EFFICIENT HARDWARE ARCHITECTURES FOR CRYPTOGRAPHIC ALGORITHMS USED IN COMPUTER AND COMMUNICATION SYSTEMS

İsmail SAN

PhD Dissertation

Graduate School of Sciences
Electrical and Electronics Engineering Program

March, 2014

ANADOLU ÜNİVERSİTESİ

JÜRİ VE ENSTİTÜ ONAYI

İsmail SAN'ın "**Efficient Hardware Architectures for Cryptographic Algorithms used in Computer and Communication Systems**" başlıklı **Elektrik-Elektronik Mühendisliği** Anabilim Dalındaki, Doktora Tezi 28.03.2014 tarihinde, aşağıdaki jüri tarafından Anadolu Üniversitesi Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin ilgili maddeleri uyarınca değerlendirilerek kabul edilmiştir.

	Adı - Soyadı	**İmza**
Üye (Tez Danışmanı) :	**Yard. Doç. Dr. Nuray AT**	.....
Üye :	**Doç. Dr. Hüseyin POLAT**	.....
Üye :	**Doç. Dr. Cüneyt AKINLAR**	.....
Üye :	**Prof. Dr. Çetin Kaya KOÇ**	.....
Üye :	**Doç Dr. Rıfat EDİZKAN**	.....

Anadolu Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ..... tarih ve ..... sayılı kararıyla onaylanmıştır.

Enstitü Müdürü

ABSTRACT

PhD Dissertation

EFFICIENT HARDWARE ARCHITECTURES FOR CRYPTOGRAPHIC ALGORITHMS USED IN COMPUTER AND COMMUNICATION SYSTEMS

İsmail SAN

Anadolu University
Graduate School of Sciences
Electrical and Electronics Engineering Program

Supervisor: Assist. Prof. Dr. Nuray AT
2014, 178 pages

Modern cryptography is an important discipline, which involves the disciplines of mathematics, electrical engineering and computer science, that provides the primitives of security services to information systems. Today, more important is how the fundamentals of cryptography are implemented and used. They involve significantly a series of computationally complex mathematical transformations and suffer from computational efficiency problems. Improving the security levels in computer and communication systems involves huge multiplications or performing the algorithm with longer key that might be a hundred digits long or longer, which make the computational complexity worse. In this dissertation, diverse efficient hardware architectures are proposed for a set of cryptographic algorithms, which are necessary to provide security functions to those systems, in order to overcome the computational efficiency challenges. Through the proposed design techniques, which accommodate resource sharing, efficient memory access scheme and tight scheduling, novel compact coprocessors are developed for many kinds of ciphers in symmetric and asymmetric cryptography. The proposed coprocessors focus on finding better trade-off between cost and performance and overcome the challenges introduced by the growing system needs.

The proposed architectures are analyzed and compared based on hardware performance metrics consisting of area, frequency and throughput measures. The experimental results and comparisons show that the methods require very low-area with adequate throughput values for many applications. They also reveal a deeper understanding of the computational efficiency of studied cryptographic algorithms. Designing efficient hardware architectures for those algorithms solves the efficiency issues and reveals the parallelism behind the algorithm.

Keywords: Cryptography; Coprocessor; Computational efficiency; Secure communication systems; Hardware architecture; FPGA.

ÖZET

Doktora Tezi

HABERLEŞME VE BİLGİSAYAR SİSTEMLERİNDE KULLANILAN KRİPTOGRAFİK ALGORİTMALAR İÇİN VERİMLİ DONANIM MİMARİLERİ

İsmail SAN

Anadolu Üniversitesi
Fen Bilimleri Enstitüsü
Elektrik-Elektronik Mühendisliği Anabilim Dalı

Danışman: Yard. Doç. Dr. Nuray AT
2014, 178 sayfa

Modern kriptografi matematik, elektrik mühendisliği ve bilgisayar bilimini içinde barındıran–bilgi sistemleri için güvenlik hizmetlerinin basit ögelerini sunan önemli bir disiplindir. Günümüzde, daha önemli olan kriptografi temellerinin nasıl uygulandığı ve kullanıldığıdır. Bu temel birimler hesaplama açısından karmaşık olan bir dizi matematiksel dönüşümü önemli ölçüde içinde barındırır ve hesaplama verimliliği sorunlarıyla karşı karşıyadır. Bilgisayar ve iletişim sistemlerinde güvenlik düzeylerini artırmak muazzam seviyede çarpma işlemi ya da hesaplama karmaşıklığını daha kötü hale getiren yüz basamak büyüklüğünde ya da daha büyük boyutta olan anahtarlar kullanan algoritmaları gerçekleştirmeyi gerektirir. Bu tezde, hesaplama verimliliği zorluklarının üstesinden gelmek amacıyla, bilişim sistemlerine güvenlik fonksiyonu kazandırmada gerekli olan kriptografik algoritmalar için çeşitli verimli donanım mimarileri önerilmiştir. Kaynak paylaşımı, verimli bellek erişim düzeni ve sıkı zamanlama gibi kavramları içinde barındıran önerilen tasarım teknikleri sayesinde, simetrik ve asimetrik kriptografiyide içeren birçok türde kriptografi algoritma için özgün verimli yardımcı işlemciler geliştirilmiştir. Önerilen yardımcı işlemciler artan sistem gereksinimleriyle ortaya çıkan zorluklarla başa çıkmaya ve maliyetler ve hesaplama performansı arasında daha iyi denge bulmaya odaklanır.

Önerilen mimariler analiz edilmiş ve alan, frekans ve çıktı ölçütlerinden oluşan donanım performans ölçütleri esas alınarak kıyaslanmıştır. Yapılan karşılaştırma ve deney sonuçları uygulanan metodun bir çok uygulama için yeterli bir çıktı performansı ile birlikte çok düşük alan gereksinim özelliklerini göstermektedir. Sonuçlar aynı zamanda çalışılan kriptografik algoritmaların hesaplama verimliliği hakkında daha derin bir anlayış ortaya koymaktadır. Bu algoritmalar için verimli donanım mimarileri tasarımı verimlilik sorunları çözer ve algoritma içindeki paralelliği ortaya çıkarır.

Anahtar Kelimeler: Kriptografi; Yardımcı işlemci; Hesaplama verimliliği; Güvenli haberleşme sistemleri; Donanım mimarileri; FPGA.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere thanks to my supervisor, Assist. Prof. Dr. Nuray At, who has guided me throughout my dissertation with her patience and unsurpassed knowledge. Her support and encouragement with good advice and friendship has been invaluable on both an academic and a personal level. Also, I would like to thank Assoc. Prof. Dr. Hüseyin Polat for his support and unique contribution in Chapter 5 of the dissertation. I would like to thank Assoc. Prof. Dr. Cüneyt Akınlar for his insightful comments on the research topics that I am interested in. I also would like to thank Prof. Dr. Çetin Kaya Koç, and Assoc. Prof. Dr. Rıfat Edizkan for their valuable comments and contributions during my dissertation defense.

I would like to thank Dr. Jean-Luc Beuchat for his invaluable comments and discussion on the research topics studied in this dissertation. I would like to acknowledge the collaboration with him for providing some results in Chapter 4 of the dissertation. I would like to thank him for his support, kind interests and friendship. I also would like to thank Dr. Eiji Okamoto and Teppei Yamazaki for the valuable collaboration.

I would like to thank Assist. Prof. Dr. İbrahim Yakut for his scientific support especially in Chapter 5. I also would like to thank him for his friendship and valuable comments and the discussions during my PhD.

I gratefully acknowledge Xilinx and the Xilinx University Program for its generous donation of materials in terms of design tools.

Finally, I take this opportunity to thank my parents, Ahmet San and Gülten San, who always support me with their love. I would like to thank my brothers, Ferhat San and Remzi San, for their endless support and friendship. I am grateful to my beloved wife, Gülşah San. I would like to express my sincerest gratitude to my wife. Her everlasting love and support enabled me to get this far. This dissertation would not have been possible without them.

İsmail San
March, 2014

TABLE OF CONTENTS

ABSTRACT ..... **i**

ÖZET ..... **ii**

ACKNOWLEDGMENTS ..... **iii**

TABLE OF CONTENTS ..... **iv**

LIST OF FIGURES ..... **ix**

LIST OF TABLES ..... **xii**

ABBREVIATIONS ..... **xiv**

1 INTRODUCTION **1**

1.1 Overview and Motivation 1

1.2 Research Problems and Contributions 3

1.3 Aim of the Dissertation 6

1.4 Literature Review 7

1.4.1 Lightweight Cryptography for Block Ciphers 7

1.4.2 Compact Hardware Architectures for Cryptographic Algorithms 9

1.4.3 High-Performance Designs for Public Key Cryptography 10

2 BACKGROUND **13**

2.1 Cryptographic Preliminaries 13

2.1.1 Block Ciphers 13

2.1.1.1 The Hummingbird encryption 13

2.1.1.2 The XTEA cryptographic algorithm 13

2.1.1.3 The Advanced Encryption Standard 15

2.1.1.4 The Threefish block cipher 17

2.1.2 Cryptographic Hash Functions 17

2.1.2.1 The hash function Grøstl 17

2.1.2.2 The hash function Keccak 18

2.1.2.3 The Skein family of hash functions 20

2.1.3 Public Key Cryptography 21

2.1.3.1 The Paillier homomorphic cryptosystem . . . 21
2.1.3.2 Modular exponentiation and multiplication . 23
2.1.3.3 High-radix Montgomery Multiplication Algorithm . . . 24
2.1.3.4 Karatsuba algorithm . . . 26
2.1.3.5 Montgomery Exponentiation Algorithm . . . 27
2.2 FPGA Specifics . . . 28
2.2.1 Xilinx Spartan-3 FPGA . . . 28
2.2.2 Xilinx Virtex-5 and-6 FPGA . . . 29
2.2.3 System-on-chip design . . . 30
2.2.4 MicroBlaze based embedded system . . . 31
2.2.4.1 Communication methods in MicroBlaze . . . 31
2.2.4.2 FSL based communication in MicroBlaze . . 31
2.2.4.3 PLB based communication in MicroBlaze . . 32
2.2.5 Zynq-based embedded system . . . 32
2.2.5.1 AXI based communication in Zynq . . . 33
2.3 Performance Evaluation . . . 35
2.3.1 Circuit size . . . 35
2.3.2 Latency . . . 35
2.3.3 Throughput . . . 36

3 LIGHTWEIGHT HARDWARE ARCHITECTURES FOR BLOCK CIPHERS 37
3.1 Introduction and Motivation . . . 37
3.2 Related Work . . . 38
3.3 Lightweight Hardware Architectures for Hummingbird . . . 40
3.3.1 Hummingbird coprocessor . . . 41
3.3.1.1 Overall architecture . . . 43
3.3.1.2 Register file, instruction memory and control unit . . . 43
3.3.1.3 Datapath of the coprocessor . . . 44
3.3.1.4 Scheduling of instructions . . . 45
3.3.2 Results . . . 47
3.3.3 Conclusions . . . 47
3.4 Lightweight Hardware Architectures for XTEA . . . 48
3.4.1 The lightweight XTEA . . . 49
3.4.1.1 Architecture . . . 51
3.4.1.2 Datapath . . . 51

3.4.1.3 Illustration of serialization 52
3.4.1.4 Controller 52
3.4.1.5 User interface 53
3.4.2 Hardware implementation 53
3.4.2.1 FPGA implementation 53
3.4.2.2 ASIC implementation 55
3.4.3 Conclusions 55

4 COMPACT HARDWARE ARCHITECTURES FOR SHA-3 CANDIDATES 57
4.1 Introduction and Motivation 57
4.2 Related Work 60
4.3 The Generic Hardware Architecture 61
4.4 Compact Hardware Architectures for KECCAK Cryptographic Hash Function 62
4.4.1 Compact KECCAK coprocessor 63
4.4.1.1 KECCAK coprocessor architecture 64
4.4.1.2 Register files, instruction memory and control unit 64
4.4.1.3 The datapath of the KECCAK coprocessor . 65
4.4.2 Instruction scheduling 68
4.4.3 Results and comparisons 69
4.4.4 Efficient SoC design for acceleration of message authentication and data Integrity on FPGAs 71
4.4.4.1 System-on-chip design 72
4.4.4.2 Communication architecture 73
4.4.4.3 Performance results 74
4.4.5 Conclusion 77
4.5 Compact Hardware Architectures for Skein Cryptographic Hash Function 78
4.5.1 The Threefish block cipher 78
4.5.2 The Skein family of hash functions 80
4.5.3 Proposed hardware architecture 82
4.5.3.1 Arithmetic and logic units for Threefish and Skein 83
4.5.3.2 Register files and control units 89
4.5.4 Results and comparisons 91
4.5.5 Conclusion 94

4.6 Compact Hardware Architectures for Grøstl Cryptographic Hash Function . . . 95
4.6.1 The Advanced Encryption Standard . . . 96
4.6.2 The hash function Grøstl . . . 97
4.6.3 A compact unified coprocessor for the AES and the Grøstl family of hash functions . . . 99
4.6.3.1 Memory organization . . . 102
4.6.3.2 Control unit . . . 104
4.6.3.3 Arithmetic and logic unit . . . 107
4.6.3.4 Optimized implementation of MixColumns and MixBytes . . . 110
4.6.4 Results and comparisons . . . 111
4.6.5 Conclusion . . . 114

5 HIGH-PERFORMANCE CRYPTOGRAPHY ON RECONFIGURABLE HARDWARE 117
5.1 Introduction and Motivation . . . 117
5.2 Related Work . . . 119
5.3 Improving the Computational Efficiency of Modular Operations for Embedded Systems . . . 125
5.3.1 Introduction . . . 125
5.3.2 Modular arithmetic coprocessor . . . 128
5.3.2.1 Design rationale . . . 128
5.3.2.2 Large-digit multiplier based on Karatsuba algorithm . . . 129
5.3.2.3 High-speed modular multiplication and exponentiation . . . 131
5.3.2.4 Arithmetic and logic unit of the coprocessor 131
5.3.2.5 Control unit . . . 132
5.3.2.6 Scheduling . . . 133
5.3.3 System-on-chip design . . . 136
5.3.4 Results and perspectives . . . 136
5.3.4.1 Methodology . . . 136
5.3.4.2 Results . . . 137
5.3.5 Conclusions . . . 141
5.4 Accelerating Privacy-Preserving Applications via Paillier Cryptoprocessor . . . 142
5.4.1 Introduction . . . 143

5.4.2 Paillier cryptoprocessor design 146
5.4.2.1 Design philosophy 146
5.4.2.2 Hardware implementation 149
5.4.2.3 Shift registers 149
5.4.2.4 Arithmetic and logic unit 149
5.4.2.5 Control unit 151
5.4.3 Empirical outcomes and discussion 152
5.4.3.1 Methodology 152
5.4.3.2 Empirical results 153
5.4.4 Case study: private matching protocol with the Paillier cryptoprocessor 155
5.4.5 Conclusions and future work 157

6 SUMMARY AND CONCLUSIONS 159

LIST OF FIGURES

2.1 Hummingbird encryption (reprinted from [12]). 14
2.2 Feistel Network Representaion of rounds of XTEA. 14
2.3 AES encryption and decryption flowcharts (reprinted from [33]) 16
2.4 The sponge construction (reprinted from [38]). 18
2.5 Simplified architecture of a Spartan-3 slice. 28
2.6 The quarter of a slice in Virtex-5 FPGA. 29
2.7 Block diagram of a Virtex-5 6-Input LUT. 30
2.8 Block diagram of an FSL with Master and Slave signal interface. 32
2.9 A generic Zynq-7000 all programmable SoC chip based embedded system and available busses. 33
2.10 Communication details. 34

3.1 Three input adder architecture in one slice. 42
3.2 Hummingbird coprocessor. 43
3.3 Datapath of the Hummingbird Coprocessor. 45
3.4 Instruction Scheduling. 46
3.5 XTEA Hardware Architecture 51
3.6 Datapath of XTEA Hardware Architecture 52
3.7 Serialization of the XTEA algorithm 53
3.8 Xilinx SRL16 shift register 54

4.1 General architecture of our coprocessors. 62
4.2 KECCAK coprocessor. 64
4.3 Instruction format of KECCAK coprocessor. 65
4.4 The Datapath of KECCAK coprocessor. 66
4.5 The column sum calculation in the θ step. 67
4.6 Instruction scheduling. 69
4.7 A general MicroBlaze based embedded system and available buses. 73
4.8 PLB connection for the KECCAK coprocessor in MicroBlaze based embedded system (KECCAK PLB SoC). 74
4.9 FSL connection for the KECCAK coprocessor in MicroBlaze based embedded system (KECCAK FSL SoC). 75
4.10 Performance results of SoC architectures without the compiler optimization. 75

4.11 Performance results of SoC architectures under compiler optimization. 76
4.12 Speedups of KECCAK FSL and KECCAK PLB with respect to KECCAK Software. 76
4.13 One of the 72 encryption rounds of Threefish-256. 80
4.14 One of the 72 decryption rounds of Threefish-256. 80
4.15 Processing a 3-block message using Skein-512-512 in normal hashing mode. 82
4.16 Architecture of the Threefish coprocessor. 83
4.17 Computation of $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ (Threefish-256). 84
4.18 Arithmetic and logic unit for Threefish encryption. 85
4.19 Computation of R3 $\leftarrow$ R1 $\boxplus$ R2 or R3 $\leftarrow$ R3 $\oplus$ R6 on a Virtex-6 device. 85
4.20 Arithmetic and logic unit for Threefish encryption and decryption. 86
4.21 Scheduling of Threefish-256 encryption. @d denotes the address of the 64-bit word d in the register file. "Rename" and "Permute" refer to lines 9 and 17 of Algorithm 6, respectively. . . . 87
4.22 Scheduling of Threefish-256 decryption. @d denotes the address of the 64-bit word d in the register file. 88
4.23 Register file of our Threefish and Skein architectures. 90
4.24 Compact implementations of several cryptographic hash functions on Virtex-6 FPGAs (512-bit digests). 93
4.25 Flowchart of the compression function f of Grøstl. 97
4.26 General architecture of our unified 8-bit coprocessor for AES and Grøstl. 102
4.27 Memory organization. 103
4.28 Address generation of P_{512} and Q_{512}. 104
4.29 Address generation of P_{1024} and Q_{1024}. 105
4.30 Latency between two consecutive rounds of P_{512} during the output transformation. 107
4.31 Implementation of MixColumns and MixBytes. 109
4.32 Multiplication by $\mathcal{I}_{\text{AES}}$, $\mathcal{M}_D$, $\mathcal{I}_{\text{Grøstl}}$, and $\mathcal{P}_{\text{Grøstl}}$. 109
4.33 Implementation of AddRoundKey and KeyExpansion. 110
4.34 Implementation of MixColumns and MixBytes on the latest Xilinx FPGAs. 111

5.1 High-speed Modular arithmetic coprocessor. 133

5.2 A view for pipelined computation of high-radix modular multiplication operation. 134
5.3 Examples of Privacy-Preserving Frameworks. 145
5.4 The architecture of Paillier encryption $\mathcal{E}$ procedure of PHC . . 150
5.5 The architecture of Paillier decryption $\mathcal{D}$ procedure of PHC . . 151
5.6 The hardware architecture of ModMult (adopted for modular multiplication operation for 2^{32} radix size) 152
5.7 Comparing the PCP with software implementation for Protocol 1157

LIST OF TABLES

2.1 Block length, key length, number of 32-bit blocks of the key (N_k), and number of rounds (N_r) of AES-128, AES-192, and AES-256. 15
2.2 Number of rounds of Threefish for different key sizes (reprinted from [35]). 17
2.3 Block length, number of column of the internal state, and number of rounds of Grøstl-n. 18

3.1 FPGA Implementation Results of the Hummingbird Encryption. 47
3.2 Performance Comparison of FPGA Implementations of Lightweight Cryptographic Algorithms. 48
3.3 Performance Comparison of FPGA Implementations of Lightweight Cryptographic Algorithms 54
3.4 Performance Comparison of FPGA Implementations of Strong Cryptographic Algorithms 55
3.5 Hand Calculated Gate Equivalents for ASIC Implementation . 56

4.1 Low-area FPGA implementation results of the KECCAK [r = 1024; c = 576]. 70
4.2 Performance comparison of FPGA implementations of SHA-3 Finalists 256. 71
4.3 Performance comparison of FPGA implementations of SHA-3 Finalists 512. 71
4.4 The configuration details of the three different SoC Architectures. 74
4.5 Permutations used by the Skein functions (reprinted from [35]). 79
4.6 Number of instructions of the algorithms of the Threefish family. 91
4.7 Compact implementations of the five SHA-3 finalists on Virtex-5 and Virtex-6 FPGAs. 92
4.8 Place-and-route results for our Threefish and Skein coprocessors on a Virtex-6 FPGA (xc6vlx75t-2). 93
4.9 Place-and-route results for hashing and AES encryption on a Virtex-6 FPGA (xc6vlx75t-2). 94
4.10 Offsets of the ShiftBytes transformation according to the row index i and the number of columns N_c. 98

4.11 Implementation of Grøstl with a single instruction. 101
4.12 Computation of the offset according to the permutation being executed. 106
4.13 Number of clock cycles required for the AES and Grøstl. 108
4.14 Place-and-route results for our unified coprocessor on Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs. The throughput of Grøstl is computed for a one-block message. 112
4.15 Place-and-route results for our Grøstl coprocessor on Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs. The throughput is computed for a one-block message. 112
4.16 Compact unified coprocessors to hash and encrypt messages. The throughput is computed for a one-block message. 113
4.17 Compact implementations of the five SHA-3 finalists on Virtex-5 and Virtex-6 FPGAs. The throughput is computed for a one-block message. 114
4.18 Unified Resource-Sharing Hardware Architectures of Cryptographic Hash Functions and Block/Stream Ciphers on FPGAs. 115

5.1 Synthesis results for the proposed modular multiplication coprocessor on Virtex-7 FPGA. 137
5.2 Synthesis results for the proposed modular exponentiation coprocessor on Virtex-7 FPGA. 138
5.3 Performance Results of Hardware Architectures for Modular Exponentiation on FPGAs. 140
5.4 The perfomance figures of various SoC architectures on Zynq-7000 based extensible processing platform. 141
5.5 Fundamental sizes of Paillier-n and its inner components 148
5.6 Number of clock cycles required for the PCP with varying n sizes 154
5.7 Performance of the PCP in terms of throughput 154
5.8 Performance of the software implementation of PHC 155

ABBREVIATIONS

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BRAM	BlockRAM
CLB	Configurable Logic Blocks
CF	Collaborative Filtering
CPU	Central Processing Unit
DES	Data Encryption Standard
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSL	Fast Simplex Link
FSM	Finite State Machine
HMAC	Keyed-Hash Message Authentication Code
IPSec	Internet Protocol Security
IV	Initialization Vector
LE	Logic Elements
LMB	Local Memory Bus
MAC	Message Authentication Codes
NIST	National Institute of Standards and Technology
PKC	Public Key Cryptography
SHA	Secure Hash Algorithm
SoC	System-on-chip
SSL	Secure Sockets Layer
UBI	Unique Block Iteration
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
VPN	Virtual Private Network

1. INTRODUCTION

1.1 Overview and Motivation

With the development in information and communication technology, microprocessor and memory based embedded systems, which are essentially employed in almost all electronic devices like smart phones, network routers and computers, need to communicate with the outside world via a broad range of communication mediums. Through the advancement of high-speed communication infrastructures like IP-based wired and wireless networks, fiber and broadband communication, several applications ask for more computational speed as well as data security at the same time. Growing needs like high-speed data connectivity and security for embedded systems in digital communication appliances increase further with the advent of computing technology.

Moore's Law states that chips, which form the basis for many electronic devices, double in speed every 18 months—has held well over more than four decades of semiconductor technology. Even if the physical limitations exist and limit this ratio of growing, it still continues to increase with this speed till no more further advancement can be achieved in semiconductor device manufacturing. Concerning the carrying data from one location to another, photonics is the fastest form of communication medium. In this connectivity, with the same trend of growth in speed of processor chips, Gilder's Law states that the amount of data transmitting out of fiber optic cables doubles at a much quicker rate—every 6 to 9 months [1]. Although these predictions have natural boundaries, the expectation of exponential growth offers high-speed processing power and high transmission rates that would increase data traffic around the world. These kinds of predictions allow us to have faster computation and communication facilities bringing some challenging design issues including higher design complexities, the need to reduce the cost and long verification and testing phases.

Embedded systems are present in many electronic devices and equipments. They are used to add a wide range of functionalities to such devices at a low cost. There are many application areas for the embedded systems including communication systems, consumer electronics, household appliances, transportation systems, medical equipments, etc. More importantly, embed-

ded systems are utilized in safety and security, medical applications and life critical applications to prevent the access from malicious attackers. Trade-off between cost and computational performance of the security systems change depending on the system requirements. Embedded systems are also widely used in telecommunication systems from telephone switches to mobile phones at the end user.

Today, variety of security attacks can easily be realized by the unauthorized user for instance via physical attacks are possible with the advent of side channel attacks. The data stored and transferred in communication systems are sensitive and/or confidential so that they should be protected from such malicious attacks and the intrusion. Hence, these systems progressively demand for higher levels of data security. Cryptography is used to protect sensitive data from unintended access. A set of cryptographic algorithms are used to provide the security, which are mainly computationally intensive algorithms. On the other hand, embedded systems have constrained resources in terms of circuit-size, memory, and throughput. To add security functionalities to such systems with cryptographic tools, area-, memory- and performance-efficient designs need to be developed.

Furthermore, pervasive computing has increasingly drawn attention in the area of information and communications technology. Today, many electronic devices in everyday life are equipped with microcontroller-based systems, having variable computing power. The number of pervasive devices is expected to increase dramatically over the next years. This will make ubiquitous computing more and more important subject in many areas of research. The security aspect of pervasive devices should also be considered since the data transferred or stored has valuable information that needs to be protected from malicious attacks. Therefore, the need for efficient hardware designs in terms of circuit-size, memory and performance in cryptography will be quite important in future.

Today, the speed of data transmission is quickly growing. There are many different problems in high speed communication. One of the most important problems is to incorporate the security needs into these communication channels without degrading the performance levels. While data transmission rates exponentially grows, in accordance with this, the complexity of the security algorithms is also increasing. This is a quite big challenge need to be solved. In this dissertation, it is considered that the problem of designing efficient hardware architectures of these security services in communication systems ranging

from low to high speed. Wide area interconnections between computers are carried out by dedicated routers and bridges to route the data on the network. Data connectivity of such embedded systems with the outside world is realized by some serial interfaces including SCI, I2C, SPI, USB, Ethernet, etc. Moreover, the advent of the Internet and new communication technologies such as the wireless Internet and hand-held communication devices create a demand for new products and the integration of current systems with the new ones. All these connectivity solutions cause information overload since a vast amount of new information is produced by the user and added to the networks via these data connectivity channels. Information overload has become bottleneck due to the rapid increase of communication networks. The enormous amount of the information that is communicated between entities in the network has security vulnerabilities since many adversaries exist in the network. Hence, the communication exist in such networks needs to be protected from malicious attacks and intrusion.

In general, information security comes into play in many applications where the data traffic contains sensitive information. Several applications including wired or wireless communication, high-speed networking and high-quality streaming demand security services if the sensitive data is involved in the communication. Cryptography is required to supply those security services to large and small communication systems that mainly involve microprocessor-based systems. For many applications, those microprocessors are specially designed to perform the tasks regarding the services of the overall system. Cryptographic algorithms, which are often computationally complex, should be investigated from a hardware design perspective and it is required to address all aspects of such high-speed and low-cost design constraints. Due to the high computational complexity of such security services, studying the architectural challenges of cryptographic algorithms from a hardware design perspective under high-speed and low-cost constraints is an important and growing field of research (see [2, 3] for instance).

1.2 Research Problems and Contributions

Security is an indispensable issue in many applications such as communication, networking, defense, and many more. In the last decade, information security has been one of the most important areas of concern with the rapid development of computers and networks. Due to the fast deployment of worldwide In-

ternet and dependence on digital information, security should be implemented and maintained to guarantee the integrity of the information. Cryptography is used to secure communication in these systems and provide accountability, fairness, accuracy, and confidentiality. Recent advances in information and communication technology demand efficient hardware architectures for cryptography and number theoretic algorithms.

To design a compact hardware architecture in terms of performance, cost and security for computationally intensive cryptographic algorithms is a challenging task. Most of the time, hardware implementations are interested in building highest performance or smallest area designs, which are also important for other information systems. High speed implementations are aimed to maximize the performance of an algorithm without considering how much hardware resources are occupied whereas the performance of an algorithm is the last order in lightweight hardware implementations. The most important design target is to minimize the required area as small as possible in lightweight implementations. Contrary to them, the compact implementations search the best possible design trade-off between cost and performance to occupy the smallest area while satisfying some performance constraints. In other words, compact implementations provide a good design trade-off between performance, cost and security for applications whose performance and area are equally important.

During our research, it is studied to provide security mechanisms to such constrained communication and information systems. It is possible to collect the research problems investigated throughout the dissertation under the three main items. The research problems studied in this dissertation can be listed, as follows:

1 *Lightweight Cryptographic Hardware Designs for Block Ciphers*: The low-cost embedded systems are rapidly becoming pervasive in our daily life. Well known applications include electronic passports, contactless payments, product tracking, access control and supply-chain management just to name a few. A considerable body of research has focused on providing cryptographic functionality to resource-constrained devices, while scarce computational and storage capabilities of low-cost smart devices make the problem challenging. This emerging research area is usually referred to as lightweight cryptography, which has to deal with the trade-off among security, cost, and performance [4]. Studying area-efficient hardware accelerators for resource-constrained devices is growing research

field. The first part presents novel lightweight hardware solutions for symmetric cryptosystems for these applications. New design strategies for the symmetric block ciphers are proposed.

2 *Compact Hardware Architectures for Cryptographic Hash Functions*: FPGA platforms present reconfigurable hardware resources whereas ASIC technology offers very cost-efficient solution for many high volume applications to assist general purpose processors in computing complex and intensive algorithms. Many hardware realizations mostly aim to accelerate computationally intensive applications while some of them aim to build compact efficient architecture where area, power and performance metrics are very important. These compact designs try to find the best trade-off between required hardware resources and performance. It is valuable to do research on designing efficient compact hardware architecture for cryptographic hash functions. This study reveals the parallelism behind the algorithms and allows one to understand the computational complexity of the algorithm. This part presents novel compact hardware solutions for cryptographic hash functions on FPGAs. New design strategies for some of the SHA-3 finalists are proposed. Mainly, the coprocessor approach is used, which takes advantage of the embedded memory blocks in FPGA to implement these cryptographic hash functions. The main advantage of this method over other existing techniques is that higher efficiency (throughput/occupied area) with respect to the previously reported FPGA implementations is achieved. This method also reveals a better understanding of the computational efficiency of those SHA-3 finalists from a hardware performance perspective in terms of resource sharing, memory access scheme, scheduling, etc.

3 *High-Performance Cryptographic Hardware Designs for Public-Key Cryptography*: Many public-key encryption schemes require high-performance computations in $\mathbb{Z}_q$ (the ring of integers with multiplication and addition modulo a positive integer q). Many of them require huge multiplications of large prime numbers like a prime number that might be 1024-bit or longer in size. For example, the Paillier and RSA schemes require multiplication and exponentiation in $\mathbb{Z}_q$. The efficient implementations of modular arithmetic operations including modular multiplication, inversion, and exponentiation have been at the center of research activities in cryptographic engineering for many years. Hence, there is an extensive

literature on methods for multiple-precision modular arithmetic [5–9]. The last part presents novel high-performance solutions for asymmetric cryptosystems on FPGAs. More specifically, new design strategies for the asymmetric Paillier cryptosystem are proposed.

Most of the presented design strategies and implementations of cryptographic applications in this dissertation target Xilinx FPGAs. Hence, the presented results can be widely applied, where FPGA technology comes into play. The following topics have been investigated in this dissertation:

- Lightweight hardware architectures for Hummingbird and XTEA block Ciphers
- Compact Hardware architectures for cryptographic hash functions including Keccak, Skein and Grøstl.
- Improvements to the FPGA-based modular multiplication and exponentiation
- Implementations of Paillier public-key cryptosystem on FPGA

1.3 Aim of the Dissertation

This dissertation aims to focus on the combination of the research topics: cryptography, efficient security accelerators for communication systems and digital hardware design in constraint environments. The main objective of this dissertation is to show how computationally intensive cryptographic algorithms can efficiently be implemented in hardware to meet challenging trade-off requirements between performance and cost in communication systems. Typical applications for the coprocessors proposed in this dissertation is the use of them as efficient accelerators in the secure communication protocols such as IPSec protocol [10]. Finally, the dissertation aims to present efficient or high performance designs for a variety of computationally intensive cryptographic algorithms for secure digital communication. It is aimed to find a good trade-off between the constrained resources according to the selected application needs.

FPGA platforms present reconfigurable hardware resources whereas application specific integrated circuit (ASIC) technology offers very efficient solution for many high volume applications to assist general purpose processors in computing complex and intensive algorithms. Such hardware realizations

mostly aim to accelerate computationally intensive applications while some of them aim to build compact efficient architecture, where area and power metrics are very important. Compact designs try to find the best trade-off between required hardware resources and performance. Moreover, lightweight hardware architecture designs for cryptographic applications aim to yield better resource utilization where very limited resources are concerned in terms of memory, computing power and battery supply [4,11]. The aim of this dissertation is to find a good trade-off between cost and performance for cryptographic algorithms to implement them in reconfigurable hardware in order to provide the secure communication in electronic devices which have variable computing power.

1.4 Literature Review

1.4.1 Lightweight Cryptography for Block Ciphers

Resource constrained environments such as RFID tags, smart cards, wireless sensor network nodes, etc. require security mechanisms to ensure security and privacy of such applications by protecting stored data and communication from malicious attacks. For this reason, special cryptographic designs that can fit into the resource constrained devices have to be developed. In fact, there are special lightweight cryptographic primitives, for instance, PRESENT [11], Hummingbird [12] and lightweight DES variants [11]. They occupy less resources than other cryptographic primitives in hardware and software.

Poschmann [11] study on the lightweight cryptography for pervasive computing devices. Currently 98.8% of all manufactured microprocessors are deployed in embedded systems and only 1.2% remaining part are used in traditional computers. This shows that ubiquitous computing will be the next paradigm in information technology as Stajano foresees in his book [13]. The thesis mainly focuses on the technical aspects of security for pervasive computing by means of following different approaches. The thesis proposes a number of new lightweight cryptographic designs and their implementations for block ciphers, hash functions and public key identification schemes. It emphasizes strongly on lightweight hardware implementations that consume as little circuit area as possible. Mainly, Poschmann proposes new lightweight DES variants and gives their security consideration and hardware implementations. He developed a new ultra-lightweight block cipher called PRESENT. Its design

criterion, description, cryptanalytic aspects and hardware implementations are provided. In addition, The author presents lightweight hash functions and public key cryptography built around PRESENT algorithm. Consequently, the author reaches this conclusion that the lightweight cryptography is increasing demand for ubiquitous computing. PRESENT cryptographic algorithm was designed with good hardware performance in mind [11]. Their performance figures of different architectures present better resource utilization than other special lightweight cryptographic algorithms.

In literature, there are special ultra lightweight hardware architectures for Hummingbird lightweight cryptographic algorithm. In [14], loop-unrolled architecture and architecture with special speed optimized datapath were developed in order to implement Hummingbird in resource constrained devices. Most compact version is the architecture with speed optimized datapath. It utilizes 273 slices on FPGAs whereas our proposed XTEA requires only 110 slices together with strong cryptographic functionality of XTEA.

In study [15], an ultra-low power implementation of XTEA, TinyXTEA hardware architecture, was developed. A 128-bit key and 64-bit data, which are stored in a memory are accessed via 8-bit data bus. Our proposed architecture gives better area utilization and its performance results show that our Lightweight-XTEA architecture is better suited for low resource environments than [15].

There are many lightweight hardware architectures for AES and Camellia cryptographic algorithms in literature. In [16, 17], lightweight hardware architectures for AES algorithm were developed. A low area design was presented in [16] achieving 2.2 Mbps throughput ratio. In fact, an application specific instruction processor (ASIP) was developed with an 8-bit datapath and minimized ROM unit. Required area is small since 8-bit datapath is used by enabling serialization. Minimization of ROM size is also important for area utilization. This design occupies 264 slices including estimated area for memories. A very compact FPGA implementation for AES algorithm in [17] is proposed to be used in low-cost embedded devices. The design presented in [17] occupies 522 slices including estimated area for Block RAMs with 166 Mbps throughput ratio.

In addition to these studies, there are many recent related works in the area of lightweight cryptography. Low-area hardware designs are very valuable for resource constrained pervasive devices. This field attracts great interests of the researchers due to the rapid development of pervasive computing.

1.4.2 Compact Hardware Architectures for Cryptographic Algorithms

Compact hardware architectures for strong cryptographic primitives such as AES, Camellia and XTEA are proposed to target for low-cost embedded applications, where area and power are very important design criterion. Such strong cryptographic primitives provide more security than their lightweight counterparts in critical applications, where more security is crucial.

In [18], special hardware implementation is presented for Camellia on FPGAs. The design aims to suit for low area and low power applications where FPGA devices are used. Special optimizations are applied by using FPGA specifics such as shift registers for storing and scheduled key, distributed RAM for storing data in order to build efficient architecture. The study achieves 318 slice utilization with a throughput of 18.41 Mbps. In [19], two different small implementation for Camellia algorithm were proposed without giving so much detail about hardware architecture designs. However, performance figures are given for presented architectures. One of them is specialized for FPGA devices. This design occupies 874 CLB units on XC4000XL device. Small implementation proposed in that study is suited for low cost devices together with high level of security of Camellia algorithm by looking at its area utilization.

Guneysu [20] proposes novel compact implementations for symmetric and asymmetric cryptosystems on reconfigurable hardware. More specifically, the thesis investigates how the embedded function cores in FPGA devices can be utilized to significantly accelerate the operation of symmetric block ciphers such as AES as well as asymmetric cryptography, e.g., Elliptic Curve Cryptography (ECC) over NIST primes. The author also provides performance metrics of implementations in Graphics Processing Units (GPU) of high performance asymmetric cryptography. A second aspect of his thesis is cryptanalysis based on FPGA.

Compact hardware designs for cryptographic algorithms are needed to provide required performance with a small overhead in area and power consumption for specific applications. Compact designs utilize algorithm intrinsic properties to achieve best trade-off between performance and cost. In the light of these studies, cryptography for ubiquitous computing is very important research subject and it will increasingly continue to draw attention of the research community including mathematics and engineering.

1.4.3 High-Performance Designs for Public Key Cryptography

Eisenbarth [21] study on the security aspect of embedded system design in his thesis. The author emphasizes mainly on two alternative signature schemes and one public key encryption scheme. The author focuses on two main aspects, which are mainly cryptographic implementation and side channel analysis. The author proposes efficient designs for public key cryptographic algorithms which use some implementational tricks in order to implement successfully certain cryptographic schemes in hardware and software. The thesis explores how to efficiently analyze the side channel resistance of embedded implementations. Possible logic designs to protect the side channel attacks are proposed in the thesis. Possible hazards of a new developed attack to KeeLoq cipher-based remote keyless entry systems are presented in the thesis. Finally, advanced methods of side channel analysis are applied to microcontroller platforms to build a disassembler by passively observing only one side channel, its power consumption.

Novotný [22] studies scalable arithmetic units operating over the binary finite field GF(2^m) in the first part of his dissertation. Novotný also proposes four different architectures of the digit-serial normal basis multiplier that is developed in his dissertation. In the second part, he focuses on cryptanalysis of GSM communication that is encrypted with A51 cipher. Two, attacks which are supported by an existing low-cost special-purpose hardware device COPACOBANA, are developed against A5/1 cipher.

Public key cryptography, especially with homomorphic encryption property, are widely used popular methods for data protection in collaborative filtering schemes with privacy. Such schemes aim to provide accurate recommendations efficiently with privacy. Paillier cryptosystem is commonly used in these schemes [23]. In order to implement the Paillier cryptosystem, two core modules, modular multiplication and exponentiation, are needed. In fact, modular multiplication and exponentiation are two of the most important arithmetic operations in modern applied cryptography. Hence, their efficient implementations have been at the center of research activities in cryptographic engineering [5–7].

There is an extensive literature on efficient implementations of modular arithmetic operations including modular multiplication, inversion, and exponentiation [5–7]. There are many studies about efficient implementations of modular arithmetic operations included multiplication and exponentiation us-

ing various techniques. Montgomery [24] and Karatsuba [25] multiplication algorithms have been the most popular ones and they still constitute a base for newly proposed schemes.

Tenca and Koç [26] propose a radix-2 scalable Montgomery multiplication architecture, which multiplicand is scanned word-by-word and the other operand, multiplier, is scanned bit-by-bit. This multiple word radix-2 Montgomery multiplication allows efficient scalable hardware implementation. Parallelism among instructions of the algorithm in different scanning bits of multiplier is possible. The main difference and advantage of the architecture compared to the other algorithms in the literature is its scalability to any operand size, which, in fact, enables to find good design trade-offs for different application needs. The bipartite algorithm proposed by Kaihara *et al.* [27] uses a different representation of residue classes modulo M, which allows the splitting the operand multiplier into two parts. The upper part of the multiplier is processed by the interleaved modular multiplication and the lower part is processed using the Montgomery algorithm. These two parts are processed in parallel yielding two-fold increase in the speed of calculation. Sakiyama *et al.* [28] propose tripartite algorithm, which combines three existing algorithms including Barret reduction based modular multiplication, the Montgomery modular multiplication, and the Karatsuba multiplication algorithms. The tripartite algorithm provides reducing the computational complexity and increasing the parallel processing capacity by integrating these three different multiplication algorithms. Chow *et al.* [29] present a Montgomery multiplier that exploits Karatsuba algorithm for cryptography applications. The design uses multiple precision arithmetic techniques in order to make the critical path delay independent to bit width of the multiplier. They recursively implement Karatsuba multiplier to achieve n-bit multiplier to use in montgomery algorithm in one pass. In contrast to that study, we present hardware architecture using high-radix Montgomery multiplication algorithm, which incorporates Karatsuba algorithm together with a coprocessor technique. Radix-2^{17} is achievable by embedded multiplier placed in almost all FPGAs. In our planned study, higher radixes that are used in Montgomery multiplication are achieved efficiently by exploiting Karatsuba algorithm. This allow us to investigate very compact trade-offs between area and computation time for hardware implementations.

The algorithms proposed in these previous works are efficient for ASIC designs. In fact, radix-2 implementation is suitable for hardware implemen-

tations, especially ASIC designs. However, for FPGA implementations, using embedded built-in multiplier enables to achieve smaller clock cycle length and also provide better resource utilization on FPGA. If the algorithm is implemented in the FPGA, then built-in modules such as Block RAMs, DSP (Digital Signal Processor), Shift Registers, etc. in FPGA should be utilized to implement the algorithm for efficient computation. In the thesis, we aim to propose an architecture which efficiently uses such modules thanks to integrating Montgomery and Karatsuba algorithms. Song *et al.* [30] present an RSA encryption/decryption hardware based on Montgomery algorithm using only one DSP block and one Block RAM together with 180 slice placed in Virtex-6 FPGA. Their aim is to use minimum logic units with maximum use of a DSP block for modular exponentiation used in RSA algorithm. Their achieved results for one core is very good since very small amount of logic is used to compute the modular exponentiation. The proposed design in that study is well suited for resource constrained devices.

High-performance cryptography is required for high-bandwidth and high-speed networking systems; for example, high-speed data transmission between data centers used on the Internet. These studies show the importance of high-performance cryptography for high-speed information and communication systems. To reach this high-speed networking, very high-performance cryptographic hardware cores should be investigated and designed. To achieve this goal, advanced design techniques together with the algorithm dependent methods should be taken into account.

2. BACKGROUND

In this chapter, important preliminaries and definitions that are essential in the following chapters are presented. After a brief description of block ciphers in Section 2.1.1, cryptographic hash functions that are studied in this dissertation, the finalists in SHA-3 competition, are described (Section 2.1.2). The algorithms within the scope of this dissertation regarding the public key cryptography are explained in Section 2.1.3. Finally, Section 2.2 and 2.3 illustrate the fundamentals regarding the hardware design.

2.1 Cryptographic Preliminaries

2.1.1 Block Ciphers

2.1.1.1 The Hummingbird encryption

Hummingbird is neither a block cipher nor a stream cipher, but a rotor machine equipped with novel rotor-stepping rules [14]. It has a hybrid structure of block cipher and stream cipher with 16-bit block size, 256-bit key size, and 80-bit internal state. A top-level description of the Hummingbird encryption is given in Figure 2.1 which consists of four 16-bit block ciphers E_{k_1}, E_{k_2}, E_{k_3} and E_{k_4}, four 16-bit internal state registers RS1, RS2, RS3 and RS4, and a 16-stage Linear Feedback Shift Register (LFSR). PT_i represents the i-th plaintext block and CT_i represents the corresponding i-the ciphertext block. The 256-bit key K is divided into four 64-bit subkeys k_1, k_2, k_3 and k_4 which are used in the four corresponding block ciphers. For further details, see [12].

2.1.1.2 The XTEA cryptographic algorithm

In this section, we describe the encryption process for the XTEA cryptographic algorithm. Note that the information given in this section can be found in [31], they are included here for a complete presentation.

XTEA is a block cipher based on a Feistel network with 64-bit block size and 128-bit key size. A Feistel network representation of the one half-round and two half-rounds (one cycle) of the XTEA encryption are given in figure 2.2.encryption (or decryption) starts with splitting the 64-bit input

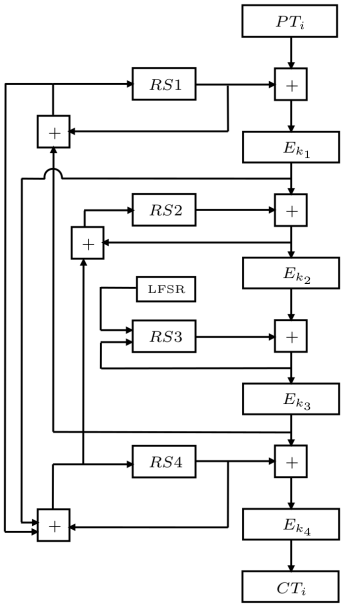


Figure 2.1: Hummingbird encryption (reprinted from [12]).

block into two halves which are then applied to a Feistel network for 32 rounds (suggested use). All additions within XTEA encryption are modulo.

In the algorithm [15, 31]:

- $\ll 4$ denotes the logical left shift by 4 bits
- $\gg 5$ denotes the logical right shift by 5 bits
- The bitwise XOR function is denoted as $\oplus$

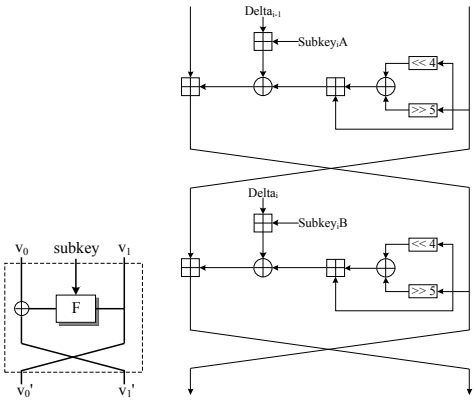


Figure 2.2: Feistel Network Representaion of rounds of XTEA.

- A permutation function:

$$f(V_1) = \{(V_1 \ll 4 \oplus V_1 \gg 5) + V_1\} \oplus subkey \tag{2.1}$$

- A subkey generation function: $sum + k(sum)$, where $k(sum)$ selects one block out of the four 32-bit blocks that comprise the key, depending on either bits 1 and 0 or bits 12 and 11 of sum register. A new value is computed between the first and the second half-round. It is incremented by a key schedule constant $\triangle$ during encryption.

2.1.1.3 The Advanced Encryption Standard

The round transformation of the AES operates on a 128-bit intermediate result, called state. The state is internally represented as a $N_l \times N_c$ array of bytes A, where N_l and N_c denotes the number of lines and columns, respectively. In the case of the AES, $N_l = N_c = 4$. Each byte $a_{i,j}$, $0 \leq i, j \leq 3$, is considered as an element of $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/(m(x))$, where the irreducible polynomial is given by $m(x) = x^8 + x^4 + x^3 + x + 1$. In the following, we encode an element of $\mathbb{F}_{2^8}$ by two hexadecimal digits: for instance, 95 is equivalent to $x^7 + x^4 + x^2 + 1$ in the polynomial basis representation. We denote the jth column of A by A_j. The number of rounds N_r as well as the number of 32-bit blocks in the cipher key N_k of the AES depend on the desired security level (Table 2.1).

Table 2.1: Block length, key length, number of 32-bit blocks of the key (N_k), and number of rounds (N_r) of AES-128, AES-192, and AES-256.

Algorithm	**Block length [bits]**	**Key length [bits]**	$\boldsymbol{N_k}$	$\boldsymbol{N_r}$
AES-128	128	128	4	10
AES-192	128	192	6	12
AES-256	128	256	8	14

The AES involves four byte-oriented transformations and their inverses for encryption and decryption, respectively [32]:

- The SubBytes step updates each byte of the state using an 8-bit S-box, denoted by S_{RD}. The inverse transformation is called InvSubBytes and denoted by S_{RD}^{-1}.

- The ShiftRows step simply consists of a cyclical left shift of the three bottom rows of the state by 1, 2, and 3 bytes, respectively.

- The MixColumns step is a permutation operating on the AES state column by column. Each column of the AES state is considered as a polynomial over $\mathbb{F}_{2^8}$, and is multiplied modulo $y^4 + 01$ by the constant polynomial $c(y) = 03 \cdot y^3 + 01 \cdot y^2 + 01 \cdot y + 02$ [32]. This operation is performed by multiplying each column of the state A by a circulant matrix $\mathcal{M}_E = \text{circ}(02, 03, 01, 01)$. During the inverse operation, called InvMixColumns, each column of the state is multiplied by $\mathcal{M}_D = \text{circ}(0E, 0B, 0D, 09)$.

- The AddRoundKey step combines the state A with a 128-bit round key. Let r denote the round index. Each byte $k_{i,4r+j}$ of the round key and its corresponding byte $a_{i,j}$ are added in $\mathbb{F}_{2^8}$ by a simple bitwise XOR operation. AddRoundKey is therefore its own inverse.

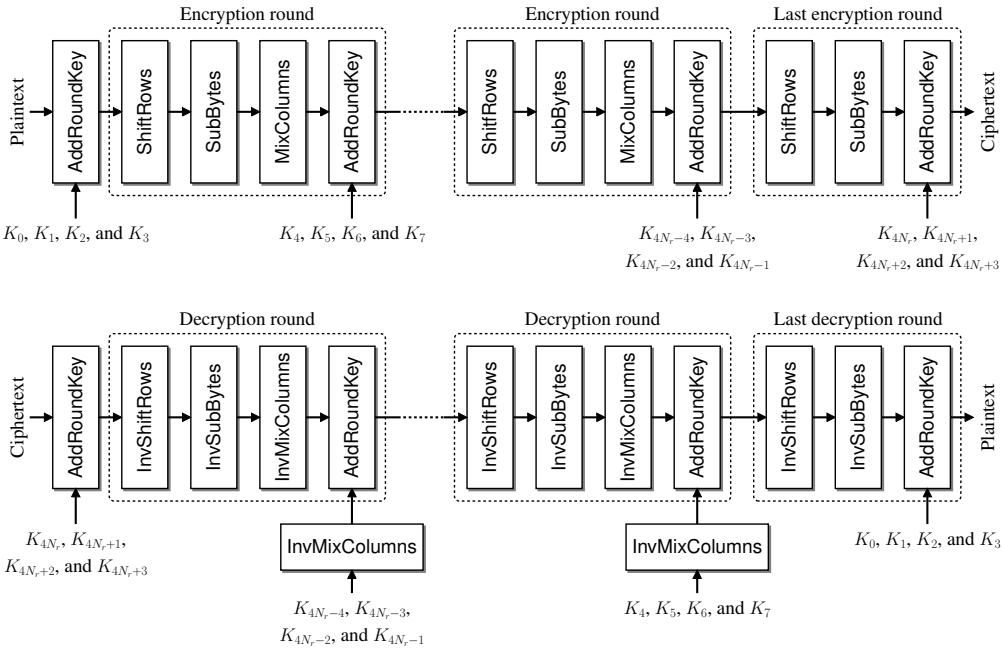


Figure 2.3: AES encryption and decryption flowcharts (reprinted from [33])

The round transformation of the AES operates on a 128-bit intermediate result, called state. The state is internally represented as a $4x4$ array of bytes A:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

Each byte $a_{i,j}$, $0 \leq i, j \leq 3$, is considered as an element of F_{2^8}. The AES involves four byte-oriented transformations namely SubBytes, ShiftRows, MixColumns and AddRoundKey, see [34] for further details.

2.1.1.4 The Threefish block cipher

Threefish operates entirely on unsigned 64-bit integers and involves only three operations: rotation of k bits to the left, bitwise exclusive OR, and addition modulo 2^{64}. Therefore, the plaintext P and the cipher key K are converted to N_w 64-bit words. Note that the number of words N_w and the number of rounds N_r depend on the key size (Table 2.2). The size of a plaintext block is given by $N_b = 8 \cdot N_w$ bytes.

Table 2.2: Number of rounds of Threefish for different key sizes (reprinted from [35]).

Key size [bits]	**# 64-bit words N_w**	**# rounds N_r**	**Block size N_b [bytes]**
256	4	72	32
512	8	72	64
1024	16	80	128

2.1.2 Cryptographic Hash Functions

2.1.2.1 The hash function Grøstl

Grøstl is a SHA-3 candidate. It is an iterated hash function with a compression function built from two fixed, large, distinct permutations. The design of Grøstl is transparent and based on principles very different from those used in the SHA-family. The two permutations are constructed using the wide trail design strategy, which makes it possible to give strong statements about the resistance of Grøstl against large classes of cryptanalytic attacks [36].

Grøstl is a family of cryptographic hash functions able to compute message digests from 8 to 512 bits [37]. We denote by Grøstl-n the algorithm with

a n-bit output and focus on the digest sizes specified for the SHA-3 competition (224, 256, 384, and 512 bits). The original message is padded, split into t message blocks of ℓ bits, and organized as an array of $N_l \times N_c$ bytes, where ℓ and N_c depend on the desired level of security (Table 2.3). The number of lines N_l is always equal to 8.

Table 2.3: Block length, number of column of the internal state, and number of rounds of Grøstl-n.

Digest size n [bits]	**Block length ℓ [bits]**	**# columns N_c**	**# rounds N_r**
8 to 256	512	8	10
264 to 512	1024	16	14

2.1.2.2 The hash function Keccak

Keccak is a family of hash functions that are based on the sponge construction and uses as a building block an iterated permutation. In the following, we first review the sponge construction, see [38] for further details.

The Sponge Construction: Sponge functions provide a way to generalize cryptographic hash functions to more general functions with arbitrary-length outputs. Figure 2.4 shows the illustration of the construction. The sponge construction operates on a state of $b = r + c$ bits, where r is the *bitrate* and c is the *capacity* which determines the attainable security level.

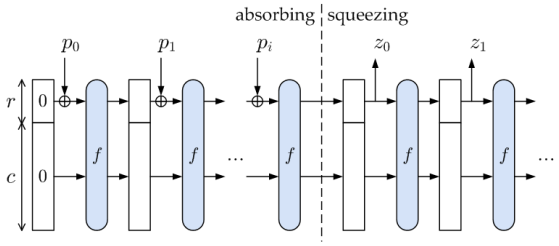


Figure 2.4: The sponge construction (reprinted from [38]).

The sponge construction consists of three phases:

- In the initialization phase, all the bits of the state are set to zero. The input message is padded and divided into blocks of r bits.
- In the absorbing phase, the r-bit input message blocks are XORed with the first r-bit of the state, and the resulting outputs are interleaved with

the function f. When all message blocks are processed, the sponge construction alters to the squeezing phase.

- In the squeezing phase, the first r-bit of the state is returned as output blocks and are also interleaved with the function f. The number of output blocks can be arbitrary and is chosen by the user.

Note that the last c-bit of the state is never directly affected by the input blocks and are never output during the squeezing phase. The sponge construction is known to have the following advantages [39]:

- Variable-length output (it can generate outputs of any length).
- Permutation-based (the underlying function f can be any permutation).
- Secure against generic attacks (the success probability of any generic attack is equal to that of the same attack applied to a random oracle).
- Moreover, this construction is flexible in the sense that different security-speed trade-offs can be achieved using the same permutation f.

KECCAK-f *Permutation:* The KECCAK-$f[b]$ is a permutation with width $b = 25 \times 2^l$, where l ranges from 0 to 6; hence, there are 7 such permutations, indicated by KECCAK-$f[b]$. The permutation KECCAK-$f[b]$ can be described on a state a, a three-dimensional array of elements over GF(2), where $a[x][y][z]$ denotes the bit in posisiton (x, y, z). The KECCAK-$f[b]$ consists of $n_r = 12 + 2l$ and each round R consists of five invertible step mappings, namely theta, rho, pi, chi, and iota [40]:

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

$\omega : a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1]$,
$\rho : a[x][y][z] \leftarrow a[x][y][z-(t+1)(t+2)/2]$,
with t satisfying $0 \le t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\mathrm{GF}(5)^{2\times 2}$,
or $t = -1$ if $x = y = 0$
$\pi : a[x][y] \leftarrow a[x'][y']$, with $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$
$\chi : a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2]$
$\iota : a \leftarrow a + \mathrm{RC}[i_r]$

where i_r denotes the round number and is the index from 0 to $n_r - 1$. These rounds are identical except the value of the round constants represented by $\mathrm{RC}[i_r]$. The addition and multiplication operations between the terms in these transformations are in $\mathrm{GF}(2)$. The round constants are given by:

$$\mathrm{RC}[i_r][0][0][2^j - 1] = \mathrm{rc}[j + 7i_r], \text{ for all } 0 \leq j \leq l,$$

and all other values of $\mathrm{RC}[i_r][x][y][z]$ are zero. The values $\mathrm{rc}[t] \in \mathrm{GF}(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$$\mathrm{rc}[t] = (x^t \mod x^8 + x^6 + x^5 + x^4 + 1) \mod x \text{ in } \mathrm{GF}(2)[x],$$

(see [40] for further details).

In order to facilitate the description of the individual mappings, the following naming conventions are used for parts of the Keccak- f state [38]. Let $w = 2^l$

- A *row* is a set of 5 bits with constant y and z coordinates
- A *column* is a set of 5 bits with constant x and z coordinates
- A *lane* is a set of w bits with constant x and y coordinates
- A *sheet* is a set of 5w bits with constant x coordinate
- A *plane* is a set of 5w bits with constant y coordinates
- A *slice* is a set of 25 bits with constant z coordinate

The Keccak team suggests using KECCAK-$f[1600]$ permutation which supports 256- and 512-bit message digests for the SHA-3 competition. In this section, a compact coprocessor is presented with its architectural details and the performance results of this coprocessor for the KECCAK-$f[1600]$ are given.

2.1.2.3 The Skein family of hash functions

The Unique Block Iteration (UBI) chaining mode allows one to build a compression function out of a tweakable encryption function. Let M be a message of arbitrary length up to $2^{99} - 8$ bits. If the number of bits in M is not a multiple of 8, we append a bit 1 followed by a (possibly empty) string of 0's. This step guarantees that M contains N_M bytes. Then, we pad M with p zero bytes so that $N_M + p$ is a multiple of the block size N_b. We can now split M into N_b-byte blocks $M_0, \ldots, M_{k-1}$, where $k = (N_M + p)/N_b$. Each block

M_i is processed with a unique tweak value T_i encoding how many bytes have been processed so far, a type field (see [35] for details), and two bits specifying whether it is the first and/or last block. The UBI chaining mode is computed as:

$$H_0 \leftarrow G,$$
$$H_{i+1} \leftarrow M_i \oplus E(H_i, T_i, M_i),$$

where G is a starting value of N_b bytes.

2.1.3 Public Key Cryptography

In this section, we introduce the basic notations and background of the proposed method. We first review the Paillier cryptosystem. In order to construct a Public Key cryptosystem (PKC) based on modular arithmetic, we use Montgomery reduction and right-to-left modular exponentiation algorithms, which are the standard algorithms for modular multiplication and exponentiation. Hence, we briefly introduce the Montgomery modular multiplication and exponentiation algorithms needed for performing modular arithmetic. We also give the preliminaries of the Karatsuba algorithm that we utilize for high-radix parallel multiplier in this study.

2.1.3.1 The Paillier homomorphic cryptosystem

In this section, we first describe the main components of the Paillier homomorphic cryptosystem (PHC). We then introduce the modular multiplication and exponentiation operations used in the PHC. We finally discuss Montgomery reduction and right-to-left modular exponentiation algorithms, which are employed to construct PCP.

Paillier [41] proposes a public key encryption scheme based on composite residuosity classes, usually referred to as the Paillier scheme. It is based on the problem to decide whether a number is an nth residue modulo n^2. The degree of classes is set to a hard-to-factor number $n = pq$, where p and q are two large prime numbers. The PHC offers the homomorphism if modular addition and multiplication are considered. These additive and multiplicative homomorphic property is utilized to maintain privacy in many PPAs. The system consists of key generation, encryption, and decryption steps. These steps use a public or a private key and some precomputed values. The encryption and decryption operations consist of modular exponentiations, multiplication, and some specific operations, which are described later in detail.

Paillier encryption uses an encryption key n, where n^2 is precomputed for encryption and decryption. Moreover, a random value r is computed from the range $[1, n-1]$ such that $gcd(r, n) = 1$. Then, a base $g \in \mathcal{B}$ is selected and checked by $gcd\left(L\left(g^{\lambda} \bmod n^2\right), n\right) = 1$. Finally, (n, g) pairs represent public keys and the pair (p, q) is kept private.

Encryption ($\mathcal{E}$) of PHC requires a modular exponentiation of base g and r. The computation is significantly enhanced by an acceptable choice of g, for instance, taking small numbers for g assures a speed-up by a factor of (1/3) [41]. We simply precompute $g^m \bmod n^2$ and the result is multiplied with $r^n \bmod n^2$. The same computation time can be achieved using two modular exponentiation modules. However, precomputation does not affect the security of the system, provided that the chosen value g fulfills the requirement $g \in \mathcal{B}$ imposed by the setting of Paillier scheme. A plaintext m, where $m, r < n$ can be encrypted and the ciphertext c can be obtained, as follows:

$$\begin{aligned} &\text{plaintext } m < n \\ &\text{select a random } r < n \\ &\text{ciphertext } c = g^m r^n \mod n^2 \end{aligned} \tag{2.2}$$

Decryption ($\mathcal{D}$) of PHC requires a modular exponentiation of base c and g. The computation is significantly improved by precalculating the modular inverse of $L(g^{\lambda} \bmod n^2)$ because g and λ are unique values for one key. A ciphertext c, where $c < n^2$ can be decrypted and the plaintext m can be obtained, as follows:

$$\begin{aligned} &\text{ciphertext } c < n^2 \\ &\text{plaintext } m = \frac{L(c^{\lambda} \bmod n^2)}{L(g^{\lambda} \bmod n^2)} \mod n \end{aligned} \tag{2.3}$$

in which

$$\begin{aligned} L(u) &= \frac{u-1}{n} \\ S_n &= \left\{u < n^2 | u = 1 \mod n\right\} \end{aligned} \tag{2.4}$$

Efficient Hardware Architecture for Paillier Cryptosystem: Today's FPGAs have enhanced hard cores such as DSP and BRAM components, which provide very high operating frequency. As emphasized by Guneysu *et al.*, "The use of device specific components lead to considerably higher system performance" [42]. For this purpose, we propose to adapt Montgomery and Karat-

suba algorithms to efficiently compute required operations of Paillier scheme on FPGA using device specific components such as DSP and BRAMs.

2.1.3.2 Modular exponentiation and multiplication

As seen from the Equation 2.2 and 2.3, PHC involves modular arithmetic. We use right-to-left binary modular exponentiation algorithm because there is no data dependency between modular multiplication operations in the algorithm. Hence, we improve the performance by means of implementing two modular multiplier modules in the PCP. We also take advantage of high-radix Montgomery modular multiplication operation due to its suitability to 7-family of FPGAs.

The PHC described in Section 2.1.3.1 is heavily based on modular exponentiation operations, which require huge computational complexity due to repetitive modular multiplications. Modular exponentiation and multiplication with big exponent and modulus (generally longer than 512-bits) are two of the most important arithmetic operations in several modern cryptographic algorithms. Efficient computation of the modular exponentiation is very important for PHC. The modular exponentiation can be realized by performing a series of modular squaring and multiplication operations. Hence, it is time-consuming in the case of large operand sizes. The goal of our high-throughput design is to decrease the execution time of each modular squaring and multiplication operations. Thus, we propose an architecture for PHC, which consists of high-speed modular multiplication module. One of the most efficient algorithm for performing modular exponentiation in hardware is the binary modular exponentiation algorithm, which is also known as square-and-multiply algorithm. We use right-to-left binary modular exponentiation to exploit the parallelism between squaring and multiplication operations. Algorithm 4 shows the pseudo-code of the modular exponentiation utilized in this study. The algorithm requires $2n$ modular multiplication operations in the worst case. Since there is no data dependency between modular squaring and multiplication in the algorithm, they can be performed in parallel. Due to parallel execution of such operations, modular exponentiation is completed in the order of $O(n\tau_{mm})$ time, where τ_{mm} is the time required to complete one modular multiplication operation.

One of the most efficient algorithm for performing modular multiplication in hardware is the Montgomery algorithm [24]. Algorithm 2 shows the pseudocode of the Montgomery multiplication in radix-2. The radix-2 Mont-

Algorithm 1 Right-to-left binary modular exponentiation

Require: Positive integers $B, E, M, -M^{-1}, R^2$ $E = \sum_{i=0}^{E_b-1} E_i 2^i$, $E_i \in \{0,1\}$, $R^2 = 2^{64d} \bmod M$
Ensure: $P = B^E \bmod M = \mathsf{ModExp}(B, E, M)$

```
1.  S_0 ← ModMult(R^2, 1, M);                    (Initial phase)
2.  Z_0 ← ModMult(R^2, B, M);
3.  for i ← 0 to n − 1 do
4.     if E_i = 1 then
5.        S_{i+1} ← ModMult(S_i, Z_i, M);          (Multiply)
6.     else
7.        S_{i+1} ← S_i;
8.     end if
9.     Z_{i+1} ← ModMult(Z_i, Z_i, M);           (Square)
10. end for
11. S_{n+1} ← ModMult(S_n, 1, M);                (Final phase)
12. return P ← S_{n+1};
```

gomery modular multiplication requires less area compared to high radix versions. However, its computational performance is worse than the Montgomery modular multiplication with high-radix values. High-radix brings area cost due to the need for high-radix multipliers. The use of high radix values in Montgomery reduction allow us to use hard multiplier blocks in FPGAs.

Algorithm 2 Radix-2 Montgomery modular multiplication

Require: odd $M, n = \lfloor log_2 M \rfloor + 1, X = \sum_{i=0}^{n-1} x_i 2^i$, with $0 \le (X, Y) < M$
Ensure: $Z = MP(X, Y, M) \equiv XY2^{-n} \bmod M)$, where $0 \le Z < M$

```
1.  S[0] = 0;                                    (Initialization)
2.  for i ← 0 to n − 1 do
3.     q_i ← (x_i · Y_0) ⊕ S[i]_0;                (Scanning)
4.     S[i + 1] ← (S[i] + x_i Y + q_i M)/2;
5.  end for
6.  if S[n] > M then
7.     S[n] ← S[n] − M;                          (Last reduction)
8.  end if
9.  return Z = S[n];
```

2.1.3.3 High-radix Montgomery Multiplication Algorithm

Many cryptosystems, especially PKC, generally computes $X \cdot Y \bmod M$ operation. One of the most efficient algorithm for performing this operation in hardware is the Montgomery algorithm [24]. Algorithm 3 shows the pseudo code for Montgomery multiplication in high-radix-2^k. The radix-2 Montgomery modu-

lar multiplication requires less area compared to high radix versions. However, its computational performance is worse than Montgomery modular multiplication with high-radix values. High-radix brings area cost due to the need for high-radix multipliers. The use of high radix values in Montgomery reduction allow to use hard multiplier blocks in FPGAs. We describe the technique in detail that we used to implement high radix Montgomery modular multiplication algorithm and the other necessary mathematical operations for a public-key cryptosystem.

Algorithm 3 Radix-2^k Montgomery Modular Multiplication

Require: $M = (M_{s-1}, \ldots, M_0)_r, X = (X_{s-1}, \ldots, X_0)_r, Y = (Y_{s-1}, \ldots, Y_0)_r$, where $0 \leq X, Y < M, r = 2^k, s = \lceil \frac{n}{k} \rceil, R = r^s$ with $gcd(M, r) = 1$ and $M' = -M^{-1} \bmod r$.

Ensure: $X \cdot Y \cdot R^{-1} \bmod M$

1. $Z[0] = (Z[0]_{s-1}, \ldots, Z[0]_0)_r$; (Initialization)
2. **for** $i \leftarrow 0$ **to** $s-1$ **do**
3. $\quad T[i] \leftarrow (Z[i]_0 + X_0 \cdot Y_i) \cdot M' \bmod r$; (Scanning)
4. $\quad Z[i+1] \leftarrow (Z[i] + X \cdot Y_i + M \cdot T[i])/r$;
5. **end for**
6. **if** $Z[s] > M$ **then**
7. $\quad Z[s] \leftarrow Z[s] - M$; (Last reduction)
8. **end if**
9. **return** $Z[s]$;

Montgomery high-radix algorithms have been proposed for improving the performance. However, as the radix increases, the design complexity and the length of clock cycle also increase dramatically due to requiring the use of larger digit multipliers. These high-radix designs generally consumes huge amounts of hardware area. So, previously, low-radix designs are more attractive for hardware implementation due to the lack of larger digit multipliers. However, dedicated 17-bit embedded multipliers can easily be found in almost all FPGAs today and implementing larger than 17-bit multiplier is also possible with Karatsuba algorithm.

By efficiently exploiting Karatsuba algorithm, one can achieve a larger bit multiplier by using less multiplier units. By using Karatsuba algorithm, doubling the bit-width of the multiplier is achieved in less amount of multiplier compared to classical multiplication. Our aim is to find a compact trade-off between the computation time and the required hardware area by efficiently exploiting Montgomery and Karatsuba algorithm together.

2.1.3.4 Karatsuba algorithm

The Karatsuba algorithm was introduced by Karatsuba and Ofman for multiplication of large integers [43]. The algorithm reduces the overall running time of the multiplication of two N-digit integers to $\mathcal{O}(N^{\log_2^3})$, as compared to $\mathcal{O}(N^2)$ in the classical algorithm. One multiplication is replaced with a three addition or subtraction operations. It is thus faster than the classical multiplication algorithm. Karatsuba algorithm reduces the number of single precision multiplications by taking advantage of two intermediate partial products.

To illustrate the Karatsuba algorithm, let X and Y be two $2k$-bit unsigned integers. One can split them as follows

$$X = 2^k \cdot X_1 + X_0 \text{ and } Y = 2^k \cdot Y_1 + Y_0$$

Four k-bit multiplication and three addition operations are required to compute the product $X \cdot Y$ in classical long multiplication operation which is illustrated in Equation 2.5.

$$\begin{aligned} X \cdot Y &\leftarrow 2^{2k} \cdot M_\beta + 2^k \cdot M_\gamma + M_\alpha \\ M_\beta &\leftarrow X_1 \cdot Y_1, \\ M_\gamma &\leftarrow X_0 \cdot Y_1 + X_1 \cdot Y_0, \\ M_\alpha &\leftarrow X_0 \cdot Y_0 \end{aligned} \tag{2.5}$$

Karatsuba observed that the middle term M_γ can be computed by using two intermediate partial products, M_α and M_β.

Equation 2.6 shows the computation of the middle term in Karatsuba method.

$$\begin{aligned} M_\gamma &\leftarrow X_0 \cdot Y_1 + X_1 \cdot Y_0 \\ &\leftarrow X_1 \cdot Y_1 + X_0 \cdot Y_0 + (X_0 - X_1) \cdot (Y_1 - Y_0) \\ &\leftarrow M_\beta + M_\alpha + K_\alpha \cdot K_\beta \end{aligned} \tag{2.6}$$

Then, from equation (2.5) and equation (2.6), the product $X \cdot Y$ is computed with three k-bit multiplications, four additions, and two subtractions. So, the Karatsuba method saves a quarter of the multiplications at the cost of some extra additions and subtractions.

2.1.3.5 Montgomery Exponentiation Algorithm

Most of PKC are based on heavily modular exponentiation operations. This operation requires huge computational complexity due to performing repetitive modular multiplication. Modular exponentiation and multiplication with very big exponent and modulus (specifically longer than 512-bits) are two of the well-known important arithmetic operations in several modern cryptographic algorithms. It has great importance in modern cryptographic algorithms for authentication, electronic signature, and key exchange.

Efficient computation of the modular exponentiation is very important for PKC. The modular exponentiation can be realized by performing a series of modular squaring and multiplication operations. Hence, it is time-consuming operation in the case of large operand sizes. Our approach to achieve high-throughput design is to decrease the execution time of each modular squaring and multiplication operations. In this article, we mainly propose an architecture for public-key cryptosystem which consists of high-speed modular multiplication module.

Algorithm 4 Right-to-Left binary modular exponentiation

Require: Positive integers $B, E, M, -M^{-1}, R^2$ $E = \sum_{i=0}^{E_b-1} E_i \cdot 2^i$, $E_i \in \{0,1\}$, $R^2 = 2^{64 \cdot d} \bmod M$
Ensure: $P = B^E \bmod M = ModExp(B, E, M)$
1. $S_0 \leftarrow ModMult(R^2, 1, M)$; (Initial phase)
2. $Z_0 \leftarrow ModMult(R^2, B, M)$;
3. **for** $i \leftarrow 0$ **to** $n-1$ **do**
4. **if** $E_i = 1$ **then**
5. $S_{i+1} \leftarrow ModMult(S_i, Z_i, M)$; (Multiply)
6. **else**
7. $S_{i+1} \leftarrow S_i$;
8. **end if**
9. $Z_{i+1} \leftarrow ModMult(Z_i, Z_i, M)$; (Square)
10. **end for**
11. $S_{n+1} \leftarrow ModMult(S_n, 1, M)$; (Final phase)
12. **return** $P \leftarrow S_{n+1}$;

One of the most efficient algorithm for performing modular exponentiation in hardware is the binary modular exponentiation algorithm which is also known as square-and-multiply algorithm. We use right-to-left binary modular exponentiation to exploit the parallelism between the squaring and multiplication operations. Algorithm 4 shows the pseudo-code for modular exponentiation that we use in this study. Algorithm 4 requires $2n$ modular multiplication

operations in worst case. Since there is no data dependency in between modular squaring and multiplication in this exponentiation algorithm, they can be performed in parallel. Thanks to the parallel execution of modular squaring and multiplication operations, modular exponentiation is completed in $n \cdot \tau_{mm}$ time where τ_{mm} is the time required to complete one modular multiplication operation.

2.2 FPGA Specifics

Most of the presented design strategies and implementations of cryptographic applications in this thesis target Xilinx FPGAs. Xilinx Inc is the current market leader in FPGA technology, hence, the presented results can be widely applied where FPGA technology comes into play. We mainly use the Spartan-3 and Virtex-5 FPGA devices, which are explained below.

2.2.1 Xilinx Spartan-3 FPGA

The Spartan-3 generation of FPGAs [44] is specifically designed to meet the needs of high volume, cost-sensitive electronic applications. They have dedicated carry logic together with various arithmetic logic gates that support wide logic and arithmetic functions.

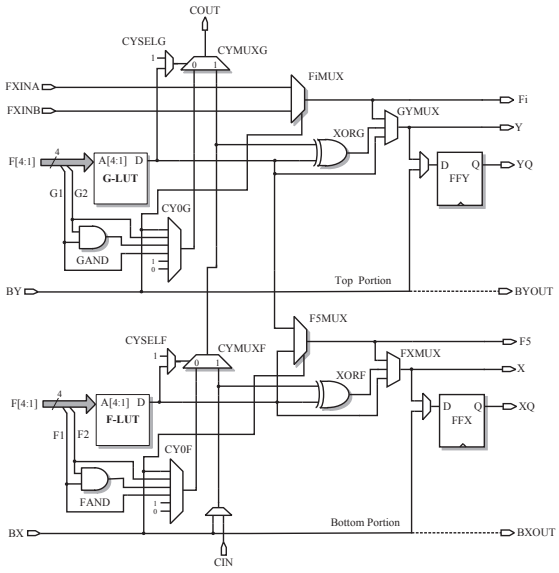


Figure 2.5: Simplified architecture of a Spartan-3 slice.

Figure 2.5 shows the architecture of a Spartan-3 slice [44]. A slice includes two 4-input Lookup Table (LUT) function generators (F-LUT and G-LUT),

two storage elements (flip flops FFX and FFY), a carry logic (multiplexers CYSELF, CYMUXF, CYSELG, CYMUXG and gates FAND, GAND, XORF, and XORG) and two wide function multiplexers (F5MUX and FiMUX). Some slices also provide two 16x1 distributed RAM and two 16-bit shift registers as an additional blocks.

2.2.2 Xilinx Virtex-5 and-6 FPGA

The Xilinx Virtex-5 FPGA provides architectural elements designed for maximum performance, higher integration, and lower power consumption making a good choice for the coprocessor approach. Figure 2.6 shows the quarter of a slice in Virtex-5 FPGAs [45]. Each slice contains four function generators (LUTs), four storage elements, arithmetic logic gates for carry chain logic, and some multiplexers. The other elements in the slice are mostly used for routing purposes.

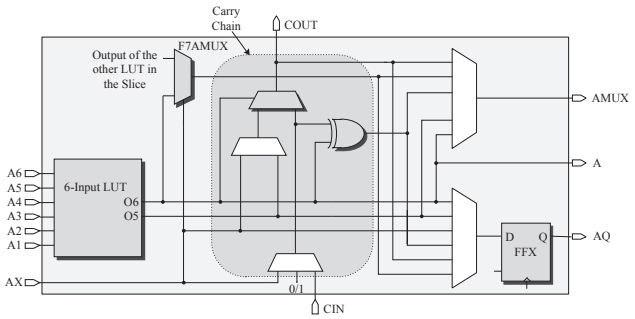


Figure 2.6: The quarter of a slice in Virtex-5 FPGA.

Each slice has multiplexers that can be used to obtain larger multiplexers. In our implementation for example, we obtained 8-input multiplexer using the F7MUX units.

The 6-input LUT architecture [46] provided by FPGA enables to obtain complex logic in a slice. Figure 2.7 illustrates the block diagram of a 6-input LUT which has six independent inputs and two independent outputs. Each LUT can implement any arbitrary 6-input Boolean function. In addition, one 6-input LUT can implement two arbitrary Boolean functions provided that two constituent 5-input LUTs share the same inputs. Note that the propagation delay is independent of the function implemented on the LUT.

The other architectural element we exploited in the thesis is dual ported 36-Kbit BRAM which has independent data, address and write enable buses. Our datapath has only LUT delay, F7MUX delay and some routing delay for

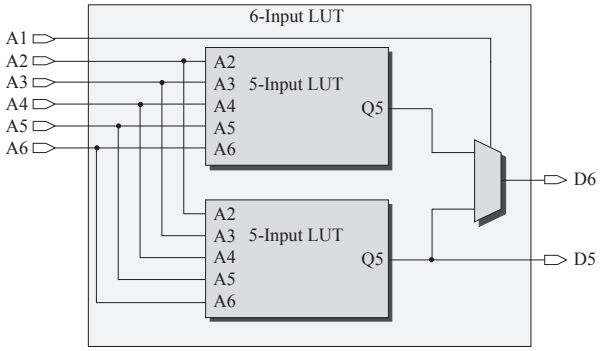


Figure 2.7: Block diagram of a Virtex-5 6-Input LUT.

control and data signaling since the combinational logic is realized in LUTs and their outputs are stored immediately in the registers inside the slices. The FPGA units including BRAMs that we used in our design can run at up to 550 MHz which is the maximum frequency rate for the device. With careful placing and routing of the components, this maximum frequency of 550 MHz can be attained yielding performance improvements in terms of overall throughput. Further throughput improvement can be obtained via replicating the hardware architecture.

2.2.3 System-on-chip design

With the rapid improvement in VLSI technology, the number of components that can be placed on a single chip has been continuously rising. This leads various computationally intensive applications to be implemented as a SoC design. A typical SoC consists of processors, peripherals, or application specific coprocessors. In this work, we propose a SoC design that accelerates modular arithmetic operations using the proposed coprocessor. Note that if a higher security level is desired, then more computational power is needed since strong cryptographic algorithms should be implemented in the system. In pure software systems, high level languages such as C ease the integration of the security primitives. We use the Zynq platform in order to include such a level of abstraction. This processor allows performing computationally intensive tasks required by the modular operations in the hardware domain linked by a custom peripheral to the processor. The proposed work employs Hardware/Software codesign methodology harnessing benefits of both development methodologies. The operations required by PKC are performed in two parts: The software part is executed on the processor and handles control tasks such as message communication. The hardware part is implemented on

the coprocessor and aims to accelerate modular arithmetic operations.

2.2.4 MicroBlaze based embedded system

Embedded systems consisting of a processor and a several custom-specific co-processor are proved to be very efficient for many applications in the field of digital signal processing, cryptography, and image processing. In such an embedded system, custom coprocessor is able to accelerate the computational intensive tasks while the processor placed in the FPGA is able to perform other tasks. The Xilinx MicroBlaze soft processor is a 32-bit RISC processor with optimized instruction set for embedded applications. The MicroBlaze soft processor includes several optional interfaces so that custom peripherals, coprocessors, memory units, and input-output units can be connected to build a complete embedded system on the FPGA via some dedicated tools provided by Xilinx.

Connecting a special-purpose hardware coprocessor on the PLB and FSL buses is an elegant way of increasing the efficiency and computational power of the MicroBlaze based embedded systems. Embedded processor based systems within FPGAs exploit the benefits of both software and hardware: flexibility of the software and processing power of the custom hardware.

2.2.4.1 Communication methods in MicroBlaze

On-chip peripherals are connected to the processor via the data and address buses. In MicroBlaze based embedded systems, two types of buses are available for communication between the coprocessor and the processor namely Processor Local Bus (PLB) and Fast Simplex Link (FSL). Between the two, the FSL bus provides faster communication channel.

2.2.4.2 FSL based communication in MicroBlaze

The FSL is a uni-directional point to point First In First Out (FIFO) based commu-nication channel bus and it provides a mechanism for unshared and non-arbitrated communication [47]. This bus is used for fast transfer of data words between the MicroBlaze and its coprocessors.

The FSL bus is driven by one Master device and drives one Slave device. Figure 2.8 illustrates the working principle of the FSL bus system and available signals. Note that the FSL can support both asynchronous and synchronous

FIFO modes. In the asynchronous FIFO mode, the master and slave sides of the FSL can operate at different clock rates. This allows the hardware accelerator, for instance, the Keccak coprocessor, to run at higher frequencies than the processor.

FSL read and write instructions transfer 32-bit width data from the FSL port to a MicroBlaze register and vice-versa. Instructions support blocking and non-blocking read/write operations.

Figure 2.8: Block diagram of an FSL with Master and Slave signal interface.

2.2.4.3 PLB based communication in MicroBlaze

The PLB v4.6 provides bus infrastructure for connecting a certain number of PLB masters or slaves into an overall PLB of a MicroBlaze based system [48]. It consists of various functional units that are required to build an embedded system such as a bus control unit, a watchdog timer and, separate address, write and read data paths units.

2.2.5 Zynq-based embedded system

With high-end processing systems like the Xilinx Zynq-7000 all programmable SoC, it is possible to fully utilize both processing system (PS) and custom coprocessors connected to the PS within a chip. This allow us to exploit both hardware and software design methodologies. In the following, we describe our SoC architectures implemented on Zynq-7000 based embedded SoC platform.

Extensible processing platform with Zynq-7000 family device combines an ARM dual-core Cortex-A9 MPCore processing system with Xilinx 28 nm unified programmable logic architecture (Figure 2.9). The ARM dual-core Cortex-A9 processor delivers high-performance capabilities by consuming less power compared to ARM Cortex-A8 solutions. ARM MPCore technology offers a coherent interconnect mechanism within the dual-core CPU platform. The Cortex-A9 processor includes several optional interfaces so that custom peripherals, coprocessors, memory units, and input-output units can be con-

nected to build a complete embedded system on the FPGA via some dedicated tools provided by Xilinx.

Connecting a application specific hardware coprocessor on the processor bus is an elegant way of increasing the efficiency and computational power of an embedded system. Embedded processor based systems within FPGAs exploit the benefits of both software and hardware: flexibility of the software and processing power of the custom hardware. In the design, we transform the slowest part of the system which is computationally intensive modular arithmetic operations into the hardware domain to accelerate the system. The addition of a custom hardware to a processor will consume FPGA resources; however, this often will not be a problem for systems since surplus resources are available in FPGAs. Hence, the careful use of these spare resources provides a powerful, simple, and most importantly cost-free solution for FPGA based systems.

Data transfer overhead has a crucial impact on the execution time even when low latency Zynq buses, AXI based interconnect, are used. Nevertheless, still very high speed-ups are achieved in the proposed work.

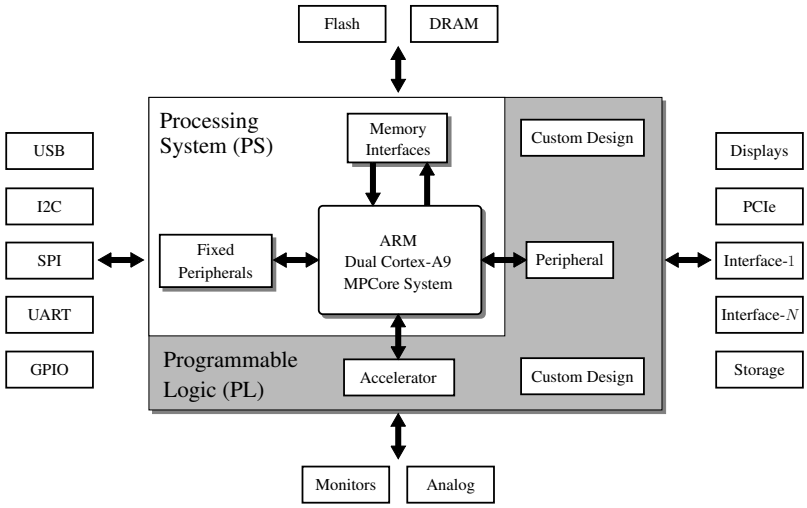


Figure 2.9: A generic Zynq-7000 all programmable SoC chip based embedded system and available busses.

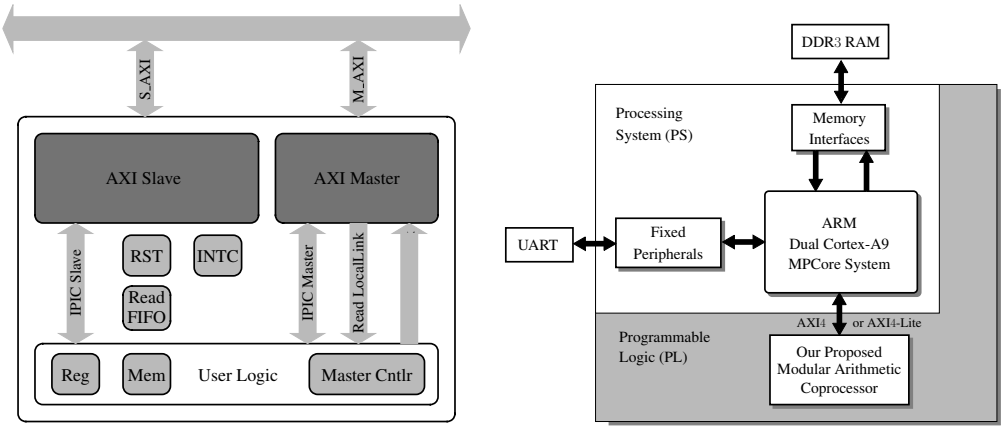
2.2.5.1 AXI based communication in Zynq

On-chip peripherals are connected to the processor via the data and address buses. In Zynq-based extensible processing platform, ARM AXI4 protocol is used for communication between the accelerator and the processor.

There are two types of AXI4 interfaces which we used in this study is as follows:

- AXI4-Lite provides simple, low-throughput memory-mapped communication (for example, to and from control and status registers).
- AXI4 meets the high-performance memory-mapped requirements.

Figure 2.10a shows the available components provided by the AXI interconnect for various control mechanisms. We utilize RST, Read FIFO and Reg modules for fast communication between the ARM processor and our accelerator. Figure 2.10b shows the connections between the proposed coprocessor and the ARM dual Cortex-A9 processor within Zynq-7000 device via the two types of AXI4 communication protocol.



(a) AXI Interface modules

(b) The configurations of the proposed SoC platforms

Figure 2.10: Communication details.

AXI4-Lite is an area-efficient subset of the AXI protocol. This type of AXI interconnect protocol is able to send only one data word per transaction with less amount of hardware cost. Thus, there is a communication overhead while sending for each data word from the processor to our modular arithmetic accelerator due to the transmitting the control data for setting up each transaction. Simple status and control registers are enough for communication.

AXI4 is a conventional single-address burst interconnect which supports up to 256 data beats per burst, the width of which is system dependent. In this type of AXI communication, the communication overhead is decreased with extra area cost. AXI4 is more suitable for compared to AXI4-Lite since it is capable of carrying more data for each transaction.

2.3 Performance Evaluation

We will capture our architectures in the VHDL language and prototype the implementations of the proposed methods for aforementioned cryptographic algorithms on a Xilinx Virtex-5 FPGA. We synthesize and place-and-route our designs by Xilinx ISE tools. We also simulate our design with Xilinx Simulator to verify the correctness of the proposed architecture. We will provide the performance evaluation results with detailed discussions and compare our performance results with the figures of existing implementations. Our goal is to find a good trade-off between hardware area with throughput for compact hardware architectures of cryptographic hash functions. The other goal is to maximize the performance of Paillier cryptosystem on FPGA.

There are three performance metrics that are used to evaluate the performance of the proposed hardware designs. They can be explained, as follows:

2.3.1 Circuit size

Required hardware area is very important in efficient hardware designs, especially for lightweight cryptography. The low-cost smart devices are rapidly growing. The most important limiting factor of lightweight devices, e.g. RFID tags and wireless sensors, is circuit size (area). The typical power consumption for RFID tags determines the area requirements of the tags. Therefore, we aim to develop area-efficient hardware architectures for lightweight cryptography. We measure the size of the architecture implemented in FPGA. We compare our achieved area metrics with the existing ones in the literature for specific algorithms.

2.3.2 Latency

Latency measures the time (we will typically consider the worst-case scenario) to complete the transformations required by the considered cryptographic algorithm.

The latency time is the time required by an implementation to derive the key dependent data required (e.g. sub-keys) to commence encryption. Our planned FPGA implementations of compact and high-performance cryptographic cores will achieve significant reduction in the latency time. On the contrary, the latency time of the lightweight cryptographic cores will be slow due to the utilized serialization technique.

2.3.3 Throughput

Throughput roughly measures the number of completed transformations per second. More precisely, The throughput corresponds to the amount of data encrypted per time unit. Our planned FPGA implementations achieve throughput improvements of 4-20 times compared with the software-based results. While the software implementations do not achieve processing rates higher than 30 Mbits/sec [34], our FPGA implementations for high-performance cryptography achieve processing rates higher than 100 Mbits/sec. For one thing, software implementations cannot exploit the inherent parallelism of a cryptographic round. For another, the operations required by each cryptographic round can be executed more efficiently in FPGAs than in a general-purpose computer. By using a superior platform configuration than the reference platform for the NIST Efficiency Test for Round 1 AES Candidates [34], higher throughput can be achieved for the software implementations. However, even in this case, the speed-up of the FPGA implementations would still be remarkable (i.e., Rijndael, Serpent). By realizing "multipleround" FPGA implementations, the throughput speed-up can be significantly improved for operation modes that allow concurrent processing of multiple blocks of data.

3. LIGHTWEIGHT HARDWARE ARCHITECTURES FOR BLOCK CIPHERS

In this chapter, two novel lightweight hardware architectures for Hummingbird and XTEA block ciphers are presented. Due to the rapid development of ubiquitous computing, resource constrained devices have been broadly used in many applications. Security need to be employed in those applications where sensitive information is transferred or stored. To provide security needs to such constrained environments, hardware architectures for these two algorithms under low-area constraint are studied and proposed in this chapter.

3.1 Introduction and Motivation

The low-cost smart devices are rapidly becoming pervasive in our daily life. Well known applications include electronic passports, contactless payments, product tracking, access control, and supply-chain management, just to name a few, are examples of resource-constrained environments. The limiting factors of such lightweight devices, e.g. RFID tags and wireless sensors, are price, power, and area. The typical power consumption for RFID tags determines the area requirements of the tags. A considerable body of research has been focused on providing cryptographic functionality to resource-constrained devices, while limited computational and storage capabilities of low-cost smart devices make the problem challenging. This emerging research area is usually referred to as *lightweight cryptography*, which has to deal with the trade-off among security, cost, and performance [4].

Hummingbird is an ultra-lightweight cryptographic algorithm aiming at resource-constrained devices. In this chapter, a novel enhanced hardware implementation of the Hummingbird cryptographic algorithm that is based on the memory blocks embedded within Spartan-3 FPGAs is proposed. The enhancement is not only from the introduction of the coprocessor approach but also from the employment of serialized data processing principles. Due to the compactness of the proposed architecture, remaining reconfigurable area in FPGAs can be used for other purposes. Comparisons to the other reported FPGA implementation of the Hummingbird cryptographic algorithm indicate that the proposed architecture outperforms the previous work in terms of both

efficiency and area. It is remarked that the architecture proposed in this section can also be used as stand-alone although it is built via coprocessor approach.

Security of resource-constrained devices like RFID tags, smart cards, and sensor networks is becoming increasingly important. Strong cryptographic algorithms such as AES and Camellia require a great deal of area than these devices can provide. To this affect, extended tiny encryption algorithm (XTEA) is seen as a powerful candidate to provide security functionality to resource-constrained devices. In this chapter, a new lightweight architecture for the XTEA cryptographic algorithm is also proposed so that it can be used in such environments. The small footprint hardware is achieved by means of 8-bit serialization of the algorithm. We compare our performance results with other reported FPGA implementations of the lightweight as well as strong cryptographic algorithms. Our proposed architecture consumes the smallest area among all with a decent efficiency.

3.2 Related Work

FPGA platforms present reconfigurable hardware resources whereas ASIC technology offers very efficient solution for many high volume applications to assist general purpose processors in computing complex and intensive algorithms. Such hardware realizations mostly aim to accelerate computationally intensive applications while some of them aim to build compact efficient architecture where area, power metrics are very important. Compact designs try to find the best trade-off between required hardware resources and performance. Moreover, lightweight hardware architecture designs for cryptographic applications [49, 50] aim to yield better resource utilization where very limited resources are concerned in terms of memory, computing power and battery supply.

Resource constrained environments such as RFID tags, smart cards, wireless sensor network nodes etc. require security mechanisms to ensure security and privacy of such applications by protecting stored data and communication from malicious attacks. For this reason, special cryptographic designs that can fit into the resource constrained devices have to be developed. In fact, there are special lightweight cryptographic primitives, for instance, PRESENT, HIGHT, Hummingbird, Clefia and lightweight DES variants. They occupy less resources than other cryptographic primitives in hardware and software.

In literature, there are special ultra lightweight hardware architectures

for Hummingbird lightweight cryptographic algorithm. In [51], loop-unrolled architecture and architecture with special speed optimized datapath were developed in order to implement Hummingbird in resource constrained devices. Most compact version is the architecture with speed optimized datapath. It utilizes 273 slices on FPGAs whereas our proposed XTEA requires only 110 slices together with strong cryptograpic functinality of XTEA. Compact efficient hardware architecture for Hummingbird cryptographic algorithm is presented in [52]. The proposed architecture uses coprocessor approach and serialization of the algorithm in order to decrease area utilization to be used in low-cost devices. Intrinsic properties of Hummingbird cryptographic algorithm is exploited to find good datapath for this coprocessor. In addition to this, special instruction scheduling is applied to allow execution of multiple instructions at the same time. This architecture gives better efficiency than [51].

PRESENT cryptographic algorithm was designed in [50] with good hardware performance in mind. Their performance figures of different architectures present better resource utilization than other special lightweight cryptographic algorithms.

Compact hardware architectures for strong cryptographic primitives such as AES, Camellia and XTEA are proposed to target for low-cost embedded applications where area and power are very important design criterion. Such strong cryptographic primitives provide more security than their lightweight counterparts in critical applications where more security is crucial.

In study [15], an ultra-low power implementation of XTEA, TinyXTEA hardware architecture, was developed. A 128-bit key and 64-bit data which are stored in a memory are accessed via 8-bit data bus. Our proposed architecture gives better area utilization and its performance results show that our Lightweight-XTEA architecture is better suited for low resource environments than [15].

There are many lightweight hardware architectures for AES and Camellia cryptographic algorithms in literature. In [16, 17], lightweight hardware architectures for AES algorithm were developed. A low area design was presented in [16] achieving 2.2 Mbps throughput ratio. In fact, an application specific instruction processor (ASIP) was developed with an 8-bit datapath and minimized ROM unit. Required area is small since 8-bit datapath is used by enabling serialization. Minimization of ROM size is also important for area utilization. This design occupies 264 slices including estimated area for memories which is higher than 110 slices of our architecture area utilization. A very

compact FPGA implementation for AES algorithm in [17] is proposed to be used in low-cost embedded devices. The design presented in [17] occupies 522 slices including estimated area for Block RAMs with 166 Mbps throughput ratio. Area utilization is bigger than area utilization of our architecture on FPGAs.

In [18,19], compact hardware implementations of Camellia cryptographic algorithm were proposed. In [18], special hardware implementation is presented for Camellia on FPGAs. The design aims to suit for low area and low power applications where FPGA devices are used. Special optimizations are applied by using FPGA specifics such as shift registers for storing and scheduled key, distributed RAM for storing data in order to build efficient architecture. The study achieves 318 slice utilization with a throughput of 18.41 Mbps. In [19], two different small implementation for Camellia algorithm were proposed without giving so much detail about hardware architecture designs. However, performance figures are given for presented architectures. One of them is specialized for FPGA devices. This design occupies 874 CLB units on XC4000XL device. Small implementation proposed in that study is suited for low cost devices together with high level of security of Camellia algorithm by looking at its area utilization.

Although XTEA cryptographic algorithm was not proposed for resource constrained environments whereas it was proposed to provide strong cryptography, our lightweight architecture of XTEA is well-suited to resource constrained devices. Together with this, related work demonstrates our architecture has very good area utilization among all compact architectures of strong and lightweight cryptographic algorithms.

3.3 Lightweight Hardware Architectures for Hummingbird

A novel ultra-lightweight cryptographic algorithm, referred to as Hummingbird, is proposed for resource-constrained devices [12]. The design of the Hummingbird cryptographic algorithm is motivated by the well-known Enigma machine. Hummingbird has a hybrid structure of block cipher and stream cipher and it has been shown to be resistant to the most common attacks to block ciphers and stream ciphers including birthday attack, differential and linear cryptanalysis, algebraic attacks, etc. [12,14,53]. Recently, a new version of the algorithm known as Hummingbird-2 is proposed which produces authen-

tication tag for each message processed. Compared to the previous version, the internal state in the algorithm has been enlarged to 128 bits and a flow of entropy from the state to the mixing function has been improved [54].

In this section, the details of compact and efficient hardware architecture for the hummingbird cryptographic algorithm are presented. The proposed architecture is suited for the applications where low-cost FPGAs are deployed such as FPGA-based tags [55]. These tags use the Spartan-3 XC3S1000 FPGA from Xilinx as a programmable device, which provides a simple package footprint, features and reasonable hardware resources so that security enhanced RFID systems can be built. Due to the compactness of our architecture, the rest part of the reconfigurable area in FPGA can be used for implementing other functionalities. For example, multiple instances of our architecture can be run in parallel so that higher throughputs can be obtained. By this way, approximately 7 multiple instances of our architecture can be installed using the same area as in [14] yielding higher throughputs. The results presented in this section have been published in [52].

3.3.1 Hummingbird coprocessor

We use the coprocessor approach, which takes advantage of the embedded memory blocks in FPGA to implement the Hummingbird cryptographic algorithm. This approach reduces the area requirements in terms of slices because only the datapath and linear feedback shift register (LFSR) module are realized in slices. With this approach, efficiency (throughput/occupied slices) of the FPGA implementation of the Hummingbird cryptographic algorithm is increased considerably with respect to the previously reported FPGA implementations of the algorithm [14], see Table 3.1.

One of the primitives needed in the implementation of Hummingbird cryptographic algorithm is a modulo 2^{16} addition of two 16-bit numbers. A 2-bit summation can be realized using one slice. Hence, 16-bit addition is implemented using eight slices. A 2-bit full-adder calculates the sum of two bits x and y on the same level and carry input (cin), if any transferred from a previous bit level, yielding $sum = x \oplus y \oplus cin$ and carry out,

$$cout \leftarrow \begin{cases} x & \text{if } x \oplus y = 0, \\ cin & \text{if otherwise} \end{cases}$$

Dedicated carry logic transfers the carry bit through CYSELF, CY0F and

CYMUXF multiplexers. F-LUT function generator produces $x \oplus y$. The output of F-LUT and *cin* bit is XORed by the XORF gate. This provides the sum. The sum result is then stored in FFX flip flop. Carry out bit is produced by CYMUXF multiplexer where x and *cin* are the inputs and the F-LUT output is used as a selection line.

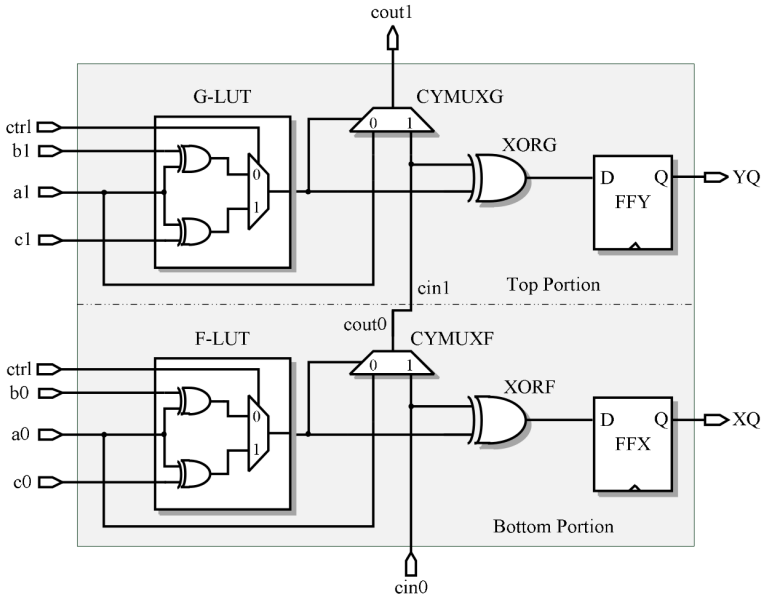


Figure 3.1: Three input adder architecture in one slice.

Figure 3.1 shows a 1-bit addition of three different input sources using elements of a Spartan-3 slice. Here, one of the sources is connected to the carry chain logic immediately. The other one is selected using the control signal in the LUT function generator. The LUT provides the result of XOR operation of two input signals $a \oplus b$ or $a \oplus c$ depending on the control signal. Carry out bit is transferred to upper slices so that to extend the 1-bit addition to 16-bit addition. The flip- flops store the sum value.

Other primitives needed in the implementation of Hummingbird cryptographic algorithm are realized in LUT function generators. There is no need to use arithmetic gates or carry chain logic provided in a slice. As described earlier, one slice includes two LUTs and two storage elements and all stages operate on 16-bit data since the datapath is 16-bit wide. So, the required slice count for each stage is eight.

We note that all Virtex and Spartan FPGAs consist of many embedded memory blocks to store large amount of data compared to flip flops in the slices. They support various configuration options including RAM, ROM, FIFOs, large look-up tables, and shift registers as wells as various data widths

(9-bit, 18-bit, etc.) and depths. We refer the reader [56] for further information. This work explores the use of these functional blocks, that is to say the coprocessor approach [57]. Specifically, the register file and instruction memory of coprocessor is implemented on such memory blocks yielding substantial reduction in the slice usage.

3.3.1.1 Overall architecture

The overall architecture of the Hummingbird coprocessor is shown in Figure 4 which consists of a register file, datapath, LFSR module, instruction memory and control unit.

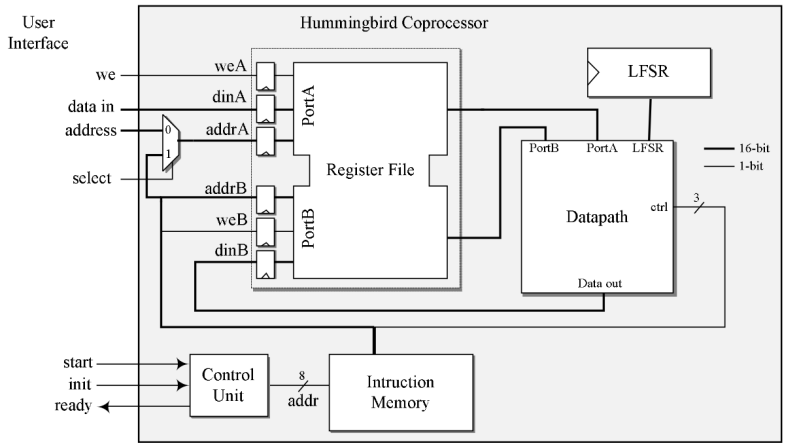


Figure 3.2: Hummingbird coprocessor.

Hummingbird coprocessor performs a modulo 2^{16} addition, XOR, and S-box layer operations required by the algorithm. It accelerates the main system performance by offloading processor intensive tasks. Essential operations are done by the coprocessor rather than implemented on a regular processor. This positively impacts the efficiency. Instructions are first preloaded to the instruction memory. They are then fetched to control the datapath.

3.3.1.2 Register file, instruction memory and control unit

The register file stores 16-bit plaintext, 256-bit secret key, four 16-bit internal state registers RS1, RS2, RS3 and RS4, and other necessary registers such as zero and temporary registers. The internal states and initial vector for LFSR are first determined by the initialization process. They are then updated after the encryption of each block. Note that the same key is usually used for

successive encryption of plaintext blocks. Thus, the coprocessor only needs to load plaintext blocks from input to the register file yielding a very efficient input interface for the coprocessor.

The instruction memory provides necessary control signals for the datapath. It also determines which register is used at each step. Write enable field for port B, web, is used to write the data to the register file from the datapath. The instruction memory is realized via an embedded memory block. Since all instructions have the same format, 14-bit fixed instructions are fetched without the need for decoding. A single 1Kx18 block RAM would suffice to store necessary instructions for initialization and encryption processes.

The control unit is actually implemented as a program counter. When init signal is asserted, the program counter addresses the instruction memory so that the instructions for the initialization process are provided to the datapath. Similarly, if start signal is asserted, the program counter addresses the encryption process instructions. After the completion of the encryption, the ready signal is set; hence, the user can access the ciphertext from the register file via user interface.

3.3.1.3 Datapath of the coprocessor

The goal is to serialize the Hummingbird cryptographic algorithm in the datapath. Figure 3.3 illustrates the proposed datapath for the coprocessor. The overall system is divided into four specific sections, namely stages. Each stage completes a small transaction on the data. Intrinsic properties of the Spartan-3 device are utilized to complete a transaction with a minimal path delay.

The datapath consists of four functional units performing operations required by the Hummingbird cryptographic algorithm. These functional units are realized in four stages and the stages are connected via pipeline flip-flops. Each stage fits into eight slices. All stages except Stage 2 need control signals to determine the data flow path.

The longest path in terms of logic and routing delay occurs in the addition of two 16-bit numbers due to the carry chain. Note that this is an unavoidable operation. To decrease the path delay in one stage, we minimize the use of consecutive logic components by introducing flip flops as pipeline registers.

Stage 0 performs a modulo 2^{16} addition on data provided by the register file or LFSR. The output of this stage feeds the Stage 1 together with the output of the last stage. In Stage 1, control signal selects either one of the sources from the previous stage or from the linear transformation step, Stage 3.

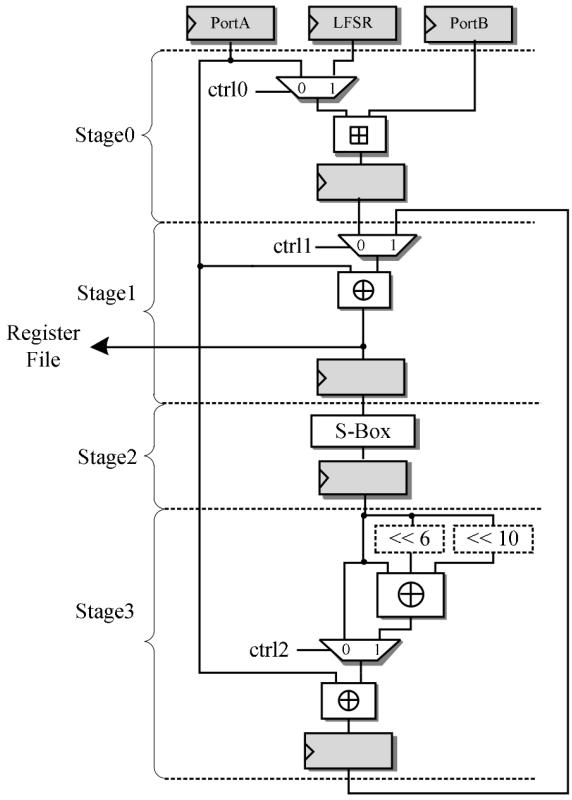


Figure 3.3: Datapath of the Hummingbird Coprocessor.

Resulting data is XORed with the port A of the register file. Stage 2 performs the substitution step on the 16-bit data coming from the previous stage. All four 4-bit S-boxes are realized in LUT function generators. The 16-bit data is passed through from these LUT primitives. The last stage, Stage 3, performs the linear transformation on the incoming data. At this stage, the control signal is necessary to select whether the linear transformation is applied to the output of the substitution step in Stage 2. Note that the rotation of bits at the last stage does not need any component. This can be done only with appropriate wiring.

3.3.1.4 Scheduling of instructions

Instruction level serialism is applied to the Hummingbird coprocessor in order to complete the encryption process with a minimum instruction count.

Figure 3.4 shows the designed instruction scheduling of the encryption process where first row indicates the instruction number and the other rows indicates the active stage.

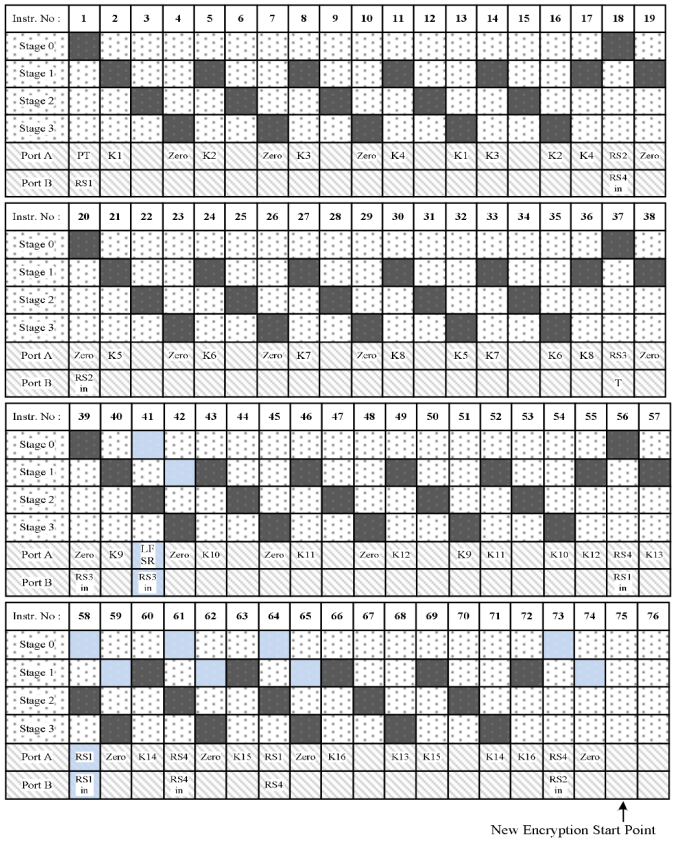


Figure 3.4: Instruction Scheduling.

Generally, each instruction performs an operation on the data. However, a small amount of instructions do not perform any operation on the data. They rather provide necessary data to the next stage. This is the case when loading the internal state from the register file. Loading from the register file impairs the performance of the overall system. Such instructions cause performance degradation since additional instructions are to be stored in the memory.

Our proposed datapath decreases the instruction count for the Hummingbird encryption in two ways. One is to manage the sequence of instructions so that every stage deals with an operation. The other one is to update the internal registers while other operations are being performed. Internal state updating is done via pipelined operation: appropriate instructions are selected to load the input data for the internal state registers without affecting the performed instruction at that time. Such pipelined instructions are colored with blue in Figure 3.4.

In general, each instruction activates only one stage performing one operation. However, a few instructions process two operations at a time. This

Table 3.1: FPGA Implementation Results of the Hummingbird Encryption.

Studies	LUTs	FFs	Area [slices]	Block RAM	Max. Freq. [MHz]	CLK cycles		Throughput [Mbps]	Efficiency [Mbps/Slices]
						Init	Encryption		
This Work	80	80	40	2	260.8	312	75	55	1.38
[14]	473	120	273	0	40.1	20	4	160.4	0.59

happens when updating the internal states. Note that state updating can be done after the encryption process is completed. But, this would increase the total instruction count. Such operations can be combined with the other instructions. The use of inactive stages improves the overall system performance by reducing the instruction count. The similar approach is used in the Hummingbird initialization process. We remark that the initialization takes 312 clock cycles where the encryption takes 75 clock cycles which includes the state updates.

3.3.2 Results

We implemented the Hummingbird coprocessor on Xilinx Spartan-3 device which is low cost FPGA by means of VHDL language. The hardware performance of the Hummingbird cryptographic algorithm is studied in [14] where the lightweight architecture is based on loop unrolling of the inner 16-bit block ciphers. That implementation realizes all operations of the block cipher in one cycle consuming resources notably. The proposed method in this work significantly reduces (273/40) the area in terms of the slice count by introducing coprocessor approach. Table 3.1 summarizes FPGA implementation results of the Hummingbird in both studies. Our proposed method requires lower area in terms of slices. Even though the throughput may seem lower than that of in [14], the overall efficiency is increased substantially (1.38/0.59).

Table 3.2 is given in order to see where our proposed method stands among other FPGA implementations of lightweight cryptographic algorithms. As seen from the table, our work has the smallest area requirement among all and also provides well enough efficiency.

3.3.3 Conclusions

As far as author's knowledge, this work presents the smallest and the most efficient FPGA implementation of the ultra-lightweight cryptographic algorithm Hummingbird thanks to the coprocessor approach. The coprocessor approach

Table 3.2: Performance Comparison of FPGA Implementations of Lightweight Cryptographic Algorithms.

Studies	Algorithm	Key Size	Block Size	FPGA	Area Slice	Frequency [MHz]	Throughput [Mbps]	Efficiency [Mbps/Slices]
This Work	Hummingbird	256	16	Spartan-3 XC3S200	40	260.8	55.64	1.38
[14]	Hummingbird	256	16	Spartan-3 XC3S200	273	40.1	160.4	0.59
[11]	PRESENT	80	64	Spartan-3 XC3S400	176	258	516	2.93
[58]	PRESENT	80	64	Spartan-3E XC3S500	271	–	–	–
[15]	XTEA	128	64	Spartan-3 XC3S50	254	62.6	36	0.14
[59]	AES	128	128	Spartan-3	1800	150	170	0.9
[60]	HIGHT	128	64	Spartan-3 XC3S50	91	163.7	65.48	0.72
[61]	SEA	126	126	Virtex-II XC2V4000	424	145	156	0.368

is enabled due to the fact that FPGAs have dedicated memory blocks. The algorithm is serialized so that each step performs just one operation on the data. The datapath of the Hummingbird coprocessor is implemented in four stages and the instruction count is reduced via pipelining technique.

The comparison of the cost of the coprocessor to that of alternative approaches for implementing Hummingbird cryptographic algorithm in terms of area, throughput and efficiency metrics can be seen from Table 3.2. With the hardware implementation improvement provided by this work, the Hummingbird will continue to be a favorite among the lightweight cryptographic algorithms for resource constrained devices like RFID tags and smart cards.

The enhanced implementation of the Hummingbird coprocessor in this study can be used as an efficient accelerator core in the System on Chip (SoC) designs. The resulted SoC design will accelerate the system performance for the applications such as networking and fast encrypted data communication thanks to the bus based approach provided by the Microblaze processor [62]. Note that the reconfigurable computing [63] provides significant speed-up which is achieved by the Microblaze processor based platform by the Xilinx Spartan-3 FPGAs.

The proposed SoC design with Hummingbird coprocessor, twofold gain is achieved: area and efficiency. This will further decrease the power consumption of the system. With the approach presented, one can achieve high enough performance and the cryptographic functionality at the same time.

3.4 Lightweight Hardware Architectures for XTEA

Tiny Encryption Algorithm (TEA), introduced by David Wheeler and Roger Needham [64], is one of the fastest and most efficient cryptographic algorithms. A new version of TEA, extended TEA (XTEA) [31], addresses a couple of

minor weaknesses found in TEA. XTEA is a 64-bit block cipher based on a Feistel network with a 128-bit key and a suggested 32 rounds. It uses only simple addition, XOR and shifts. The code is lightweight which makes the algorithm an ideal candidate to provide data security services for resource-constrained devices.

There have been various implementations of XTEA addressing specific requirements such as high throughput, compact area, etc [15]. In this study, we present a serial implementation of XTEA, which takes advantage of bit-slice design methodology yielding an ultra-compact design.

Our architecture is similar to the serial AES and Camellia architectures in [65] and [66], respectively. However, both architectures are optimized for ASIC implementation. We furthermore take advantage of slice structures of XILINX FPGAs to come up with a design specifically optimized for embedded security applications. Such applications utilize compact and inexpensive FPGAs, while still managing to reach compact figures making our design also suitable for custom ASICs used in lightweight applications such as RFIDs, sensors nodes.

Even though XTEA was originally proposed for software implementations, its simple design makes it also very suitable for hardware implementations. Hardware implementations of the TEA were first studied in [67, 68] targeting RFID tags. Later, the design space exploration of hardware implementations of XTEA is given in [15].

In this study, we have proposed lightweight hardware architecture for the XTEA cryptographic algorithm. The proposed architecture is well-suited for the applications where low-cost FPGAs are deployed such as FPGA-based tags [69]. These tags use the Spartan-3 XC3S1000 FPGA from Xilinx as a programmable device, which has a simple package footprint, features and reasonable hardware resources so that security enabled RFID systems can be built. Thanks to our lightweight architecture for XTEA, the rest part of the reconfigurable area in FPGA can be used for implementing other functionalities. For example, multiple instances of our architecture can be run in parallel so that higher throughputs can be obtained. The results presented in this section have been published in [70].

3.4.1 The lightweight XTEA

In this section, we propose a lightweight architecture for the XTEA cryptographic algorithm on FPGAs which are composed of configurable logic blocks

(CLB) and a programmable interconnection network. Our design uses the low cost Xilinx FPGA series Spartan-3 [44]. Xilinx divides each CLB into slices and each slice on a Spartan-3 contains two sets of a 4-input look-up table (LUT) followed by a flip-flop. Moreover, slices have dedicated circuitry and connections for fast carry propagation. Hence, the result of additions can be stored within the same slices avoiding any additional wire delay.

Several optimizations and/or enhancements are implemented on the XTEA architecture:

- Since the half-rounds in one cycle of XTEA are very similar, we implement only one half-round and let it run in sequence.
- The 8-bit serialization is applied. Note that using a smaller path reduces the size of the adders and XORs by one fourth. On the other hand, it would require the use of additional multiplexers and increases the number of clock cycles needed for one encryption.

In the design, the 64-bit plaintext is stored in eight 8-bit state registers. At each clock cycle, 8-bit data is processed, the state registers are shifted to the right, and the next 8-bit data is provided to the datapath. This is repeated four times, since the operands in the XTEA cryptographic algorithm are in 32-bit length.

Two important things to be handled in the serialization of XTEA are related to how one can perform the XTEA operations, specifically addition and shift operations, in 8-bit.

- The serialization of the 32-bit addition to 8-bit is handled by storing the carry information for the higher octet addition. The lower octets are processed first and its carry is used in the upper 8-bit addition. After the most significant octet addition, carry information is cleared for the next 32-bit addition operation.
- Shift operations in a 8-bit data bus is handled as follows: Common pattern for the rotations is found in the 8-bit representation of the 64-bit plaintext. The rotations are then accomplished by small multiplexers with selecting their input from special state registers. Remark that the directions of the rotations used in XTEA are different. Further discussion on the serialization is given in Section 3.4.1.3.

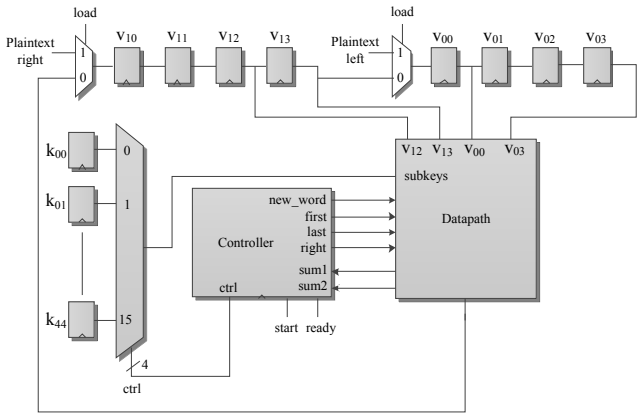


Figure 3.5: XTEA Hardware Architecture

3.4.1.1 Architecture

The overall proposed architecture for the XTEA cryptographic algorithm is shown in figure 3.5 which consists of state and key registers, a few multiplexers, datapath, and a controller. As the names imply, state and key registers store the state and key data respectively. The Datapath performs half-round operations and the controller produces necessary control signals for the datapath.

In the architecture, the 128-bit key can be loaded in parallel or serial; on the other hand, the plaintext is loaded into the state registers in 8-bit chunks. In the following sections, we give details for the datapath and controller, and describe the user interface.

3.4.1.2 Datapath

The XTEA cryptographic algorithm is serialized in the datapath so that fewer logic elements are employed in its hardware implementation. Figure 3.6 illustrates the proposed datapath.

The 8-bit data width is used in the architecture. The XTEA algorithm splits the 64-bit input block (plaintext) into two halves, namely V_0 and V_1. Hence, the addition, XOR, and shift operations are originally described with 32-bit data. We use four 8-bit registers for each half of the data. Due to the introduced serialism, one half-round is completed in four clock cycles. The other half-round is very similar to the first halfround – only subkey selection is changed.

Summation register $(sum3, sum2, ..., sum0)$ handles the key scheduling

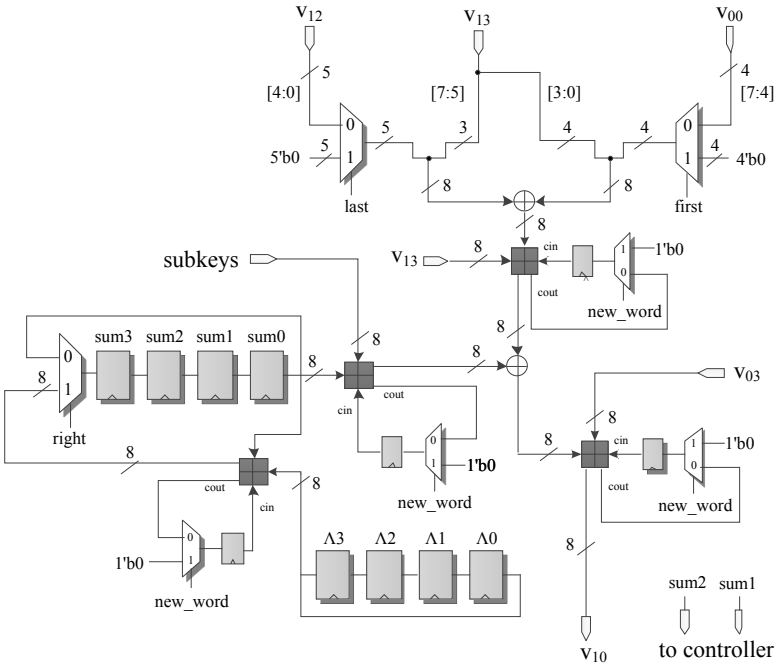


Figure 3.6: Datapath of XTEA Hardware Architecture

where the cumulative sum of the delta is performed. Specifically, sum2 and sum1 parts of the summation register determine which 8-bit key to be used for the half-round in the current cycle.

3.4.1.3 Illustration of serialization

We exploit serial data processing principles to obtain a small-footprint design in terms of the required area. The serialization of the XTEA cryptographic algorithm is illustrated in figure 3.7. First, the lowest octets (V_{13} and V_{03}) of the 32-bit state registers are processed. Next, these lowest 8-bit data are moved to the next registers and the higher octets of the states are processed successively. This transaction takes about four clock cycles; hence, one half-round is completed in four clock cycles. One round (cycle) of the XTEA cryptographic algorithm is consequently completed in eight clock cycles.

3.4.1.4 Controller

The controller unit is designed to provide necessary control signals for the datapath. In the implementation, complex architectures such as Finite State Machine (FSM) or instruction memory are avoided rather a special mechanism is used as a controller that determines the half-rounds and scheduling for the

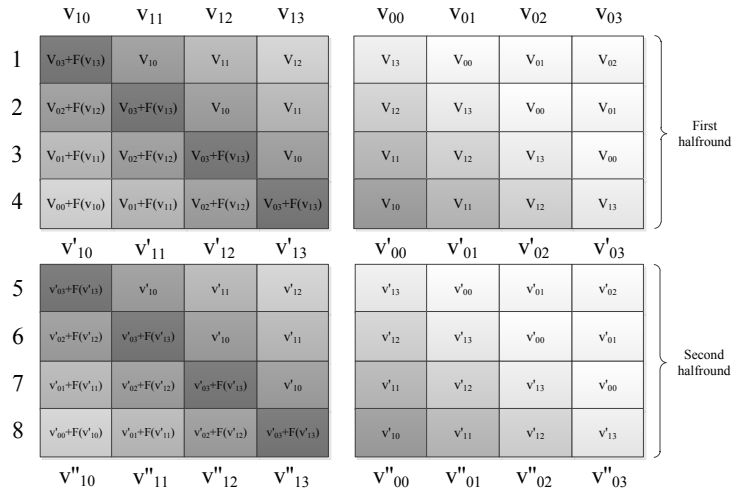


Figure 3.7: Serialization of the XTEA algorithm

datapath. With this simple choice for the controller, significant area reduction is achieved compared to the FSM or instruction memory-based controllers.

3.4.1.5 User interface

The user controls the proposed architecture with start and ready signals shown in Figure 2 which actually control the counter mechanism in the controller module. Initially, the user loads the 64-bit plaintext and 128-bit key to their assigned registers. After the loading, the Lightweight XTEA architecture is run via asserting the start signal. The XTEA encryption is completed in 256 (8 clock cycles per round 32 rounds) clock cycles and the ciphertext become available in the state registers.

3.4.2 Hardware implementation

The proposed design is suitable for both FPGAs and ASIC platforms. We present an FPGA implementation and hand calculated ASIC figures of the Lightweight XTEA in Tables 3.3, 3.4 and 3.5 in the appendix.

3.4.2.1 FPGA implementation

In this section, we compare our performance results with other reported FPGA implementations of the lightweight as well as strong cryptographic algorithms in Table 3.3 and Table 3.4, respectively.

The performance metrics of FPGA often changes with available elements

Table 3.3: Performance Comparison of FPGA Implementations of Lightweight Cryptographic Algorithms

	Algorithm	Key Size	Block Size	FPGA	Area (Slices)	Freq. (MHz)	T/put (Mbps)	Efficiency (Mbps/#Slices)
Lightweight-XTEA	XTEA	128	64	Spartan-3 XC3S50-5	**110**	94.3	23.6	0.214
Kaps [15]	XTEA	128	64	Spartan-3 XC3S50-5	254	62.6	36	0.14
Fan et al. [51]	Hummingbird	256	16	Spartan-3 XC3S200-5	273	40.1	160.4	0.59
San et al. [52]	Hummingbird	256	16	Spartan-3 XC3S200-5	40	260.8	55.64	1.38
Poschmann et al. [50]	PRESENT	80	64	Spartan-3 XC3S400-5	176	258	516	2.93
	PRESENT	128	64		202	254	508	2.51
Guo et al. [58]	PRESENT	80	64	Spartan-3 XC3S500	271	–	–	–
Bulens et al. [71]	AES	128	128	Spartan-3	1800	150	1700	0.9
Yalla et al.l. [60]	PRESENT	128	64	Spartan-3 XC3S50-5	117	113.8	28.46	0.24
	HIGHT	128	64		91	163.7	65.48	0.72
Mace et al.l. [61]	SEA	128	126	Virtex II XC2V4000	424	145	156	0.368

such as BlockRAM (BRAMs), DSP Blocks and slice structures. The presented architecture uses only slices rather than using BRAMs or DSP building blocks. This should be taken into consideration for a fair comparison with other architectures.

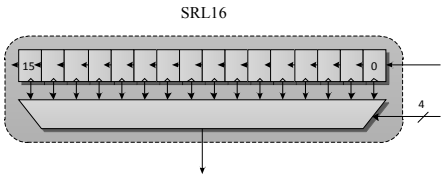


Figure 3.8: Xilinx SRL16 shift register

Note that the XTEA cryptographic algorithm uses a 128-bit key. With our 8-bit data width architecture it would normally require sixteen 8-bit storage units and a 16x1 8-bit multiplexer. On the other hand, one slice contains two registers in Spartan-3 FPGAs. Hence, 128-bit storage is realized by 64 slices and the 16x1 8-bit multiplexer is realized by 32 slices. A total of 96 slices are needed for the key handling. Instead, we propose to use the dedicated shift register resources that exist in Spartan-3 FPGAs, namely SRL16 (a shift register look up table) shown in figure 3.8. Existing LUT units in slices can be configured to form shift register as SRL16 components. So, the reported number of slices in this study take into account the use of SRLs. This embedded component is very well-suited for the key storage. Due to the introduced serialization, parallel access to the key contents is not needed. Hence, four slices are enough to realize the key storage by means of SRLs available in slices. In conclusion, exploiting the dedicated resources available in FPGAs,

required area for the key handling is reduced from 96 to 4 slices. In addition, special care is devoted to realize the datapath and state registers on FPGA to attain the highest frequency possible.

We implemented our lightweight architecture on Xilinx Spartan-3 device which is low cost FPGA family by means of VHDL language. Our design requires only 110 slices and a throughtput of 23.6 Mbps is attained. Table 3.3 shows the performance comparison of FPGA implementations of lightweight cryptographic algorithms. The hardware performance of the XTEA cryptographic algorithm is also studied in [15] where the implementation required 254 slices. Our proposed method needs lower area in terms of slices (110/254). Even though the throughput is lower than that of in [15], the overall efficiency is increased considerably (about %153). Table 3.4 shows the performance comparison of FPGA implementations of strong cryptographic algorithms. Observe that our lightweight XTEA architecture requires the smallest slice count on FPGA given in Table 3.4.

3.4.2.2 ASIC implementation

Our architecture uses simple logic elements which are efficiently implemented in hardware. In this section we give ASIC implementation figures in Table 3.5. A similar study is given in [15] where XTEA cryptographic algorithm required about 3490 GE compared to our design requires roughly 2111 GE which demonstrates that our architecture gives better area utilization in ASIC platforms.

3.4.3 Conclusions

In this study we have proposed a lightweight architecture for the XTEA cryptographic algorithm. Thanks to the introduced serialization, the small-footprint hardware is achieved. Comparison to other reported FPGA implementations of the lightweight and strong cryptographic algorithms indicates that our design

Table 3.4: Performance Comparison of FPGA Implementations of Strong Cryptographic Algorithms

	Algorithm	**Key Size**	**Block Size**	**FPGA**	**Area (Slices)**	**Freq. (MHz)**	**T/put (Mbps)**	**Efficiency (Mbps/#Slices)**
Lightweight-XTEA	XTEA	128	64	Spartan-3 XC3S50-5	**110**	94.3	23.6	0.214
AES 8-bit [16]	AES	128	128	XC2S15-6	264	67	2	0.01
AES [17]	AES	256	16	XC2S30-6	522	60	166	0.32
Camellia-2a [18]	Camellia	80	64	XC2S30-6	399	90.8	13.28	0.03
Camellia [19]	Camellia	128	64	XC4000XL	874	20	122.01	0.14

Table 3.5: Hand Calculated Gate Equivalents for ASIC Implementation

Module	Area (GE)	Percentage (%)
State Registers	379	0.18
Key Registers	853	0.40
Key Scheduling	214	0.10
Datapath	520	0.25
Controller	145	0.07
Total	2111	1

consumes the smallest area among all with a decent efficiency. The proposed hardware implementation achieves a data throughput up to 23.6 Mbps at frequency 94.3 MHz and the performance in terms of throughput to area ratio equal to 0.214. The efficiency ratio is well beyond the ratios which are achieved by some strong cryptographic algorithm implementations shown in Table 3.4.

Our design is well suited for low-cost ubiquitous computing devices that have limited resources and demand lightweight cryptographic hardware. Strong cryptographic algorithms such as AES and Camellia require a great deal of area than resource constrained devices can provide. To this end, XTEA is seen as a powerful candidate to provide security functionality to such devices.

As a future work, the lightweight XTEA architecture can also be used as an efficient hardware core in the System on Chip (SoC) designs where area resources are very constrained and strong security is needed. The resulted SoC design will enable the system security for the applications such as secure communication and secure data storage thanks to the bus based approach provided by the Microblaze processor [62].

4. COMPACT HARDWARE ARCHITECTURES FOR SHA-3 CANDIDATES

This chapter presents novel compact coprocessors for three cryptographic hash functions that are finalists in SHA-3 competition. NIST organized this competition for a new SHA-3 standard. Main criterions for the evaluations of the candidate algorithms are consisting of security, computational efficiency, performance in hardware and its flexibility. To investigate and analyze the computational efficiency and performance in hardware, various compact coprocessors for KECCAK, Grøstl and Skein cryptographic hash functions are developed. Such coprocessors reveals a better understanding of the computational efficiency of these cryptographic hash functions in terms of resource sharing, memory access scheme, scheduling, etc. Finally, resource sharing between the cryptographic hash functions and block/stream ciphers is also studied. Unified hardware architectures which efficiently meet the security requirements of a system to provide confidentiality and integrity are studied for those cryptographic hash algorithms that incorporates a block or stream cipher such as Skein which involves Threefish block cipher.

4.1 Introduction and Motivation

In today's information systems, there are plenty of different types processor architectures being investigated for their efficient designs and high-throughput at a low-cost for a wide variety of applications. These processors have different constraints for varying system requirements. Therefore, the processors have to perform a very broad range of different tasks to do, along with some quality of service regarding where they are used. Security is one of the important computationally intensive tasks. Security is provided with the use of cryptographic algorithms that mainly consist of series of mathematical transformations with a predetermined fashion. It is a significant problem to build computation methods for security algorithms where they provide as much as possible high-throughput in a small footprint. This problem is arisen since the speed of data transmission is increasing while available resources that can be used to build such a high-speed communication device is decreasing.

One of the most significant design trade-offs for hardware architectures

is to find a balance between throughput and area. Although there are several different architectures that some of them aim to increase the throughput or some others reduce the resource utilization, trade-offs between several cryptographic hash functions in terms of the memory, area, and throughput are worth to investigate for embedded systems. In this chapter, several hardware architecture designs are presented with their design trade-offs and related architectural considerations.

The compact hardware implementations are well-suited for a range of applications where satisfying both speed and area requirements is very important such as secure communication in embedded systems. For instance, a time critical application running in an embedded system demand to complete the execution of the application in a certain period of time. In addition to this, security is provided to this system to protect the data in communication between the system and the outside world by a set of computationally intensive cryptographic algorithms. The biggest problem is to provide the security to those systems by satisfying both speed, area and security concerns. Accelerating the computation of cryptographic algorithms to protect network traffic and confidential data within some area constraints is rapidly becoming significant in Today's processor technology. There are some instruction set extensions using special crypto coprocessor to facilitate the execution of cryptographic algorithms (see Intel's Westmere processors).

Cryptographic hash functions are one of the most important of all security primitives that play a fundamental role in networking and communication security including their use for data integrity and message authentication. Hash functions take a message of arbitrary-length as input and produce an output of fixed-length referred to as hash-value, or simply hash [9]. A hash function with a domain D and range R, $h : D \rightarrow R$ is many-to-one, that is, $|D| > |R|$, implying the existence of collisions (pairs of inputs with identical output). The cryptographic hash functions are usually used to produce a condensed representation of the input called the message digest.

In response to the successful cryptanalysis of the MD4, MD5, and SHA-0 hash functions, the National Institute of Standards and Technology (NIST) has decided to develop a new algorithm to augment the Secure Hash Standard (FIPS 180-2). The public SHA-3 competition was announced in November 2007. After two rounds of internal reviews and feedback from the cryptographic community, the NIST selected 5 candidates out of 64 to advance to the final round. The following criteria are adapted for comparing candidate

algorithms [72]:

- Security
- Computational efficiency
- Memory requirements
- Hardware and software suitability
- Simplicity
- Flexibility

If security remains the main criterion, computational efficiency, memory requirement, flexibility, and simplicity are also of great significance.

As seen from the above list, the performance of a candidate algorithm is mostly measured by its software and/or hardware implementation characteristics. We note that software implementations of cryptographic hash algorithms are often too slow to be used in real-time applications including real-time embedded systems, routers, or storage devices. Moreover, due to the use of shared memory by software solutions, they are less secure than their hardware equivalents. To remedy these drawbacks, dedicated hardware architectures are desirable so that fast, secure and high throughput can be achieved.

Our objective is to achieve the best possible trade-off between the required area, the speed of the computation and the security.

Our main aims are:

1. to find parallelism between independent tasks of the algorithms.
2. to decrease the data dependencies and hazards to achieve high parallelism and exploit this parallelism in hardware better.
3. to reuse same resources for different executions of the algorithms.
4. to design pipelined datapath for the independent tasks so that each pipeline units can operate on different set of inputs at the same time.

One of the aims of this thesis to study Keccak, Skein, and Grøstl hash algorithms, which are the proposed hash algorithms that moved to the third round [35, 36, 39].

4.2 Related Work

FPGA platforms provide reconfigurable hardware resources that can be used as an efficient co-processing solution. With the coprocessor approach, certain types of applications can be accelerated by means of tackling computationally intensive tasks on hardware side. Moreover, this approach yields a compact architecture in terms of area. In this study, we propose a compact KECCAK coprocessor with considerably high performance to be used in real-time applications. In the following, we refer two previous studies along this line of research and compare them with our work.

One [73] is a lightweight implementation of the KECCAK hash algorithm for RFID applications developed for ASIC platforms. Specifically, it presents a compact implementation of the permutation KECCAK-f[400] to be used in lightweight cryptographic applications. In that study, the state information is stored in registers increasing slice utilization for its FPGA implementation. On the other hand, our architecture uses Block-RAMs that are already available in FPGAs. Thus, the slice utilization is kept quite low in our case.

The other study [39] is a lightweight hardware implementation of the KECCAK hash algorithm with a low-area coprocessor core. In that architecture, input blocks are transmitted to the low-area coprocessor where the hash computation is performed, relinquishing resources of host system and off-loading CPU's tasks. In this respect, it is similar to our work. We note, however, the fact that our architecture requires approximately 1/3 of the area required for the above mentioned study yielding the most efficient solution for lightweight devices. Table 4.1 gives low-area FPGA implementation results of the KECCAK-256 hash algorithm in both studies. Another distinct feature of our architecture is the special design of the datapath with carefully planned instruction scheduling that allows instructions to use different stages at any time. On the contrary, the low-area coprocessor in [39] lacks this type of parallel instruction processing worse it has bubbles between instructions.

Recall that the KECCAK hash algorithm is based on the sponge construction and uses an iterated permutation as a building block. The sponge construction operates on a state of b-bits and large states are often desirable to attain a certain security level. On the other hand, the large state sizes may be cumbersome in hardware implementations since the operations must be performed using the large state contents. In straightforward implementations, a vast amount of silicon area is needed to handle the large state size

that questions the practicality and/or suitability of the considered algorithm. Thus, a special attention is to be given for handling the large state sizes.

Our proposed architecture solves the above mentioned problem in the following way: The large state is stored in separate RAM units on the FPGAs; hence, there is no need to use slice registers. With this approach, only one or two elements can be accessed at once allowing to reduce the datapath area where the KECCAK operations are performed. Note that accessing one or two elements simultaneously may seem to contradict with the intrinsic properties of the KECCAK hash algorithm; however, we deal with this seemingly problem by introducing advanced techniques.

4.3 The Generic Hardware Architecture

All of our architectures presented in this section consist of a register file organized into w-bit words and implemented by means of dual-ported memory, an ALU, and a control unit (Figure 4.1). The user loads messages, plaintext blocks or ciphertext blocks into a port of the register file. A few control bits allows one to select the algorithm and the desired level of security if the coprocessor supports multiple levels of security. When the coprocessors are hashing or encrypting a message, the intermediate results are always written to the other port of the register file. In the following, we assume that our coprocessors are provided with padded messages. A hardware wrapper interface for Skein, and several other hash functions comprising communication and padding is described in [74].

We follow here the design strategy outlined in [33, 57, 75–78]. The first step consists in defining the minimal instruction set to implement a block cipher and a hash function. Then, an in-depth study of the scheduling allows us to build the ALU and organize the data in the register file. During this step, we

- try to minimize the number of control bits to keep the instruction memory as compact as possible;
- take advantage of FPGA specifics to optimize the slice count;
- identify the available parallelism and pipeline the ALU accordingly.

Eventually, we design the control unit. Generally, the instruction memory is automatically generated by a C program. In order to keep the instruction ROM as compact as possible, for instance, our C program for Skein is able to

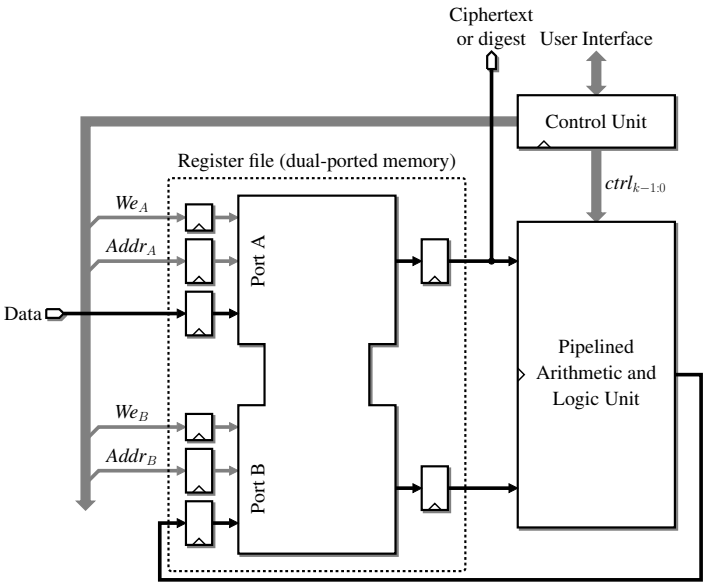


Figure 4.1: General architecture of our coprocessors.

compress the code, and to generate the VHDL description of the decompression unit.

4.4 Compact Hardware Architectures for Keccak Cryptographic Hash Function

In this section, KECCAK hash algorithm, one of the proposed hash algorithms that moved to the final round along with Grøstl [36], Skein [35], JH [79] and BLAKE [80] is studied. KECCAK is a family of hash functions that are based on the sponge construction and uses an iterated permutation from a set of seven permutations as a building block [38]. Security protocols such as IPSec, SSL and VPNs employ many cryptographic primitives. In order to provide data integrity and message authentication in such protocols, cryptographic hash functions are used. With advancement in technology, some of these protocols and applications call for higher throughput. With this purpose in mind, we study to increase the efficiency of this hash function algorithm based on FPGA platform. The results presented in this section have been published in [75, 81].

In particular, we investigate the compact hardware implementation of the KECCAK hash algorithm. The contribution of this work is the use of coprocessor approach [57] in conjunction with custom-designed datapath and instruction format. The coprocessor approach takes advantage of the embedded memory blocks that exists in most of the FPGA platforms [82]. Similarly, the

custom-designed datapath utilizes intrinsic properties of special units placed in the FPGA. With the architecture proposed in this paper, the required area for the implementation of the KECCAK hash algorithm is reduced significantly compared to the previously reported FPGA implementations [39] of the algorithm. Due to the reduction in the area and special solutions employed, the efficiency (throughput/area) of the algorithm is increased considerably. Note that the reconfigurable computing [63] provides significant speed-up which is achieved by the MicroBlaze processor-based platform by the Xilinx FPGAs and the reconfiguration of such coprocessors provides flexible hardware acceleration [62].

In this section, we also employ Hardware/Software codesign methodology and present a few embedded System on Chip (SoC) designs that achieves high operating frequency and throughput together with relatively smaller area than loop unrolled or pipelined architectures for KECCAK hash algorithm. Hardware/Software codesign methodology is used to exploit the benefits of both development methodologies in the design of SoC by using Xilinx embedded platform. The proposed SoC architectures were synthesized using VHDL and implemented on Xilinx Virtex5 device. Performance analysis of proposed SoC designs are presented and reveals a significant achievement. Briefly, in this study, we propose SoC designs which uses the Light-KECCAK coprocessor on Xilinx FPGAs with MicroBlaze soft processor.

4.4.1 Compact Keccak coprocessor

In this work, we use the coprocessor approach which takes advantage of the memory units that already exist on the FPGAs resulting a compact design for the KECCAK hash algorithm. With this approach, the area requirement in the implementation of the algorithm is reduced greatly since slice resources are only used for the datapath and certain control units. Specifically, our architecture uses the Block RAM (BRAM) and slice units embedded within the Xilinx Virtex-5 FPGAs to implement state and summation registers. We note that previous work on the implementation of the KECCAK hash algorithm employ parallel, loop-unrolled or pipelined architectures that use an extensive amount of slices on FPGAs. Another contribution of this study is the introduction of the lane serial implementation of the KECCAK hash function where state registers are accessed via dual-port BRAM units.

4.4.1.1 Keccak coprocessor architecture

We present a compact KECCAK coprocessor that efficiently uses the logic elements that exist on FPGAs, yielding the smallest area FPGA implementation of the KECCAK hash algorithm among other previously reported studies in [39].

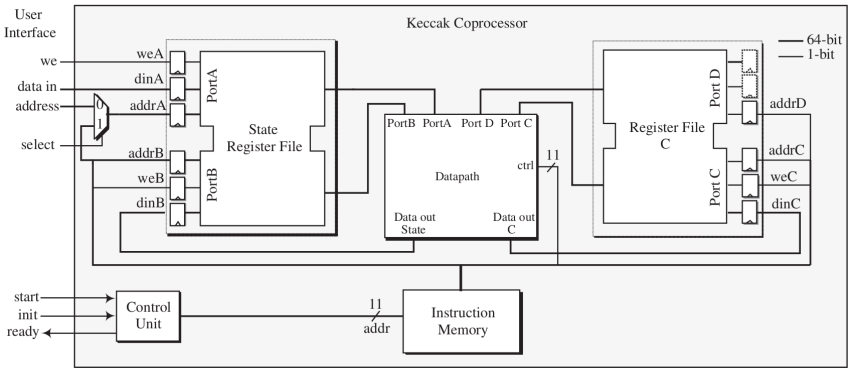


Figure 4.2: KECCAK coprocessor.

The overall architecture of our KECCAK coprocessor is shown in Figure 4.2. It consists of two register files, a datapath, an instruction memory, and a control unit. The datapath is divided into four stages (Stage0 to Stage3) to complete one round of the KECCAK-f permutation.

4.4.1.2 Register files, instruction memory and control unit

Our KECCAK coprocessor has two register files that are realized by two dual port BRAM units available on the FPGA. The dual port configuration is proved to be beneficial in the datapath design since certain step mappings of the KECCAK-f permutation need two lane inputs at once. One exception is the θ (theta) step in the KECCAK-f permutation which needs to access five lanes at once in order to calculate the column sums on the fly. We explain how we handle this case in this section.

The *state register file* as the name implies stores the 1600-bit KECCAK state. Recall that in the sponge construction, the message affects the bitrate part (r-bit) of the state. When initialization signal (init) is asserted, the input state is stored into the state register file via user interface. Following the assertion of the start signal, the KECCAK-f permutation starts to run on the datapath. During this computation, the state register file is used to store intermediate round outputs.

The *register file C* stores intermediate five 64-bit data of column sums

for the theta step. It also stores 24-round constants for the iota step. The register file C provides the column sums at its dual ports, Port C and Port D, respectively.

The *instruction memory* provides necessary control signals for the datapath. It also decides which register is used at each step. Figure 4.3 shows the adapted instruction format for our architecture. Write enable field for port B, *weB*, is used to write the next state to the state register file from the datapath. Write enable field for port C, *weC*, is used to write the column sum result to the register file C from the datapath. The instruction memory is realized by an embedded memory block. Since 33-bit fixed instructions are used, there is no need for decoding, and a double 1Kx36 block RAMs would be sufficient to store all necessary instructions for the KECCAK-f permutation computation.

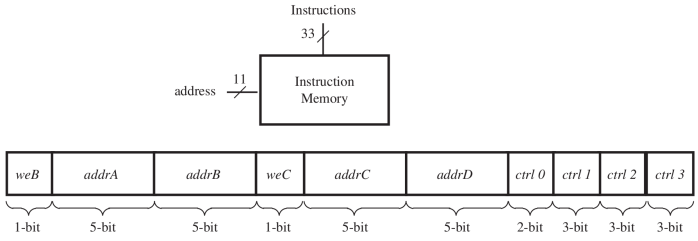


Figure 4.3: Instruction format of KECCAK coprocessor.

The *control unit* is designed as a program counter. After the initialization and the assertion of the start signal, the program counter addresses the hashing process instructions and the computation starts. After the completion of the KECCAK-f permutation, the ready signal is set by the control unit; thus, user can access the message digest from the register file via user interface.

4.4.1.3 The datapath of the Keccak coprocessor

The datapath of the KECCAK coprocessor is devised to serialize the KECCAK-f permutation and is shown in Figure 4.4. We divide the datapath into four functional units, called stage. Each stage assists to the computation of the steps in one round of the KECCAK-f permutation. Recall that one round consists of five steps: θ (handled in Stage0), ρ (Stage1 and Stage2), π (Stage3), χ (Stage3), and ι (Stage0). During the design, special attention is paid not to duplicate functional units in the datapath, yielding minimum hardware area for the implementation of the datapath.

The architecture uses pipeline registers between the stages which serve in many aspects including pipelining, system frequency improvement, and better

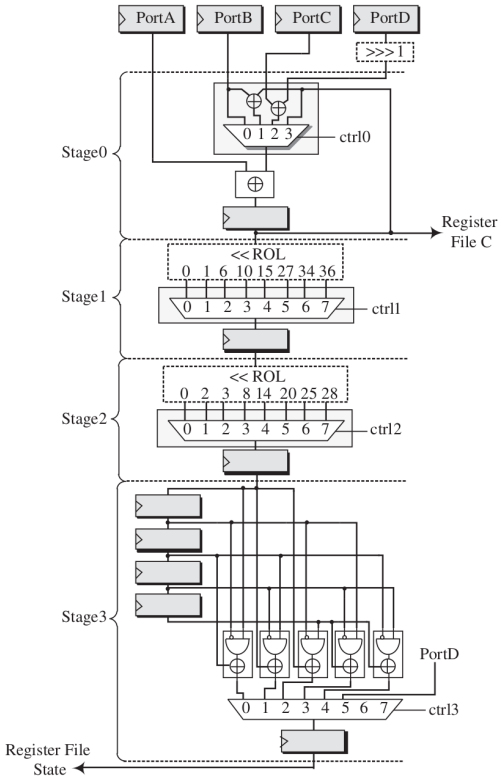


Figure 4.4: The Datapath of KECCAK coprocessor.

utilization of slices on the FPGA. The pipeline registers store intermediate outputs of the stage. At each clock cycle, the data is transferred to the next stage. In this way, multiple instructions can be executed using pipelining technique.

We enable embedded registers at the outputs of the register files to reduce the communication delay between the datapath and memory units. So, minimum clock latency and maximum clock frequency are achieved.

In the following, we summarize the operation of the datapath:

The θ step of the KECCAK-f permutation is handled in Stage0. Recall that the θ step needs to access five lanes at once in order to calculate the column sums. However, our architecture uses dual port configuration allowing only access to one or two elements at the same time. This seemingly problem is remedied by iterative addition of two elements at a time until all elements are used up which takes about 16 instructions (since addition of five elements takes about 3 instructions) as illustrated in Figure 4.5. During the process, resulting sums are written in the register file C. At last, the θ step is completed with addition of two column sums and one lane of the state.

The ρ step of the KECCAK-f permutation is handled in the Stage1 and

Instr No	Port A	Port B	weC	addrC	Stage0 register
1	a00	a10	0	-	-
2	a20	a30	0	-	a00 ^ a10
3	a40	0	0	-	a00 ^ a10 ^ a20 ^ a30
4	a01	a11	1	0	a00 ^ a10 ^ a20 ^ a30 ^ a40
5	a21	a31	0	-	a01 ^ a11
6	a41	0	0	-	a01 ^ a11 ^ a21 ^ a31
7	a02	a12	1	1	a01 ^ a11 ^ a21 ^ a31 ^ a41
⋮	⋮	⋮	⋮	⋮	⋮
15	a44	0	0	-	a04^ a14 ^ a24 ^ a34
16	a44	-	1	4	a04 ^ a14 ^ a24 ^ a34 ^ a44

Figure 4.5: The column sum calculation in the θ step.

Stage2. As seen from Figure 4.4, these stages have similar structures: they both contain 8x1 MUXes. However, Virtex-5 FPGA has only 6-input LUTs. Hence, we use two 6-input LUTs and one F7MUX to obtain 8x1 MUX. Specifically, eight input lines of the MUX are obtained by using two LUTS with four input lines from each LUT. Note that 8x1 MUX requires three control bits, ctrl1 for the MUX in Stage1 and ctrl2 for the MUX in Stage2. Since four input lines of each LUT are used, two least significant bits of the control signal (ctrl1 or ctrl2) are connected to LUTs. The remaining most significant bit of the control signal chooses either LUT via F7MUX. Further, the unused bits of LUTs ([2 bits per LUT per stage $*$ 2 LUTs per stage $*$ 2 stages] = 8-bits) are used to transfer rotated inputs of the Stage0 content.

The ρ step of the Keccak-f permutation could have been handled by one stage or three stages instead of two stages. However, in the first case (one stage implementation) the area required for the implementation would be quite large. Similarly, for the second case (three stage implementation), it would not be suited for the Virtex-5 FPGA family we considered.

The π step of the Keccak-f permutation is just a substitution and does not require any logic for its implementation. It is handled in Stage3 implicitly by an addressing technique yielding no delay in the datapath.

The χ step of the Keccak-f permutation is also handled in Stage3. Note that the χ step needs to access three lanes at once. We employ five registers, AND, XOR, and NOT gates in the implementation of this stage. The pipelining technique allows us to execute multiple instructions concurrently; for example, as the Stage1 performs the rho operation on the current lane, the Stage0 performs the θ operation on the next lane. Owing to this, the datapath is usually run full with multiple instructions.

The ι step of the Keccak-f permutation is handled in Stage0. Specifi-

cally, after the completion of the chi step, it performs an XOR operation with a round constant.

As a final comment, the absorbing phase of the sponge construction is implemented either at the outside of the coprocessor or by adding new instructions to the Stage0.

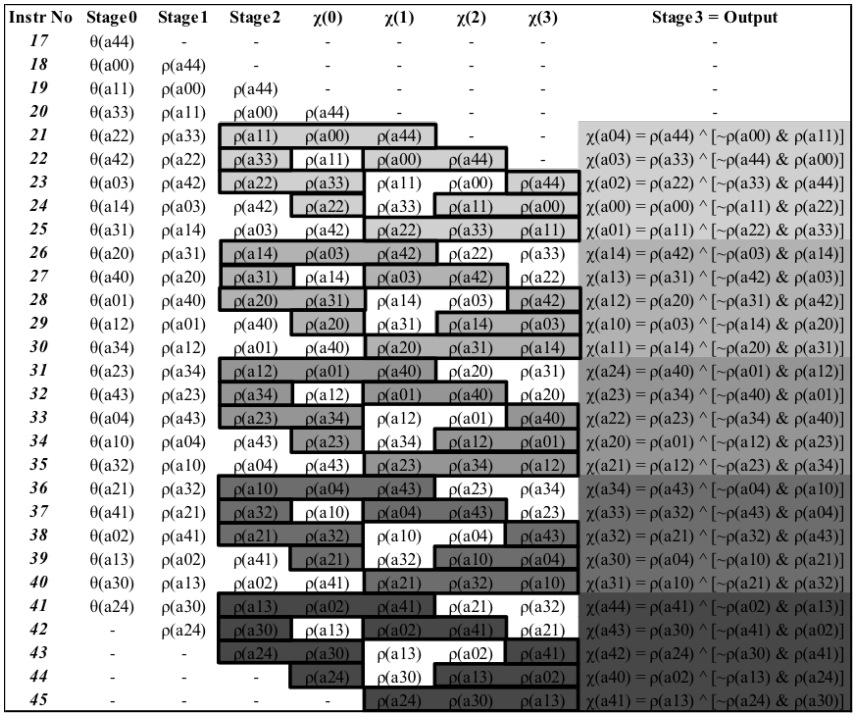
4.4.2 Instruction scheduling

We apply instruction level parallelism to the KECCAK coprocessor in order to increase the efficiency of the overall system. One of the goals in our design is to compute the KECCAK-f permutation with as low instructions as possible. This goal can only be achieved by considering intrinsic properties of the algorithm, several implementation techniques and evaluating them diligently.

Recall that the KECCAK-f permutation has 24 rounds for a 1600-bit state, $l = 6$ and $n_r = 12 + 2l = 24$. The first two rounds are completed with 94 instructions. Next two rounds are completed with 88 instructions. Thus, a total of 1062 $(94+88*22/2)$ instructions are needed to complete the KECCAK-f permutation computation.

In the proposed architecture, one round of the KECCAK-f permutation is completed in 25 clock cycles thanks to the instruction level parallelism technique. Figure 4.6 shows the instruction scheduling for the design. Observe that the datapath actually starts to produce outputs of the first round after 21 instructions. Note also each instruction can activate several stages of the datapath. For example, the instruction 19 activates Stage0, Stage1, and Stage2 at the same time.

In summary, our proposed datapath reduces the number of instructions required for the KECCAK-f permutation computation in two ways. First, the stages can be used in a parallel manner. This parallel operation of the stages is provided by pipeline registers used between stages as follows: at the end of each clock cycle, all computations performed in the current stage are written into the corresponding pipeline register for that stage, and they are then used in succeeding stages at the next clock cycle. Second, the serialization of the KECCAK state is employed which reduces the width of the datapath from 1600-bit state length to 64-bit lane width. We finally remark that data is introduced to the datapath in certain order. This order is determined by the pi and chi steps of KECCAK-f permutation.



Instr No	Stage0	Stage1	Stage2	χ(0)	χ(1)	χ(2)	χ(3)	Stage3 = Output
17	θ(a44)	-	-	-	-	-	-	-
18	θ(a00)	ρ(a44)	-	-	-	-	-	-
19	θ(a11)	ρ(a00)	ρ(a44)	-	-	-	-	-
20	θ(a33)	ρ(a11)	ρ(a00)	ρ(a44)	-	-	-	-
21	θ(a22)	ρ(a33)	ρ(a11)	ρ(a00)	ρ(a44)	-	-	χ(a04) = ρ(a44) ^ [¬ρ(a00) & ρ(a11)]
22	θ(a42)	ρ(a22)	ρ(a33)	ρ(a11)	ρ(a00)	ρ(a44)	-	χ(a03) = ρ(a33) ^ [¬ρ(a44) & ρ(a00)]
23	θ(a03)	ρ(a42)	ρ(a22)	ρ(a33)	ρ(a11)	ρ(a00)	ρ(a44)	χ(a02) = ρ(a22) ^ [¬ρ(a33) & ρ(a44)]
24	θ(a14)	ρ(a03)	ρ(a42)	ρ(a22)	ρ(a33)	ρ(a11)	ρ(a00)	χ(a00) = ρ(a00) ^ [¬ρ(a11) & ρ(a22)]
25	θ(a31)	ρ(a14)	ρ(a03)	ρ(a42)	ρ(a22)	ρ(a33)	ρ(a11)	χ(a01) = ρ(a11) ^ [¬ρ(a22) & ρ(a33)]
26	θ(a20)	ρ(a31)	ρ(a14)	ρ(a03)	ρ(a42)	ρ(a22)	ρ(a33)	χ(a14) = ρ(a42) ^ [¬ρ(a03) & ρ(a14)]
27	θ(a40)	ρ(a20)	ρ(a31)	ρ(a14)	ρ(a03)	ρ(a42)	ρ(a22)	χ(a13) = ρ(a31) ^ [¬ρ(a42) & ρ(a03)]
28	θ(a01)	ρ(a40)	ρ(a20)	ρ(a31)	ρ(a14)	ρ(a03)	ρ(a42)	χ(a12) = ρ(a20) ^ [¬ρ(a31) & ρ(a42)]
29	θ(a12)	ρ(a01)	ρ(a40)	ρ(a20)	ρ(a31)	ρ(a14)	ρ(a03)	χ(a10) = ρ(a03) ^ [¬ρ(a14) & ρ(a20)]
30	θ(a34)	ρ(a12)	ρ(a01)	ρ(a40)	ρ(a20)	ρ(a31)	ρ(a14)	χ(a11) = ρ(a14) ^ [¬ρ(a20) & ρ(a31)]
31	θ(a23)	ρ(a34)	ρ(a12)	ρ(a01)	ρ(a40)	ρ(a20)	ρ(a31)	χ(a24) = ρ(a40) ^ [¬ρ(a01) & ρ(a12)]
32	θ(a43)	ρ(a23)	ρ(a34)	ρ(a12)	ρ(a01)	ρ(a40)	ρ(a20)	χ(a23) = ρ(a34) ^ [¬ρ(a40) & ρ(a01)]
33	θ(a04)	ρ(a43)	ρ(a23)	ρ(a34)	ρ(a12)	ρ(a01)	ρ(a40)	χ(a22) = ρ(a23) ^ [¬ρ(a34) & ρ(a40)]
34	θ(a10)	ρ(a04)	ρ(a43)	ρ(a23)	ρ(a34)	ρ(a12)	ρ(a01)	χ(a20) = ρ(a01) ^ [¬ρ(a12) & ρ(a23)]
35	θ(a32)	ρ(a10)	ρ(a04)	ρ(a43)	ρ(a23)	ρ(a34)	ρ(a12)	χ(a21) = ρ(a12) ^ [¬ρ(a23) & ρ(a34)]
36	θ(a21)	ρ(a32)	ρ(a10)	ρ(a04)	ρ(a43)	ρ(a23)	ρ(a34)	χ(a34) = ρ(a43) ^ [¬ρ(a04) & ρ(a10)]
37	θ(a41)	ρ(a21)	ρ(a32)	ρ(a10)	ρ(a04)	ρ(a43)	ρ(a23)	χ(a33) = ρ(a32) ^ [¬ρ(a43) & ρ(a04)]
38	θ(a02)	ρ(a41)	ρ(a21)	ρ(a32)	ρ(a10)	ρ(a04)	ρ(a43)	χ(a32) = ρ(a21) ^ [¬ρ(a32) & ρ(a43)]
39	θ(a13)	ρ(a02)	ρ(a41)	ρ(a21)	ρ(a32)	ρ(a10)	ρ(a04)	χ(a30) = ρ(a04) ^ [¬ρ(a10) & ρ(a21)]
40	θ(a30)	ρ(a13)	ρ(a02)	ρ(a41)	ρ(a21)	ρ(a32)	ρ(a10)	χ(a31) = ρ(a10) ^ [¬ρ(a21) & ρ(a32)]
41	θ(a24)	ρ(a30)	ρ(a13)	ρ(a02)	ρ(a41)	ρ(a21)	ρ(a32)	χ(a44) = ρ(a41) ^ [¬ρ(a02) & ρ(a13)]
42	-	ρ(a24)	ρ(a30)	ρ(a13)	ρ(a02)	ρ(a41)	ρ(a21)	χ(a43) = ρ(a30) ^ [¬ρ(a41) & ρ(a02)]
43	-	-	ρ(a24)	ρ(a30)	ρ(a13)	ρ(a02)	ρ(a41)	χ(a42) = ρ(a24) ^ [¬ρ(a30) & ρ(a41)]
44	-	-	-	ρ(a24)	ρ(a30)	ρ(a13)	ρ(a02)	χ(a40) = ρ(a02) ^ [¬ρ(a13) & ρ(a24)]
45	-	-	-	-	ρ(a24)	ρ(a30)	ρ(a13)	χ(a41) = ρ(a13) ^ [¬ρ(a24) & ρ(a30)]

Figure 4.6: Instruction scheduling.

4.4.3 Results and comparisons

In this study, we design and implement a small-footprint Keccak coprocessor on Xilinx Virtex-5 FPGA. We call this architecture as Light-Keccak hereinafter. The hardware performance of the Keccak hash algorithm was also studied in [38]; their design was based on finite state machine with a datapath. In [38], it is reported that the computation of the Keccak-f permutation takes 5160 clock cycles whereas in this study the same computation takes only 1062 clock cycles yielding an improvement of (5160/1062). Moreover, the proposed architecture in this work, Light-Keccak, reduces the area required for the hardware implementation significantly (448/151) thanks to the coprocessor approach. We compare the two architectures (Light-Keccak and [38]) implementing Keccak-256 hash algorithm in Table 4.1. As seen from the table, the efficiency of the algorithm is increased substantially compared to the [38], (3.32/0.117). Note that the increase in efficiency is not only due to the improvements on the computation time and the area but also due to the higher operating frequency achieved with our architecture. Generic architectural platform for hardware implementations of all variants of BLAKE hash family functions is presented in [83]. Performance figures given in [83] indi-

cates that presented design is well suited when high throughput is needed. Multi module hashing system for BLAKE hash functions family given in [84] is also aimed to provide high performance figures. A key difference compared to our study is that our proposed architecture is designed to find best trade off between throughput and area. FPGA implementation of fully autonomous architecture for KECCAK cryptographic hash algorithm is proposed by [85]. Due to fully autonomous architecture, the design requires more area resources than our proposed coprocessor. Table 4.2 and 4.3 include the performance figures given in [38, 57, 80, 83, 86, 87] for KECCAK, BLAKE, Skein and Grøstl cryptographic hash algorithms. Tables show where our proposed design for KECCAK stands among other FPGA implementations of the other SHA-3 finalists (except JH) with message digests 256 and 512 bits in Table 4.2 and 4.3, respectively. As seen from the tables, the Light-KECCAK performs quite well among others. Observe that the area stays the same for both message digest sizes in our architecture.

We give several remarks about our architecture in the following:

- The architecture studied in this work uses 64-bit datapath width. However, the KECCAK coprocessor given here can also operate on the 32-bit datawidth to perform KECCAK-$f[800]$ without the need for change in any functional stages. Specifically, only the datapath width should be decreased from 64-bit to the 32-bit. Instructions and other parts in the coprocessor remains the same.

- In the implementation of our architecture, we use memory blocks that already exists on the FPGA. These blocks are primarily used to store the state, round constants, and control instructions.

- One may think that the use of pipeline registers in the implementation should step up the required area. However, in our design, we utilize registers that are already available in slices as pipeline registers rather

Table 4.1: Low-area FPGA implementation results of the KECCAK [r = 1024; c = 576].

Studies	LUTs	FFs	Area [slices]	Memory Blocks	Max. Freq. [MHz]	CLK cycles	Throughput [Mbps]	Efficiency [Mbps/Slices]
Light-KECCAK	150	135	151	3	520	1062	501.4	3.32
KECCAK Team (Bertoni *et al.*) [38]	–	244	448	0	265	5160	52.5	0.12

Table 4.2: Performance comparison of FPGA implementations of SHA-3 Finalists 256.

	Algorithm	Block Size	FPGA	Area [slices]	Memory Blocks	Freq. [MHz]	Throughput [Mbps]
Light-KECCAK	KECCAK[r = 1024; c = 576]	1024	xc5vlx50-2	151	3	520	501
KECCAK Team (Bertoni *et al.*) [38]	KECCAK[r = 1024; c = 576]	1024	xc5vlx50-2	448	–	265	52.5
Kitsos *et al.* [85]	KECCAK[r = 1024; c = 576]	1024	–	4745	–	215	11900
Beuchat *et al.* [57]	BLAKE-32	512	xc5vlx50-2	56	2	372	225
Aumasson *et al.* [80]	BLAKE-32	512	xc5vlx110	390	–	91	412
Sklavos *et al.* [83]	BLAKE-32	512	Virtex	3101	–	50	1280
Long [86]	Skein-256	512	xc5vlx50-3	≈ 1000	–	115	408.7
Grøstl Team (Gauravaram *et al.*) [87]	Grøstl-256	512	xc2vp40-7	3000–4000	–	75–125	400

Table 4.3: Performance comparison of FPGA implementations of SHA-3 Finalists 512.

	Algorithm	Block Size	FPGA	Area [slices]	Memory Blocks	Freq. [MHz]	Throughput [Mbps]
Light-KECCAK	KECCAK[r = 512; c = 1088]	512	xc5vlx50-2	151	3	520	251
KECCAK Team (Bertoni *et al.*) [38]	KECCAK[r = 512; c = 1088]	512	xc5vlx50-2	448	–	265	≈ 26
Beuchat *et al.* [57]	BLAKE-64	1024	xc5vlx50-2	108	3	358	314
Aumasson *et al.* [80]	BLAKE-64	1024	xc5vlx110	939	–	59	468
Sklavos *et al.* [83]	BLAKE-64	1024	Virtex	11800	–	27	987
Long [86]	Skein-512	512	xc5vlx50-3	≈ 1877	–	115	817.4
Grøstl Team (Gauravaram *et al.*) [87]	Grøstl-512	1024	xc2vp40-7	6000–8000	–	35–60	300

than employing new ones. Therefore, we are able to reach nearly the maximum frequency that Virtex-5 FPGA can operate.

4.4.4 Efficient SoC design for acceleration of message authentication and data Integrity on FPGAs

Authentication protocols based on cryptographic hash algorithms are extensively used in today's networking and streaming applications. These protocols authenticate the content of a packet to be transmitted as follows: At the transmitter, an integrity check value for a packet is computed using a cryptographic hash algorithm, this value is then added into a specific authentication header with payload, and transmitted over a channel. At the receiver, the integrity check value is computed for a received packet using the same cryptographic hash algorithm. If none of the fields differ, it is assumed that the content of a packet is not altered and/or in error. This provides a mechanism to verify the integrity of a packet transmitted over a channel. However, it does not provide

confidentiality since there is no encryption involved. Thus, the cryptographic hash algorithm provides authentication but not the privacy. This work focuses on the acceleration of data integrity and message authentication services using the KECCAK hash algorithm.

We adapt Hardware/Software codesign approach in order to exploit the advantages of both methodologies. The design rationale of the proposed architecture is to accelerate the performance of computational intensive tasks used by the KECCAK cryptographic hash algorithm. We use specially designed KECCAK coprocessor that achieves high throughputs in a small area. Owing to the serialization exploited in the KECCAK coprocessor, the required area for the design is relatively smaller than the ones presented in the literature [38, 88]. Furthermore, the low latency presented by the coprocessor empowers to operate in higher frequencies [46].

In this study, we attach the KECCAK coprocessor to MicroBlaze processor via both the Processor Local Bus (PLB) and the Fast Simplex Link (FSL) bus. The latter connection performs relatively better than that of the former bus attachment, since the FSL directly links the KECCAK coprocessor to the execution stage of the MicroBlaze processor. The benefit of a microprocessor based system together with a custom hardware is realized via embedded system approach by Xilinx FPGAs. Results show that a substantial gain in the performance of computational intensive tasks used by the cryptographic hash algorithm is achieved. Moreover, we emphasize the fact that this performance improvement is obtained without significantly increasing the required area (silicon gate count) of the system.

4.4.4.1 System-on-chip design

With the rapid improvement in VLSI technology, the number of components that can be placed on a single chip has been continuously rising. This leads various computationally intensive applications to be implemented as a SoC designs. A typical SoC consists of processors, peripherals, or application specific coprocessors. In this work, we propose a SoC design that accelerates the KECCAK cryptographic hash algorithm. Note that if a higher security level is desired, then more computational power is needed since strong cryptographic algorithms should be implemented in the system. In pure software systems, high level languages such as C ease the integration of the security primitives. We use the MicroBlaze soft processor in order to include such a level of abstraction. This processor allows performing computational tasks required by the

cryptographic algorithm in the hardware domain linked by a custom peripheral to the processor. The proposed work employs Hardware/Software codesign methodology harnessing benefits of both development methodologies.

Message authentication and data integrity services are given in two parts: The software part is executed on the processor and handles control tasks such as message communication. The hardware part is implemented on the coprocessor and aims to accelerate the KECCAK cryptographic hash algorithm. In the design, we transform the slowest part of the system which is computationally intensive cryptographic KECCAK hash algorithm into the hardware domain as to accelerate the system. The addition of a custom hardware to a processor will consume FPGA resources; however, this often will not be a problem for systems since surplus resources are available in FPGAs. Hence, the careful use of these spare resources provides a powerful, simple, and most importantly cost-free solution for FPGA based systems.

Data transfer overhead has a crucial impact on the execution time even when low latency MicroBlaze buses, FSL or PLB buses, are used. Nevertheless, still very high speed-ups are achieved in the proposed work. Figure 4.7 illustrates a general MicroBlaze based embedded system and its available bus for communication with the outside world.

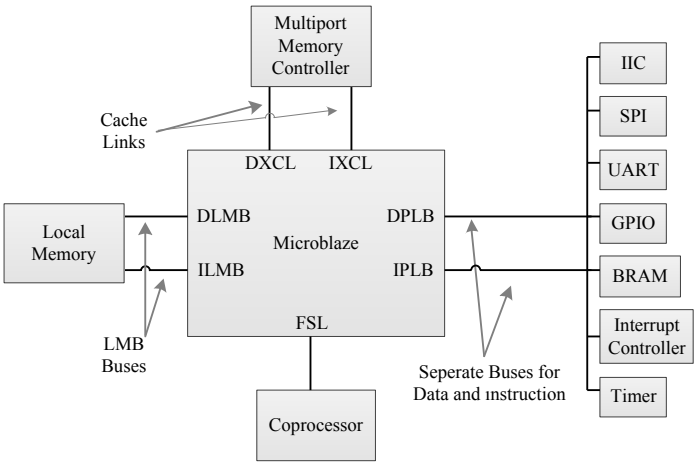


Figure 4.7: A general MicroBlaze based embedded system and available buses.

4.4.4.2 Communication architecture

Figure 4.8 and 4.9 show the two different communication mechanisms between the KECCAK coprocessor and the MicroBlaze via the PLB and FSL buses, respectively. The details of these connections are given in Section 2.2.4.1.

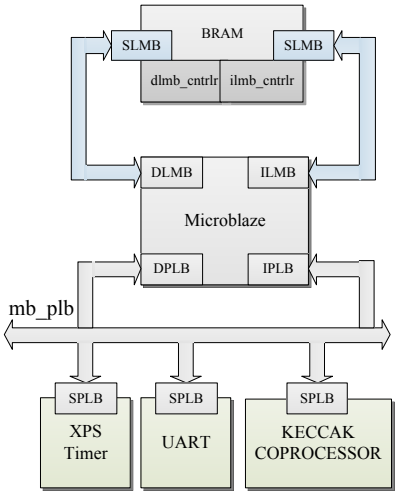


Figure 4.8: PLB connection for the KECCAK coprocessor in MicroBlaze based embedded system (KECCAK PLB SoC).

4.4.4.3 Performance results

Several SoC architectures are experimented for different SoC configurations. Table 4.4 lists the configuration parameters for the KECCAK Software, KECCAK PLB, and KECCAK FSL architectures.

KECCAK Software is an architecture without the KECCAK coprocessor, that is, the KECCAK hash algorithm is realized in a pure software domain. The C code for the KECCAK hash algorithm is written with the same configuration parameters as its corresponding hardware cores.

KECCAK FSL and KECCAK PLB architectures have the KECCAK coprocessor as a peripheral unit in the MicroBlaze based embedded system. In KECCAK PLB, the coprocessor is connected to the MicroBlaze via the PLB bus (see Figure 4.8) whereas, in KECCAK FSL, this connection is made via the FSL bus (see Figure 4.9).

Several sets of measurements are taken during experiments revealing the

Table 4.4: The configuration details of the three different SoC Architectures.

SoC Architecture	Keccak coprocessor	Barel shifter	DDR2 sdram	Area [slices]	BRAM blocks
KECCAK Software	–	✓	✓	5127 (69120)	19 (148)
KECCAK PLB	✓	–	✓	5651 (69120)	21 (148)
KECCAK FSL	✓	–	✓	5837 (69120)	25 (148)

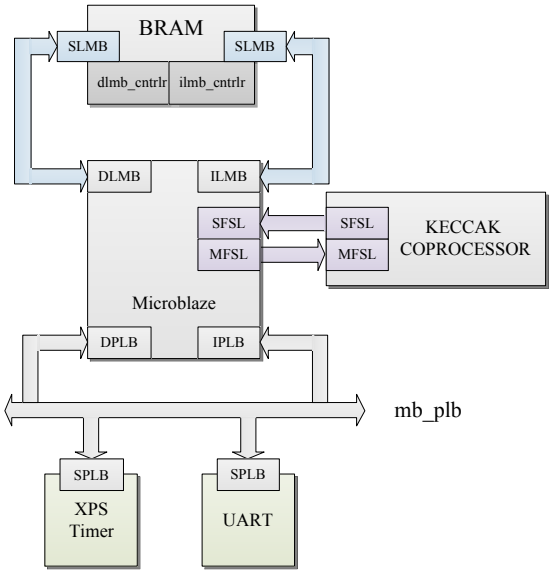


Figure 4.9: FSL connection for the KECCAK coprocessor in MicroBlaze based embedded system (KECCAK FSL SoC).

benefits of using hardware based KECCAK hash algorithm core compared to the software based realization of the KECCAK hash algorithm in terms of execution time. Figure 4.10 shows execution times of the three different SoC architectures for different cache configurations without the compiler optimization where the vertical axis has a logarithmic scale, ICache and DCache denote the instruction and data caches, respectively. Note that different cache configurations affect the system performance. However, for all cases, the KECCAK Software architecture has the highest execution time to compute hash function among others.

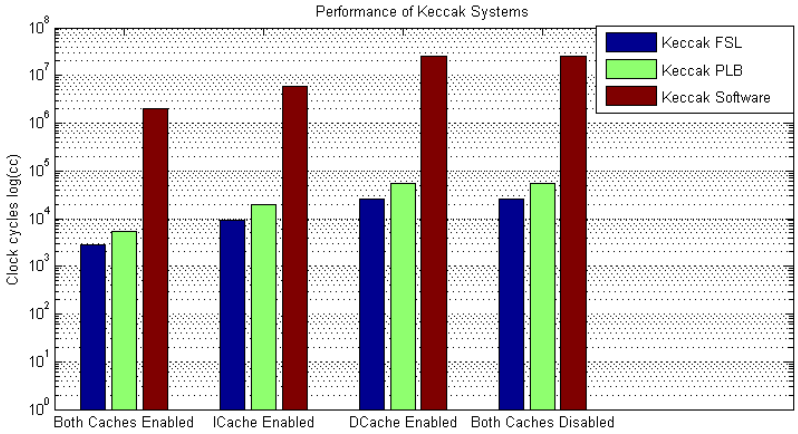


Figure 4.10: Performance results of SoC architectures without the compiler optimization.

Execution times of the three different SoC architectures for different cache configurations under the best compiler optimization are given in Figure 4.11 where the vertical axis has a logarithmic scale. Again for all cases, the KECCAK Software architecture requires the highest execution time to perform the hash operation.

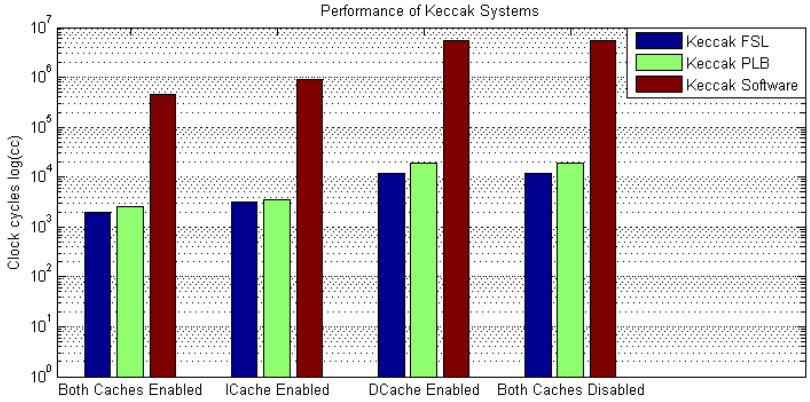


Figure 4.11: Performance results of SoC architectures under compiler optimization.

Figure 4.12 shows the speedups obtained by using the hardware based KECCAK hash algorithm cores namely KECCAK FSL and KECCAK PLB architectures with respect to the KECCAK Software. As seen from the figure, both architectures have significant performance improvement over to pure software counterpart. KECCAK FSL achieves better speedup values since the communication time needed for data transfer is smaller than that of the KECCAK PLB architecture.

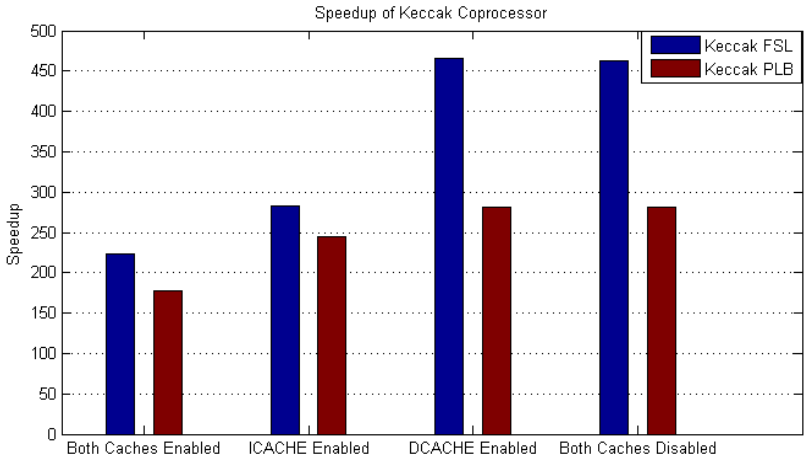


Figure 4.12: Speedups of KECCAK FSL and KECCAK PLB with respect to KECCAK Software.

4.4.5 Conclusion

In this study, we propose an efficient hardware architecture for the Keccak hash algorithm. Owing to the intrinsic properties of the algorithm, the serialization is possible which resulted the reduction in the area. With the proposed architecture, the area needed by the algorithm is reduced and the efficiency is increased significantly on FPGAs. Our architecture also achieves high enough performance for real-time applications. The latency is decreased greatly compared to the one in [39] by means of using advanced digital design techniques such as pipelining and efficient instruction scheduling for special datapath. Multiple instructions are executed in parallel since the functional units are staged via registers thanks to the serialization.

This kind of coprocessor is also suitable for smart cards or wireless sensor networks where area is particularly important since it occupies a low-area in FPGA. With its high throughput per area efficiency rate, our Keccak coprocessor validates to be a best candidate for low-cost devices. In general, there is a trade-off between the security and performance. However, with the approach presented, one can achive high enough performance and the security at the same time. Although our architecture is built via coprocessor approach, it can also be used stand-alone via user interface.

System performance in a pure software design is accelerated by using dedicated hardware. In this study, we used Xilinx embedded system platform utilizes to accelerate cryptographic hashing algorithms in embedded domain. MicroBlaze processors of Xilinx embedded platform allow us to connect the custom hardware via some peripheral connections. Our Keccak Coprocessor is connected via PLB and FSL buses to the MicroBlaze processor. The experiments show that the speed-up provided by the proposed embedded SoC system is very high compared to the pure software design without so much increase in the area. The SoC architectures are able to accelerate the system performance for the applications such as networking and fast authenticated data communication thanks to the bus based approach provided by the MicroBlaze processor [62].

As future work, the Keccak coprocessor attached to PCI-e interface or another high speed communication channel can provide fast data integrity and authentication services to the host CPU. The proposed architecture can also be implemented on different platforms such as ASIC to compare its performance with other reported ASIC implementations of Keccak hash function.

4.5 Compact Hardware Architectures for Skein Cryptographic Hash Function

In this section, from a hardware design perspective, computational efficiency of SHA-3 finalist Skein [35] cryptographic hash algorithm is studied and design details of a novel hardware implementation of the Skein is presented. The SHA-3 finalist Skein is built from the tweakable Threefish block cipher, defined with a 256-, 512-, and 1024-bit block size. A low-area coprocessor for Threefish is designed and it is described how to implement Skein on this architecture. We harness the intrinsic parallelism of Threefish to design a pipelined ALU and interleave several tasks in order to achieve a tight scheduling. From our point of view, the main advantage of Skein over other SHA-3 finalists is that the same coprocessor allows one to encrypt or hash a message. The results presented in this section have been published in [77, 78].

As emphasized by Kerckhof *et al.*, "fully unrolled and pipelined architectures may sometimes hide a part of the algorithms' complexity that is better revealed in compact implementations" [89]. In order to have a deeper understanding of the computational efficiency of Skein (resource sharing, memory access scheme, scheduling, etc.), we aim to design a low-area coprocessor on FPGA. Furthermore, such an implementation is valuable for constrained environments, where some security protocols mainly rely on cryptographic hash functions (see for instance [90]). The key element of our approach is to take advantage of the parallelism of the algorithms to deeply pipeline our Arithmetic and Logic Unit (ALU), and to avoid data dependencies by interleaving independent tasks. The main contribution of this study is a lightweight implementation of Threefish (Section 4.5.3). Then, we describe how to implement Skein on our coprocessor (Section 4.5.3). We have prototyped our architecture on a Xilinx Virtex-6 device and discussed our results in Section 5.4.3.2.

4.5.1 The Threefish block cipher

The design philosophy of Threefish is that "a larger number of simple rounds is more secure than fewer complex rounds" [35]. The key schedule can be computed in a few clock cycles, which is an important consideration in order to build a compression function from a block cipher.

The key schedule generates the subkeys from a block cipher key $K = (k_0, k_1, \ldots, k_{N_w-1})$ and a 128-bit tweak $T = (t_0, t_1)$. K and T are extended with

one parity word (Algorithm 5, lines 1 and 2). Each subkey is a combination of N_w words of the extended key, two words of the extended tweak, and a counter s (Algorithm 5, lines 5 to 9). Note that the extended key and the extended tweak are rotated by one word position between two consecutive subkeys.

Algorithm 5 Key schedule of Threefish.

Require: A block cipher key $K = (k_0, k_1, \ldots, k_{N_w-1})$; a tweak $T = (t_0, t_1)$; the constant C_{240} = `1BD11BDAA9FC1A22`.

Ensure: $N_r/4 + 1$ subkeys $k_{s,0}$, $k_{s,1}$, ..., k_{s,N_w-1}, where $0 \leq s \leq N_r/4$.

1. $k_{N_w} \leftarrow C_{240} \oplus \bigoplus_{i=0}^{N_w-1} k_i$;
2. $t_2 \leftarrow t \oplus t_1$;
3. **for** $s \leftarrow 0$ **to** $N_r/4$ **do**
4. **for** $i \leftarrow 0$ **to** $N_w - 4$ **do**
5. $k_{s,i} \leftarrow k_{(s+i) \bmod (N_w+1)}$;
6. **end for**
7. $k_{s,N_w-3} \leftarrow k_{(s+N_w-3) \bmod (N_w+1)} \boxplus t_{s \bmod 3}$;
8. $k_{s,N_w-2} \leftarrow k_{(s+N_w-2) \bmod (N_w+1)} \boxplus t_{(s+1) \bmod 3}$;
9. $k_{s,N_w-1} \leftarrow k_{(s+N_w-1) \bmod (N_w+1)} \boxplus s$;
10. **end for**
11. **return** $k_{s,0}$, $k_{s,1}$, ..., k_{s,N_w-1}, where $0 \leq s \leq N_r/4$;

Table 4.5: Permutations used by the Skein functions (reprinted from [35]).

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(i)$	$N_w = 4$	0	3	2	1												
	$N_w = 8$	2	1	4	7	6	5	0	3								
	$N_w = 16$	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

A series of N_r rounds (Figure 4.13 and Algorithm 6, lines 4 to 19) and a final subkey addition (Algorithm 6, line 21) are applied to produce the ciphertext. The core of a round is the simple non-linear mixing function $\text{Mix}_{d,j}$ (Algorithm 6, lines 13 and 14). It consists of an addition, a rotation by a constant $R_{d \bmod 8,j}$ (repeated every eight rounds and defined in [35, Table 4]), and a bitwise exclusive OR. A word permutation $\pi(i)$ (defined in Table 4.5) is then applied to obtain the output of the round (Algorithm 6, line 17). Furthermore, a subkey is injected every four rounds (Algorithm 6, line 7).

Figure 4.14 describes a decryption round of Threefish-256. It consists of the inverse word permutation followed by the inverse MIX functions. Since subtraction is the inverse of addition operation, additions are substituted by subtractions. It is also changed where all these operations are applied (see

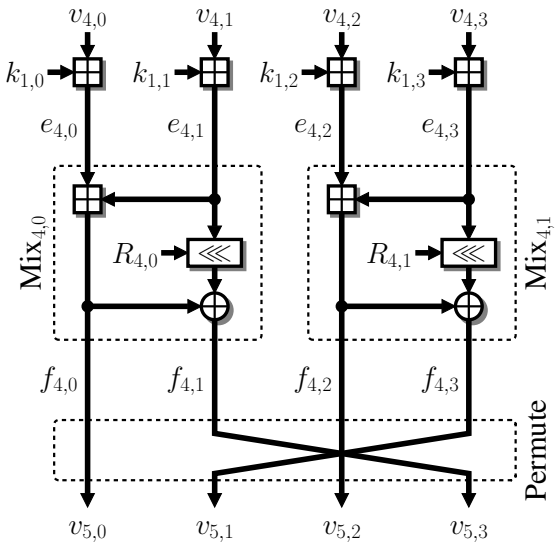


Figure 4.13: One of the 72 encryption rounds of Threefish-256.

Figure 4.14). Note that subkeys are injected in reverse order. These subkeys are again injected every four rounds.

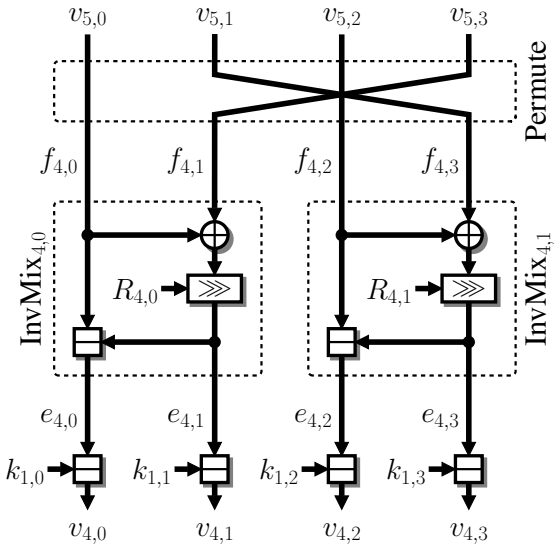


Figure 4.14: One of the 72 decryption rounds of Threefish-256.

4.5.2 The Skein family of hash functions

In this study, the normal hashing mode is considered and refer the reader to [35] for a description of the other modes of operation including Skein-MAC and tree hashing with Skein. Skein is built on three invocations of UBI. These three UBI invocations allow one to compute the digest of a message

Algorithm 6 Encryption with the Threefish block cipher.

Require: A plaintext block $P = (p_0, p_1, \ldots, p_{N_w-1})$; $N_r/4 + 1$ subkeys $k_{s,0}$, $k_{s,1}$, $\ldots$, k_{s,N_w-1}, where $0 \le s \le N_r/4$; $4N_w$ rotation constants $R_{i,j}$, where $0 \le i \le 7$ and $0 \le j \le N_w/2$.

Ensure: A ciphertext block $C = (c_0, c_1, \ldots, c_{N_w-1})$.

1. **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
2. $\quad v_{0,i} \leftarrow p_i$;
3. **end for**
4. **for** $d \leftarrow 0$ **to** $N_r - 1$ **do**
5. $\quad$ **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
6. $\quad\quad$ **if** $d \bmod 4 = 0$ **then**
7. $\quad\quad\quad e_{d,i} \leftarrow v_{d,i} \boxplus k_{d/4,i}$; (Key injection)
8. $\quad\quad$ **else**
9. $\quad\quad\quad e_{d,i} \leftarrow v_{d,i}$; (Rename)
10. $\quad\quad$ **end if**
11. $\quad$ **end for**
12. $\quad$ **for** $j \leftarrow 0$ **to** $N_w/2 - 1$ **do**
13. $\quad\quad f_{d,2j} \leftarrow e_{d,2j} \boxplus e_{d,2j+1}$; ($\text{Mix}_{d,j}$)
14. $\quad\quad f_{d,2j+1} \leftarrow f_{d,2j} \oplus (e_{d,2j+1} \lll R_{d \bmod 8,j})$;
15. $\quad$ **end for**
16. $\quad$ **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
17. $\quad\quad v_{d+1,i} \leftarrow f_{d,\pi(i)}$; (Permute)
18. $\quad$ **end for**
19. **end for**
20. **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
21. $\quad c_i \leftarrow v_{N_r,i} \boxplus k_{N_r/4,i}$; (Key injection)
22. **end for**
23. **return** $C = (c_0, c_1, \ldots, c_{N_w-1})$;

$M = (M_0, M_1, \ldots, M_{k-1})$ of up to $2^{99} - 8$ bits, where each part of the message is a block of N_w 64-bit word. This process is explained with an example: how to hash a message of 170 bytes $M = (M_0, M_1, M_2)$ with Skein-512-512 (Figure 4.15):

1. Configuration block. Define a 32-byte configuration string C that contains the length of the digest size (in bits), a schema identifier, and a version number [35, Table 7]. In the simple hashing mode, compute the N_b-byte block G_0:

$$G_0 \leftarrow \text{UBI}(0, C, T_{\text{cfg}} 2^{120}).$$

   Note that G_0 only depends on the digest size and can easily be precomputed (see [35, Appendix B]).

2. Message. Note that M_0 and M_1 contain 64 bytes of data each, and M_2

is the padded final block with 42 bytes of data. The tweak encodes whether processing block is the first or last block of M, and the number of bytes processed so far. In this example, the message is then processed as follows:

$$G_1 \leftarrow \mathrm{UBI}(G_0, M, T_{\mathrm{msg}} 2^{120}).$$

Note that it requires three calls to Threefish-512 for this example.

3. Output transform. A third call to UBI is required to achieve hashing-appropriate randomness:

$$H \leftarrow \mathrm{UBI}(G_1, 0, T_{\mathrm{out}} 2^{120}).$$

This transform allows one to produce arbitrary digest sizes (up to 2^{64} bits). If a single output block H is not enough, one can use Threefish in counter mode to produce the digest.

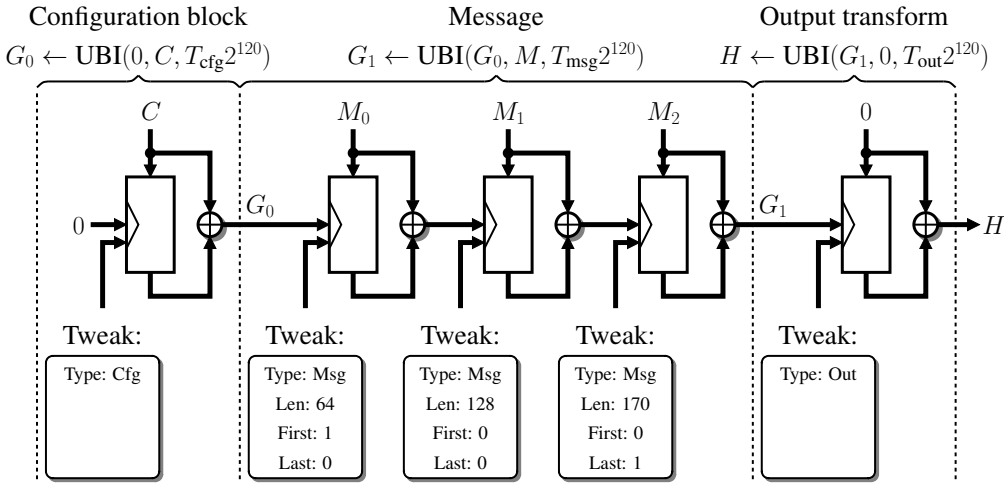


Figure 4.15: Processing a 3-block message using Skein-512-512 in normal hashing mode.

4.5.3 Proposed hardware architecture

A low-area coprocessor for Threefish is first presented with the design methods which efficiently exploit the intrinsic properties of the algorithm. Then, the hardware architecture proposed for the Skein cryptographic hash function is mainly constructed on this Threefish compact coprocessor.

This short description of Threefish gives us the first hints on designing a dedicated coprocessor (Figure 4.16). Our architecture consists of a register file

implemented by means of dual-ported memory, an ALU, and a control unit. The register file is organized into 64-bit words, and stores a plaintext block, an internal state ($e_{d,i}$, where $0 \leq i \leq N_w - 1$), an extended block cipher key, an extended tweak, the constant C_{240}, and all possible values of s involved in the key schedule. Thanks to this approach, the word permutation $\pi(i)$ and the word rotation of the key schedule are conveniently implemented by addressing the register file accordingly. Since the round constants repeat every eight rounds (Algorithm 6, line 14), we decided to unroll eight iterations of the main loop of Threefish (Algorithm 6, lines 4 to 19). The rotation constants $R_{d,i}$ are included in the microcode executed by the control unit. Note that our register file is designed for Threefish-1024 (*i.e.* N_w = 16 and N_r = 20). It is therefore straightforward to implement the two other variants of the algorithm on our architecture.

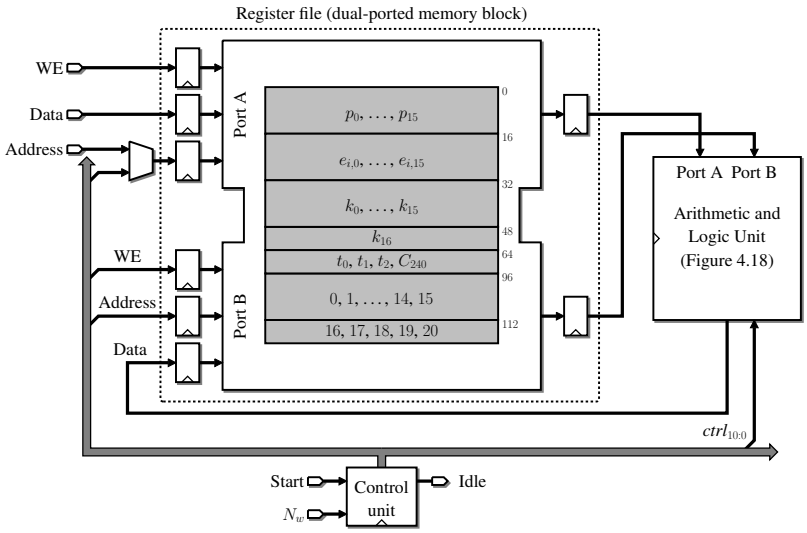


Figure 4.16: Architecture of the Threefish coprocessor.

The user loads messages, plaintext blocks or ciphertext blocks into port A. A few control bits allows one to select the algorithm and the desired level of security. When the coprocessors are hashing or encrypting a message, the intermediates results are always written to port B. In the following, we assume that our coprocessors are provided with padded messages.

4.5.3.1 Arithmetic and logic units for Threefish and Skein

The next step consists in defining the architecture of the ALU and the instruction set of our coprocessor. Our first ALU implements Threefish encryption and Skein. In the following, Ri denotes a 64-bit register. Figure 4.17 illustrates

our scheduling of the two mixing functions $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ of the fifth round of Threefish-256:

- The operand $e_{4,1}$ is loaded in register R1; at the same time, we start the computation of $e_{4,1} \lll R_{4,0}$; this operation requires three clock cycles and intermediate results are stored in R4, R5, and R6.
- Then, $e_{4,0}$ is loaded in register R2; the content of R1 is not modified (*i.e.* R1 must be controlled by an enable signal).
- We execute the instruction $\text{R3} \leftarrow \text{R1} \boxplus \text{R2}$ and obtain $f_{4,0}$.
- R3 and R6 contain $f_{4,0}$ and $e_{4,1} \lll R_{4,0}$, respectively. The instruction $\text{R3} \leftarrow \text{R3} \oplus \text{R6}$ allows us to compute $f_{4,1}$.

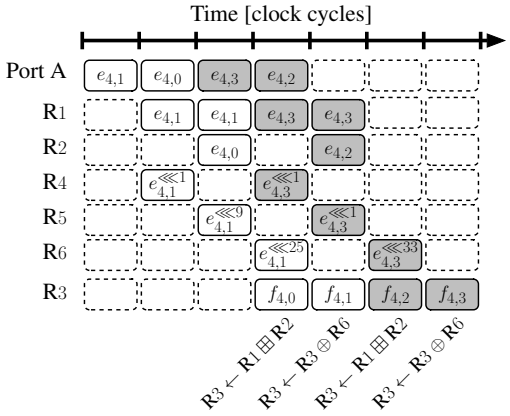


Figure 4.17: Computation of $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ (Threefish-256).

We schedule $\text{Mix}_{4,1}$ as soon as $e_{4,0}$ has been read, and manage to keep the pipeline continuously busy. In summary, our ALU must be able to carry out any rotation of a 64-bit word and to perform the following operation (Figure 4.18):

$$\text{R3} \leftarrow \begin{cases} \text{R1} \boxplus \text{R2} & \text{when } \text{ctrl}_{10} = 0, \\ \text{R3} \oplus \text{R6} & \text{otherwise,} \end{cases} \tag{4.1}$$

where ctrl_{10} denotes a control bit. Let us define two 64-bit operands a and b such that:

$$(a, b) = \begin{cases} (\text{R1}, \text{R2}) & \text{when } \text{ctrl}_{10} = 0, \\ (\text{R3}, \text{R6}) & \text{otherwise.} \end{cases}$$

It is well-known that $a \boxplus b = (a \vee b) \boxplus (a \wedge b)$ and $a \oplus b = (a \vee b) \boxminus (a \wedge b)$, where $\vee$, $\wedge$, and $\boxminus$ denote the bitwise OR, the bitwise AND, and the subtraction modulo

2^{64} of two operands, respectively [91]. Thus, Equation (4.1) can be rewritten as follows:

$$\mathrm{R3} \leftarrow (a \vee b) \boxplus ((a \wedge b) \oplus \mathrm{ctrl}_{10}) \boxplus \mathrm{ctrl}_{10}. \tag{4.2}$$

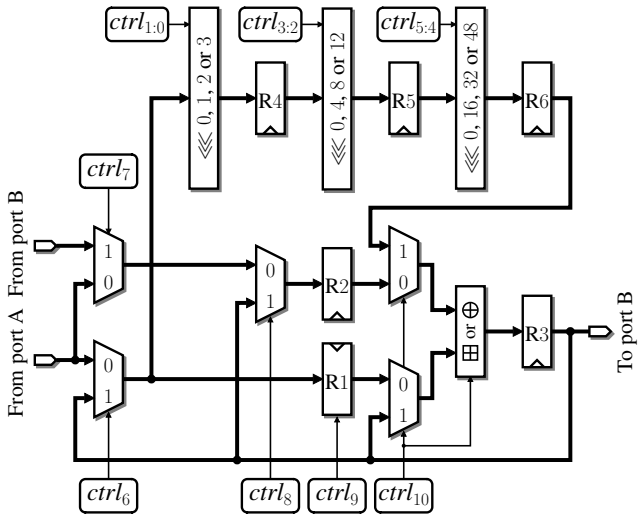


Figure 4.18: Arithmetic and logic unit for Threefish encryption.

Figure 4.19 describes the implementation of Equation (4.2) on a Virtex-6 device. Since there is a single control signal to choose the arithmetic operation and to select a and b, Equation (4.2) involves only five variables, and is advantageously implemented by 64 LUT6_2 primitives and dedicated carry logic.

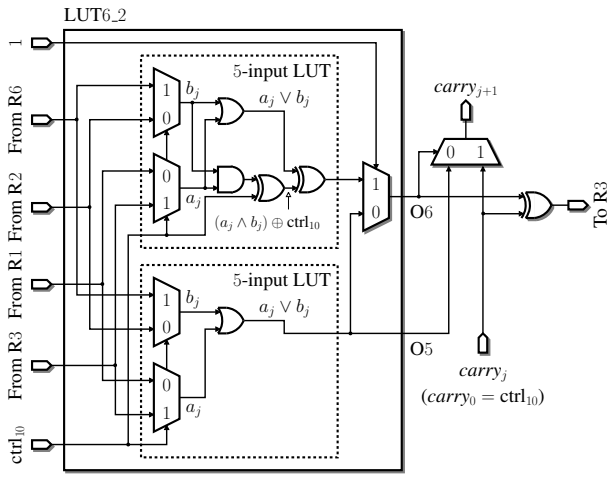


Figure 4.19: Computation of R3 ← R1 ⊞ R2 or R3 ← R3 ⊕ R6 on a Virtex-6 device.

In order to reduce the number of operands stored in the register file, we interleave the key schedule (Algorithm 5) and the encryption process (Algo-

rithm 6). This approach allows us to generate the subkeys on-the-fly. It is however necessary to compute t_2 and k_{N_w} before the first key injection. The easiest way to compute t_2 would be to load t_0 and t_1 in registers R1 and R2, respectively, and to execute the instruction R3 $\leftarrow$ R1 $\oplus$ R2. Unfortunately, this solution requires one more control bit to select the inputs of the arithmetic operator, and it is not possible to implement the multiplexers and the adder on the same LUT6_2 primitive anymore. Since the critical path of our coprocessor is located in the 64-bit adder, an extra level of LUTs would decrease the clock frequency. However, we are able to compute t_2 using only the functionalities defined by Equation (4.1). Since $t_2 = (t_0 \boxplus 0) \oplus (t_1 \lll 0)$, it suffices to execute the following instructions:

$$
\begin{array}{lll}
\text{R4} \leftarrow t_1 \lll 0, & & \\
\text{R1} \leftarrow t_0, & \text{R2} \leftarrow 0, & \text{R5} \leftarrow \text{R4} \lll 0, \\
\text{R3} \leftarrow \text{R1} \boxplus \text{R2}, & \text{R6} \leftarrow \text{R5} \lll 0, & \\
\text{R3} \leftarrow \text{R3} \oplus \text{R6}. & &
\end{array}
$$

This approach assumes that we can read simultaneously two values from the register file. Thanks to the multiplexer controlled by ctrl_7, we can load data from port A or port B into register R_2 (Figure 4.18). A similar strategy allows us to compute k_{N_w}.

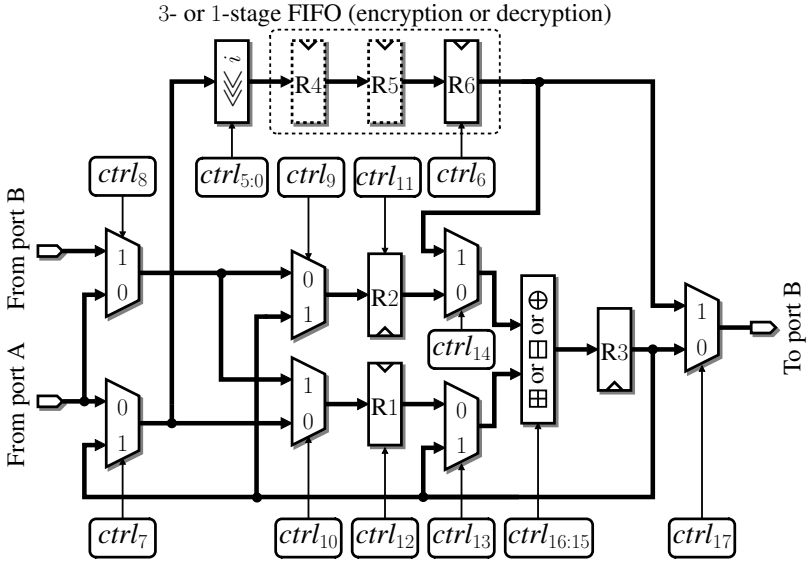


Figure 4.20: Arithmetic and logic unit for Threefish encryption and decryption.

The implementation of the key injection is more straightforward. Note that the multiplexers controlled by ctrl_6 and ctrl_8 allow us to bypass the register file and to use the content of R3 as an input to the ALU. Let us consider for instance the first key injection of Threefish-256: $e_{0,2}$ is defined as $p_2 \boxplus k_{0,2} = p_2 \boxplus k_2 \boxplus t_1$ and is computed as follows:

$$\begin{aligned} &\text{R1} \leftarrow k_2, && \text{R2} \leftarrow t_1, \\ &\text{R3} \leftarrow \text{R1} \boxplus \text{R2} \\ &\text{R1} \leftarrow \text{R3}, && \text{R2} \leftarrow p_2, \\ &\text{R3} \leftarrow \text{R1} \boxplus \text{R2}. \end{aligned}$$

Figures 4.21 and 4.22 describe how we schedule the instructions of the encryption and decryption of Threefish-256, respectively.

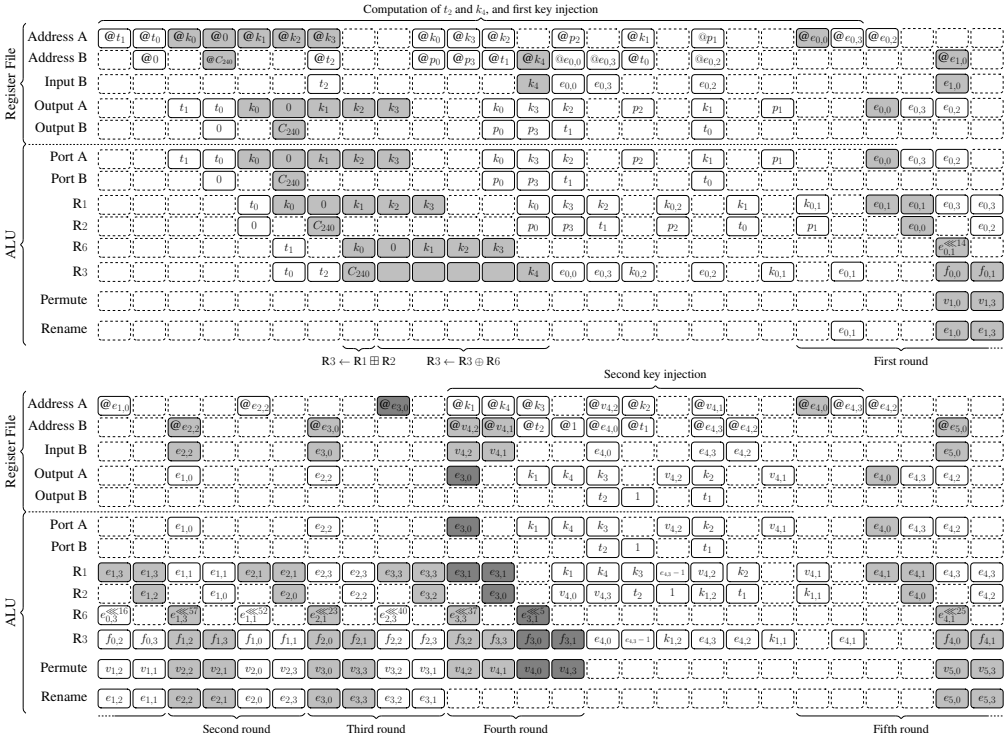


Figure 4.21: Scheduling of Threefish-256 encryption. @d denotes the address of the 64-bit word d in the register file. "Rename" and "Permute" refer to lines 9 and 17 of Algorithm 6, respectively.

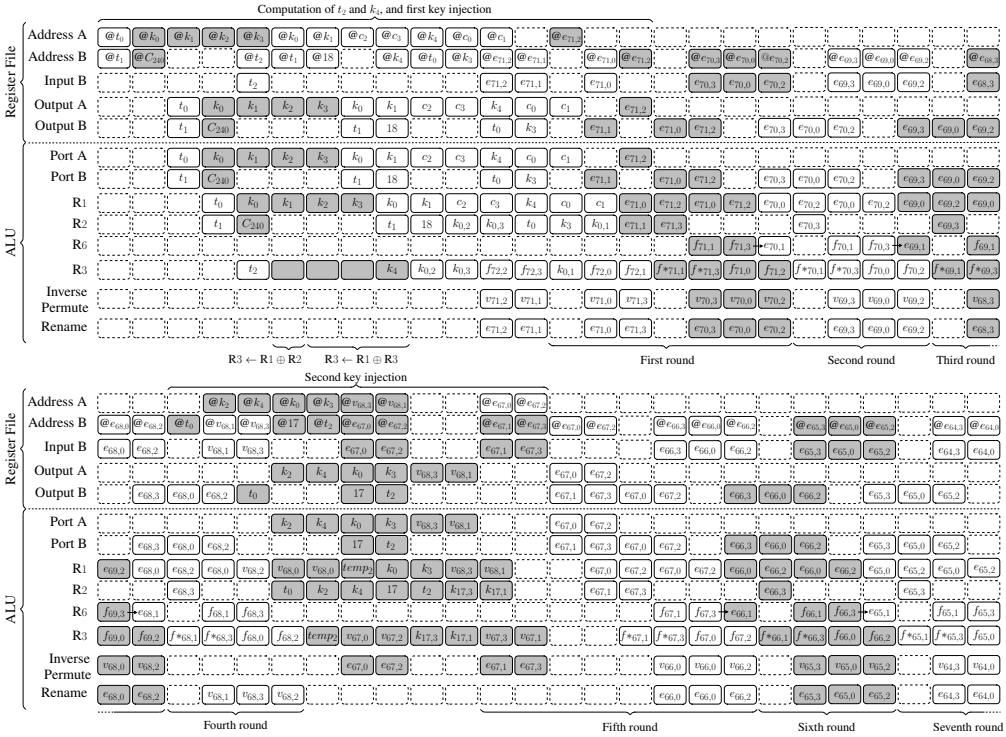


Figure 4.22: Scheduling of Threefish-256 decryption. @d denotes the address of the 64-bit word d in the register file.

The UBI chaining mode can be combined with the final key injection of Threefish encryption. It suffices to modify line 21 of Algorithm 6 as follows:

$$e_{N_r,i} \leftarrow v_{N_r,i} \boxplus k_{N_r/4,i};$$
$$c_i \leftarrow e_{N_r,i} \oplus p_i.$$

The only difference between this operation and the mixing function $\text{MIX}_{d,j}$ is that no permutation is applied to the second operand of the bitwise exclusive OR.

The definition of ALU up to this point is general and applicable to support Threefish encryption at all levels of security. It is also applicable to support Skein at all levels of security by means of extra instructions for UBI chaining mode in the instruction memory. One can have different coprocessors which are able to hash or encrypt at different levels of security with this ALU by changing the instructions in the control unit.

In order to support Threefish decryption, it is essential to incorporate the inverses of the operations required by Threefish encryption. Therefore, it is necessary to analyze these inverses. The inverse of the MIX function being

purely sequential, Threefish decryption has less parallelism than encryption. We suggest to modify our ALU as follows to fully support both encryption and decryption (Figure 4.20):

- The inverse of the Mix function and the inverse of the key injection require a subtraction modulo 2^{64}. Our modified ALU is therefore able to perform a new operation: R3 $\leftarrow$ R1 $\boxminus$ R2. Because of the additional control bit required to select the operation, it is not possible to implement our arithmetic operator by means of 64 LUT6_2 anymore. Thus, the slice count and the critical path are expected to increase.

- The output of the inverse Mix function is provided either by the arithmetic operator (e.g. $e_{4,0}$ on Figure 4.14) or the rotation unit (e.g. $e_{4,1}$ on Figure 4.14). The multiplexer controlled by ctrl_{17} allows us to select the word we store in the register file.

- Since the inverse of the Mix function is sequential, we have to perform the rotation in a single clock cycle. We suggest to take advantage of the SRL16E primitive available on Xilinx devices to implement a FIFO whose depth is dynamically adjusted according to the algorithm selected by the user: one and three stages for decryption and encryption, respectively.

4.5.3.2 Register files and control units

Control unit is responsible to generate all the addresses for the register file and the signals which manage the data flow in the ALU. For example, Virtex-6 FPGAs embed several configurable memory blocks that can for instance store 1024 36-bit words or 2048 18-bit words. The control unit mainly consists of a program counter that addresses an instruction memory implemented by means of a memory block. For instance, for the Skein-512-512 coprocessor that does not support decryption of Threefish-512, the control unit consists of an instruction memory, a small table to generate the address of $k_{(s+i) \bmod (N_w+1)}$ during the key schedule (Algorithm 5), and a simple finite-state machine. Since we unroll only eight rounds, we manage to keep the instruction memory compact. In the case of Threefish-512, we need for instance 145 instructions and 881 clock cycles to encrypt a plaintext block in UBI mode.

In order to process a k-block message M, we load the precomputed value of G_0 in the register file of our coprocessor. Then, $k+1$ calls to Threefish allow us to process the message and to perform the output transform. In the case of

Skein-512-512, the throughput is given by $T = \frac{512 \cdot k \cdot f}{(k+1) \cdot 881}$ bits/s, where f denotes the clock frequency of our architecture.

For all proposed coprocessors, which supports different set of functionalities among encryption and decryption of Threefish and Skein, the register file is organized into 64-bit words, and stores a plaintext block, an internal state ($e_{d,i}$, where $0 \leq i \leq N_w - 1$), an extended block cipher key, an extended tweak, the constant C_{240}, and all possible values of s involved in the key schedule (Figure 4.23). Thanks to this approach, the word permutation $\pi(i)$ and the

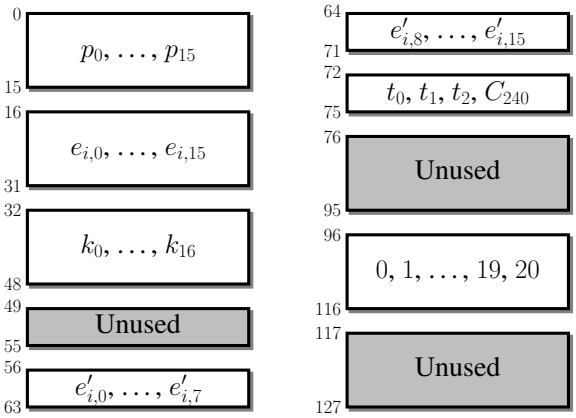


Figure 4.23: Register file of our Threefish and Skein architectures.

word rotation of the key schedule are conveniently implemented by addressing the register file accordingly. Since the round constants repeat every eight rounds (Algorithm 6, line 14), we decided to unroll eight iterations of the main loop of Threefish (Algorithm 6, lines 4 to 19). The rotation constants $R_{d,i}$ are included in the microcode executed by the control unit. Note that our register file is designed for Threefish-1024 (*i.e.* $N_w = 16$ and $N_r = 80$). It is therefore straightforward to implement the two other variants of the algorithm on our architecture. The number of clock cycles required for Threefish encryption and decryption according to the key size is summarized in Table 4.6, where our unified coprocessor for the Skein and Threefish algorithms is considered (see its ALU in Figure 4.20). Because of the output transform, $k+1$ invocations of the tweakable encryption function are necessary to hash a k-block message with Skein. There is a latency of 5 clock cycles between two consecutive Threefish encryption. Thus, the throughput of Skein is given by:

$$throughput = \frac{8 \cdot N_b \cdot k \cdot f}{(k+1) \cdot latency\ of\ Threefish\ with\ UBI + 5 \cdot k},$$

where f denotes the clock frequency.

Table 4.6: Number of instructions of the algorithms of the Threefish family.

Algorithm	# instructions
Threefish-256 encryption	490
Threefish-256 encryption with UBI	501
Threefish-256 decryption	469
Threefish-512 encryption	860
Threefish-512 encryption with UBI	882
Threefish-512 decryption	1092
Threefish-1024 encryption	1874
Threefish-1024 encryption with UBI	1912
Threefish-1024 decryption	2351

Instructions, which are stored in the instruction memory, carry all information needed to control the data flow in the ALUs proposed in this work and generates the required addresses for the Register File. The length of the instructions of the coprocessors varies depending on which algorithms are supported. We compressed the instructions for all coprocessors in different ways. Our aim is to fit all instructions into 18-Kb block RAM utilizing as small as possible logic area for decoding. Compressed instructions are stored in a Block-RAM and then decompressed in the coprocessor by small decoding circuit thanks to our optimized instruction compression.

The same approach, which we followed for the design of the control units for unified coprocessor for the Skein and Threefish algorithms, can easily be applied to the other coprocessors that consist of similar set of mathematical operations.

4.5.4 Results and comparisons

We captured our architectures in the VHDL language and prototyped our coprocessors on a Xilinx Virtex-6 FPGA with average speedgrade for this work. Table 4.7 summarizes our results for fully autonomous implementation of Skein-512-512 coprocessor (the coprocessor which supports only one level of security, 512) and the figures published by other researcher focusing on compact coprocessors (we refer the reader to the SHA-3 Zoo [92] for an overview of high-speed designs). Note that we considered the least favorable case, where the message consists of a single block, to compute the throughput. If we increase the size of the message, the throughput of our coprocessor converges asymptotically to 160 Mbits/s. Note also that some of the other hardware architectures of Skein makes only a single call to Threefish-512 with-

out performing output transform (see the results in Table 4.7), whereas our architecture performs complete UBI invocations for Skein-512-512.

Most of the architectures described in the open literature focus on a single level of security (Table 4.7). We took advantage of the intrinsic parallelism of Skein to interleave the computation of independent tasks of Threefish. A careful scheduling allowed us to decrease pipeline bubbles and memory collisions. We also addressed FPGA-specific issues and described how to share slices between addition and bitwise exclusive OR of two operands. As a consequence, our coprocessors provide the end-user with hashing and encryption at all levels of security, while offering a better area–time trade-off.

Let us assume that all SHA-3 finalists provide the levels of security expected by the NIST. Then, according to Table 4.7, BLAKE, Keccak, and Skein seem to be the best candidates for compact implementations on FPGA. From our point of view, the main advantage of Skein over other SHA-3 finalists is that the same coprocessor also allows one to encrypt or hash a message.

Table 4.7: Compact implementations of the five SHA-3 finalists on Virtex-5 and Virtex-6 FPGAs.

	Algorithm	**FPGA**	**Area [slices]**	**36k memory blocks**	**Frequency [MHz]**	**Throughput [Mbps]**
Aumasson *et al.* [80]	BLAKE-256	xc5vlx110	390	–	91	412
Beuchat *et al.* [57]†	BLAKE-256	xc6vlx75t-2	52	2	456	194
Jungk [93]	BLAKE-256	xc6v	260	–	263	590
Jungk [93]	BLAKE-256	xc6v	419	–	204	908
Kaps *et al.* [94]	BLAKE-256	xc6vlx75t-1	163	1	197	327
Kaps *et al.* [94]	BLAKE-256	xc6vlx75t-1	166	–	268	445
Aumasson *et al.* [80]	BLAKE-512	xc5vlx110	939	–	59	468
Beuchat *et al.* [57]†	BLAKE-512	xc6vlx75t-2	81	3	374	280
Kerckhof *et al.* [89]	BLAKE-512	xc6vlx75t-1	192	–	240	183
Yamazaki *et al.* [95]	BLAKE (unified coprocessor)	xc5vlx50-2	138	3	342	2×150 (BLAKE-256) 264 (BLAKE-512)
Jungk [96]	Grøstl-256	xcv5	470	–	354	1132
Kerckhof *et al.* [89]	Grøstl-512	xc6vlx75t-1	260	–	280	640
Jungk [96]	JH-256	xcv5	205	–	341	27
Kerckhof *et al.* [89]	JH-512	xc6vlx75t-1	240	–	288	214
Bertoni *et al.* [39]	Keccak[$r = 1024, c = 576$]	xc5vlx50-3	448	–	265	52
Kerckhof *et al.* [89]	Keccak[$r = 1024, c = 576$]	xc6vlx75t-1	144	–	250	68
San & At [75]	Keccak[$r = 1024, c = 576$]	xc5vlx50-2	151	3	520	501
This Work	Skein-512-512	xc6vlx75t-1	132	2	276	80
Jungk [96]‡	Skein-512-256	xcv5	555	–	271	237
Jungk [93]‡	Skein-512-256	xc6v	406	–	316	277
Kaps *et al.* [94]	Skein-512-256	xc6vlx75t-1	207	1	166	17
Kaps *et al.* [94]	Skein-512-256	xc6vlx75t-1	193	–	193	21
Kerckhof *et al.* [89]‡	Skein-512-512	xc6vlx75t-1	240	–	160	179
Latif *et al.* [97]‡	Skein-256-256	xc5vlx110-3	821	Not specified	119	1610

†Modified to implement the tweaked version submitted for the final round of the SHA-3 competition.
‡Single call to Threefish-512.

Tables 4.8 summarizes our place-and-route results measured with ISE 14.2 for our various coprocessors which supports different set of functionalities of Skein and Threefish algorithms. In this table, for the throughput values of

Skein algorithm, both the least favorable case, where the message consists of a single block, and the most favorable case, where the message consists of larger blocks, are considered to compute the throughput of Skein.

Table 4.8: Place-and-route results for our Threefish and Skein coprocessors on a Virtex-6 FPGA (xc6vlx75t-2).

Supported algorithms	Area [slices]	# block RAMs	Freq. [MHz]	Throughput [Mbits/s]									
				Skein-256-256	Skein-512-512	Skein-1024-1024	Threefish-256		Threefish-512		Threefish-1024	
							Enc	Dec	Enc	Dec	Enc	Dec
Threefish-256 Enc	139	2	366	–	–	–	191	–	–	–	–	–
Threefish-512 Enc	146	2	355	–	–	–	–	–	211	–	–	–
Threefish-1024 Enc	179	2	346	–	–	–	–	–	–	–	189	–
Threefish-256 Dec	241	2	272	–	–	–	–	148	–	–	–	–
Threefish-512 Dec	275	2	293	–	–	–	–	–	–	137	–	–
Threefish-1024 Dec	286	2	278	–	–	–	–	–	–	–	–	121
Skein-256-256	154	2	348	88—176	–	–	–	–	–	–	–	–
Skein-512-512	149	2	321	–	92—185	–	–	–	–	–	–	–
Skein-1024-1024	193	2	344	–	–	91—183	–	–	–	–	–	–
Unified Threefish Encryptions (all levels of security)	145	3	294	–	–	–	153	–	175	–	160	–
Unified Threefish (all levels of security)	277	3	267	–	–	–	139	145	158	125	145	116
Unified Threefish Encryptions and Skein (all levels of security)	150	3	295	75* 150**	85* 170**	78* 156**	154	–	175	–	161	–
Unified Threefish and Skein (all levels of security)	292	3	279	70* 140**	80* 160**	74* 148**	145	152	166	130	152	121

*One block of message is considered to compute the throughput.
**Message consists of larger blocks is considered. If we increase the size of the message, the throughput of our coprocessors converges asymptotically to these values.

We report in Figure 4.24 the latest lightweight implementation results of several cryptographic hash functions. Besides our coprocessors for BLAKE-512 and Skein-512-512, we selected Grøstl [76], JH [89], SHA-2-512 [98], and SHA-3-512 (Keccak $[r = 1024, c = 576]$) [75]. In this context, BLAKE is obviously the best choice for lightweight implementations on FPGA. From a different point of view, Skein has the advantage that it consists of Threefish–block cipher. It means that the same coprocessor supporting several levels of security also allows one to encrypt or hash a message.

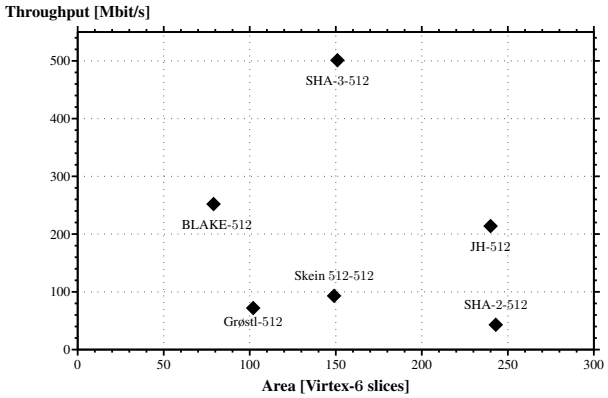


Figure 4.24: Compact implementations of several cryptographic hash functions on Virtex-6 FPGAs (512-bit digests).

Lightweight implementations of ECHO & AES proposed in [33] and Grøstl & AES[1] proposed in [76] (Table 4.14) are also unified coprocessors that support both encryption and hashing functions. Given that all symmetric cryptographic functions (including authenticated encryption) can be efficiently implemented with Keccak, we would get the following figures with a unified architecture based on [75]:

- hashing with arbitrary length at a security level of 256 bits: 501 Mbits/s;
- authenticated encryption at a security level of 256 bits: more than 501 Mbits/s (the generic security of keyed sponges allows one to use less capacity than for hashing, hence a larger rate and a proportionally larger throughput) [101].

Table 4.9: Place-and-route results for hashing and AES encryption on a Virtex-6 FPGA (xc6vlx75t-2).

FPGA	Area [slices]	Frequency [MHz]	Throughput [Mbits/s]				
			AES-128	AES-192	AES-256	256-bit digest	512-bit digest
AES & ECHO [33]	155	397	219	186	161	92	48
AES & Grøstl [76]	169	393	217	184	159	92	69

4.5.5 Conclusion

The block cipher Threefish and the hash functions Skein are based on the same arithmetic operations. The design philosophy followed in this study allows one to design lightweight coprocessors for hashing and encryption. The key element of the approach is to take advantage of the parallelism of the algorithms to:

- deeply pipeline the ALU to achieve a high clock frequency;
- avoid data dependencies by interleaving independent tasks.

Furthermore, it is described how to design compact control units thanks to a careful organization of the register file, loop unrolling, and a simple compression algorithm. Our architectures are mainly designed for embedded systems. Thus, it would be interesting to conduct side-channel and fault injection attacks in future work.

[1]Note that Järvinen [99] proposed the first unified coprocessor for AES-128 (encryption and key expansion) and Grøstl-256. Recently, Rogawski & Gaj [100] designed a parallel coprocessor for Grøstl-based HMAC and AES in the counter mode. Both architectures are optimized for high-speed implementations, and it is therefore difficult to make a comparison with our lightweight coprocessors.

4.6 Compact Hardware Architectures for Grøstl Cryptographic Hash Function

In this section, the problem of designing efficient hardware architecture, where throughput, delay and area metrics are all significantly important, is considered. The aim is to find an efficient method to design a compact coprocessor that is low-area and high-throughput as much as possible. Accordingly, we investigate a compact design problem for the AES and the Cryptographic Hash Function Grøstl [36] and propose a novel unified hardware architecture able to compute both algorithm. The initial results presented in this section have been published in [76].

Since the SHA-3 finalist Grøstl [37] is strongly inspired by the Advanced Encryption Standard (AES) [32], it is tempting to design a compact unified hardware architecture which supports both algorithms. Such an implementation is valuable for constrained environments, where some security protocols mainly rely on cryptographic hash functions (see for instance [90]). Furthermore, as emphasized by Kerckhof *et al.*, "fully unrolled and pipelined architectures may sometimes hide a part of the algorithms' complexity that is better revealed in compact implementations" [89]. In order to have a deeper understanding of the computational efficiency of several SHA-3 candidates (resource sharing, memory access scheme, scheduling, etc.), we already designed five low-area coprocessors [33,57,75,77,102]. In particular, we proposed a compact unified architecture for the SHA-3 round 2 candidate ECHO [103] and the AES. The main originality of our work was to describe the AES by means of a single instruction [33]. Since ECHO is built around the round function of the AES, it is rather straightforward to design a unified Arithmetic and Logic Unit (ALU) for both algorithms. In the conclusion of our article, we stated that our design strategy could be applied to Grøstl and AES. However, even if Grøstl borrows the the S-box of the AES, the construction of the diffusion layers is only based on the design philosophy of the AES. Contrary to ECHO, Grøstl can not be implemented with the AES instruction set of Intel Westmere processors [104], and it seems much more challenging to build a compact unified coprocessor. We bring a solution to this problem in this thesis.

4.6.1 The Advanced Encryption Standard

After an initial AddRoundKey step, an AES encryption involves $N_r - 1$ repetitions of a round composed of the four byte-oriented transformations described in Section 2.1.1.3. Eventually, a final encryption round, in which the MixColumns step is omitted, produces the ciphertext (Figure 2.3). We consider here the equivalent decryption algorithm described in [32, Section 3.7.3]. Its main advantage over the straightforward decryption process is that encryption and decryption rounds share the same datapath (Figure 2.3). Nevertheless, the round keys are introduced in reverse order for decryption. A key expansion algorithm allows one to derive the round keys involved in the AddRoundKey steps from the cipher key. Let us consider an array consisting of 4 rows and $4 \cdot (N_r + 1)$ columns. The cipher key is copied in the first N_k columns of the array, and the next columns are defined recursively (see [32, Section 3.6] for details).

If an AES coprocessor is built around an 8-bit datapath, the ShiftRows and InvShiftRows steps are implemented by accordingly addressing the register file organized into bytes. As a result, these operations are virtually for free and do not require dedicated hardware in the ALU. It is then possible to describe encryption, decryption, and key expansion with a single instruction [33]:

$$R_k \leftarrow \mathcal{A} \cdot g(R_i) \oplus \mathcal{B} \cdot R_j, \tag{4.3}$$

where

- R_i, R_j, and R_k are vectors of N_l bytes.
- $\mathcal{A}$ and $\mathcal{B}$ are matrices of $N_l \times N_l$ bytes. Let us define the identity matrix $\mathcal{I}_{\text{AES}} = \text{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00})$ and the permutation matrix $\mathcal{P}_{\text{AES}} = \text{circ}(\texttt{00}, \texttt{01}, \texttt{00}, \texttt{00})$. The latter matrix is essential to the key schedule. We showed in [33] that $\mathcal{A}$ can be any of the four matrices we introduced so far, whereas $\mathcal{B}$ is either $\mathcal{M}_D$ or the identity matrix $\mathcal{I}_{\text{AES}}$.
- g is a function applied to each byte of R_i. In addition to S_{RD} and $\text{S}_{\text{RD}}^{-1}$, we need the identity function to implement the key expansion and the first AddRoundKey step of AES encryption or decryption. The first instruction of the decryption process (Figure 2.3) is for instance

$$A_0 \leftarrow \mathcal{I}_{\text{AES}} \cdot A_0 \oplus \mathcal{I}_{\text{AES}} \cdot K_{4N_r}.$$

Since the ALU processes the operands byte by byte, the computation of Equation (4.3) involves at least N_l cycles. In order to achieve a high clock frequency on FPGA, it is however necessary to make the pipeline deeper. The main challenge is to schedule the instructions to avoid pipeline bubbles as much as possible.

4.6.2 The hash function Grøstl

In this work, we assume that our coprocessor is provided with a padded message and refer the reader to [37, Section 3.6] for a description of the padding algorithm. A hardware wrapper interface for Grøstl (and several other hash functions) comprising communication and padding is for instance described in [74]. Starting from an initial chaining input $H^{(0)} = IV_n$, the message blocks $M^{(1)}, \ldots, M^{(t)}$ are processed by a compression function f as:

$$H^{(i)} \leftarrow f(H^{(i-1)}, M^{(i)}),$$

where $1 \leq i \leq t$. Figure 4.25 describes the datapath of the compression function f that consists of a key schedule, and two permutations P_ℓ and Q_ℓ operating on two $8 \times N_c$ array of bytes P and Q. Each byte is considered as an element of $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/(m(x))$, where $m(x) = x^8 + x^4 + x^3 + x + 1$ is the irreducible polynomial of the AES.

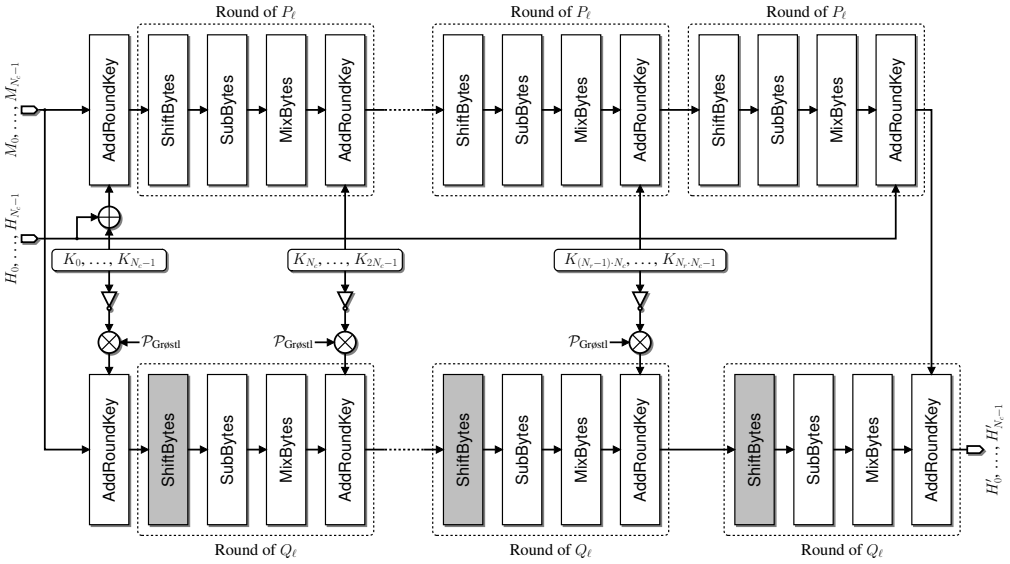


Figure 4.25: Flowchart of the compression function f of Grøstl.

Each permutation involves a first key injection followed by N_r rounds

consisting of four byte-oriented transformations similar to those of the AES (N_r depends on the digest size and is defined in Table 2.3):

- The ShiftBytes step cyclically rotates the ith row of P and Q to the left by $\sigma_P(i)$ and $\sigma_Q(i)$ bytes, respectively. Let $T = \mathsf{ShiftBytes}(P)$ and $U = \mathsf{ShiftBytes}(Q)$. We have:

$$t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c},$$
$$u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c},$$

  where $0 \le i \le 7$ and $0 \le j \le N_c - 1$. The offsets $\sigma_P(i)$ and $\sigma_Q(i)$ depend on the row index i and the number of columns N_c, and are defined in Table 4.10.

Table 4.10: Offsets of the ShiftBytes transformation according to the row index i and the number of columns N_c.

i	0	1	2	3	4	5	6	7
$\sigma_P(i)$	0	1	2	3	4	5	6	$\frac{N_c}{2}+3$
$\sigma_Q(i)$	1	3	5	$\frac{N_c}{2}+3$	0	2	4	6

- The SubBytes step updates each byte of P and Q using the AES S-box, denoted by $\mathrm{S_{RD}}$.

- The MixBytes step is performed by multiplying each column of P and Q by the circulant matrix $\mathcal{M}_{\text{Grøstl}} = \mathrm{circ}(\mathsf{02}, \mathsf{02}, \mathsf{03}, \mathsf{04}, \mathsf{05}, \mathsf{03}, \mathsf{05}, \mathsf{07})$.

- The AddRoundKey step combines P and Q with two ℓ-bit round keys. The key expansion requires $N_r \cdot N_c$ 64-bit constants that can be computed on-the-fly according to Algorithm 7. Since the round counter r and the loop index j are 4-bit numbers ($N_r \le 14$ and $N_c - 1 \le 15$), each $k_{i,r \cdot N_c + j}$ can be seen as an element of $\mathbb{F}_{2^8}$.

  Besides the round constants, the key expansion involves the chaining input H and a permutation matrix $\mathcal{P}_{\text{Grøstl}} = \mathrm{circ}(\mathsf{00}, \mathsf{01}, \mathsf{00}, \mathsf{00}, \mathsf{00}, \mathsf{00}, \mathsf{00}, \mathsf{00})$ (Figure 4.25). Finally, note that the output of P_ℓ serves as the key of the last round of Q_ℓ.

Algorithm 8 describes how we implement Grøstl using the instruction defined by Equation (4.3). The ShiftBytes step (*i.e.* computation of T_j and U_j on lines 10, 17, 25, and 32) is performed by accordingly addressing the

Algorithm 7 Computation of the round constants.

Require: N_r (number of rounds of the permutation P) and N_c (number of columns of the internal state).

Ensure: The $N_r \cdot N_c$ round constants required to compute the permutation P.

1. **for** $r \leftarrow 0$ **to** $N_r - 1$ **do**
2. **for** $j \leftarrow 0$ **to** $N_c - 1$ **do**
3. $k_{0,r \cdot N_c + j} \leftarrow j \parallel r$;
4. **for** $i \leftarrow 1$ **to** 7 **do**
5. $k_{i,r \cdot N_c + j} \leftarrow 00$;
6. **end for**
7. **end for**
8. **end for**
9. **Return** $K_0, K_1, \ldots, K_{N_r \cdot N_c - 1}$;

register file organized into bytes. As a result, these operations are virtually for free and do not require dedicated hardware in the ALU. Since the ShiftBytes transformations performs cyclical left shifts of the rows of the state, we have to be careful not to overwrite bytes that are still involved in the forthcoming MixBytes steps ($p_{1,0}$ is for instance needed to update the eighth column of P, and should not be overwritten when updating the first column). We solve this problem by introducing two $8 \times N_c$ arrays of bytes P' and Q' to store intermediate results. The coprocessor we will describe in Section 4.6.3 embeds a number of pipeline stages, and several clock cycles are required to process a byte of the state. In order to avoid data dependency issues between two consecutive rounds of a given permutation, we interleave the computation of P_ℓ and Q_ℓ.

After the last message block has been processed, an output transformation based on P_ℓ generates the n-bit digest D (Algorithm 9). The function $\text{trunc}_n(P')$ (line 19) discards all but the n trailing bits of P'. Table 4.11 provides the reader with a summary of the instructions involved in the compression function and the output transformation.

4.6.3 A compact unified coprocessor for the AES and the Grøstl family of hash functions

Figure 4.26 describes how we modified the 8-bit coprocessor proposed by Beuchat *et al.* [33] in order to share the same datapath between Grøstl and the AES. The architecture is built around an 8-bit datapath and consists of three main components:

Algorithm 8 Compression function f of Grøstl.

Require: A ℓ-bit message block M and a chaining value H.
Ensure: A new chaining value H'.

for $j \leftarrow 0$ **to** $N_c - 1$ **do**
  $P_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot M_j \oplus \mathcal{I}_{\text{Grøstl}} \cdot (K_j \oplus H_j)$;
end for
for $j \leftarrow 0$ **to** $N_c - 1$ **do**
  $Q_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot M_j \oplus \mathcal{P}_{\text{Grøstl}} \cdot \neg K_j$;
end for
for $r \leftarrow 1$ **to** $N_r - 1$ **do**
  for $j \leftarrow 0$ **to** $N_c - 1$ **do**
    for $i \leftarrow 0$ **to** 7 **do**
      $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
    end for
    $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_{r \cdot N_c + j}$;
  end for
  $P \leftarrow P'$;
  for $j \leftarrow 0$ **to** $N_c - 1$ **do**
    for $i \leftarrow 0$ **to** 7 **do**
      $u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c}$;
    end for
    $Q'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(U_j) \oplus \mathcal{P}_{\text{Grøstl}} \cdot \neg K_{r \cdot N_c + j}$;
  end for
  $Q \leftarrow Q'$;
end for
for $j \leftarrow 0$ **to** $N_c - 1$ **do**
  for $i \leftarrow 0$ **to** 7 **do**
    $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
  end for
  $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot H_j$;
end for
$P \leftarrow P'$;
for $j \leftarrow 0$ **to** $N_c - 1$ **do**
  for $i \leftarrow 0$ **to** 7 **do**
    $u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c}$;
  end for
  $H'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(U_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot P_j$;
end for
Return $H'_0, \ldots, H'_{N_c-1}$;

Algorithm 9 Output transformation.

Require: An intermediate hash value H
Ensure: A n-bit digest D

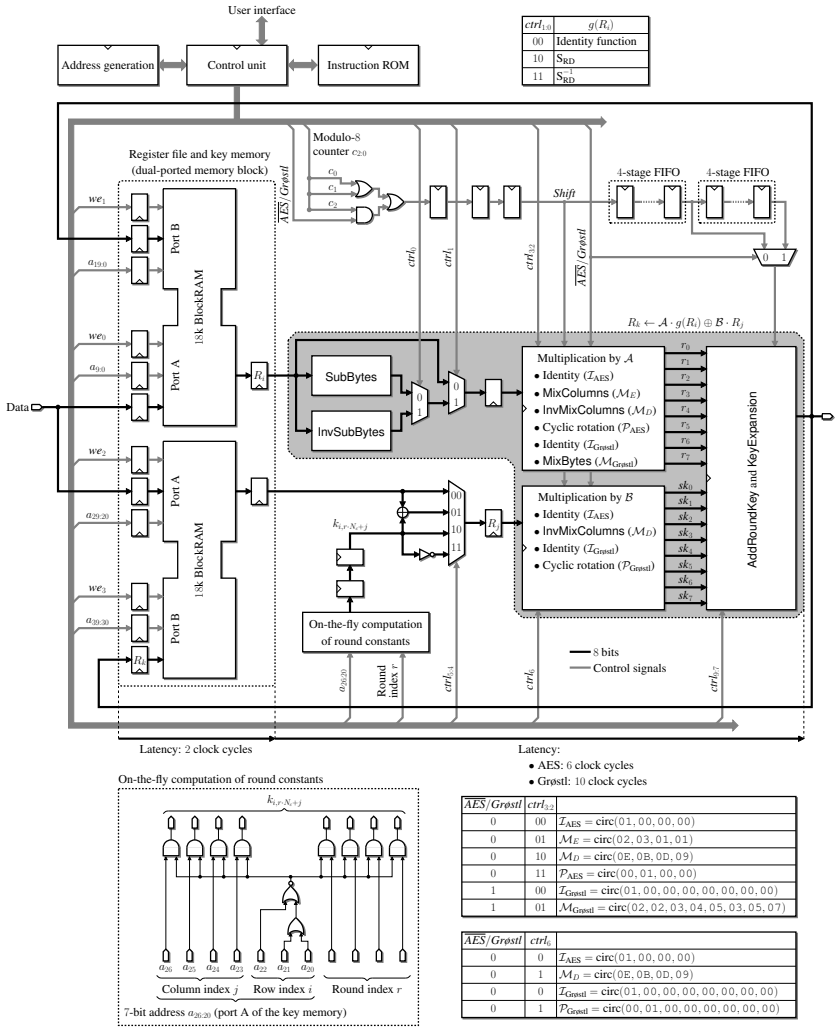
1. **for** $j \leftarrow 0$ **to** $N_c - 1$ **do**
2. $\quad P_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot H_j \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_j$;
3. **end for**
4. **for** $r \leftarrow 1$ **to** $N_r - 1$ **do**
5. $\quad$ **for** $j \leftarrow 0$ **to** $N_c - 1$ **do**
6. $\quad\quad$ **for** $i \leftarrow 0$ **to** 7 **do**
7. $\quad\quad\quad t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
8. $\quad\quad$ **end for**
9. $\quad\quad P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_{r \cdot N_c + j}$;
10. $\quad$ **end for**
11. $\quad P \leftarrow P'$;
12. **end for**
13. **for** $j \leftarrow 0$ **to** $N_c - 1$ **do**
14. $\quad$ **for** $i \leftarrow 0$ **to** 7 **do**
15. $\quad\quad t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
16. $\quad$ **end for**
17. $\quad P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \mathrm{S}_{\mathrm{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot H_j$;
18. **end for**
19. $D \leftarrow \mathrm{trunc}_n(P')$;
20. **Return** D;

- a register file and a key memory implemented by means of a single dual-ported memory block; address bits and write enable signals are denoted by $a_{39:0}$ and $we_{3:0}$, respectively;
- a control unit responsible for the address generation and the selection of the parameters $\mathcal{A}$, $\mathcal{B}$, and g (Table 4.11);
- an ALU implementing the instruction defined by Equation (4.3), the key expansion mechanism of the AES, and the computation of the round

Table 4.11: Implementation of Grøstl with a single instruction.

Operation	R_k	$\mathcal{A}$	g	R_i	$\mathcal{B}$	R_j
Algorithm 8, line 2	P_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	M_j	$\mathcal{I}_{\text{Grøstl}}$	$K_j \oplus H_j$
Algorithm 8, line 5	Q_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	M_j	$\mathcal{I}_{\text{Grøstl}}$	$\neg K_j$
Algorithm 8, line 12 and Algorithm 9, line 9	P'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	T_j	$\mathcal{I}_{\text{Grøstl}}$	$K_{r \cdot N_c + j}$
Algorithm 8, line 19	Q'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	U_j	$\mathcal{P}_{\text{Grøstl}}$	$\neg K_{r \cdot N_c + j}$
Algorithm 8, line 27 and Algorithm 9, line 17	P'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	T_j	$\mathcal{I}_{\text{Grøstl}}$	H_j
Algorithm 8, line 34	Q'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	U_j	$\mathcal{I}_{\text{Grøstl}}$	P_j
Algorithm 9, line 2	P_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	H_j	$\mathcal{I}_{\text{Grøstl}}$	K_j

constants of Grøstl.



$ctrl_{1:0}$	$g(R_i)$
00	Identity function
10	S_{RD}
11	S_{RD}^{-1}

$\overline{AES}$/Grøstl	$ctrl_{2:2}$	
0	00	$\mathcal{I}_{\text{AES}} = \text{circ}(01, 00, 00, 00)$
0	01	$\mathcal{M}_E = \text{circ}(02, 03, 01, 01)$
0	10	$\mathcal{M}_D = \text{circ}(0E, 0B, 0D, 09)$
0	11	$\mathcal{P}_{\text{AES}} = \text{circ}(00, 01, 00, 00)$
1	00	$\mathcal{I}_{\text{Grøstl}} = \text{circ}(01, 00, 00, 00, 00, 00, 00, 00)$
1	01	$\mathcal{M}_{\text{Grøstl}} = \text{circ}(02, 02, 03, 04, 05, 03, 05, 07)$

$\overline{AES}$/Grøstl	$ctrl_0$	
0	0	$\mathcal{I}_{\text{AES}} = \text{circ}(01, 00, 00, 00)$
0	1	$\mathcal{M}_D = \text{circ}(0E, 0B, 0D, 09)$
0	0	$\mathcal{I}_{\text{Grøstl}} = \text{circ}(01, 00, 00, 00, 00, 00, 00, 00)$
0	1	$\mathcal{P}_{\text{Grøstl}} = \text{circ}(00, 01, 00, 00, 00, 00, 00, 00)$

Figure 4.26: General architecture of our unified 8-bit coprocessor for AES and Grøstl.

4.6.3.1 Memory organization

Since we consider an 8-bit datapath, the memory of our coprocessor is organized into bytes. We will show below that ten address bits are needed to access message blocks and intermediate data, thus allowing us to implement the register file and the key memory by means of a single Virtex-6 block RAM configured as two independent 18 Kb RAMs (Figure 4.26).

Recall that each variable of Grøstl-n is an array of $8 \times N_c$ bytes and that the compression function of Grøstl requires six variables: a chaining value H, a

message block M, P, P', Q, and Q' (Algorithm 8). Since $N_c \leq 16$ (Table 2.3), we define six chunks of 128 bytes in the register file (Figure 4.27), and address each byte with 10 bits:

- the three most significant bits select the desired variable;
- the next four bits encode the column index j;
- the three least significant bits define the row index i.

The addresses of $h_{i,j}$ and $p_{i,j}$ are for instance given by $8j + i$ and $256 + 8j + i$, respectively. The key memory stores:

- a copy of the chaining value H required to implement the key schedule (lines 2 and 27 of Algorithm 8, and line 17 of Algorithm 9);
- the initial chaining values IV_{224}, IV_{256}, IV_{384}, and IV_{512};
- a copy of P needed to perform the AddRoundKey step of the last round of Q_ℓ (line 34 of Algorithm 8).

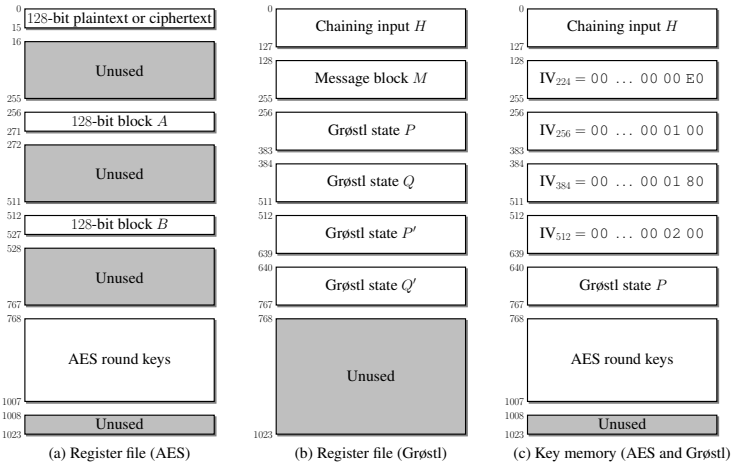


Figure 4.27: Memory organization.

We keep the memory organization proposed in [33] for the AES. Note that the execution of Grøstl-n does not overwrite the round keys of the AES. As long as the AES master key is not modified, it is therefore possible to switch between the hash function and the block cipher with no need for the AES KeyExpansion step. In the following, we show that our careful organization of the data in the register file and in the key memory allows one to design a control unit based on a 256-bit counter, a 128-bit counter, and a simple Finite State Machine (FSM).

4.6.3.2 Control unit

The control bits of the ALU, the read and write addresses of the register file and the key memory, and the write enable signals are computed by a control unit that mainly consists of an address generator and an instruction memory. At first glance, it seems that each algorithm (AES key expansion, AES encryption, AES decryption, P_ℓ, and Q_ℓ) requires a different addressing scheme. However, we described a way to generate all read and write addresses of the AES and the hash function ECHO [103] by means of a modulo-16 counter and a modulo-256 counter in our previous work [33]. The same design philosophy allows us to generate the addresses of Grøstl. Since the internal state contains up to 128 bytes, we have to replace the modulo-16 counter by a modulo-128 counter. Our control unit generates a read address and its corresponding write address at each clock cycle. Since our coprocessor embeds several pipeline stages, it is mandatory to delay write addresses and write enable signals accordingly. Furthermore, the latency depends on the algorithm being executed (Figure 4.26). On Xilinx devices, an efficient solution consists in synchronizing control signals by means of SRL16 primitives, whose depth can be dynamically adjusted.

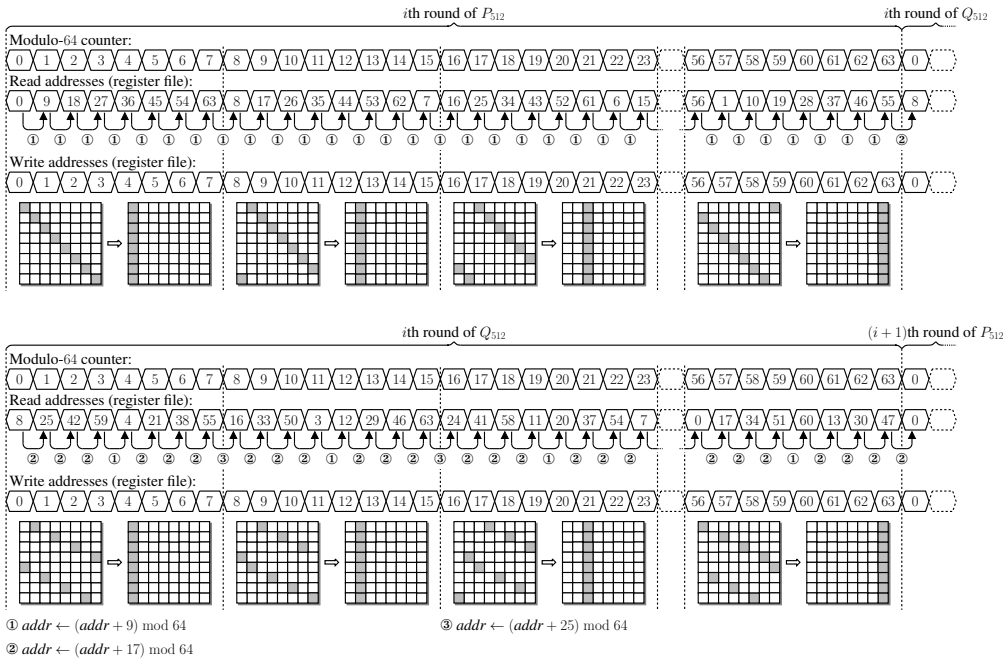


Figure 4.28: Address generation of P_{512} and Q_{512}.

The three most significant bits of read and write addresses select a block of 128 bytes in the memory, and their generation is quite straightforward. We

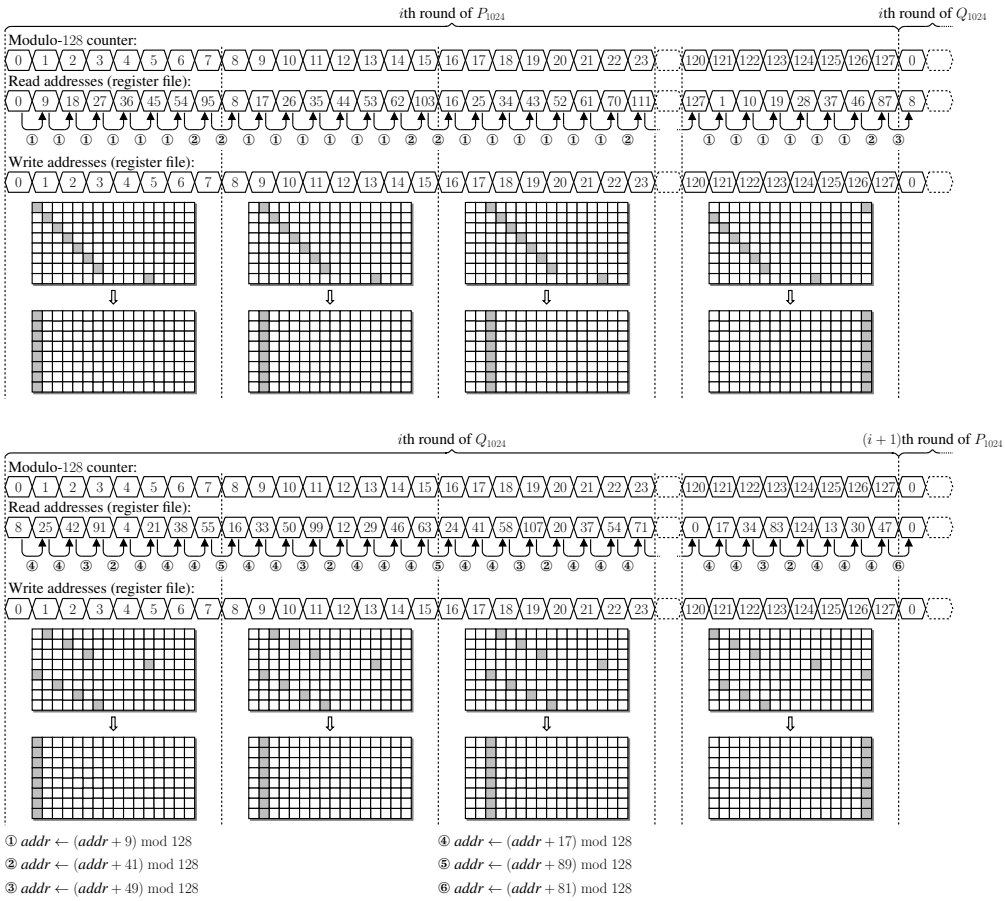


Figure 4.29: Address generation of P_{1024} and Q_{1024}.

refer the reader to our open source VHDL code and to [33] for further details. Therefore, we focus only on the generation of the seven least significant bits (*i.e.* the location of a byte in the internal state) in the following. Note that we

- interleave the computation of P_ℓ and Q_ℓ in order to avoid data dependencies between two consecutive rounds and
- implement the ShiftBytes step by accordingly addressing the register file.

Figure 4.28 and 4.29 summarize the address generation process of Grøstl-256 and Grøstl-512, respectively. At each clock cycle, a new read address is generated by adding a 7-bit offset to the current read address. The rules summarized in Table 4.12 allow one to compute the offset according to the permutation being executed. They involve the following signals:

- depending on the value of ℓ, *Counter* is a modulo-64 counter or a modulo-128 counter used to enumerate the bytes of the internal state (it sim-

ply consists of the six or seven least significant bits of the modulo-256 counter, and defines the write address of the register file);

- *Switch* is a 1-bit signal equal to one if and only if we are performing the last step of P_ℓ or Q_ℓ;
- Ω is a 1-bit flag indicating that the output transformation is carried out.

Table 4.12: Computation of the offset according to the permutation being executed.

	P_{512}	P_{1024}	Q_{512}	Q_{1024}
$Offset_6$	0	0	0	$Counter_2 \wedge Counter_1 \wedge Counter_0$
$Offset_5$	0	$Counter_2 \wedge Counter_1$	0	$(\neg Counter_2) \wedge Counter_1$
$Offset_4$	$(\neg\Omega) \wedge Switch$		$Counter_2 \vee \neg Counter_1 \vee (\neg Counter_0)$	
$Offset_3$	$\Omega \vee (\neg Switch)$		$Counter_1 \wedge Counter_0 \wedge (\neg Switch)$	
$Offset_{2:0}$	$0 \parallel 0 \parallel 1$			

Consider for instance Q_{1024} and assume that $Counter = (0010001)_2 = 17$. We compute the offset according to Table 4.12 and obtain:

$$\begin{aligned} Offset &= 0 \wedge 0 \wedge 1 \parallel (\neg 0) \wedge 0 \parallel 0 \vee (\neg 0) \vee (\neg 1) \parallel \\ &\quad 0 \wedge 1 \wedge (\neg 0) \parallel 0 \parallel 0 \parallel 1 \\ &= (0010001)_2 = 17. \end{aligned}$$

Since the current read address is equal to 41, the next read address is given by $(41 + Offset) \bmod 128 = 58$. *Counter* is now equal to $(0010010)_2 = 18$ and the new offset is given by:

$$\begin{aligned} Offset &= 0 \wedge 1 \wedge 0 \parallel (\neg 0) \wedge 1 \parallel 0 \vee (\neg 1) \vee (\neg 0) \parallel \\ &\quad 1 \wedge 0 \wedge (\neg 0) \parallel 0 \parallel 0 \parallel 1 \\ &= (0110001)_2 = 49. \end{aligned}$$

The computation of $Offset_4$ involves seven inputs: Ω, *Switch*, the three least significant bits of *Counter*, and two bits to define the permutation (P_{512}, Q_{512}, P_{512} or Q_{512}). On modern Xilinx devices, it is implemented by means of two 6-input Look-Up Tables (LUTs) and a dedicated multiplexer (F7AMUX or F7BMUX). Since $Offset_5$ and $Offset_6$ share the same five inputs, they are generated thanks to a LUT with two independent outputs (LUT6_2 primitive).

A fourth LUT allows us to compute $Offset_3$. Thus, we defined an extremely lightweight address generation process for Grøstl. It can easily be combined with the addressing scheme of the AES described in [33].

The output transformation requires special attention: since it involves only P_ℓ, five idle clock cycles between two consecutive rounds are mandatory to avoid memory collisions. Let us consider the ith round of Grøstl-256 to describe the problem (Figure 4.30). The control unit generates the address of $p_{7,6}$ (read operation) and $p_{7,7}$ (write operation) at time t. However, our coprocessor includes $D = 12$ pipeline stages and we write the new value of $p_{7,7}$ in the register file at time $t + D$. In order to update the first column of the internal state P, we have to read $p_{0,0}$, $p_{1,1}$, $p_{2,2}$, $p_{3,3}$, $p_{4,4}$, $p_{5,5}$, $p_{6,6}$, and $p_{7,7}$. The latter is available on port A of the register file at time $t + D + 1$, which means that $p_{0,0}$ can be read at time $t + D - 6 = t + 6$. It is therefore necessary to introduce five idle clock cycles between two rounds.

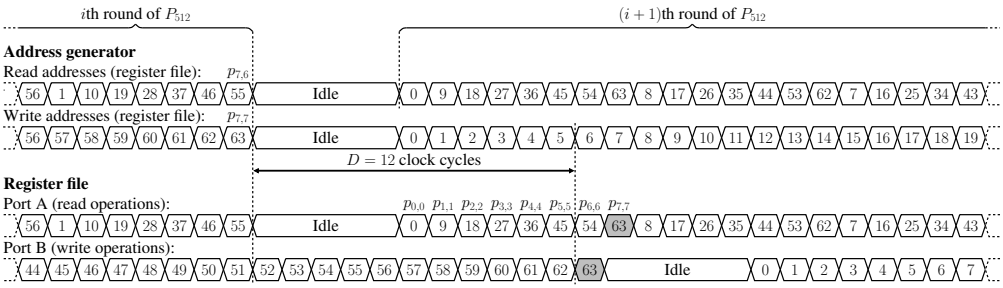


Figure 4.30: Latency between two consecutive rounds of P_{512} during the output transformation.

Table 4.13 summarizes the number of clock cycles required for the AES and Grøstl. In the case of the AES, we obtain exactly the same results as in [33]. Thanks to our careful organization of the memory, we achieve a perfectly tight scheduling (no idle cycle) for the compression function of Grøstl.

4.6.3.3 Arithmetic and logic unit

The SubBytes and InvSubBytes steps are often considered as the most critical part of the AES, and several architectures for S_{RD} and S_{RD}^{-1} have already been described in the open literature (see for instance [105] for a comprehensive bibliography). On Xilinx Virtex-6 FPGAs, the best design strategy consists in implementing the AES S-boxes as 8-input tables [59]. Two control bits $ctrl_{1:0}$ allow us to perform SubBytes, InvSubBytes, or to bypass this stage when g is the identity function.

Table 4.13: Number of clock cycles required for the AES and Grøstl.

Algorithm		# cycles
AES-128	Key expansion	365
	Encryption/decryption	231
AES-192	Key expansion	421
	Encryption/decryption	273
AES-256	Key expansion	476
	Encryption/decryption	315
Grøstl-256	Compression function	1411
	Output transformation	757
Grøstl-512	Compression function	3843
	Output transformation	1993

In the case of the AES, the first matrix multiplication of Equation (4.3) can involve any of the four circulant matrices defined in Section 4.6.1. Grøstl requires only $\mathcal{I}_{\text{Grøstl}}$ and $\mathcal{M}_{\text{Grøstl}}$. Let us define the control signal $\overline{\text{AES}}$/Grøstl whose role is to identify the algorithm being executed. Together with two control bits $ctrl_{3:2}$, this signal allows us to select matrix $\mathcal{A}$. The choice of matrix $\mathcal{B}$ turns out to be simpler and requires a single extra control bit $ctrl_6$ (Figure 4.26).

Since we emphasize reducing the usage of FPGA resources, we adopt the multiply-and-accumulate approach proposed by Hämäläinen *et al.* [106], and need N_l clock cycles to multiply one column of the state or the round key array by a circulant matrix (Figures 4.31 and 4.32). We compute a first partial product and store the result in registers r_0 to r_{N_l}. Then, at each clock cycle, the intermediate result is rotated and accumulated with a new partial product. This process involves a *Shift* control signal to distinguish between the first step and the subsequent ones. Such a signal can be generated by computing the bitwise OR of the bits of a modulo-N_l counter. Let us consider the three bits $c_{2:0}$ of a modulo 8 counter. Since $N_l = 4$ and $N_l = 8$ for AES and Grøstl, respectively, we define

$$\begin{aligned} \mathit{Shift} &\leftarrow \begin{cases} c_1 \vee c_0 & \text{if } \overline{\text{AES}}/\text{Grøstl} = 0, \\ c_2 \vee c_1 \vee c_0 & \text{if } \overline{\text{AES}}/\text{Grøstl} = 1 \end{cases} \\ &= (c_2 \wedge \overline{\text{AES}}/\text{Grøstl}) \vee c_1 \vee c_0. \end{aligned}$$

Note that the rotation mechanism depends on the algorithm being executed: the AES involves registers r_0, r_1, r_2, and r_3, whereas Grøstl requires eight

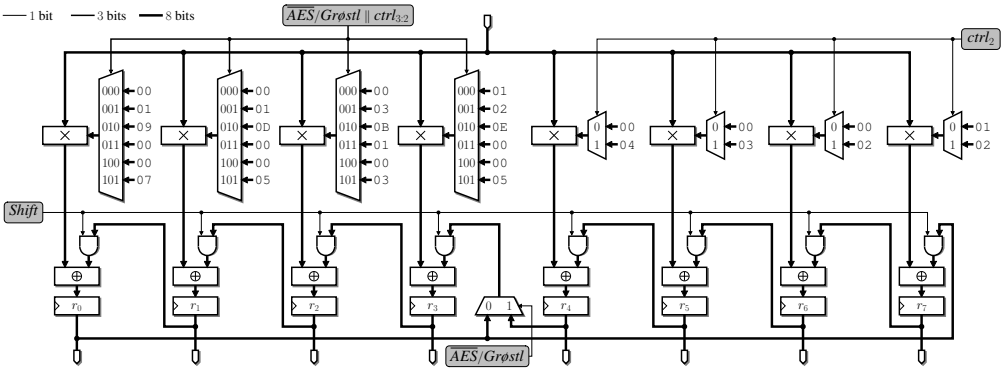


Figure 4.31: Implementation of MixColumns and MixBytes.

registers to store intermediate results. Therefore, the feedback mechanism is implemented by means of a multiplexer controlled by $\overline{\text{AES}}$/Grøstl. We describe in the following paragraph how to optimize the MixColumns and MixBytes steps on the latest Xilinx FPGAs.

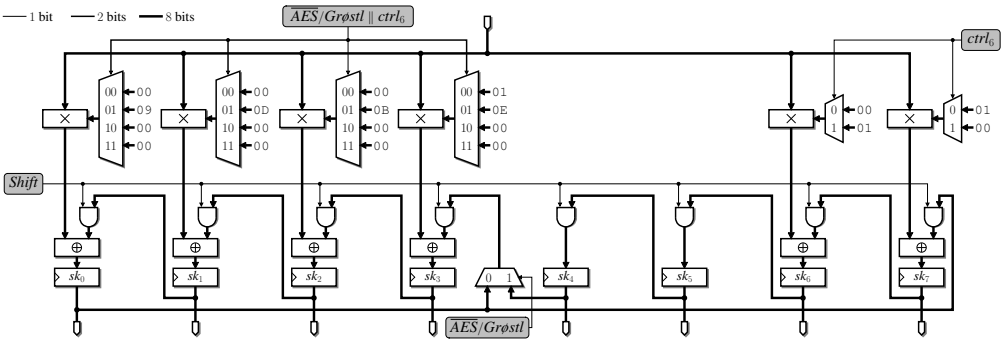


Figure 4.32: Multiplication by $\mathcal{I}_{\text{AES}}$, $\mathcal{M}_D$, $\mathcal{I}_{\text{Grøstl}}$, and $\mathcal{P}_{\text{Grøstl}}$.

Figure 4.33 describes the component we designed to perform the AddRoundKey and KeyExpansion steps. Since our matrix multiplication units output N_l bytes, we perform N_l additions over $\mathbb{F}_{2^8}$ in parallel and store the result in a shift register. Then, we write data byte by byte in the register file, and a modulo-N_l counter controls the process. Therefore, it suffices to delay our *Shift* signal by a total of N_l clock cycles, which is the latency of a matrix-vector multiplication. Additional hardware resources allow us to deal with the round constant RC involved in the key expansion of the AES (see [33] for details).

The last operation we have to consider is the AddRoundKey step of Grøstl. In order to compute $p'_{i,j}$ (Algorithm 8, line 12), we generate $k_{i,r\cdot N_c+j}$ on-the-fly. Recall that:

- i is a 3-bit row index;

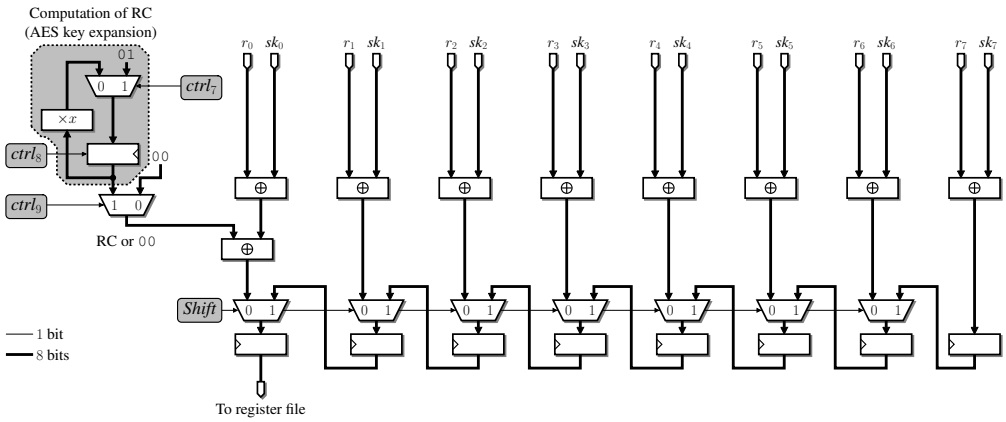


Figure 4.33: Implementation of AddRoundKey and KeyExpansion.

- j is a 4-bit column index;
- r is a 4-bit round index.

The indices i and j are given by the control signal $a_{26:20} = j \parallel i = 8j + i$ (Figure 4.26). According to Algorithm 7, we have:

$$k_{i,r \cdot N_c + j} \leftarrow \begin{cases} j \parallel r & \text{when } i = 0, \\ 00 & \text{otherwise.} \end{cases}$$

Since $i = a_{22:20}$ and $j = a_{26:23}$, we can rewrite the above equation as follows:

$$k_{i,r \cdot N_c + j} \leftarrow \begin{cases} a_{26:23} \parallel r & \text{when } a_{20} \vee a_{21} \vee a_{22} = 0, \\ 00 & \text{otherwise,} \end{cases}$$

and compute $k_{i,r \cdot N_c + j}$ by means of a 3-input NOR gate and eight 2-input AND gates. A multiplexer controlled by $ctrl_{5:4}$ allows us to inject $k_{i,r \cdot N_c + j}$, $\neg k_{i,r \cdot N_c + j}$, $k_{i,j} \oplus h_{i,j}$ (first key injection of P_ℓ), or a variable stored in the key memory.

4.6.3.4 Optimized implementation of MixColumns and MixBytes

Figure 4.34 describes how we take advantage of the LUT6_2 primitive in order to optimize MixColumns and MixBytes steps on the latest Xilinx FPGAs. The

first step of a multiplication by $\mathcal{M}_{\text{Grøstl}}$ is for instanced performed as follows:

$$
\begin{aligned}
r_0 &\leftarrow \texttt{09} \cdot a_0, \\
r_1 &\leftarrow \texttt{0C} \cdot a_0 + \texttt{01} \cdot a_0 = \texttt{0D} \cdot a_0, \\
r_2 &\leftarrow \texttt{09} \cdot a_0 + \texttt{02} \cdot a_0 = \texttt{0B} \cdot a_0, \\
r_3 &\leftarrow \texttt{0C} \cdot a_0 + \texttt{02} \cdot a_0 = \texttt{0E} \cdot a_0, \\
r_4 &\leftarrow \texttt{04} \cdot a_0, \\
r_5 &\leftarrow \texttt{03} \cdot a_0, \\
r_6 &\leftarrow \texttt{02} \cdot a_0, \\
r_7 &\leftarrow \texttt{02} \cdot a_0.
\end{aligned}
$$

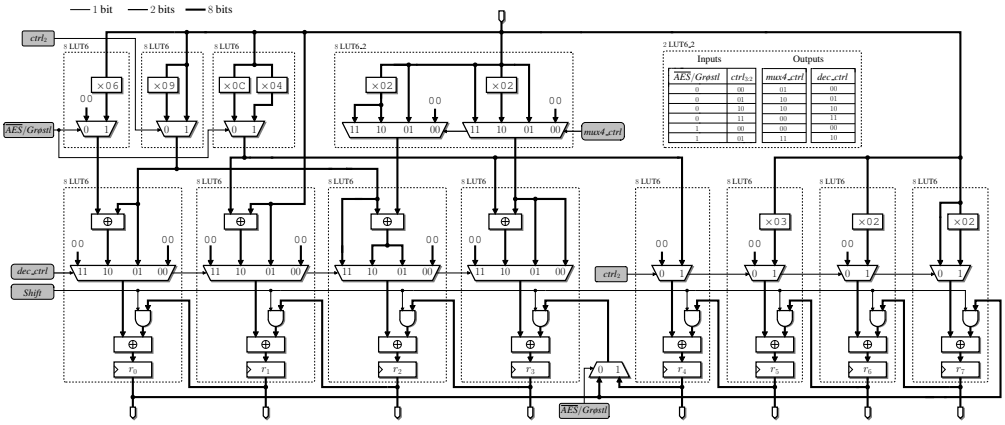


Figure 4.34: Implementation of MixColumns and MixBytes on the latest Xilinx FPGAs.

4.6.4 Results and comparisons

We captured our architecture in the VHDL language and prototyped our coprocessor on several Xilinx FPGAs with average speed-grade. Table 4.14 summarizes our place-and-route results measured with ISE 14.2. In order to evaluate the hardware overhead introduced by the AES, we designed a second coprocessor that implements only Grøstl-256 and Grostl-512 (Table 4.15). It is possible to reduce the number of slices by implementing a subset of the functionalities (*e.g.* a single level of security, AES without key expansion, etc.).

Our coprocessor requires a similar number of slices and achieves the same clock frequency as the architecture we designed for ECHO and AES [33]. However, the implementation of Grøstl on our architecture involves a smaller num-

Table 4.14: Place-and-route results for our unified coprocessor on Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs. The throughput of Grøstl is computed for a one-block message.

FPGA	Area [slices]	Frequency [MHz]	Throughput [Mbits/s]				
			AES-128	AES-192	AES-256	Grøstl-256	Grøstl-512
Virtex-6 (xc6vlx75t-2)	169	393	217	184	159	92	69
Artix-7 (xc7a100t-2)	188	265	146	124	107	62	46
Kintex-7 (xc7k70t-2)	172	438	242	205	177	103	76
Virtex-7 (7vx330t-2)	185	415	229	194	168	98	72

Table 4.15: Place-and-route results for our Grøstl coprocessor on Virtex-6, Artix-7, Kintex-7, and Virtex-7 FPGAs. The throughput is computed for a one-block message.

FPGA	Area [slices]	Frequency [MHz]	Throughput [Mbits/s]	
			Grøstl-256	Grøstl-512
Virtex-6 (xc6vlx75t-2)	102	413	97	72
Artix-7 (xc7a100t-2)	131	331	78	58
Kintex-7 (xc7k70t-2)	111	450	106	78
Virtex-7 (7vx330t-2)	108	480	113	84

ber of instructions and the throughput turns out to be slightly higher than the one of ECHO. Therefore, two conclusions we drew in our previous work can be transposed here:

- Helion Technology [107] is selling a tiny AES core that implements encryption, decryption, and key expansion at all levels of security. The coprocessor occupies only 88 Virtex-6 slices and achieves a throughput of 83 Mbps in the case of AES-128. Our unified coprocessor is almost twice as big, but we achieve a better encryption/decryption rate and improve the area–time product compared to the tiny AES core designed by Helion Technology. Thus, combining the hash function Grøstl with the AES does not impact the overall performance of the latter.

- The unified core for SHA-1, SHA-224/256, and SHA-384/512 designed by Helion Technology [98] turns out to be larger and slightly slower than our coprocessor. Furthermore, the Helion commercial core must be supplemented with an AES core to provide the same functionalities as our architecture. Assuming that the security of Grøstl is at least as good as the one of SHA-2, Grøstl is a clear winner for resource-constrained devices.

  We report in Table 4.16 the main features of four unified lightweight co-

processors able to hash and encrypt messages. Järvinen [99] proposed the first unified coprocessor for AES-128 (encryption and key expansion) and Grøstl-256. Recently, Rogawski & Gaj [100] designed a parallel coprocessor for Grøstl-based HMAC and AES in the counter mode. Both architectures are optimized for high-speed implementations, and it is therefore difficult to make a comparison with our unified coprocessor.

Table 4.16: Compact unified coprocessors to hash and encrypt messages. The throughput is computed for a one-block message.

	Supported algorithm(s)	**FPGA**	**Area [slices]**	**36k memory blocks**	**Frequency [MHz]**	**Throughput [Mbits/s]**
At *et al.* [78]	BLAKE-224, BLAKE-256 BLAKE-384, BLAKE-512, and ChaCha	xc6vlx75t-2	144	3	335	2×144 (BLAKE-224/256)
						255 (BLAKE-384/512)
						2×551 (8-round ChaCha)
						2×390 (12-round ChaCha)
						2×246 (20-round ChaCha)
Beuchat *et al.* [33]	ECHO-256, ECHO-512, AES-128, AES-192, and AES-256	xc6vlx75t-2	155	1	397	92 (ECHO-256)
						48 (ECHO-512)
						219 (AES-256)
						186 (AES-192)
						161 (AES-256)
This work	Grøstl-256, Grøstl-512, AES-128, AES-192, and AES-256	xc6vlx75t-2	169	1	393	92 (Grøstl-256)
						69 (Grøstl-512)
						217 (AES-128)
						184 (AES-192)
						159 (AES-256)
At *et al.* [78]	Skein-256-256, Skein-512-512, Threefish-256, Threefish-512, and Threefish-1024	xc6vlx75t-2	292	3	279	70 (Skein-256-256)
						80 (Skein-512-512)
						145 (Threefish-256 Enc.)
						166 (Threefish-512 Enc.)
						152 (Threefish-1024 Enc.)

Table 4.17 summarizes several lightweight FPGA implementations of hash functions (see for instance [108, 109] for a survey of parallel architectures). We consider here the least favorable case for Grøstl, in which a single block is processed. The throughput of Grøstl-256 is for instance given by:

$$throughput = \frac{512 \cdot \#blocks}{(1411 \cdot \#blocks + 757) \cdot T},$$

where T denotes the clock period. When the number of blocks increases, one can neglect the cost of the output transformation and the throughput tends asymptotically to $512/(1411 \cdot T)$.

Most of the architectures described in the open literature focus on a single level of security. In this context, BLAKE [80] is obviously the best choice for low-area implementations on FPGA. However, as soon as a circuit must support several levels of security, Grøstl will offer the most compact solution. Hence, Grøstl seems to be an excellent candidate for lightweight processors providing all levels of security.

Table 4.18 summarizes the results of unified resource-sharing hardware

Table 4.17: Compact implementations of the five SHA-3 finalists on Virtex-5 and Virtex-6 FPGAs. The throughput is computed for a one-block message.

	Supported algorithm(s)	FPGA	Area [slices]	36k memory blocks	Frequency [MHz]	Throughput [Mbits/s]
[80]	BLAKE-256	xc5vlx110	390	–	91	412
[57]*	BLAKE-256	xc6vlx75t-2	52	2	456	194
[93]	BLAKE-256	xc6v	235	–	231	518
[93]	BLAKE-256	xc6v	404	–	185	823
[94]	BLAKE-256	xc6vlx75t-1	163	1	197	327
[94]	BLAKE-256	xc6vlx75t-1	166	–	268	445
[80]	BLAKE-512	xc5vlx110	939	–	59	468
[57]*	BLAKE-512	xc6vlx75t-2	81	3	374	280
[89]	BLAKE-512	xc6vlx75t-1	192	–	240	183
[95]	BLAKE-256 and BLAKE-512	xc5vlx50-2	138	3	342	2×150 (BLAKE-256) 264 (BLAKE-512)
[96]†	Grøstl-256	xc5v	470	–	354	1132
[93]†	Grøstl-256	xc6v	328	–	365	1168
[94]	Grøstl-256	xc6vlx75t-1	241	1	244	115
[94]	Grøstl-256	xc6vlx75t-1	263	–	359	240
[110]†	Grøstl-256	xc6vlx75t-2	82	1	410	154
[89]†	Grøstl-512	xc6vlx75t-1	260	–	280	640
This work	Grøstl-256 and Grøstl-512	xc6vlx75t-2	102	1	413	97 (Grøstl-256) 72 (Grøstl-512)
This work	Grøstl-256, Grøstl-512, AES-128, AES-192, and AES-256	xc6vlx75t-2	169	1	393	92 (Grøstl-256) 69 (Grøstl-512)
[96]	JH-256	xc5v	205	–	341	27
[93]	JH-256	xc6v	193	–	385	29
[93]	JH-256	xc6v	424	–	365	1112
[94]	JH-256	xc6vlx75t-1	196	1	243	148
[94]	JH-256	xc6vlx75t-1	171	–	252	154
[89]	JH-512	xc6vlx75t-1	240	–	288	214
[93]	Keccak[$r = 1088, c = 512$]	xc6v	397	–	197	1071
[94]	Keccak[$r = 1088, c = 512$]	xc6vlx75t-1	129	1	260	74
[94]	Keccak[$r = 1088, c = 512$]	xc6vlx75t-1	106	–	299	135
[39]	Keccak[$r = 576, c = 1024$]	xc5vlx50-3	448	–	265	52
[89]	Keccak[$r = 576, c = 1024$]	xc6vlx75t-1	144	–	250	68
[75]	Keccak[$r = 576, c = 1024$]	xc5vlx50-2	151	3	520	501
[97]‡	Skein-256-256	xc5vlx110-3	821	Not specified	119	1610
[96]‡	Skein-512-256	xc5v	555	–	271	237
[93]‡	Skein-512-256	xc6v	406	–	318	277
[94]	Skein-512-256	xc6vlx75t-1	207	1	166	17
[94]	Skein-512-256	xc6vlx75t-1	193	–	193	21
[77]	Skein-512-512	xc6vlx75t-1	132	2	276	80
[89]‡	Skein-512-512	xc6vlx75t-1	240	–	160	179

*Modified to implement the tweaked version submitted for the final round of the SHA-3 competition.
†Without output transformation.
‡Single call to Threefish-512.

architectures of cryptographic hash functions and symmetric-key cryptographic algorithms (block/stream ciphers).

4.6.5 Conclusion

The design philosophy we proposed in [33] allowed us to develop a low-area coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function Grøstl at all levels of security. Our architecture is built around an 8-bit datapath and the ALU performs a single instruction that allows for implementing both algorithms. Despite the various addressing

Table 4.18: Unified Resource-Sharing Hardware Architectures of Cryptographic Hash Functions and Block/Stream Ciphers on FPGAs.

	Supported algorithm(s)	FPGA	Area [slices]	36k memory blocks	Frequency [MHz]	Throughput [Mbits/s]
Low-Area Architectures						
This work	Grøstl-256, Grøstl-512, AES-128†, AES-192†, and AES-256†	xc6vlx75t-2	169	1	393	92 (Grøstl-256)
						69 (Grøstl-512)
						217 (AES-128)
						184 (AES-192)
						159 (AES-256)
Beuchat *et al.* [33]	ECHO-256, ECHO-512, AES-128†, AES-192†, and AES-256†	xc6vlx75t-2	155	2-18k memory blocks	397	92.3 (ECHO-256)
						48.7 (ECHO-512)
						219.9 (AES-128)
						186.1 (AES-192)
						161.3 (AES-256)
At *et al.* [77]	Skein-512-512 and Threefish-512	xc6vlx75t-1	132	2	276	80 (Skein-512-512)
						160 (Threefish-512)
At *et al.* [78]	Skein-256-256, Skein-512-512, Threefish-256, Threefish-512, and Threefish-1024	xc6vlx75t-2	292	3	279	70 (Skein-256-256)
						80 (Skein-512-512)
						145 (Threefish-256 Enc.)
						166 (Threefish-512 Enc.)
						152 (Threefish-1024 Enc.)
At *et al.* [78]	BLAKE-256, BLAKE-512 and ChaCha	xc6vlx75t-2	144	3	335	2×144 (BLAKE-256)
						255 (BLAKE-512)
						2×551 (8-round ChaCha)
						2×390 (12-round ChaCha)
						2×246 (20-round ChaCha)
Balanced Architectures						
Järvinen [111]	Fugue-256 and AES-128†	Cyclone-III	4875⋆	0	59.8	957 (Fugue-256)
						383 (AES-128)
Järvinen [111]	Grøstl-256 and AES-128†	Cyclone-III	12520⋆	0	55.79	1428 (Grøstl-256)
						714 (AES-128)
High-Speed Architectures						
Rogawski and Gaj [112]	Grøstl and AES†	Cyclone-III	23758 LE⋆	0	144	2378 (HMAC-Grøstl-256 and (AES-128-CTR)
Rogawski *et al.* [113]	HMAC-Grøstl-256 and AES-128-CTR	Virtex-6	2447	0	255	376 (@40Bytes)
						4212 (@infinity)
Rogawski *et al.* [113]	HMAC-Grøstl-512 and AES-256-CTR	Virtex-6	5074	0	219	233 (@40Bytes)
						5215 (@infinity)

⋆Configurable logic area is measured as Logic Elements (LE) in Cyclone-III FPGA device.
†AES Key Expansion is included.
‡Single call to Threefish-512.

schemes required for the different steps of Grøstl and the AES, our control unit remains compact: all addresses are generated by means of a modulo-128 counter and a modulo-256 counter. Thanks to an alternative description of Grøstl and a meticulous organization of the memory, we manage to implement the compression function f (Algorithm 8) without any pipeline stall. The key element of our approach is to take advantage of the parallelism of Grøstl to deeply pipeline the ALU to achieve a high clock frequency and avoid data dependencies by interleaving independent tasks.

At the cost of 67 Virtex-6 slices, one can add the AES functionalities to a Grøstl coprocessor. Despite of the differences between the two algorithms (size of the internal state, coefficients of the circulant matrices, key schedule, etc.), resource sharing is possible. Assuming that the security guarantees of Grøstl are at least as good as the ones of the other algorithms reported in Tables 4.17 and 4.16, Grøstl is a valuable candidate for low-area cryptographic coprocessors.

Our architecture is mainly designed for embedded systems. Thus, it would be interesting to conduct side-channel and fault injection attacks in fu-

ture work. The generic countermeasures proposed by Güneysu and Moradi [114] could be applied to improve the security of our implementation. Furthermore, since the ALU executes the same instruction at each clock cycle, our design strategy could offer a protection against some attacks.

5. HIGH-PERFORMANCE CRYPTOGRAPHY ON RECONFIGURABLE HARDWARE

This chapter presents several new high-speed coprocessors for modular arithmetic operations and Paillier cryptosystem. First of all, coprocessors based on high-radix Montgomery algorithm for modular multiplication operation for various sizes are presented. The design details of this coprocessor are given with its architectural considerations. Then, via using this coprocessor, Montgomery based modular exponentiation coprocessors are developed for various sizes. These kinds of operations over a modulo is extensively used in many different algorithms, especially in public-key cryptography. For instance, Paillier cryptosystem, which in fact ensures privacy requirements in privacy-preserving applications, involves modular multiplications and exponentiation operations of long numbers in size. Finally, a high-speed coprocessor for the Paillier cryptosystem is demonstrated using the proposed very high-radix Montgomery based modular arithmetic coprocessors.

5.1 Introduction and Motivation

Security protocols such as IPSec, SSL and VPNs used in many communication systems employ various cryptographic algorithms in order to protect the data from malicious attacks. Thanks to public-key cryptography, a public channel which is exposed to security risks can be used for secure communication in such protocols without needing to agree on a shared key at the beginning of the communication. Public-key cryptosystems such as RSA, Rabin and ElGamal cryptosystems are used for various security services such as key exchange and key distribution between communicating nodes and many authentication protocols. Such public-key cryptosystems usually depend on modular arithmetic operations including modular multiplication and exponentiation. These mathematical operations are computationally intensive and fundamental arithmetic operations which are intensively used in many fields including cryptography, number theory, finite field arithmetic, and so on. This paper is devoted to the analysis of modular arithmetic operations and the improvement of the computation of modular multiplication and exponentiation from hardware design perspective based on FPGA. Two of the well-known algorithms namely

Montgomery modular multiplication and Karatsuba algorithms are exploited together within our high-speed pipelined hardware architecture. Our proposed design presents an efficient solution for a range of applications where area and performance are both important. The proposed coprocessor offers scalability which means that it supports different security levels with a cost of performance. We also build a system-on-chip design using Xilinx's latest Zynq-7000 family extensible processing platform to show how our proposed design improve the processing time of modular arithmetic operations for embedded systems.

Due to increasing popularity of protecting confidentiality, privacy-preserving collaborative filtering (PPCF) has been receiving increasing attention. In order to protect confidential data while estimating recommendations, various methods have been proposed. Cryptographic techniques, especially with homomorphic encryption property, are widely used popular methods for data protection in collaborative filtering schemes with privacy. Such schemes aim to provide accurate recommendations efficiently with privacy. However, since privacy and performance are two conflicting goals, efficiency becomes worse due to the utilized homomorphic encryption methods. Although various approaches like performing some computations off-line, utilizing preprocessing, employing hybrid algorithms, and so on can be used to enhance online performance, the improvements are still limited and alternative approaches should be used.

In this part of the dissertation, we aim to propose an efficient hardware architecture for the Paillier cryptosystem, which incorporates high radix Montgomery multiplication with Karatsuba algorithm. To the best of our knowledge, this is the first attempt for hardware implementation of the Paillier cryptosystem to improve the efficiency of a data protection scheme. In the architecture that we planned to develop, we harness the intrinsic parallelism of modular multiplication and exponentiation to design a pipelined ALU and interleave several tasks in order to achieve a tight scheduling.

In one hand, PPCF schemes should provide recommendations without violating data owners' confidentiality. On the other hand, they should estimate predictions with decent online performance. As stated previously, efficiency and privacy are two conflicting goals. In other words, enhancing one of them makes the other worse. PPCF schemes should be able to provide predictions to many users during online interactions. Unlike other prediction systems, online requirements are very critical for the overall success of PPCF schemes. Hence, various software oriented approaches have been proposed to improve

online performance or eliminate the overheads due to homomorphic encryption (HE) schemes. Such approaches include dimension reduction, preprocessing, performing some computations off-line, utilizing hybrid CF algorithms, and so on [23]. Even if such methods are used for enhanced performance, their effects on online efficiency are limited.

In addition to software oriented schemes, special hardware components created for HE schemes can be used to make efficiency better. In this section, we will take the lead in that direction. In general, there are three ways to implement any design including the Paillier cryptosystem: software, ASIC, and FPGA. The software implementation is flexible but its computational complexity can be high at times; moreover, storing the private key in the computer memory will endanger the security of the system. The ASIC implementation is faster and more secure than the software implementation. However, the ASIC implementation is not flexible enough and its longer design cycle and higher development cost make the ASIC implementation less attractive and not preferable choice to build a cryptosystem. On the other hand, FPGA offers flexible, powerful, and reliable hardware platform. Xilinx Virtex family FPGA devices, for example, have several dual ported, highly scalable embedded Block RAMs (BRAM), DSP blocks (DSP48E1), and configurable logic blocks.

In this chapter, we mainly focus on the compact hardware implementation of the modular multiplication for Paillier cryptosystem. Specifically, we aim to propose combining Montgomery and Karatsuba algorithms. Our planned design uses embedded multiplier and adder units available on FPGAs. The proposed architecture presents an alternative approach to enhance the efficiency of PPCF schemes.

5.2 Related Work

There are many studies about efficient implementations of modular arithmetic operations included multiplication and exponentiation using various techniques to decrease the computation of these operations, especially in public key cryptography. Montgomery [24] and Karatsuba [25, 43] multiplication algorithms are the most efficient and popular algorithms.

High-radix algorithms [5, 115] have been proposed for improving the performance. However, as the radix increases, the design complexity and the length of clock cycle also increase dramatically due to requiring the use of

larger digit multipliers. These high-radix designs generally consumes huge amounts of hardware area. So, previously, low-radix designs are more attractive for hardware implementation due to the lack of larger digit multipliers. However, dedicated 17-bit embedded multipliers can easily be found in almost all FPGAs today and implementing larger than 17-bit multiplier is also possible with Karatsuba algorithm. By efficiently exploiting Karatsuba algorithm, one can achieve a larger bit multiplier by using less multiplier units. By using Karatsuba algorithm, doubling the bit-width of the multiplier is achieved in less amount of multiplier compared to classical multiplication. Employing Karatsuba algorithm in a pipelined mode allows to perform high-radix Montgomery modular multiplication based on these Karatsuba coprocessors within much less clock cycles. Our aim is to find a compact trade-off between the computation time and the required hardware area by efficiently exploiting Montgomery and Karatsuba algorithms together. Analysis of the design trade-offs for high-radix modular multipliers is found in [116].

Tenca and Koç proposed a radix-2 scalable Montgomery multiplication architecture [117] which multiplicand is scanned word-by-word and the other operand, multiplier, is scanned bit-by-bit. This multiple word radix-2 Montgomery multiplication allows efficient scalable hardware implementation. Parallelism among instructions of the algorithm in different scanning bits of multiplier is possible. The main difference and advantage of the architecture compared to the other algorithms in the literature is its scalability to any operand size which in fact enables to find good design trade-offs for different application needs. Tenca *et al.* [118] describe a scalable Montgomery multiplier which is adjustable to constrained areas and capable to work on any given precision of the operands. The algorithm proposed in [118] uses Booth encoding technique and a radix-8 scalable multiplier is implemented to show the performance of the design. In contrast to [118], our proposed hardware architectures operate with very high-radix values to reveal the computational efficiency of very high-radix changes in the modular multiplication and exponentiation.

The study [119] proposes an automatic generation of modular multipliers especially for FPGA based systems. Beuchat and Muller [119] present an algorithm which can select best possible high-radix carry-save representation for predetermined modulus. A synthesizable VHDL code for given parameters is automatically generated by the method proposed in [119].

Multiplication of large numbers is an important arithmetic operation in polynomial multiplication for signal processing, coding theory [120]. For a

comprehensive survey of different algorithms to perform polynomial multiplication, we refer the reader two studies Bernstein [121] and Nedjah *et al.* [122]. In order to evaluate the complexity of an algorithm, computational efficiency, memory requirement and resource sharing offered by the algorithm have great importance. There are many other studies on the implementations of large digit multiplication operation using various methods. We select Karatsuba multiplication algorithm [43] since it is well-suited to FPGA based environment.

Dinechin *et al.* [123] aim to build large multiplier by using fewer DSP48E blocks. The proposed architectures in [123] achieve very high frequencies since the design consumes few additional logic that have little impact on the critical path of the architecture. The presented architectures [123] in for different bit width multiplication differs from each other. However, our architecture is same for different multiplication sizes. Our architecture is generic and scalable. In addition, our coprocessor efficiently exploits DSP48E1 blocks and requires a small amount of logic. Suzuki [124] presents a method that can exhibit the the maximum performance of available FPGA resources for modular exponentiation operation. The study exploits the embedded building blocks including DSP48E, Block RAM and SRL16 to maximize the utilization of FPGA resources. Due to the need to the store preliminary input X_{2i+1} in [124], Block RAM utilization is required and the number of Block RAM changes with different parameters of the design. Gao *et al.* [125] proposes FPGA-based efficient large size signed multipliers using multi-granular small embedded blocks to be used in the applications such as scientific computation, cryptography, and data intensive systems. Our proposed method differs from this study since our proposed hardware architecture employs a powerful pipelining mechanism with a very tight scheduling using DSP48E1 blocks and a small amount of configurable logic slices. We compare this study with our coprocessor results in section 5.3.4. Chow *et al.* [29] presented a Montgomery multiplier that exploits Karatsuba algorithm for cryptography applications. The design uses multiple precision arithmetic techniques in order to make the critical path delay independent to bit width of the multiplier. They recursively implement Karatsuba multiplier to achieve n-bit multiplier to use in Montgomery algorithm in one pass. In contrast to that study, we implement pipelined Karatsuba multiplier in an iterative manner together with very high-radix Montgomery multiplier in this study.

Bo *et al.* [126] presents a hardware architecture for RSA encryption based

on Montgomery modular multiplication. They propose an hardware architecture which is able to perform necessary operations for the RSA encryption using only one DSP48E1, one Block RAM and a few slices. Moreover, the DSP48E1 block works almost full which yields an efficient solution for low-cost applications. Their proposed architecture is well-suited to many applications where resources are constrained. In [127], Bo *et al.* aims to accelerate the architecture presented in [126] using CRT technique. There is another recent study in designing coprocessor for Modular arithmetic operations. Bautista *et al.* [128] recently proposes a mathematical coprocessor which is able to perform modular arithmetic operations based on FPGA. Their proposed design is based on Montgomery algorithm for modular multiplication.

Our motivation of this study is to find a good trade-off between the area and the computation time of modular multiplication and exponentiation operations for hardware implementations. For this purpose, we propose a pipelined method which utilizes both Karatsuba and Montgomery modular multiplication algorithms by using embedded DSP blocks in FPGA to obtain efficient hardware design. We provide the performance figures of our design in Tables 5.1, 5.2 and 5.4. The comparisons with respect to other reported studies in the literature are given in Table 5.3.

There are various studies focusing on homomorphic cryptosystems (HC) from introduction of "privacy homomorphism" notion by Rivest et al. [129] to Gentry's fully HC proposal [130]. Well-known asymmetric key encryption methods like RSA and ElGamal have multiplicative homomorphic property [131]. In other words, ciphertext of the actual product can be obtained after the multiplication of two encrypted numbers with the same key using such schemes. Paillier [41] proposes a cryptosystem based on probabilistic public key encryption infrastructures, which are firstly considered for privacy homomorphisms by Goldwasser and Micali [132].

Through a bunch of HCs, the one proposed by Paillier [41] takes so much attention in the context of PPAs [133–136]. Paillier [41] proposes an additive homomorphic encryption method with self-blinding property. By means of the specified method, ciphertexts can be processed to get encrypted version of their sums. From this addition property, multiplication operations can be performed between a ciphertext and a plaintext. Moreover, self-blinding property allows mapping a plaintext into possibly many different ciphertexts. By the way, the same plaintexts cannot be recognized from their ciphertexts. While the PHC provides verifiable correctness and trustworthiness without revealing informa-

tion about the confidential data, its foremost shortcoming is its computational cost. For example, according to the experimental evaluation of the PHC in terms of privacy-preserving recommenders in [133], the authors point out the need for some performance amendments. Other typical applications of the PHC are secure e-voting [134] and e-auction [135] schemes in which protecting the voters' and auctioneers' privacy is the main goal. Parkes et al. [135] propose a practical protocol based on homomorphic cryptography for conducting provably fair sealed-bid auctions. In order to perform their privacy-preserving auction protocol in satisfactory performance, they draw attention to possible efficient hardware-oriented implementations.

There is an extensive literature on efficient implementations of modular arithmetic operations including modular multiplication, inversion, and exponentiation [5, 24, 137]. Montgomery [24] multiplication has been one of the most popular algorithms; and it still constitutes a base for newly proposed schemes. It also operates with long integers to be represented by a radix (generally power of two) [138]. Moreover, high-radix Montgomery reduction method is well-suited to FPGA-based hardware environments due to the embedded building multiplier modules available in Xilinx FPGAs [5]. Various high-radix Montgomery modular exponentiation implementations have been proposed in the open literature for improving the performance [5, 126]. However, as the radix increases, the design complexity and the length of clock cycle also increase dramatically due to the use of larger digit multipliers. These high-radix designs generally consume huge amounts of hardware area. Thus, low-radix designs are more attractive for hardware implementations. However, dedicated 17-bit embedded multipliers can easily be found in almost all FPGAs today; and implementing larger than 17-bit pipelined multipliers is also possible with using a few DSP blocks. Analysis of the design trade-offs for high-radix modular multipliers can be found in [139].

In order to respond the privacy requirements of PPAs within a reasonable time, the computational complexity of cryptographic algorithms, which provide privacy to such applications, needs to be reduced. Such complexity can be improved by means of an application specific hardware design. Using such secure hardware designs, Bhattacharjee et al. [140] describe a collaborative data mining and analysis solution for enterprisers having confidential data to each other. The security of their scheme is based on IBM PCIXCC secure coprocessors as main processing units on encrypted data belong to a number of parties. Similar work is done by Li [141] in which solutions are proposed to

join various autonomous databases in a privacy-preserving manner via secure coprocessors. The proposed scheme allows trusted third party to access secure coprocessor operations and the trust notion is consolidated through a secure hardware usage. Smith and Safford [142] discuss the feasibility of privacy protection with secure coprocessors and describe a model consisting of a single server with encrypted records and a number of coprocessors enabling database queries with privacy.

Hardware-based approaches could be preferred to improve the performance of computational and/or data intensive applications. Mueller [143] shows how data intensive operations can be accelerated by multi-core systems via FPGAs. The authors in [144] cope with the calculation of frequent items in a data collection and demonstrate how it can be implemented using FPGAs. They obtain a throughput rate four times higher than the software-based implementations. Wen et al. [145] enhance hardware-based association rule mining schemes with hashing and pipelining. Sawada and Nishi [146] propose an FPGA-based hardware architecture including special type of a memory and a cache mechanism for privacy-preserving data publishing methods based on both k-anonymity and l-diversity notions. Their implementation improves the performance up to 50 times faster than a conventional RAM-based architecture.

Rather than the above solutions utilizing commercially available secure coprocessors [140–142], we design high-speed coprocessor architecture in order to perform computational intensive homomorphic operations exploited by numerous PPAs. We also employ high-radix Montgomery algorithm, which incorporates the pipelined 32-bit multiplier modules, in order to use hard blocks in FPGAs. Moreover, the privacy of our scheme is based on security achieved by HCs rather than secure coprocessors and we propose a new hardware architecture to increase the efficiency of PPAs. Similar to the schemes proposed in [144,145], we focus on improving the performance of source exploiting tasks; however, our scheme additionally concentrates on how efficient solutions can be provided through privacy-preserving dimension as in [146]. Our proposal also differs from the work [146] because we focus on a cryptographic mechanism rather than anonymity-based approaches. The motivation of the study is to achieve an high-speed and high-throughput design of modular multiplication and exponentiation required by the PHC to come up an efficient design for PPAs.

5.3 Improving the Computational Efficiency of Modular Operations for Embedded Systems

Security protocols such as IPSec, SSL and VPNs used in many communication systems employ various cryptographic algorithms in order to protect the data from malicious attacks. Thanks to public-key cryptography, a public channel which is exposed to security risks can be used for secure communication in such protocols without needing to agree on a shared key at the beginning of the communication. Public-key cryptosystems such as RSA, Rabin and ElGamal cryptosystems are used for various security services such as key exchange and key distribution between communicating nodes and many authentication protocols. Such public-key cryptosystems usually depend on modular arithmetic operations including modular multiplication and exponentiation. These mathematical operations are computationally intensive and fundamental arithmetic operations which are intensively used in many fields including cryptography, number theory, finite field arithmetic, and so on. This paper is devoted to the analysis of modular arithmetic operations and the improvement of the computation of modular multiplication and exponentiation from hardware design perspective based on FPGA. Two of the well-known algorithms namely Montgomery modular multiplication and Karatsuba algorithms are exploited together within our high-speed pipelined hardware architecture. Our proposed design presents an efficient solution for a range of applications where area and performance are both important. The proposed coprocessor offers scalability which means that it supports different security levels with a cost of performance. We also build a system-on-chip design using Xilinx's latest Zynq-7000 family extensible processing platform to show how our proposed design improve the processing time of modular arithmetic operations for embedded systems. The results presented in this section have been published [147].

5.3.1 Introduction

With the advancement of communication technology and information systems, networking and data streaming applications call for higher data rates as well as security at the same time. There are many such emerging applications especially for embedded systems that need to communicate, store or manipulate confidential data. This makes security is a primary concern in the design of embedded systems for certain applications. Security is provided by the set

of cryptographic algorithms which are usually computationally intensive algorithms. Hence, embedded security is a constantly developing field along with the research on efficient hardware design of cryptographic algorithms. To meet security needs of such applications, there are several security protocols such as IPSec, SSL and VPNs which are used between a pair of communicating hosts called sender and recipient. Further, with the introduction of Public-Key Cryptography (PKC), a public channel can be used for secure communication. PKC have also many benefits in such protocols such as key-distribution and various authentication protocols due to the fact that PKC does not require a secure initial key exchange between the sender and the recipient.

Most of the public-key cryptosystems use modular arithmetic operations, specifically, modular multiplication and exponentiation. These two operations are also widely used in other fields. Hence, their efficient computation is quite important for the security of embedded systems. There are many different algorithms and computation models in the literature to perform modular multiplication such as Montgomery, Booths, Karatsuba, etc. Moreover, repetitive use of modular multiplication is needed in the modular exponentiation operation. Therefore, efficient design of modular multiplication has a great importance in the computational efficiency of modular exponentiation.

In this study, we focus on high-speed pipelined hardware implementation of the modular multiplication and exponentiation operations on FPGAs. In [148], San and At designed a compact coprocessor which efficiently exploits intrinsic properties of Karatsuba algorithm on Xilinx Virtex-5 FPGA devices. Here, in particular, the design strategy in [148] is improved in terms of latency. Improved pipelined version of Karatsuba coprocessors employed in the proposed architecture in this study. Hence, a pipelined hardware implementation for the modular multiplication and exponentiation operations using both Karatsuba and Montgomery algorithms is proposed.

Today's FPGAs have enhanced hard blocks such as DSP and Block RAM components yielding very high operating frequencies. As emphasized by Güneysu, "the use of device specific components lead to considerably higher system performance" [42]. With this approach in mind, we have adapted Karatsuba and high-radix Montgomery modular multiplication algorithms to efficiently compute the modular multiplication on FPGA using device specific components such as DSP and shift register components. In order to extend arithmetic precision to be used in larger modulus and make modular multiplication algorithm more compact for larger bits, Karatsuba and Montgomery

algorithms can be used by exploiting the embedding multiplier modules existing in almost all FPGAs in pipelined mode. In this article, we first efficiently combine very high-radix Montgomery and Karatsuba algorithms for performing modular multiplication algorithm using these embedded hard cores on latest high-speed FPGAs. This gives us very powerful results in those cases where the parallel and pipelined processing is available. Then, we implement compact modular exponentiation architecture using the proposed high-speed modular multiplication blocks. We provide hardware performance figures of our methods on FPGA in terms of latency, area, and frequency. We achieve efficient high-radix modular multiplications by means of pipelined Karatsuba coprocessor. The proposed hardware uses efficiently Karatsuba algorithm to multiply large numbers in a compact manner with saving multiplier blocks. We also analyze the effects of radix changes on our hardware architecture. The comparison of our hardware performance results with the results of other reported hardware architectures for modular multiplication and exponentiation is also given. The results show that the architectures proposed in this study for modular multiplication and exponentiation yield good performance with a reasonable area in hardware for embedded systems where FPGA comes into play.

The main contribution of this paper is to present a high-speed hardware architecture for modular multiplication and exponentiation using both Karatsuba and Montgomery modular multiplication algorithms with the presented design methods using embedded building blocks on latest Xilinx FPGA. The proposed method efficiently exploit embedded multipliers and adder units on FPGA. The presented architecture aims to provide an efficient architecture to be used where modular arithmetic is definitely required such as Coding theory, Cryptography, especially PKC, DSP and so on. There are many extensive studies in the literature related to increase the efficiency of multiplication [5, 123, 126, 149, 150], especially in PKC. From our point of view, the main advantage of this method over other existing methods is that iterative utilization of hardware resources with pipelining and tight scheduling brings better performance with smaller logic area. Our method also attains very high frequency. The other advantage of our architecture is its scalability which respect to the operand size.

Two levels of scalability are considered:

- The design should require hardware resources as small as possible.
- The design allows multiplication on larger size on the same architecture

with small modifications.

In this study, we also propose a system-on-chip (SoC) design which uses our efficient modular arithmetic hardware architecture as a coprocessor with Xilinx's latest Zynq-7000 family extensible processing platform. We adapt Hardware/Software codesign approach in order to exploit the advantages of both methodologies. The design rationale of the proposed architecture is to accelerate the performance of computational intensive tasks required by the modular arithmetic operations. Furthermore, the low latency presented by the coprocessor empowers to operate in higher frequencies within the SoC platforms.

5.3.2 Modular arithmetic coprocessor

In this study, our main aim is to propose a high-speed and efficient coprocessor in order to perform modular arithmetic operations efficiently including multiplication and exponentiation. The proposed coprocessor is able to meet the requirements of embedded systems in terms of area and performance. Performing these operations in hardware domain accelerate overall system performance of the application which includes modular arithmetic operations.

There are several design approaches that can be applied to cryptographic algorithms for an efficient hardware implementation such as a low-area coprocessor design approach [76], a compact coprocessor design approach [148], high-speed architecture design approach [42] and other design approaches. The most important optimization goals for the coprocessor proposed in this study are to achieve high frequency values for high-speed performance and to utilize the pipelining technique for increasing the throughput. The description of our design strategies for our hardware architecture described in detail in the following sections to show how the coprocessor achieve the demanding constraints of modular arithmetic operations.

5.3.2.1 Design rationale

Modular exponentiation operation in nature consists of performing repetitive modular multiplications. If the size of the modulus is increased, the required number of multiplications are also increased. These loop based repetitions of the multiplication are computed in sequential order. Such iterative modular multiplication operations need to be computed very efficiently to achieve higher performance results. Unfortunately, performing modular multiplication

operation using radix-2 Montgomery reduction requires a large amount of time with the advantage of consuming a small amount of area. For the applications where the processing time is more important than the required hardware area, high-radix Montgomery reduction is used. However, high-radix Montgomery reduction requires high-radix multipliers which consumes more area and more latency with respect to lower radix sizes. In this study, our aim is to find good trade-off for high-performance modular arithmetic operations for embedded systems. We look for the various design trade-offs by implementing our hardware architecture in several high-radix sizes. We take advantage of the intrinsic parallelism in the Montgomery high-radix modular multiplication and Karatsuba algorithms by efficiently exploiting embedded DSP building blocks of FPGAs. In summary, our design philosophy for this study consists of following design methods:

- *Pipelining*: we propose a pipelined ALU to achieve maximum clock frequency. We separate the necessary computation into a set of equal stages. We balance each stages by utilizing DSP48E1 blocks in order to achieve high frequency values. Each stage in the ALU processes one part of the algorithm. Our pipelined ALU allows a parallel execution of different parts of the algorithm at the same time. Hence, the stages of our pipelined ALU operates simultaneously using different resources which increases the throughput of the system.

- *Interleaving*: we interleave the independent operations performing modular multiplication algorithm that we used in this study. This allows us to obtain a tight scheduling which increases our throughput substantially.

- *Reusing*: we reuse a modular multiplication component for a series of modular multiplication operation, which cannot be parallelized, in modular exponentiation operation to decrease the resource utilization.

The design philosophy that we follow in this study allows us to develop a scalable and high-speed coprocessor for the modular arithmetic operations. All the presented design strategies significantly improve the performance of computationally intensive modular arithmetic operations.

5.3.2.2 Large-digit multiplier based on Karatsuba algorithm

Here, we adapt the Karatsuba algorithm for DSP block in Xilinx FPGA. There is an overhead of long additions required by Karatsuba algorithm which in

fact decreases the critical path of hardware implementations. However, our hardware computes these long additions sequentially. Thus, it does not cause to decrease in frequency for our proposed coprocessor. The pseudo code of our adapted Karatsuba algorithm for FPGA which is given in Algorithm 10.

Algorithm 10 Adapted Karatsuba Algorithm by using DSP blocks in Xilinx FPGA

Require: n : size of the Karatsuba algorithm
$X \in \{0, 1, \ldots, 2^n - 1\}$, $Y \in \{0, 1, \ldots, 2^n - 1\}$,
Ensure: $Z = Karatsuba(X, Y)$, $Z \in \{0, 1, \ldots, 2^{2 \cdot n} - 1\}$

1. **if** $n = 17$ **then**
2. $Z \leftarrow X \cdot Y$; (17-bit Embedded Multiplier)
3. **else**
4. $M_\alpha \leftarrow Karatsuba(X_0, Y_0)$; (Parallel call 1)
5. $M_\beta \leftarrow Karatsuba(X_1, Y_1)$; (Parallel call 2)
6. $M_\gamma \leftarrow Karatsuba(X_0 - X_1, Y_1 - Y_0)$; (Parallel call 3)
7. $Z\left[\frac{n}{2} - 1 : 0\right] \leftarrow M_\alpha\left[\frac{n}{2} - 1 : 0\right]$; (Least)
8. $S_\alpha \leftarrow \left(M_\alpha \gg \frac{n}{2}\right) + M_\alpha + M_\beta + M_\gamma$;
9. $Z\left[n - 1 : \frac{n}{2}\right] \leftarrow S_\alpha\left[\frac{n}{2} - 1 : 0\right]$; (Middle)
10. $Z\left[2 \cdot n - 1 : n\right] \leftarrow \left(S_\alpha \gg \frac{n}{2}\right) + M_\beta$; (Most)
11. **end if**
12. **return** Z;

This short description of Karatsuba algorithm gives some insights on designing a dedicated coprocessor. San and At [148] designed a compact coprocessor which efficiently exploits intrinsic properties of Karatsuba algorithm on Xilinx Virtex-5 FPGA devices. The design strategy is based on coprocessor approach. The architecture in [148] consists of a register file implemented by means of dual-ported memory, an ALU, and a control unit. The register file is organized into 34-bit words, and stores multiplication operands. In this paper, move one step forward and propose a improved version in terms of latency. In this study, we extended it to come up with an high-performance hardware architecture by exploiting the pipelining mechanism. This allows us to use in high-radix Montgomery modular multiplication algorithm to increase the computational efficiency of modular arithmetic. Presented pipelined hardware implementing long integer multiplication consumes a few amount of DSP blocks existing in FPGAs. It exploits 17-bit multipliers and 48-bit adder units in DSP blocks to compute the multiplication of high-radix integers in parallel. The pipeline mechanism that we add to our Karatsuba coprocessor [148] increases the clock frequency and aids to achieve high throughput.

5.3.2.3 High-speed modular multiplication and exponentiation

The architecture of modular multiplication component has a great impact on the performance of the coprocessor since modular exponentiation consists of a series of modular multiplications. When the design techniques which are presented in Section 5.4.2.1 are applied to high-radix Montgomery modular multiplication algorithm, we come up with an Algorithm 11 to enhance the computation of the PKC. Algorithm 11 gives our description of radix-2^w Montgomery modular multiplication on Virtex-7 FPGA. We show our pipelining mechanism in time-sliced fashion in the lines of Algorithm 11. Each line of the algorithm is responsible for different stages of the pipeline so that each of them can be executed in parallel.

In order to increase the performance of modular exponentiation, we interleave the q_i computation (Algorithm 11, lines 3 and 6) and the computation of next value of S_{i+1} (Algorithm 11, lines 12 and 18). This approach allows us to generate the q_i values on-the-fly. Initially, it is necessary to compute the first q_0 content before the first computation of S_1. It is mandatory since S_1 depends on q_0. We manage to perform this operation during loading the input data to the input shift registers. After initial computing of q_0 content, the q_i computation is synchronized with the S_{i+1} computation.

5.3.2.4 Arithmetic and logic unit of the coprocessor

Our architecture is mainly built around a w-bit datapath. The ALU performs all the operations required by the algorithms of interest. In Figure 5.1, we illustrate our pipelined ALU for the high-radix modular multiplication architecture, $ModMult$ function given in Algorithm 11. It requires to design high-radix multipliers in the datapath. On FPGAs, the best design strategy consists in implementing this multiplier by means of DSP48E1 blocks. The control signals allow us to perform the operations defined in the Algorithm 11 very efficiently. Our ALU has several pipeline units to improve the performance of the computation of modular multiplication operation. We manage to keep these pipeline units continuously busy without any pipeline stall. Hence, we substantially improve the performance of the modular operations. Thanks to these pipeline mechanism, data is supplied by FIFO elements which are realized by SRL32 primitives. These FIFO elements for input and output are illustrated in Figure 5.1. In this figure, Ri denotes a w-bit register and they stores the output of high-radix multipliers to compute the next value of S (Al-

Algorithm 11 Computation method of radix-2^w Montgomery multiplication on our coprocessor

Require: radix-2^w, $d = \lceil \frac{n}{w} \rceil$, $w \cdot d \geq n + 3$, $X, Y, M, S_i \in \{0, 1, \ldots, 2^n - 1\}$, $-M^{-1} \in \{0, 1, \ldots, 2^w - 1\}$, $X = \sum_{i=0}^{d-1} 2^{w \cdot i} \cdot X_i, X_i \in \{0, 1, \ldots, 2^w - 1\}$, $X_d = 0$, $Y = \sum_{i=0}^{d-1} 2^{w \cdot i} \cdot Y_i, Y_i \in \{0, 1, \ldots, 2^w - 1\}$, $M = \sum_{i=0}^{d-1} 2^{w \cdot i} \cdot M_i, M_i \in \{0, 1, \ldots, 2^w - 1\}$, $M_d = 0$, $S_i = \sum_{j=0}^{d-1} 2^{w \cdot j} \cdot S_{(i,j)}, S_{(i,j)} \in \{0, 1, \ldots, 2^w - 1\}$, $S_d = 0$, $\delta_0, \delta_1, \delta_2 \in \{0, 1, \ldots, 2^{47} - 1\}$, $\delta_{i,high} = \delta_{i,(47:w)}$, $\delta_{i,low} = \delta_{i,(w-1:0)}$

Ensure: $S_d \leftarrow X \cdot Y \cdot 2^{-w \cdot d} \bmod M = ModMult(X, Y, M)$,

1. $S_0 \leftarrow 0$; (Initialization)
2. **for** $i \leftarrow 0$ **to** $d - 1$ **do**
3. *// On-the-fly q_i computation*
4. $\alpha \leftarrow X_0 \cdot Y_i$;
5. $\beta \leftarrow \alpha + S_{i,0}$;
6. $q_i \leftarrow \beta \cdot (-M^{-1}) \bmod 2^w$; ($q_i$ computation)
7. *// Pipelined computation of next value of S*
8. $\delta_1 \leftarrow [X_0 \cdot Y_i]_{(w-1:0)} + [q_i \cdot M_0]_{(w-1:0)}$;
9. $\delta_2 \leftarrow \delta_1 + S_{i,0}$;
10. $S_{(i+1,0)} \leftarrow \delta_{2,low}$; (First word of S_{i+1})
11. $C \leftarrow \delta_{2,high}$;
12. **for** $j \leftarrow 1$ **to** d **do**
13. $\delta_0 \leftarrow [X_{j-1} \cdot Y_i]_{(2w-1:w)} + [q_i \cdot M_{j-1}]_{(2w-1:w)}$;
14. $\delta_1 \leftarrow \delta_0 + [X_j \cdot Y_i]_{(w-1:0)} + [q_i \cdot M_j]_{(w-1:0)}$;
15. $\delta_2 \leftarrow \delta_1 + S_{i,j} + C$;
16. $S_{(i+1,j-1)} \leftarrow \delta_{2,low}$; (Remaining words of S_{i+1})
17. $C \leftarrow \delta_{2,high}$;
18. **end for**
19. **end for**
20. **if** $S_d \geq M$ **then**
21. $S_{d+1} \leftarrow S_d - M$; (Last reduction)
22. **end if**

gorithm 11, lines 16). Each stage in the pipeline is responsible for different lines of Algorithm 11. The w-bit blocks of the modular multiplication result are stored in R8 sequentially and they are routed to the output shift registers in a serial manner. When the computation of current S value is completed, the output of the shift register provide this value to the adder network to compute next S value.

5.3.2.5 Control unit

The control unit consists of a counter mechanism and shift registers. The counter mechanism simply counts the number of iterations required for the

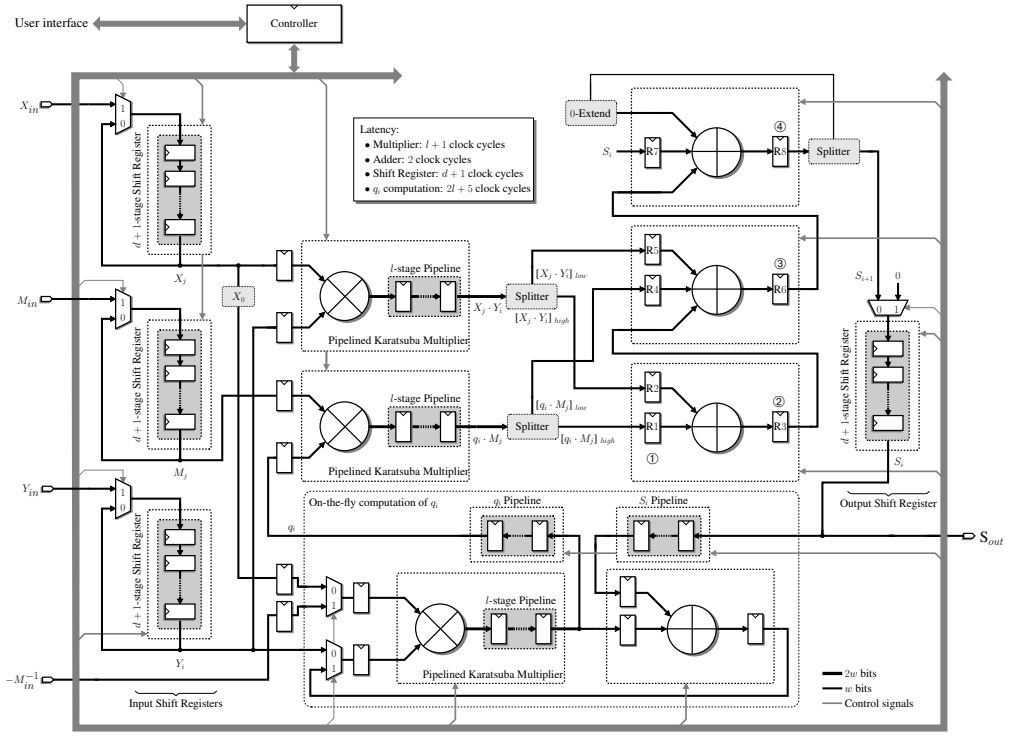


Figure 5.1: High-speed Modular arithmetic coprocessor.

size of the exponents of the modular exponentiation. It has three counters: two of them are counting the inner and outer loop of the Algorithm 11, the last one is required to count the exponent size. Shift registers in the control unit are responsible to manage the control signals for the datapath in order to manage the pipeline delays.

The user starts the modular multiplication or exponentiation by loading the input operands into input shift registers and applying a pulse for Start control signal. Then, fully autonomous modular arithmetic coprocessor computes the required output value and assert a Ready signal to indicate that output is ready.

5.3.2.6 Scheduling

We achieve very tight scheduling for our pipelined datapath by our control unit. Control unit is relatively simple due to the nature of the process structure of our pipeline. The operations are always performed in sequence. Hence, we just need to count and start new process with a counter mechanism. In this section, we show one frame of the scheduling to demonstrate how we achieve this tight scheduling. Thanks to the pipelining mechanism of Algorithm 11, we

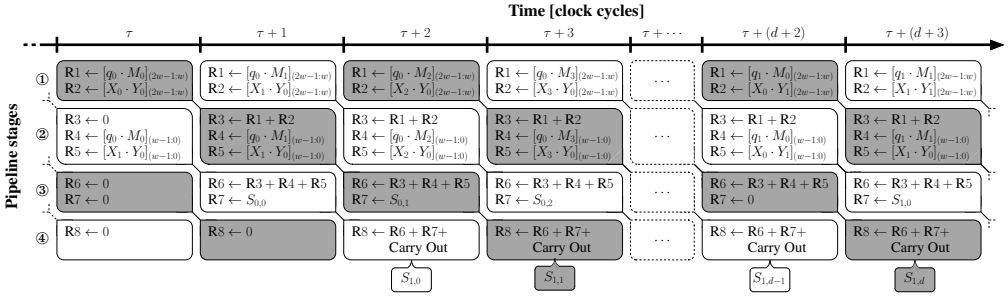


Figure 5.2: A view for pipelined computation of high-radix modular multiplication operation.

manage to keep our ALU full without any pipeline stall. Figure 5.2 illustrates our scheduling of our high radix Montgomery multiplication implementation:

- $q_0 \cdot M_0$ and $X_0 \cdot Y_0$ multiplication operations are executed and the most significant w-bit content is stored in register R1 and R2, respectively. At the same time, the least significant w-bit content is stored in register R4 and R5, respectively.

- Then, we execute the instruction R3 ← R1 + R2 and obtain δ_0 (see Algorithm 11, line 13), the most significant w-bit parts of the multiplications are added and provided to next adder with one cycle delay; at the same time, we execute the instruction R6 ← R3 + R4 + R5 and obtain δ_1 (see Algorithm 11, line 14), the least significant w-bit parts of the multiplications are added with the addition of the previous most significant w-bit parts; we also execute the instruction R8 ← R6 + R7 + Carry Out to compute δ_2 (see Algorithm 11, line 15). In this clock cycle, $S_{(0,0)}$ is also loaded in register R7. This synchronization is required for summation corresponding parts of the results. While these instructions are performed on the adder units, new $q_0 \cdot M_1$ and $X_1 \cdot Y_0$ multiplication operations are executed on the multiplier units and the most and the least significant w-bit contents are stored in their corresponding registers.

- In each clock cycle, these addition instructions are performed to sum the provided inputs from multipliers without any stall. Addition of the most significant w-bit parts of the multiplication results are always stored in R3 and this result is always summed with the least significant w-bit of the consequent multiplication results. Here, n-bit multiplication is realized in sequence by w-bit words. One cycle delay between these two additions is required to match corresponding w-bit parts of the n-bit multiplication

result. The result of the addition of the least significant parts added to previous S content (see Algorithm 11, line 15).

- At the end of the $d+1$ clock cycles, the whole S value is computed and stored into the output FIFO. d clock cycle is required due to the depth of $d = \lceil \frac{n}{w} \rceil$. One more clock cycle is required for the division operation which in fact exist in the Montgomery modular multiplication algorithm.

- After $d+1$ clock cycles, new inputs are provided to the multipliers and adders. The core of the algorithm (see Algorithm 11, line 12 and 18) is performed without any stall in these multiplier and adder network.

On-the-fly computation of q_i allows us to implement modular multiplier without any pipeline stall. q_i content is always required in multiplications. Hence, in order to start a new multiplication q_i should be ready. While initial data is loading to input FIFOs, very first q_0 content is started to be computed. When storing the data is completed, the multiplications can be started since q_0 is ready at the input of the multiplier. The core of the Algorithm 11 is started to compute next S_1 value. While this operation is in progress in the multiplier and adder network, next q_1 is calculating on-the-fly computation unit. When the last element of S_1 is ready, the next S_2 value computation can be started without any stall since q_1 is ready on the input of the multiplier. This on-the-fly computation requires an extra 1 DSP48E1 blocks and some control signals. However, it gives great performance improvement in terms of throughput.

There is a limit to have an architecture to compute next S values without any stall. A FIFO is placed in the output of the q_i generation in order to right scheduling. It needs to have a certain delay sizes in compliance with the parameters of the modular multiplier. This FIFO is called q_i pipeline and it is drawn in Figure 5.1. The number of stages of this pipeline is calculated by the equation below.

$$\text{The number of stages of } q_i \text{ pipeline} = (d+1) - [2 * (4+l) + 3]$$

For instance, d is equal to 16 for modulo size 1024 and radix size 64. Using this equation, it is necessary to implement the Karatsuba multiplier with a latency which is equal to 3. It is not allowed to have more latency in the Karatsuba multiplier blocks for these design parameters.

5.3.3 System-on-chip design

An application can be considered to show our high-speed coprocessor architecture is well-suited for many embedded systems which needs public key cryptography. In order to evaluate the performance achieved by our coprocessor compared to the pure software solution of modular exponentiation operation, we build three different SoC architectures on Xilinx's latest Zynq-7000 family extensible processing platform. These three different SoC scenarios show the compactness of the algorithm in terms of required area and achieved processing time.

5.3.4 Results and perspectives

We present our key results that we achieved in this Section. We conduct several experiments to measure the performance of our study. In the following, we first explain our methodology that we follow for performance evaluation of our proposed design for modular arithmetic coprocessor. We then define the metrics used to evaluate the performance and give our empirical results and comparisons with other published studies.

5.3.4.1 Methodology

In this study, proposed design strategies and implementations of our modular arithmetic coprocessor target Xilinx FPGAs. Hence, the results of this study are also valuable where FPGA technology comes into play. We mainly use Virtex-7 family of FPGAs to prototype our architecture. The Xilinx 7-family of FPGAs provide architectural elements designed for maximum performance, higher integration, and lower power consumption making a good choice for the high-speed architectures. We select Virtex-7 member of 7-family of FPGAs since it consists of more programmable logic cells together with higher performance compared to the other members, Artix7 and Kintex-7 FPGAs. We captured our modular arithmetic coprocessors in the VHDL language and evaluate the performances of a fully autonomous implementations of our architecture on Xilinx Virtex-7 (7VX330T-3) FPGA. We also use ZedBoard based on the Xilinx's latest Zynq-7000 All Programmable SoC in order to get resource utilization of our SoC architectures. We measure the processing time in ms for SoC architectures with a timer module in software. We synthesized our hardware architecture with Xilinx PlanAhead and Vivado development

tools. We also simulate our design with Xilinx ISim Simulator to verify the correctness of the proposed architectures.

5.3.4.2 Results

The evaluation metrics of our modular multiplication and exponentiation coprocessors on FPGA are given in Tables 5.1 and 5.2 at different operand sizes. We consider here the less favorable case for modular exponentiation, a high number of modular multiplication operations are performed. In other words, we take the base, exponent, and modulus which all have the same bit length for the modular exponentiation operation. This is significantly important in the comparison of the architectures in terms of latency. Some of the studies in the literature use exponents which have average number of bits for calculating the latency. Therefore, the latency is significantly low. In our study, we select the worse case scenario for latency calculation.

Table 5.1: Synthesis results for the proposed modular multiplication coprocessor on Virtex-7 FPGA.

FPGA	n	Radix size [w]	Area [slices]	DSP48E1 blocks	Frequency [MHz]	Processing time [μs]
Virtex-7 (7vx330t-3)	512	16	128	7	478	2.29
		32	382	13	288	1.02
	1024	16	170	7	479	8.83
		32	536	13	490	2.24
		64	1257	27	211	1.39
	2048	16	235	7	479	34.75
		32	567	13	490	8.64
		64	1785	27	403	2.73

Tables 5.1 summarizes our synthesis results in terms of the evaluation metrics we defined in Section 2.3. We measured these results with the Xilinx PlanAhead and Vivado tools for the modular multiplication operation. The modular multiplication coprocessor requires for instance 1257 slices and 27 DSP48E1 blocks and achieves processing time of 1.39 μs for the modular multiplication of 1024-bit operands. One can see the scalability of our coprocessor by looking at different modulo sizes where the the same radix size is used. For instance, when radix size is equal to 16, the coprocessors for 512, 1024 and 2048-bit modulo sizes uses same number of 7 DSP48E1 blocks with a small increase in area utilization due to the need of varying sizes of FIFO elements

for input and output. Control mechanism does not effect dramatically when the modulo size is increased thanks to the usage of simple counter mechanism.

Table 5.2 summarizes our synthesis results measured with the Xilinx PlanAhead and Vivado tools for the modular exponentiation operation. The modular exponentiation coprocessor requires for instance 3046 slices and 54 DSP48E1 blocks and achieves competitive processing time of 1.52 ms for the modular exponentiation of 1024-bit operands. The maximum clock frequency of the DSP48E1 is 741 MHz where all registers in the DSP48E1 are used. Thanks to this high-frequency value, we obtained very high-operating frequencies above 400 MHz for the proposed coprocessors. However, for instance, the coprocessor, where modulo size is 512 and radix size is 32, operates on 288 MHz which is relatively small. This is caused by the need to run the pipelined Karatsuba multiplier with a small latency. When the ratio of modulo size to radix size is equal to 32 or above, we have flexibility to run the Karatsuba coprocessor with higher latency which allows us to achieve higher frequencies. We employ some of the embedded registers in the DSP48E1 according to the constraints in our architecture due the our pipeline mechanism (Figure 5.1). Due to the this fact and also the controller mechanism delays, we achieved the operating frequencies given in Table 5.1 and 5.2. It is possible to improve the achieved frequencies by using more embedded registers but this causes to have a pipeline stall.

Table 5.2: Synthesis results for the proposed modular exponentiation coprocessor on Virtex-7 FPGA.

FPGA	**n**	**Radix size [w]**	**Area [slices]**	**DSP48E1 blocks**	**Frequency [MHz]**	**Processing time [ms]**
Virtex-7 (7vx330t-3)	512	16	343	14	458	1.23
		32	801	26	283	0.54
	1024	16	385	14	468	9.28
		32	1060	26	485	2.33
		64	3046	54	201	1.52
	2048	16	553	14	370	92.25
		32	1092	26	413	21.03
		64	3558	54	399	5.68

In Xilinx's 7VX330T-3 FPGA, there are 1120 DSP48E1 blocks which is lower than other members of Virtex-7 FPGAs. Even if the selected device has the lowest number of DSP48E1 blocks compared the other members of Virtex-7 FPGAs, one can fit 36 and 18 instances for the modular multiplica-

tion and exponentiation coprocessors in the FPGA device, respectively. If the embedded multiplier blocks are not employed in the design, surrounding slices of these multipliers can be used for routing purposes or other logic functionalities. Multiplier blocks are surrounded by our coprocessors and the efficiency of resource utilization on FPGA is increased. Thanks to the use of embedded multipliers in pipelined mode, significant performance improvement in terms of latency and frequency is attained. This leads to substantial increase at the attained performance for parallel modular arithmetic operations which are extensively required in many secure protocols such as RSA with homomorphic property.

Table 5.3 illustrates the results of latest hardware architectures for the modular exponentiation operation (see for instance [151] for various hardware architectures). We consider the least favorable case for modular exponentiation operation where base, exponent and modulus have all same size. Some of the reported studies in Table 5.3 take into account that the average number of modular multiplications is required for the exponentiation. This work attains very good performance results compared to the other studies in terms of processing time which is the main optimization goal in this study. We achieve for instance 1.52 ms for 1024 bit operands whereas all other reported studies are less than in processing time of one-block of data. Our proposed architecture is much better than some of the proposed designs with the cost of extra usage of DSP48E1 blocks. Fortunately, today's FPGA devices have a large amount of programmable logic components and embedded DSP blocks as it can be seen in our selected FPGA device.

The latency of our Montgomery modular multiplication operation is calculated using the following equation.

$$\text{Latency} = (d+1) * (d+1) + (l+1)$$

When radix size is increased, the depth of the multiplier is decreased and the latency is decreased significantly. Thanks to this, we are able to operate on higher radix sizes and having much less latency compared to the others.

In order to show the effects of our hardware architectures in a real embedded system, we build a SoC design whose framework is explained in detail in Section 5.3.3. We compare the empirical findings obtained from all different SoC designs that we built. We prototype our Zynq-7000 based SoC designs using Xilinx PlanAhead tool.

Pure software SoC design is in fact consists of only dual core Cortex-A9

Table 5.3: Performance Results of Hardware Architectures for Modular Exponentiation on FPGAs.

	Algorithm	FPGA	n	Area [slices]	18K-bit RAM	DSP blocks	Frequency [MHz]	Processing time [ms]
Nakano *et al.* [152]	Redundant	XC2VP30-6	512	5868	15	32 18-bit Mul.	52.57	0.645
			1024	11589	29	64 18-bit Mul.	52.9	2.521
	Non-Redundant	XC2VP30-6	512	3911	15	31 18-bit Mul.	30.40	1.113
			1024	7708	29	61 18-bit Mul.	18.46	7.218
Tang *et al.* [153]	Using Parallel Multipliers	XC2V3000-6	512	8235	–	32 18-bit Mul.	99.26	0.59 †
			1024	14334	–	62 18-bit Mul.	90.11	2.33 †
Blum and Paar [5]	High-radix Montgomery	XC40250XV-9	512	3413 CLBs	–	–	–	2.93
			1024	6633 CLBs	–	–	–	11.95
Suzuki [124]	High-radix Montgomery	XC4VFX12-10	512	4190	7	17 DSP48	200–400	0.261
			1024	4190	7	17 DSP48	200–400	1.71
This Work	High-radix Montgomery and Karatsuba	7VX330T-3	512	801	–	26 DSP48E1	283	0.54
			1024	3046	–	54 DSP48E1	201	1.52
			2048	3558	–	54 DSP48E1	399	5.68
Bo *et al.* [126]	High-radix Montgomery	XC6VLX240T-1	512	180	1 36K-bit	1 DSP48E1	447	4.99
			1024	180	1 36K-bit	1 DSP48E1	447	36.37
			2048	180	1 36K-bit	1 DSP48E1	447	277.26
Bo *et al.* [127]	High-radix Montgomery	XC6VLX240T-1	1024	201-Slice Register 374-Slice LUTs	1 36K-bit	1 DSP48E1	410	11.263

†Average number of modular multiplication is considered.

processor and some I/O hardware modules without any accelerator core for modular arithmetic. Modular arithmetic operations are realized by means of pure software. The other two SoC designs include our modular arithmetic coprocessor with different communication mechanisms. One of them is connected to Cortex-A9 processor with AXI4-Lite and the other one is connected with AXI4 interface. We give the performance results of our all different SoC designs in Table 5.4. As seen from the table, our speed-up is for instance about 121 for 1024-bit operands. The speed-up achieved with our modular arithmetic coprocessor can be increased where parallel execution of modular arithmetic operations is possible. Such a scenario always exists in many public-key cryptosystems since they operate on high-dimensional matrices of data with privacy. Such evaluation is given before in this Section.

According to the results of our proposed modular arithmetic coprocessor, it can be seen that significant performance improvement is attained. Furthermore, we also evaluate our modular arithmetic coprocessor on different SoC platforms described in Section 5.3.3 to emphasize on the contribution of our proposed hardware architecture from systems perspective.

This study contains three important features.

- The device specific units in FPGA are well-suited for the pipelined ALUs

Table 5.4: The perfomance figures of various SoC architectures on Zynq-7000 based extensible processing platform.

SoC Architecture	**n**	**Mod. Exp. Coproc.**	**DDR3 SDRAM**	**Area [slices]**	**DSP48E1 blocks**	**Processing time [ms]**
Zynq-7000 based	512	–	✓	–	–	116
Pure Software	1024	–	✓	–	–	373
SoC architecture	2048	–	✓	–	–	1265
Zynq-7000 based	512	✓	✓	1363	26	1.54
Mod. Exp. Coproc.	1024	✓	✓	5876	54	3.06
SoC with AXI4-Lite	2048	✓	✓	5945	54	22.67
Zynq-7000 based	512	✓	✓	1576	26	1.52
Mod. Exp. Coproc.	1024	✓	✓	6152	62	3.04
SoCwith AXI4	2048	✓	✓	6224	62	22.65

for our overall design.

- We use very high radix Montgomery algorithm together with compact large pipelined Karatsuba multipliers.
- We also keep the pipeline always full to increase the processing time thanks to the our scheduling.

5.3.5 Conclusions

We take advantage of both high-radix Montgomery modular multiplication and Karatsuba algorithms and harness the intrinsic parallelism of these algorithms to design a pipelined ALU. We also interleave independent tasks in the algorithms in order to achieve a very tight scheduling. The design philosophy we proposed in this study led us to develop a high-performance scalable coprocessor for modular arithmetic operations including modular multiplication and exponentiation at different operand sizes. From our point of view, the main advantage of this method over other existing methods is that the algorithm combines the efficiency of both Montgomery and Karatsuba algorithms in a compact manner using built-in device specific hardware modules placed in FPGAs.

Despite the various control signals required for the different steps of pipelined computation of Algorithm 11, our control unit remains compact:

almost all control signals are generated by means of a counter, a start signal and its delayed versions. Thanks to an alternative description of 2^w radix Montgomery modular multiplication and a careful organization of the pipelined datapath, we manage to implement the modular arithmetic operations (multiplication and exponentiation) without any pipeline stall. The key element of our approach to achieve high-throughput design is to take advantage of the parallelism of modular arithmetic operations to

- deeply pipeline the ALU to achieve a high clock frequency;
- find parallelism between independent tasks of the modular arithmetic operations.
- decrease the data dependencies and hazards to achieve more parallelism and exploit this parallelism in hardware better.
- reuse same resources for different executions of the algorithm.
- design pipelined datapath for the independent tasks so that each pipeline units can operate on different set of inputs at the same time.

This study also reveals a better understanding of the modular arithmetic operations in terms of computation for FPGA based applications and also includes empirical performance analysis of the algorithm from hardware perspective. Accelerating the computation of cryptographic algorithms especially for public key cryptography to protect network traffic and confidential data within some area constraints is rapidly becoming significant in today's processor technology.

5.4 Accelerating Privacy-Preserving Applications via Paillier Cryptoprocessor

Privacy-preserving applications are receiving increasing attention due to growing popularity of computing with privacy. To protect confidential data in such applications, various methods have been proposed. Cryptographic techniques, especially with the homomorphic encryption property, are widely used privacy-preserving schemes. Paillier cryptosystem is extensively utilized as a homomorphic encryption scheme to ensure privacy requirements in many privacy-preserving applications. However, overall performance of the applications employing Paillier cryptosystem intrinsically degrades due to the fact

that the cryptosystem heavily depends on modular multiplication and exponentiation operations.

In this study, we scrutinize how to tackle with such performance degradation of privacy-preserving applications utilizing Paillier cryptosystem. We propose a high-speed Paillier cryptoprocessor to accelerate homomorphic operations performed in such cryptosystem and correspondingly overcome the performance losses. For this purpose, we first exploit field-programmable gate arrays for necessary hardware realization of our proposed high-speed architecture. We then take advantage of the parallelism among the operations in the Paillier cryptosystem to fully pipeline our datapath. We also utilize interleaving among independent operations to enhance computations by avoiding data dependencies. To validate the improved performance promised by our architecture, we conducted several experiments using our Paillier cryptoprocessor. Our empirical outcomes demonstrate that the proposed cryptoprocessor meets the performance requirements needed by privacy-preserving applications employing Paillier cryptosystem. The architecture of Paillier cryptoprocessor developed in this section mainly consists of the same design details of the architecture presented in [154].

5.4.1 Introduction

With recent advancements in information and communication technologies, e-applications and information systems make life easier. Such systems promise a range of activities for people from getting news to selling real estates, even socially interaction through social networks. For many individuals, they also facilitate and promote the works throughout a business world. However, on the flip side, they cause various severe privacy risks [155]. Without any protection, users might be subjected to profiling and faced with price discrimination. Moreover, they can receive disturbing amount of spam mails, messages, phone calls, or e-mails for unsolicited marketing. Since collected users' personal data are regarded as confidential and valuable assets, data collectors are obliged to protect such data [156]. Hence, in order to benefit from e-applications and information systems effectively, both individuals and institutions can perform online transactions if and only if data confidentiality is protected.

There are various studies responding the privacy requirements in the context of e-applications and information systems from privacy preserving OLAP queries [157] to secure electronic auction infrastructures [135]. Privacy-preserving applications (PPAs) in general provide various services while pre-

serving privacy. Such applications might concern about protecting individual privacy in either server-based or peer-to-peer (P2P) configurations. Furthermore, they provide solutions for partitioned or distributed data-based scenarios. Some examples of such cases are shown in Figure 5.3. While the bottom part of the Figure 5.3 exhibits the frameworks for considering individual privacy concerns, the top part is related to requirements in corporate privacy domain in case of partitioned or distributed data. In the individual privacy framework, there are not only some applications [158, 159] concerning how user profiles can be collected and processed while ensuring their privacy, but also some proposals [160, 161] realizing P2P computations with privacy. In the general context of corporate privacy framework, in addition to two-party cases exemplified in the Figure 5.3, there can be more parties desiring to benefit from each other's data without any information leakage about own data. Most popular examples for this domain are secure multi-party computations (SMCs) in which a computation is secure if no party knows anything except its own input and the final results of the computed function at the end of the computation [162]. Several privacy-preserving computation protocols are proposed in the literature to achieve SMCs [157]. In addition to them, there are also data publishing schemes in which privacy is achieved using anonymization techniques such as k-anonymity [163], l-diversity [164], and so on. To preserve privacy, cryptographic techniques with homomorphic encryption property are widely used. Paillier homomorphic cryptosystem (PHC) is popularly utilized homomorphic cryptosystem (HC). Due to its versatility, PHC is widely applied by many PPAs such as recommenders [133], classifiers [165], e-voting [134], e-auctions [135], and so on.

Privacy and efficiency are two clashing goals; thus, PPAs often suffer from performance bottlenecks. Such bottlenecks may be handled with the aid of off-line computation phase having labor-intensive tasks. Likewise, clustering [166] and preprocessing approaches [167] can contribute to the efficiency of PPAs. In addition to such software-based solutions, hardware-based ones are also offered to tackle with the performance problems of PPAs [146]. Although there are various approaches for improving the performance of PPAs, those PPAs utilizing PHC are still suffering from poor performance due to computationally intensive PHC. Like other alternatives, PHC performs not well due to the clumsy nature of cryptographic mechanisms [133, 136]. Even if software related schemes are proposed to improve such poor efficiency, such enhancements will be limited. Therefore, new proposals are required to ameliorate the

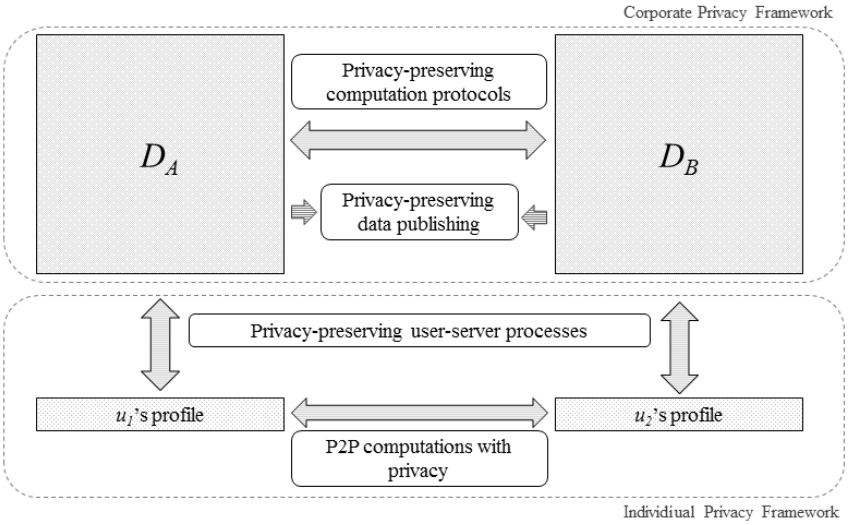


Figure 5.3: Examples of Privacy-Preserving Frameworks.

performance of the PPAs whose privacy guarantee is based on PHC.

Hardware-based solutions such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs) can be applied to efficiently implement computationally intensive algorithms. While ASICs have predetermined functionality to perform a particular task, FPGAs are able to perform an arbitrary task by combining logic gates, flip-flops, and memory units due to their reconfigurability property [168]. On the other hand, the ASIC implementation is faster and more secure than the software implementation [169]. However, the ASIC implementation is not flexible enough and its longer design cycle and higher development cost make it less attractive and not preferable choice to build a cryptosystem in low volume designs [170]. The software-oriented implementation is flexible; however, its computational complexity is higher compared to its hardware equivalent. Moreover, storing the private key in the computer memory will endanger the security of the system. On the other hand, in contrast to CPU-based software solutions, FPGAs offer more flexibility in highly parallel data processing, low latencies, and high throughput rates [144]. Teubner et al. [144] show that FPGAs consume significantly far lower power than conventional PCs. In addition to them, the use of hard blocks available in FPGAs such as multipliers, block memories, and I/O features allow to significantly reduce the area gap between ASIC and FPGA [171]. FPGAs having such prominent features can be a solution to fulfill performance implications of PHC. This motivation leads us to design a popular homomorphic scheme in an FPGA-based environment.

In order to implement PHC, two core modules, modular multiplication and exponentiation, are needed. In fact, modular multiplication and exponentiation are two of the most important arithmetic operations in modern applied cryptography. Hence, their efficient implementations have been at the center of research activities in cryptographic engineering [5, 126, 137]. In this study, we mainly focus on the high-speed hardware implementation of the PHC. Specifically, we use Montgomery modular multiplication and right-to-left modular exponentiation algorithms. We select Xilinx 7-family FPGA devices due to its high logic capacity and processing performance. The 7-family of FPGAs have several dual ported, highly scalable embedded Block RAMs (BRAM), DSP blocks (DSP48E1), and configurable logic blocks. Our design effectively uses these embedded multiplier and adder units available on Xilinx FPGAs. More specifically, we propose new design strategies. We take advantage of the parallelism among the operations in the PHC and utilize interleaving among independent operations to enhance computations. To the best of our knowledge, this is the first study designing an efficient Paillier cryptoprocessor (PCP) for enhancing the overall performance of PPAs based on the PHC.

5.4.2 Paillier cryptoprocessor design

There are several design approaches that can be applied to cryptographic algorithms for efficient hardware implementations such as a low-area coprocessor design approach [76], a compact coprocessor design approach [75], high-speed architecture design approach [42], and so on. The requirements of PPAs show that high-speed and high-throughput are the most important optimization goals in designing PCP. In this study, we exploit the parallelism in the operations and utilize pipelining as a digital design strategy to improve overall throughput.

5.4.2.1 Design philosophy

We scrutinize how to design a high-speed architecture so that the coprocessor meets the requirements of PPAs. In order to achieve such task, we utilize a design philosophy, which consists of the following four fundamental design methods.

1. *Pipelining*: In order to achieve maximum clock frequency and increase the throughput, pipelining is employed in the datapath. To compute the main operations of the PCP, a set of arithmetic operation modules

connected in series execute in parallel by inserting buffers between elements. It allows us to break up long strings of data and shorten critical paths. Necessary computations are seperated into a set of equal stages. We balance each stage by utilizing DSP48E1 blocks in order to achieve high frequency values.

2. *Parallel Execution*: Each stage in the overall datapath performs one part of the algorithm. Our pipelined datapath allows parallel execution of independent tasks of the algorithm at the same time. Hence, the stages of our pipelined datapath operate simultaneously using different resources, which increases the throughput of the system.

3. *Interleaving*: We interleave independent operations performing modular multiplication and exponentiation utilized in this study. This allows us to obtain a tight scheduling, which increases our throughput substantially.

4. *Resource Sharing*: We reuse a modular multiplication (ModMult) component for a series of modular multiplication operations, which cannot be parallelized, in the PCP to enhance resource utilization.

This design philosophy that we follow in this study allows us to develop a scalable and high-speed processor for encryption and decryption procedures of the PHC. All these design methods can significantly improve the PHC's performance.

We display the required sizes for PHC and its inner components in Table 5.5. As seen from Algorithm 11, there are two loops, which are controlled by i and j indices. Moreover, one can see how these indices are changed compared to size of the key, as shown in Table 5.5. Note that the modular multiplication size is two times greater than the Paillier size. PHC mainly involves three operations: $2n$-bit modular multiplication, n-bit modular exponentiation, and $L(u)$ operation. We apply the following design strategies to efficiently compute these operations in order to build the high-speed PCP.

1. *Modular Multiplication*: We employ high-radix Montgomery modular multiplication algorithm (Algorithm 11) for modular multiplication operations. Specifically, we adapt the modular multiplication coprocessor presented in the previous section. We select radix 2^{32} version in order to efficiently utilize the DSP48E1 blocks because these blocks are able to add three 48-bit operands in just one clock cycle. The 2^{32} radix Montgomery multiplication consists of 32-bit multiplication operations.

Table 5.5: Fundamental sizes of Paillier-n and its inner components

Key size n [bits]	**ModMult size $2n$ [bits]**	**# of rounds j**	**# of pipeline step i for each round j**
512	1024	64	65
1024	2048	128	129
2048	4096	256	257

However, the DSP48E1 blocks are not able to perform 32-bit multiplication. Hence, we first implement 32-bit 6 cycle pipelined multiplication module utilizing 4 DSP48E1 blocks.

2. *Modular Exponentiation*: We use right-to-left binary modular exponentiation algorithm (Algorithm 4), which has repeated modular multiplication operations depending on the exponent value. It is in fact simply square and multiply algorithm according to the exponent. We mainly adapt the modular exponentiation coprocessor presented in the previous section.

3. $L(u)$ *Operation*: There are two different ways of implementing this operation. One of them is realized by means of a multiplication with a precomputed value. It is possible first decrement u by one with one of the DSP48E1 blocks. Then, $(n^{-1} \bmod 2^{|n|})$ or $(n^{-1} \bmod n+1)$ is multiplied with the computed $u-1$ in order to perform $L(u) = \frac{u-1}{n}$. The other way is mainly consists of a integer division of very long numbers. There is no need to apply a decrement by one operation required by the $L(u)$ operation since quotient of the division by n always conveys the correct result of $\frac{u-1}{n}$ operation. The division takes varying numbers of clock cycles to complete the process of division depending on the values of the inputs. We accommodate this method in our architecture. A long integer division is computed with a specific pipelined division accelerator. In this approach there need to be a precomputation of the multiplication input.

   The algorithm proposed in [172] is adopted for the long integer division in this study. The main feature of the algorithm depends on the quotient selection with the difference of the most-significant non-zero bit positions of the divisor and dividend. Hence, it dramatically reduces the total number of long subtractions required in the division. This helps to

improve the computational performance of the division operation. The result of the division operation is calculated in varying number of steps according to the the most-significant non-zero bit positions of the divisor and dividend. In this study, we used this algorithm and implement it in digit-serial fashion.

5.4.2.2 Hardware implementation

Our architecture consists of shift registers, an ALU, and a control unit. Shift registers are needed to store the input and the output operands. Our pipelined ALU utilizes the DSP48E1 blocks to perform the required 2^{32} radix multiplication operations. Our control unit is responsible for providing necessary control signals for the datapath in order to compute the Paillier autonomously. We assume that our PCP is provided with the required input values including input operands, modulo number, and its 32-bit inverse value ($-M^{-1}$).

5.4.2.3 Shift registers

Our proposed high-speed PCP operates on 32-bit blocks of data in a sequential manner. We take advantage of this to keep the data in shift registers, which can be implemented in SRL32 blocks existing in 7-family of Xilinx FPGAs. Using SRL32 blocks consume less resources than storing the data in the slice registers of FPGA. Shift registers are organized into 32-bit words to store necessary operand values.

5.4.2.4 Arithmetic and logic unit

The PHC mainly consists of encryption and decryption. We describe the computation flow of encryption $\mathcal{E}$ and decryption $\mathcal{D}$ of the PHC, which includes modular multiplication and exponentiation operations, in Figure 5.4 and 5.5, respectively. The $\mathcal{E}$ function requires two ModExp components. Each ModExp component mainly consists of two ModMult components. These ModMult modules can perform a series of modular multiplications required in Algorithm 4 in parallel because they operate using different input resources. Finally, to obtain the final result of the encryption, outputs of two modular exponentiations are multiplied by one of the existing ModMult modules. One of the ModMult modules is reused for the last modular multiplication operation required in the $\mathcal{E}$. Furthermore, one of the modular exponentiation $g^m \bmod n^2$ components can

be precomputed once for all, which is given in [41]. When this optimization is applied, there is no need to have the second ModExp; thus,this optimization brings resource saving.

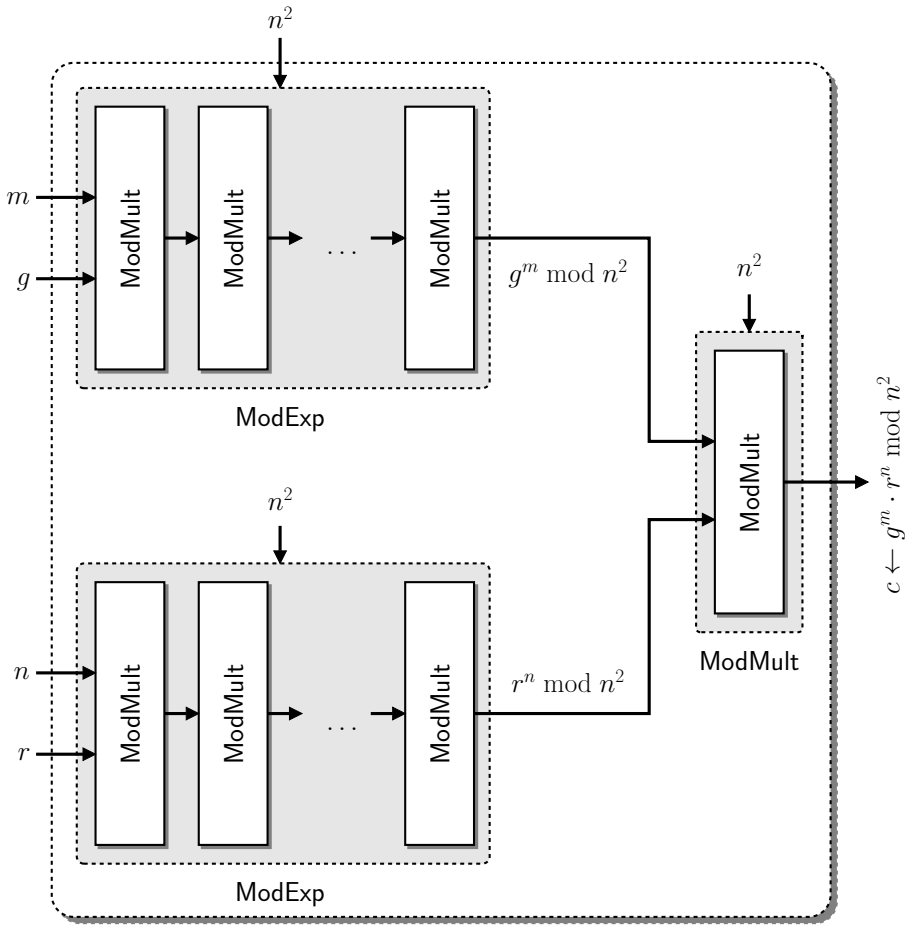


Figure 5.4: The architecture of Paillier encryption $\mathcal{E}$ procedure of PHC

Figure 5.5 illustrates the computation flow of decryption, which includes modular multiplication, exponentiation, and $L(u)$ operation. The $\mathcal{D}$ needs a ModExp component. The operation of the ModExp component is the same as in $\mathcal{E}$. The result of the modular exponentiation is fed into $L(u)$ module to perform $\frac{u-1}{n}$ operation. The precomputed value of $L(g^{\lambda} \bmod n^2)^{-1} \bmod n$ and the result of the $L(u)$ operation are multiplied with the existing ModMult module to obtain the final outcome of the decryption.

Our architecture is mainly built around 32- and 48-bit datapath. The main datapath performs all operations required by the PHC. The datapath allows us to realize the encryption and the decryption of PHC with high performance because it has very high operating frequency and the number of clock cycles needed for the Paillier $\mathcal{E}$ and $\mathcal{D}$ are small due to our design rationale.

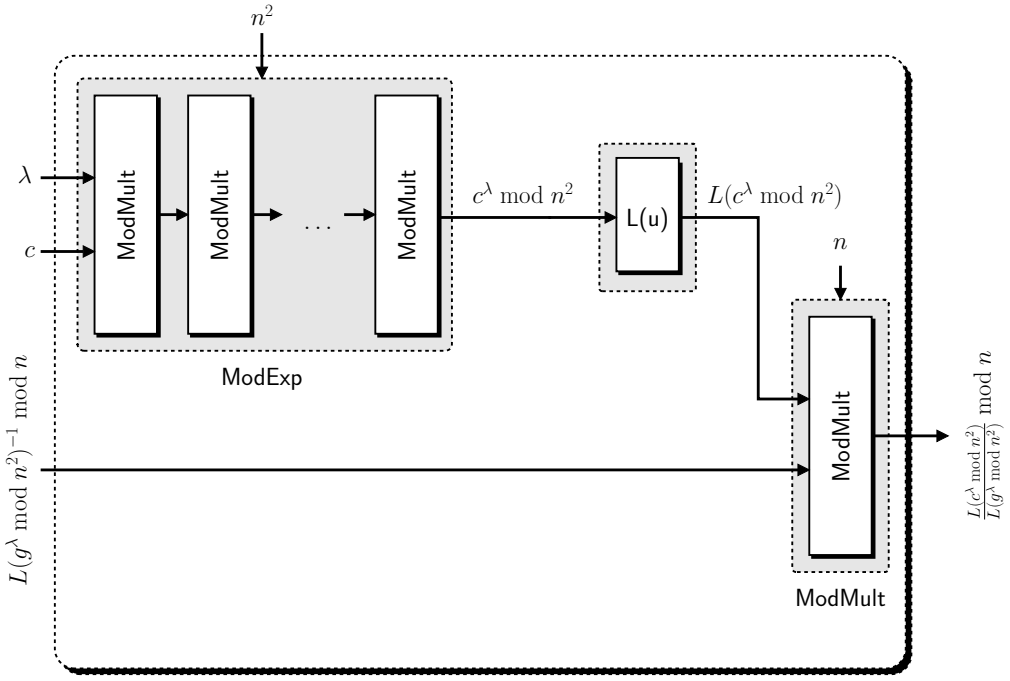


Figure 5.5: The architecture of Paillier decryption $\mathcal{D}$ procedure of PHC

In Figure 5.6, 2^{32} radix size version of the modular multiplier coprocessor is used for for the high-radix modular multiplication architecture ModMult. In this figure, Ri denotes a 48-bit register and they compute the next value of S with the inputs introduced by the high-radix multipliers. Each stage in the pipeline is responsible for different lines of Algorithm 11. The 32-bit blocks of the modular multiplication result are stored in R8 sequentially and they are routed to the output shift registers in a serial manner. The high-radix modular multiplication operation is regarded as the most critical part of our proposed PCP. It requires to design high-radix multipliers in the datapath. On FPGAs, the best design strategy consists of implementing this multiplier by means of DSP48E1 blocks. The control signals allow us to perform the operations defined in Algorithm 11 very efficiently. The datapath has several pipeline units to improve the performance of the computation of modular multiplication operation.

5.4.2.5 Control unit

The control unit consists of a counter mechanism and shift registers. The counter mechanism simply counts the number of iterations required for the size of Paillier scheme. It has three counters. Two of them are counting the

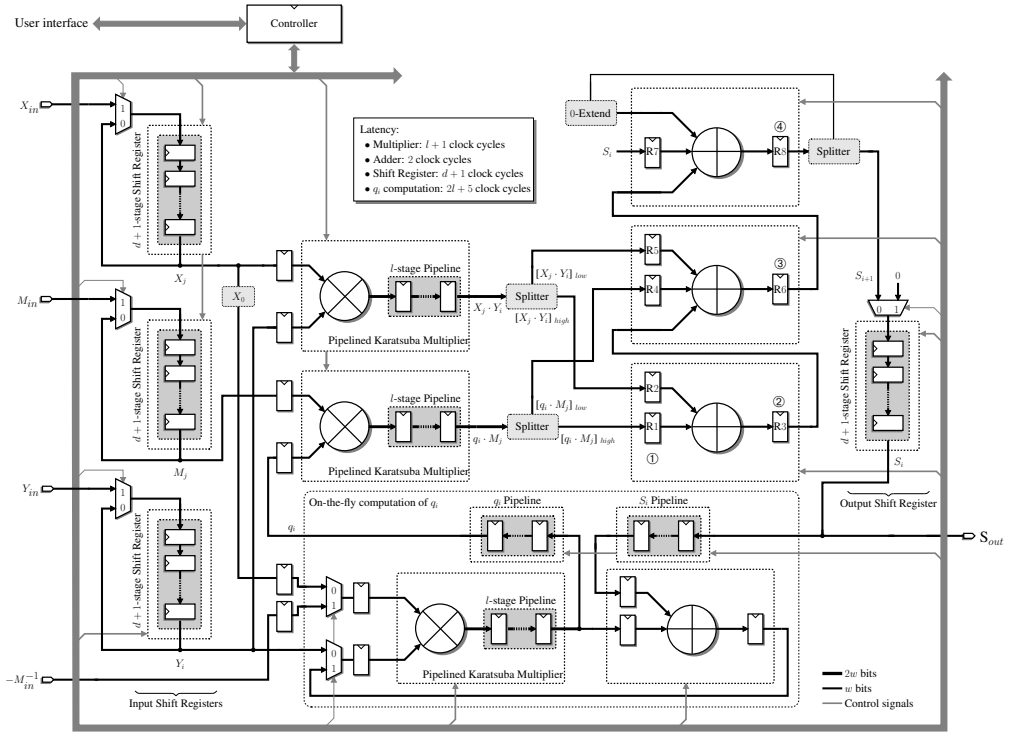


Figure 5.6: The hardware architecture of ModMult (adopted for modular multiplication operation for 2^{32} radix size)

inner and outer loop of Algorithm 11 while the last one is required to count the exponent size. Shift registers in the control unit are responsible for managing the control signals for the datapath in order to cope with the pipeline delays.

The user starts the Paillier encryption or decryption by loading the input operands into input shift registers and applying a pulse for start control signal. Then, fully autonomous PCP computes the required output value and asserts a ready signal to indicate that output of Paillier encryption or decryption is ready.

5.4.3 Empirical outcomes and discussion

In this section, we present our empirical result. We first explain our methodology for performance evaluation. After the metrics used to evaluate the performance of our design are defined, empirical results are displayed and discussed.

5.4.3.1 Methodology

Our proposed high-speed PCP's implementations and the presented design strategies target Xilinx FPGAs. Therefore, the results of this study are valu-

able, where FPGA technology comes into play. We mainly used 7-family of FPGAs to prototype our architecture. The Xilinx 7-family of FPGAs provide architectural elements designed for maximum performance, higher integration, and lower power consumption making a good choice for the high-speed architectures. We captured our PCP in the VHDL language and evaluated the performance of a fully autonomous implementation of our architecture on several Xilinx FPGAs. We synthesized our hardware architecture with Xilinx ISE 14.2. We also simulated our design with Xilinx ISim simulator to verify the correctness of the proposed architecture. PHC was implemented by using Java language in general purpose CPU to confirm that the proposed PCP works properly. We compared the performances of software implementation our hardware-based design. We provided the performance evaluation results with detailed discussions and compared our performance results with the figures of Java implementation.

5.4.3.2 Empirical results

We first performed a set of experiments to show how our proposed PCP's latency changes with varying n sizes. Recall that we used Xilinx ISE 14 development tools to perform our simulations. Also note that the datapath width of the PCP is 32-bit. We considered the least favorable case for PCP in which the exponent has all 1s, which means that maximum number of modular multiplication operations are performed for each size. After calculating the related latencies in terms of number of cycles for varying n sizes from 512 to 2048, we displayed the outcomes in Table 5.6. Notice that the values listed in the third and the fourth columns (d and number of ModMult) were obtained, as defined in Algorithm 3. Number of pre- and post-cycles represent the number of cycles required to fill the pipeline and to get the outcomes of our datapath, respectively.

As expected and seen from Table 5.6, latency improves with decreasing n sizes. However, security becomes worse with decreasing n sizes. Although number of cycles spent for decryption are slightly more than the ones spent for encryption, they are very close to each other.

We then conducted some simulations to determine the throughput of the PCP. We used three different FPGA devices. We calculated throughput values for varying n sizes and displayed them in Table 5.7. The throughput values listed in the table are computed for one block message. We also presented area, DSP block numbers, and frequency values, which are the synthesis results of

Table 5.6: Number of clock cycles required for the PCP with varying n sizes

Paillier	modulo size n	# of words $[d]$	# of ModMult $[e+1]$	# of pre- and post-cycles	# of cycles
Encryption	512	16	257	100	70004
Decryption			258	118	70294
Encryption	1024	32	513	100	541828
Decryption			514	134	542918
Encryption	2048	64	1025	100	4264100
Decryption			1026	166	4268326

Xilinx ISE 14.2. We calculated the throughput of the PCP, as follows:

$$Throughput = \frac{n \cdot f}{(n+\epsilon) \cdot (N \cdot (N+1) + \tau_p)},$$

where n represents the Paillier size, f denotes the operating frequency, N is the number of blocks for the Paillier size ($N = \frac{2n}{32}$), ϵ represents the number of pre- and post-modular multiplications required for Montgomery exponentiation and τ_p is the pipeline delay.

Table 5.7: Performance of the PCP in terms of throughput

FPGA	Area [slices]	DSP48E1 blocks	Frequency [MHz]	Throughput [Kbits/s]		
				$n = 512$	$n = 1024$	$n = 2048$
Artix-7 (xc7a100t)	3880	32	160	584	150	38
Kintex-7 (xc7k70t)	3720	32	172	628	162	41
Virtex-7 (7vx330t)	3850	32	180	658	169	43

According to the outcomes presented in Table 5.7, with increasing n sizes, throughput decreases; however, security enhances. Although the results are very close to each other for three different FPGA devices, the best outcomes are observed for Virtex-7. As seen from Table 5.7, our PCP for PHC requires 3850 slices and 32 DSP48E1 blocks; and it achieves competitive throughout of 169 Kbits/s for encryption of 1024-bit message blocks. Our empirical results show that our PCP performs well especially for applications in which time constraints are important. This phenomenon is observed due to the fact that our design takes advantage of pipelining, which allows us to use different stage of the datapath at a time as well as to operate at higher frequencies.

In order to compare our proposed cryptoprocessor with the software-based implementation of the PHC in terms of throughput, we finally conducted another set of simulations. We used Intel's E6420 CPU-based PC machine to

evaluate the PHC's software-based implementation. The PC has two cores, 2.13GHz clock rate (which is much more higher than the frequency of our architecture), and 4MB cache. The implementation was realized using Java platform. Throughput values were computed for one-block message. After measuring throughput values for varying security levels (different n sizes), we presented them in Table 5.8.

Table 5.8: Performance of the software implementation of PHC

CPU (processor number)	# of cores	Clock speed [Ghz]	Cache [MB]	Throughput [Kbits/s]		
				$n = 512$	$n = 1024$	$n = 2048$
E6420	2	2.13	4	124	32	8

The results presented in Table 5.8 show that throughput improves with decreasing n sizes as it is expected. If we compare our design's outcomes displayed in Table 5.7 with the results representing the software implementation of PHC shown in Table 5.8, one can figure out that our architecture performs five times faster with respect to throughput. Our design improves throughput from 124 and 8 to about 600 and 40 for n being 512 and 2048, respectively. Hence, on average, speed-up due to our design is five for one-block message. Moreover, our design makes it possible to execute PHC in parallel. However, software implementation always runs in sequential manner. The speed-up achieved by the PCP might be higher, where many Paillier encryptions are needed to be computed in parallel. Such a scenario always exists in many PPAs because they operate on high-dimensional matrices of data with privacy. Thus, for those PPAs in which parallel PHC computations are possible, our cryptoprocessor can perform more than one PHC computations in single latency of PCP. To explain it concretely, our PCP is instantiated more than 16 times in an average 7 family of FPGAs. This means that 16 Paillier encryptions can be computed in parallel.

5.4.4 Case study: private matching protocol with the Paillier cryptoprocessor

According to the empirical results, substantial performance improvement is attained due to our design. Furthermore, we evaluated our PCP based on an example PPA described in the following to emphasize on the contribution of our proposed hardware architecture. Computing over sets with privacy is required in several applications [173]. Secure set intersection problem can be

faced in many applications for online collaborating parties. The protocol [174] given below allows to perform a set-intersection operation between two sets of a given application with privacy.

Protocol 1 PM-Semi-Honest Intersection Protocol

Require: $\mathcal{C}$'s input is a set $X = \{x_1, \ldots, x_{k_\mathcal{C}}\}$, $\mathcal{S}$'s input is a set $Y = \{y_1, \ldots, y_{k_\mathcal{S}}\}$. The elements in the input sets are taken from a domain of size N.

1. $\mathcal{C}$ performs the following:

   (a) She chooses the secret-key parameters for a semantically-secure homomorphic encryption scheme, and publishes its public keys and parameters. The plaintexts are in a field that contains representations of the N elements of the input domain, but is exponentially larger.

   (b) She uses interpolation to compute the coefficients of the polynomial $P(y) = \sum_{u=0}^{k_\mathcal{C}} \alpha_u y^u$ of degree $k_\mathcal{C}$ with roots $\{x_1, \ldots, x_{k_\mathcal{C}}\}$.

   (c) She encrypts each of the $(k_\mathcal{C}+1)$ coefficients by the semantically-secure homomorphic encryption scheme and sends to $\mathcal{S}$ the resulting set of ciphertexts, $\{\mathsf{Enc}(\alpha_0), \ldots, \mathsf{Enc}(\alpha_{k_\mathcal{C}})\}$.

2. $\mathcal{S}$ performs the following for every $y \in Y$,

   (a) He use the homomorphic properties to evaluate the encrypted polynomial at y. That is, he computes $\mathsf{Enc}(P(y)) = \mathsf{Enc}\left(\sum_{u=0}^{k_\mathcal{C}} \alpha_u y^u\right)$.

   (b) He chooses a random value r and computes $\mathsf{Enc}(rP(y) + y)$.

   He randomly permutes this set of $k_\mathcal{S}$ ciphertexts and sends the result back to the client $\mathcal{C}$.

3. $\mathcal{C}$ decrypts all $k_\mathcal{S}$ ciphertexts received. She locally outputs all values $x \in X$ for which there is a corresponding decrypted value.

We also implemented this secure set intersection protocol proposed by [174] using the PCP as a case study. We selected the $k_\mathcal{C} = 4$ and $k_\mathcal{S} = 8$ for the set sizes of this protocol. Accordingly, $\mathcal{C}$ needs to deal with 5 parallel Paillier encryptions for providing privacy of the coefficients of the polynomial $P(y)$. The resulting set of Paillier encryptions of the coefficients are sent to $\mathcal{S}$. $\mathcal{S}$ homomorphically multiplies and adds each element of the set Y with the coefficients taken from $\mathcal{C}$ with privacy. The computed $\mathsf{Enc}(P(y))$ value is then multiplied with a random r value and added the element of the set Y with exploiting homomorphic property of the PHC. The result of this operation is sent back to $\mathcal{C}$ to find whether it is an intersection of the sets or not. $\mathcal{C}$ finally decrypts

the value and outputs the decrypted values for which there is a match with the sets.

We implemented the private matching set intersection protocol described in Protocol 1 using both our proposed PCP and the software-based platform with varying n sizes in order to show the improvements due to our design. We utilized the same methodology and hardware and software settings described previously. After estimating the times (in milliseconds) required to complete the protocol for the parameters set above, we showed them for our hardware design and software implementation in Figure 5.7.

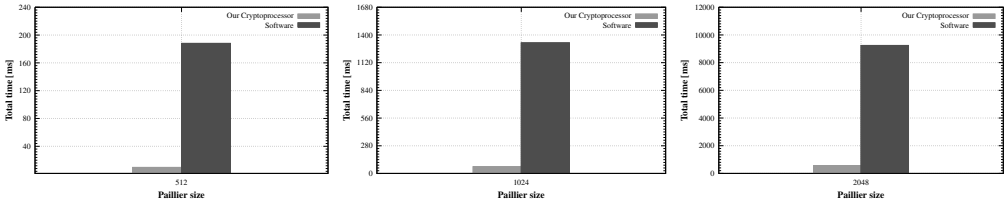


Figure 5.7: Comparing the PCP with software implementation for Protocol 1

As seen from Figure 5.7, our proposed hardware PCP significantly performs better than the software implementation of PHC for all n sizes. For instance, our proposed hardware design performs 19 times better than the software implementation when n is 1024. Similar speed-ups are observed for other n sizes. These results show that the PCP is able to perform Paillier computations efficiently in the PPAs, where time constraints are stringent. The outcomes also verify that our design definitely performs better than software implementation.

5.4.5 Conclusions and future work

Performance bottlenecks occurred in privacy-preserving applications due to Paillier homomorphic cyrptosystem make researchers to search alternative approaches in order to handle with such problems. Improvements introduced by software related approaches are limited. This led us to search hardware-based solutions. Thus, we proposed a high-speed Paillier cryptoprocessor. The proposal can be utilized to overcome computational cumbersome brought by Paillier homomorphic cyrptosystem in privacy-preserving applications. Moreover, it may gain insights for solving some computational challenges faced in such applications. Also note that our study is the first attempt on designing hardware architecture for Paillier homomorphic cryptosystem.

The design philosophy we proposed in this study led us to develop a

high-performance cryptoprocessor at different levels of security for Paillier cryptosystem. Our hardware architecture is built around mostly 48-bit datapath and it is able to compute the necessary operations for implementing encryption and decryption processes of the PHC. Despite the various control signals required for the different steps of pipelined computation, our control unit remains compact. In other words, almost all control signals are generated by means of a counter, a start signal, and its delayed versions. We manage to implement the Paillier cryptosystem (including encryption and decryption) with only a few pipeline stall thanks to the descriptions of 2^{32} radix Montgomery modular multiplication and exponentiation coprocessors and a careful organization of overall pipelined datapath. The key element of our approach to achieve high throughput design is to take advantage of the parallelism of Paillier to

1. deeply pipeline the ALU to achieve a high clock frequency,
2. find parallelism between independent tasks,
3. decrease the data dependencies and hazards to achieve more parallelism and exploit this parallelism for better design,
4. reuse same resources for different executions of the algorithm enhancing resource sharing, and
5. design pipelined datapath for the independent tasks so that each pipeline units can operate on different set of inputs at the same time.

Moreover, we performed several experiments and simulations to evaluate the proposed design with respect to latency, throughput, and time to complete encryptions and decryptions. Our empirical outcomes showed that the proposed architecture performs much better than software implementations of the Paillier homomorphic cryptosystem. Our results show that the PCP proposed in this study is excellent solution which remedy the performance problems raised by PPAs. We mainly design our architecture for PPAs in information systems. Hence, it would be interesting to conduct side-channel and fault injection attacks in future work.

6. SUMMARY AND CONCLUSIONS

In this dissertation, the problem of designing computationally efficient hardware architectures for different types of cryptographic algorithms, which protects the data transactions in computer and communication systems, is mainly studied. Due to the Moore's Law and Gilder's Law, next generation of computing and communication systems present huge amount of computational power and high bandwidth of data transfer while they need to incorporate sophisticated, powerful and cumbersome tasks which are generally computationally intensive. Security is one of such services which require huge amount of computational power. It is also significantly important in many applications where data traffic involves sensitive information. Security is realized by the use of a set of cryptographic algorithms. Thus, studying efficient hardware architectures under low-area, compact and high-speed design constraints for cryptographic algorithms is required and growing field of research.

In the first part of the dissertation, lightweight hardware architectures for block ciphers to provide secure communication for low-cost embedded systems are simply presented. Security is an important aspect for today's low cost embedded systems. Costs are also main concerns in embedded systems and pervasive computing applications. A considerable body of research has been focused on providing cryptographic functionality to resource-constrained devices, while scarce computational and storage capabilities of low-cost smart devices make the problem challenging. This emerging research area is usually referred to as lightweight cryptography which has to deal with the trade-off among security, cost, and performance. Therefore, security should be provided to such systems in a small amount of circuit size.

The second part of the dissertation is devoted to the compact implementations of different cryptographic hash functions on FPGA. In particular, the dissertation covers Secure Hash Algorithm-3 (SHA-3) finalists namely Keccak, Skein and Grøstl, which take great interests from the cryptography community, from hardware design perspective. We use the coprocessor approach, which takes advantage of the embedded memory blocks in FPGA to implement these cryptographic hash functions. This method reveals a better understanding of the computational efficiency of SHA-3 candidates in terms of resource sharing, memory access scheme, scheduling, etc.

In the last part of the dissertation, it is aimed to propose hardware architectures of high-performance cryptography for privacy-preserving collaborative filtering (PPCF). Cryptographic techniques, especially with homomorphic encryption property, are widely used popular methods for data protection in PPCF schemes. Such schemes aim to provide accurate recommendations efficiently with privacy. However, since privacy and performance are two conflicting goals, efficiency becomes worse due to the utilized homomorphic encryption methods. Although various approaches like performing some computations offline, utilizing preprocessing, employing hybrid algorithms, and so on can be used to enhance online performance, the improvements are still limited and alternative approaches should be used. In this part, we aim to propose high-performance hardware architecture for the Paillier cryptosystem.

In this dissertation, efficient hardware architectures for several cryptographic algorithms are proposed. For instance, regarding the KECCAK coprocessor, owing to the intrinsic properties of that algorithm, the serialization is possible which resulted the reduction in the area. With the proposed architecture, the area needed by the algorithm is reduced and the efficiency is increased significantly by means of proposed pipelining mechanism on FPGA. The proposed architecture also achieves high enough performance for real-time applications. The latency is decreased greatly by means of using advanced digital design techniques such as pipelining and efficient instruction scheduling for special datapath. Multiple instructions are executed in parallel since the functional units are staged via registers thanks to the serialization. This technique is mainly employed in other architecture designs of the studied cryptographic algorithms.

Unified hardware architecture implementing different cryptographic primitives is valuable since it shares resources between the cryptographic primitives and presents compact performance results, which satisfy small-area and performance requirements. We also focus on unified architectures to see resource sharing models between the algorithms which shares same execution units. The block cipher Threefish and AES, and the hash functions Skein and Grøstl are based on the same arithmetic operations, respectively. In this dissertation, it is showed that the same design philosophy allows one to design compact coprocessors for hashing and encryption. We also develop a low-area unified coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function Grøstl at all levels of security. The proposed architecture is built around an 8-bit datapath and the ALU performs a single

instruction that allows for implementing both algorithms. Despite the various addressing schemes required for the different steps of Grøstl and the AES, our control unit remains compact: all addresses are generated by means of a modulo-128 counter and a modulo-256 counter. Thanks to an alternative description of Grøstl and a meticulous organization of the memory, we manage to implement the compression function f (Algorithm 8) without any pipeline stall. At the cost of 67 Virtex-6 slices, one can add the AES functionalities to a Grøstl coprocessor. Despite of the differences between the two algorithms (size of the internal state, coefficients of the circulant matrices, key schedule, etc.), resource sharing is possible. Assuming that the security guarantees of Grøstl are at least as good as the ones of the other SHA-3 finalists, Grøstl is the best candidate for low-area cryptographic coprocessors. The results show that our unified coprocessors efficiently share the resources in hardware by achieving close performance values to the only coprocessor architectures.

Performance bottlenecks occurred in privacy-preserving applications due to Paillier homomorphic cyrptosystem make researchers to search alternative approaches in order to handle with such problems. Improvements introduced by software related approaches are limited. This led us to search hardware-based solutions. Thus, we proposed a high-speed Paillier cryptoprocessor. The proposal can be utilized to overcome computational cumbersome brought by Paillier homomorphic cyrptosystem in privacy-preserving applications. Moreover, it may gain insights for solving some computational challenges faced in such applications. Also note that our study is the first attempt on designing hardware architecture for Paillier homomorphic cryptosystem. The design philosophy we proposed in this study led us to develop a high-performance cryptoprocessor at different levels of security. Our hardware architecture is built around mostly 48-bit datapath and it is able to compute the necessary operations for implementing modular multiplication. Despite the various control signals required for the different steps of pipelined computation, our control unit remains compact. In other words, almost all control signals are generated by means of a counter, a start signal, and its delayed versions. Due to an alternative description of 2^{32} radix Montgomery modular multiplication and a careful organization of the pipelined datapath, we managed to implement the Paillier cryptosystem (including encryption and decryption) without any pipeline stall. Moreover, we performed several experiments and simulations to evaluate the propsosed design with respect to latency, throughput, and time to complete encryptions and decryptions. Our empirical outcomes showed that

the proposed architecture performs much more better than software implementations of the Paillier homomorphic cryptosystem. Our results show that the PCP proposed in this study is excellent solution which remedy the perfomance problems raised by PPAs. We mainly design our architecture for PPAs in information systems. Hence, it would be interesting to conduct side-channel and fault injection attacks in future work.

The design philosophy followed throughout this dissertation for various cryptographic algorithms allows us to develop several efficient and high-performance hardware architectures. The key element of our approach is to take advantage of the parallelism of the algorithms to:

1. deeply pipeline the ALU to achieve a high clock frequency,
2. find parallelism between independent tasks,
3. decrease the data dependencies and hazards to achieve more parallelism and exploit this parallelism for better design,
4. reuse same resources for different executions of the algorithm enhancing resource sharing, and
5. design pipelined datapath for the independent tasks so that each pipeline units can operate on different set of inputs at the same time.

Furthermore, it is described how to design compact control units thanks to a careful organization of the register file, loop unrolling, and a tight scheduling. The proposed architectures are mainly designed for embedded systems.

As future work, the Keccak coprocessor embedded in PCI-e interface or another high speed communication channel can provide fast data integrity and authentication services to the host cpu through PCI-e. The Keccak coprocessor can also be used as an efficient accelerator core in the System on Chip (SoC) designs. The resulted SoC design will accelerate the system performance for the applications such as networking and fast authenticated data communication thanks to the bus based approach provided by the MicroBlaze processor [62]. Note that the reconfigurable computing [63] provides significant speed-up which is achieved by the MicroBlaze processor-based platform by the Xilinx FPGAs. The proposed architecture can also be implemented on different platforms such as ASIC to compare its performance with other reported ASIC implementations of Keccak hash function.

The architectures proposed in this dissertation are mainly designed for embedded systems. Thus, it would be interesting to conduct side-channel and

fault injection attacks in future work. For instance, for the unified hardware architectures of AES and Grøstl, our design strategy could offer a protection against some attacks since the ALU executes the same instruction at each clock cycle.

The proposed coprocessors provide better understanding of the computational performance of the interested algorithms. It is also important to see how much parallelism exists in the algorithms from hardware design perspective. This is very substantial information for especially today's advanced parallel multi-core processor technology. Same parallelism can efficiently be exploited in such multi-core processors to enhance the performance of the algorithm. Our study also presents performance analysis of the interested algorithms.

The proposed architectures are significantly important to accelerate the performance of the cryptographic algorithms covered in this dissertation. They can be used to extend the instruction sets of a wide variety of recent processors used in embedded systems. It requires a little circuit-area while providing significant increase in the execution performance of the algorithms.

REFERENCES

[1] George Gilder. Fiber keeps its promise: Get ready. bandwidth will triple each year for the next 25. *Forbes (Apr)*, 1997.

[2] Ç.K. Koç. *Cryptographic Engineering*. Springer, 2008.

[3] F. Rodriguez-Henriquez, N.A. Saqib, A.D. Pérez, and C.K. Koc. *Cryptographic Algorithms on Reconfigurable Hardware*. Signals and Communication Technology. Physica-Verlag, 2007.

[4] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Design & Test*, 24(6):522–533, 2007.

[5] T. Blum and C. Paar. High-radix Montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[6] Jean-Luc. Beuchat. Modular Multiplication for FPGA Implementation of the IDEA Block Cipher. In *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pages 412–422, 2003.

[7] Bo Song, K. Kawakami, K. Nakano, and Y. Ito. An RSA Encryption Hardware Algorithm Using a Single DSP Block and a Single Block RAM on the FPGA. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 140–147, 2010.

[8] Nadia Nedjah and Luiza de Macedo Mourelle. A Review of Modular Multiplication Methods ands Respective Hardware Implementation. *Informatica (Slovenia)*, 30(1):111–129, 2006.

[9] Daisuke Suzuki and Tsutomu Matsumoto. How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011.

[10] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[11] Axel York Poschmann. *Lighweight Cryptography - Cryptographic Engineering for a Pervasive World*. PhD dissertation, Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Germany, 2009.

[12] Daniel W. Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M. Smith. Hummingbird: Ultra-lightweight cryptography for resource-constrained devices. *Lecture Notes in Computer Science*, 6054:3–18, 2010.

[13] Frank Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, 2002.

[14] Xinxin Fan, Guang Gong, Ken Lauffenburger, and Troy Hicks. FPGA implementations of the Hummingbird cryptographic algorithm. In *Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 48–51, 2010.

[15] Jens-Peter Kaps. Chai-tea, cryptographic hardware implementations of xTEA. *Lecture Notes in Computer Science*, 5365:363–375, 2008.

[16] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop*, pages 427–440, 2005.

[17] Pawel Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop*, pages 319–333, 2003.

[18] Panasayya Yalla and Jens-Peter Kaps. Compact FPGA Implementation of Camellia. In *19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic*, pages 658–661, 2009.

[19] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia : A 128-Bit Block Cipher Suitable for Multiple Platforms Design and Analysis. *Lecture Notes in Computer Science*, 2012:39–56, 2001.

[20] Tim Güneysu. *Cryptography And Cryptanalysis On Reconfigurable Devices - Security Implementations for Hardware and Reprogrammable Devices*. PhD dissertation, Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Germany, 2009.

[21] Thomas Eisenbarth. *Cryptography and Cryptanalysis for Embedded Systems*. PhD dissertation, Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Germany, 2009.

[22] Martin Novotný. *Time-Area Efficient Hardware Architectures For Cryptography And Cryptanalysis*. PhD dissertation, Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Germany, 2009.

[23] Ibrahim Yakut and Huseyin Polat. Privacy-preserving hybrid collaborative filtering on cross distributed data. *Knowledge and Information Systems*, 30(2):405–433, 2012.

[24] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[25] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

[26] Alexandre F. Tenca and Çetin Kaya Koç. A Scalable Architecture for Montgomery Multiplication. In *Cryptographic Hardware and Embedded Systems - CHES 1999, 1st International Workshop*, pages 94–108, 1999.

[27] Marcelo E. Kaihara and Naofumi Takagi. Bipartite modular multiplication. *Lecture Notes in Computer Science*, 3659:201–210, 2005.

[28] Kazuo Sakiyama, Miroslav Knezevic, Junfeng Fan, Bart Preneel, and Ingrid Verbauwhede. Tripartite modular multiplication. *Integration, the VLSI Journal*, 44(4):259–269, 2011.

[29] Gary C. T. Chow, Ken Eguro, Wayne Luk, and Philip Leong. A karatsuba-based montgomery multiplier. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 434–437. IEEE Computer Society, 2010.

[30] Bo Song, K. Kawakami, K. Nakano, and Y. Ito. An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 140–147, nov. 2010.

[31] David J. Wheeler and Roger M. Needham. TEA Extensions. In *Technical Report, Cambridge University*, pages 363–366, October 1997.

[32] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002.

[33] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. A low-area unified hardware architecture for the AES and the cryptographic hash function ECHO. *Journal of Cryptographic Engineering*, 1(2):101–121, 2011.

[34] FIPS. *Advanced Encryption Standard (AES)*. NIST, nov 2001.

[35] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family (version 1.3). Available at http://www.skein-hash.info, October 2010.

[36] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 3), 2011.

[37] P. Gauravaram, L.R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S.S. Thomsen. Grøstl – a SHA-3 candidate. Available at http://www.groestl.info, 2011.

[38] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Keccak Sponge Function Family: Main Document, Version 2.1*, 19th June 2010. http://keccak.noekeon.org/Keccak-main-2.1.pdf.

[39] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keccak Implementation Overview (version 3.2)*, May 2012.

[40] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications, version 2. Submission to NIST (Round 2), 2009.

[41] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. *Lecture Notes in Computer Science*, 1592:223–238, 1999.

[42] Tim Güneysu. Utilizing hard cores of modern FPGA devices for high-performance cryptography. *J. Cryptographic Engineering*, 1(1):37–55, 2011.

[43] A. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers by automata. 145:293–294, 1962. English translation in *Soviet Physics Doklady* **7** (1963), 595–596.

[44] Xilinx Inc. *UG331: Spartan-3 Generation FPGA User Guide*, June 2011.

[45] Xilinx Inc. *UG190: Virtex-5 user guide*, March 2012.

[46] Adrian Cosoroaba and Frederic Rivoallon. *WP245: Achieving Higher System Performance with the Virtex-5 Family of FPGAs*. Xilinx Inc., July 2006.

[47] Xilinx Inc. *DS531: LogiCORE Fast Simplex Link (FSL) v20 Bus*, April 2010.

[48] Xilinx Inc. *DS531: LogiCORE Processor Local Bus (PLB) v4.6*, September 2010.

[49] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann, and Leif Uhsadel. A survey of lightweight-cryptography implementations. *IEEE Des. Test*, 24(6):522–533, 2007.

[50] A. Poschmann. *Lightweight Cryptography - Cryptographic Engineering for a Pervasive World*. PhD thesis, Europäischer Universitätsverlag, 2009.

[51] Xinxin Fan, Guang Gong, K. Lauffenburger, and T. Hicks. FPGA implementations of the Hummingbird cryptographic algorithm. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 48–51, 2010.

[52] Ismail San and Nuray At. Compact Hardware Architecture for Hummingbird Cryptographic Algorithm. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 376–381, 2011.

[53] Xinxin Fan. *Efficient Cryptographic Algorithms and Protocols for Mobile Ad Hoc Networks*. PhD dissertation, Department of Electrical and Computer Engineering, University of Waterloo, Canada, 2010.

[54] Daniel W. Engels, Markku-Juhani O. Saarinen, Peter Schweitzer, and Eric M. Smith. The hummingbird-2 lightweight authenticated encryption algorithm. In *RFIDSec, RFID. Security and Privacy - 7th International Workshop, RFIDSec 2011, Amherst, USA, June 26-28*, pages 19–31, 2011.

[55] Martin Feldhofer, Manfred Aigner, Thomas Baier, Michael Hutter, Thomas Plos, and Erich Wenger. Semi-passive RFID development platform for implementing and attacking security tags. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–6, 2010.

[56] Xilinx Inc. *XAPP463: Using Block RAM in Spartan-3 Generation FPGAs*, March 2005.

[57] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. Compact implementations of BLAKE-32 and BLAKE-64 on FPGA. In J. Bian, Q. Zhou, and K. Zhao, editors, *Proceedings of the 2010 International Conference on Field-Programmable Technology–FPT 2010*, pages 170–177. IEEE Press, 2010.

[58] Xu Guo, Zhimin Chen, and Patrick Schaumont. Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors. In *Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 106–115, 2008.

[59] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. *Lecture Notes in Computer Science*, 5023:16–26, 2008.

[60] Panasayya Yalla and Jens-Peter Kaps. Lightweight Cryptography for FPGAs. In *ReConFig'09: 2009 International Conference on Reconfigurable Computing and FPGAs, Cancun, Quintana Roo, Mexico, 9-11 December 2009, Proceedings*, pages 225–230. IEEE Computer Society, 2009.

[61] F. Mace, F.-X. Standaert, and J.-J. Quisquater. FPGA Implementation(s) of a Scalable Encryption Algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):212–216, 2008.

[62] I. Gonzalez and F.J. Gomez-Arribas. Ciphering algorithms in microblaze-based embedded systems. *Computers and Digital Techniques, IEE Proceedings* -, 153(2):87 – 92, 2006.

[63] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):193 – 207, mar 2005.

[64] David J. Wheeler and Roger M. Needham. TEA, a Tiny Encryption Algorithm. In *Fast Software Encryption: Second International Workshop, FSE'94 Leuven, Belgium, 14-16 December 1994, Proceedings*, pages 363–366, 1994.

[65] P. Hamalainen, T. Alho, M. Hannikainen, and T.D. Hamalainen. 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, 2006.

[66] Elif Bilge Kavun and Tolga Yalcin. A pipelined camellia architecture for compact hardware implementation. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 305 –308, 2010.

[67] P. Israsena. Securing Ubiquitous and Low-cost RFID using Tiny Encryption Algorithm. In *Wireless Pervasive Computing, 2006 1st International Symposium on*, 2006.

[68] P. Israsena. Design and Implementation of Low Power Hardware Encryption for Low Cost Secure RFID Using TEA. In *Information, Communications and Signal Processing, 2005 Fifth International Conference on*, pages 1402–1406, 0-0 2005.

[69] Martin Feldhofer, Manfred Josef Aigner, Michael Hutter, Thomas Plos, Erich Wenger, and Thomas Baier. Semi-passive rfid development platform for implementing and attacking security tags. In IEEE, editor, *Workshop on RFID / USN Security and Cryptography*, pages 1–6, 2010.

[70] Ismail San and Nuray At. Lightweight hardware architecture for XTEA cryptographic algorithm. In *International Conference on Embedded Systems and Intelligent Technology (ICESIT) 2012*, 2012.

[71] Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In *Proceedings of the Cryptology in Africa 1st international conference on Progress in cryptology*, pages 16–26, 2008.

[72] James E. Hill. Announcing the Development of New Hash Algorithm(s) for the Revision of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard. Technical report, DEPARTMENT OF COMMERCE, January 2007.

[73] Elif Bilge Kavun and Tolga Yalcin. A lightweight implementation of Keccak hash function for radio-frequency identification applications. *Lecture Notes in Computer Science*, 6370:258–269, 2010.

[74] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. A hardware wrapper for the SHA-3 hash algorithms. In *Signals and Systems Conference (ISSC 2010), IET Irish*, pages 1–6, June 2010.

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

[76] Nuray At, Jean-Luc Beuchat, Eiji Okamoto, Ismail San, and Teppei Yamazaki. A low-area unified hardware architecture for the AES and the cryptographic hash function Grøstl. Cryptology ePrint Archive, Report 2012/535, 2012. http://eprint.iacr.org/.

[77] N. At, J.-L. Beuchat, and İ. San. Compact implementation of Threefish and Skein on FPGA. In *Proceedings of the Fifth IFIP International Conference on New Technologies, Mobility and Security–NTMS 2012*, pages 1–5, 2012.

[78] N. At, J.-L. Beuchat, E. Okamoto, I. San, and T. Yamazaki. Compact hardware implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 61(2):485–498, 2014.

[79] Hongjun Wu. The hash function JH. Submission to NIST (Round 3), 2011.

[80] J.-P. Aumasson, L. Henzen, W. Meier, and R.C.-W. Phan. SHA-3 proposal BLAKE (version 1.4). Available at http://www.131002.net/blake, January 2011.

[81] Ismail San and Nuray At. Efficient SoC design for accelerator of message authentication and data integrity on FPGAs. In *Proceedings of the International Symposium on Computing in Science & Engineering (ISCSE) 2011*, pages 409–418, 2011.

[82] K. Elizeh and N. Nicolici. Embedded memory binding in FPGAs. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 457 –462, june 2010.

[83] N. Sklavos and P. Kitsos. BLAKE hash function family on FPGA: From the fastest to the smallest. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 139–142, July 2010.

[84] Nicolas Sklavos. Multi-module hashing system for SHA-3 & FPGA integration. In *Proceedings of the 2011 21st International Conference on*

Field Programmable Logic and Applications, FPL '11, pages 162–166, 2011.

[85] P. Kitsos and N. Sklavos. On the hardware implementation efficiency of SHA-3 candidates. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 1240–1243, Dec 2010.

[86] Men Long and Intel Corporation. Implementing skein hash function on xilinx virtex-5 fpga platform, 2009.

[87] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST (Round 2), 2008.

[88] Abdulkadir Akin, Aydin Aysu, Onur Can Ulusel, and Erkay Savaş. Efficient hardware implementations of high throughput SHA-3 candidates Keccak, Luffa and Blue Midnight Wish for single- and multi-message hashing. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 168–177. ACM, 2010.

[89] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. Meurice de Dormale, and F.-X. Standaert. Compact FPGA implementations of the five SHA-3 finalists. In *Proceedings of the ECRYPT II Hash Workshop*, 2011.

[90] J. Zhai, C.M. Park, and G.-N. Wang. Hash-based RFID security protocol using randomly key-changed identification procedure. *Lecture Notes in Computer Science*, 3983:296–305, 2006.

[91] H.S. Warren. *Hacker's Delight*. Addison-Wesley, 2002.

[92] The SHA-3 zoo. http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[93] B. Jungk. Evaluation of compact FPGA implementations for all SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, March 2012.

[94] J.-P. Kaps, P. Yalla, K.K. Surapathi, B. Habib, S. Vadlamudi, and S. Gurung. Lightweight implementations of SHA-3 finalists on FPGAs. In *The Third SHA-3 Candidate Conference*, March 2012.

[95] T. Yamazaki, J.-L. Beuchat, and E. Okamoto. BLAKE-256, BLAKE-512のコンパクトな統合実装. *IEICE*暗号と情報セキュリティ実装技術小特集号, J-95A(5):416–424, 2012.

[96] B. Jungk. Compact implementations of Grøstl, JH and Skein for FPGAs. In *Proceedings of the ECRYPT II Hash Workshop*, 2011.

[97] K. Latif, M. Tariq, A. Aziz, and A. Mahboob. Efficient hardware implementation of secure hash algorithm (SHA-3) finalist - Skein. In *Proceedings of the International Conference on Computer, Communication, Control and Automation–3CA2011*, 2011.

[98] Helion Technology. FULL DATASHEET–Tiny hash core family for Xilinx FPGA. Revision 2.0 (11/06/2010).

[99] K. Järvinen. Sharing resources between AES and the SHA-3 second round candidates Fugue and Grøstl. In *The Second SHA-3 Candidate Conference*, August 2010.

[100] M. Rogawski and K. Gaj. A high-speed unified hardware architecture for AES and the SHA-3 candidate Grøstl. In *Proceedings of the* 15*th Euromicro Conference on Digital System Design*, September 2012.

[101] The Keccak Team. Personal communication, September 2012.

[102] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. A compact FPGA implementation of the SHA-3 candidate ECHO. Cryptology ePrint Archive, Report 2010/364, 2010.

[103] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 proposal: ECHO. Available at http://crypto.rd.francetelecom.com/echo, 2009.

[104] R. Benadjila, O. Billet, S. Gueron, and M.J.B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. *Lecture Notes in Computer Science*, 5912:162–178, 2009.

[105] K. Gaj and P. Chodowiec. FPGA and ASIC implementations of the AES. In Ç.K. Koç, editor, *Cryptographic Engineering*, pages 235–294. Springer, 2009.

[106] P. Hämäläinen, T. Alho, M. Hännikäinen, and T.D. Hämäläinen. Design and implementation of low-area and low-power AES encryption hardware core. In *Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools–DSD'06*, pages 577–583. IEEE Computer Society, 2006.

[107] Helion Technology. OVERVIEW DATASHEET–Ultra-low resource AES (Rijndael) cores for Xilinx FPGA. Revision 1.3.0.

[108] M. Knežević, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, Ü. Kocabaş, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki. Fair and consistent hardware evaluation of fourteen round two SHA-3 candidates. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(5):827–840, May 2012.

[109] K. Latif, M.M. Rao, A. Aziz, and A. Mahboob. Efficient hardware implementations and hardware performance evaluation of SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, March 2012.

[110] T. Yamazaki. Compact implementation of hash functions on FPGA. Master's thesis, Graduate School of Systems and Information Engineering, University of Tsukuba, 2012.

[111] Kimmo Järvinen. Sharing resources between AES and the SHA-3 second round candidates fugue and grøstl. 2010.

[112] M. Rogawski and K. Gaj. A high-speed unified hardware architecture for AES and the SHA-3 candidate Grøstl. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 568–575, 2012.

[113] Marcin Rogawski, Kris Gaj, and Ekawat Homsirikamol. A high-speed unified hardware architecture for 128 and 256-bit security levels of AES and the SHA-3 candidate Grøstl. *Microprocess. Microsyst.*, 37(6-7):572–582, August 2013.

[114] T. Güneysu and A. Moradi. Generic side-channel countermeasures for reconfigurable devices. *Lecture Notes in Computer Science*, 6917:33–48, 2011.

[115] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*, pages 193–199, 1995.

[116] Colin D. Walter. Space/time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Trans. Comput.*, 46(2):139–141, 1997.

[117] Alexandre F. Tenca and Çetin Kaya Koç. A Scalable Architecture for Montgomery Multiplication. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 94–108, 1999.

[118] Alexandre F. Tenca, Georgi Todorov, and Çetin Kaya Koç. High-radix design of a scalable modular multiplier. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '01, pages 185–201, 2001.

[119] Jean-Luc Beuchat and Jean-Michel Muller. Automatic Generation of Modular Multipliers for FPGA Applications. *IEEE Trans. Computers*, 57(12):1600–1613, 2008.

[120] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, page 224, 2006.

[121] Daniel J. Bernstein. Multidigit Multiplication for Mathematicians, 2001.

[122] Nadia Nedjah and Luiza de Macedo Mourelle. A review of modular multiplication methods ands respective hardware implementation. *Informatica (Slovenia)*, 30(1):111–129, 2006.

[123] F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 250–255, 2009.

[124] Daisuke Suzuki and Tsutomu Matsumoto. How to maximize the potential of fpga-based dsps for modular exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011.

[125] Shuli Gao, Dhamin Al-Khalili, and Noureddine Chabini. Efficient scheme for implementing large size signed multipliers using multigranular embedded dsp blocks in fpgas. *Int. J. Reconfig. Comput.*, pages 1:1–1:11, 2009.

[126] S. Bo, K. Kawakami, K. Nakano, and Y. Ito. An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA. *International Journal of Networking and Computing*, 1(2):277–289, 2011.

[127] Bo Song, Y. Ito, and K. Nakano. CRT-based DSP decryption using Montgomery modular multiplication on the FPGA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 532–541, 2011.

[128] J.N. Bautista, O. Alvarado-Nava, and F.M. Perez. A mathematical coprocessor of modular arithmetic based on a FPGA. In *Technologies Applied to Electronics Teaching (TAEE), 2012*, pages 32–37, 2012.

[129] R. L. Rivest, L. Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–180, 1978.

[130] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[131] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007(1):15:1–15:15, 2007.

[132] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[133] Ibrahim Yakut and Huseyin Polat. Arbitrarily distributed data-based recommendations with privacy. *Data & Knowledge Engineering*, 72:239–256, 2012.

[134] Ivan Damgård, Mads Jurik, and JesperBuus Nielsen. A generalization of Paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9:371–385, 2010.

[135] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. A. Thorpe. Practical secrecy-preserving, verifiably correct and trustworthy auctions. In *Proceedings of the 8th international conference on Electronic commerce*, pages 70–81, 2006.

[136] Anirban Basu, Jaideep Vaidya, Theo Dimitrakos, and Hiroaki Kikuchi. Feasibility of a privacy preserving collaborative filtering scheme on the Google App Engine: A performance case study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 447–452, 2012.

[137] J.-L. Beuchat. Modular multiplication for FPGA implementation of the IDEA block cipher. In *Proceedings of the 14th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 412–422, 2003.

[138] Guilherme Perin, Daniel Gomes Mesquita, and J. B. Martins. Montgomery modular multiplication on reconfigurable hardware: Systolic versus multiplexed implementation. *International Journal of Reconfigurable Computing*, 2011:6:1–6:10, 2011.

[139] C. D. Walter. Space/time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139–141, 1997.

[140] Bishwaranjan Bhattacharjee, Naoki Abe, Kenneth Goldman, Bianca Zadrozny, Vamsavardhana R. Chillakuru, Marysabel del Carpio, and Chid Apte. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, 2006.

[141] Yaping Li. *Privacy Preserving Joins on Secure Coprocessors*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[142] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3):683–695, mar 2001.

[143] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, aug 2009.

[144] J. Teubner, R. Muller, and G. Alonso. Frequent item computation on a chip. *IEEE Transactions on Knowledge and Data Engineering*, 23(8):1169–1181, aug. 2011.

[145] Ying-Hsiang Wen, Jen-Wei Huang, and Ming-Syan Chen. Hardware-enhanced association rule mining with hashing and pipelining. *IEEE Transactions on Knowledge and Data Engineering*, 20(6):784–795, jun 2008.

[146] J. Sawada and H. Nishi. Hardware acceleration and data-utility improvement for low-latency privacy preserving mechanism. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications*, pages 499–502, 2012.

[147] Ismail San and Nuray At. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture*, http://dx.doi.org/10.1016/j.sysarc.2013.10.013.

[148] Ismail San and Nuray At. On increasing the computational efficiency of long integer multiplication on FPGA. In *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1149–1154, 2012.

[149] Joachim von zur Gathen and Jamshid Shokrollahi. Efficient fpga-based karatsuba multipliers for polynomials over f_2. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography*, pages 359–369, 2005.

[150] E.-H.Y. Wajih, M. Mohsen, Z. Medien, and B. Belgacem. Efficient hardware architecture of recursive Karatsuba-Ofman multiplier. In *Design and Technology of Integrated Systems in Nanoscale Era, 2008. DTIS 2008. 3rd International Conference on*, pages 1–6, 2008.

[151] Lejla Batina, Siddika Berna Örs, Bart Preneel, and Joos Vandewalle. Hardware architectures for public key cryptography. *Integr. VLSI J.*, 34(1-2):1–64, 2003.

[152] Koji Nakano, Kensuke Kawakami, and Koji Shigemoto. Rsa encryption and decryption using the redundant number system on the fpga. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, 2009.

[153] S.H. Tang, K.S. Tsui, and P.H.W. Leong. Modular exponentiation using parallel multipliers. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 52–59, 2003.

[154] Ismail San, Nuray At, Ibrahim Yakut, and Huseyin Polat. Accelerating privacy-preserving applications via Paillier cryptoprocessor. *Submitted for publication.*

[155] Lorrie Faith Cranor. Designing personalized user experiences in ecommerce. chapter 'I didn't buy it for myself': privacy and Ecommerce personalization, pages 57–73. Kluwer Academic Publishers, 2004.

[156] OECD. *Guidelines on the protection of privacy and transborder flows of personal data*, volume 11. 1981.

[157] YOUWEN Zhu, LIUSHENG Huang, TSUYOSHI Takagi, and MINGWU Zhang. Privacy-preserving OLAP for accurate answer. *Journal of Circuits, Systems and Computers*, 21(01):1250009, 2012.

[158] Ibrahim Yakut and Huseyin Polat. Privacy-preserving Eigentaste-based collaborative filtering. *Lecture Notes in Computer Science*, 4752:169–184, 2007.

[159] Tanzima Hashem, Lars Kulik, and Rui Zhang. Countering overlapping rectangle privacy attack for moving kNN queries. *Information Systems*, 38(3):430–453, 2013.

[160] Cihan Kaleli and Huseyin Polat. P2P collaborative filtering with privacy. *Turkish Journal of Electrical Engineering & Computer Sciences*, 18(1):101–116, 2010.

[161] J. A. M. Naranjo, L. G. Casado, and M. Jelasity. Asynchronous privacy-preserving iterative computation on peer-to-peer networks. *Computing*, 94(8–10):763–782, 2012.

[162] Dinusha Vatsalan, Peter Christen, and Vassilios S. Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 2012.

[163] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness Knowledge-based Systems*, 10(5):557–570, 2002.

[164] Xin Jin, Nan Zhang, and Gautam Das. ASAP: Eliminating algorithm-based disclosure in privacy-preserving data publishing. *Information Systems*, 36(5):859–880, 2011.

[165] Emmanouil Magkos, Manolis Maragoudakis, Vassilis Chrissikopoulos, and Stefanos Gritzalis. Accurate and large-scale privacy-preserving data mining using the election paradigm. *Data & Knowledge Engineering*, 68(11):1224–1236, 2009.

[166] Alper Bilge and Huseyin Polat. A comparison of clustering-based privacy-preserving collaborative filtering schemes. *Applied Soft Computing*, 13(5):2478–2489, 2013.

[167] Alper Bilge and Huseyin Polat. Improving privacy-preserving NBC-based recommendations by preprocessing. In *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pages 143–147, 2010.

[168] Edson Pedro Ferlin, Heitor Silverio Lopes, Carlos R. Erig Lima, and Mauricio Perretto. A FPGA-based reconfigurable parrallel architecture for high-performance numerical computation. *Journal of Circuits, Systems and Computers*, 20(05):849–865, 2011.

[169] D. Bayhan, S.B. Ors, and G. Saldamli. Analyzing and comparing the Montgomery multiplication algorithms for their power consumption. In *Proceedings of 2010 International Conference on Computer Engineering and Systems*, pages 257–261, 2010.

[170] Bernard Murphy and Sanjay Churiwala. SpyGlass application in an FPGA to ASIC conversion flow. White paper, Atrenta, 2009.

[171] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, feb. 2007.

[172] A.A. Hiasat and H.S. Abdel-Aty-Zohdy. A high-speed division algorithm for residue number system. In *IEEE International Symposium on Circuits and Systems, ISCAS '95*, volume 3, pages 1996–1999, 1995.

[173] Lea Kissner and Dawn Song. Privacy-preserving set operations. *Lecture Notes in Computer Science*, 3621:241–257, 2005.

[174] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. *Lecture Notes in Computer Science*, 3027:1–19, 2004.