, Nevada, USA, June 27-30, 2005*, pages 48–54. CSREA Press, 2005.

[25] Jungk, B. Evaluation Of Compact FPGA Implementations For All SHA-3 Finalists.

[26] Jungk, B. and Apfelbeck, J. Area-Efficient FPGA Implementations of the SHA-3 Finalists. In *ReConFig*, pages 235–241, 2011.

[27] Jungk, B. and Reith, S. On FPGA-Based Implementations of the SHA-3 Candidate Grøstl. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 316 –321, December 2010.

[28] Kaeslin, H. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication.* Cambridge University Press, 1 edition, apr 2008.

[29] Kaps, J. P. and Sunar, B. Energy comparison of AES and SHA-1 for ubiquitous computing. In Xiaobo Zhou and Oleg Sokolsky and LuYan and Eun-Sun Jung and Zili Shao and Yi Mu and Dong-Chun Lee and Daeyoung Kim Young-Sik Jeong and Cheng-Zhong Xu, editor, *2nd IFIP International Symposium on Network Centric Ubiquitous Systems (NCUS 2006), Seoul, Korea, August 1-4, 2006, Proceedings*, volume 4097 of *Lecture Notes in Computer Science*, pages 372–381. Springer, 2006.

[30] Kaps, J. P. and Yalla, P. and Surapathi, K. K. and Habib, B. and Vadlamudi, S. and Gurung, S. Lightweight Implementations of SHA-3 Finalists on FPGAs.

[31] Katashita, T. Groestl Compact. http://www.rcis.aist.go.jp/special/SASEBO/, Februar 2010.

[32] Kavun, E. B. and Yalcin, T. On the Suitability of SHA-3 Finalists for Lightweight Applications.

[33] Kerckhof, S. and Durvaux, F. and Veyrat-Charvillon, N. and Regazzoni, F. and de Dormale, G. M. and Standaert, F. X. Compact FPGA Implementations of the Five SHA-3 Finalists. In Prouff, Emmanuel, editor, *CARDIS*, volume 7079 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2011.

[34] Kim, M. and Ryou, J. and Choi, Y. and Jun, S. Low Power AES Hardware Architecture for Radio Frequency Identification . In Hiroshi Yoshiura and Kouichi Sakurai and Kai Rannenberg and Yuko Murayama and Shinichi Kawamura, editor, *First International Workshop on Security (IWSEC 2006), Kyoto, Japan, October 23-24, 2006, Proceedings*, volume 4266 of *Lecture Notes in Computer Science*, pages 353–363. Springer, October 2006.

[35] Knudsen, L. R. and Robshaw, M. *The Block Cipher Companion.* Information security and cryptography. Springer, 2011.

[36] Küsters, R. and Wilke, T. *Moderne Kryptographie - Eine Einführung.* Vieweg + Teubner, 2011.

[37] Latif, K. and Rao, M. M. and Aziz, A. and Mahboob, A. Efficient Hardware Implementations and Hardware Performance Evaluation of SHA-3 Finalists.

[38] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 3, pages 113–114. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[39] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 7. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[40] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 6. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[41] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 9, page 353. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[42] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 9. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[43] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 9, page 355. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[44] Menezes, A. J. and Vanstone, S. A. and Oorschot, P. C. V. *Handbook of Applied Cryptography*, chapter 9, page 348. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[45] Moradi, A. and Poschmann, A. and Ling, S. and Paar, C. and Wang, H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Paterson, Kenneth, editor, *Advances in Cryptology EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-20465-4_6.

[46] MSPGCC4. http://sourceforge.net/projects/mspgcc4/files/mspgcc4/mspgcc4-20110312.zip/download, 03 2011.

[47] National Institute of Standards and Technology. *FIPS PUB 46-3: Data Encryption Standard (DES)*. pub-NIST, pub-NIST:adr, oct 1999. supersedes FIPS 46-2.

[48] National Institute of Standards and Technology. FIPS 180-3, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-3. Technical report, Department of Commerce, aug 2008.

[49] Nikova, S. I. and Rijmen, V. and Schläffer, M. Using normal bases for compact hardware implementations of the AES S-box. In R. Ostrovsky and R. De Prisco and I. Visconti, editor, *6th International Conference Security in Communication Networks (SCN)*, Lecture Notes in Computer Science, pages 236–245. Springer Verlag, 2008. Work done before joining UTwente.

[50] NIST. SHA-3 Cryptographic Hash Algorithm Competition. Webpage (Last accessed on 2012-03-01), . http://csrc.nist.gov/groups/ST/hash/sha-3/.

[51] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.

[52] NIST. Overview of the AES Development Effort. http://csrc.nist.gov/archive/aes/index.html, February 2001.

[53] NIST. CRYPTOGRAPHIC HASH ALGORITHM COMPETITION. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html, December 2010.

[54] Rivest, R. L. The MD5 Message-Digest Algorithm (RFC 1321). `http://www.ietf.org/rfc/rfc1321.txt?number=1321`.

[55] Rivest, R. L. . The RC5 Encryption Algorithm. In , pages 86–96. Springer-Verlag, 1995.

[56] Rivest, R. L. The MD4 Message Digest Algorithm. In Alfred Menezes and Scott A. Vanstone, editor, *CRYPTO*, volume 537 of *LNCS*, pages 303–311. Springer, 1990.

[57] Rivest, R. L. and Robshaw, M. J. B. and Sidney, R. and Yin, Y. L. The RC6 Block Cipher.

[58] Schneier, B. The Blowfish Encryption Algorithm — One Year Later. *Dr Dobbs*, 20:137, 1995.

[59] Schneier, B. and Kelsey, J. and Whiting, D. and Wagner, D. and Hall, C. and Ferguson, N. Twofish: A 128-Bit Block Cipher. In *In First Advanced Encryption Standard (AES) Conference*, 1998.

[60] Sharif, M. U. and Shahid, R. and Rogawski, M. and Gaj, K. Use of Embedded FPGA Resources in Implementations of Five Round Three SHA-3 Candidates. In *CRYPT II Hash Workshop 2011*, 2011.

[61] Texas Instruments. MSP430x1xx Family. http://www.ti.com/lit/ug/slau049f/slau049f.pdf, 2006.

[62] Tillich, S. and Feldhofer, M. and Issovits, W. and Kern, T. and Kureck, H. and Mühlberghuber, M. and Neubauer, G. and Reiter, A. and Köfler, A. and Mayrhofer, M. Compact Hardware Implemenations of the SHA-3 Candidates ARIRANG, BLAKE, Grøstl and Skein. In Mario Auer and Wolfgang Pribyl and Peter Söser, editor, *Proceedings of Austrochip 2009, October 7, 2009, Graz, Austria*, pages 69 – 74, 2009.

[63] Wang, M. Y. and Su, C. P. and Huang, C. T. and Wu, C. W. An HMAC processor with integrated SHA-1 and MD5 algorithms. In Masaharu Imai, editor, *Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004 (ASP-DAC), Yokohama, Japan, January 27-30, 2004, Proceedings*, pages 456–458. IEEE, January 2004.

[64] Wolkerstorfer, J. and Oswald, E. and Lamberger, M. An ASIC Implementation of the AES SBoxes. In Preneel, Bart, editor, *Topics in Cryptology CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 29–52. Springer Berlin / Heidelberg, 2002.

[65] Xilinx. Using Block RAM in Spartan-3 Generation FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf, 03 2005.

[66] Xilinx. Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp464.pdf, 03 2005.

[67] Xilinx. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs. http://www.xilinx.com/support/documentation/application_notes/xapp465.pdf, 05 2005.

[68] Xilinx. Spartan-3 Generation FPGA User Guide. Webpage (Last accessed on 2012-04-04), June 2011. http://www.xilinx.com/support/documentation/user_guides/ug331.pdf.