Center for Research and Advanced Studies

Computer Science Department

Implementation of Cryptographic Algorithms for High-speed and Constrained Devices

A dissertation submitted by

Jose Abraham Bernal Gutierrez

For the degree of

Doctor in Computer Science

Advisors:

Dr. Francisco Rodr guez Henr quez
Dr. Cuauhtemoc Mancillas Lopez

Mexico City April 2023

Centro de Investigación y de
Estudios Avanzados del IPN

Departamento de Computación

Implementacion de algoritmos criptogra cos para dispositivos de alta velocidad y dispositivos restringidos

Tesis que presenta

Jose Abraham Bernal Gutierrez

para obtener el grado de

Doctor en Ciencias en Computacion

Directores de tesis:

Dr. Francisco Rodr guez Henr quez
Dr. Cuauhtemoc Mancillas Lopez

Ciudad de México Abril 2023

Acknowledgments

Agradezco al Consejo de Ciencia y Tecnología (CONACYT) el apoyo brindado para realizar mis estudios y así alcanzar el grado de doctor en ciencias en computación.

También agradezco al Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional (Cinvestav IPN) la oportunidad de ingresar al programa de doctorado en el departamento de computación.

Resumen

Esta tesis presenta los trabajos desarrollados centrados en la criptografía simétrica y la criptografía de clave pública. Hoy en día, el despliegue de muchos dispositivos con diferentes capacidades da lugar a entornos heterogéneos. Estos entornos deben garantizar la seguridad de los datos recogidos y procesados por los dispositivos en el entorno desplegado. Para proteger los dispositivos en función de sus capacidades, implementamos enfoques criptográficos en distintos proyectos que parten de la crip-

sobre dos enfoques matemáticos, el primero basado en el esquema RSA con RNS y la implementación general del módulo multiplicador base para mejorar la velocidad y explotar los DSPs disponibles en FPGAs con un enfoque en un multiplicador de propósito general que acepta cualquier número primo de 512 bits. Esta propuesta presenta algunos problemas debido a la generalización de la implementación. Puede utilizar muchos dispositivos si se desea alcanzar altas velocidades; por otro lado, puede utilizar menos y lograr una velocidad menor. El segundo multiplicador utiliza los enteros para realizar las multiplicaciones necesarias en un esquema de criptografía de curvas elípticas ECC centrado en la curva ECC25519; esta segunda propuesta utiliza tres algoritmos de multiplicación el primero es RNS, el método de multiplicación Karatsuba, y la multiplicación del método de la escuela. Comparamos las tres propuestas y presentamos resultados interesantes cuando logramos paralelizar simultáneamente algunas operaciones y procesar múltiples resultados intermedios con arquitecturas dedicadas para conseguir alta velocidad con bajo consumo de recursos.

Abstract

This thesis presents the studies that we have developed focused on symmetric cryptography and public-key cryptography. Today, deploying many devices with different capabilities gives rise to heterogeneous environments. These environments must ensure the security of the data harvested and processed by the devices in the deployed environment. To protect the data stored and harvested by the devices, we implement cryptographic approaches into different focuses emanating from symmetric cryptography and public-key cryptography. The scope of this thesis consists of implementing both methods into devices with constrained resources such as memory, speed, and power and, therefore, into high-speed devices like FPGAs.

At the beginning of this thesis work, we present symmetric cryptography implementations. Some of the architectures described participated in the NIST LWC contest in standardizing Lightweight cryptography for constrained devices. Therefore, we made some implementations of LWC algorithms on hardware with a constrained focus on FPGAs and the architectures we proposed based on their algorithms. Some developed architectures and implementations were ranked among the first twenty places in a benchmarking performed by George Mason University. Finally, we present a software implementation of the Speedy algorithm, designed for hardware performance, but in software has a poor performance. This implementation uses a technique named "bitslice" which consists of rearranging the data bits to be processed as fast as possible, improving the speed and performance against traditional software implementations, and achieving an acceptable performance on microcontrollers like the ARM family, which has constrained resources and a small amount of memory available compare with conventional microprocessors.

In the remaining chapters, we work on public-key cryptography and propose two arithmetic multipliers for use in public-key cryptography on two mathematical approaches, the first based on the RSA scheme with RNS and the general implementation of the base multiplier module to improve the speed and exploit the digital signal processors

(DSPs) available in FPGAs with a focus on a general purpose multiplier which accepts arbitrary 512 bits prime number. This approach has some issues due to the generalization of the implementation. It can use many devices if achieving high speeds is desired; on the other hand, it can use fewer and achieves a lower rate. The second multiplier uses the integers to perform the multiplications needed into a scheme of elliptic curve cryptography ECC focused on the curve ECC25519; this second proposal uses three multiplication algorithms: Residue Numeric system (RNS), the Karatsuba multiplication method, and the schoolbook multiplication. We compared the three proposals and presented exciting results when we could simultaneously parallelize some operations and process multiple intermediate results with dedicated architectures to achieve high speed with low resource consumption.

Contents

1 Introduction 1
- 1.1 Motivation . . . 4
- 1.2 Problem statement . . . 4
- 1.3 Objectives . . . 5
- 1.4 State-of-the-art . . . 6
  - 1.4.1 Security on constrained devices . . . 8
  - 1.4.2 Security on high-speed devices . . . 11
- 1.5 Summary . . . 12

I Theoretical Background 13

2 Preliminaries 15
- 2.1 Mathematical Background . . . 15
- 2.2 Algebra . . . 15
  - 2.2.1 Groups . . . 16
  - 2.2.2 Rings . . . 16
  - 2.2.3 Fields . . . 17
  - 2.2.4 Extension of fields . . . 17
- 2.3 Integer arithmetic operations . . . 18
  - 2.3.1 Addition and Subtraction . . . 20
  - 2.3.2 Multiplication . . . 20
  - 2.3.3 Modular reduction . . . 22
- 2.4 Greatest Common Divisor (GCD) . . . 24
- 2.5 Chinese Residue Theorem CRT . . . 25
- 2.6 Summary . . . 26

3 Cryptography 27
3.1 Advanced Encryption Standard (AES) . . . 28
3.1.1 Byte Substitution layer . . . 32
3.1.2 ShiftRows . . . 32
3.1.3 MixColumns . . . 34
3.1.4 Add Round Key . . . 34
3.1.5 Key Schedule . . . 35
3.1.6 AES 128-bit Key schedule . . . 35
3.2 Block Cipher Modes of Operation . . . 36
3.2.1 Electronic Codebook (ECB) . . . 36
3.2.2 Cipher Block Chaining Mode (CBC) . . . 37
3.2.3 Counter Mode (CTR) . . . 39
3.3 Message Authentication Codes (MACs) . . . 40
3.3.1 HMAC . . . 41
3.3.2 MAC from block ciphers . . . 43
3.3.3 MAC Verification . . . 44
3.4 Hash Functions . . . 44
3.5 Lightweight Cryptography . . . 44
3.5.1 Performance . . . 45
3.5.2 Lightweight primitives . . . 46
3.5.3 Lightweight Block ciphers . . . 46
3.5.4 Lightweight MACs . . . 46
3.6 Authenticated encryption with associated data . . . 47
3.7 Hardware API for Lightweight Cryptography . . . 48
3.8 Public Key Cryptography . . . 52
3.9 RSA . . . 53
3.9.1 RSA encryption scheme . . . 54
3.9.2 RSA signature scheme . . . 55
3.10 Elliptic curve scheme . . . 56
3.10.1 Groups . . . 56
3.10.2 Generalization of discrete logarithm problem . . . 57
3.10.3 Elliptic curve groups . . . 57
3.10.4 Key generation in elliptic curves . . . 58
3.10.5 Encryption scheme with elliptic curves . . . 58
3.11 Summary . . . 59

4 Field Programmable Gate Array (FPGA) and Advance RISC Machine (ARM) technologies 61
4.1 Field Programmable Gate Arrays (FPGAs) 62
4.1.1 Logic elements 62
4.2 Digital Signal Processor (DSP) 67
4.2.1 DSP48E2 71
4.2.2 Xilinx FPGA Families 73
4.3 Architecture of Xilinx 7 family 75
4.3.1 Configurable Logic Block (CLB) 75
4.3.2 Look-Up Table (LUT) 78
4.4 Advance RISC Machine (ARM) 80
4.4.1 Register Set 83
4.5 ST Microelectronics 84
4.5.1 Memory protection unit 88
4.5.2 General-purpose I/O (GPIO 88
4.5.3 Direct memory access (DMA) 88
4.5.4 Random number generator (RNG) 89
4.5.5 AES in hardware 90

II Symmetric Key Cryptography 93

5 Lightweight authenticated encryption with associated data in hardware 95
5.1 Authenticated Encryption with Associated Data 96
5.2 GMU LWC Interface 97
5.3 Implemented Authenticated Ciphers 100
5.3.1 Preliminaries 100
5.3.2 Hardware design principles 100
5.3.3 LOTUS and LOCUS 100
5.3.4 LOTUS 102
5.3.5 LOCUS 103
5.3.6 ESTATE 106
5.3.7 COMET 111
5.3.8 Oribatida 114
5.4 Results 116
5.4.1 Discussion of results 121
5.5 Summary 123

6 Speedy Block cipher on ARM with Bitslice 125
6.1 Speedy block cipher . . . 125
6.1.1 Speedy S-Box . . . 127
6.1.2 Speedy specification . . . 128
6.1.3 Round function . . . 129
6.1.4 Key schedule . . . 130
6.2 Bitslicing . . . 130
6.2.1 Substitution Box (SB) . . . 131
6.2.2 Shift Columns (SC) . . . 132
6.2.3 MixColumns (MC) . . . 132
6.2.4 AddRoundKey (AR) and AddRoundConstant (AC) . . . 133
6.3 Results . . . 133
6.3.1 Differential Attack . . . 135

III Public Key Cryptography 137

7 A DSP-based FPGA design and implementation of a fast RNS multiplier 139
7.1 Our Contributions . . . 140
7.2 Preliminaries . . . 141
7.2.1 Notation . . . 141
7.2.2 Montgomery reduction . . . 142
7.2.3 Residue Number System and Modular Arithmetic . . . 143
7.2.4 RNS Montgomery modular reduction . . . 147
7.2.5 FPGA and DSP technology . . . 149
7.3 Related works . . . 152
7.4 Design of a DSP48 -based architecture for a field multiplier . . . 153
7.4.1 Basic RNS multiplier with reduction . . . 154
7.4.2 Multiplier array *MulDM* . . . 159
7.4.3 RNS addition with reduction . . . 159
7.4.4 Addition tree . . . 160
7.5 Implementation . . . 161
7.5.1 Implementation of the modular reduction Algorithm 19 . . . 162
7.5.2 Montgomery Implementation . . . 163
7.6 Results . . . 163
7.6.1 RNS word multiplier with reduction . . . 164
7.6.2 Discussion and comparison . . . 166

8 Hardware accelerator for the elliptic curve ECC25519 169
8.1 Karatsuba Proposal 170
8.2 Schoolbook proposal 176
8.3 RNS 181
8.4 Results 184
8.5 Summary 187

IV Summary 189

9 Conclusions 191
9.1 Lightweight authenticated encryption with associated data 191
9.2 Speedy Block cipher on ARM-M4 with Bitslice 191
9.3 A DSP-based FPGA design and implementation of a fast RNS multiplier 192
9.4 Hardware accelerator for the elliptic curve ECC25519 192

10 Future Work 193

List of Figures

3.1 Communication between parties using symmetric cryptography over an insecure channel 28
3.2 General use of AES in encryption mode. 29
3.3 AES input array as an initial state. 30
3.4 Matrix representation for a state in AES. 30
3.5 AES encryption diagram 31
3.6 Shift Rows (SR) input state. 33
3.7 Shift Rows (SR) output state. 34
3.8 Electronic codebook operation mode (ECB). 36
3.9 CBC block diagram for encryption and decryption. 38
3.10 Counter mode general diagram 39
3.11 Message authentication codes general diagram and verification 40
3.12 HMAC diagram 42
3.13 MAC implementation based on CBC operation mode 43
3.14 Architecture diagram for single pass core used in AEAD by GMU. . . 50

4.1 Two different four input LUTs 63
4.2 LUT with register and *XOR* 63
4.3 FPGA Island connection between CLBs 64
4.4 Nearest neighbor structure 64
4.5 Logic blocks in an island fashion with connection block and switch boxes in the same architecture 65
4.6 Programmable connection block 66
4.7 Hybrid structure of nearest neighbor and segmented structure 66
4.8 Hierarchical structure with a cluster of logic blocks 67
4.9 DSP48E1 architecture. 68
4.10 DSP pipeline configuration window. 70
4.11 DSP48E2 internal architecture 72

4.12 DSP interconnection . . . 73
4.13 Internal ALU found inside a DSP slice . . . 74
4.14 CLB internal arrangement and interconnection matrix. . . . 76
4.15 ASMBL architecture with components as columns. . . . 77
4.16 CLBs and Slices with Carry inputs and outputs. . . . 79
4.17 Slice M architecture. . . . 80
4.18 ARM M4 general specs . . . 82
4.19 ARM general purpose registers. . . . 83
4.20 ST Nucleo-144 board characteristics. . . . 85
4.21 ST Nucleo-144 schematic with ST-Link. . . . 86
4.22 STM32L4A6ZG ARM microcontroller internal architecture. . . . 87
4.23 Random Number Generator (RNG) embedded in STM32L4A6ZG. . . . 89
4.24 AES block diagram. . . . 90

5.1 Top-level block diagram of LWC core (based on the scheme found at [69]). Here, sw= external key width, w= external data width, ccsw= internal key width and ccw= internal data width. . . . 99
5.2 Block diagram for associated data processing and Tag generation for LOCUS and LOTUS. . . . 101
5.3 Block Diagram of LOTUS mode for encryption. . . . 103
5.4 LOTUS/LOCUS hardware architecture. . . . 104
5.5 Block Diagram of LOCUS mode for encryption. . . . 105
5.6 tweGift-64 design implementation, for a 32-bit datapath. . . . 106
5.7 ESTATE Deterministic Authenticated Cipher. . . . 108
5.8 Architecture for ESTATE. . . . 109
5.9 Block diagram for COMET. . . . 112
5.10 COMET architecture for 8-bit and 32-bit. . . . 113
5.11 Encryption version of Oribatida AEAD Algorithm (scheme based on the found at [16]). . . . 115
5.12 LWC architecture for Oribatida AEAD. . . . 116

6.1 Latency of 15nm NanGate (image from Speedy paper [80] page 518, figure 1) . . . 127
6.2 Speedy 6-bit Substitution Box architecture with two-level NAND trees and input buffers (image from Speedy paper [80] page 522, figure 2) . . . 128
6.3 Speedy algorithm as block diagram (image from Speedy paper [80] page 524) . . . 129

7.1 Simplified DSP48E1 architecture . . . 151

7.2 Two-word schoolbook multiplication method 155
7.3 Proposed RNS component-wise multipliers 157
7.4 RNSModule components array 159
7.5 Addition with reduction 160
7.6 Addition tree 161
7.7 General architecture performing the modular reduction of Algorithms 19 and 22 162

8.1 Karatsuba diamond adder 171
8.2 Karatsuba diamond with DSPs 172
8.3 Karatsuba diamond results 172
8.4 Karatsuba diamond results from DSPs 173
8.5 Diamond of the second level of DSPs results 173
8.6 Last addition for Karatsuba diamond adder with carry 173
8.7 Karatsuba multiplier critical path 175
8.8 Diamond proposal for schoolbook method 177
8.9 Diamond with DSPs 178
8.10 Diamond of results from the first level of additions 178
8.11 Diamond results on the first level of additions on DSPs 179
8.12 Diamond results from the second level of additions 179
8.13 Carry addition for the inner diamond in schoolbook 179
8.14 Addition of the three inner diamonds to get the final result 180
8.15 Critical path schoolbook method 180
8.16 Critical path RNS proposal 183

List of Algorithms

1 Addition of non-negative multi-precision integers 20
2 Subtraction of non-negative multi-precision integers 20
3 Multiplication of positive multi-precision integers 21
4 Karatsuba multiplication of positive multi-precision integers 22
5 Montgomery reduction *Redc* of multi-precision integers 23
6 Euclid extended *gcd* of positive integers 24
7 Chinese remainder 26
8 AES algorithm with *Nb* as the number of bytes processed 29
9 Key generation in RSA. 54
10 Simple RSA encryption. 54
11 Simple RSA decryption. 55
12 Simple RSA signature. 55
13 Simple RSA signature verification. 55
14 Simple Key generation on elliptic curves. 58
15 Encryption "ElGamal" in elliptic curves. 59
16 Decryption "ElGamal" in elliptic curves. 59
17 ShiftColumns (SC) assembly code. 132
18 Montgomery Multiplication. 142
19 RNS Modular Reduction [64]. 147
20 RNS Jeljeli Modular Reduction. 150
21 Basic RNS reduction module. 154
22 RNS Montgomery Modular Reduction in HW. 156

List of Tables

1.1 Classification and characteristics 7
1.2 Layers, challenges, and requirements in security 10

3.1 Substitution Box represented with hex values. 33
3.2 NIST Lightweight Cryptography Standardization process timeline . . 47

4.1 Summary of the difference between characteristics of the DSP48E1 and DSP48E2 75
4.2 Comparative table of Xilinx 7-series resources 75

5.1 Valid segments in LWC API communication protocol. 98
5.2 Utilization of resources, throughput and TPA for implemented architectures on the xc7a12tcsg325-3 FPGA. 117
5.3 Overhead in resources for complete design (LWC+Cryptocore) compared with Cryptocore alone. %usage is the percentage of utilization of available resources on the xc7a12tcsg325-3 FPGA. 119
5.4 Comparison of our LWC implementations regarding to the existing literature. 121

6.1 Speedy 6-bit Substitution Box 129
6.2 Speedy bit-slice representation into the six registers as individual bits with b_j^i where i refers to the $i-th$ block and j refers to the $j-th$ bit of the bock i 131
6.3 Table with the results from [80] and our implementation, we only measure the clock cycles for the encryption process via the microcontroller embedded instructions. 135

7.1 Comparative table for the three proposed RNS word multipliers with reduction 164
7.2 Total resources of both implementations 165

7.3 Comparative table with total memory required 166

8.1 Karatsuba multiplication with reduction, resources, and speed 185
8.2 SchoolBook multiplication with reduction, resources, and speed . . . 185
8.3 Adder with reduction, resources, and speed 186
8.4 Subtraction with reduction, resources, and speed 186

Chapter 1

Introduction

Since the beginning of modern societies, security has been a necessary practice to protect sensitive information; otherwise, if that information becomes discovered by an opponent or adversary, it could compromise the security of a society, system, and even the user's health.

In ancient Rome, the emperor Julio Caesar put into practice a method to encrypt the information he wanted to send to his generals by performing a shift on the alphabet to send confidential information to them. If an opponent intercepts an encrypted letter, she cannot decrypt it; the encryption used is Caesar method encryption.[1] ."

Miniaturization and technological advances help developing new devices with computational power like those used in Wireless Sensor Networks (WSNs) and the Internet of Things (IoT). These devices have embedded a small computer with constrained resources as a primary processing system. Usually, these devices assist humans with computational power embedded in everyday objects [122] like a cup, microwave, refrigerator, Tv, Etc. Likewise, they communicate through a ubiquitous network to recompile information from the environment. In this scenario, the user can perform actions through an interface to modify the environment behavior, such as turning on/off lights or using any other device to perform any action the user wants, depending on the device's capacities [116].

Ubiquitous computing involves a large-scale deployment of embedded devices in different areas and scenarios like e-healthcare, farming, smart home, and smart cities. Nevertheless, the security aspects in different applications as performance and low

[1] https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar "reviewed on 25/10/21"

power consumption, are essential [100]. Therefore, it is necessary to help ubiquitous computing to improve its security, and cryptography is the ideal science that protects data and information against no authorized third parties [85, 126].

Cryptology is a general area that comprises two main sub-areas. The first one is cryptography, which is in charge of protecting data and communications from third parties, and cryptanalysis breaks the security of a cryptographic system and helps to ensure that a cryptographic system is secure; the most relevant security aspects are

- Confidentiality: the information remains secret against third parties.
- Authentication: an authorized party can generate and access an information source.
- Data Integrity: protect the information against third parties modifications.

Lightweight cryptography [103] becomes a subarea of cryptography, focusing on algorithms oriented for devices with constrained resources to protect data streamed or stored in these devices. One solution is using block ciphers since they show excellent performance on embedded devices and provide confidentiality and data integrity in the implemented devices [136, 128, 56].

When developing a new device or application, security is not prioritized in information systems; data integrity or secure communication with other systems or devices is not essential. Although therefore, developing co-processors to improve the device's security and secure the sensitive information it manipulates and stores inside itself is a must. The developers must always be aware of the resources available and remember that the application could need more resources than those available for cryptography solutions.

The constrained resource devices used by Wireless Sensor Networks and the Internet of Things; are deployed in hostile environments where an adversary can access them and their data completely. In this scenario, the adversary can attack devices directly to obtain their secret information. Those attacks are known as side-channel attacks, which measure the power consumption of a device when it performs a cryptography task, for example, the encryption of a block of data. Unfortunately, implementing cryptographic algorithms does not warrant a countermeasure against these attacks. An inadequate implementation and poor countermeasures allow attackers to get information about the devices and sometimes complete access and control to confidential data.

Today there are more communication scenarios using devices with constrained resources; these include tasks performed by applications that use devices such as RFID tags, sensors, and wireless sensors, all of them over different applications like healthcare, and the Internet of Things (IoT), among some high-speed devices like regular processors and servers with a large number of resources available. Constrained resource devices are typically present in these wireless environments, and the sensitive data exchanged among them demands security services such as confidentiality, integrity, and authentication. Unfortunately, public-key cryptography is not feasible due to its highest hardware requirements and the computational power needed to run the algorithms in this scope, and their use is focused on high-speed devices like FPGAs due to their speed and specialized characteristics.

An efficient and high-performance way to provide some of the previously listed services is by implementing an authenticated encryption algorithm with associated data (AEAD), which reduces resources on constrained devices and improves the versatility of high-speed implementations. However, the standard cryptographic algorithms to provide these services are unsuitable for constrained resource devices due to the same hardware and computational power requirements. Therefore, public-key cryptography is present on almost all devices with no restrictions and can handle data harvested from constrained devices, and due to the high speed achieved by high-speed devices, public-key cryptography is used in this kind of scenario.

Hardware restrictions have several drawbacks regarding the area usage (resources available) and power consumption that needs to be solved in constrained devices used in wireless environments without any permanent power source, and it is not always possible to ensure enough power is available.

These devices can use a battery (or another portable power source) with a limited amount of stored energy becomes a restriction to use traditional cryptography algorithms to provide security services. Also, thinking in lightweight cryptography, the developers must reduce the duty cycle of the implemented algorithms for a limited amount of time to increase the power source lifetime available for other tasks.

The deployed devices are in heterogeneous places where the area used by the cryptographic algorithms is constrained. Therefore, their footprint should be minimal to perform all tasks with low power consumption to increase the time to perform the device's principal tasks online.

Constrained resources in ubiquitous environments need lightweight cryptography primitives to offer security services as the device application requires. Additionally, other tasks (such as the data exchange) would prioritize using the available resources (storage if available, and other essential tasks), leaving limited resources available for cryptographic primitives. On the other hand, high-speed devices can handle many tasks thanks to their available sources, allowing them to keep running as long as possible to perform critical tasks.

A lightweight cryptography primitive should use the lowest possible hardware resources with low power consumption without adding more hardware complexity or resource consumption, leaving most resources available for other critical tasks. On the other hand, public-key cryptography and traditional symmetric-key cryptography can take advantage of the resources available on high-speed devices to handle the data encryption and other security services required or queried from constrained devices and other sources.

1.1 Motivation

Recently, constrained devices in different applications have grown in ubiquitous environments, and devices with computer capacity in reconfigurable hardware and heterogeneous scenarios use sensors and actuators. However, most of the algorithms developed with a lightweight focus, known as Lightweight cryptography (LWC), are focused on being used in constrained resources devices. Therefore, the meaning of "lightweight" is not always the same because they must cover several points, like reducing power consumption and the area used by the cryptography hardware. On the other hand, high-speed implementations are required to handle the communication of large amounts of constrained devices and to provide some expensive security services.

1.2 Problem statement

The suitable cryptographic algorithms for lightweight applications mark a trend in lightweight cryptography. They offer security to tiny electronic devices with computational power, like wireless sensor networks (WSN) and Internet Things (IoT) devices. Moreover, this aims at applications where devices have a limited power source, like a battery. Therefore, high-speed hardware always is needed to handle a large amount of data and to provide security services to other devices.

The area used by any device and its internal components to perform its tasks directly affects the power consumption at run time because this represents the required power by the device to perform any task in a time-lapse.

Consequently, if the total area of the design and the internal components is small enough on area consumption, it shows a growing power consumption on the same tasks because the implemented algorithm requires more run time to perform cryptography tasks.

The works of [45, 88] show that some implementations could use a large amount of area, and sometimes these implementations are more energy-efficient when running lightweight algorithms. However, this trade-off implies no area restrictions and execution time due to the small number of plaintexts the crypto-core needs to process, i.e., power consumption is the primary goal.

On the other hand, other works focus on power efficiency, as shown in work realized by [136], and some others aim for better performance regarding the area used by the implemented algorithm.

1.3 Objectives

To implement lightweight cryptography (LWC) algorithms on various constrained resource devices, focus on high-speed hardware and its characteristics to develop both high-speed and low-speed hardware. Moreover, perform a performance analysis of the implemented algorithms on the processors, micro-controllers, and FPGAs used.

- Implement public-key algorithms on high-speed devices focusing on key pair generation and basic mathematical operations.
- Implement hardware-oriented algorithms using techniques to improve the algorithm's performance on embedded devices.
- Implement LWC algorithms by being aware of the resources focus on the low-speed microcontrollers and FPGAs.

- Implement and design a lightweight and public-key cryptography crypto-processor on FPGA.
- Test our proposals and implementations.
- Compare the performance of our implementations and proposals against others from state-of-the-art.

1.4 State-of-the-art

Lightweight Cryptography provides security solutions to devices with constrained resources[1]. In addition, the academic community develops research that includes the implementation of cryptography standards, design, and analysis of cryptography algorithms and protocols.

Its focus is on embedded devices like wireless sensor networks, RFID devices, and IoT; the evolution of IoT implies the interconnection of many embedded devices, usually with constrained resources and their interaction with the users [49, 48]. IoT encompasses all the above because of its similarities to the embedded computer [62].

It focuses on embedded devices with constrained resources in wireless sensor networks, RFID devices, and IoT. Those scenarios imply the interconnection of many embedded devices, and their interaction with the users [49, 48]. Also, the constrained resources devices encompass all the above scenarios due to their similarities to a traditional computer [62].

As a general description, the Oxford English Dictionary defines the IoT as development where daily objects have a network connection that allows sending and receiving data to help users improve their daily activities[2].

The target devices usually have restricted resources by their limited computing power, RAM, ROM, and others. In addition, these devices have a microcontroller as a central processing unit. Another kind of processor used is system-on-Chip which has integrated all modules that conform a complete computer into an integrated circuit. Another kind of processing unit used is the ARM microprocessors and Field

[1] http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf (reviewed 25/Oct/2021)

[2] https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project (reviewed 10/Nov/2021)

Programmable Gate Array (FPGA); the FPGAs consist of a logic block as a basic reconfigurable unit consisting of LUT's (lookup tables), Flip-Flops and multiplexers [134, 57, 2, 95].

The main characteristics of constrained devices are their limited resources. In some scenarios, they use batteries as a power source; they can acquire data from the environment by using sensors and send the data obtained to other devices using Wi-Fi, Ethernet, or Bluetooth communication;

Besides, they can perform actions in the environment, like turning on something or activating any other device to perform actions in the environment. Shortly, there is a list of some microcontrollers and microprocessors used in constrained devices.

- Microcontrollers of 8, 16, and 32 bits like AVR (Microchip), for example, ATMega328P (Used by Arduino UNO platform) or the model ATM2560.
- The SoC Snapdragon 32-bit ARM of 28 nm is from the Nvidia Tegra family with ARM and CUDA technologies like Tegra 3, TK1, and TX1.
- The 32-bit ARM processors are the most used in platforms of IoT; for example, minicomputers and Raspberry Pi platforms.

The constrained devices classification shown in table 3 presents three classes with general characteristics.

	Speed	Ram	Rom	Power
Class I	4 Mhz	1 KB	4 - 16 KB	1.5mA
Class II	4 - 8 Mhz	4 - 10 KB	48 - 128 KB	2 - 8 mA
Class III	13 - 180 Mhz	256 - 512 KB	4 - 32 MB	40 mA

Table 1.1: Classification and characteristics

Table 1.1 shows the three classes in the state-of-the-art, but it must be taken into account that there is a fourth classification to the constrained devices which will not

have kind of mention in this work. The devices in class I have the most constrained resources; all devices barely run a minimal operating system.

Devices with many more resources in the III class can run more sophisticated operating systems and even Java virtual machines. Finally, the devices with enough resources to perform "complex" tasks belonging to class II are the most commonly used for sensor nodes and IoT[82].

The IoT uses microcontrollers and microprocessors in the second class; they are an essential technology for the future for their capacity to get data from the environment without needing an electric network, for example, automobiles, smart homes, industry, to monitoring e-Healthcare, and farming.

When a developer designs an IoT device, the are restrictions like reducing power consumption when performing any task for longer battery life. In addition, the devices must be low-cost, and characteristics directly impact the security offered on these devices depending on the application[1].

1.4.1 Security on constrained devices

As previously mentioned, IoT has become a vital part of human life; besides, it is not well-defined or entirely secure. Today, all the proposed architectures must face many problems and challenges in security, and the technologies used in IoT do not have an acceptable security level.

Security in practice for constrained devices is essential because many attacks like Distributed Deny of Service (DDoS) exist. Those direct attacks on these devices cause serious security problems such as data alteration, privacy risks, and data modification.

The security challenges on constrained devices are due to their less complex nature, and traditional security aims at gadgets with more resources. The current cryptography implemented ranges from the implementation of block cipher algorithms to the development of alternative solutions like authentication servers, key-servers[2], Some Lightweight algorithms like PRESENT, PHOTON, and SPONGENT included in standards ISO (ISO/IEC 29192-5:2016) [9]. The following list shows some detected

[1] https://www.nist.gov/itl/applied-cybersecurity/iot-cybersecurity-considerations
[2] https://www.owasp.org/index.php/Guide_to_Cryptography

security problems on these systems [62]..

- Authentication: Used to identify users and devices over a heterogeneous network that uses traditional cryptography like certificates or the Public-Key-Infrastructure PKI on constrained resource devices difficult.
- Confidentiality: ensures that the information remains secret against third parties; a block cipher ensures that the only one who owns the secret key can access the data.
- Fault tolerance: if a node fails or gets jeopardized, the system must provide the security services available to the other nodes or users, i.e., the whole system remains running.
- Anonymity: users can hide their identity and the sensitive information they want, i.e., any user remains anonymous while interacting with the system.

Previous items consider some vulnerabilities in Open Web Application Security Project (OWASP). In addition, there is a section oriented to IoT [1] even though there is a guide that emphasizes some security aspects to take into account for the developers, manufacturing companies of IoT devices, and the public consumer [2].

Some lightweight cryptography primitives such as block ciphers, hash functions, stream ciphers, and message authentication codes have been proposed during the past decade. They offer some advantages in performance against traditional cryptography standards; also, there are lightweight block cipher proposals with some benefits over the National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) standard [46].

Just one algorithm cannot satisfy all security requirements, and weak points have priority to ensure the security of the constrained device. Below is a list of some options that offer one security aspect to implement as the application requires.

- Elliptic curves cryptography (ECC): Digital signatures to ensure the data comes from a trusted party and the possibility to share a secret between two parties.

[1] https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Vulnerabilities
[2] https://www.owasp.org/index.php/IoT_Security_Guidance

- Two-step authentication: offers user identification and authentication when logging into the system.
- Maintenance log: provides security audits, non-repudiation, and intruder detection.

Table 1.2: Layers, challenges, and requirements in security

Layer	Security challenges	Security requirements
perception layer	Resource restrictions, DoS vulnerable and interference	Lightweight encryption, sensor data protection, Key agreement
Network layer	Network congestion, virus spread DDoS attacks	Authentication and identity encryption anti DDoS Secure communication
Support layer	Large data processing, Data filtering	secure communication secure computing
Application layer	Data privacy and leakage Access control Application challenges	Key agreement and authentication Privacy protection Security administration

Table1.2 shows a compilation of challenges for each layer and every security requirement for each one [1, 65, 121, 137].

The perception layer consists of hardware devices; the main problem in this layer is the constrained resources, low computational power, and storage. It also must have low power consumption while running any algorithm; this requires a high level of encryption techniques or similar solutions with unfavorable performance impact in practice.

Network layer: this layer faces congestion caused by lots of data traffic, and the IoT cannot implement some secure communication mechanisms; on the other hand, it already has some others implemented like TLS/SSL, IP sec, anti-DDoS, and others [127].

Support layer: cloud computing and other use security applications at the system architecture or application levels. Therefore, this layer uses the best encryption

algorithms, protocols, antivirus, and string security techniques.

Application layer: deals with data privacy, access control mechanisms, and data leakage because data sharing is the most crucial characteristic in IoT environments where the data is stored or shared between two or more parties.

1.4.2 Security on high-speed devices

As a difference against the constrained resource devices, traditional cryptography focuses on high-speed implementations taking advantage of the resources available on regular processors like Intel and AMD processors, using technologies like intrinsic instruction sets and large-size registers embedded in the processors. Therefore public-key cryptography requires basic mathematical operations to generate and create a shared secret between two parties.

The mathematical operations required consist of additions and multiplications, and these last operations require a large number of clock cycles to complete just one; this is the reason why constrained devices are not suitable to handle public-key cryptography, but high-speed devices can offer new ways to implement basic operations to perform faster solutions on this area.

Many works focus on implementing just the basic operations to improve the performance of public-key algorithms on hardware; some applications are RSA and ECDH algorithms, and others are out of the scope of this thesis work. The paper [40] presents the modular exponentiation using Intel's AVX512 intrinsics set available on some of the 10th series families and newer. Work [7] computes the reduction modulo $\mathcal{F}$ using a multiplier based on the Karatsuba algorithm. One application of RSA is presented in work [47], which implements an RSA Key generation protocol focusing on a two-party setting. Also, they propose an approach for multiplying two large integers. In the same scope, the Montgomery for reduction algorithm presented later in this thesis focuses on ARM devices. This work given in [22] uses the mentioned algorithm to implement isogeny-based cryptography on ARM architectures. Finally, the work [31] presents a scheme of Homomorphic encryption proposal using real numbers and uses the Residue Numeric System RNS shown later in this thesis work to improve the performance of their implementation.

1.5 Summary

This chapter summarizes the challenges and activities carried out in this thesis; it also offers the objectives and a brief state-of-the-art along with the problem statement, which is the primary motivation for this thesis work. Furthermore, restricted device types are classified based on their features, resources, and applications. Also, it presents requirements of traditional cryptographic algorithms and some implementations and applications on high-speed devices like regular processors and FPGAs.

Part I

Theoretical Background

Chapter 2

Preliminaries

In this first part of this thesis works, the theoretical background needed to develop the works presented in the following chapters presents a brief introduction to definitions of the theories used to develop this thesis work focused on public-key cryptography and symmetric key cryptography as well as the mathematical background needed to understand the designs developed in the chapters 7 and 8.

2.1 Mathematical Background

This chapter shows the essential mathematical background to understand and develop cryptography systems. Then, we use this knowledge to create general cryptography systems in hardware and software.

Usually, using the integer ring $\mathbb{Z}$ is the mathematical basis of cryptography. Once we can compute with integers is possible to build on top of them fields, elliptic curves, and even more complex objects. To perform faster integer arithmetic is why the hardware plays an important role when implementing arithmetic on computing systems.

2.2 Algebra

Here we present some abstract algebra fundamentals to understand the Public-key algorithms presented later in this thesis work.

2.2.1 Groups

As presented in [34] chapter 2, section 2.1, a group gets defined by the following.

Given a set S, a *composition law* $\times$ of S *into itself* is a mapping from the Cartesian product $S \times S \to S$, Common notations for the image of (x, y) under this mapping are $x \times y$, $x \cdot y$ or simply xy. When the law is *commutative*, i.e., when the images if (x, y) and (y, x) under the composition law are the same $\forall x, y \in S$, it is customary to denote it by "+".

Another definition says: A group G is a set with a composition law $\times$ such that,

- $\times$ is *associative*, that is $\forall x, y, z \in G$ we have $(xy)z = x(yz)$
- $\times$ has a unit element e, i.e., $\forall x \in G$ we have $xe = ex = x$
- for every $x \in G$ $\exists$ y, an *inverse* of x such that $xy = yx = e$.

Remarks

1. The group G is *commutative* or *abelian*, if the composition law is commutative. As previously mentioned, the law often gets denoted by $+$ or $\oplus$ and the unit element by 0 for this case.
2. The unit of a group G is necessarily unique and is the inverse of an element x that is denoted with x^{-1}. if G is commutative the inverse of x gets denoted by $-x$.
3. The *order* or *cardinality* of a group G is *finite* it its order is finite.

Example: Let $x \in G$. The set $\{x^n \mid n \in \mathbb{Z}\}$ is the subgroup of G *generated* by x. It is denoted by $\langle x \rangle$

2.2.2 Rings

Also, in the same book [34] section 2.1.2, chapter 2 present various definitions for rings; here, we present some of them and an example.

A ring R is a set together with two composition laws $+$ and $\times$ such that

- R is commutative respect to $+$

- $\times$ is associative and has a unit element 1, which is different from 0 of the unit $+$
- $\times$ is disruptive over $+$, that is $\forall x, y, z \in R, x(y+z) = xy + xz$ and $(y+z)x = yx + zx$.

Remarks

1. The ring R is said to be *commutative*, if the law x is commutative.
2. A commutative ring R such that $\forall x, y \in R$, the equality $xy = 0$ implies that $x = 0$ or $y = 0$ is called *integral domain.*

Example: the set $\mathbb{Z}$ of integers together with the usual addition and multiplication is a ring. The set $\mathbb{Z}[X]$ of polynomials with coefficients in $\mathbb{Z}$ together with the addition and multiplication of polynomials is a ring.

2.2.3 Fields

Once more, in the same book [34] in the same chapter 2 section 2.1.3, we found different definitions of Fields, but we only present some of them here.

A field K is a commutative ring such that every nonzero element is invertible.

Example: the set of rational numbers $\mathbb{Q}$ with the usual addition and multiplication law is a field. The quotient set $\mathbb{Z}/p\mathbb{Z}$ with the induced integer addition and multiplication is also a field for any prime number p.

2.2.4 Extension of elds

Usually, in cryptography, some operation becomes easier to implement using an extension of fields, allowing the creation of more complex arithmetic. Some properties of algebraic extensions of fields are presented from the book [34] chapter 2, section 2.2.1.

De nition: Let K and L be two fields, we can say that L is an *extension field* of K if there exists a field homomorphism K into L. Such an extension field is denoted by L/K.

Remark: as previously mentioned, a field homomorphism is always injective, so we

shall identify K with the corresponding subfield of L when considering L/K.

Example: let $\mathbb{R}$ the field of real numbers with usual addition $+$ and multiplication $\times$. Therefore, $\mathbb{R}$ is an extension of $\mathbb{Q}$. Consider the element $\sqrt{2} \in \mathbb{R}$ and the subset of $\mathbb{R}$ of elements with the form $a + \sqrt{2}b$ with $a, b \in \mathbb{Q}$. If we put for $a + \sqrt{2}b$ and $a' + \sqrt{2}b'$

$$(a + \sqrt{2}b) + (a' + \sqrt{2}b') = a + a' + \sqrt{2}(b + b') \tag{2.1}$$

and

$$(a + \sqrt{2}b) \times (a' + \sqrt{2}b') = aa' + 2bb' + \sqrt{2}(ab' + a'b), \tag{2.2}$$

we can see that we obtain a field denoted by $\mathbb{Q}(\sqrt{2})$, which is an extension of $\mathbb{Q}$.

2.3 Integer arithmetic operations

As mentioned at the beginning of this chapter, Integer arithmetic is fundamental and must perform as efficiently as possible. Although modern computers operate with relatively large integers thanks to technologies like advanced vector extension (AVX) [81, 35] instruction set, we can design reliable components in hardware to perform the same operation presented here with better performance than software implementations.

Modern computers have highly optimized operations for single-precision integers. Therefore, they can perform the following functions from [34] chapter 10, section 10.1.3.

- *Comparison* of two singles return a boolean 0 or 1.
- *Bitwise complement* of a single u, that is $\overline{u} = b - 1 - u$.
- *Bitwise conjunction, disjunction* and *exclusive disjunction* of the singles u and v, that is respectively $u \wedge v, u \vee v$ and $u \oplus v$ used to denote *XOR*.
- The *right* and *left* shifts of t bits for a single u, denoted by $u \ggg t$ and $u \lll t$ respectively, corresponding to $\lfloor u/2^t \rfloor$ and $u2^t \bmod b$.

  Addition of two singles u and v results in a single w and a carry bit c equal to 0 or 1 so that the result is $u + v = kb + w$.

  Subtraction of a single v from a single u, that is $u - v$, results in a single w and a carry c. If $u \geq v$ then $w = u - v$ and $c = 0$, otherwise $w = b + u - v > 0$ and $c = -1$, sometimes called *borrow* for its unsigned representation.

- *Multiplication* of two singles u, v results in a double size result $w = u \times v$.
- *Division* of a double size u by a single v, when the quotient $q = \lfloor u/v \rfloor$ and the remainder $r = u \bmod v$ are both singles. This computes q and r simultaneously.

We must understand the basic facts of integers and computers to use large integers. For example, let $b \geq 2$ an integer is known as *base* and sometimes *radix*, and it is possible to represent any integer $u > 0$ as the sum.

$$u = u_{n-1}b^{n-1} + \cdots + u_1 b + u_0 \tag{2.3}$$

provided $0 \leq u_i < b$ and $u_{n-1} \neq 0$. We call this representation the *base* b of u and is denoted by $(u_{(n-1)}, \ldots, u_0)_b$. witch each u_i as the digits of u. Finally the most significant digit is denoted by u_{n-1} and least significant by u, therefore, the length n of $(u_{(n-1)}, \ldots, u_0)_b$ is denoted $|u|_b$.

In any digital system like a computer, the base b is a power of 2 stored as an internal sequence of 0 and 1, known as bits. So, also exists elementary operations on bits like the following:

- *complement* of x denoted by $\overline{x}$, if $x = 0$ then $\overline{x} = 1$
- *conjunction* of x and y, $x \wedge y$ the output is equal to 1 is both x, y equal 1.
- *disjunction* of x and y, $x \vee y$ the output equals 1 if any of both are equal to 1.
- *exclusive disjunction* of x and y, known as *XOR*. The operation x xor y equals 1 if and only if exactly one of the values x, y equals 1.

2.3.1 Addition and Subtraction

In the book [34] chapter 10, section 10.2 presents the addition algorithm used to add multi-precision integers; the same is presented here in the algorithm 1.

Algorithm 1 Addition of non-negative multi-precision integers

Require: Two n-word integers $u = (u_{n-1}, \ldots, u_0)_b$ and $v = (v_{n-1}, \ldots, v_0)_b$.
Ensure: The $(n+1)$-word integer $w = (w_n, \ldots w_0)_b$ such that $w = u + v$, w_n being 0 or 1.

1: $k \leftarrow 0$ [k is the carry]
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad w_i \leftarrow (u_i + v_i + k) \bmod b$ $[0 \leqslant w_i < b]$
4: $\quad k \leftarrow \lfloor (u_i + v_i + k)/b \rfloor$ [$k = 0$ or 1]
5: **end for**
6: $w_n \leftarrow k$
7: **return** $(w_n, \ldots, w_0)_b$

For the subtraction, we present the algorithm 2, just a simple change of sign in the algorithm 1 to provide $(u - v)$ instead $(u + v)$ equipped $u \geq v$.

Algorithm 2 Subtraction of non-negative multi-precision integers

Require: Two n-word integers $u = (u_{n-1}, \ldots, u_0)_b$ and $v = (v_{n-1}, \ldots, v_0)_b$ such that $u \geq v$.
Ensure: The n-word integer $w = (w_{n-1}, \ldots w_0)_b$ such that $w = u - v$.

1: $k \leftarrow 0$ [k is the carry]
2: **for** $i = 0$ **to** $n - 1$ **do**
3: $\quad w_i \leftarrow (u_i - v_i + k) \bmod b$ $[0 \leqslant w_i < b]$
4: $\quad k \leftarrow \lfloor (u_i - v_i + k)/b \rfloor$ [$k = 0$ or -1]
5: **end for**
6: $w_n \leftarrow k$
7: **return** $(w_{n-1}, \ldots, w_0)_b$ [if $k = -1$ then $u < v$]

2.3.2 Multiplication

The most critical process in any crypto-system is multiplication, the most time-consuming part for many applications. Therefore the complexity parameter of a multiplication algorithm is essential for any complete arithmetic system. In this section, we present two multiplication algorithms Schoolbook method and Karatsuba.

School book

The simplest method known for at least four millennia [34].

Algorithm 3 Multiplication of positive multi-precision integers

Require: An m-word integer $u = (u_{m-1}, \ldots, u_0)_b$ and an n-word integer $v = (v_{n-1}, \ldots, v_0)_b$.
Ensure: The $(m+n)$-word integer $w = (w_{m+n-1}, \ldots, w_0)_b$ such that $w = uv$.

1: **for** $i = 0$ **to** $n-1$ **do**
2: $\quad w_i \leftarrow 0$ [see remark 1]
3: **end for**
4: **for** $i = 0$ **to** $n-1$ **do**
5: $\quad k \leftarrow 0$
6: $\quad$ **if** $v_i = 0$ **then**
7: $\quad\quad w_{m+i} \leftarrow 0$ [optional test]
8: $\quad$ **else**
9: $\quad\quad$ **for** $j = 0$ **to** $m-1$ **do**
10: $\quad\quad\quad t \leftarrow v_i u_j + w_{i+j} + k$ $[0 \leq t < b^2]$
11: $\quad\quad\quad w_{i+j} \leftarrow t \bmod b$ $[0 \leq w_{i+j} < b]$
12: $\quad\quad\quad k \leftarrow \lfloor t/b \rfloor$ $[0 \leq k < b]$
13: $\quad\quad$ **end for**
14: $\quad\quad w_{m+i} \leftarrow k$
15: $\quad$ **end if**
16: **end for**
17: **return** $(w_{m+n-1}, \ldots, w_0)_b$

Remarks:

1. The algorithm performs the following operations *multiply* and *add*

$$(w_{n+m-1}, \ldots, w_0)_b \leftarrow (u_{n-1}, \ldots, u_0)_b \times (v_{m-1}, \ldots, v_0) + (w_{m-1}, \ldots, w_0)_b \quad (2.4)$$

2. This method computes intermediate results uv_i before adding them. In the algorithm 3, we multiply and add simultaneously inside the j loop.

3. The optional test in line 7 is useless if the base b is small.

Karatsuba

In 1960, Karatsuba proposed a method [71] in $\mathcal{O}(n^{\log_2(3)})$, with *log* as the logarithm base 2. The work [73] shows the technique of the so-called Karatsuba method. For clarity, set $R = b^n$, $d = 2n$, and $u = (u_{d-1}, \ldots, u_0)$ and $v = (v_{d-1}, \ldots, v_0)$ as two d-word integers. The procedure involves splitting u and v into the least and most significant parts.

$$uv = U_1V_1R^2 + ((U_0 + U_1)(V_0 + V_1) - U_1V_1 - U_0V_0)R + U_0V_0. \tag{2.5}$$

A *priori* four multiplications are needed to compute uv, but multiplication by R is a shift. So there are more additions, and only three multiplications are required. Which are U_1V_1, $(U_1 + U_0)(V_1 + V_0)$, U_0V_0. Therefore we can use a recursive approach to reduce the size of the operands until they are small enough to use the school book method. The algorithm 4 from [34] chapter 10 section 10.3.2. shows the procedure.

Algorithm 4 Karatsuba multiplication of positive multi-precision integers

Require: An n-word integer $u = (u_{n-1}, \ldots, u_0)_b$ and an m-word integer $v = (v_{m-1}, \ldots, v_0)_b$ the size $d = max\{m, n\}$, and a threshold d_0.
Ensure: The $(m + n)$-word integer $w = (w_{m+n-1}, \ldots, w_0)_b$ such that $w = uv$.
1: **if** $d \leq d_0$ **then**
2: **return** uv [use algorithm 3]
3: **end if**
4: $p \leftarrow \lfloor d/2 \rfloor$ and $q \leftarrow \lceil d/2 \rceil$
5: $U_0 \leftarrow (u_{q-1}, \ldots, u_0)_b$ and $V_0 \leftarrow (v_{q-1}, \ldots, v_0)_b$
6: $U_1 \leftarrow (u_{p+q-1}, \ldots, u_q)_b$ and $V_1 \leftarrow (v_{p+q-1}, \ldots, v_q)_b$ [pad with $0's$ if necessary]
7: $U_s \leftarrow U_0 + U_1$ and $V_s \leftarrow V_0 + V_1$
8: compute recursively U_0V_0, U_1V_1 and U_sV_s
9: **return** $U_1V_1b^{2q} + (U_sV_s - U_1V_1 - U_0V_0)b^q + U_0V_0$

2.3.3 Modular reduction

In many situations, it is necessary to compute the remainder of a euclidean division. However, in practice, we use a prime number N, and the essential operation for the prime field arithmetic is the reduction. The naive way to compute u mod N consists in dividing u by N to obtain the remainder. In this chapter, we present the Montgomery reduction algorithm, and Chapter 7 offers two methods focused on hardware implementations.

Montgomery reduction

Peter L. Montgomery introduced a way to represent elements of $\mathbb{Z}/N\mathbb{Z}$ such that arithmetic and multiplication become easy[89]. We have the following from [34] chapter 10, section 10.4.2.

Definition: let R an integer greater than N and a co-prime with it. The *Montgomery representation* of $x \in [0, N-1]$ is $[x] = (xR) \bmod N$. The *Montgomery reduction* of $u \in [0, RN-1]$ is $Redc(u) = (uR^{-1} \bmod N)$.

When R is a power of the radix b, there is an algorithm to reduce u. Let $N' = (-N^{-1}) \bmod R$ and let k the unique integer in $[0, N-1]$ such that $k \equiv uN' \pmod R$. It becomes clear that $(u+kN)$ is a multiple of R. Let $t = (u+kN)/R$. Remembering that N, R are relative prime, implies that $t \equiv uR^{-1} \pmod N$. Finally, $0 \leq u < RN$ shows that $0 \leq t < 2N$ so that t or $t-N$ is equal to the desired result $Redc(u)$. The algorithm 5 shows the entire process for multi-precision integers.

Algorithm 5 Montgomery reduction *Redc* of multi-precision integers

Require: An n-word integer $N = (N_{n-1}, \ldots, N_0)_b$ such that $gcd(N, b) = 1$, $R = b^n$, $N' = (-N^{-1}) \bmod N$ and a $2n$-word integer $u = (u_{2n-1}, \ldots, u_0)_b < RN$.

Ensure: The n-word integer $t = (t_{n-1}, \ldots, t_0)_b$ such that $t = Redc(u) = (uR^{-1}) \bmod N$.

1: $(t_{2n-1}, \ldots, t_0)_b \leftarrow (u_{2n-1}, \ldots, u_0)_b$
2: **for** $i = 0$ **to** $n-1$ **do**
3: $\quad k_i \leftarrow (t_i N') \bmod b$
4: $\quad t \leftarrow t + k_i N b^i$
5: **end for**
6: $t \leftarrow t/R$
7: **if** $t \geqslant N$ **then**
8: $\quad t \leftarrow t - N$
9: **end if**
10: **return** t

Remarks

1. It is immediate that $[x] = Redc((xR^2) \bmod N)$ and $Redc([x]) = x \ \forall x \in [0, N-1]$. Also, the value $R^2 \bmod N$ can be precomputed.

2. The algorithm 5 requires $n^2 + n$ multiplications to compute Montgomery reduction.

3. Classical reduction computes the reduction while processing each digit of u from left to right. Montgomery reduction can operate in both senses *left-to-right* and *right-to-left*.

4. In the case $u \geqslant RN$ the algorithm 5 does not return $t = uR^{-1} \bmod N$ but $t \equiv uR^{-1} \pmod{N}$.

2.4 Greatest Common Divisor (GCD)

As presented in [34] chapter 10 section 10.6.1, given two integers x and N, the algorithm computes $d = gcd(x, N)$ and the integers u and v such that $xu + Nv = d$. In practice, this is the method to compute the inverse of an element in $(\mathbb{Z}/N\mathbb{Z})*$. Therefore it is linked to the Chinese remainder theorem presented in the section 2.5. The algorithm 6 offers the extended Euclid *gcd* procedure.

Remarks

Algorithm 6 Euclid extended *gcd* of positive integers

Require: Two positive integers x, N such that $x < N$.
Ensure: Integers (u, v, d) such that $xu + Nv = d$, with $d = gcd(x, N)$.

1: $A \leftarrow N, B \leftarrow x, U_A \leftarrow 0, U_B \leftarrow 1$
2: **while** $B \neq 0$ **do**
3: $\quad q \leftarrow \lfloor A/B \rfloor$
4: $\quad \begin{bmatrix} A \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$
5: $\quad \begin{bmatrix} U_A \\ U_B \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \begin{bmatrix} U_A \\ U_B \end{bmatrix}$
6: **end while**
7: $d \leftarrow A, u \leftarrow U_A$, and $v \leftarrow (d - xu)/N$
8: **return** (u, v, d)

- The use of the variables V_A, V_B from the equation 2.4 such that

$$V_A = 1, V_B = 0 \text{ and } \begin{bmatrix} V_A \\ V_B \end{bmatrix} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix} \begin{bmatrix} V_A \\ V_B \end{bmatrix} \tag{2.6}$$

  we see that $xU_A + NV_A$ and $xU_B + NV_B$ always are equal respectively A and B during the execution. Adding and updating these two variables during the execution avoids the division in line 7.

- Throughout the execution of the algorithm the $|U_A|, |U_B$ in relation to $(|V_A|, |V_B|) \leq N/A$ in relation to (x/A).
- The necessary steps is $\approx \mathcal{O}(lgN)$. But there exist better performance implementations in the state of the art if the algorithm has a careful implementation.

2.5 Chinese Residue Theorem CRT

To understand the Chinese residue theorem, we need the following theorem and corollary from book [34] chapter 2 theorem 2.23 and corollary 2.24.

Theorem: Let $I_1, \ldots, I_k$ a pairwise co-prime ideals of R. Then

$$R/\prod_{i=1}^{k} I_i \simeq \prod_{i=1}^{k} R/I_i. \tag{2.7}$$

Corollary: Let $n_1, \ldots, n_k$ a pairwise co-prime integers, such that $gcd(n_i, n_j) = 1$ for $i \neq j$. Also, for any integers x_i $\exists$ an integer x such that

$$\begin{cases} x \equiv x_1 \ (mod\ n_1) \\ x \equiv x_2 \ (mod\ n_2) \\ \qquad\qquad\vdots \\ x \equiv x_k \ (mod\ n_k). \end{cases} \tag{2.8}$$

Therefore, x is unique modulo $\prod_{i=1}^{k} n_i$.

Now, suppose one wants to know the solution of the system 2.8 where the n_i' are pairwise co-prime integers and the x_i's fixed integers. The corollary above ensures that there is a unique solution modulo $N = n_1 n_2 \ldots n_k$. Such a solution is easy to find. Let $N_i = N n_i$. Since the n_i's are pairwise co-prime we have $gcd(N_i, n_i) = 1$ $\forall i$. And the computation of the extended gcd gives a_i such that $a_i \equiv 1 \pmod{n_i}$. Furthermore, the solution is given by the following equation.

$$x = a_1 N_1 x_1 + a_2 N_2 x_2 + \cdots + a_k N_k x_k. \tag{2.9}$$

The following algorithm 7 from [34] chapter 10 section 10.6.4 performs efficiently and computes x inductively. at each step, given an integer x such that $x \equiv x_i (\text{mod } n_i) \forall \text{i} \leq j$, it find x satisfying $x \equiv x_i (\text{mod } n_i)$ $\forall i \leq j+1$.

Remark

Algorithm 7 Chinese remainder

Require: Pairwise co-prime integers $n_1, \ldots, n_k$ and integers x_i for $1 \leq i \leq k$.
Ensure: An integer x such that $x \equiv x_i (\text{mod } n_i)\ \forall\ 1 \leq i \leq k$.

1: $N \leftarrow n_1$ and $x \leftarrow x_1$
2: **for** $i = 2$ **to** k **do**
3: compute u, v such that $un_i + vN = 1$ [use extended *gcd* algorithm]
4: $x \leftarrow un_i x + vNx_i$
5: $N \leftarrow Nn_i$
6: $x \leftarrow x \bmod N$
7: **end for**
8: **return** x

1. It is possible to generalize the algorithm 7 in a straightforward way to the polynomial ring $K[X]$ where K is a field.

2. The *residue number system* presented in chapter 7 is specially useful where the computations modulo N are performed modulo several primes p_i fitting a word such that $\prod_i pi > N^2$.

2.6 Summary

In this chapter, we presented a brief introduction to some basic concepts needed to understand the research and development of this thesis work; therefore, we explain the basic operations required for any modern crypto-system and some definitions with remarks available.

The mathematical background comprises the basic definitions used in algebra, like groups, rings, fields, etc. Therefore, the numbers used in practice are always integers with their respective arithmetic operations and algorithms for each operation needed.

We encourage our readers to study the basic mathematical background more profoundly to understand the complex algorithms presented in the later chapters of this thesis.

Chapter 3

Cryptography

Cryptography is closely linked to electronic communications but is an old practice with examples from older ages using hieroglyphics as a non-standard language in ancient Egypt. Also, there are cases of many cultures of secret writing in ancient Greece with (scytale) of Sparta, or the Caesar cipher, the cryptology as the main tern splint in two branches:

- Cryptography is the science of hiding information.
- Cryptanalysis is the science and art focused on breaking cryptosystems.

The main focus of this chapter is cryptography and implementations based on symmetric algorithms in which two parties previously shared a secret and methods for encryption and decryption to communicate secretly with each other. Usually, Bob wants to communicate with Alice, sending a message using a key to (encrypt) the message before it is shipped, and Alice uses the same Key to (decrypt) it and recovers the message. This message is known as (*Plaintext*), and the encrypted message is known as (*Ciphertext*). The shared Key distinguishes communicating parties from any other who may be eavesdropping on their communication using a public channel like the Internet. Figure 3.1 illustrates the steps previously mentioned.

The variables shown in the figure are:

- m is the plaintext.
- c is the ciphertext.
- k is the Key.

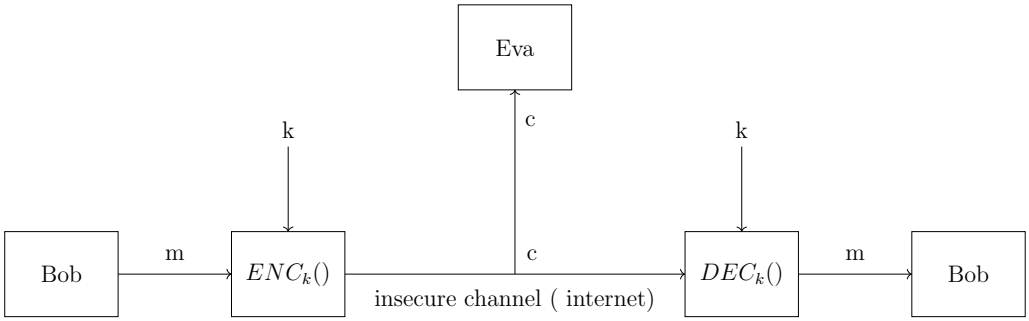

Figure 3.1: Communication between parties using symmetric cryptography over an insecure channel

In the rest of this chapter, we focus on using the symmetric cipher AES and a little history of it and its characteristics. Then, present some modes of operation, and finally, show the use of this cipher in other ways.

3.1 Advanced Encryption Standard (AES)

In 1997 the National Institute of Standards and Technology (*NIST*) launched a call to develop a new cipher to replace Data Encryption Standard (*DES*) [104] with a new standard named Advanced Encryption Standard (*AES*). The algorithm selection for AES was open and administrated by NIST after three evaluation rounds, and the scientific community discussed the advantages and disadvantages of the submitted algorithms. Finally, in 2001 NIST declared the block cipher (Rijndael [37]) as the new AES and published it in FIPS NUM 197 [42]; Vincent Rijmen and Joan Daemen, both Belgian cryptographers designed the algorithm.

The call for proposal had the following mandatory requirements for all AES candidates:

- Block cipher with 128-bit block size.
- The key lengths support 128, 192, and 256 bits.
- Efficiency in software and hardware.

AES is a block cipher used to protect electronic data and can encrypt and decrypt data, i.e., it converts the message or plaintext to an encrypted result called ciphertext, and the decryption recovers the original message from ciphertext to plaintext. As previously mentioned, AES quires a 128-bit input block and a secret key of 128, 192,

and 256-bit, producing a ciphertext of 128-bit length. With the first Key, the algorithm can derivate the following Round Key used in the next round of the algorithm.

Figure 3.2 shows the AES algorithm as a black box, as can be seen; the inputs are the plaintext m and the Key k. The Key can have three sizes; the output c is the ciphertext. The right side of the figure also shows the decryption process in the same way. In this case, the inputs are the ciphertext c and the Key k, and the output is the plaintext m.

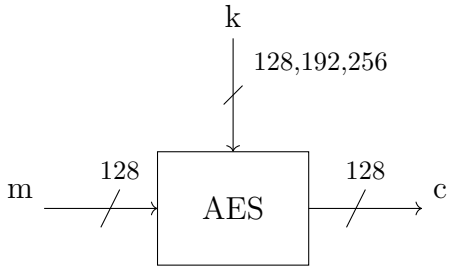

Figure 3.2: General use of AES in encryption mode.

Algorithm 8 AES algorithm with Nb as the number of bytes processed

Require: byte $in[4 \times Nb]$, byte $out[4 \times Nb]$, word $w[Nb \times (Nr + 1)]$, rounds Nr
Ensure: byte $out[4 \times Nb]$

byte-$(state)[4, Nb]$
$(state) \leftarrow in$
ARK $(state), w[0, Nb - 1]$
for each $r = 0$ to $r = Nr - 1$ **do**
    SB$-(state)$
    SR$-(state)$
    MC$-(state)$
    ARK $(state), w[round \times Nb, (round + 1) \times Nb - 1]$
end for
SB$-(state)$
SR$-(state)$
ARK$-(state), w[Nr \times Nb, (Nr + 1) \times Nb - 1]$
$out \leftarrow (state)$

$$b = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline b_0 & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ \hline \end{array}$$

Figure 3.3: AES input array as an initial state.

$$b = \begin{array}{|c|c|c|c|} \hline b_0 & b_4 & b_8 & b_{12} \\ \hline b_1 & b_5 & b_9 & b_{13} \\ \hline b_2 & b_6 & b_{10} & b_{14} \\ \hline b_3 & b_7 & b_{11} & b_{15} \\ \hline \end{array}$$

Figure 3.4: Matrix representation for a state in AES.

The algorithm 8 shows the AES algorithm here, and we can see four steps or transformations inside the loop. This loop can variate depending on the Key size, and the functions are:

- AddRoundKey
- ByteSubstitution
- ShiftRows
- MixColumns

The AES algorithm uses different variables, and the following presents the description of each of the four functions.

Figure 3.3 shows the state as a matrix of $[4 \times Nb]$ bytes as an arrangement of bytes as input. Furthermore, 3.4 shows a matrix of 4 times 4 bytes used in 128-bit encryption, and here we describe each function in a general way.

Figure 3.5 shows a diagram of the AES steps in a general way with each transformation needed by the algorithm; as can be seen, the first step is the Key addition layer in this state, and the input plaintext is XORed with the round Key 0. After that, the result becomes an input to the Byte substitution layer, later the state goes to the ShiftRows Layer and finally to the MixColumn Layer.

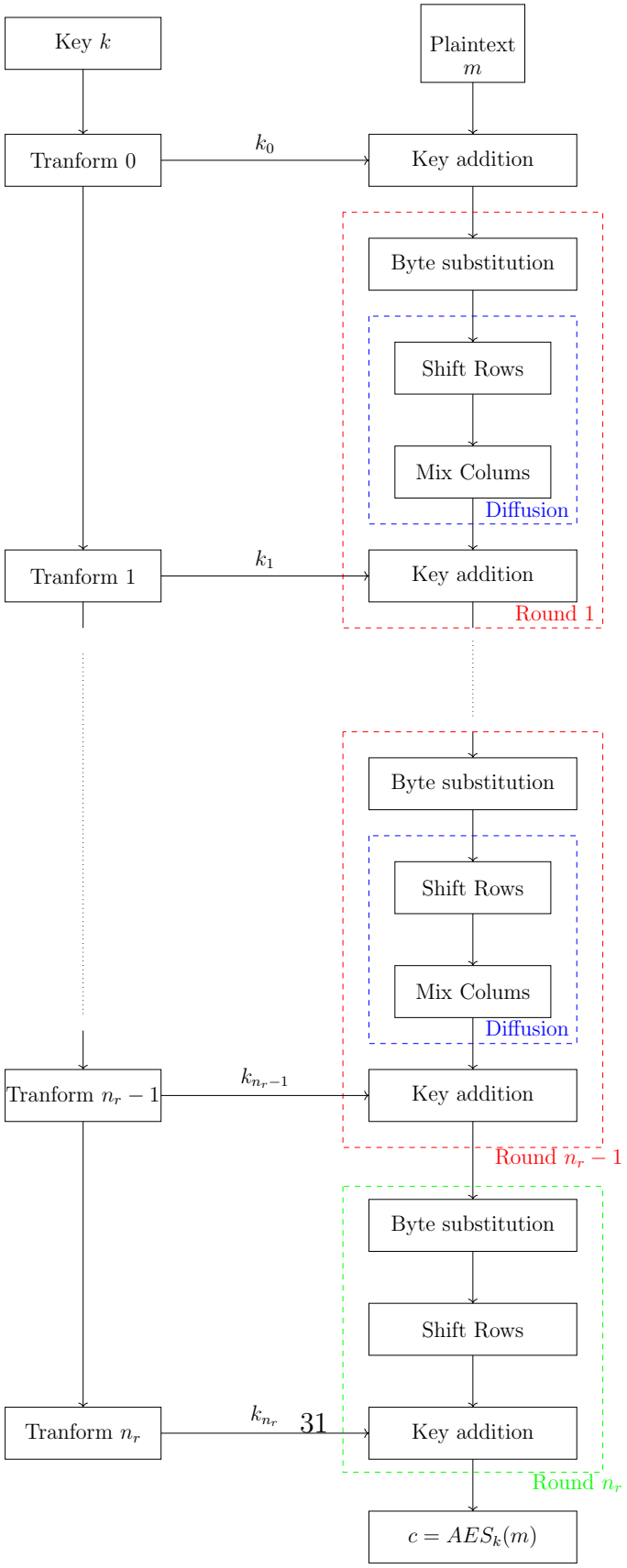

Figure 3.5: AES encryption diagram

These four steps repeat as needed for Nr - 1 time required by the algorithm and the Key size. For the last round, there are three functions to perform the current state. First is the Byte substitution, then the state goes to ShiftRows, and finally, Key addition (add round key). In the last round, the omission of the Mix Column Layer is a must, and in the Mix Column Layer, we explain why this step is omitted.

3.1.1 Byte Substitution layer

The first layer in each round is the Byte Substitution with 16 parallel S-boxes with a byte width; all S-boxes are equal and have de same values. In each state, Byte A_i gets replaced, i.e., replaced by another byte B_i:

$$S(A_i) = B_i \tag{3.1}$$

The only nonlinear element of AES is the S-Box it holds that

$$ByteSub(A) + ByteSub(B) \neq ByteSub(A + B) \tag{3.2}$$

For states A and B. The S-box is a bijective mapping of $2^8 = 256$ possible input elements to a one-to-one mapped output element. Also, the S-Box has a reverse feature needed in the decryption process. The S-box usually gets implemented as a 256-bit lookup table with fixed entries as given in table 3.1, and the software implementations use this kind of table.

Suppose an inputs byte to the S-Box as $A_i = (0xFF)$, the output value is

$$SBox(0xFF) = (0x16) \tag{3.3}$$

S-Box is bijective. It does not have fixed points and any input value A_i such that $S(A_i) = A_i$, for example, $S(0x00) = 0x63$.

Comparing the AES software against hardware implementations, the focus of software usually goes to using a lookup table. The hardware sometimes takes advantage of realizing the S-Boxes as digital circuits which compute each value on-the-fly.

3.1.2 ShiftRows

This step makes a cyclic transformation to the current state; it shifts each row to a fixed number of bytes. For example, it shifts three bytes to the second row, the third

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 3.1: Substitution Box represented with hex values.

row shifts two bytes, and the fourth row shifts only one byte. There are no changes in the first row, and this transformation's purpose is to increase the diffusion of AES. In figure 3.6 the input state matrix is given as $B = (B_0, B_1, ..., B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

Figure 3.6: Shift Rows (SR) input state.

The output is the new state shown in figure 3.7.

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	$\longrightarrow \gg 3$
B_{10}	B_{14}	B_2	B_6	$\longrightarrow \gg 2$
B_{15}	B_3	B_7	B_{11}	$\longrightarrow \gg 1$

Figure 3.7: Shift Rows (SR) output state.

3.1.3 MixColumns

In this step, a linear transformation that mixes each column of the current input state, every input byte influences the four output bytes, bringing high diffusion in AES. Furthermore, the ShiftRows and MixColumns combination ensures that after three rounds, each state byte depends on all 16 bytes of the plaintext. Therefore, we denote B as the 16-byte input state and the output state as C:

$$MixColumns(B) = C, \tag{3.4}$$

Where B is the output state from the ShiftRows step as in the figure 3.7, then each 4-byte individual column is a vector and gets multiplied by a fixed 4×4 matrix. This matrix has constant elements and performs the multiplication and addition of the coefficients in $GF(2^8)$. The following matrix shows how to perform MixColumns for the second 4 bytes.

$$\begin{bmatrix} B_4 \\ B_9 \\ B_{14} \\ B_3 \end{bmatrix} \times \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} C_4 \\ C_5 \\ C_6 \\ C_7 \end{bmatrix}$$

Then we compute each output 4-byte array C_i by multiplying the 4-byte input array column B_i, with the same constant matrix.

3.1.4 Add Round Key

This step uses the master Key and derives the subkeys needed by AES. This operation combines two inputs performing an XOR equal to an addition in the Galois Field $GF(2)$, and the Key Schedule derivates the round Keys.

3.1.5 Key Schedule

Use the Master Key (Key provided by the user) and derives all necessary Sub-Keys for the AES implementation. The Sub-Keys needed equals the number of rounds plus one due to the Key used by the first round, as shown in figure 3.5. For example, for a Key length of 128-bit, the rounds need $n_r = 10$ and the 11 subkeys each of 128-bit length.

3.1.6 AES 128-bit Key schedule

This implementation uses 11 subkeys stored on an array with elements $W[0], ..., W[43]$, and figure AESKeySchedule depicts the calculation for each Key.

The first step is to denote the original key as K_0; this key gets copied into the first elements of the array W; now we can compute the next as follows, as shown in the figure, the leftmost word of a subkey $W[4i]$ where $(i = 1, ..., 10)$ gets computed by:

$$W[4i] = W[4(i-1)] + g(W[4i-1]). \tag{3.5}$$

Denoting $g()$ as a nonlinear function with 4-byte input and outputs and the three left words of a subkey get computed as:

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j] \tag{3.6}$$

where $(i = 1, ..., 10)$ and $j = 1, 2, 3$. Furthermore, the $g()$ function rotates its four input bytes, performing a byte-wise substitution with the S-Box adding a (round coefficient) RC. This coefficient of 8-bit width belongs to the Galois field $GF(2^8)$ and gets added to the leftmost byte in function $g()$. Each RC varies according to the next rule:

$$\begin{aligned} RC[1] &= x^0 = (0x01), \\ RC[2] &= x^1 = (0x02), \\ RC[3] &= x^2 = (0x04), \\ &\vdots \\ RC[10] &= x^9 = (0x36). \end{aligned}$$

Function $g()$ adds non-linearity to the key schedule and removes symmetry in AES, and these properties are necessary to thwart some attacks on block ciphers.

3.2 Block Cipher Modes of Operation

The previous section explains using a block cipher with only one block of plaintext, but in practice, we want to encrypt large amounts of data, so we introduce some modes of operation. [19] in this section.

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Galois Counter Mode (GCM)

This section focuses on the enlisted operation modes and describes them next.

3.2.1 Electronic Codebook (ECB)

The most straightforward way to encrypt a fixed-sized plaintext can encrypt or decrypt only one block at once, let $e_k(x_i)$ the encryption process of a plaintext x_i with key k and the decryption process as $e^{-1}(y_i)$ with the ciphertext y_i with Key k. In this operation mode, the block cipher process each encryption individually. Figure 3.8 shows the encryption and decryption process with $e()$ as the block cipher, b as the block size, and x_i, yi plaintext and ciphertext, respectively.

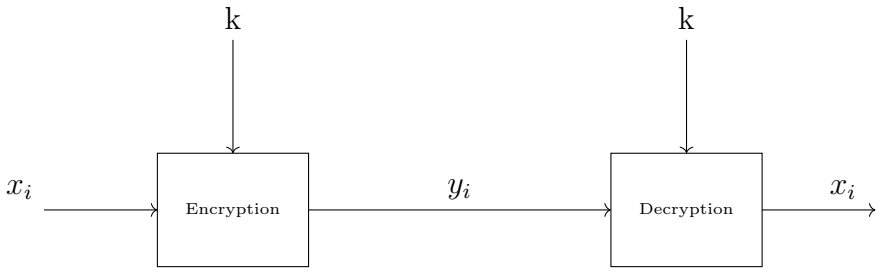

Figure 3.8: Electronic codebook operation mode (ECB).

$$Encryption : y_i = e_k(x_i), i > 1$$
$$Decryption : x_i = e_k^{-1}(y_i) = e_k^{-1}(e_k(x_i)), i > 1$$

To verify the ECB mode:

$$e_k^{-1}(y_i) = e_k^{-1}(e_k(x_i)) = x_i \tag{3.7}$$

Some advantages of ECB mode are the unnecessary synchronization between parties. For example, if the receiver does not get all encrypted blocks due to transmission problems, it is possible to decrypt received blocks, and the same happens with problems caused by any other sources. Also, parallelizing those block ciphers working on ECB mode is possible, e.g., while one encryption unit processes the first block, the second process the next block, and the same for all encryption units.

3.2.2 Cipher Block Chaining Mode (CBC)

There are two main ideas behind the CBC design. First, the encryption of all blocks makes them a chain such that ciphertext y_i depends not only on plaintext x_i, it depends on all previous plaintexts, and the use of an Initialization Vector (IV) adds randomness. As we know, y_i results from encrypting the plaintext x_i. This result is feedback to the cipher input and *XORed* with the next block x_{i+1}. The output from the XOR is then encrypted, yielding the y_{i+1} ciphertext, and then this result is used to encrypt the following plaintext x_{i+2}.

The first ciphertext, y_1, depends on plaintext x_1, and the IV, the second ciphertext, depends on the IV, x_1 and x_2. The third ciphertext depends on IV, x_1, x_2, x_3, and so on for all the other plaintexts left; the last ciphertext results from a function of all plaintexts and the IV. Figure 3.9 shows the encryption process used in the CBC operation mode in encryption/decryption.

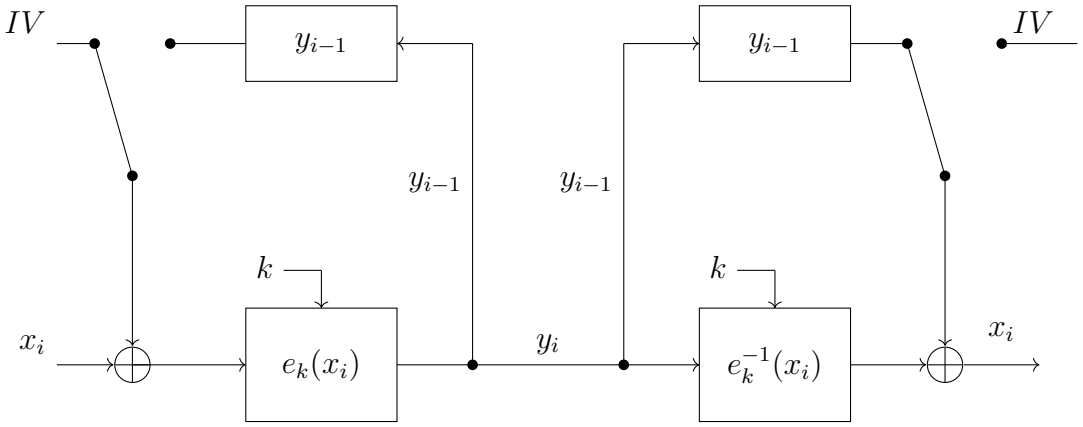

Figure 3.9: CBC block diagram for encryption and decryption.

The expression $e_k^{-1}(y_i) = x_i \oplus y_{i-1}$ illustrates the general decryption process for all ciphertexts except the first y_0 ciphertext also the right side of figure 3.9 illustrates the complete decryption process. It begins with the decryption of the first ciphertext y_1, the plaintext result should be XORed with the IV to get the plaintext x_1 i.e., $x_1 = IV \oplus e_k^{-1}(y_1)$ and is described as:

$$Encryption(1^{st} \text{ block}) : y_1 = e_k(x_1 \oplus IV)$$

$$Encryption(\text{general block}) : y_i = e_k(x_i \oplus y_{i-1}), i \geq 2$$

$$Decryption(1^{st} \text{ block}) : x_1 = e_k^{-1}(y_1) \oplus IV$$

$$Decryption(\text{general block}) : x_i = e_k^{-1}(y_i) \oplus y_{i-1}, i \geq 2$$

We obtain to verify that the decryption reverses the encryption on the first block:

$$d(y_1) = e_k^{-1}(y_1) \oplus IV = e_k^{-1}(e_k(x_1 \oplus IV)) \oplus IV = (x_1 \oplus IV) \oplus IV = x_1 \qquad (3.8)$$

For all subsequent blocks, we have:

$$d(y_i) = e_k^{-1}(y_i) \oplus y_{i-1} = e_k^{-1}(e_k(x_i \oplus y_{i-1})) \oplus y_{i-1} = (x_i \oplus y_{i-1}) \oplus y_{i-1} = x_i \qquad (3.9)$$

Therefore can choose a new IV every time we encrypt. If we encrypt a message with the first IV and perform a second encryption of the same message with a different IV, the two resulting ciphertexts look entirely unrelated.

3.2.3 Counter Mode (CTR)

Therefore exists the possibility to use a block cipher as a stream cipher; this is the case of the Counter Mode, which computes the Keystream on the fly. Also, the block cipher inputs are a counter with a different value every time the block cipher computes a new Key. Figure 3.10 shows the encryption and decryption process of the Counter Mode.

The user should not use the same input value in this operation mode. Otherwise, if an attacker has access to one of the encrypted ciphertexts with the same inputs, it is possible to compute the Keystream and immediately decrypt the other ciphertext. In practice, the use of AES allows inputs of 128-bit. So, the user should first choose a nonce (number used only once) with a length of 96 bits. The counter value uses the remaining 32-bit with zeroes. The counter increments by one during encryption, but the IV remains static. This example can encrypt up to 2^{32} block without choosing a new IV.

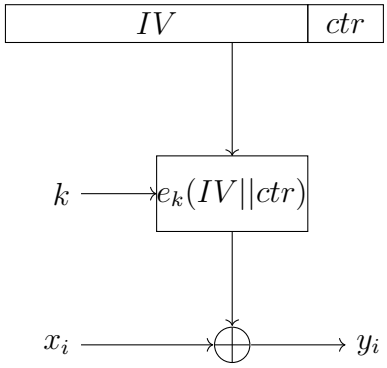

Figure 3.10: Counter mode general diagram

The following functions describe the Counter Mode: the block cipher encryption $e()$ with size b, and x_i as the input plaintext and y_i the ciphertext both with length b, the concatenation of the IV and the CTR_i is denoted by $(IV \parallel CTR_i)$.

$$Encryption : y_i = e_k(IV \parallel CTR_i) \oplus x_i, i \geq 1$$

$$Decryption : x_i = e_k(IV \parallel CTR_i) \oplus y_i, i \geq 1$$

Alice can generate the public value of the concatenation $(IV \parallel CTR_1)$ and then send it to Bob with the first ciphertext. The counter used in practice uses integer values

but can have more complex values. Like ECB, this operation mode has the advantage of parallelization with more encryption units deployed.

3.3 Message Authentication Codes (MACs)

The Message Authentication Code (MAC), also known as cryptographic checksums or Keyed hash functions, is widely used in cryptography, these functions provide security services such as message integrity and authentication, but they do not provide non-repudiation. Therefore, MACs are much faster than digital signatures used in public key cryptography [98].

The MACs create an authentication tag and append it to a message, which differs from digital signatures used in Public Key Cryptography. Instead, MACs use Symmetric Key k to generate and verify the tag t.

$$m = MAC_k(x) \tag{3.10}$$

Figure 3.11 illustrates the MAC calculation and verification. MAC is used in practice because Alice and Bob want to detect manipulations of the message x in transit. For this, both parties share a secret Key, k, and Bob computes the MAC as a function of the message and sends both message and tag to Alice. Alice receives the message and the tag t and proceeds to verify both with the MAC function using the same steps by Bob.

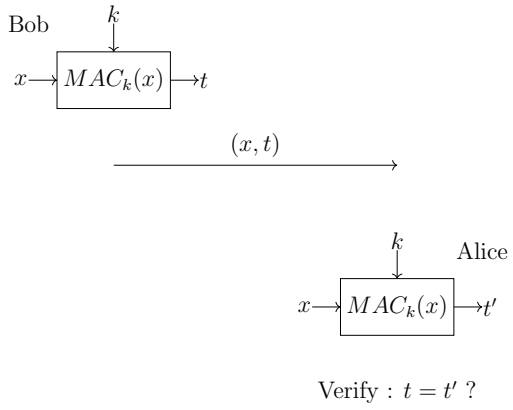

Figure 3.11: Message authentication codes general diagram and verification

The MAC verification will fail if there are any alterations in the message x; this is the message integrity security service. Furthermore, Alice is now sure since Bob is

the originator of the message and only he has the secret Key. If an adversary, Eva, intercepts the message and makes changes, those changes rice on an invalid tag t', and the verification fails. The following list shows some properties of the MACs.

- Cryptographic checksum: generates a cryptographic tag t for a given message.
- Symmetric: parties share a secret key k to sign and verify the tag t and message x.
- Arbitrary message size: MAC functions allow messages of arbitrary length.
- Fixed output: generate a fixed-size authentication tag.
- Message integrity: can detect any message manipulations.
- Message authentication: the receiving party knows the message's origin.

In practice, the construction of MAC has two different ways, the first from block ciphers and the second from hash functions. In addition, Transport Layer Security (*TLS*) protocol in web applications to secure communications and *IPsec* protocol in network applications.

All hash-based MACs always hash the Key k together with the message, allowing various possible ways to perform HMAC, first one:

$$m = MAC_k(x) = h(k \parallel x) \tag{3.11}$$

Also called the secret prefix MAC, and second:

$$m = MAC_k(x) = h(x \parallel k) \tag{3.12}$$

Known as the secret suffix MAC, concatenation denoted by "$\parallel$" in both cases means the concatenation of the message x and the key k. The following step is to process the resultant string. Due to the properties of modern hash functions, both approaches are cryptographic checksums.

3.3.1 HMAC

Figure 3.12 shows this scheme proposed in work [12]. The first step begins by expending Key k with zeroes on the left so that the result k^+ has b bits length as the input block. Next, the expanded Key is *XORed* with the inner pad, which has the bit pattern.

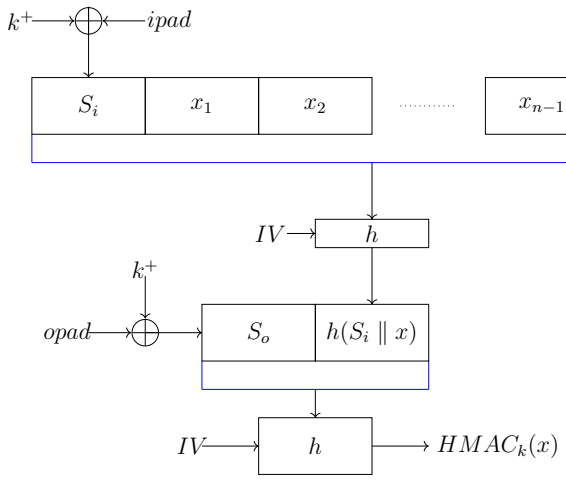

Figure 3.12: HMAC diagram

$$ipad = 0x36, 0x36, ..., 0x36 \tag{3.13}$$

The output from the XOR goes to the first hash function. Also, the message blocks $(x_1, x_2, ..., x_n)$, the second hash process the padded Key k with the output of the first *Hash*, and once again, the Key k gets expanded with zeroes and *XORed* with the outer pad:

$$opad = 0x5C, 0x5C, ..., 0x5C \tag{3.14}$$

The input for the outer Hash comes from the result of the XOR operation, and the other is from the inner Hash. Finally, the output from the outer Hash is the Message authentication code of message x, and the following formula shows the general construction of the HMAC.

$$HMAC_k(x) = h[(k^+ \oplus opad) \parallel h[(k^+ \oplus ipad) \parallel x]]. \tag{3.15}$$

The output length becomes more extensive than the b width. For example, the hash function SHA-1 produces an output of 160-bit length and accepts an input width of $b = 512 - bit$. Therefore, the message x can have an arbitrary length and get processed by the inner hash function; meanwhile, the outer Hash has to process two blocks, as shown in the figure 3.12; thus, the HMAC construction has overhead and low speed.

3.3.2 MAC from block ciphers

One widespread implementation of MAC using block cipher is the use of AES block cipher, but it is possible to use any block cipher. Figure 3.13 shows the primary setting for using a block cipher in a MAC application.

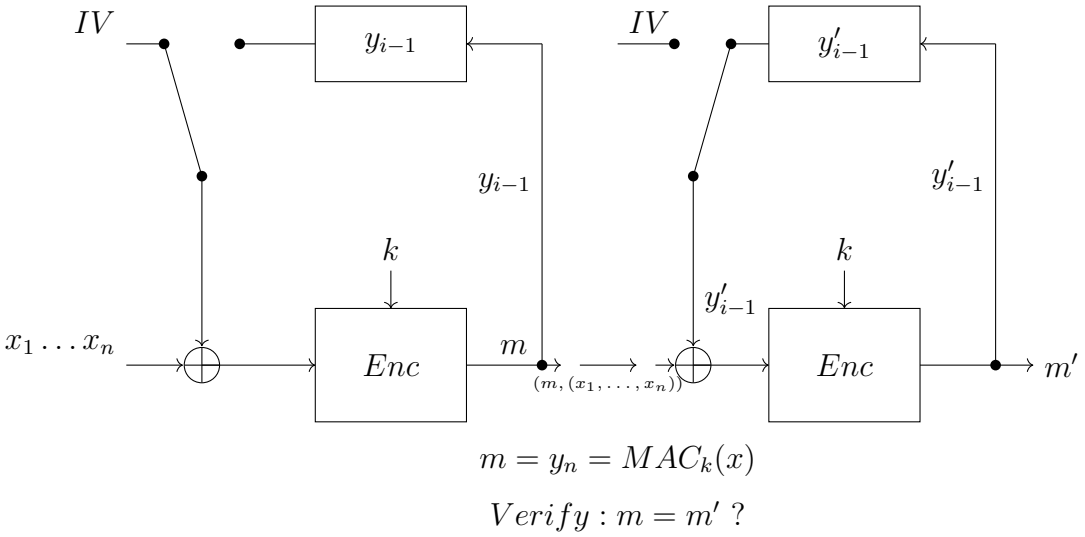

Figure 3.13: MAC implementation based on CBC operation mode

The MAC generation starts by dividing the message x into blocks x_i, with $i = 1, ..., n$, a secret Key k and IV are needed, then proceed to process the first iteration of the MAC algorithm as:

$$y_1 = e_k(x_1 \oplus IV) \tag{3.16}$$

With IV as a random value, for all following message blocks, we use the actual x_i and the previous output y_{i-1} as input for the encryption algorithm:

$$y_i = e_k(x_i \oplus y_{i-1}) \tag{3.17}$$

Finally, when processing the last block, the MAC output is y_n in the last round:

$$m = MAC_k(x) = y_n \tag{3.18}$$

Moreover, all $(y_i, \ldots, y_{n-1})$ are internal values used to compute the final MAC.

3.3.3 MAC Verification

To verify the MAC produced, the user needs to repeat the steps used to generate the MAC. The verification compares the computed MAC m' with the received MAC m. If both are equal, $m = m'$ means the correct verification of the message; if $m \neq m'$ means an alteration of the message or MAC value m during the transmission.

3.4 Hash Functions

Also known as a (*digest*), the *HASH* function is widely used in various protocols and computes a digest from an input message; the digest has a fixed-length bit-string. However, the digest produces a unique representation of a message like a fingerprint, i.e., no other message has the same fingerprint. Therefore this kind of cryptographic function has primary use in Message authentication Codes *MACs* and other protocols of Public Key Cryptography.

Like any other function in cryptography, the HASH must fulfill specific security characteristics like the three shown in the following list.

- Preimage Resistance: given an input hash, h is infeasible to find the message $x, h = H(x)$, i.e., given a fingerprint, we cannot make the inverse operation $x = H^{-1}(h)$, this means the Hash is a one-way function.
- Second Preimage Resistance: given a message x_1 and its hash h_1 is infeasible to find another message x_2 such that their hashes are equal $H(x_1) = H(x_2)$.
- Collision Resistance: it means that it should not be possible to find two different messages, x_1 and x_2, with the same Hash, $x_1 \neq x_2$ then $h(x_1) = h(x2)$, this is the most challenging property to achieve.

3.5 Lightweight Cryptography

It is a subfield of cryptography that provides cryptographic solutions tailored for constrained resource devices like RFID cards, wireless sensor networks, and smart cards. However, shifting from desktop computers to small devices brings many new challenges in security and privacy because it is difficult to apply conventional cryptographic standards to constrained-resource devices.

Many cryptography standards have a tradeoff between the security provided, performance, and resources required and optimized for desktop and server environments. These requirements make them complex and sometimes impossible to implement in constrained resource devices. When they are suitable, they have poor performance. There exists an extensive investigation made by the academic community focused on lightweight cryptography, including efficient hardware implementations (chapter 5 presents some hardware implementations of NIST lightweight contest), software implementations, and implement efficient conventional cryptographic standards, therefore the design and analysis of lightweight algorithms and protocols.

NIST begins the LWC project to evaluate the performance of current cryptography algorithms approved by NIST on constrained resource devices to understand the need for dedicated LWC standards.

LWC primarily focuses on the highly constrained devices found in RFID tags, wireless sensor networks (WSNs), and the Internet of Things (IoT); those devices use 8-bit, 16-bit, and 32-bit microcontrollers. Each has an instruction set with small, simple instructions; this results in using many clock cycles when executing a cryptography algorithm.

3.5.1 Performance

Cryptography algorithms have a tradeoff between performance and resources required for a determined security level. The performance measure considers the resources required for hardware implementation, gate area, gate equivalents, or logic blocks. In software, this directly affects the consumption of registers used, RAM, and ROM memory.

The most relevant metrics in constrained devices are power and energy consumption, and power has particular relevance in devices that harvest power from their surroundings, usually electromagnetic fields. Energy becomes essential to battery-operated devices with a fixed amount of energy stored. Sometimes, the batteries are difficult and impossible to recharge or replace.

Latency focuses on real-time applications, like automotive, where the response of critical components like car control is required. The latency is the time required since the input enters the circuit and gets an output, i.e., the time to process some data.

Throughput is the production rate of new outputs, unlike conventional algorithms in *LWC*; this is not a goal, but moderate rates are still required.

3.5.2 Lightweight primitives

Several proposals for LWC primitives include block ciphers, hash functions, message authentication codes, and stream ciphers. Although they differ from conventional algorithms and focus only on LWC applications, this may constrain an attacker's power.

3.5.3 Lightweight Block ciphers

The block ciphers must achieve performance advantages over AES-128 when implemented on constrained devices and achieve the following characteristics.

- **Smaller block sizes**: the block ciphers can use a smaller block size than AES-128, for example, 64-bit or 80-bit. Also, a smaller block size limits the maximum amount of plaintext block encryption.
- **Smaller Key sizes**: Present [20] use a small key size of 96 bits.
- **Simpler rounds**: the operations and components used in LWC usually become simpler than those used in conventional block ciphers. For example, using smaller to 8-bit S-boxes presents an advantage in saving the area used by the implementation.
- **Minimal implementations**: several operation modes and protocols require only the encryption function of a block cipher like the COmet [114] comet presented in chapter 5 and only require the implementation of one operation (encryption or decryption). This lead to implementing only the required functions of a block cipher and may require fewer resources than the entire block cipher implementation.

3.5.4 Lightweight MACs

As presented earlier in this chapter, a message authentication code MAC generates a tag from a message and a secret Key. The Key and Tag both needed to verify the authenticity and integrity of the message. The size of the tag has 64-bit for typical applications.

NIST has initiated the LWC standardization process to evaluate lightweight cryptographic algorithms suitable for constrained environments. Next, table 3.2 shows the timeline of the whole standardization process.

Date	Event
July 2015	NIST first LWC workshop
October 2016	NIST second LWC workshop
March 2017	Publication of NIST IR8114 report on LWC [84]
April 2017	Profiles for LWC standardization process (draft)
August 2018	Federal Register Notice, Requirements, and evaluation criteria for the LWC process
February 2019	Submission deadline
April 2019	Anouncement of the first-round candidates
August 2019	Anouncement of the second-round candidates
October 2019	Status report on the first round of LWC standardization process NIST IR8268 [124]
November 2019	Third NIST LWC workshop
September 2020	Deadline for optional status updates
October 2020	Fourth LWC workshop (virtual)
March 2021	Announcement of the finalists
July 2021	Status report on the second round of the LWC standardization process, NIST IR8369 [115]
May 2022	Fifth LWC workshop (virtual)

Table 3.2: NIST Lightweight Cryptography Standardization process timeline

3.6 Authenticated encryption with associated data

An authenticated cipher or an authenticated-encryption scheme encrypts and authenticates messages using a public nonce and a Secret Key k. The sender and receiver previously share the secret Key k. They can use different combinations of block ciphers, stream ciphers, message authentication codes MACs, and Hash functions.

Also, an authenticated encryption with associated data ($AEAD$) algorithm has four-byte strings and produces one-byte string output. The four inputs can have variable lengths: plaintext, associated data, a fixed-length nonce, and a fixed-length Secret Key.

This scheme can recover plaintext from a valid ciphertext, i.e., the ciphertext corresponds to the plaintext given associated data (*AD*), *nonce*, and *Key*, and only returns the plaintext if the verification process of the ciphertext is valid. This algorithm ensures two security services, confidentiality, and integrity of the messages. It expects these algorithms to maintain security while the nonce remains unique (never used more than once with the same key).

Therefore, AEAD algorithms should use a key of at least 128-bit length, a nonce of at least 96-bit, and a tag of at least 64-bits. Also, a limit exists for only processing inputs smaller than $(2^{50} - 1)$ bytes.

3.7 Hardware API for Lightweight Cryptography

The cryptographic engineering research group of George Mason University [1] created a hardware Application Programming Interface (API) to help hardware developers create a homogeneous interface for implementations for lightweight authenticated ciphers, hash functions, and cores with both functionalities. The API meets all requirements of all candidates submitted to the NIST LWC standardization contest.

A standard API for all hardware implementations aims to:

- Fair benchmarking between different implementations of the same algorithm.
- Compatibility between implementations of the same algorithm.
- Creation of a standard development package to simplify and accelerate the design process.

Exists a previous attempt to standardize a hardware API [70] took place during the SHA-3 contest [50, 58]. They used the interface proposed for the group from George Mason University (GMU) [50, 58, 51]. This interface and protocol were adopted later in the LWC contest. Also, in the subsequent Competition for Authenticated Encryption: Security, Applicability, and Robustness contest (*CAESAR*), the API was used, but the CAESAR committee made all decisions.

A recommended criterion is listed below in the requirements the API needs.

[1] https://cryptography.gmu.edu/athena/

- **Encryption/Decryption**: This implementation must be into the core, and only one can execute simultaneously. This requirement shows the ability to share resources between decryption and encryption.
- **Hash**: The algorithm has this option. The developer has to implement two versions of the LWC core performing encryption. Decryption and hashing. And encryption and decryption only.
- **Key Scheduling**: The LWC core must have this implementation inside because each variant of the scheduling unit has its requirements specific to each algorithm.
- **Incomplete Blocks**: The LWC core can handle incomplete blocks, whether ciphertext, plaintext, associated data, or Hash messages.
- **Padding**: The hardware implementation has to deal with padding directly.

The API also has its maximum supported input sizes of Associated Data (AD), plaintext, ciphertext, and hash message; the following lists the maximum sizes for each.

- $[2^{16} - 1]$ default.
- $[2^{32} - 1]$ used for compatibility with CAESAR API.
- $[2^{50} - 1]$ minimum limit established by NIST for algorithms submitted to the LWC standardization contest.

The core supports ciphertext sizes of at least $2^{16} - 1$ by default, and each implementer can eliminate the size limits (with hard effort) on the:

1. Maximum clock frequency.
2. The total number of clock cycles for short messages.
3. Throughput for long messages.

The designers must meet all requirements presented in the specification of the Hardware API for LWC (LWC_HW_implementers_Guide [70]). Therefore the developer must ensure his code is in Hardware Description Language (HDL), ensuring its portability by various tools. Figure 3.14[1] shows the top-level block diagram of the LWC core. This architecture helps the designers understand all inputs and outputs

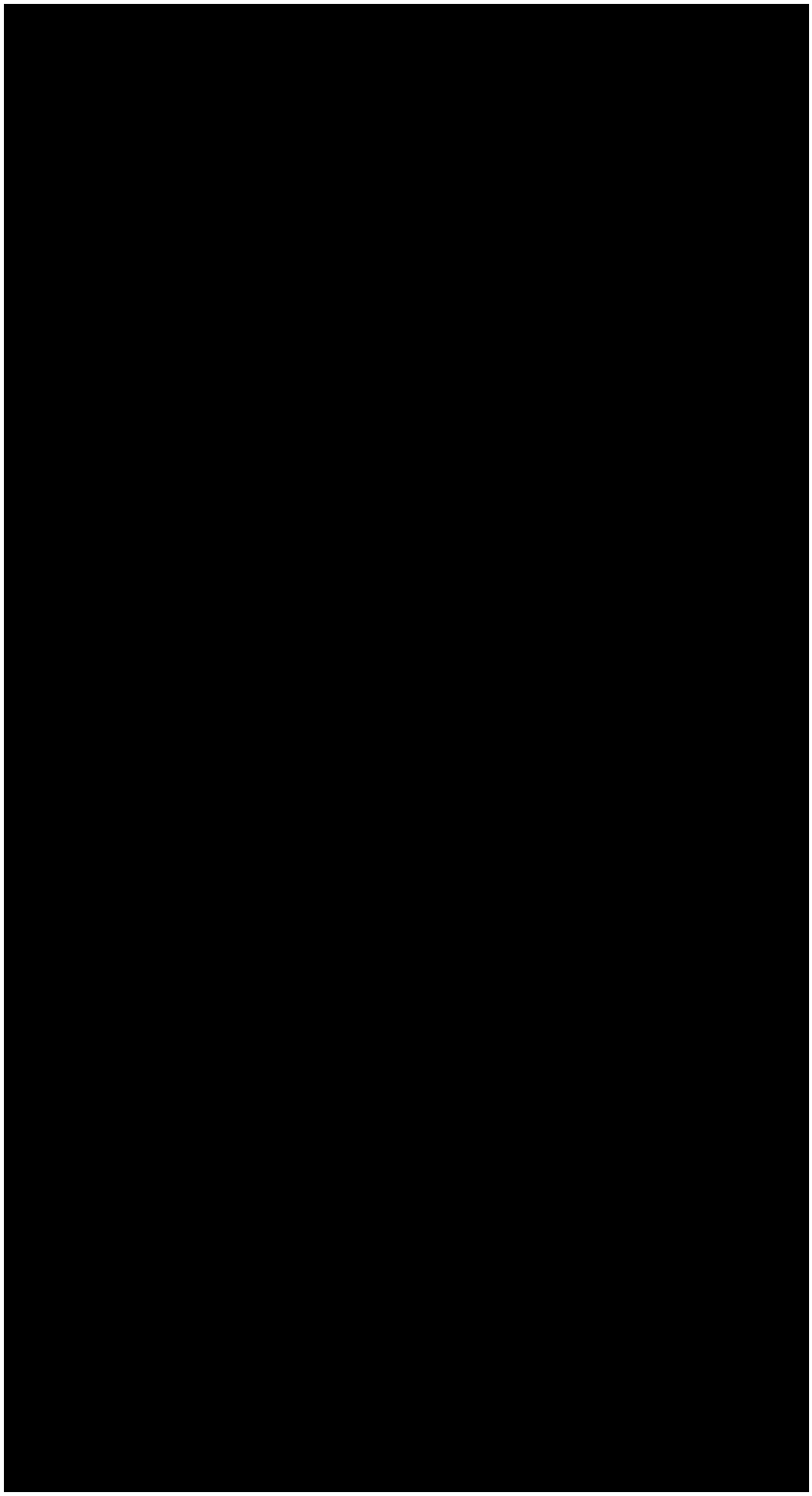

Figure 3.14: Architecture diagram for single pass core used in AEAD by GMU.

the algorithm needs to implement. The diagram shows four units: *Pre-processor, CryptoCore, Header FIFO, and PostProcessor*.
The preprocessor handles the following tasks:

- Parsing segment headers.
- Load Keys.
- Parsing input data directly to the CyrptoCore, also incomplete blocks.
- Keep track of the data left to process.

The PostProcessor handles:

1. Clears portion of the output (words) of ciphertext or plaintext with zeroes.
2. Set the header for output data blocks.
3. Set status block with the result of authentication.
4. Header FIFO: A FiFo of $4 \times w$ that stores all segment headers needed at the output.

The official Athena API website[1] has complete documentation for All features, characteristics, and requirements presented here. Therefore extra information is available to get a comprehensive understanding of the functionalities of this API, and even there is extra documentation focused on implementation against side-channel attacks focused on protected implementations.

[1]Image borrowed from LWC_HW_implementers_Guide version 2.0
[1]https://cryptography.gmu.edu/athena/

3.8 Public Key Cryptography

As another application of cryptography, Public Key Cryptography focuses on providing different security services that Symmetric Key can not handle. The main goal is to share a secret between two or more parties, even if they have never met, and if the channel used is insecure, secure data transmission becomes available for all parties in the channel [55].

The advantage of symmetric cryptography is its high efficiency, but it faces some problems; the main problem here is the (key distribution problem), and it requires a channel with both authentication and secret for the distribution of keys. The second problem is (key management) because in a party with N participants, each participant has to store different keying material with each other N entities, and each of them has to maintain keying material with the other $(N-1)$ participants.

In contrast to Symmetric-key, Public-key cryptography requires the participants to exchange authentic keying material but not necessarily secret. Each participant chose a key pair (e, d) with e as the Public key and Private key d related to e (the private key should remain secret). Both keys have the property that becomes infeasible to find the private key just from knowing the public key of someone.

Public-key cryptography offers the following security services:

- **Con dentiality**: Bob wants to communicate with Alice. First, Bob must obtain Alice's public Key e_A and use it in an encryption function ENC from any public-key encryption scheme. To compute the ciphertext $c = ENC_{e_A}(m)$ and transmits c to Alice, who uses the decryption function DEC and her private Key d_A to recover the plaintext $m = DEC_{d_A}(c)$. The presumption is that an adversary with knowledge of e_A but no d_A cannot decrypt the ciphertext c. This example shows no need for an established secret between parties. Also, the only requirement is for Bob to obtain an authentic copy of e_A. Otherwise, Bob would encrypt the plaintext m using the Public Key of Eva e_E pretending to be Alice, and then Eve can recover the message encrypted by Bob.

- **Non-repudiation**: In this scenario, Bob uses a signature generation algorithm, SIGN, and his private Key d_B to compute a signature of a message m with the signature algorithm $s = SIGN_{d_B}(m)$. Alice will receive both the message m and the signature s from Bob; now, Alice already has an authentic copy of Bob's public Key e_B. She can use the same signature algorithm to verify the

signature s. She can assure from the message m and the Private Key d_B that Bob sent that message and his signature. Therefore, a third party can verify the signature s and message m if Bob denies having signed the message m.

Unlike a handwritten signature, Bob's signature s depends on the message m, which prevents forging a signature only from s to a different message m' and claims that Bob's signed message m'.

Using the previous services is how Public-key cryptography solves the three problems. Symmetric-key cryptography can not solve Key distribution, Key management, and non-repudiation. Therefore, all Public-key cryptography algorithms have lower performance than Symmetric-Key cryptography algorithms.

Summarizing in Public-key cryptography, each party selects a Key pair so that the problem of deriving the Private Key from the public Key becomes a challenging problem that is believed intractable. Public-key schemes use Numeric-theoretic problems as a base for their security, and some schemes are:

1. RSA public-key encryption and signature, whose complex relays in the integer factorization problem.

2. The discrete logarithm problem in elliptic curves cryptographic schemes.

3.9 RSA

It uses the algorithm presented in [55] the algorithm 9 to generate a Key pair. The Public Key consists of a pair of integers (n, e) with the modulus (n) as a product of two random primes (p, q) with the same length in bits, here the encryption exponent e satisfies

$$1 < e < \phi \tag{3.19}$$

and

$$gcd(e, \phi) = 1 \tag{3.20}$$

with

$$\phi = (p - 1)(q - 1) \tag{3.21}$$

The private key d satisfies

$$1 < d < \phi \tag{3.22}$$

and

$$e \times d \equiv 1(\text{ modulo } \phi). \tag{3.23}$$

The security remains in the fact that determinate d given a public-key (e, n) has difficulty in determinating the factors (p, q) from n.

Algorithm 9 Key generation in RSA.

Require: l as the length of the key pair.
Ensure: Key pair (d, e) and public parameter n.
1: Select two random primes p, q with same length $(l/2)$.
2: Compute n, ϕ with $n = p \times q$, and $\phi = (p-1)(q-1)$.
3: Randomly select e with $1 < e < \phi$, and $gcd(e, \phi) = 1$.
4: Compute d with $1 < d < \phi$, and $ed \equiv 1(\text{ mod } \phi)$.
5: Return (e, d, n).

3.9.1 RSA encryption scheme

Both schemes of encryption and decryption use the following fact

$$m^{ed} \equiv m(\text{ modulo } n) \tag{3.24}$$

for any integer m. Algorithms 10 and 11 in [55, 86] show Both basic Public-key procedures of RSA encryption and decryption. The decryption scheme has the following feature

$$c^d \equiv (m^e)^d \equiv m(\text{ modulo } n) \tag{3.25}$$

as shown in the equation above 3.25. The security of this scheme relies on the difficulty of calculating the plaintext m from the ciphertext $c = m^e$ modulo n from the public parameters n and e. The challenge is to find the eth roots (modulo n), and suppose it is as tricky as the integer factorization problem.

Algorithm 10 Simple RSA encryption.

Require: Plaintext $m \in [0, n-1]$ and Public Key pair (e, n).
Ensure: The c ciphertext.
1: Compute $c = m^e$ modulo n.
2: Return c.

Algorithm 11 Simple RSA decryption.

Require: Ciphertext c, Public Key pair (e, n) and private key d.
Ensure: The m plaintext.
1: Compute $m = c^d$ modulo n.
2: Return m.

3.9.2 RSA signature scheme

In the signature and verification algorithms [55, 86] presented in 12 and 13, the sender of a message m has to obtain the digest with a hash function $h = H(m)$; here, h serves as the fingerprint of message m. Now the signer can use his private Key d to compute the *eth* root s of h modulo n; $s = h^d$ modulo n.

The signer can transmit the message m and the signature s to someone verifying the signature. Nevertheless, first, the receiver has to calculate the hash $h = H(m)$ and obtains $h' = s^e$ modulo n from s, and he accepts the signature as valid for m provided that $h = h'$. In this scheme, security relies on the capacity of a forger to calculate *eth* roots modulo n without knowing the private Key d from the transceiver.

Algorithm 12 Simple RSA signature.

Require: plaintext m, Public Key pair (e, n) and private key d.
Ensure: The signature s.
1: Compute Hash $h = H(m)$.
2: Compute signature $s = h^d$ modulo n.
3: Return s.

Algorithm 13 Simple RSA signature verification.

Require: Signature s, Public Key pair (e, n) and plaintext m.
Ensure: The signature accept or rejected.
1: Compute Hash $h = H(m)$.
2: Compute $h' = s^e$ modulo n.
3: **if** $h = h'$ **then**
4: Valid signature.
5: **else**
6: Invalid signature.
7: **end if**

The most expensive operation in RSA is modular exponentiation, i.e., calculate

$$m^e \text{ modulo } n \tag{3.26}$$

for encryption and

$$c^d \text{ modulo } n \tag{3.27}$$

for decryption. For practical purposes, both encryption exponents are small, e.g., $e = 3$ or $e = 2^{16} + 1$, and the private key d has the same length as n.

3.10 Elliptic curve scheme

To understand this topic, we present some concepts from group theory and introduce their generalization. Therefore we look at elliptic curves as groups and present their utilization in discrete logarithm implementations [55].

3.10.1 Groups

An abelian group $(G, \cdot)$ comprises a set G with an operation binary called $* : G \times G \rightarrow G$ satisfying the properties:

1. **Associativity**: $a \cdot (b \cdot c) = (a \cdot b) \cdot c \; \forall \; a, b, c \; \in G$.
2. **Identity**: $\exists \; e \in G$ such that $a \cdot e = e \cdot a = a, \; \forall \; a \in G$.
3. **Inverses**: for each $a \in G, \exists \; a^{-1} \in G$, named inverse of a such that $a \cdot a^{-1} = a^{-1} \cdot a = e$.
4. **Conmutativity**: $a \cdot b = b \cdot a \; \forall \; a, b \in G$.

In a group operation $(+)$ addition or multiplication $(\cdot)$, the first operation, called additive group, has an identity element known as 0, and the inverse of an element a is called $-a$. For multiplication, the group is known as a multiplicative group. It also has an identity element denoted by 1, and the inverse of an element a becomes denoted by a^{-1}. Finally, the group becomes a finite field G as a finite set, and the number of elements in G is known as the order of G.

To illustrate, suppose a prime number p, and let $\mathbb{F}_p = 0, 1, 2, 3, ..., p - 1$ as the set of integers modulo p. Hence $(\mathbb{F}_p, +)$ with $+$ as the addition of integers modulo p, as a finite additive group of order p and with 0 as the identity element. Moreover,

$(\mathbb{F}_p^*, \cdot)$ also, $\mathbb{F}_p^*$ denotes all the nonzero elements in $\mathbb{F}_p$ and the operation $(\cdot)$ as the multiplication of integers modulo p, as the multiplicative group with order $p-1$ and 1 as the identity element. Therefore the triplet $(\mathbb{F}_p, +, \cdot)$ is a finite field known as $\mathbb{F}_p$.

A finite multiplicative group G of order n and g in G, the minor positive integer t such that $g^t = 1$ known as the order of g, and always exists a t as a divisor of n. The set $\langle g \rangle = g^i : 0 \leq i \leq t-1$ for all powers of g is itself a group under the same operation as G, known as a cyclic subgroup of G generated by g. For the addition, the order of $g \in G$ is the minor positive divisor t of n such that $tg = 0$, and $\langle g \rangle = ig : 0 \leq i \leq t-1$. Hence, tg is an element acquired by adding t times g. Therefore, G has a g element of order n, G is a cyclic group, and g is a generator of G.

The problems based on Discrete Logarithm (DL), the parameters (p, q, g) as presented at the beginning of the RSA section, the multiplicative group $(\mathbb{F}_p^*, \cdot)$ has an order of $p-1$ and is a cyclic group. Additionally, $\langle g \rangle$ is a cyclic subgroup with order q.

3.10.2 Generalization of discrete logarithm problem

Suppose a multiplicative cyclic group $(G, \cdot)$ of order n and generator g. the domain parameters g and n, the Private Key a random integer x from the interval $[1, n-1]$, the Public Key $y = g^x$. Then, the attacker has to obtain x given g, n, and y, known as the discrete logarithm problem in G.

A Discrete Logarithm (DL) system needs fast algorithms to compute group operations for efficiency. Also, this problem requires intractability [86].

If there are two cyclic groups with the same order n, they essentially are the same; this means they have the same structure but can write the elements differently. Each different kind of representation of group results in varying speeds in algorithms for computing a group operation for solving the DL problem [55].

3.10.3 Elliptic curve groups

With p a prime number and $\mathbb{F}_p$, the finite field in integers modulo p and E and elliptic curve over $\mathbb{F}_p$ defined with the equation of the form

$$y^2 = x^3 + ax + b, \tag{3.28}$$

where $a, b \in \mathbb{F}_p$ satisfying $4a^3 + 27b^2 \neq 0(\text{ modulo } p)$. The pair (x, y), with $x, y \in \mathbb{F}_p$ as a point in the curve if (x, y) satisfies the equation above [34, 55]. The point at infinity, denoted by ∞, also is in the curve. Therefore, we denote all points on E with $E(\mathbb{F}_p)$. As an example, the finite field F_7 has an Elliptic curve E with the following equation 3.29

$$y^2 = x^3 + 2x + 4, \tag{3.29}$$

then all the points of E are

$$E(\mathbb{F}_7) = \infty, (0, 2), (0, 5), (1, 0), (2, 3), (2, 4), (3, 3), (3, 4), (6, 1), (, 6,).$$

Therefore, knowing the points is possible to perform arithmetic operations like addition, subtraction, multiplications, and inversion in $\mathbb{F}_p$ with the respective coordinates x_1, y_1, x_2, y_2, and with the use of ∞ as identity in the abelian group.

3.10.4 Key generation in elliptic curves

With an elliptic curve, defining E over a finite field $\mathbb{F}_p$, and P a point in $E(\mathbb{F}_p)$, and P has prime order n, this generates a cyclic subgroup $E(\mathbb{F}_p)$ generated by P

$$\langle P \rangle = \infty, P, 2P, 3P, ..., (n-1)P. \tag{3.30}$$

The public domain parameters are the prime p, the elliptic curve E equation, and the point P of order n. In addition, the integer d is selected uniformly random from the interval $[1, n-1]$ and used as the Private Key. Finally, it computes the Public Key as $Q = dP$. Furthermore, the discrete logarithm problem relies on determining d with the knowledge of the public parameters Q and the domain parameters E, P as a point in the curve E, and the prime p. The algorithm 14 shows the Key pair generation in a generic elliptic curve scheme.

Algorithm 14 Simple Key generation on elliptic curves.

Require: Domain parameters (P, E, p, n).
Ensure: Private and public Key (d, Q).
1: Uniformly random select $d \in_R [1, n-1]$.
2: Compute public key $Q = d \times P$.
3: Return (d, Q).

3.10.5 Encryption scheme with elliptic curves

Algorithms 15 and 16 show the procedures used in encryption and decryption, respectively [86, 34, 55]. For example, Bob wants to send a message to Alice. The

first step consists of representing the message m as a point M in the elliptic curve E; then, the message gets encrypted by adding it to kQ_A with k, a selected random integer, and Q_A as Alice's Public Key. Now Bob transmits the points $C_1 \equiv kP$ and $C_2 = M + kQ_A$ to Alice who use her Private Key d_A to compute

$$d_A C_1 = d_A(kP) = k(d_A P) = k_{Q_A}) \tag{3.31}$$

and recovers $M = C_2 - kQ_A$. If an Eavesdropper Eve wants to recover the message M needs to compute kQ_A from the domain parameters, Q_A and $C_1 = kP$, we can say that this problem is the analog to factorization problem in RSA.

Algorithm 15 Encryption "ElGamal" in elliptic curves.

Require: Domain parameters (P, E, p, n), public key Q and plaintext m.
Ensure: Ciphertext (C_1, C_2).
1: Convert m as the point $M \in E(\mathbb{F}_p)$.
2: Choose $k \in_R [1, n-1]$.
3: Compute $C_1 = kP$ and $C_2 = M + kQ$.
4: Return (C_1, C_2).

Algorithm 16 Decryption "ElGamal" in elliptic curves.

Require: Domain parameters (P, E, p, n), private key d and ciphertext (C_1, C_2).
Ensure: Plaintext m.
1: Compute $M = C_2 - dC_1$.
2: Extract m from M.
3: Return m.

3.11 Summary

This chapter shows the two perspectives on cryptography, symmetric and public. Both have advantages and disadvantages depending on the applications and the security services required by the application's environment. Therefore, in later chapters, we use the theory presented here to develop new hardware applications from both cryptography perspectives.

Chapter 4

Field Programmable Gate Array (FPGA) and Advance RISC Machine (ARM) technologies

In this chapter, the architecture of both FPGA (Field Programmable Gate Arrays) and ARM (Advance RISC Machine) microcontrollers is presented; also their architecture, different kinds of devices used in the projects, and their characteristics, architectures, resources as well as some particular components used by some of the architectures designed by us.

The chapter has two parts; the first focuses on FPGA devices and families with particular attention to their characteristics and internal components like DSPs (Digital Signal Processors). The second part consists of the general architecture of the 32-bit ARM microcontroller M4, its general purpose registers, and the crypto-core with some of its characteristics like block cipher capability, HASH functions, and MAC (Message Authentication Code) compatibility.

The technology behind reconfigurable hardware looks like a computer with its architecture specialized in implementing any algorithm in a hardware fashion. The main objective of reconfigurable hardware is the possibility of combining software's flexibility with the performance of hardware. These machines have several fixed circuitry assigned as needed since the designed architecture could need more resources or just a few as it may apply.

A difference between a general-purpose processor is the low performance the software

can achieve because the microprocessors and microcontrollers have fixed circuitry and data path that impact the performance. Usually have higher speeds but low performance compared with the FPGAs; these devices can change their behavior with the advantage of only loading a new design to the device.

Today, different applications use FPGA technology to deploy and develop prototypes faster. In addition, they are also used as components in embedded systems in industrial applications [106], video, image, and sound processing [110, 77, 99, 123], security network and their infrastructure [68], and cryptography [67, 59, 78]. This last topic is the main objective of this thesis work.

4.1 Field Programmable Gate Arrays (FPGAs)

In a general way, a Field Programmable Gate Array consists of a large number of interconnections between the different resources inside the FPGA. Modern versions of these devices have many embedded components like memory blocks, digital signal processors, and more, but the capability to interconnect those devices is the primary resource. They contain logic elements that perform logic and arithmetic operations, and the interconnections route the connections among blocks to send and receive data to process.

4.1.1 Logic elements

The truth table is the most common way to represent a circuit; they can model almost any kind of circuit, whatever it could be. In addition, they can represent digital systems relating the inputs with the outputs [108]. Those tables are Boolean functions defined with the letter f as the relation of the possible outputs associated with all possible inputs of a digital system [24]. We can say that these tables are the heart of an FPGA.

The look-up tables (LUTs) implement truth tables in a hardware fashion; they usually consist of N inputs and only one output, i.e., they present only 2^{2^n} possible boolean functions of N variable inputs. However, the FPGA can use more LUTs connected as a cascade to implement functions with more than N inputs. For example, the LUTs implemented in the Xilinx 7-family have six input LUTs [131], but in earlier families like 6, 5, 4, and 3, the LUTs used to have only four available inputs. Figure 4.1 shows two ways used to implement a four input and one output LUTs.

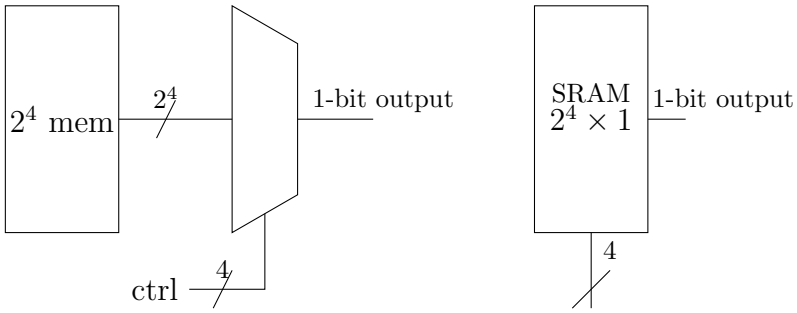

Figure 4.1: Two different four input LUTs

LUTs have internally grouped configurable logic blocks (CLBs) as shown in [33], and their organization varies among all families and FPGA manufacturers. However, in this thesis work, Xilinx is the only brand used, and the name used for their elementary logic block is *Slice*. The logic blocks usually consist of multiple components like multiplexers, extra logic components, registers, and carry inputs. Figure 4.2 shows the diagram of a four-input LUT with *XOR* at the output and a flip-flop register.

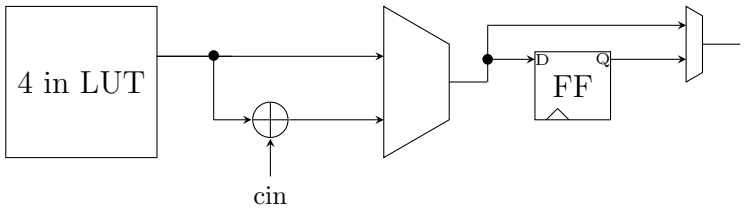

Figure 4.2: LUT with register and *XOR*

The interconnection of each logic block to develop highly complex systems is a must in the FPGA. Those components are configurable and usually have a matrix structure with multiple tracks vertically and horizontally. There are three ways used to interconnect logic blocks, and figure 4.3 shows the island placement interconnection on FPGAs

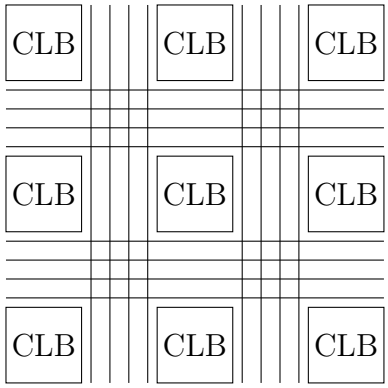

Figure 4.3: FPGA Island connection between CLBs

Nearest neighbor: figure 4.4 shows the simplest structure used to interconnect the logic blocks; a two-way connection with the nearest neighbor in each direction, north, south, east, and west, and there is no kind of elements to bypass any logic block, each signal go through each logic block. This last aspect directly impacts the performance and increases the delay.

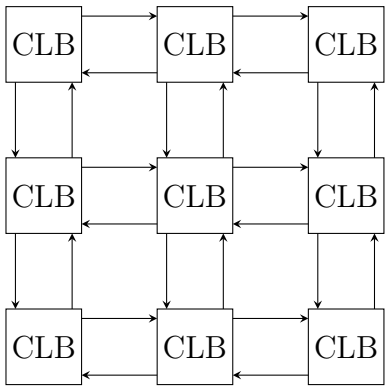

Figure 4.4: Nearest neighbor structure

Also, as each CLB interconnects with another in any direction imposes a limitation on the capabilities to interact directly with other CLB in other neighborhoods; this disadvantage allows them to interact with closer CLB in the same area directly.

Segmented: figure 4.5 shows the segmented structure; here, the component in-

terconnections are:

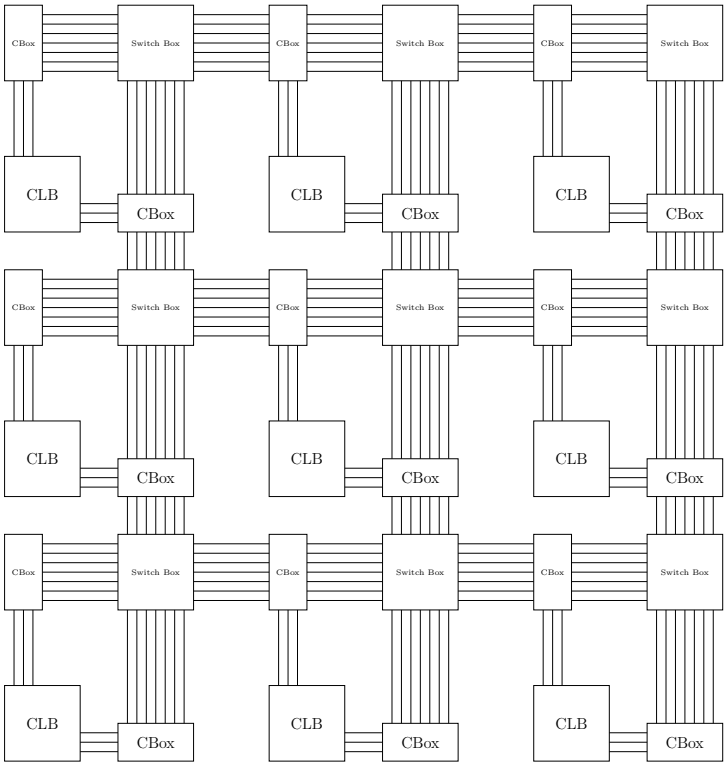

Figure 4.5: Logic blocks in an island fashion with connection block and switch boxes in the same architecture

- **Tracks:** The buses have vertical and horizontal tracks connected directly through connection blocks (CBs). The cross formed by vertical and horizontal tracks consists of a switch box used to interconnect the CBs with them.
- **Connection Blocks:** They handle inputs and outputs from the logic blocks, connecting each of them to several tracks. This kind of connection allows selecting which link is active.
- **Switch Boxes:** This is a way to connect logic blocks between layout routes between vertical and horizontal crossings. If a connection is needed, the programming of switch boxes creates a new path to send/receive data between logic blocks. Figure 4.6 shows a CLB with come programmable connections used to interconnect the CLB with its neighbors, and some works focused on switch box design are [43, 44].

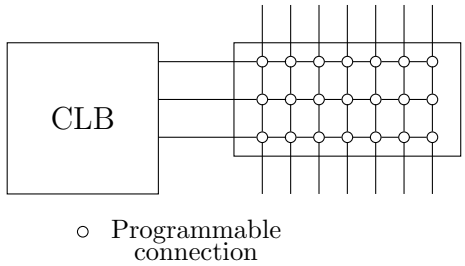

Figure 4.6: Programmable connection block

The segmented architecture offers excellent flexibility and performance, allowing the connection of CLBs in this architecture requires all signals to go through switch and connection boxes, increasing the delay in the circuit design. However, in some scenarios, this has the advantage of allowing a direct connection between components. Figure 4.7 shows the hybrid connection architecture.

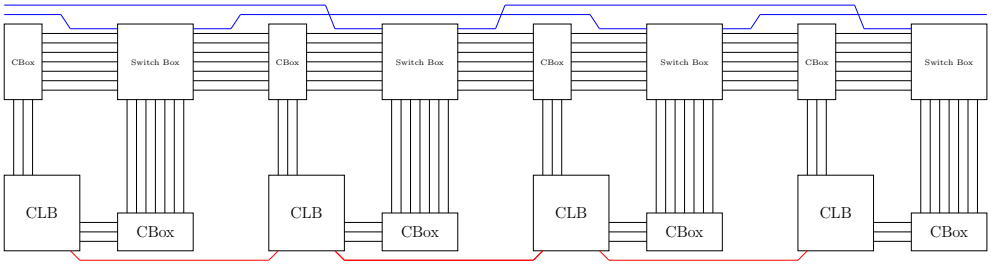

Figure 4.7: Hybrid structure of nearest neighbor and segmented structure

Hierarchical

In some FPGA architectures, the logic blocks have a cluster arrangement hierarchically. Therefore, the appropriate use of these blocks means an improvement in reducing the delays and enhancing signal routing. Thus, the essential components of this kind of architecture look like segmented architecture. Figure 4.8 shows the hierarchical architecture with up to 64 Logic blocks. This requires fewer switches and has faster logic compared with a segmented model. Some studies about this structure are [79, 39].

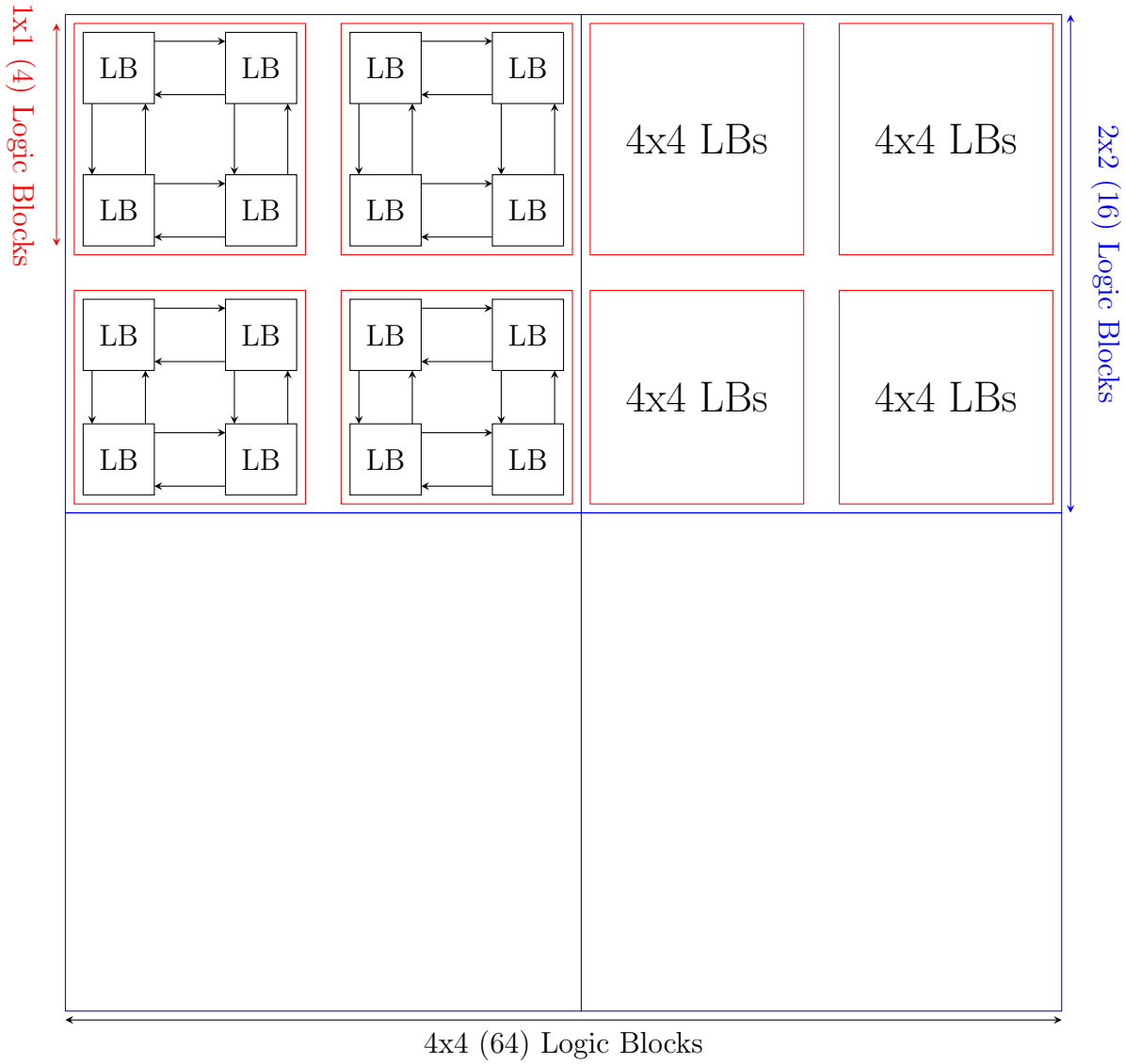

Figure 4.8: Hierarchical structure with a cluster of logic blocks

4.2 Digital Signal Processor (DSP)

The Digital Signal Processor (DSP) is essential in modern FPGA architectures. These devices can handle basic operations like arithmetic, logic, and also comparisons of the inputs; thus, they can process single instruction multiple data in their internal

architecture. We use two kinds of DSPs in the works presented in this thesis. The first is the DSP48E1 [129] found in all Xilinx 7 families. Also, they have low power consumption and scalable capability across families of FPGAs.

Figure 4.9[1] shows the DSP internal architecture. It has four input ports, A, B, C, and D, and all of them have different bus sizes; for example, the A port has a 30-bit width while B has 18-bit width, so it is possible to can concatenate them to use a port of 48-bit width. The D port has a 25-bit width, and the A port is used in the pre-adder to add two operands. Finally, the C port has 48-bit width and goes directly to the DSP's Arithmetic Logic Unit (ALU). This component can perform different operations using up to 48-bit width inputs.

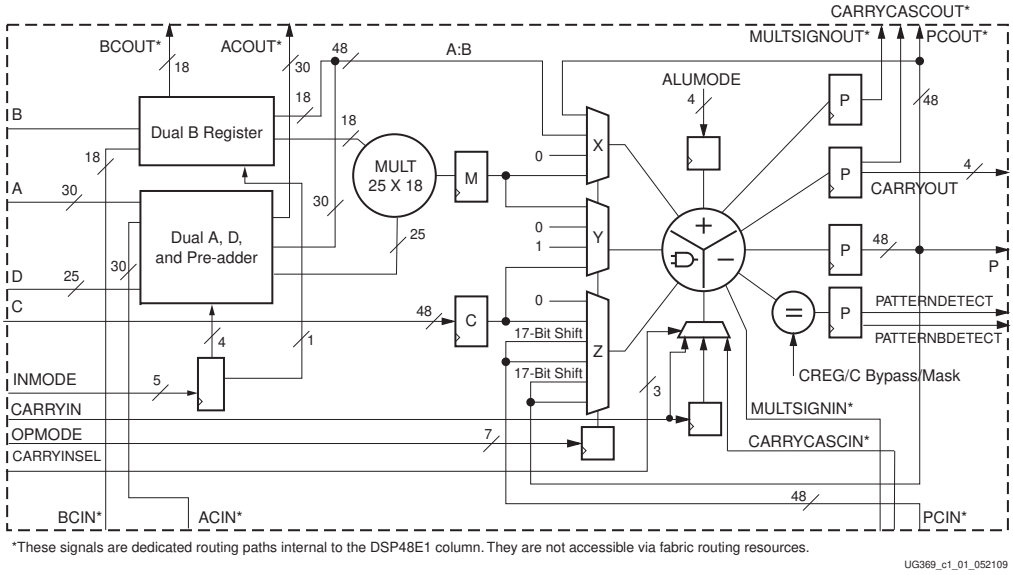

Figure 4.9: DSP48E1 architecture.

Thus, the DSPs have another kind of input; these come directly from other DSPs, making the tile arrangement of DSPs into the FPGA possible. In addition, each DSP column has a dedicated interconnection network that allows each DSP to share data using the ports ACIN, BCIN, CARRYIN, and PCIN to input data. Furthermore, the ports ACOUT, BCOUT, CARRYCASCOUT, and PCOUT must send data from the DSP interconnection network.

[1]Figure taken from Xilinx official documentation

The DSP interconnection network allows the broadcast of an input or output to other DSPs in the same network; this reduces the FPGA interconnection resources used. Therefore the cascade interconnection is only available in the same column, and they cannot connect to other columns directly.

The arithmetic portion of the DSP slice (DSP48E1) consists of a two's complement multiplier of 25×18 bits preceded by a pre-adder of 25-bit. The pre-adder output connects directly to the multiplier, and the output of the multiplier is connected outright to one of three 48-bit data path multiplexers. Finally, the data on the outcomes of those three multiplexers are connected to the ALU to perform any operation required.

Here present some examples of some operations performed by the DSP slice. For example, an addition operation, if the operands have up to 25-bit width, looks like the following formula $(A + D) = P$, remembering that A has 30-bit width and D has 25-bit width, and they are connected directly to the internal pre-adder. To perform the same process using other ports $(C + CONCAT) = P$ with CONCAT port, the concatenation of the $A|B$ ports into a 48-bit width port and C a native 48-bit port, the add ALU in the DSP slice performs the arithmetic addition.

Another essential operation for us is the multiplication performed by the internal multiplier. The asymmetric multiplier can complete only the following process $A \times B = P$. Therefore, the DSP slice can perform multiple operations using the INMODE port. This port has a 6-bit and allows to DSP slice to perform up to 2^6 individual arithmetic logic operations.

Another advantage of these components is combining internal operations thanks to the internal registers found in the DSP slice. The user can enable or disable de registers to exploit the pipeline with up to 6 stages or only activate a few registers as needed. Figure 4.10 shows the configuration windows that enable/disable the registers.

Some possible operations are:

$$(A \times B) + C = P$$

$$(A + D) \times B = P$$

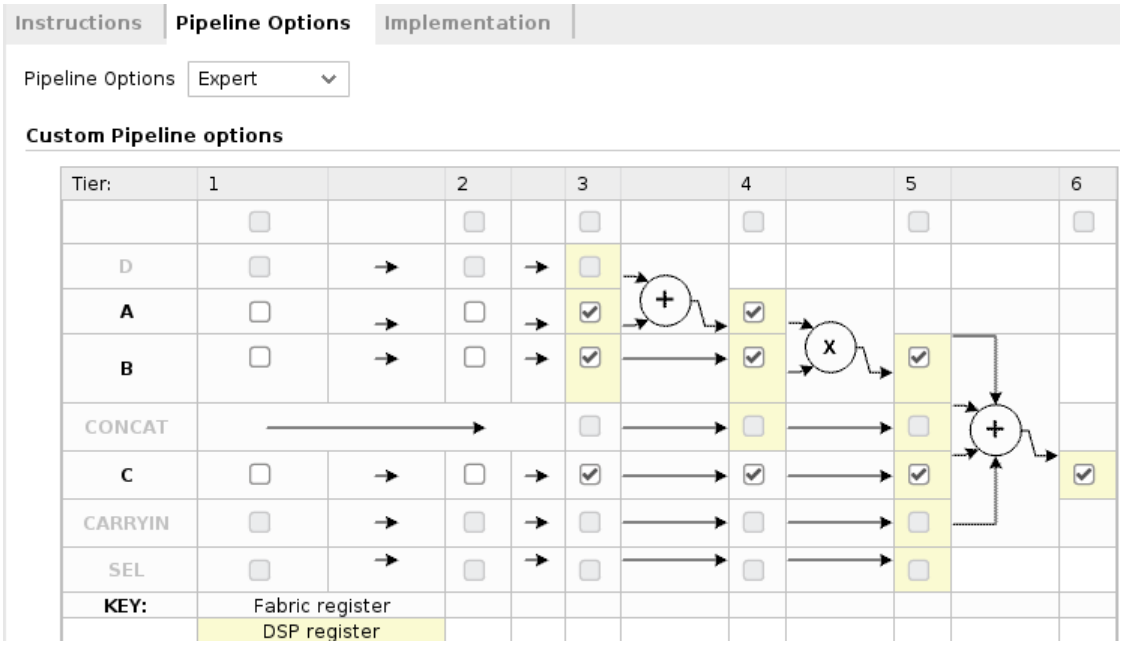

Figure 4.10: DSP pipeline configuration window.

$$(A + D) \times B + C = P$$

The previous operations use the standard ports embedded in the DSP slice, and one DSP can perform one or multiple individual or combined functions as required.

The operations using the dedicated interconnection networks are:

$$(ACIN \times B) + C = P$$

$$(A \times BCIN) + C = P$$

$$(ACIN \times B) = P$$

Previous functions exploit the standard inputs connected directly to the FPGA interconnection network. Using the dedicated DSP slice network, they can combine both inputs to benefit the hybrid connection.

Therefore another embedded function in the DSP slice is the ability to achieve

a 17-bit right shit directly on the output port P or in the input port PCIN. This possibility allows faster data processing when multiplication over significant inputs.

The DSP slice has two possible outputs, the first goes directly to the FPGA interconnection network, and the other possibility is to send the data through their dedicated interconnection network to another DSP in the same column to perform another arithmetic/logic operation.

4.2.1 DSP48E2

This DSP slice in the Xilinx 7 family UltraScale [132] series has these programable logic devices that are also efficient for digital signal processing and can implement fully parallel algorithms as their predecessors. They also have a high speed with a smaller size in the circuit focus; this version enhances the applications' speed and efficiency beyond digital processing, such as dynamic bus shifters, memory address generators, and memory-mapped I/O registers.

This enhanced DSP slice has the same standard inputs ad its predecessor, *DSP48E1*, with the following differences:

- 18 x 27 two's complement multiplier.
- Pre-adder grew to 27-bit wide.

It is possible to use a 48-bit accumulator with the possibility of up to 96-bit cascaded cable to build large accumulators, adders, and counters.

Therefore, the ALU still has 48-bit inputs to perform the following bit-wise operations: AND, OR, NOT, NAND, XOR, NOR, and XNOR. The pipeline still has six stages with four columns of dedicated registers with the possible use in cascade to send data to the next DSP slice in the same column.

There are some differences between the *DSP48E2* and the *DSP48E1* and enlisted here:

1. the multiplier improved in the width of port A to 27-bit.
2. the A and D register now have 27-bit widths.
3. now is possible to select between A or B as input for the pre-adder

Possible operations are:

- $(A + D)$, or, $(B + D)$
- The pre-adder output can be squared.
- The ALU can now handle four inputs at the same time.

Both DSP slices use signed arithmetic, which sometimes reduces the ability to benefit from the full use of the input width.

Figure 4.11 shows the internal components of this DSP slice and the buses used internally. Also, figure 4.12 illustrates the DSP tile with block ram on the left side, and this block RAM consists of two 18K bit individual blocks in the center showing the CLBs interconnection network, and on the right side illustrates two DSP slices.

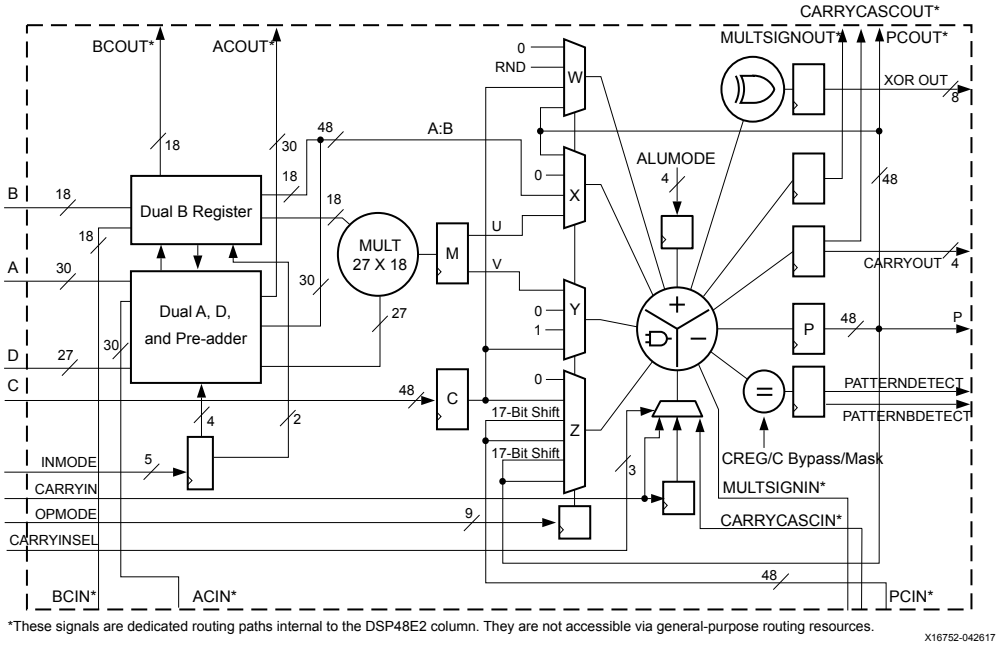

Figure 4.11: DSP48E2 internal architecture

Other independent functions are:

- Multiply.

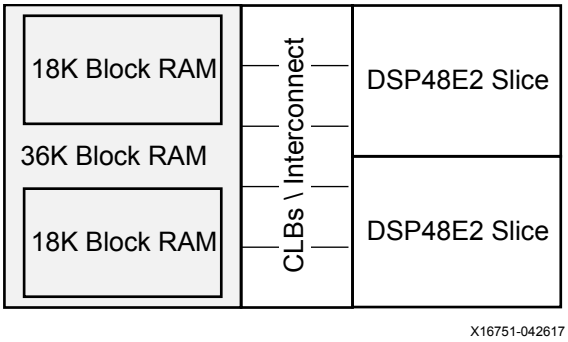

Figure 4.12: DSP interconnection

- Multiply and accumulate
- Multiply and add $(A \times B) + C$
- Four-input adds are only available in the ALU.
- Barrel-shift.
- Comparator.

Figure 4.13 illustrates the three possible inputs for the ALU unit, and those inputs come from the three Multiplexers (MUX) X, Y, and Z, respectively.

Both of the DSP slices presented here have a similar architecture with significant changes (see table 4.1) that improve the performance of the designs made by the user. Therefore, depending on the needs, the possibility of handling vast DSP slices is solved by the DSP slice found in the ultra-scale family; this variant is presented later in this chapter.

4.2.2 Xilinx FPGA Families

Xilinx has had different families of FPGAs since its foundation. In this section, we use the Xilinx-7 family, usually with varying processes of fabrication and sub-brands, and each has a fixed amount of resources depending on the sub-brand. The following list shows the sub-brands and their general view of them.

Spartan 7: they are the cheapest option, with enough resources for connectivity and processing applications in industrial, automotive, and communications. It

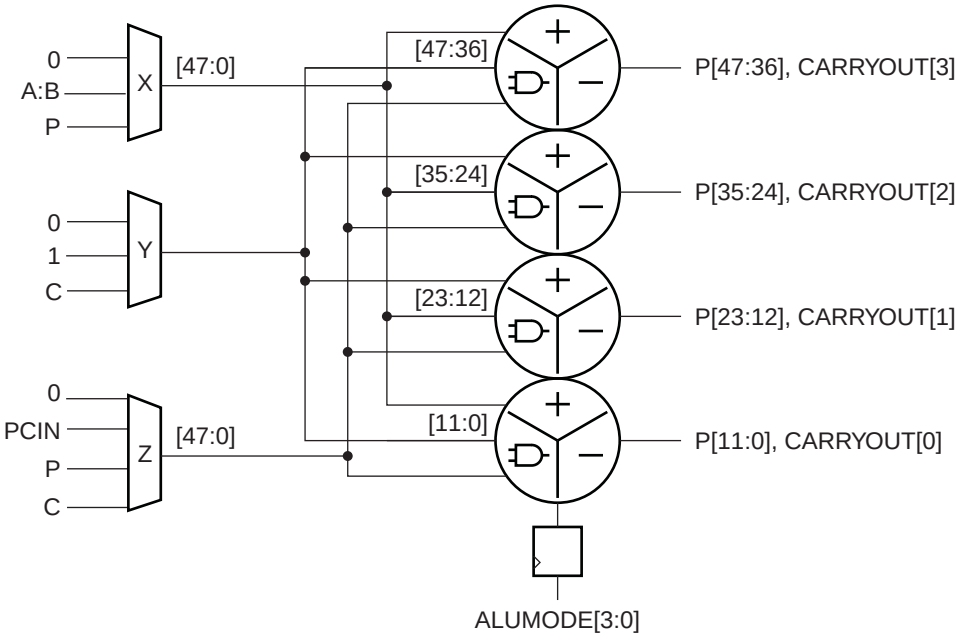

Figure 4.13: Internal ALU found inside a DSP slice

has between 6-102k logic cells and has the lowest-power consumption and DSP slices, and this family has up to 551 MHz of speed processing.

Artix-7: They have better performance than Spartan and greater processing bandwidth and portability. It came with 740 DSP48E1 slices with DDR3 RAM Memory support and had between 13k- 200k logic cells with 2x logic, 2.5x block RAM, and 5.7x more DSP slices than previous generation families. Chapter 6 of this thesis work presents using an Artix-7 FPGA to implement some Lightweight cryptography algorithms among the ATHENA tool.

Kintex-7: This type of FPGA has high DSP slice ratios and relies on other technologies like PCI express generation 3 and 10 Gigabit ethernet connectivity; this family usually is focused on wireless and video solutions. Therefore has up to 478K logic cells, and its DSP slice can achieve 629 MHz with 1920 DSP slices.

Virtex-7: This sub-brand of FPGA has the highest quantity of resources with the highest speed available above the previous sub-brands. It is also optimized. They have the same resources but with the difference of better performance; therefore can have up to 2 million logic cells, 85Mb Block RAM, and 3600 DSP slices. Table 4.2

	DSP48E1	DSP48E2
Multiplier	25 x 18	27 x 18
Pre-adder	25 bits	27 bits
D input	25 bits	27 bits
AD register	25 bits	27 bits
Pre-adder out squaring	no	yes
Four input add	no	yes

Table 4.1: Summary of the difference between characteristics of the DSP48E1 and DSP48E2

	Logic Cells	DSPs	Memory
Spartan-7	600-102400	10-160	180k-4320k
Artix-7	12800-215360	40-740	720k-13140k
Kintex-7	65600-477760	240-1920	4860k-34380k
Virtex-7	582720-1139200	1260-3360	28620k-67680k

Table 4.2: Comparative table of Xilinx 7-series resources

shows the resources available for the different families presented above.

Chapter 7 presents some architectural designs used to multiply large numbers to perform RNS multiplications, and chapter 8 shows a dedicated implementation of two different multipliers focused in the elliptic curve $2^{255} - 19$ using integer arithmetic and comparing Schoolbook multiplier against Karatsuba multiplier [72].

4.3 Architecture of Xilinx 7 family

This section shows the general architecture of the FPGAs used in this thesis work and their components. This topic is essential to understand because all our designs have different requirements and goals.

4.3.1 Con gurable Logic Block (CLB)

Xilinx FPGA older families have different kinds of CLB; it is essential when a designer develops any new architecture, and understanding its characteristics allows one to

exploit its capabilities. However, CLBs found in Xlinix-7 families [133] have the same kind of CLB [130] presented here.

The primary logic sources are the CLBs to implement sequential and combination circuits; each CLB has a direct connection to the switch matrix allowing access to the general routing matrix. Also, each CLB contains a pair of slices.

The LUT implemented in these CLBs allows the configuration as either a 6-input LUT with one output or a two 5-input with individual output sharing the addresses or logic inputs. In addition, there is the option to store the output of one of the 5-input Lut into a flip-flop directly, and figure 4.14 shows the general architecture of the CLB.

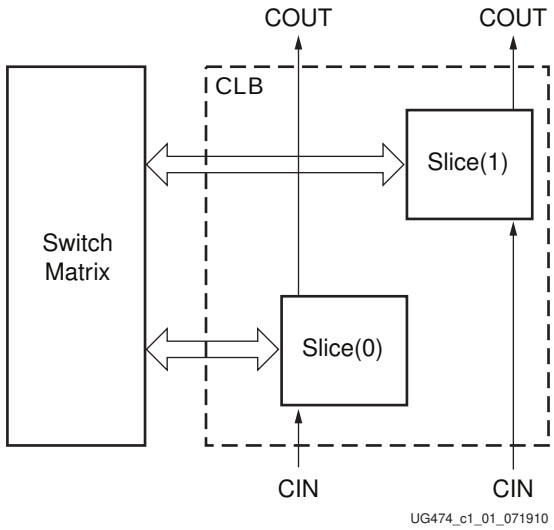

Figure 4.14: CLB internal arrangement and interconnection matrix.

Another important use of the LUT is the capability to use them as different kinds of components with other options; the first option allows using them as a distributed 64-bit RAM or a 32-bit shift register or as two 16-bit shift registers. Modern tools can use that logic, arithmetic, and memory features. A designer with enough experience can instantiate them and configure them as required. Below present a list of the main characteristics of a CLB.

- 6-bit look-up table (LUT).

- Dual 5-input LUTS (optional).
- Shift register logic and distributed RAM capability.
- High-speed carry logic for arithmetic functions.
- Use of wide multiplexers for efficiency.

As mentioned, all 7-series families have scalable resources, providing a homogeneous architecture. However, the amount of CLB differentiates all the Xilinx FPGA families. Therefore, the device capacity is directly related to the number of logic cells supplied with the equivalent of a classic 4-input LUT and one flip-flop.

Each family has its own CLB arrangement in the 7-series; the arrangement is in a column fashion. They use a proprietary technology developed by Xilinx named Advanced Silicon Modular Block (ASMBL) that allows FPGAs with a mix of features to optimize the interconnections and use of the resources. Figure 4.15 shows the component fount into each column across all the FPGA.

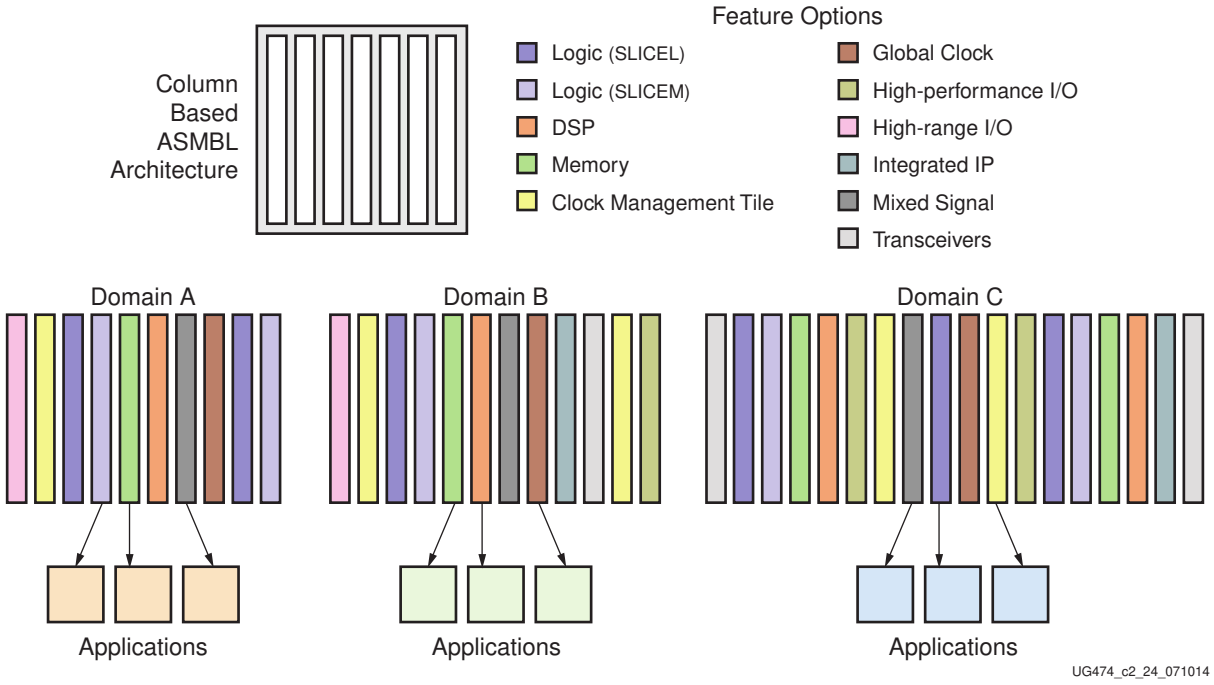

Figure 4.15: ASMBL architecture with components as columns.

The Stacked Silicon Interconnection (SSI) enables super logic regions SRLs. They are another layer to combine communications between resources in a layer fashion, allowing the creation of FPGAs with many interconnections between components.

CLBs

In the family presented here, each CLB has two slices, and each Slice has four 6-bit LUTS and eight storage elements:

$Slice_0$ at the bottom of the CLB in the left corner

$Slice_1$ at the top of the CLB

The slices do not have a direct connection, and the organization of each Slice is a column. For example, the figure 4.16 shows each Slice has its carry independent from the other, and each Slice has:

- Four LUTs.
- Eight storage elements.
- Multiplexers used in wide functions.
- Carry logic.

As mentioned above, all listed aspects allow the Slices to provide arithmetic functions and logic along with extra tasks such as storage, distributed RAM, and shift registers. Figure 4.17 shows the memory architecture implemented by SLICEM, and figure SLICEL shows all elements that are part of any SLICE architecture. Remember that each CLB can contain two Slices of type SLICEL or a combination of both.

4.3.2 Look-Up Table (LUT)

As mentioned, the LUT in the 7-series family can use 6-bit look-up tables. Those individual inputs A_1 to A_6, with the outputs O_5 and O_6, can implement up to four function generators (A to D) in a single Slice. The following list presents some of the capabilities of the LUTs.

- A 6-bit boolean function.
- Two 5-bit boolean functions with shared inputs between them.

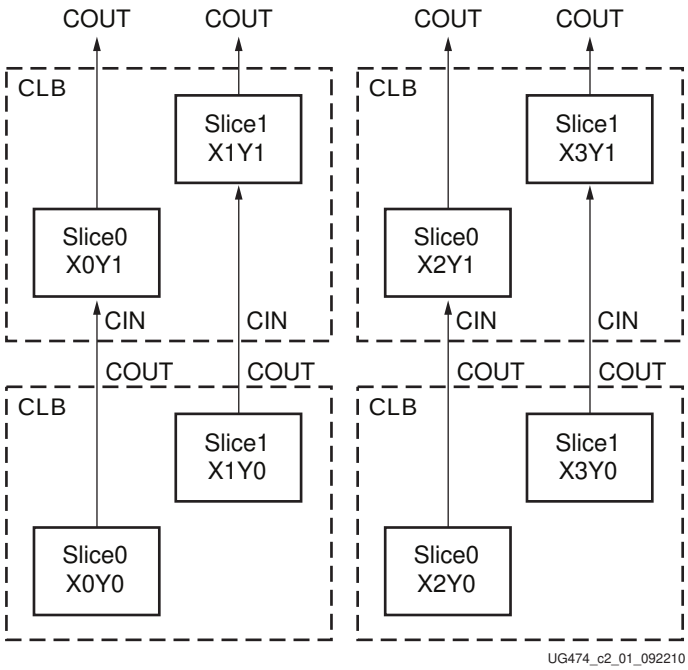

Figure 4.16: CLBs and Slices with Carry inputs and outputs.

- Up to two boolean functions with three or two inputs.

When designing a circuit, the number of inputs is an essential topic in the case of LUTs, and they can change their signal propagation independent of the function implemented. Usually, the outputs of a LUT are A, B, C, D, O_6, or any of the following multiplexers AMUX, BMUX, CMUX, DMUX, and the output O_5 can handle the output as required. In addition, three additional multiplexers, F7AMUX, F7BMUX, and F7CMUX, can combine up to four single functions to provide functions with up to eight inputs using a single Slice. If more than eight inputs are needed, one multiplexer named F8MUX combines all LUTs in a Slice.

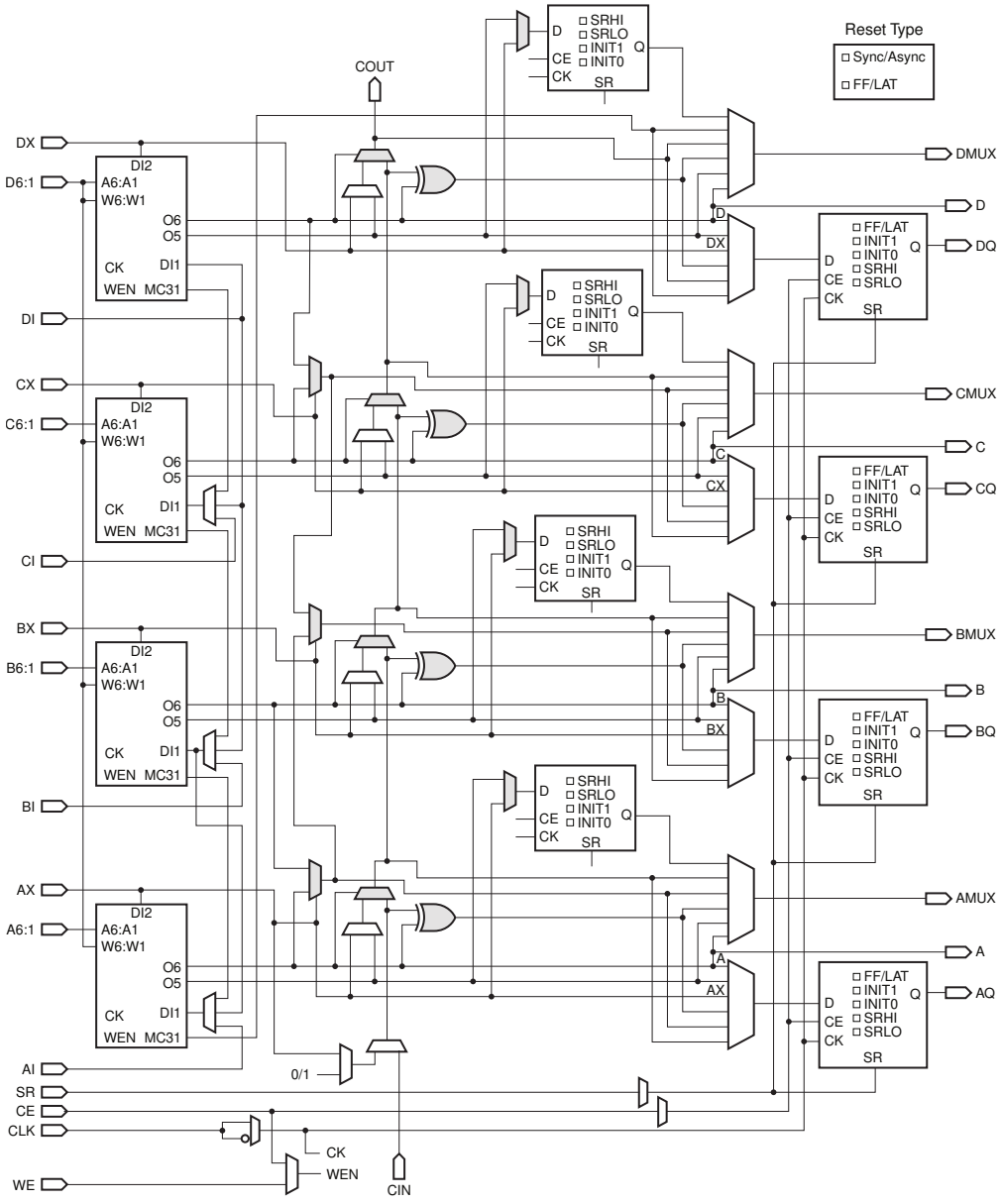

Figure 4.17: Slice M architecture.

4.4 Advance RISC Machine (ARM)

Microprocessors and microcontrollers can execute any algorithm, usually using their fixed data path and with some dedicated resources. Here the ARM microcontrollers

are explained with some of their characteristics and resources used in later chapters of this thesis work.

In 1985, the story Acron computers were looking to make their RISC-based processor, remembering that (Reduced Instruction set computer) is a kind of computer architecture design focused on using small instructions to reduce their execution time. So in 1987, Acron RISC Machine launched this first processor using 3μm technology [125].

There are several families of ARM microprocessors, each with different characteristics, depending on the scenario and the type of family recommended to perform tasks [15]. For example, with simple 8-bit and 16-bit microcontrollers, a programmer can know their architecture and benefit from this capacity with assembly and other languages like C or C++. Also, ARM has 32-bit processors like the cortex-M0 and M0+. If the application needs more resources and performance, the cortex-A family can run operating systems like Linux and windows for embedded systems. When working at higher development levels, it is not necessary to work directly with the processor resources like memory or registers; it is helpful to understand the architecture when debugging code or to optimize functions at a low level like assembly.

ARM offers three leading families to use in System on Chip (SoC) presented in the table Cortex families in this thesis. The use of cortex-M is a must because of the lightweight cryptography contest presented in chapter 5 and shows the characteristics of the microcontroller ARM-M4.

A processor is not enough to execute instructions; it needs an interface to access memory and peripherals. Several components are inside a microcontroller, like a CPU, a memory protection unit, a Digital Signal Processor (DSP), a Float-point Unit, etc. Figure 4.18 shows the main features and the CPU used in the architecture Cortex-M4, the CPU inside this microcontroller is an Arm-v7-M family processor. This architecture can process data, and its instruction set adds extra addressing modes, conditional execution, bit-processing, and full multiplication support.
Also, it can support 32-bit Simple Instruction Multiple Data (SIMD) to handle intensive single-precision and floating-point tasks; therefore, the FPU can deliver double-precision results. There are several manufacturers of microcontrollers, and the following list shows them and some microcontrollers models. In this thesis work, we used the ST Microelectronics brand.

- ST Microelectronics: ARM-M0, M3, M4, and M7

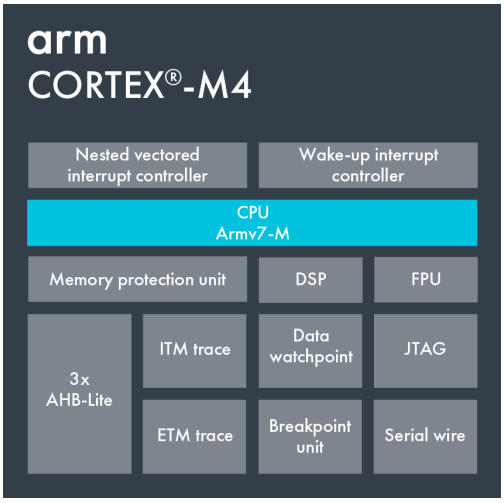

Figure 4.18: ARM M4 general specs

- Microchip: ARM-M0+, M23
- NXP Semiconductors: ARM-M33

Each has its characteristics, but they share the same ARM architecture; the instruction set varies depending on the architecture, and here present some of them.

- Thumb(-1): All the processor's basic operations with 56 instructions; this set is focused on process data and control I/O.
- Thumb(-2): is an expansion of the Thumb(-1) instruction set and adds more instructions for data processing with a conditional branch that depends on previous results and instructions exclusive to use when multiple requests to the system.
- DSP extension instructions: include single instruction multiple data (SIMD) instructions and Multiply-accumulate (MAC) principally used on multimedia applications.
- Single-precision floating-point instructions.
- Double-precision and floating-point instructions: Enable 64-bit precision.

4.4.1 Register Set

The register set used in the Cortex-M family has sixteen 32-bit general-purpose registers and are as follows:

- R0-R7 are the lower registers.
- R8-R12 are the next group of general purpose registers.
- R13 is the stack pointer with two operation modes, MSP (main Stack Pointer) and PSP (Process Stack Pointer). The update of this register is according to the mode in which the core is working.
- R14, named Link Register (LR), contains the return address of a call to a subroutine or function.
- R15 The program counter (PC) contains the address to the next instruction in the program.

The last presented registers, R13-R15, have specific functionalities; those sixteen are the ARM core registers (R0-R15). Figure 4.19 shows the register structure.

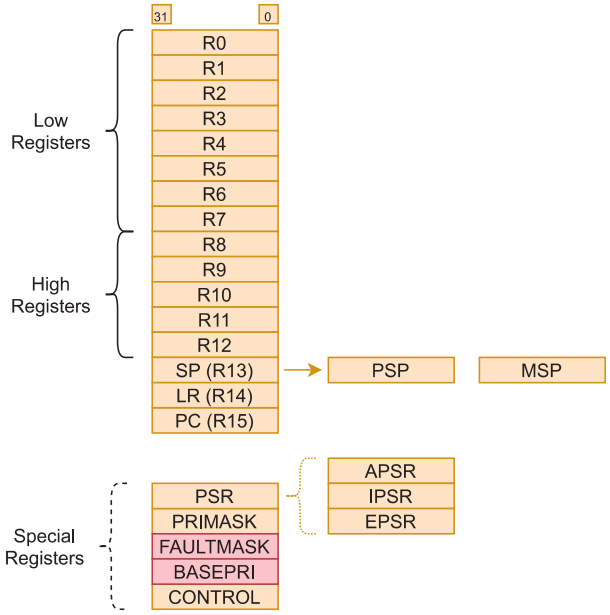

Figure 4.19: ARM general purpose registers.

4.5 ST Microelectronics

Now present the board and ARM processor used in this thesis work. The processor is an ARM Cortex-M4 from the ST Microelectronics brand; the model of the Cortex-M4 processor is STM32L4A6ZG [118] and embedded on a Nucleo-144 board [120]. Figure 4.20 shows the board and all hardware components, and its features of it are:

- USB port with OTG capability.
- Three user LEDs.
- Two user push buttons.
- Micro USB connector.
- On-board ST-LINK debugger programmer with USB capability.

Also, it has several jumpers to change the power input source and behavior, and it contains Zio connectors compatible with Arduino boards.

The embedded ST-Link is a tool integrated into the Nucleo-144 board. It supports debugging and programming features located on the upper of the board. Figure 4.21 shows the connection between the microcontroller and the st-link board with the LEDs, push buttons, and connectors connected directly to the I/O ports on the microcontroller.

The microcontroller has a frequency of up to 80 MHz. In addition, it implements a memory protection unit (MPU) that enhances security applications and embeds high-speed memories (1 MB of flash memory and 320KB SRAM). The security capabilities consider readout protection, write protection, proprietary code readout protection, and firewall for flash memory and SRAM protection [117].

The features allow communication with different devices through the following standard interfaces.

- Four I2C.
- Three SPI.
- Three USART.

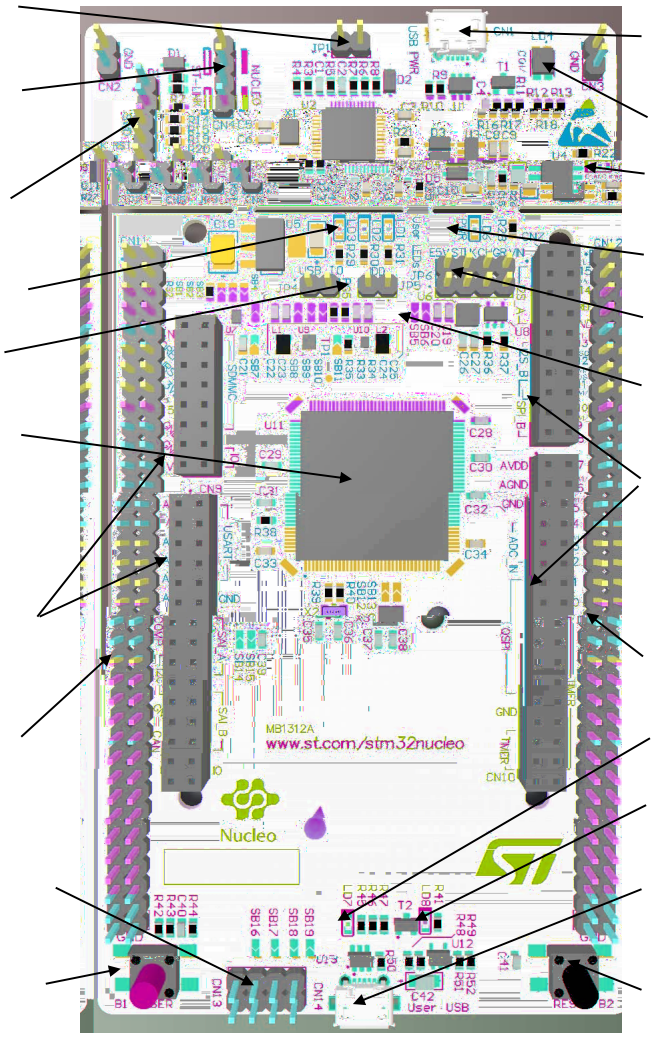

Figure 4.20: ST Nucleo-144 board characteristics.

- Two serial audio interfaces (SAI).
- Two CAN.
- Camera interface.
- DMA controllers.

An important feature of stm32l4a6zg is the capability to use an embedded AES (Advanced Encryption Standard) and HASH hardware accelerator. Figure 4.22 shows

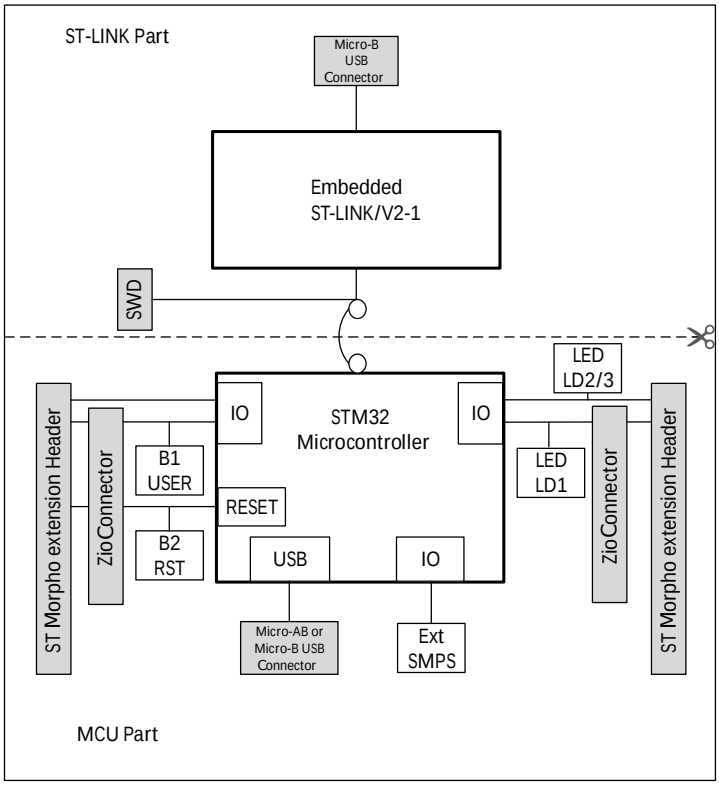

Figure 4.21: ST Nucleo-144 schematic with ST-Link.

the general architecture of the microcontroller embedded on the board Nucleo-144 and its characteristics.

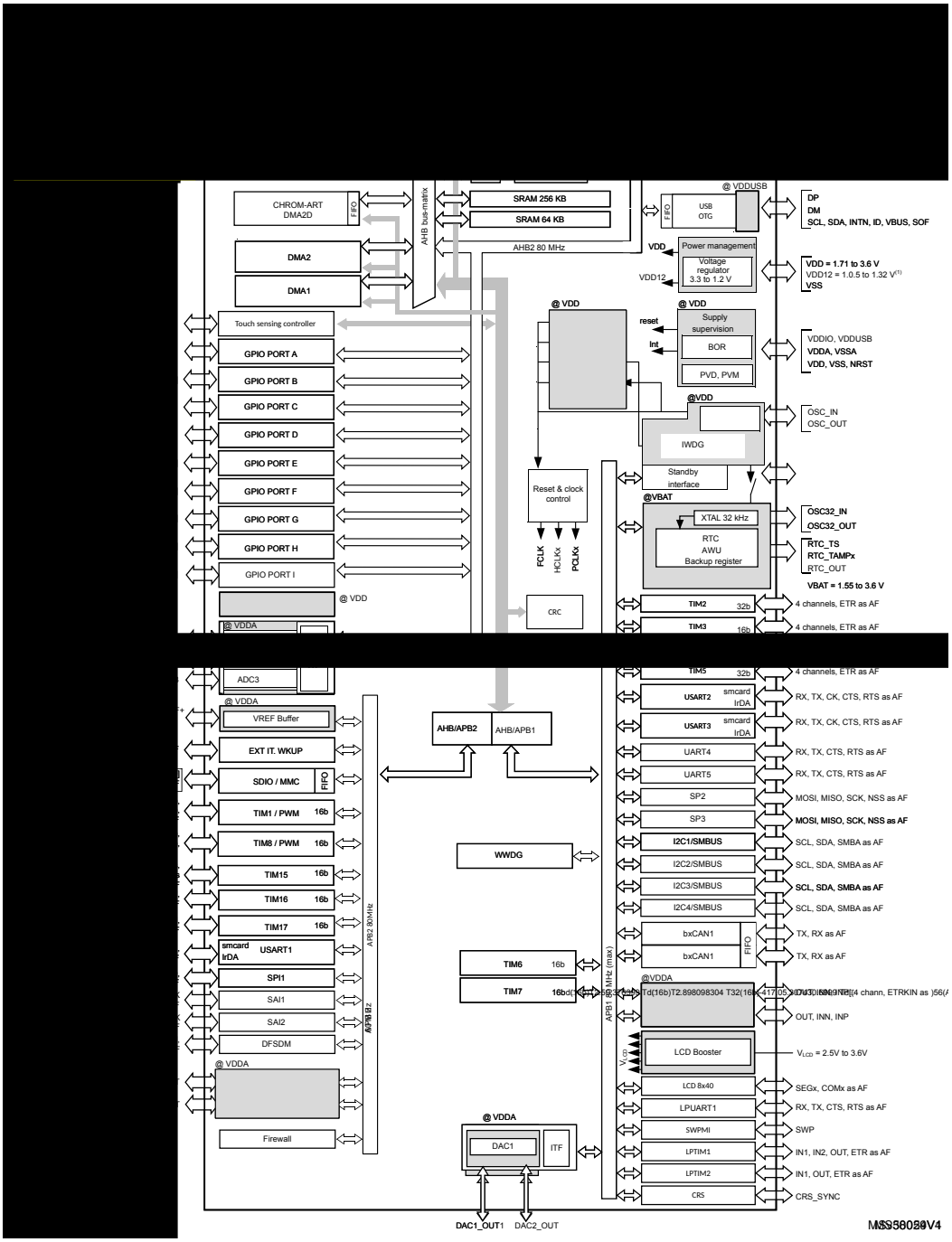

Figure 4.22: STM32L4A6ZG ARM microcontroller internal architecture.

4.5.1 Memory protection unit

The MPU manages the memory access to prevent one task from accidentally corrupting the memory resources or any resource used by any other job. That memory area can protect up to eight regions, and the protection sizes are between 32 bytes and four gigabytes of addressable memory. This unit is helpful where applications have code that needs protection against misbehavior of other tasks; if the program accesses a prohibited memory location, the MPU can detect it and take action; the MPU is optional and bypassed if it is not needed.

4.5.2 General-purpose I/O (GPIO

The software allows the configuration of each GPIO and can alternate as output, input, or a peripheral function; most pins share analog or digital processes. Depending on the application, the user can use each GPIO as agreed on its application.

4.5.3 Direct memory access (DMA)

Provide high-speed data transfer between the memory and peripherals and moves the data faster without CPU intervention allowing the CPU to focus its resources on other operations. The microcontroller presented here has 14 channels, each managing memory access requests from multiple peripherals simultaneously, with one for immediate handling of the priority on each request. The following list shows the DMA supported for this device.

- Configurable channels (14 in total).
- Independent hardware requests to each channel.
- Four priority levels are independently configurable via software and triggered by themselves.
- Memory-to-memory and memory-to-peripheral, and peripheral;-to-peripheral data transfer.
- Direct access to Flash memory, SRAM, and peripherals as source or destination.

4.5.4 Random number generator (RNG)

This microcontroller can use a hardware implementation of a random number generator to generate a 32-bit length random number by an integrated analog circuit. This generator continuously provides 32-bit samples of entropy based on noise from the analog source. The following list shows some RNG features, and figure 4.23 shows its architecture.

Analog entropy source delivers 32-bit random numbers processed with linear-feedback shift registers (LFSR). Every 42 RNG cycle (dedicated clock) produces one 32-bit random sample. As shown in figure 4.23, there are several components, like two analog noise sources fed by an independent clock source. Then the data collected goes to a sampling module; this module sends data to an LFSR for post-processing, and after data goes to the data shift register, the data goes to a register with access to the user.

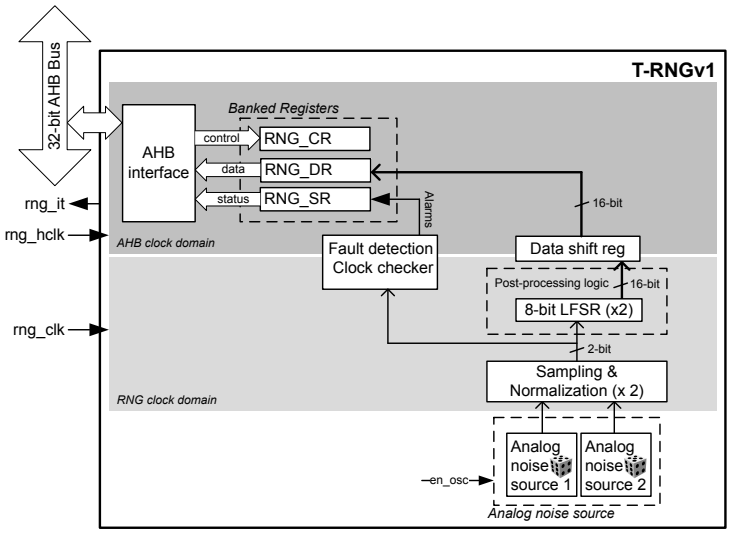

Figure 4.23: Random Number Generator (RNG) embedded in STM32L4A6ZG.

4.5.5 AES in hardware

As mentioned in the chapter 3, AES has implementations on software and hardware, and this microcontroller has an AES hardware accelerator implemented in the same architecture. This implementation allows the user to use the AES block cipher with the performance of the hardware and with some operation modes directly implemented in the same circuit [119] .

This accelerator encrypts and decrypts data using the AES algorithm defined in Federal Information Processing Standards (FIPS) publication 197 [66]. This hardware implementation allows operation modes like (ECB, CBC, and CTR) specified on FIPS publication 800-38A [41], and others based on AES with two different key sizes, 128 and 256 bits. Therefore supports DMA transfers for incoming and outcoming data with a single DMA channel for each data transfer, and the slave peripheral is accessible through 32-bit word single access.

Figure 4.24 shows the architecture of the AES block cipher implemented in hardware. It shows the dedicated registers to store the secret key, initialization vector, and data input (plaintext of ciphertext). Also, there is a module to connect directly with a DMA interface, and the AES core is on the right side of the figure.

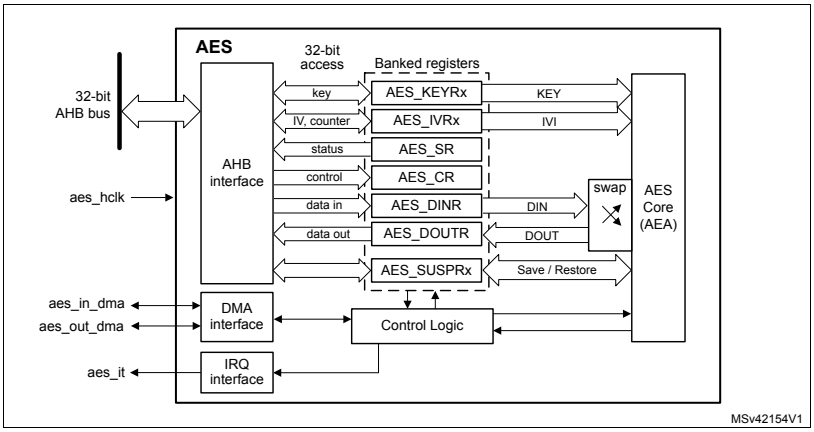

Figure 4.24: AES block diagram.

The cryptographic core has the following components.

- AES algorithm.

- Binary Galois field multiplier (GF_2).
- Input key.
- An initialization vector (IV).
- Chaining algorithm logic (counter mode, feedback).

The core can work with a 128-bit length stored in four 32-bit registers and with a Key size of 128-bit or 256-bit, as required the IV requires a 96-bit vector and a 32-bit counter, and the core can perform the following operation modes.

- Electronic code book (ECB).
- Cipher block chaining (CBC).
- Counter (CTR).
- Galois counter mode(GCM).
- Galois message authentication code (GMAC).
- Counter with CBC-MAC (CCM).

Summary

This chapter shows two different technologies used in developing this thesis work and the capabilities of each device. Meanwhile, the first part evolves FPGA's general characteristics with a brief explanation. The second half focuses on ARM microcontrollers and their characteristics.

Part II

Symmetric Key Cryptography

Chapter 5

Lightweight authenticated encryption with associated data in hardware

In August 2018, the U.S. National Institute of Standards and Technology (NIST) initiated a process to solicit nominations from any interested party for candidate algorithms to be considered for lightweight cryptographic (LWC) standards suitable for use in constrained environments, where the performance of current NIST cryptographic standards exceeds the hardware limits of the devices to be protected. Fifty-six candidates were qualified for the Round 1. For Round 2, 32 candidates were selected.

As a part of the evaluation criteria for the NIST LWC competition, each candidate is evaluated in several aspects, including security evaluation of the algorithms against known attacks, side-channel and fault attack resistances, cost, performance, third-party analysis, suitability for hardware and software implementations.

The Hardware API for Lightweight Cryptography by George Mason University (GMU LWC) was established as the interface to perform the hardware evaluation process. This chapter aims to contribute to the analysis of five candidates of NIST LWC in terms of the established criteria of cost, performance, and suitability for hardware implementations. In this way, these extensive evaluation processes open for anyone who wants to participate guarantee that the proposed cryptographic solutions are reliable for their use in constrained hardware environments, without degrading the main security features that a typical cryptographic primitive should hold, i.e., high-security features with a low footprint.

To be part of these extensive evaluation processes, in the chapter 5 is presented the hardware implementation for five NIST LWC candidates: COMET[114], ESTATE[29], LOCUS[27] , LOTUS[27], and Oribatida[16]. All of these hardware implementations were done in the FPGA Xilinx Artix-7 xc7a12tcsg325-3, with area restrictions, where a medium-scale study of these five NIST LWC Round2 candidates is presented making COMET to have more information than the other participants about it's design and a comparative against software implementation and a software with an AES in hardware. The number of slices, FFs, LUTs, frequency, and throughput are the main aspects to be presented in the chapter are compared to determine which is the best option, how they were adapted to LWC conditions, the different datapath-size implemented for some of these candidates, plus pros and cons among them. A extensive behavioral benchmarking that guaranteed their correctness.

The contributions of this chapter can be summarize as: We provide the first implementations of five NIST LWC candidates that compliant the GMU LWC interface. All our designs hold the restrictions on resources utilization in the official FPGA benchmark. Our implementation compared with the ones published in the unique existing work [105] are smaller as we present implementations with small datapaths 32 bits and 8 bits.

5.1 Authenticated Encryption with Associated Data

Privacy and authentications are requirements to establish secure communication. Privacy means that only authorized entities can understand the message. Authentication guarantees that the entity sending a message is the expected by using a secret key previously agreed, and ensures data integrity distinguishing any change in the message.

Block ciphers are cryptographic primitives used to provide privacy, and when they are used in a mode of operation, they can provide privacy and authentication, as well. NIST has recommended Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) as secure modes for encryption [60]. On the other hand, message authentication codes (MACs) are modes of operations that provide authentication and data integrity. Two widely known MACs are CBC-MAC [61] and PMAC [107].

Authenticated encryption algorithms offer privacy, authentications, and data in-

tegrity. The simple way to construct an AE is by implementing an Online Encryption (OE) mode and a MAC independently.

Sometimes, the messages could include supplemental information that cannot be encrypted but must be authenticated; this supplemental information is called associated data. Some examples of associated data are the header of a network packet, which has to remain as plaintext, but its integrity should be preserved. A typical example is the TCP/IP protocol, where there are several flags that indicate how the rest of the content in the packet should be decoded and presented. In order to provide authentication to these kind of data, an extended algorithm of authenticated encryption was proposed to handle associated data; it is called Authenticated Encryption with Associated Data (AEAD).

In recent years, AEAD algorithms have received particular attention after the call submissions of Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR) and the NIST LWC. According to NIST LWC, an AEAD algorithm is defined by two operations: authenticated encryption and verified decryption. Both operations need to input a public message number $Npub$(nonce), a secret key K, and an associated data AD. Particularly for encryption, the plaintext PT is an input, and the outputs are $Npub$, AD, ciphertext CT, and tag T. In authenticated decryption, the specific inputs are CT and T; the outputs are the PT and the tag verification; a local T' is computed and checked if it is equal to the tag provided in the input, and releases the messages if and only if $T' = T$. For authenticated encryption and authenticated decryption AD, CT, and PT can be zero-length.

5.2 GMU LWC Interface

The Hardware API for Lightweight Cryptography (LWC) was proposed [69] to guarantee the fairness of benchmarking and compatibility among implementations of the same algorithm by different designers. The interface defines two types of data, secret, and public. The key input is the unique secret data; the public data are message blocks, AD blocks, ciphertext blocks, and verification Tag. For the outputs, they are all public, ciphertext, decrypted message, and the tag.

There are three possible sizes for the input/output buses 8, 16, 32 bits. This small sizes-bus allows the implementation of the final AEAD core in low-end FPGA devices

and is compatible with more common processors and micro-controllers used in IoT technology. The input/output data are handle by preprocessing and postprocessing components that defines a communication protocol to communicate with the developed cores. All the signals names related to the key have *sdi* as a prefix; the signals in the public bus have the prefix *pdi*, and the names signals in the output bus use the prefix *do*.

The API has a component called CryptoCore. CryptoCore is the starting point for hardware designers; in this part, the cryptographic implementation is integrated into the LWC API. The block diagram of the LWC core is shown in Figure 5.1. The block Two-Pass FIFO and its input/output signals (denoted as a_i,a_o,b_i,b_o,c_i,c_o,d_i,d_o,e_i,e_o,f_i,f_o), plus *hash* and *do_last* signals are optional as they are used only when the implemented cipher is two passes, i.e., when the AEAD authenticates the AD and the message before encrypts.

The communication protocol organizes the input blocks as segments; the valid segments are shown in Table 5.1. GMU LWC Interface also includes segments for hash functions; however, this functionality is not used for the present implementations.

Encoding	Segment
0001	Associated data (AD)
0100	Message blocks (PT)
0101	Decrypted message (CT)
1000	Tag (T)
1100	Key (K)
1101	Npub
0111	Hash message
1001	Hash value

Table 5.1: Valid segments in LWC API communication protocol.

The communication protocol does not accept all the possible combinations of segments; in general, all the ciphers receive the sequence of the segments as $Npub$, PT, CT as input and gives as output CT, T for encryption. In decryption, the input is $Npub$, CT, T, and the output is PT. For double pass algorithms, it is necessary to decrypt the message before performing the authentication, so the tag T needs to be received before CT. We have done some modifications to allow the LWC interface to

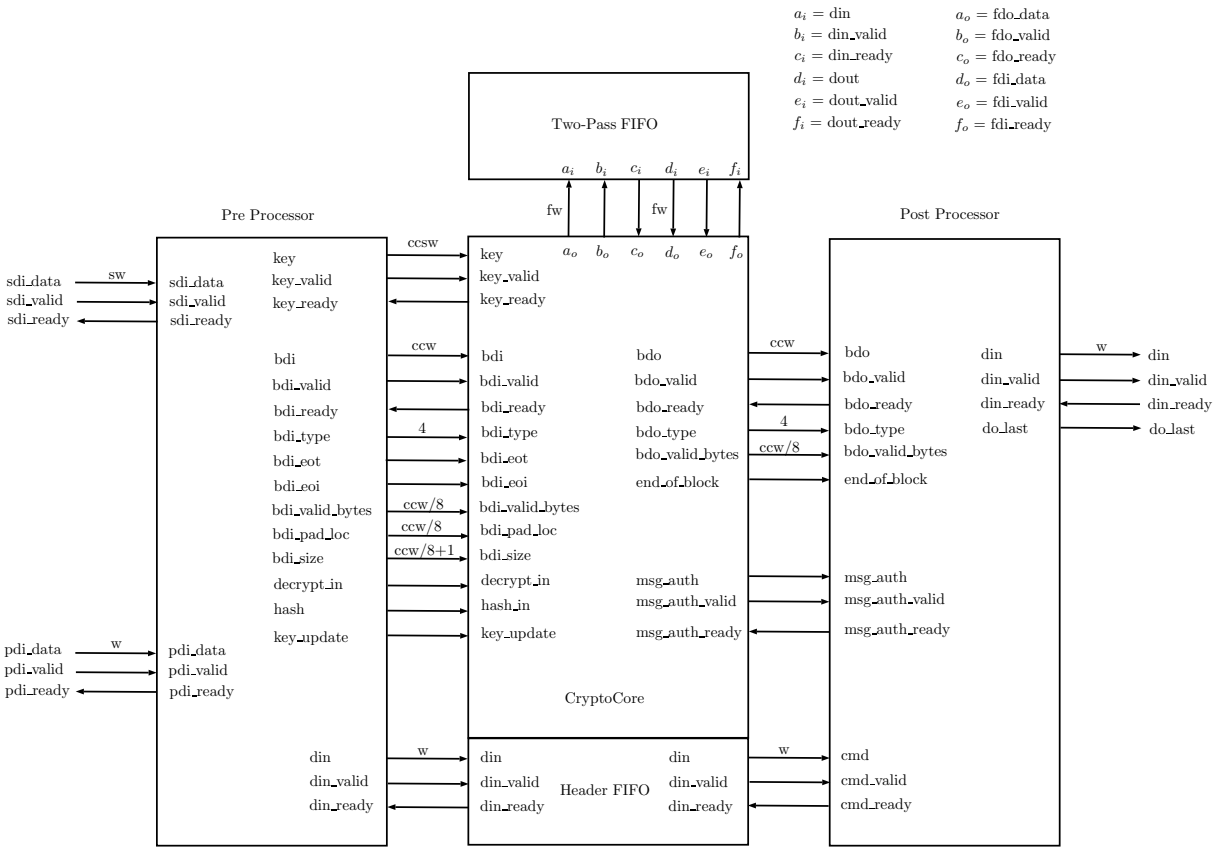

Figure 5.1: Top-level block diagram of LWC core (based on the scheme found at [69]). Here, sw= external key width, w= external data width, ccsw= internal key width and ccw= internal data width.

handle the order of segments as $Npub$, AD, T, CT for decryption.

When the last block of AD or message is incomplete, it is necessary to pad it. For each authenticated cipher or hash function it is necessary to define their own padding rule. To know when a block needs or not padding, the LWC interface provides specific signals bdi_valid_bytes, bdi_pad_loc and bdi_size. Similarly for the output, we need to indicate to the LWC interface if the last block of CT or PT is incomplete and for that it provides the signal *bdo_valid_bytes*.

5.3 Implemented Authenticated Ciphers

This section summarizes each candidate, and then describes and analyzes their respective hardware implementation.

5.3.1 Preliminaries

Along this manuscript, we refer the input message as message blocks or PT blocks. When the input is an encrypted message we use encrypted message blocks or CT blocks and AD blocks for associate data blocks. The term Npub and nonce are equivalent in our descriptions. All the n-bit binary strings are considered as elements of the field $GF(2^n)$ and the addition is defined as a logical XOR denoted as $\oplus$, and the product as polynomial multiplication modulo and irreducible polynomial of degree n. The circular shift bit-wise operation is defined as $>>$ or $<<$ depending of the direction.

5.3.2 Hardware design principles

For the next sections of this manuscript, we refer as CryptoCore to the particular implementation of a candidate algorithm. The register-transfer level implementation design abstraction is used for all the implementations; our designs use only synchronous registers and multiplexers. All the CryptoCore implementations were adapted to the Hardware API LWC. The inputs and outputs are compatible with the API. For each implementation in CryptoCore, a 32-bit datapath is presented, but some also have an 8-bit datapath.

5.3.3 LOTUS and LOCUS

LOTUS and LOCUS are authenticated ciphers that provide Release Unverified Plaintext (RUP) security. Both were presented in the NIST competition in [27] and then published an extended version with detailed security analysis in [28]. LOTUS and LOCUS are a construction based on a tweakable block cipher, in this case, tweGift-64.

The main goal of its design is to provide high-performance capability and suitability for low-end and memory-constrained devices. The high performance can be reached with parallel implementations. The structure followed by LOTUS and LOCUS is based on OTR [87] and OCB [107]. However, two in a row block ciphers are used instead of just one. Such structures allow constructing a parallel authenticated cipher.

For both AEAD, the initialization phase, the Associated Data processing (ADP), and the tag generation (TAG) are the same. Figure 5.2 shows the graphical representation for ADP and TAG. ADP is based on a variant of the hash layer of parallelizable MAC (PMAC), for which it is possible to make the parallelization. TAG aims to provide RUP security, and thus the checksum of all intermediate outputs and the output of each AD block is used to calculate the checksum.

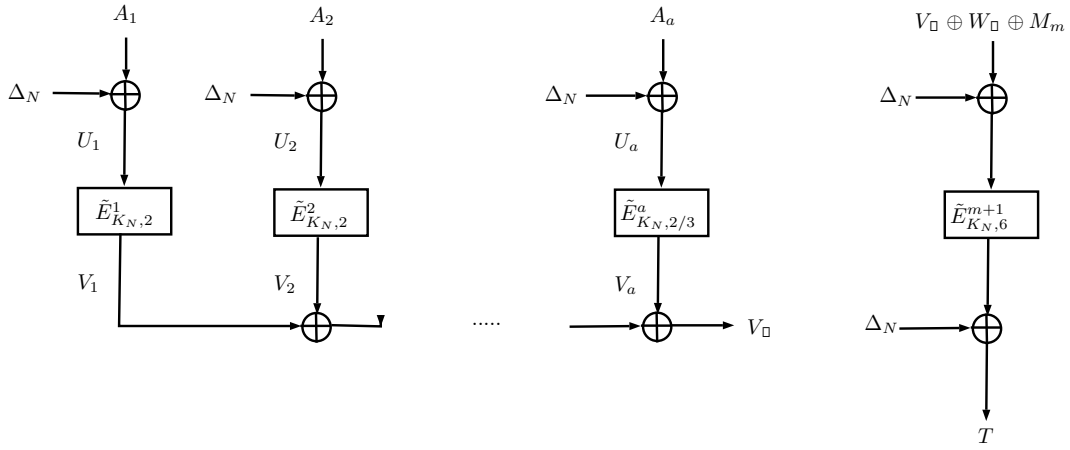

Figure 5.2: Block diagram for associated data processing and Tag generation for LOCUS and LOTUS.

Both LOTUS-AEAD and LOCUS-AEAD in encryption mode defines as inputs an encryption key $K \in \{0,1\}^k$, a nonce $N \in \{0,1\}^k$, an associated data $AD \in \{0,1\}^*$, and a message $M \in \{0,1\}^*$, and outputs the ciphertext $C \in \{0,1\}^{|M|}$, and a tag $T \in \{0,1\}^n$. With fixed $n = 64$, $k = 128$.

The dependent Keys and tweak schedules are an integral part of these AEAD modes. The key and tweak change for each block cipher call. During the initialization

phase, 0^n is ciphered with K to obtain T; K is XORed with N for generating the nonce-dependent key K_N; the nonce-dependent masking key ΔN is computed by cipher Y with K_N. The dependent Keys are computed by α-*multiplication*, $A \cdot \alpha$, where $A \in \mathbb{F}_{2^{128}}$, the multiplication is reduced using the irreducible polynomial $P(x) = x^{128} + x^7 + x^2 + x + 1$. The tweak changes depending on the type of data that is processing and is defined $Twe \in \{0,1\}^\tau$ where $\tau = 4$.

The CrytoCore for LOTUS/LOCUS hardware implementation of 32-bit datapath is shown in Figure 5.4. The main components are a tweGift-64, registers, and multiplexers. The register are Key_mode, Key_alpha, *delta*, $Checksum$, and $regX1$. *delta* stores the nonce-dependent; $Checksum$ stores the checksum of the output for processed AD blocks and the intermediate outputs of the di-block; $regX1$ stores every $M2_i$ block message XORed with *delta*, this is because, in the bus bdi, the message is not more available when the output of the last cipher is. The control is not displayed, but it is implemented by using a Finite State Machine FSM with 11 states.

The key schedule is the most expensive part of the mode, in area terms. Key_mode and Key_alpha are registers of 128-bit length. The first one stores K, since it can be used to encrypt several messages. It needs four clock cycles to be loaded entirely and rotated with 32-bit. Key_alpha implements the dependent keys when there is a new key, and when the key is ready, the register is loaded by 32-bit each clock cycle, as same as Key_mode.

Also, Key_alpha is updated with the value of the nonce-dependent key K_N during the initialization phase; later, it is updated with each dependent Keys, i.e., the result of the α-*multiplication*. This task is done in 1 clock cycles because the result is loaded in parallel. Finally, where there is a new message to process, and there is no new key, the register is updated with the content of Key_mode, and a parallel charge also performs it.

5.3.4 LOTUS

Using a block cipher only in encryption mode, this kind of constructions are known as inverse free. Particularly, LOTUS was inspired by OTR [87]; data is parsed into 2n-bit di-blocks. Figure 5.3 presents the block diagram of LOTUS mode, similar to a two-round Feistel, but with two successive block ciphers in both layers.

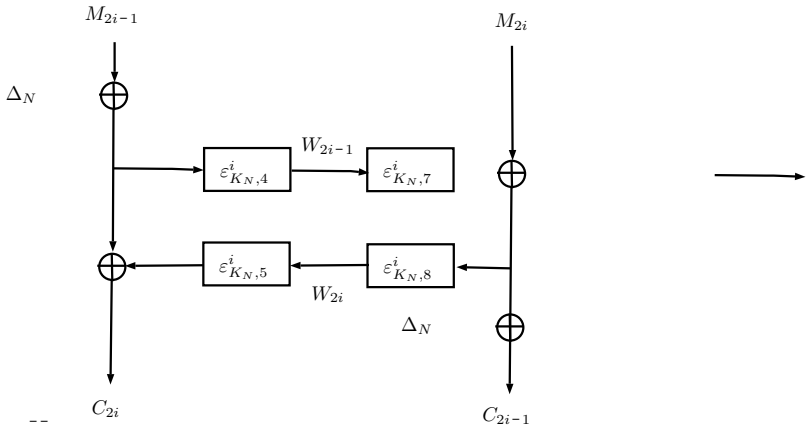

Figure 5.3: Block Diagram of LOTUS mode for encryption.

The tweakable block cipher is instantiated four times for each two message blocks. A new dependent Key is calculated for di-block, i.e., the same key for four block ciphers. However, the tweak changes depending on the layer; the upper layer tweak is 0100, and for the bottom layer, 1101. For the last di-blocks, the tweaks are 1100 and 1101, respectively.

5.3.5 LOCUS

LOCUS was published along with LOTUS in [27] and [25]. It also uses tweGift-64 as an underlying block cipher, but LOCUS requires the inverse function of tweGift-64. Figure 5.5 presents the block diagram of LOCUS mode. In general, LOCUS in based on OCB.

The message is parsed into n-bit blocks. For each block, a mask with the nonce-dependent is applying, then it is encrypted by two successive tweGift-64 and masking again. Similar to LOTUS, the intermediate outputs are used for calculating the checksum. For the last message block, the input is the message length, and XORed the output with the final message block. The key is updated by $\alpha - multiplication$ before each block processing and tweak=4 and tweak= 5 for non-final and final blocks,

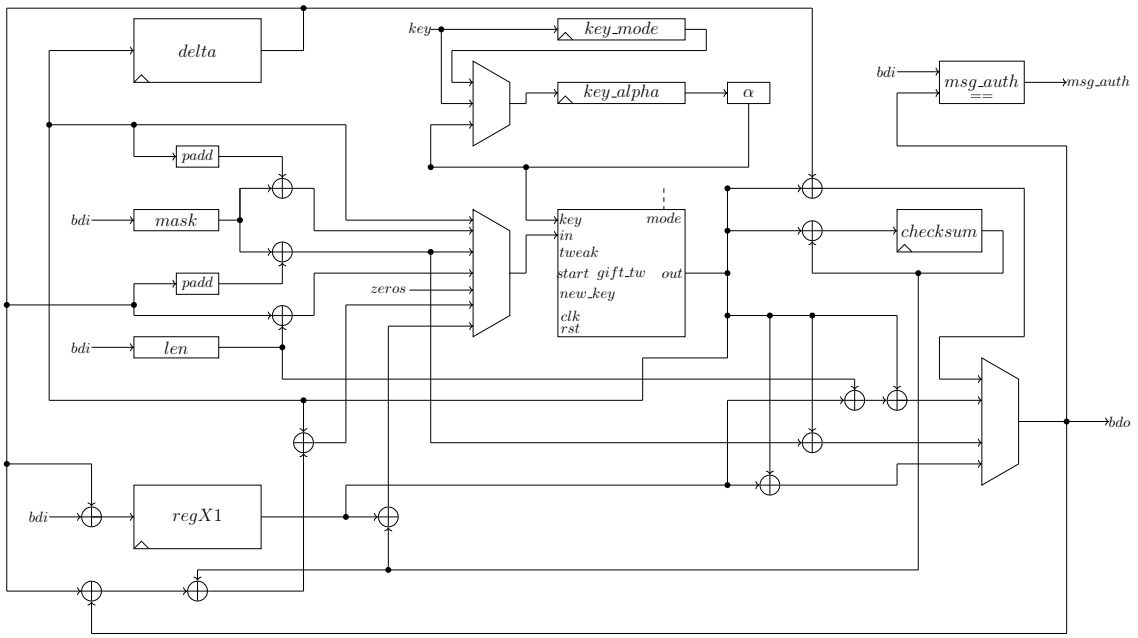

Figure 5.4: LOTUS/LOCUS hardware architecture.

respectively.

Brief description of tweGift-64

Gift is a lightweight block cipher introduced in [10]. It is a Substitution-Permutation-Network (SPN) with the option to handle 64-bit or 128-bit as block length. The substitution layer is performed using a 4-bit S-box, the permutation layer is bit-wise, and a key addition phase is performed at the end of each round. The key schedule is linear, i.e., it is based on shift registers. The tweaked version of Gift called tweGift was presented in [25], where a 4-bit tweak is injected in Gift and AES in order to enhance the output space of the ciphers paying a small cost.

The datapath designed for tweGift-64 is shown in Figure 5.6. A serial implementation was done with a 32-bit datapath to make the tweGift-64 core cheap in area. However, although the design is focused toward low-end devices, we add some extra registers that allow efficient processing of the tweGift-64 core. The core allows simultaneous entry of the key and the message.

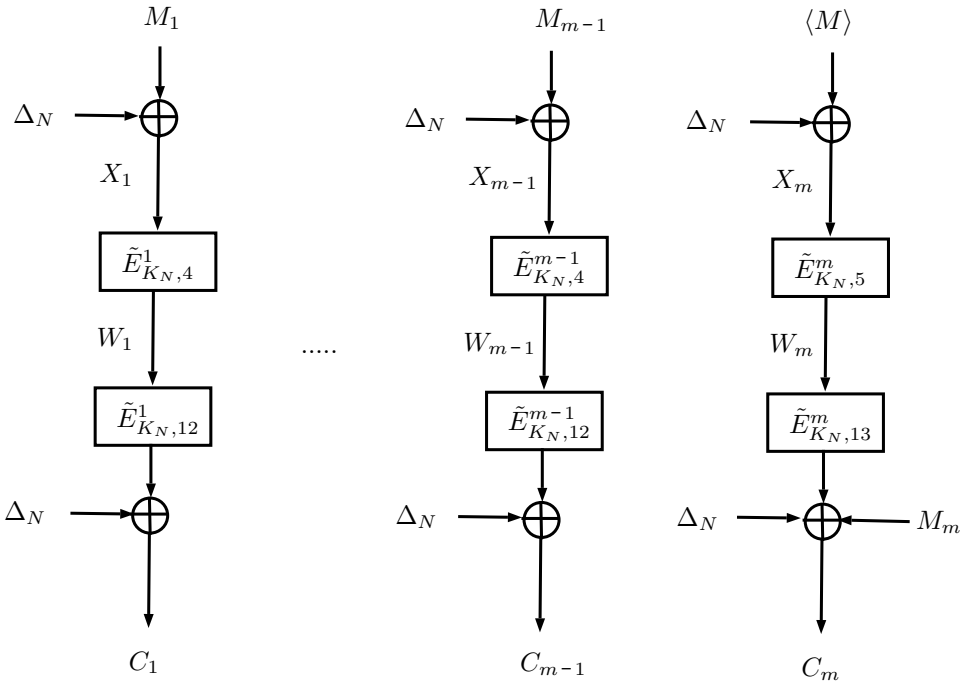

Figure 5.5: Block Diagram of LOCUS mode for encryption.

For encryption, the 32-bit input is divided in chops of 4-bit, an S-box is applied to each chop. Its output is loaded into the status register of 64-bit implemented as two registers of 32-bit in a row. Then, when both registers are loaded, the control commands parallel loading of the value resulting from the permutation; then, the information is shifted, and the output of the register is XORed with the corresponding round key. This process is done for the 28 rounds stated in the cipher algorithm.

The core has two inputs to calculate the round key, a *key* 64-bit, and a *tweak* 4-bit. The input *key* is stored in the Key_state register. The *round_key* is extracted from the 32 least significant bits of the K. In particular, our design has a copy of these 32 bits; it allows that after the first block output, the core can admit a new M and K.

The tweak is connected to *tweExp* component, and its output is stored in a shift register of 16-bit length. *tweExp* performs the function of $tweak \in \{0,1\}^\tau \rightarrow tweak \in$

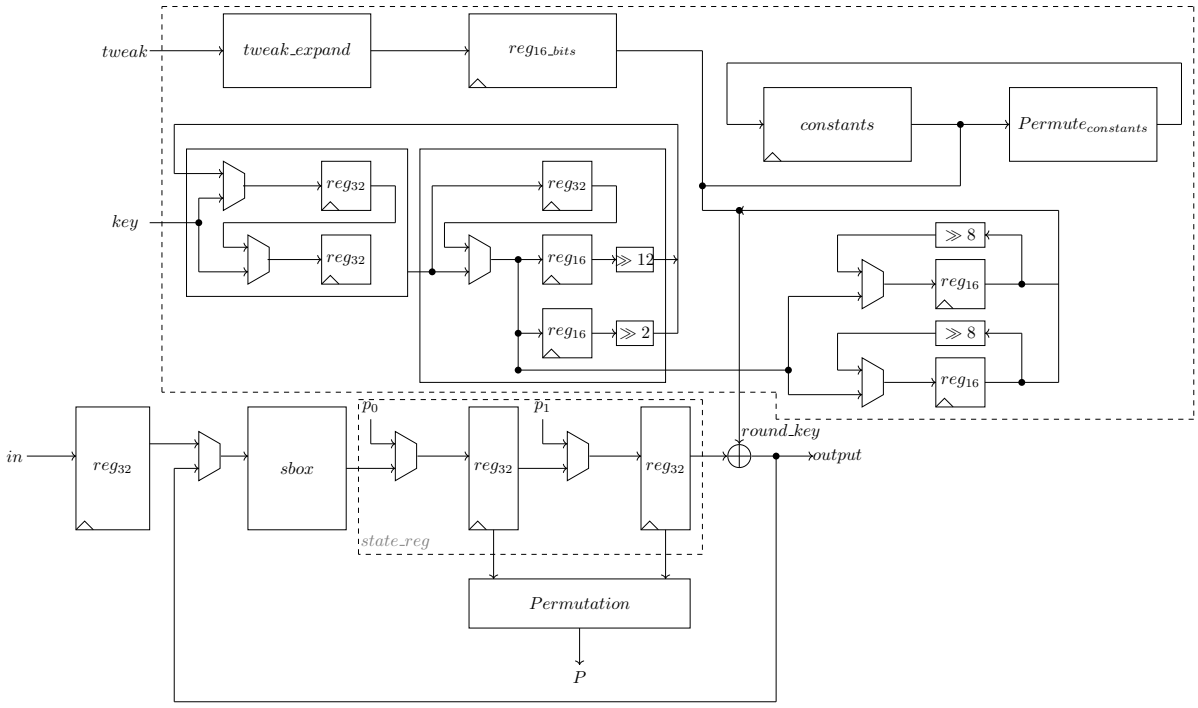

Figure 5.6: tweGift-64 design implementation, for a 32-bit datapath.

$\{0,1\}^{16}$ and tweak$= x_3, x_2, x_1, x_0$ using a linear code described below:

$$S = (x_3 \oplus x_2 \oplus x_1 \oplus x_0) \tag{5.1}$$

$$\begin{aligned} TweExp = (&S \oplus x_3, S \oplus x_2, S \oplus x_1, S \oplus x_0, x_3, x_2, x_1, x_0) \\ &||(S \oplus x_3, S \oplus x_2, S \oplus x_1, S \oplus x_0, x_3, x_2, x_1, x_0), \end{aligned} \tag{5.2}$$

Finally, Gift algorithm uses round constants calculated by an affine LFSR.

5.3.6 ESTATE

ESTATE is a deterministic authenticated encryption (Mac-then-encrypt AE mode) constructed using tweGift-128 and tweAES-128 as underling tweakable block ciphers. It was presented in the NIST competition in the specification document [26] and then published with security proofs in [29]. ESTATE follows the design of Sundae cipher [8]. It combines the MAC algorithm $FCBC^{\star}$ based on classic mode cipher

block chaining (CBC) and encryption algorithm based on output feedback (OFB). The authentication tag T is generated by FOCB receiving as input the AD and M blocks; then, the generated tag is used as IV of OFB mode to encrypt the message. T depends on all the message blocks; we have to go through all message blocks to authenticate the message and store each message block in external memory for the encryption processing. The external memory is implemented as FIFO by the LWC interface, and it is external to the CryptoCore design; hence it does not add additional cost in area.

It is important to note the simplicity of the ESTATE algorithm, all the tweakable block ciphers calls use the same key, and it does not include multiplication by 2 or 2^2 as Sundae, that saves two inputs to the multiplexer in the input of the tweakable block cipher. The tweak is a 4-bit value as in LOCUS and LOTUS. When the nonce is processed and there are no message and AD blocks, tweak= 8; otherwise, tweak= 1. For the AD and message blocks tweak= 0, except for the last blocks. Besides the padding rule, if the block is complete tweak= 2, otherwise tweak= 3. For the last message block, tweak= 4 if it is complete or tweak= 0 if it is not. For the special case, when the message is empty, the tweak to process the last message change to tweak= 4 when it is complete and tweak= 5 when it is incomplete. Figure 5.7 illustrates the operations performed by ESTATE when there are AD blocks and M blocks.

sESTATE is a variant of ESTATE that reduces the latency of the mode; it uses a round-reduced variant of the tweakable block cipher to process the AD blocks except the last. In this study, we only implement ESTATE since sESTATE uses almost the same area, and latency reduction is not part of the scope of this chapter, where we examine the area of different NIST LWC Round2 candidates.

We implement ESTATE using tweAES-128 and tweGift-128. We present two architectures using 32-bit and 8-bit datapaths for both underlying tweakable block ciphers. In Figure 5.8 we show the implemented architecture for ESTATE.

In this case, the tweAES-128 block cipher was implemented with 32-bit datapath. Because its output must be feedbacked, a FIFO to store the output is needed (labeled as $FIFO_{AES}$). The padding layer is implemented using four multiplexers with three 8-bit inputs, where the first one is the input value bdi from LWC interface, the second one is the padding byte 01, and the last one is byte 00. When all the input values in bdi are valid, the four multiplexers select the first value. For instance, if only the

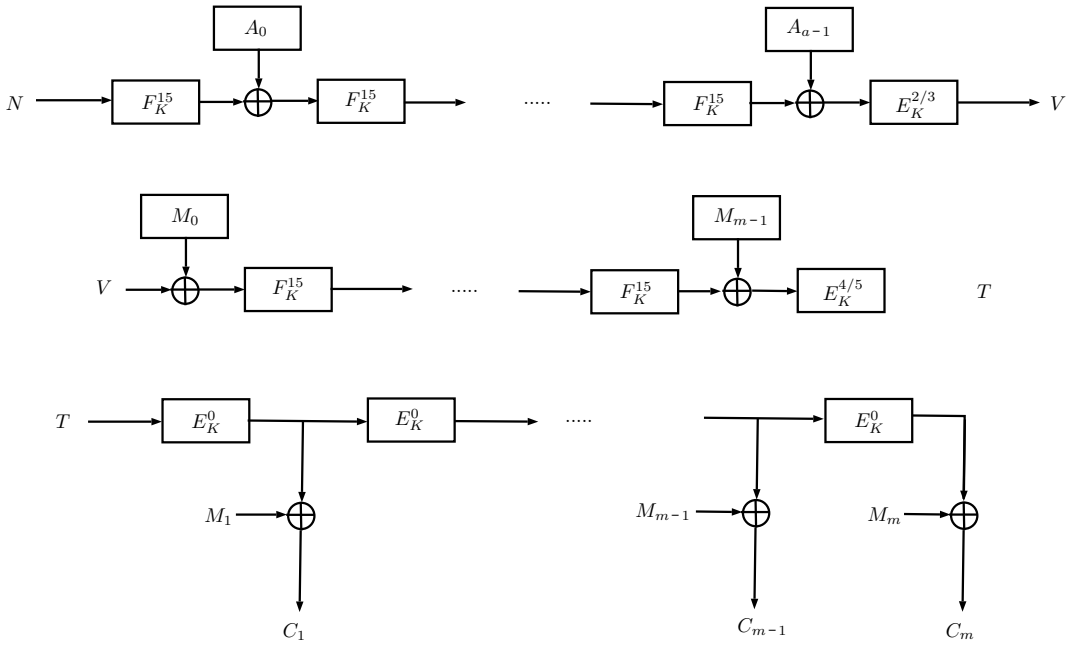

Figure 5.7: ESTATE Deterministic Authenticated Cipher.

first byte in bdi is valid, only the first multiplexer selects its first input, the second selects the padding byte 01, and the other two selects 00. After padding, the inputs are XORed to the feedback from tweAES-128. As the authentication tag is computed before start to encrypt, the component $FIFO_{tg}$ stores the tag and gives it as output after all the encrypted message is computed.

Changes for Decryption: For authenticated decryption, it is necessary to decrypt the message first and then compute the authentication tag. We modify the order of segments to the following order: $Npub$, AD, T, PT/CT. For decryption, the first step is to process AD blocks and store the value in $FIFO_{AES}$ in the register $estateR$. Just after the last AD block is loaded, the tag for verification is loaded and stored in $FIFO_{tg}$. With the tag stored in $FIFO_{tg}$, the message inputs from bdi port are decrypted, and the output to the bdo port is taken as valid decrypted message, while in parallel, all the outputs blocks are also stored in the external FIFO and later they are used to compute the authentication tag. The verification process is done by the component $Verify_{tg}$, where such component is a comparator. It receives as inputs the tag store in $FIFO_{tg}$ and the output from tweAES-128. The control unit is a finite state machine. It generates all the control bits needed by the components in

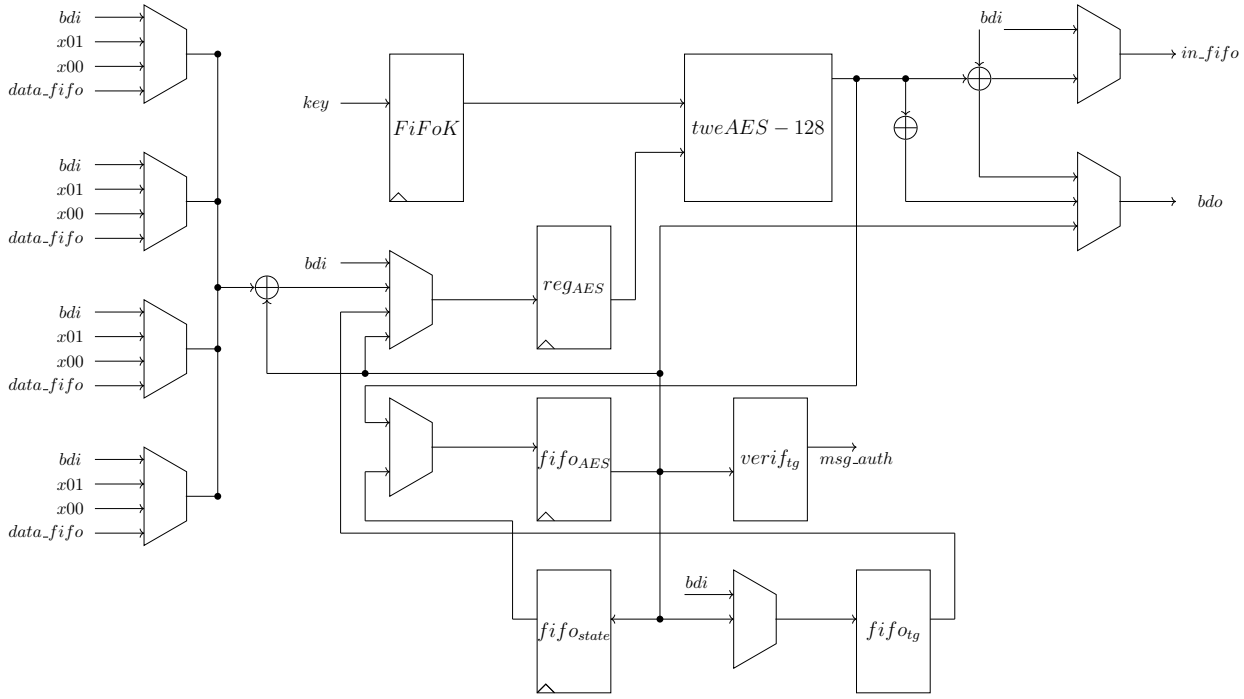

Figure 5.8: Architecture for ESTATE.

the architecture.

The four implementations of ESTATE reported in section use the same architecture, and the only change is the AES core, where it could be tweAES-128 with 8-bit datapath or tweGift-128 either with 8-bit or 32-bit datapath.

Brief description of AES

The Advanced Encryption Standard (AES) [38] is the standard for encryption defined by NIST in 2001. Its original name is Rijndael. It is a SPN computed in rounds, where each round includes four transformations: SubstitutionBytes (S-box), ShiftRows, MixColumns, and AddRoundKey. A key schedule function generates the round keys. ESTATE used the tweaked version of AES with a key length of 128-bit, so then rounds are executed; hence ten rounds keys are also computed. Our implementation of tweAES-128 with 8-bit datapath is based on the implementation of AES presented

in [94]. Such design needs a permuted input, and it gives a permuted output that was done to save some multiplexers as the design was implemented for ASIC technology.

FPGAs multiplexes, from two to four inputs, can be implemented using only one LUT per byte, so we modified the original architecture to avoid the permuted input and output; this change in FPGAs is for free. For the 32-bit datapath architecture, our design is based on the architecture presented in [109], which uses Tboxes to avoid the matrix multiplication of MixColums, by combining SubstitutionBytes and MixColumns in a large table. As we attempt to avoid BRAMs, we compute TBoxes using four Sboxes (implemented using LUTs) and the matrix multiplication for MixColumns.

Below we explain the latency of our implementations of tweAES and tweGift-128. All the implementations computes the round keys on-the-fly and have registers to store the key or the actual value in the round. Such registers are called key state and state, respectively.

- tweAES-8: The version of tweAES with 8-bit datapath uses eight clock cycles to compute the AddRoundKey and SubstitutionBytes, ShiftRows takes one clock cycle, and finally MixColums is performed in four clock cycles. So, the latency per round is 21 clock cycles, and ten rounds take 210 clock cycles and 16 additional clock cycles to outputs the encrypted block, given a total latency of 226 clock cycles. This architecture uses only one Sbox. As the last round does not include the MixColumns transformation, to maintain the uniformity of the design, the state register is disabled, avoiding compute the MixColums in the 10-th round.

- tweAES-32: The computation of the round is column-wise, so each round takes four clock cycles as the ShiftRows and MixColumns are performed in parallel, just selecting the correct bytes from the state to compute the MixColums. Four additional clock cycles are used to give the output. So, the total latency or tweAES is 44 clock cycles. This implementation uses four Sboxes.

- tweGift-128: Both implementations 8-bit datapath and 32-bit datapath use the same architecture; the only changes are that for 32 bits it uses 8 Sboxes (4-bit Sbox) while for 8 bits only 2 Sboxes are used. The permutation step takes one clock cycle for both cases and the computation of the round is five clocks cycles and 17 clock cycles for 32-bit datapath an 8-bit datapath, respectively. Gift-128

executes 40 rounds in 32 bits, so the total latency is 204 clock cycles, and 8 bits is 696 clock cycles.

For tweAES the tweak is expanded from four bits to eight bits using the same linear code as in tweGift-64; in the case of tweAES the tweak is injected, making a $\oplus$ with the least significant bit of each byte in the top two rows of the state matrix. The injection of the tweak is performed every two rounds.
For the tweGift-128, the tweak is expanded using the same linear code to a 32-bit value; then, it is XORed to the bits in position $4 + 3$ for $i = 0, ..., 31$ of the state register. Such operation is applied every five rounds.

5.3.7 COMET

COMET was presented in [114], it is an authenticated encryption algorithm that combines Counter encryption mode, and Beetle authenticated cipher [30] using AES as underlying block cipher. Its basic operations are φ and ϱ, where φ is used to update the AES key while ϱ combines the input AD or message blocks with feedback from the AES cipher and generates the ciphertext blocks. $\varrho(K)$ receives as input a 128-bit string and performs multiplication by 2 of the low half part of the input key $K = K^H||K^L$ as $\varrho(K) = K^H||2K^L$, the multiplication is reduced using the irreducible polynomial $P(x) = x^{64} + x^4 + x^3 + x + 1$. The function ϱ takes as input two 128-bit strings and outputs two 128-bit strings, one is taken as the input of the block cipher, and the other is taken as ciphertext if required.

Internally ϱ performs a shuffle of four 32-bit words of the feedback input, first X is split as $X \xrightarrow{4} (X_3, X_2, X_1, X_0)$ and then shuffle as $X' = X'_1||X_0||X2 \ggg 1||X_3$.. The value Xi is XORed to the padded input blocks and fed to the block cipher. The input is truncated to the ciphertext generation's input block length and XORed to the X' value. The key is XORed with 6-bit constant words called control words. The first is $ctrl_{ad} = 000010$, and it is XORed the key just after the nonce's encryption. If the last AD block is incomplete, then the key is XORed with $ctrl_{p\_ad} = 000100$. Before the first message block is processed the key is XORed with the value $ctrl_{pt} = 001000$, and then if the last message block is incomplete, the key is XORed with $ctrl_{p\_pt} = 010000$. Finally, when the tag is computed, the key is XORed with $ctrl_{tag} = 100000$. In Figure 5.9, COMET algorithm is illustrated.

COMET can be instantiated using three different block ciphers: AES, CHAM, and Speck. In subsection, we present the implementation of COMET only with AES.

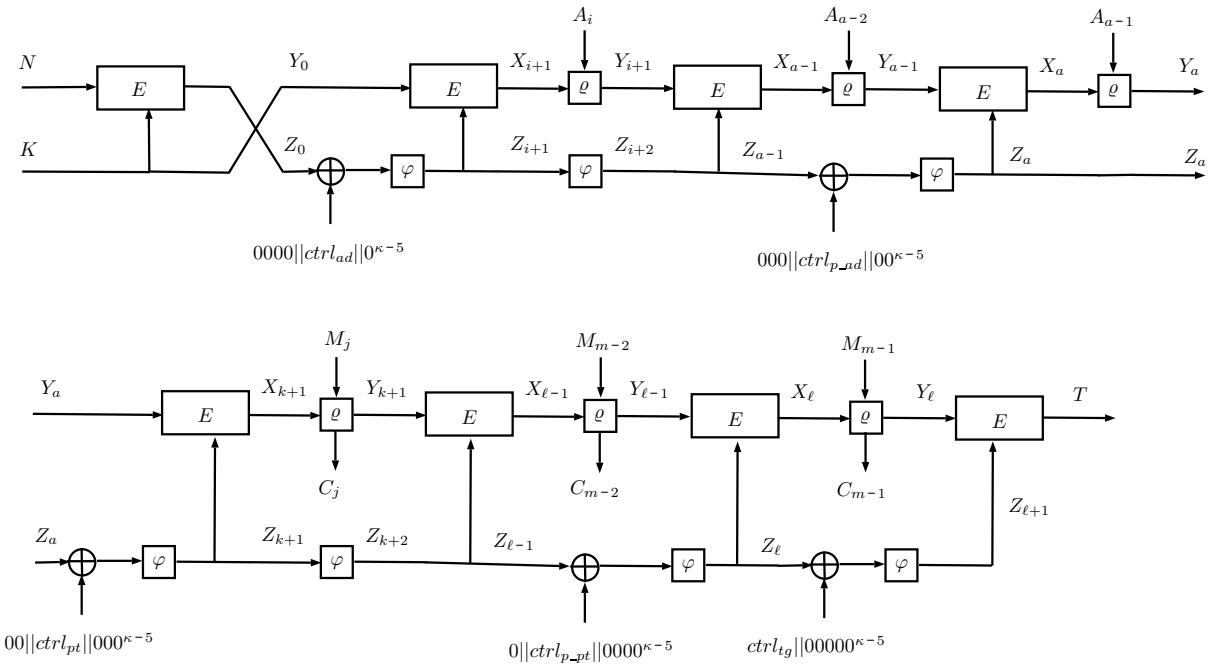

Figure 5.9: Block diagram for COMET.

We implement COMET using the architecture in Figure 5.10 for 8-bit datapath and 32-bit datapath. Two dedicated components were developed to compute shuffle and permute operations; the shuffle component computes and stores X', and stores also its original input X as it is needed to compute the feedback input to the AES. The permute component only computes the operation permute we have explained above; it consists only of a 128-bit register capable of multiplying by 2 its 64 low bits. The output bdo is selected by a multiplexer between tag produced directly by the block cipher, plaintext, or ciphertext, which comes from the function φ. The register mkey stores the secret key loaded from the LWC interface as many messages can be encrypted with the same key, the key is updated only if the interface requests it. The registers Y_reg and Z_reg are used to store the outputs of functions φ and $\varrho(K)$, respectively. The register adptct stores the data input, and the register ptctr allows to compute the ciphertext or plaintext. Finally, the authentication is done by the component is_{auth} that is only a word-wise comparator; the word size can be 8 or 32 depending on the size of the datapath. The control signals are generated using a finite state machine.

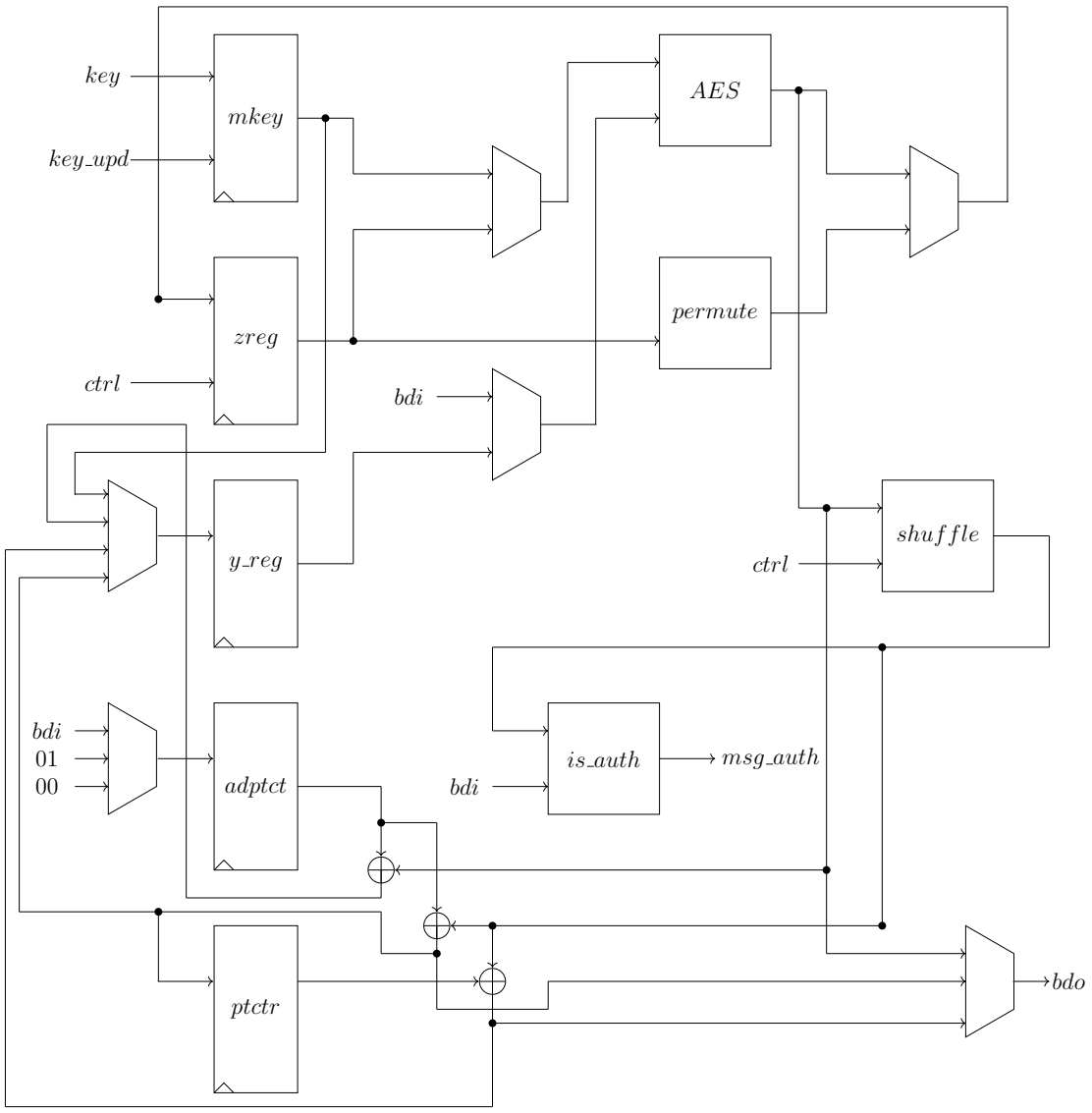

Figure 5.10: COMET architecture for 8-bit and 32-bit.

The data flow for encryption and decryption is the same; the order of the processing of the blocks is the following: $Npub$, AD, PT/CT and T.

5.3.8 Oribatida

Oribatida is an authenticated cipher based on a sponge function, and it uses a public permutation simP based on block cipher Simon [11]. There are two Oribatida versions, one with a permutation size of 192-bit and the other one with 256-bit [16]. In this manuscript, we implement only the 256-bit size version; it is denoted as Oribatida-256. Oribatida-256 manages a 128-bit message (M) and AD (A) blocks. The key and nonce size are also of 128 bits. SimP performs 34 rounds twice (P') or four times (P) depending on which type of data is processing.

In Figure 5.11, the Oribatida mode of operation is shown, where the first chart a) shows the nonce/AD stage. First, the nonce (N) and the key (K) are concatenated and used as the starting values to generate the seed that helps to process the received AD, i.e., U_1 and V_1. Then, there are a stages to process the respective a blocks A_i of 128-bit that form A, where the first $a-1$ are computed with P' and the last one is computed with P. Every U_i output coming from P' or P is XORed with the respective A_i.

The last block A_a is padded if needed before being processed by P. For the second chart b), M is encrypted. To obtain the ciphertext C, each 128-bit block of M is XORed with U_{a+j} and V_{j-1}, i.e. $X_{a+j} = M_j \oplus U_{a+j} \oplus V_{j-1}$ (just the 64 LSB of V_{j-1}) and then each X_{a+j} is used as input for P together with $Y_{a+j} = V_{a-j}$. The last block M_m is padded before being processed by P and the C_m is just of the size of M_m. With the last P for M, the tag T is computed by extracting the MSB places of the output of P. An additional note should be done regarding to the domains d_N, d_A, and d_E, where each one is XORed against V_0, V_a and V_{a+m}, respectively.

These domains help to indicate if A or M is padded or not. Regarding the pad for the A_a and M_m, the remaining empty bytes are filled with the sequence with $0x80$.... Also, in Figure 5.11 observe that each block of the ciphertext is masked using the capacity part of the permutation, this allows Oribatida to offers RUP security.

Now, to be compliant with the LWC contest specifications, our implementation of Oribatida-256 supports any M/A size. That means, A and M are conveyed in a packet that can be authenticated and encrypted regardless of their size or if they are not included in that packet. Fig. 5.12 describes the implemented architecture for Oribatida-256; a sequential architecture was adopted to save area. Due to the nature of this architecture, the current input block must be fully processed before processing the next one. The blocks $(N||K)$, A_a, and M_m are XORed with a domain of 4-bit. The domain is chosen depending on if A or M should be padded or not. Special

Figure 5.11: Encryption version of Oribatida AEAD Algorithm (scheme based on the found at [16]).

attention deserves the decryption operation when C_m is processed, because here M_m (plus pad) obtained from C_m is used as a new frame to obtain the respective T' with an encryption operation, otherwise T' will not be computed correctly.

The input and output of the LWC interface work with 32-bit inputs. We implement a component to pre-process the input blocks labeled as $acc/pad/tag$ for the A, M, and T; meanwhile, acc_k_128 receives the four 32-bit key frames and constructs K from these frames. The pad is XORed to A_a and M_m if necessary. On the other hand, the output blocks are split into 32-bit blocks and sent to LWC interface by the component $split32$.

The block simP-n computes the permutation-based on Simon, where it executes P (for n=4) and P' (for n=2) operations, it receives X and Y in a concatenated way X——Y as input and generates U——V as output. Details about SimP-n can be checked at [16]. The architecture uses three multiplexer components, two to select the

input X——Y to simP-n, and the other selects the correct output, i.e., which value is sent to the component *split*32. The verification tag is done using a 128-bit comparator with two inputs, one comes from *acc/pad/tag* that contains the verification tag, and the second input comes from the output U of the simP-n. The control unit is a finite state machine; it generates the signals to follow the correct operation flow of the Oribatida algorithm and generates the values of the domain depending on if there are of not A and M blocks or if the last block of them is complete or not.

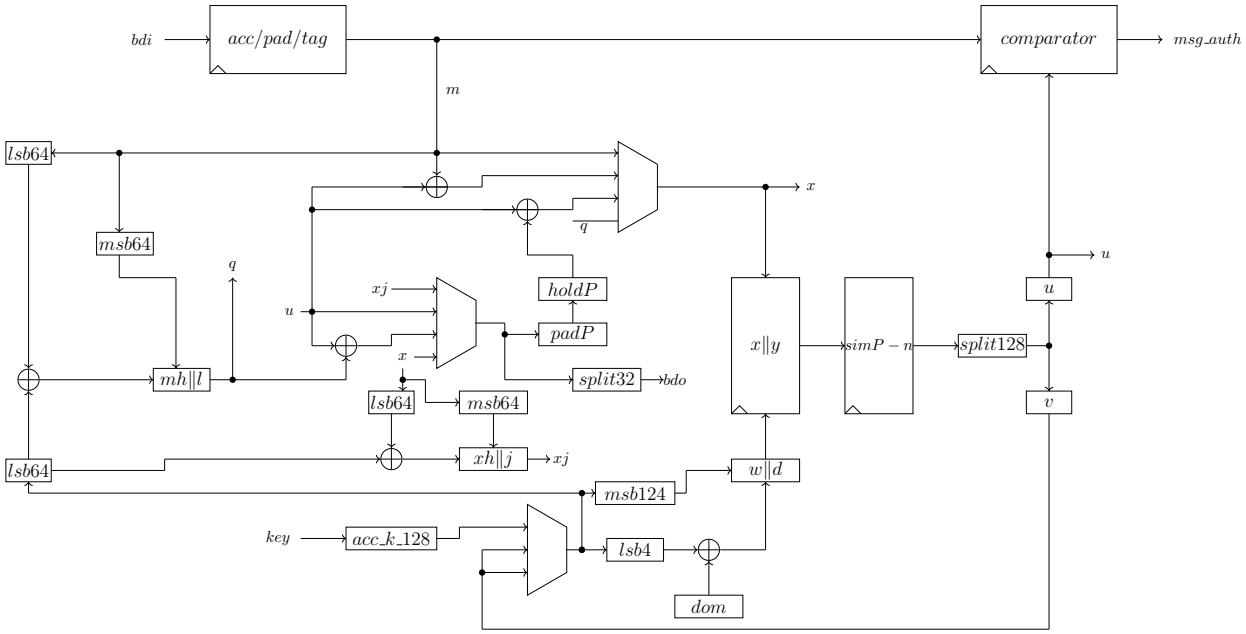

Figure 5.12: LWC architecture for Oribatida AEAD.

5.4 Results

In this section, we present the result of the implementation process of the developed architectures. We use Xilinx Vivado version 2020.1 with the target FPGA Artix 7 xc7a12tcsg325-3, with 8,000 LUTs – 16,000 FFs – 40 18Kbit BRAMs – 40 DSPs – 150 I/O. All the utilized resources and times were obtained after place and route. In Table 5.2, we show the area in LUTs, Flip Flops (FF), and slices; we add a column for the number of clock cycles used to produce each block of ciphertext as we use

it to compute the throughput. For each implemented AEAD, its datapath size is appended to its name.

Mode	LUT	FF	Slices	Freq (MHz)	Clock cycles per block	Throughput E/D	TPA (Mbps/LUT)
LOCUS_32	1846	1005	521	166.04	114	93.21	0.050
LOTUS_32	1525	908	454	132.45	114	74.36	0.049
ESTATE-AES_32	1359	733	420	202.02	88	293.85	0.216
ESTATE-AES_8	797	416	228	227.27	552	57.21	0.072
ESTATE-Gift_32	1055	869	324	233.97	408	73.40	0.070
ESTATE-Gift_8	821	558	248	259.74	1392	23.88	0.029
COMET_8	1052	1031	346	190.33	297	92.47	0.088
COMET_32	1737	1551	565	196.85	70	427.40	0.246
Oribatida-256_256	1432	1319	465	246.24	137	230.06	0.161

Table 5.2: Utilization of resources, throughput and TPA for implemented architectures on the xc7a12tcsg325-3 FPGA.

Table 5.2 has a comparative abstract of the implementation results in terms of utilization resources, throughput, and TPA. The utilization resources are in terms of LUT, FFs, and slices. As these implementations aim to be feasibly implemented in restricted devices, the best results are using fewer resources. The Throughput metric is the rate of process one complete block message and is calculated by $LUT/latency$. Finally, TPA is a metric very useful to evaluated the performance/area tradeoff, it is computed by Throughput/Area.

The results in Table 5.2 show that COMET_32 is the fastest flowing by ESTATE-AES_32. In comparing these two implementations, their frequency is very closed as both use the same AES implementation. The throughput of COMET_32 is better because it uses fewer clock cycles per block than ESTATE-AES_32, 70 vs. 88. ESTATE-AES_32 is a double pass algorithm so that it uses two calls to AES per input block, and it offers RUP security; hence it is more secure than COMET_32. In terms of utilized resources, COMET_32 is more significant as it needs more additional registers than ESTATE-AES_32.

The biggest design is LOCUS_32; despite it uses tweGift-64 as a core, the large

multiplexers it needs for input and key of tweGift-64 make the area increase. The same case is with LOTUS_32 as it also needs large multiplexers to feed tweGift-64. Comparing with ESTATE-Gift_32, which uses Gift-128, LOCUS_32 and LOTUS_32 are both more significant than it, the main reason is that the definition of ESTATE is straightforward as it uses the same key for all the block cipher calls and needs only four inputs to the underlying tweakable block cipher.

For 8-bit datapath architectures, ESTATE-AES_8 is the smallest-even than ESTATE-Gift_8, in the number of LUTs and FFs. Regarding the speed, ESTATE-Gift_8 is faster, but as it needs more clock cycles per block, its throughput is lower than ESTATE-AES_8, which is reflected in better TPA for ESTATE-AES_8. Particularly, it is important to note an incrementation in area around 70% for ESTATE-AES between the 8-bit and 32-bit datapath architecture; also, the throughput increased is more than 500%. In ESTATE-Gift, the increase in area between the architecture of 8-bit datapath and the one with 32-bit datapath is around 28%, and the increase in throughput is 307%. Oribatida-256 with permutation simP takes 137 clock cycles, which is much more than AES implemented with 32-bit datapath almost like Gift128. Oribatida-256 is smaller than LOCUS_32, LOTUS_32, and COMET_32, in the number of LUTs but smaller in the number of FFs only for the last one.

The best in TPA is COMET_32 as its throughput is the best, its area is not too much bigger than ESTATE-AES_32. The worst in TPA is ESTATE-Gift_8 as it requires 1392 clock cycles per block while ESTATE-AES_8 requires 552 clock cycles.

In Table 5.3, we show the real overhead introduced by the LWC interface, i.e, which amount of the resources corresponds to the implementation of the AE algorithm and which correspond to the LWC interface. To get such results, we implement the entity CryptoCore as the top entity. The overhead is shown both in LUTs and FFs. For all the presented cases, the number of FFs and LUTs are below 2,000, which is one of the conditions for hardware components that the LWC contests suggest to use, even with the inclusion of the overhead of the extra components that LWC uses to complement CryptoCore for each evaluated solution. In summary, the overhead is below 300 LUTs and 250 FFs if we consider the worst cases for each feature (ESTATE-AES_32 with 294 LUTs and ESTATE-Gift_32 with 244 FFs). Along with the LUTs and FFs for the complete design, we put the percentage of utilization of such resources.

We can see that our biggest design LOCUS occupy 23.07% and 6.28% of available LUTs and FFs and the smallest ESTATE-AES_8 9.96% and 2.60% of available

LUTs and FFs. With this information we can see that the available resources to develop an application using any of our designs as a security component is at least around 74%.An example of an application could be a wireless transmitter, using LOCUS to encrypt and authenticated the network packets. Almost 74% of resources are left over to implement the rest of the components to achieve the functionality.

	Complete design		Mode only			
Mode	LWC + Cryptocore		Cryptocore		Overhead	
	LUT/%usage	FF/%usage	LUT	FF	LUT	FF
LOCUS	1846/23.07	1005/6.28	1640	956	195	52
LOTUS	1525/18.77	908/5.67	1327	854	183	52
ESTATE-AES_32	1359/16.99	733/4.58	1065	505	294	228
ESTATE-AES_8	797/9.96	416/2.60	587	344	209	72
ESTATE-Gift_32	1055/13.19	869/5.43	788	625	267	244
ESTATE-Gift_8	821/10.26	558/3.49	535	479	286	79
COMET_8	1052/13.15	1031/6.44	803	951	249	80
COMET_32	1737/21.71	1551/9.69	1462	1451	275	100
Oribatida	1432/17.90	1319/8.24	1248	1172	184	147

Table 5.3: Overhead in resources for complete design (LWC+Cryptocore) compared with Cryptocore alone. %usage is the percentage of utilization of available resources on the xc7a12tcsg325-3 FPGA.

The more significant overhead in FFs is introduced to the ESTATE_32, as ESTATE is a double pass cipher, it needs an extra FIFO to store the plaintext or decrypted message. To communicate with such FIFO, 64 ports should be added to the LWC core; however, there are not enough input/output ports available on the target FPGA xc7a12tcsg325-3 (150); we implement a wrapper to convert 32-bit vectors to 16-bit vectors, that change allowed us to fit the design in the target FPGA but increasing the number of utilized FFs.

Table 4 shows the results obtained in our implementations against those previously reported in the literature. First, we need to clarify that these solutions are different from those included in this manuscript. They were implemented using iterated full path round strategy; in our study, only the implementation for Oribatida-256 follows such strategy. All the ciphers implemented in [20] are single pass, and they

offer security compared with COMET but not as LOCUS, LOTUS, STATE, and Oribatida-256, which provides RUP security as is expected, using a full datapath for AES results in better throughput and more area. For example, our implementation of COMET_32 is almost 1000 LUTs smaller than the one presented in [20] but nearly four times slower. Gift-COFB in [20] is implemented with a 128-bit datapath while our implementation of ESTATE-Gift_32 uses a 32-bit datapath, the reduction in area is almost 50%, and the reduction in throughput is almost eight times as ESTATE is a double pass cipher, i.e., it needs two block cipher calls per message block. As our implementation needs more clock cycles to encrypt one 128-bit block, Gift-128 uses 40 rounds, and our 32-bit approach uses 204 clock cycles, 5 per round, and finally 408 per message block, causing a low throughput.

Despite the difference in the compared architectures, it may help to determine if the solutions here tested to have a better performance against those obtained in [20], by considering that this is a Contest that evaluates if a solution is better than the others. At first sight with the LUTs parameter, the winner in this ranking is ESTATE-AES_8 (ranked as 1, because it is the solution with the lower number of LUTs, 797), followed by ESTATE-Gift_8 and COMET_8. Additionally, the number of LUTs occupied by the solutions presented in [20] are larger than those shown in this manuscript, with the notorious exception of SpoC (ranked as 5 for LUTs, with a value of 1172), that is better than the following solutions here presented: LOCUS, LOTUS, ESTATE-AES_32, COMET_32, and Oribatida-256_256 (ranked as 10, 8, 6, 9 and 7, respectively). The worst case is Schwaemm, with 4313 LUTs.

Now, comparing among all the solutions in terms of throughput, ASCON-AEAD is the best because it has the largest throughput value (1683.2 Mbps), followed by COMET-AES and Gift-COBF (2 and 3, respectively). About the solutions here presented, COMET_32 is the first one with the best ranking (ranked as 5, with a value of 427.40 Mbps), followed by ESTATE-AES_32, COMET-CHAM, and Oribatida-256_256 (ranked as 6, 7, and 8, respectively). The worst case is ESTATE-Gift_8, with a throughput of 23.88 Mbps.

Nonetheless, for a fair comparison among all the presented solutions, it is necessary to consider the TPA that belongs to each candidate. For the sake of clarity, the content of Table 4 is sorted by TPA, from the largest to the lowest value. Then, ASCON-AEAD (0.887) and COMET-AES (0.584) have the best TPA record. However, reviewing the number of LUT components for the last one exceeds 2,000 allowed units (see column 2). From the solutions presented, COMET_32 arises as to the best candi-

date (ranked as 4, with 0.246), followed by ESTATE-AES_32 and Oribatida-256_256. The worst case is ESTATE-Gift_8, with a TPA of 0.029. Additionally, Gift-COFB (0.329) is better than any solution in the group of the candidates evaluated in this manuscript. ESTATE-AES_32 (0.216) and Oribatida (0.161) are better than SpoC (0.132), COMET-CHAM (0.128) and Schwaemm (0.121).

Mode -	**LUTs** -	Ranking LUTs	**Freq** (MHz)	**Throughput** E/D Mbps	Ranking Thr.	TPA Mbps/LUT	Ranking TPA
ASCON-AEAD[105]	1898	11	263.00	1683.2	1	0.887	1
COMET-AES[105]	2753	14	251.00	1606.40	2	0.584	2
Gift-COFB[105]	1932	12	263.00	635.20	3	0.329	3
COMET_32	1737	9	196.85	427.40	5	0.246	4
ESTATE-AES_32	1359	6	202.02	293.85	6	0.216	5
Oribatida-256_256	1432	7	246.24	230.06	8	0.161	6
SpoC[105]	1172	5	268.00	154.50	9	0.132	7
COMET-CHAM[105]	2214	13	201.00	282.70	7	0.128	8
Schwaemm[105]	4313	15	106.00	521.80	4	0.121	9
COMET_8	1052	3	190.33	92.47	10	0.088	10
ESTATE-AES_8	797	1	227.27	57.21	14	0.072	11
ESTATE-Gift_32	1055	4	233.97	73.40	12	0.070	12
LOTUS	1530	8	132.013	74.11	11	0.048	13
LOCUS	1835	10	121.951	68.46	13	0.037	14
ESTATE-Gift_8	821	2	259.74	23.88	15	0.029	15

Table 5.4: Comparison of our LWC implementations regarding to the existing literature.

5.4.1 Discussion of results

By considering the results shown in Table 5.3 and Table 5.4, the designs here presented are competitive when compared to those found at [105]. According to Table 5.3 when the solutions are synthesized in FPGA xc7a12tcsg325-3 (8000 LUTs and 16000 FFs available) with the use of the LWC CryptoCore, when considering the number of LUTs, the biggest solution is LOCUS, with 23.07% of usage, being ESTATE-AES_8 the lowest one with 9.96%. If FFs are considered, Oribatida is the biggest with 8.24%, and ESTATE-AES_8 is the lowest with 2.60%. Then, ESTATE-AES_8 has lower area

usage in general terms, either with LUTs and FFs. Nevertheless, it is interesting to see when the solution and the overhead are analyzed individually, LOCUS has a lower overhead than ESTATE-AES_8 (195 LUTs, 52 FF vs 209 FFs, 72 FFs); meanwhile with Oribatida just the number of LUTs are lower (209 vs. 184). It is important to highlight that all the tested solutions in this manuscript are below 2000 LUTs and 2000 FFs, which is a recommendation to be followed when a new solution is proposed and implemented.

Gift-COFB (third), and and COMET_32 (fifth) are the best options, being the first the best one. If the user needs lower area usage in terms of LUTs regardless of the throughput, ESTATE-AES_8 (first), ESTATE-Gift_8 (second), and COMET_8 (third) are the best choices, notwithstanding ESTATE-Gift_8 has the worst throughput among all the evaluated solutions (0.029). Then, ESTATE-AES_8 and COMET_8 becomes in feasible solutions because they also have an acceptable tradeoff in terms of TPA (0.072 and 0.088, respectively).

5.5 Summary

This chapter is another effort to help in the evaluation of the NIST LWC Contest by implementing our architectures, with some of the existing candidate solutions using the adopted LWC-API. This was done to have a fair comparison between implementations. We have presented the implementation of some authenticated ciphers which offer RUP security, and we show that they can be implemented using similar resources as traditional authenticated ciphers. All the presented solutions fit into the FPGA of reference xc7a12tcsg325-3 (8000 LUTs and 16000 FFs available), being the largest tested solution LOCUS, with 23.07% of LUTs and Schwaemm with 53.91%.

The main goal regarding to the area usage has been covered because each implemented solution has less than 2000 LUTs and 2000 FFs. For that reason, Schwaemm has been discarded as feasible solution. The TPA for ESTATE-AES_32 (0.216) and Oribatida (0.161) (which offers RUP security), plus COMET_32 (0.246) are competitive when compared to those presented in [105]. Nevertheless, ASCON-AEAD (0.887) and Gift-COFB (0.329) have better TPA, where their respective ratio is 1.5188 times in favor of ASCON-AEAD. Taking COMET_32 as reference, the ratio against to ASCON-AEAD is of 2.6960.

One of the goals of chapter was to find solutions that reduce the area usage as low as possible, as shown in the cases of ESTATE-AES_8 (797 LUTs), ESTATE-GIFT_8 (821 LUTs) and COMET_8 (1052 LUTs). Even when ESTATE-GIFT_8 has low LUTs usage, their TPA is the worst among all the evaluated solutions (0.029). Then, ESTATE-AES_8 (0.072) and COMET_8 (0.088) are considered as feasible ones. Nonetheless, if the throughput is not relevant at all, ESTATE-GIFT_8 can be considered as feasible.

The next steps in these efforts are the implementation of additional architectures that help to reduce the area usage by trying the minimization of the TPA, i.e., few area usage with high throughput. Another future task will be the implementation of another set of lightweight solutions to determine the full features that they have (LUTs, slices, FFs, Throughput, among others) to do more accurate comparisons.

Chapter 6

Speedy Block cipher on ARM with Bitslice

This chapter briefly explains a short project developed during a quick research visit to the *Technology Innovation Institute* (TII) in Abu Dhabi, United Arab Emirates. The implementation consists of a hardware-based block cipher on a microcontroller ARM Cortex-M4 with low speed and low performance. The project focuses on using the internal registers on the microcontroller to reduce de memory access to enhance the performance and speed of the cipher. Therefore, the use of the techniques to process each block of data differently than the proposed by the authors of the block cipher.

6.1 Speedy block cipher

Speedy [80] aims to achieve ultra-low latency as a dedicated integrated circuit, tailoring this cryptographic application to maximum hardware speed. The first step was to analyze the logic gates and topology suited for ultra-low latency encryption. This block cipher has different instantiations with different Key and block sizes and varying rounds.

The authors propose using a 6-bit width S-box that aims to 64-bit high-end CPUs, and they choose a less common multiple of 6 and 64 to set a default size of 192 bits to block size and Key size, known as SPEEDY-r-192. Besides, they claim that it can achieve 128-bit security when it iterates over $r = 6$ rounds and achieves complete 192-bit security when $r = 7$ rounds while offering a decent level of protection for almost all applications when $r = 5$.

The first stage of the development and design of this cipher begins with the research of the behavior of the available Logic gates and their physical construction to find the best components to achieve ultra-low latency. Next, they analyze different logic gates, and their structure, *and in table 1, section 2 of the Speedy paper* [80], they publish the results obtained from this experiment. That table shows the number of inputs of each gate labeled as Fan-In, the latency obtained by each gate in picoseconds, and other results. Also, for this experiment, all gates use the library Open Cell Library (OCL) with NanGates of 45nm.

In most hardware implementations, we aim to achieve the fastest possible circuit, establishing a relation between the inputs and outputs. The authors demonstrate two myths regarding the latency of logic circuits.

1. *Myth*: Each CMOS (Complementary Metal Oxide Semiconductor) standard cell has a fixed delay, and it does not matter how many instantiations we have; all have the same latency to a path.

   Truth: The delay of a CMOS cell involves the transition in the time of the input signals, which influence the previous cells and electrical environment.

2. *Myth*: Adding a gate to a circuit's path without any other changes increases the latency.

   Truth: Adding a well-placed buffer or inverter (when applicable) to a path can decrease the overall latency on a capacitive load.

To illustrate the previous two claims, we use figure 6.1 from the paper, which shows on the left side an XOR gate and its output connected to eight *XOR* gates. On the right, they add a buffer between the first *XOR* output and the inputs of the eight *XOR*s. Also, we can see the latency of each component individually and the total latency calculation from the sum of all latencies.

On the figure's left side *(a)*, the total latency is l =29.169073 ps. On the other hand, the right side has a latency of l =18.675571 ps despite the more considerable path depth on the right side.

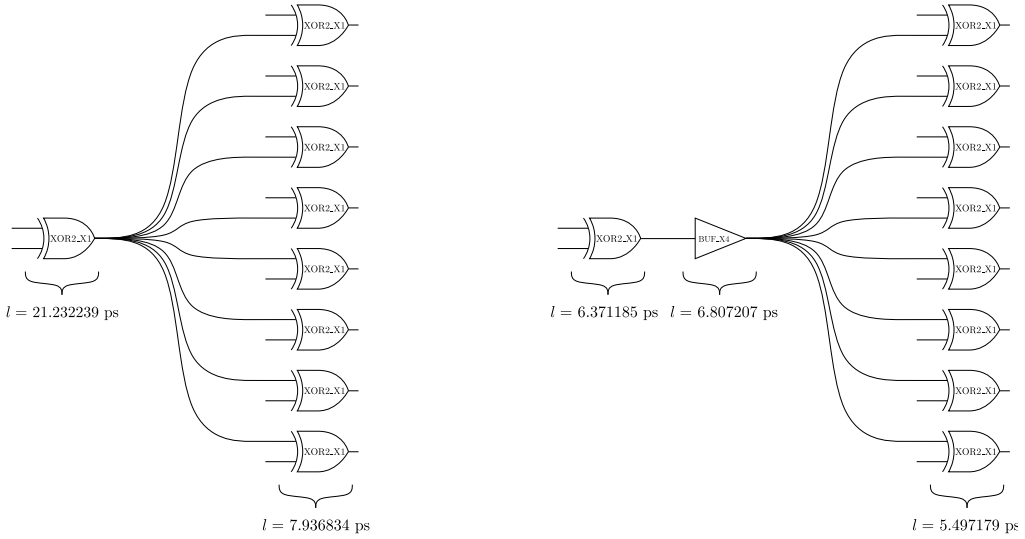

Figure 6.1: Latency of 15nm NanGate (image from Speedy paper [80] page 518, figure 1)

6.1.1 Speedy S-Box

After several tests and research, the authors present their 6-bit S-box in table 6.1. In that table, we see the six bits denoted by the letter x_i with $i = 0, \ldots, 5$ as shown in the figure 6.2, a couple of bits x_0x_1 indicate the row and values from 0 to 3 and the four bits left denote the column, this way the substitution returns the output value for each 6-bits. Also, figure 6.2 shows the S-box as a two-level NAND logic gate. Each input uses a buffer and a NOT gate to slit the input signal x_i to all NAND gates. Therefore we show the following equations to calculate each output bit from the S-box as y_i.

$$y_0 = (x_3 \wedge \neg x_5) \vee (x_3 \wedge x_4 \wedge x_2) \vee (\neg x_3 \wedge x_1 \wedge x_0) \vee (x_5 \wedge x_4 \wedge x_1), \quad (6.1)$$
$$y_1 = (x_5 \wedge x_3 \wedge \neg x_2) \vee (\neg x_5 \wedge x_3 \wedge \neg x_4) \vee (x_5 \wedge x_2 \wedge x_0) \vee (\neg x_3 \wedge \neg x_0 \wedge x_1), \quad (6.2)$$
$$y_2 = (\neg x_3 \wedge x_0 \wedge x_4) \vee (x_3 \wedge x_0 \wedge x_1) \vee (\neg x_3 \wedge \neg x_4 \wedge x_2) \vee (\neg x_0 \wedge \neg x_2 \wedge \neg x_5), \quad (6.3)$$
$$y_3 = (\neg x_0 \wedge x_2 \wedge \neg x_3) \vee (x_0 \wedge x_2 \wedge x_4) \vee (x_0 \wedge \neg x_2 \wedge x_5) \vee (\neg x_0 \wedge x_3 \wedge x_1), \quad (6.4)$$
$$y_4 = (x_0 \wedge \neg x_3) \vee (x_0 \wedge \neg x_4 \wedge \neg x_2) \vee (\neg x_0 \wedge x_4 \wedge x_5) \vee (\neg x_4 \wedge \neg x_2 \wedge x_1), \quad (6.5)$$
$$y_5 = (x_2 \wedge x_5) \vee (\neg x_2 \wedge \neg x_1 \wedge x_4) \vee (x_2 \wedge x_1 \wedge x_0) \vee (\neg x_1 \wedge x_0 \wedge x_3). \quad (6.6)$$

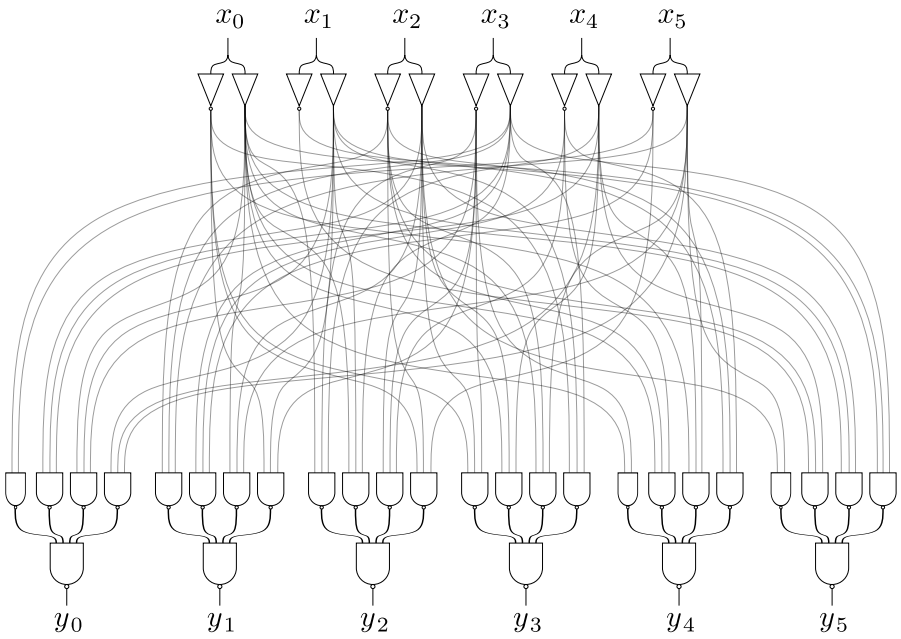

Figure 6.2: Speedy 6-bit Substitution Box architecture with two-level NAND trees and input buffers (image from Speedy paper [80] page 522, figure 2)

6.1.2 Speedy specification

Speedy-r-6ℓ comprises a block size of 6ℓ bits and r rounds; hence, we can see the internal state as a $\ell \times 6$ rectangle array of bits denoted with $x[i, j]$ with i as the rows and j as the bit of the column of the state with $x \leq i < l$ and $0 \leq j < 6$, and all indexes begin with zero, and the authors consider the zero-th bit or word as the most significant one.

For the initialization, the cipher receives a 6ℓ-bit plaintext and initializes the internal state with the same order as the bits are received, i.e., first, fill $x[0, 0]$, then $x[0, 1]$, and so on. Then applies the 1/round functions $\mathcal{R}_r$ (with $0 \leq r < 1/$) to the internal state, and each round function has the following four procedures: (2x)SubBox, (2x)SiftColumns, MixColumns, AddRoundConstant, and AddRoundKey. Consider $x \in \mathbb{F}_2^{\ell \times 6}$ as input and $y \in \mathbb{F}_2^{\ell \times 6}$ as the output with $0 \leq i < \ell$ and $0 \leq j < 6$ such that the procedure gets defined as in the figure 6.3.

- SubBox (SB): apply the 6-bit S-box to each state row, as shown in figure 6.2, with the values from table 6.1.
- SiftColums (SC): Each j-th column gets rotated vertically by j bits.

x_0x_1	$x_2x_3x_4x_5$															
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	08	00	09	03	38	10	29	13	0C	0D	04	07	30	01	20	23
1.	1A	12	18	32	3E	16	2C	36	1C	1D	14	37	34	05	24	27
2.	02	06	0B	0F	33	17	21	15	0A	1B	0E	1F	31	11	25	35
3.	22	26	2A	2E	3A	1E	28	3C	2B	3B	2F	3F	39	19	2D	3D

Table 6.1: Speedy 6-bit Substitution Box

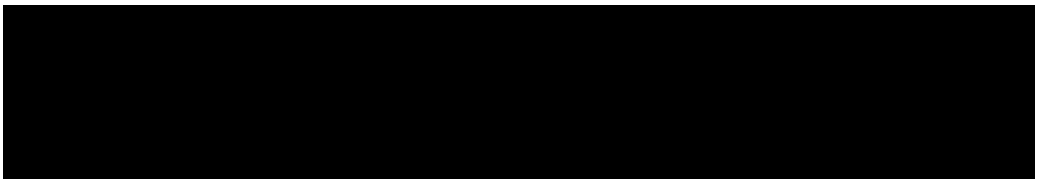

Figure 6.3: Speedy algorithm as block diagram (image from Speedy paper [80] page 524)

- MixColumns (MC): Each state column gets multiplied by a cyclic binary matrix.
- AddRoundKey (A_{k_r}): the round key k_r of 6ℓ-bits is *XOR*ed to the whole state.
- AddRoundConstant (A_{C_r}): Like the PRINCE algorithm from paper [21] all round constants came from the number $\pi - 3 = 0.141519\ldots$ and *table 5* from the Speedy paper [80] shows the first 100×64 bits constants, such that the 6ℓ-bit constant c_0 gets *XOR*ed to the whole state. So they are considering that c_0 is the first constant from the table and so on for all values shown in the table.

6.1.3 Round function

With the procedures from above, this first sound gets defined as

$$\mathcal{R}_r = A_{c_r} \circ MC \circ SC \circ SB \circ SC \circ SB \circ A_{k_r}, \tag{6.7}$$

And in the last round, the linear layer gets omitted and instead has an extra Key addition.

$$\mathcal{R}_{1/-1} = A_{k_{1/}} \circ SB \circ SC \circ SB \circ A_{k_{1/-1}}. \tag{6.8}$$

6.1.4 Key schedule

The cipher uses a 6ℓ-bit secret Key and initializes it as the zeroth round Key (k_0), then applies the permutation **PB** shown in *table 6 page 526* from Speedy paper [80] to compute the following round Key such that the in the next permutation, all the bits change as

$$k_{r+1} = PB(k_r) \text{ with } k_{r+1[i',j']} = k_{r[i,j]}, \tag{6.9}$$

such that

$$(i', j') := P(i, j) \text{ with } (6i' + j') \equiv (\beta \cdot (6i + j) + \gamma) \bmod 6\ell, \tag{6.10}$$

With i' and j' as the quotient of the division $(\beta \cdot (6i + j) + \gamma) \bmod 6\ell$ to 6 respectively. Therefore the parameters β and γ depend on the block length of the cipher and the condition of $gcd(\beta, 6\ell) = 1$.

6.2 Bitslicing

Bit-slicing was introduced by [18] as a replacement for the lookup tables to enhance the speed of the DES algorithm in software. This method consists of bit-slicing the n-bit data to one on n registers, resulting in the possibility of processing multiple blocks in parallel using bitwise instructions. Therefore this technique performs well when using large registers.

For the Speedy software implementation, using an ARM-M4 (please refer to chapter 4 section 4.4) with 16 general-purpose registers of 32-bit, each allows processing the 192-bits of the plaintext on six registers. The first step begins with relocating the plaintext in a bit-slice fashion, as shown in table 6.2. Almost all procedures operate efficiently with this relocation, but the *S-Box* procedure performs with the combination of bitwise operations. Hence all steps on the algorithm perform in parallel over all bits of the plaintext. And the implementation of the procedures *ShiftColumns* and *MixColumns* become more accessible with the use of the *barrel-shift* assembly instruction available in some ARM microcontrollers.

To rearrange the input bits of the block in a bit-slice fashion, the technique used by [83] *SWAPMOVE* presented in the following procedure shows how to perform the

bitslice conversion using bitwise operations.

$$\begin{aligned} &SWAPMPOVE(A, B, M, n) : \\ &\quad T = (B \oplus (A \ll n)) \wedge M \\ &\quad A = A \oplus (T \gg n) \end{aligned} \tag{6.11}$$

	Block 0	*Block* 1	*Block* 2	...	*Block* 29	*Block* 30	*Block* 31
reg_0	b_0^0	b_0^1	b_0^2	...	b_0^{29}	b_0^{30}	b_0^{31}
reg_1	b_1^0	b_1^1	b_1^2	...	b_1^{29}	b_1^{30}	b_1^{31}
reg_2	b_2^0	b_2^1	b_2^2	...	b_2^{29}	b_2^{30}	b_2^{31}
reg_3	b_3^0	b_3^1	b_3^2	...	b_3^{29}	b_3^{30}	b_3^{31}
reg_4	b_4^0	b_4^1	b_4^2	...	b_4^{29}	b_4^{30}	b_4^{31}
reg_5	b_5^0	b_5^1	b_5^2	...	b_5^{29}	b_5^{30}	b_5^{31}

Table 6.2: Speedy bit-slice representation into the six registers as individual bits with b_j^i where i refers to the $i-th$ block and j refers to the $j-th$ bit of the bock i .

Therefore, this block cipher gets stored into five blocks of 6 bits when we use the technique *SWAPMOVE* from the bit 0^{th} to 29^{th}, and we need to move one by one the 30^{th} and 31^{st} bits at a time. The first step consists of fulfilling the registers with 32 bits each from the plaintext. The second step moves complete blocks from other registers to the next register, i.e., the second block b^1 of 6-bits precedes the block b^0 in the same register, but this second step swaps this block with the first 6-bits in the second register, and so on for the following complete blocks in the register, then make the same for the following registers and blocks.

The third step moves the bits from incomplete blocks to a different register to create new complete blocks and move those "empty" bits left in the registers to the right of each. By the way, all empty places become full of blocks without moving any of the blocks 30^{th} and 31^{st}. Finally, in the last step, we have each block in a column way, then we can swap each bit to move each block as a column, as shown in figure 1 page 6 paper [76].

6.2.1 Substitution Box (SB)

It gets implemented by the combination of logical operators to reduce the memory consumption of a lookup table. This focus allows processing the 32 blocks of 6 bits

parallelly thanks to the bitslicing. Each function needed gets performed individually and sequentially but with different equations because the microcontroller does not have a NOT ($\neg$) instruction. Still, it has an *or-not* (ORN) instruction with $a \wedge \neg b$, and the equations from 6.1 to 6.6 need a conversion to use the ORN instruction resulting in the following set of equations.

$$y_0 = x_3 \wedge (\neg x_5 \vee (x_4 \wedge x_2) \vee (x_1 \wedge ((\neg x_3 \wedge x_0) \vee (x_5 \wedge x_4)))) \quad (6.12)$$
$$y_1 = x_5 \wedge ((x_3 \wedge \neg x_2) \vee (x_2 \wedge x_0) \vee (\neg x_5 \wedge x_3 \wedge \neg x_4) \vee (\neg x_3 \wedge \neg x_0 \wedge x_1)) \quad (6.13)$$
$$y_2 = x_0 \wedge ((\neg x_3 \wedge x_4) \vee (x_3 \wedge x_1) \vee (\neg x_3 \wedge \neg x_4 \wedge x_2) \vee (\neg x_0 \wedge \neg x_2 \wedge x_5)) \quad (6.14)$$
$$y_3 = x_2 \wedge ((\neg x_0 \wedge \neg x_3) \vee (x_0 \wedge x_4) \vee (x_0 \wedge \neg x_2 \wedge x_5) \vee (\neg x_0 \wedge x_3 \wedge x_1)) \quad (6.15)$$
$$y_4 = (x_0 \wedge \neg x_3) \vee (\neg x_0 \wedge x_4 \wedge x_5) \vee ((\neg x_4 \wedge \neg x_2) \wedge (x_0 \vee x_1)) \quad (6.16)$$
$$y_5 = x_2 \wedge (x_5 \vee (x_1 \wedge x_0)) \vee (\neg x_1 \wedge ((\neg x_2 \wedge x_4) \vee (x_0 \wedge x_3))) \quad (6.17)$$

6.2.2 Shift Columns (SC)

Due to the bit-slicing, all blocks are now in a column way such that all bits have a representation of a transposition in the row direction. The implementation of this procedure consists of the use of the assembly instruction *ROR*, which consists of the rotation of n bits to the right of a register. Moreover, the instruction performs over the 32 blocks in parallel, and the whole procedure needs only six instructions. The algorithm 17 shows the algorithm in assembly to execute the Shift Columns.

Algorithm 17 ShiftColumns (SC) assembly code.

Require: Data output from SB procedure stored in general purpose registers.
Ensure: Shift Columns process stored in general purpose registers

```
1: "mov r0, r6;"
2: "mov r1, r12, ror #31;"
3: "mov r2, r8, ror #30;"
4: "mov r3, r9, ror #29;"
5: "mov r4, r10, ror #28;"
6: "mov r5, r11, ror #27;"
```

6.2.3 MixColumns (MC)

Like Shift Columns, the processing becomes parallel to all blocks, and each row performs an XOR, as shown in the equation 6.18.

$$y[i] = x[i] \oplus (x[i] \ll a_0) \oplus (x[i] \ll a_1) \oplus (x[i] \ll a_2) \\ \oplus (x[i] \ll a_3) \oplus (x[i] \ll a_4) \oplus (x[i] \ll a_5) \tag{6.18}$$

Due to the modification of the value while the MC execution, the new values are saved on different registers to avoid change from the importance of SC. MC uses the XOR instruction and the barrel shifter simultaneously, and all the procedures use 36 *EOR* (XOR Barrel-Shifter) only.

6.2.4 AddRoundKey (AR) and AddRoundConstant (AC)

The conversion of both Round Keys and Round Constants to bit-slice following the same vision becomes a must for the correct operation of the encryption process. Therefore, this procedure allows for a reduction in advance with only an *XOR* on both ARK and ARC and stores the values in memory. Since the process began, it only accessed memory six times in the encryption.

6.3 Results

The team who implemented the Speedy algorithm in an ARM-M3 [76] used an *ArduinoDUE* (*AT91SAM3X8E*) powered by an ARM Cortex-M3 with a clock speed of 84 MHz; it has 512KB of flash memory and 96KB of RAM. We use an ST Nucleo-144 development board with the characteristics presented in the chapter 4 section 4.5. The board uses an ARM Cortex-M4 *stm32l4a6zg* with a frequency of 80 MHz.

Both implementations do not consider the Key scheduling, assuming that the precomputation of the *Round Keys* and the *Round Constants* stored in memory, as a difference between the traditional ciphers, which encrypt blocks of 128-bits Speedy encrypts 192-bits. Table 6.3 compares the reference C implementation of *Speedy-7-192* and other results from different implementations of block ciphers in cycles per byte (cpb), and all implementations use the ECB encryption mode.

Comparing the Speedy-6-192 against AES-128 and GIFT-128, the 75.2 cpb shows that Speedy with the same security level is 1.6x faster than 120.4 cpb and 1.3x than 104.1 cpb of both ciphers, respectively for the reference code available in [80] for ARM.

Our implementations consist of the same techniques with different focuses. We

write procedures to stay as long as possible in the microcontroller registers and write some functions in combination with C language and assembly.

The functions SBox, MixColumns, and ShiftColumns are from the paper [76] with some modifications taking into account only the code of those functions we used barely. The difference is how we implemented the bitslice process to convert any data from a regular to a bitslice representation; once this conversion finishes, the encryption algorithm begins to run.

The first developed version is the adaptation of the reference code from the authors, and it only has the compiler improvements.

The second implementation uses the ASM language only and is an "unrolled" version of the algorithm. The plaintext is processed once the conversion to bitslice has finished, and the Round Keys needed are stored in memory and loaded as required. With the previous conversion of Round keys and Round constants to bitslice, the algorithm access that data stored in memory.

The third version is a copy of the previous implementation with the difference in the operation "*XOR*" used in the *AddRoundKey* and the *AddRoundConstant* in the algorithm. Here it is possible to previously precompute an *XOR* between the round Key $K[1]$ and round constant $Rc[0]$ to save some clock cycles during the encryption process

$$K[i+1] \oplus Rc[i] \tag{6.19}$$

for $i = \{0, \ldots, nr-1\}$ and nr as the rounds required for the encryption algorithm, i.e., we perform this operation to reduce the memory access when the round Keys and round constants are loaded. Performing this operation can reduce the memory access and the memory needed to store data to store only the plaintext and the keys and reduce the clock cycles required to encrypt one data block.

For the fourth implementation, the implemented code combines C language and Assembly using parameters to send and receive data between functions. The shared data is the Plaintext, Round constants, and the keys needed for the algorithm. We seek to reduce memory access using the debugger tool. We look to stay as much as possible with the microprocessor registers with the required data and only load the data needed to perform the required operations.

During the execution, the program modifies the registers $r0$ to $r3$. In these reg-

isters, the variable addresses get stored there. Therefore, moving the data stored in these locations to other registers is mandatory before the C function ends. Then, when a new process begins, we recover the data from the registers without memory access.

The final implementation is a fusion between the third and fourth implementations with the advantage of executing the algorithm with a dynamic number of rounds.

All implementations do not have any Key scheduling because this function needs to be stored in memory to get some improvements. In addition, the registers in the microprocessor are 32-bit long, so there is no improvement to the regular Key schedule, but a bitslice version has possibilities.

Table 6.3 shows the results from all our implementations, including the reference code and results from work [76]

Implementation	Speed (cpb)	Block size
AES-128 [80]	120.4	128
GIFT-128 [80]	104.1	128
Speedy-7-192 (reference M3 [80])	15,407	192
Speedy-5-192 (ARM M3 [80])	65.7	192
Speedy-6-192 (ARM M3 [80])	75.2	192
Speedy-7-192 (ARM M3 [80])	85.1	192
Speedy-7-192 (reference M4)	358,306 cc	192
Speedy-6-192 (ASM unrolled our)	2562 cc	192
Speedy-6-192(ASM unrolled ARKARC our)	2338 cc	192
Speedy-6-192 (ASM Fun our)	2554 cc	192
Speedy-6-192 (ASM Fun ARKARC our)	2270 cc	192

Table 6.3: Table with the results from [80] and our implementation, we only measure the clock cycles for the encryption process via the microcontroller embedded instructions.

6.3.1 Di erential Attack

On the date we did this project, there was no known attack. However, recently in [23], a differential attack against *Speedy-7-192* successfully broke the security claim

by the authors of [80] that the version mentioned above offers 192 bits of security. Therefore, they introduce the attacks to the authors of Speedy, and now they are awkward about it. Those kinds of attacks are out of the scope of this thesis work. Still, they are critical in cryptography and valuable to verify the security of any new or old proposal.

Source Code

The source code and projects are available in the following links:

https://os5.mycloud.com/action/share/9748f690-e15a-4f42-8aa6-08ff4553fbfe
https://github.com/JoseABernalG/Speedy-ARM4.git

Libraries and functions source files are located in the next address for each project, for example,

*Speedy/Core/Inc/*Functions.h

And the C code is located in:

*Speedy/Core/Src/*Functions.c and main.c

list of all individual projects as presented in this report with the name given in the code:

- Speedy (reference code from [80])
- Speedy ASM Unrolled
- Speedy Unrolled ASM ARKARC
- Speedy ASM Fun
- Speedy ASM Fun ARKARC

And all C codes have commentaries about each function created.

Part III

Public Key Cryptography

Chapter 7

A DSP-based FPGA design and implementation of a fast RNS multiplier

Modular multiplication is a fundamental operation for modern cryptography schemes. It is essential for modular exponentiation in public key crypto-systems based on discrete logarithms and for the RSA encryption scheme. Furthermore, prime field multiplication is an important arithmetic operation for computing elliptic curve scalar multiplications and bilinear pairings.

Several authors have proposed the Residue Number System (RNS) as an alternative for computing modular arithmetic over large integer operands [13, 36, 92, 93, 102]. Based on the Chinese Remainder Theorem (CRT), RNS's main attractiveness is representing a large integer utilizing a set of smaller independent numbers. In this way, one trades the computational cost of a single arithmetic operation over two large operands by calculating independent smaller modular operations that can be computed in parallel. This unique characteristic is especially relevant for hardware implementations, where handling the large operands usually required for cryptographic applications can become problematic. Across the years, RNS arithmetic has been used in hardware implementations of RSA [52, 5, 113, 74, 96, 97], in elliptic-curve based cryptography [6, 17, 3, 54, 111, 112] and also in pairing-based cryptography [135, 32].

Due to its high flexibility and speed capabilities, the target device for our implementation is a Field Programmable Gate Array (FPGA) device. The enhanced Digital Signal Processing (DSP48) slices and memory blocks present in the latest

models of FPGA devices are used to develop an efficient hardware architecture for the field multiplication operation. The architecture obtained is then compared with the existing state-of-the-art solutions to measure the overall efficiency of our FPGA-based solution. The designs presented in this chapter were modeled using VHDL and the Xilinx Vivado tool for synthesis and place-and-route simulations.

Modern FPGA devices come equipped with specialized hard-core units (DSP48) slices. For Xilinx, a DSP48 slice includes a 25×18 multiplier and 27×18, DSP48E1 and DSP48E2[1], respectively; an ALU unit that permits to perform arithmetic and logical operations such as additions, subtractions and Boolean operations over 48-bit operands. The operation output can be stored on an accumulator allowing it to perform the traditional digital signal processing operations $a = a + b \cdot c$. The DSP resources enhance speed and efficiency. DSP48 slices enjoy a faster operating frequency than the one associated with designs using the FPGA fabric of logical components. Moreover, they are designed with a Pipeline for performance and lower power. A configurable number of pipeline stages architecture, from 0 to 3, permits to accommodate several operands simultaneously.

7.1 Our Contributions

This chapter presents two efficient pipeline architectures for Montgomery multipliers based on the Residue Number System (RNS). One salient feature of our hardware design is that it is DSP48 slice-based, using very few Look-up Table fabric resources of the FPGA device, leaving the programmable logic for other tasks; also, the using of DSP48 allows for obtaining a higher frequency clock compared to the one implemented only with LUTs. To enhance the efficiency of the RNS Montgomery Multiplier, two small multipliers with reduction designed specifically for DSP48E1 and DSP48E2 are presented; these are in charge of computing the product of each residue multiplication.

In the context of the RNS arithmetic, we also perform a detailed study comparing the merits of a Montgomery reduction procedure proposed in [74] versus the one adopted in [64]. This paper proposes an improved Montgomery reduction algorithm in [74] to be implemented in hardware DSP oriented. We conclude that the former is better because implementing Jeljeli is more practical since it does not require a base change

[1]https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

or handling the double bases. That is required at the cost of duplicating the initial base and adding three words, and we print both the base and its RNS representation. In addition, the tables and precomputes grow in the same way. Therefore it is possible to reduce the data dependencies and lower the quantity cycles required to execute the algorithm without the need for a control unit since it is possible to take advantage of the pipeline of the DSPs used; in addition, since it is completely designed in DSP, the working frequency is not reduced as much as long as there are not too many instances of them.

The remainder of this chapter is organized as follows. In Section 7.2, we provide the theoretical and technological preliminaries. We discuss the main related works in Section 7.3. Section 7.4 we present an analysis of basic arithmetic modules for implementing the multipliers regarding DPS resources and their interaction. The implementation for both multipliers is presented in Section 7.5. The results are in Section 7.6.

7.2 Preliminaries

7.2.1 Notation

The notation for this chapter is:
w: the size of a word given this design.

$\langle a \rangle_m = a \bmod m,$ where $a_m \in [0, m)$

Base $\mathcal{B}$=$\{m_1, \ldots, m_n\}$, where $gcd(m_i, m_j) = 1$ for $i \neq j$.

Base $\mathcal{B}'$=$\{m'_1, \ldots, m'_n\}$, where $gcd(m'_i, m'_j) = 1$ for $i \neq j$.

$|B|$: size of a set B

$$M = \prod_{i=1}^{n} m_i, \; M' = \prod_{i=1}^{n} m'_i, \text{ where } gcd(M, M') = 1$$

$M_i = M/m_i,\ M_i' = M'/m_i'.$

$\{a\}_B = [\langle a\rangle_{m_1}, \ldots, \langle a\rangle_{m_n}].$

$\{a\}_{B'} = [\langle a\rangle_{m_1'}, \ldots, \langle a\rangle_{m_n'}].$

$\{a\}_{BB'} = [\{a\}_B, \{a\}_{B'}].$

Transpose a^T:

$$\{a\}_B^T = \begin{bmatrix} \langle a\rangle_{m_1} \\ \vdots \\ \langle a\rangle_{m_n} \end{bmatrix}$$

Single word Montgomery multiplication.
$\{a\}_m \bigotimes_M \{b\}_m = \{ab2^{-w}\}_m$

$\{a\}_B \bigotimes_M \{y\}_B = \{ab2^{-w}\}_B$
$= [\{ab2^{-w}\}_{m_1}, \ldots, \{ab2^{-w}\}_{m_n}]$

Algorithm 18 Montgomery Multiplication.

Require: $x, y \in [0, p)$, p with $\gcd(R, p) = 1$ and $R > p$
Ensure: $(v = x \cdot y \cdot R^{-1})_p$

1: $c \leftarrow x \cdot y$
2: $c(-p)^{-1} \bmod R$
3: $r \leftarrow x + q \cdot p$
4: $v \leftarrow r/R$
5: **if** $v \geq p$ **then**
6: $\quad v \leftarrow v - p$
7: **end if**

7.2.2 Montgomery reduction

Modular multiplication is a fundamental operation for public key infrastructure. The efficient implementation of this operation is important since, to establish a shared secret, the following operation has to be performed several times.

$$c = a \cdot b \bmod p \tag{7.1}$$

where p is a prime number of 2^w-bits. In this context, p is usually a large number, the range of w in the order of hundreds to thousands of bits.

The main disadvantage of using Eq. 7.1 is that its implementation involves a division that results in more expensive than simple multiplication. A popular alternative is the well-known Montgomery Multiplication [90] algorithm shown in Algorithm 18. The idea behind this is to perform the next operation.

$$v = x \cdot y \cdot R^{-1} \text{mod } p \tag{7.2}$$

where p is a odd number, usually a prime and $R = 2^w$ has to satisfies $\gcd(R, p) = 1$ and $R > p$. Choosing this setting, line 4 of the Algorithm 18 can be performed by a simple shifting operation. In line 6, there is a final subtraction to fit the result less than p. To multiply x and y, they are converted to Montgomery form xR mod p and yR mod p, respectively.

Lets define $MM(a, b)$ as the function of Montgomery Multiplication the Algorithm 18, $x' = MM(x, R^2) = xR^2R^{-1}$ mod p and $y' = MM(y, R^2) = yR^2R^{-1}$ mod p; then let be z' the result of multiplication of x' and y', $z' = MM(x', y') = (xR)(yR)R^{-1}$ mod $p = xyR$ mod p; finally, for reverting the Montgomery form of z' another call to MM(), $z = MM(z', 1) = (xyR) \cdot 1 \cdot R^{-1}$ mod p. Montgomery algorithm results inefficient when only one modular multiplication is needed. However, when modular multiplication is often applied, this algorithm performs best.

7.2.3 Residue Number System and Modular Arithmetic

An element $a \in \mathbb{Z}_p$ is represented in radix-r as the array $a = \sum_{i=0}^{n-1} a_i r^i$, where $r = 2^w$ and $0 \leq a_i < r$, and we say that the operand a has a word length of n words. For the sake of simplicity, we will only consider operands with an even word size. Particularly, we are interested in the case $n = 8$ required for computing a 256 RNS multiplication with reduction.

The Residue Number System based arithmetic relies on the ancient Chinese Remainder Theorem (CRT). It is defined as follows, Let $\mathcal{B}$ be a set of ℓ pairwise co-prime integer moduli $\mathcal{B} = \{m_1, m_2, \ldots, m_\ell\}$, called an RNS-basis, and let M be defined as, $M = \prod_{i=1}^{i=\ell} m_i$. Invoking the CRT, a number $a \in \mathbb{Z}_M$ can be uniquely represented by the ℓ-tuple $a = (a_1, a_2, \ldots, a_\ell)$, where each a_i is the residue of a modulo m_i. In the

remainder of this operation, the a mod m will be written as $a = |a|_m$.

Let a and b be two k-bit integers with $a, b < M$, represented as the RNS tuples $a = (a_1, a_2, \ldots, a_\ell)$ and $b = (b_1, b_2, \ldots, b_\ell)$. The RNS addition, subtraction, and multiplication of these two RNS numbers denoted by $\oplus$, $\ominus$, and $\otimes$, respectively, can be performed component-wise as

$$\begin{aligned} c = a \oplus b &= (|a_1 + b_1|_{m_1}, \ldots, |a_\ell + b_\ell|_{m_\ell}), \\ c = a \ominus b &= (|a_1 - b_1|_{m_1}, \ldots, |a_\ell - b_\ell|_{m_\ell}), \\ d = a \otimes b &= (|a_1 \cdot b_1|_{m_1}, \ldots, |a_\ell \cdot b_\ell|_{m_\ell}). \end{aligned} \tag{7.3}$$

In this way, the addition, subtraction, and multiplication of elements in $\mathbb{Z}_M$ can be performed using smaller computations modulo m_i, which are independent and can be performed concurrently. Hence, in a hardware platform with ℓ processing units, the computational cost of computing any RNS arithmetic operation in Equation (7.3) is approximately the same as a single operation modulo m_i. Those processing units are implemented using the DSP48 blocks available in virtually all contemporary FPGA devices.

The moduli m_i described above are usually selected as $m_i = 2^w - \mu_i$, where the μ_i values are chosen as small as possible and guarantee pairwise co-primality. If $\mu_i < 2^{\lfloor w/2 \rfloor}$, then the reduction modulo m_i of Equation (7.3), can be efficiently performed by repeating at most twice the operation $d_i = t_i \bmod 2^w + \mu_i \cdot \lfloor t_i / 2^w \rfloor$, where $t_i = a_i \cdot b_i$. This ensures that $d_i \in [0, 2^w]$. Since $2^w > m_i$, one may need to compute a final reduction at the cost of at most one subtraction operation. To assure a constant-time implementation, this reduction should be carried out by executing two unconditional reductions, followed by one conditional subtraction by m_i (*cf.* Algorithm 21).

RNS modular multiplication

Modular multiplication is often performed by first computing the integer multiplication $d = a \otimes b$, followed by a reduction modulo p so that it is guaranteed that the final integer value lies in the range $[1, p-1]$.

In the following, we describe the reduction approach proposed in [13, 92] (see also its hardware and as described in [64, 63]). This approach allows performing a modular

reduction $d \bmod p$ without leaving the RNS domain
.

Let d be a large integer represented in RNS using a basis composed of ℓ one-word moduli. Let us assume that d must be reduced modulo an n-word prime number p. Then, a strategy to perform the modular reduction $d \bmod p$ can be obtained from a direct application of the RNS recovery formula based on the Chinese Reminder Theorem (CRT) as.

$$d = \left| \sum_{i=1}^{\ell} \gamma_i \cdot M_i \right|_M = \left(\sum_{i=1}^{\ell} \gamma_i \cdot M_i \right) \bmod M, \quad \text{where } \gamma_i \triangleq \left| d_i \cdot M_i^{-1} \right|_{m_i}. \quad (7.4)$$

Note that the right-hand side of Equation (7.4) involves a costly reduction modulo M. A way around this issue is to rewrite the value a as

$$d = \sum_{i=1}^{\ell} \gamma_i \cdot M_i - \alpha \cdot M, \text{ with } \gamma_i \triangleq \left| a_i \cdot M_i^{-1} \right|_{m_i}. \quad (7.5)$$

Where α is a positive integer, and by construction $0 \leq d/M < 1$. From Equation (7.5), we can compute α as

$$\alpha = \left\lfloor \sum_{i=1}^{\ell} \frac{\gamma_i}{m_i} \right\rfloor \quad (7.6)$$

Since $\gamma_i < m_i$, we have that $0 \leq \alpha < \ell$. Using the fact that $m_i \approx 2^w$, the ratio γ_i/m_i can be approximated by only considering the σ most significant bits of the quotient $\gamma_i/2^w$ as follows,

$$\hat{\alpha} \triangleq \left\lfloor \sum_{i=1}^{\ell} \frac{\left\lfloor \frac{\gamma_i}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor, \quad (7.7)$$

where σ is an integer in the range $[1, w]$ and $0 < \Delta < 1$ is an error correcting parameter.

The integer part of the summation in Eq. (7.7) can be efficiently computed by considering the output carry c produced by adding the σ most significant bits of the coefficients γ_i with $i = 0, 1, \ldots, \ell$. Notice that the output carry c is an integer in the range $[0, \ell]$. The cost of computing $\hat{\alpha}$ of Eq. (7.7) is that of adding ℓ σ-bit integers. We now observe that the value

$$z = \sum_{i=1}^{\ell} \gamma_i \cdot |M_i|_p - |\alpha \cdot M|_p \,, \tag{7.8}$$

is congruent to $d \bmod p$, as desired. However, due to the approximated computation of the constant α of Equation (7.6), in general $z \geq d$. Fortunately, the estimated valued $\hat{\alpha}$, gives a close enough approximation of α so that $z \approx d$.

Exploiting the framework just described, Algorithm 19 computes the RNS vector $z \equiv d \bmod p$ as defined in Eq. (7.8). At first and off-line, the RNS vector $\left|M_j^{-1}\right|_{b_j}$ for $j \in \{1, \ldots, \ell\}$, as well as the tables of RNS vectors $|M_i|_p$ for $i \in \{1, \ldots, \ell\}$ and $|\alpha \cdot M|_p$ for $\alpha \in \{1, \ldots, \ell - 1\}$, are computed. The combined amount of memory required by this RNS vector and two tables is 10 Kbits that is $(rows \times w \times \ell)$ for each precomputed vector.

In the first loop (Steps 4-8), ℓ processing units concurrently compute ℓ copies of the RNS vector γ described in Eq. (7.4). Although the computational cost of these Steps is ℓ RNS multiplications, as long as all these products are computed in parallel, their associated latency should be very close to the latency associated with one RNS multiplication. The second loop of Algorithm 19 (Steps 10-15) completes the computation of the RNS vector z. Step 11 performs ℓ and $\ell - 1$ RNS multiplications and additions, respectively. All these ℓ RNS multiplications can be computed in parallel, but the additions must be performed sequentially using a binary tree adder. Step 12 computes α by adding ℓ σ-bit integers.

In Step 13, the RNS vector z is obtained by performing one RNS multiplication and one RNS subtraction. In summary, the latency associated with Algorithm 19 is the

combined latency of three RNS multiplications plus one RNS subtraction plus the addition of ℓ σ-bit numbers.

Algorithm 19 RNS Modular Reduction [64].

Require: The integer d given in ℓ-moduli RNS representation, the ℓ-moduli RNS-basis $\mathcal{B}$, and parameters $r, \sigma,$ and Δ.

Ensure: RNS vector z, such that its integer representation is $z \equiv d \bmod p$.

1: *precompute* RNS vector $\left|M_j^{-1}\right|_{b_j}$ for $j \in \{1, \ldots, \ell\}$
2: *precompute* Table of RNS vectors $|M_i|_p$ for $i \in \{1, \ldots, \ell\}$
3: *precompute* Table of RNS vectors $|\alpha \cdot M|_p$ for $\alpha \in \{1, \ldots, \ell - 1\}$
4: **for** each block i **do**
5: **for** each thread j **do**
6: $\gamma_i \leftarrow \left| d_j \cdot \left| M_j^{-1} \right|_{b_j} \right|_{b_j}$
7: **end for**
8: **end for**
9: **for** each thread i **do**
10: **for** each thread j **do**
11: $z_j \leftarrow \left| \Sigma_{i=1}^{\ell} \gamma_i \cdot \left| |M_i|_p \right|_{b_j} \right|_{b_j}$
12: $\alpha \leftarrow \left\lfloor \Sigma_{i=1}^{\ell} \frac{\left\lfloor \frac{\gamma_i}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor$
13: $z_j \leftarrow \left| z_j - \left| |\alpha \cdot M|_p \right|_{b_j} \right|_{b_j}$
14: **end for**
15: **end for**
16: **return** $z = (z_1, \ldots, z_\ell)$

7.2.4 RNS Montgomery modular reduction

Peter L. Montgomery proposed his famous modular reduction in 1985 [91], which can be briefly described as follows. Let us define the Montgomery parameter R as $R = r^n$, where before n represents the number of words necessary to represent the prime

modulus p in radix $r = 2^w$. Hence, $r^{n-1} < p < r^n$. The Montgomery representation $\tilde{a}$ of an element $a \in \mathbb{Z}_p$ is computed as $\tilde{a} = a \cdot R \bmod p$.

Let us assume that the elements $a, b \in \mathbb{Z}_p$ have Montgomery's representation given as $\tilde{a}$ and $\tilde{b}$, respectively. Let c' be defined as $c' = \tilde{a} \cdot \tilde{b}$. Then, the Montgomery product of $\tilde{a}$ and $\tilde{b}$ is defined as $\tilde{c} = \tilde{a} \cdot \tilde{b} \cdot R^{-1} \mod p$ and can be computed as

$$\tilde{c} = \frac{c' + (\mu \cdot c' \mod R) \cdot p}{R} \equiv c' \cdot R^{-1} \mod p. \tag{7.9}$$

the parameter μ given as $\mu = -p^{-1} \mod R$, can be pre-computed off-line. It can be shown that when $0 \leq c' < p^2$, the result $\tilde{c}$ in Equation (7.9) is in the interval $[0, 2p[$. Hence, at most, a single conditional subtraction is needed to obtain $0 \leq \tilde{c} < p$.

It was first shown by Posch and Posch [101] that the Montgomery reduction of Equation (7.9) could be adapted to the RNS representation set. Several ingenious refinements were later introduced in [75], and many more recent papers have discussed other exciting aspects of this topic (See [4] for a comprehensive survey). In the remainder of this section, an RNS Montgomery modular reduction especially designed for hardware implementations, is presented.

The adaptation of the k-bit Montgomery reduction to RNS arithmetic requires handling two distinct RNS-basis $\mathcal{B} = \{m_1, m_2, \ldots m_\ell\}$ and $\mathcal{B}' = \{m'_1, m'_2, \ldots m'_\ell\}$ such that $\gcd(M, M') = \gcd(M, p) = 1$, where $\ell = \lceil \frac{k}{w} \rceil = n$, and $M = \prod_{i=1}^{\ell} m_i$ and $M' = \prod_{i=1}^{\ell} m'_i$. In addition, the Montgomery parameters of Equation (7.9) must be represented using the two RNS bases $\mathcal{B}$ and $\mathcal{B}'$. It is customary to choose $R = M$, so that the modular operation in the second term of the numerator of R in Equation (7.9) is automatically computed. Also, the parameter μ must be specified as an RNS vector $-p^{-1} \bmod R$ represented in base $\mathcal{B}$. Notice that since $R = M$, one must perform an RNS basis transformation from $\mathcal{B}$ to $\mathcal{B}'$, so that the division by R of Equation (7.9) can be computed. Since it is customary to return the result on the original basis, a second RNS basis transformation from $\mathcal{B}'$ to $\mathcal{B}$ must be computed.

The procedure to compute an RNS Montgomery modular reduction is presented in Algorithm 22. It is noted that the multiplication $d_{\mathcal{B}}$ by μ in Step 5, is carried

out in base $\mathcal{B}$. Because of the design choice $R = M$, the reduction modulo R is implicitly applied to this computation. Thus, the product in this step is equivalent to compute $\mu \cdot d \bmod R$ of Equation (7.9). In Step 11, the equivalent to the value $d + (\mu \cdot d \bmod R) \cdot p$ from Equation (7.9) is computed. This operation is performed in base $\mathcal{B}'$ because its result is a multiple of R; thus, it would always be equal to zero in base $\mathcal{B}$.

Finally, in Step 12, a division by R is computed. This corresponds to the RNS representation of $d \cdot R^{-1} \bmod p$ in base $\mathcal{B}'$. The reason why this computation is performed in base $\mathcal{B}'$, is because the value M^{-1} is not defined in base $\mathcal{B}$. Thus, throughout the algorithm, it becomes necessary to perform two *base extensions*, which consist of transforming a number given in base $\mathcal{B}$ (resp. $\mathcal{B}'$) into a number in base $\mathcal{B}'$ (resp. $\mathcal{B}$.) The first base extension (Steps 6 to 10) is made to derive an approximation δ from the value of γ in Step 5. This permits the computation of the value $(d+(\mu \cdot d \bmod R) \cdot p)/R$ in base $\mathcal{B}'$. The second base extension (Steps 14 to 16) is performed at the end of the algorithm to obtain the RNS representation of the result computed in Step 12 in base $\mathcal{B}$.

It can be seen that RNS Montgomery modular reduction requires that all the operations must be performed in both RNS-basis $\mathcal{B}$ and $\mathcal{B}'$ to maintain compatibility with the reduction algorithm.

7.2.5 FPGA and DSP technology

Nowadays, most FPGA devices are equipped with embedded hard cores known as Digital Signal Processing (DSP48) slices or blocks. Due to its dedicated and advanced architecture, DSP48 slices can perform multiple arithmetic and Boolean operations, such as additions, subtractions, multiplications, logical shifts, and comparisons. The DSP48 blocks have a dedicated column-organized interconnection network. This feature helps to share data among all the DSP48 blocks and use them to take advantage of their outstanding processing speed.
Figure 7.1 show a general schematic view of the internal DSP48 slice components. DSP48 slices count with four and two input and out ports, respectively. One of the output ports is directed to the Look-Up Table (LUT) FPGA fabric and other FPGA components. The other output forty-eight-bit port called $PCOUT$ is connected to the DSP48 interconnection network. Besides, each input port has its own register,

Algorithm 20 RNS Jeljeli Modular Reduction.

Ensure: The RNS vectors $d_{\mathcal{B}}$ and $d_{\mathcal{B}'}$, the ℓ-moduli RNS-basis $\mathcal{B}$ and $\mathcal{B}'$, a prime p.

Require: The RNS vectors $z_{\mathcal{B}}$ and $z_{\mathcal{B}'}$ corresponds the integer representation of $z \equiv d \bmod p$.

1: *precompute* RNS vectors $|\mathrm{M}_i^{-1}|_{m'_i}, |\mathrm{M'}_i^{-1}|_{m'_i}, |\mathrm{M}^{-1}|_{m'_i}$ and $|p|_{m'_i}$ for $i \in \{1, \ldots, \ell\}$.

2: *precompute* Matrices of vectors $|\mathrm{M}_i|_{m'_i}$ and $|\mathrm{M'}_i|_{m_j}$ for }

1:11.9895- 0 G /F102 1199552 Tf 1.02 0 0 10.161 655.9471.9895

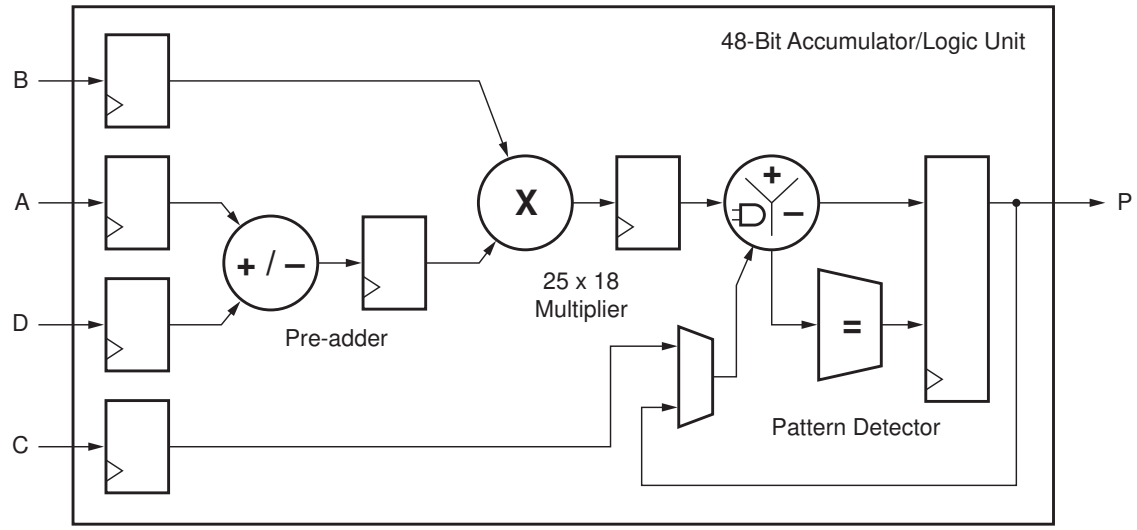

Figure 7.1: Simplified DSP48E1 architecture

The DSP48 ALU has an internal adder that takes as input operands the input registers of the ports A and D. The internal 25×18-bit signed multiplier. The ALU can be programmed for performing forty-eight-bit addition, subtraction, and logical operations.

The DSP named DSP48 has a more advanced architecture that is only available in the FPGA Xilinx family of devices *ultrascale* named DSP48E2[1]. It has several differences concerning the previous architecture. Some of the most significant architectural changes are: The built-in multiplier can handle 27×18 bit operands. Moreover, the *pre-adder* unit also increased its operands to 27 bits. Hence, the input ports A and D and associated registers also increased to 27 bits. It is now possible to perform up to sixty-four different arithmetic and logical operations with up to four input operands. These operations can be programmed using a 6-bit control word. We want to point out that DSP48 blocks offer better performance than the LUT blocks available in the FPGA fabric because LUTs work with an independent clock and often reach at least 500MHz whether internal registers are internally used or not.

[1] https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

Knowing these devices' characteristics and capabilities is essential since this allows us to take advantage of their internal architecture and functionalities. Thus, it is possible to design circuits with adequate length input operands without creating more complex circuits that impact the architecture's performance to be designed.

For the problem addressed in this chapter, the input operands are of length w, which coincides with the length of each segment of the RNS. Thus it is possible to implement several DSPs with different word widths in both DSP architectures presented in this section. According to the multiplier capabilities integrated into the Xilinx FPGA's family 7, the size of $w = 34$ bits was determined.

7.3 Related works

In this section, we present some related work to the main topic of this chapter. Kawamura *et al.* introduced in [74] the "*Cox-Rower*" architecture. This architecture allows the parallelization of "*Rower*" components used for computing RNS field multiplication using Montgomery reduction. This architecture also counts with an accumulator with modular reduction. The unit "*Cox*" is the evaluator of the factor $\hat{k}$. The components *Cox* and *Rower* can be seen as part of the datapath.

A hardware implementation was presented in [17], using similar ideas related to the "*Cox-Rower*" architecture. The main difference lies in the quantity of required *Rower* units. In [17] this number is reduced to $\frac{n}{2}$. Also, the authors employ an extra 6-bit "*Cox*" unit that permits to compute an extra RNS channel. The authors of [17] report their results in several FPGA families such as the Xilinx *"Spartan6, Virtex5 and kintex7"*. Their version of the "*cox-rower*" architecture requires both DSP48 slices and also slices from the FPGA fabric. The main application of the multiplier designed in [17] was to perform elliptic curve cryptography. The authors analyze different design options to evaluate time-space tradeoffs. In the case of the paper by [96], the proposed architecture is a variant of the "*cox-rower*" where each "*Cox*" unit is inside the "*Rower*" component. The "Rower " units are interconnected in a ring instead of a communication bus.

In [53], the ALU unit was developed using exclusively DSP48 slices. The authors decided to work with 16-bit operands. The accumulator required by the *Cox-Rower*

architecture was moved to the output of the first multiplier. Furthermore, the authors employed a 10-stage pipeline and included a Montgomery ladder procedure for performing the elliptic curve scalar multiplication.

We also included in our comparison analysis the RNS implementations in software with applications to elliptic curve cryptography. In [4], several variants of the Montgomery reduction algorithm are proposed along with its implementation in software.

7.4 Design of a DSP48 -based architecture for a eld multiplier

In this chapter, the proposed field multiplier DSP48 -based architecture implemented component-wise RNS multiplier followed by RNS reduction using Algorithm 22 and Algorithm 19. Both of these algorithms require RNS additions, subtractions, and multiplications. For an n-word prime number, the size of the RNS basis for Algorithms 22 and 19 is of $2n + 3$ and $\approx n + 2$, respectively. The former requires just one basis, whereas the latter requires two different bases. Both algorithms require the pre-computation of several RNS vectors and tables, which will be described in the remainder of this section. These values were stored using the Flip-flop and LUT blocks in the target FPGA devices.

Another important task is to add n products as efficiently as possible using a binary addition tree.

Previous works, such as [17], have used the DSP48 blocks for performing the RNS multiplication but the FPGA fabric for performing the field reduction.

In the remainder of this section, we will present the design of each one of these blocks, keeping in mind the following goals.

- To achieve a latency fast enough to compete not only with other hardware designs but also with recent CPU and GPU implementations.

- To exploit parallel or pipeline approaches or a combination of them.
- To provide a design primarily based on DSP48 blocks. This implies the usage of 34×34 bit multipliers.

7.4.1 Basic RNS multiplier with reduction

As discussed in §7.2.3, an RNS multiplication is performed component-wise using one of the ℓ one-word modulus taken from the RNS basis $\mathcal{B}$. Let us recall that the word size is k bits. The basic operation procedure is presented in Algorithm 21. Algorithm 21 requires three inputs, namely, the two one-word operands A_i and B_i, along with the value μ corresponding to a modulus $m_i \in \mathcal{B}$ of the form, $m_i = 2^w - \mu_i$. In the case that the word product $A_i \cdot B_i$ is larger than 2^w then a fast reduction process is performed as shown in Steps 4-6 of Algorithm 21.

Algorithm 21 Basic RNS reduction module.

Require: The RNS words A_i and B_i, the μ_i parameter of the RNS basis modulus $m_i = 2^w - \mu_i$.

Ensure: A

Consequently, we define the word size of our architecture as $w = 34.$ bits; in this way, we try to maximize the size of each port a, b to 18 bits, using these sizes to get a two's complement result. Also, the multiplication result is sent via the PCOUT port, and in the other DSP, apply a right shift of 17 bits to use the 17 most significant bits in its operation, wherever it is easier to perform the reduction operation using the internal interconnection network dedicated to DSP slices with a bus up to 48 bits of length.

We tried to maximize the use of ports A and B of the DSP48 ; that is, if we used the asymmetric multiplier, asymmetric shifts became a problem. That is why we chose to use a 34-bit word, and we can use the DSP48 with a symmetric multiplier without any major problem. It also has the advantage that the results can be sent through the $PCOUT$ port of the DSP48 , and therefore, it was easier to make the reduction directly without the need to complete a 68-bit result. Another advantage is that the sums carried out within the same DSP48 are kept within the size of the internal bus of the DSP48 , which is 48 bits, which avoids having to complete the bus to make more operations.

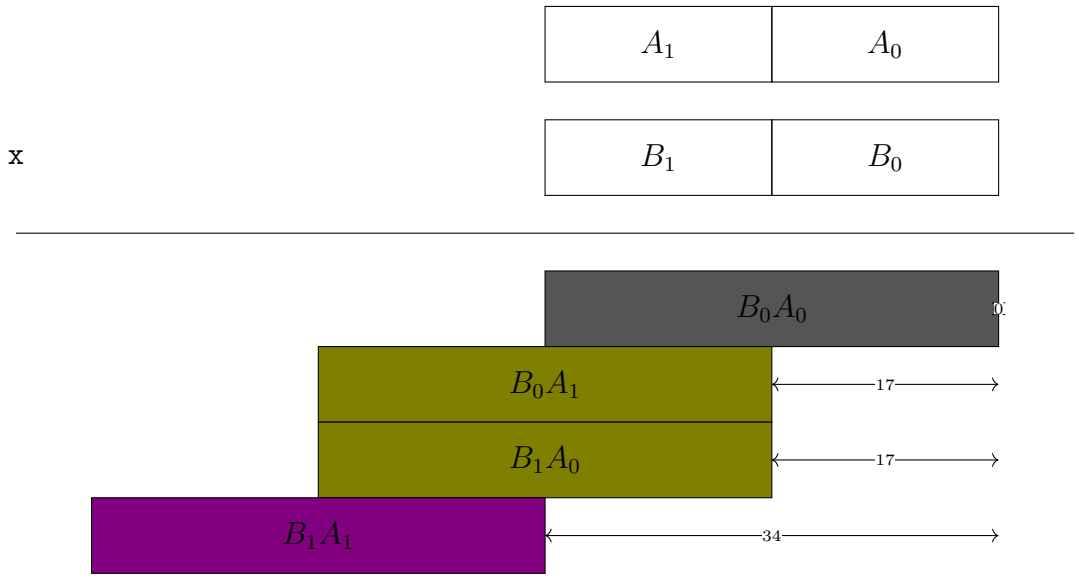

Figure 7.2: Two-word schoolbook multiplication method

Algorithm 22 RNS Montgomery Modular Reduction in HW.

Ensure: The RNS vectors $d_{\mathcal{B}}$ and $d_{\mathcal{B}'}$, the ℓ-moduli RNS-basis $\mathcal{B}$ and $\mathcal{B}'$, a prime p.

Require: The RNS vectors $z_{\mathcal{B}}$ and $z_{\mathcal{B}'}$ corresponds the integer representation of $z \equiv d \bmod p$.

1: Tables of RNS vectors $|\alpha \cdot (-\mathrm{M})|_{m'_i}$ and $|\alpha \cdot (-\mathrm{M}')|_{m_i}$ for $\alpha, i \in \{1, \ldots, \ell\}$.

2: RNS vectors $\mathrm{PC}_1 \leftarrow \left||P|_{m_i} \cdot |M_i^{-1}|_{m_i}\right|_{m_i}$, $\mathrm{PC}_2 \leftarrow \left||M^{-1}|_{m'_i} \cdot |M_i'^{-1}|_{m'_i}\right|_{m'_i}$, $\mathrm{PC}_3 \leftarrow \left||M^{-1}|_{m'_i} \cdot |M_i'^{-1}|_{m'_i} \cdot P|_{m'_i}\right|_{m'_i}$, $\mathrm{PC}_4 \leftarrow \left||M^{-1}|_{m'_i} \cdot |p|_{m'_i}\right|_{m'_i}$, and $|M^{-1}|_{m'_i}$.

3: **for** each processing unit i **do**

4: $\quad \theta_i \leftarrow \left|d_{\mathrm{B_i}} \cdot |\mathrm{PC}_1|_{m_i}\right|_{m_i}$

5: $\quad \gamma_i \leftarrow \left|d_{\mathrm{B'_i}} \cdot |\mathrm{PC}_2|_{m'_i}\right|_{m'_i}$

6: $\quad \gamma'_i \leftarrow \left|d_{\mathrm{B'_i}} \cdot |\mathrm{M}^{-1}|_{m'_i}\right|_{m'_i}$

7: **end for**

8: $\alpha \leftarrow \left\lfloor \Sigma_{j=1}^{\ell} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} \right\rfloor$

9: **for** each processing unit i **do**

10: $\quad \delta_i \leftarrow \left|\Sigma_{j=1}^{\ell} \left||\mathrm{M}_i|_{m'_j} \cdot \theta_j\right|_{m'_i} + \left|\alpha(-\mathrm{M})\right|_{m'_i}\right|_{m'_i}$

11: $\quad \theta_j \leftarrow \left|\gamma_i + (|\delta_i \cdot \mathrm{PC}_3|_{m'_i})\right|_{m'_i}$

12: $\quad z_{\mathcal{B}'_i} \leftarrow \left|\gamma'_i + (|\delta_i \cdot \mathrm{PC}_4|_{m'_i})\right|_{m'_i}$

13: **end for**

14: $\alpha \leftarrow \left\lfloor \Sigma_{j=1}^{\ell} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + 0.5 \right\rfloor$

15: **for** each processing unit i **do**

16: $\quad z_{\mathcal{B}_i} \leftarrow \left|\Sigma_{j=1}^{\ell} \left||\mathrm{M}'_i|_{m_j} \cdot \theta_j\right|_{m_i} + \left|\alpha(-\mathrm{M}')\right|_{m_i}\right|_{m_i}$

17: **end for**

18: **return** $z_{\mathcal{B}}$ and $z_{\mathcal{B}'}$

Two different design alternatives were studied from Algorithm 21. These designs are depicted in Figure 7.3.

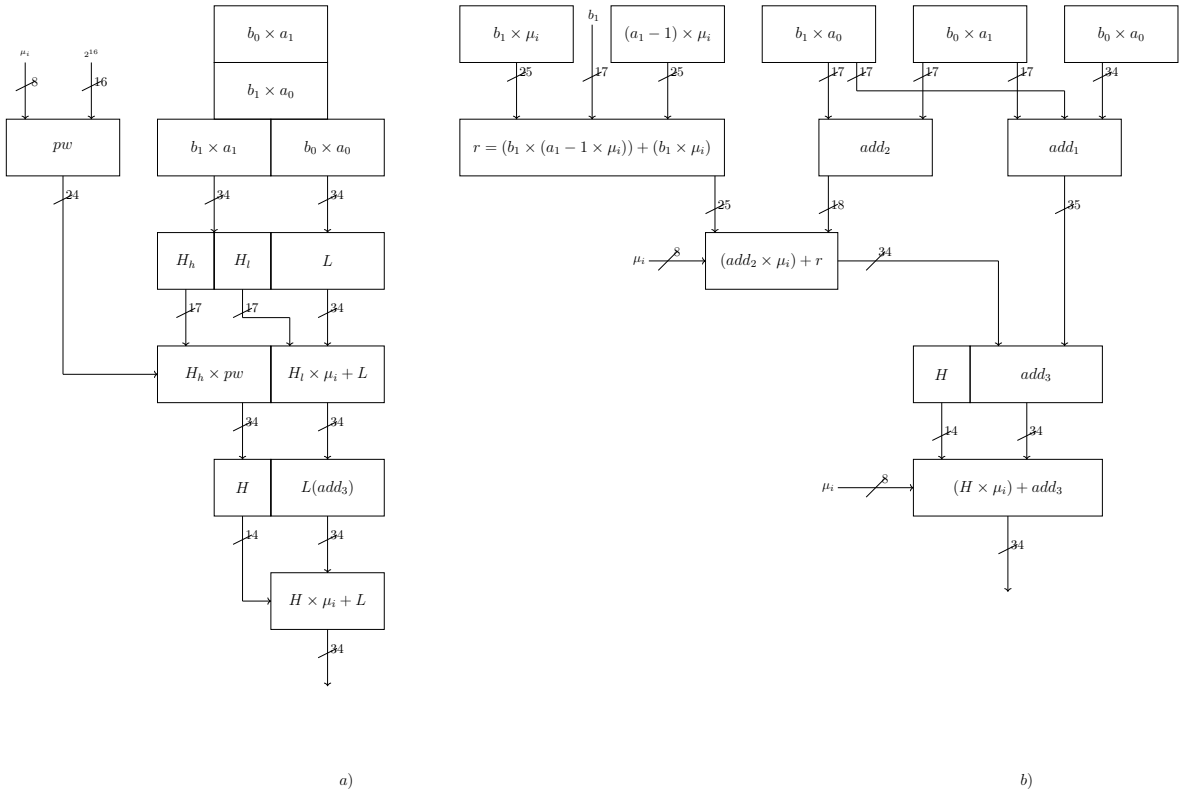

Figure 7.3: Proposed RNS component-wise multipliers

Figure 7.2 calculates two 34-bit words using the schoolbook method by dividing them into four sub-operands extracted from the two-word operands A and B. All the operations are performed concurrently, *i.e.*, each DSP48 block computes in the first clock cycle the operation $B_i \cdot A_i$ for $i = 0, 1$, and outputs its result to the following DSP48 through the port $PCOUT$. This accumulates the product just computed with a previous result. Moreover, the seventeen least significant bits of the DSP48 blocks B_0A_0 and B_1A_0 must be stored because they become the thirty-four least significant bits of the product $A \cdot B$.

Considering that a subsequent reduction by the element μ_i corresponding to the modulus m_i of the basis $\mathcal{B}$ must be performed, the next stage of this component performs two reduction steps. First, sixty-eight bits are reduced from the schoolbook

multiplication of the rst fty-one bits. In the second step, the output is reduced to thirty-four bits using the parameter to perform the reduction. This module is performed by the combination of nine DSP48 blocks. Taking advantage of the internal structure of the DSP48 block, a one-stage pipeline was designed for performing this task.

The rst design is shown in Figure 7.3.a follows more the pattern of a multiplier with a diamond shape. This helps reduce the critical path and take better advantage of the internal pipeline in the DSP48 blocks. In this design, the lower and higher products A_0B_0 and A_1B_1 are added with the cross products A_0B_1 and A_1B_0: Besides saving additions, this arrangement permits to work directly in the reduction phase without worrying about the carry-out that will be processed during the rst stage. This design only requires eight DSP48 blocks.

The second design shown in Figure 7.3.b, is fully oriented to the most modern FPGA devices in the Xilinx series "ultrascale". This family of FPGA devices is equipped with 27 18 bit multipliers. This permits the proposal of another RNS word multiplier architecture that can achieve the desired result with fewer pipeline stages and a shorter critical path. The design involves the manipulation of the components of the operation, $b_i \quad a_i \bmod {}_i$: As can be seen from Figure 7.3.c, this approach yields a ve-level architecture.

The two RNS word multiplier designs shown in Figure 7.3 are referred to in the sequel as RNSModule RNSModule is the most basic component of our RNS multiplier architecture. Because of design and technological reasons, we decided to adopt the second design of Figure 7.3.b and dismiss the usage of the others. This selection is because the limited DSP blocks in the Virtex-ultrascale series do not exceed 12k DSP block. Also, we want to reduce the signal tra c into the FPGA to keep a high frequency. Wherever we want to use a large complete multiplier, the DSP available is not enough with the third design of the RNSModule.

7.4.2 Multiplier array MulDM

As mentioned, RNSModule is the basic module to achieve the RNS parallelism. Given a basis B of cardinality `;, replicating this module ` times to obtain an RNS product su ces. Hence, the RNS product $c_i = a \cdot b \bmod m_i$ with $i \in (0; \ldots; ` - 1)$; can be processed in parallel using the multiplier array named MulDM as shown in Figure 7.4.

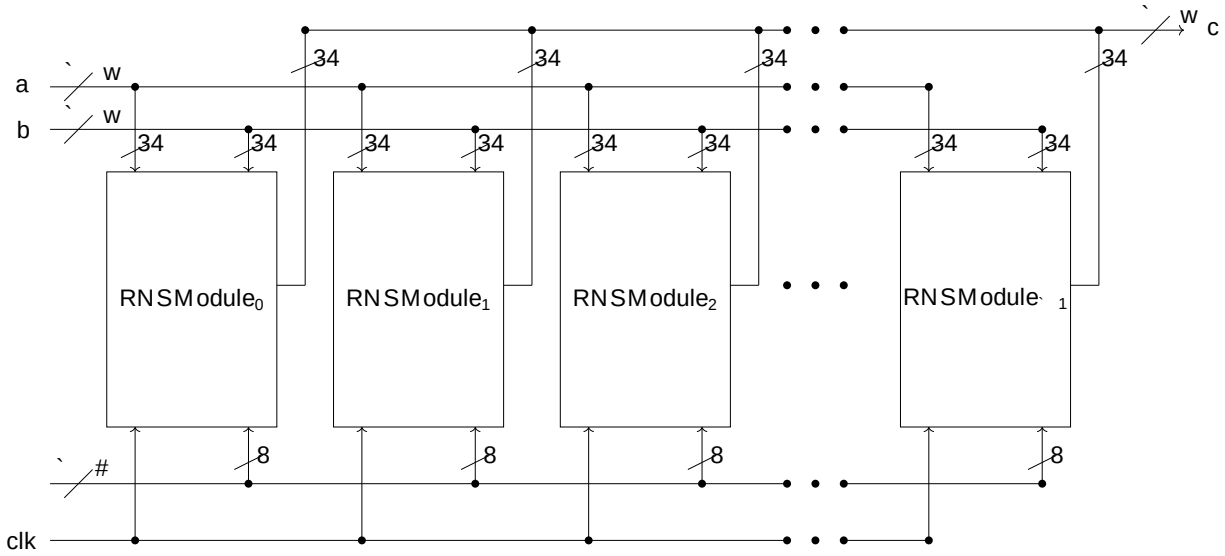

Figure 7.4: RNSModule components array

Each module of the array RSNModule is interconnected to a databus named a; b. This bus distributes the one w-bit word operands a_i and b_i: Similarly, another databus named provides all the reduction parameters $_i$ corresponding to the modulus m_i that belong to the RNS basis B: This way, the array RSNModule can produce an RNS product every six cycles. Since array RSNModule has been designed in the pipeline, it admits a new set of operands every clock cycle.

7.4.3 RNS addition with reduction

A required operation for the Montgomery reduction Algorithm 22 is to operate $a_i + b_i \bmod m_i$: This module is depicted in Figure 7.5. It consists of two stages; in the

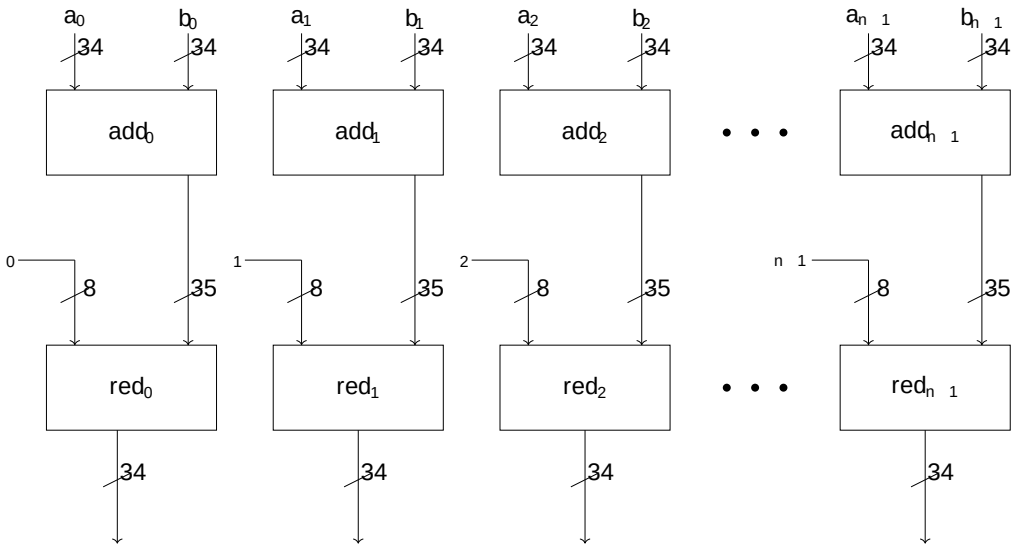

Figure 7.5: Addition with reduction

rst stage, a component-wise RNS addition with operands $a_i + b_i$ is performed. In the second stage, a thirty-four-bit reduced result modulo $_i$ is output. This component was fully designed using DSP48 blocks and requires two clock cycles to perform one RNS addition.

7.4.4 Addition tree

As mentioned before, the RNS reduction algorithms discussed in §7.2.3 require the addition of the result of ` independent products. To perform this task and take advantage of the DSP48 block internal structure, we designed the addition tree shown in Figure 7.6. In the rst level, pairs of operands are added, requiring a total of $\frac{`}{2}$ adder modules. In the second level of the adder tree, three operands are added at once. This is accomplished by taking advantage of the fact that the DSP48 blocks are equipped with two outputs. One of the DSP48 outputs feeds the next DSP48 block using its PCIN input. This allows for reducing the cost that would have been associated with a pure binary adder tree.

Hence, for an RNS basis of cardinality`; the adder tree of Figure 7.6 requires

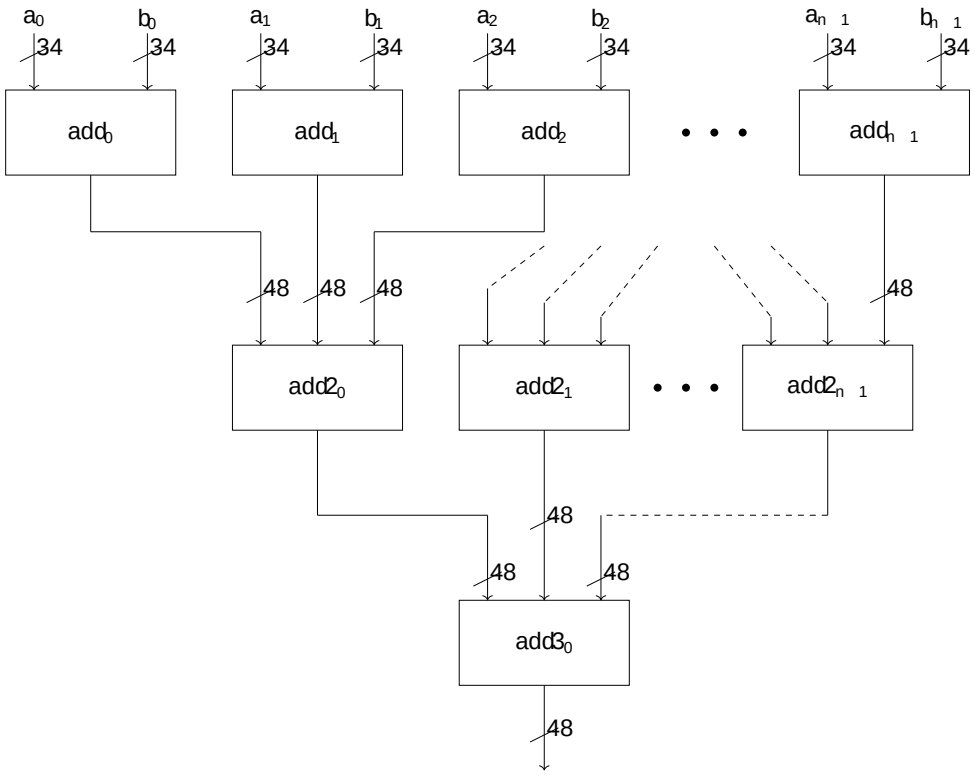

Figure 7.6: Addition tree

$N = 1 + \lceil \log_3 \frac{\ell}{2} \rceil$ levels and a maximum of $\lceil \frac{\ell}{2} \rceil + (N - \log_3(\frac{\ell}{2} - 1)) - 1$ DSP48 blocks, as an example, supose a base with $\ell = 19$ then $N = 4$ and 16 DSP48 blocks will be needed.

7.5 Implementation

Our implementation performed all the operations on DSP48 slices, leaving only the control unit using LUTs. Two FPGA devices were targeted: a high-end Virtex-7 ultrascale+ (xcvu7p- va2104-3-e) Moreover, this family has many DSP slices, including the recent DSP48E2 model, equipped with a 27 18 bit multiplier.

In the remainder of this section, each of the two Montgomery reduction algorithms is analyzed.

7.5.1 Implementation of the modular reduction Algorithm 19

In Figure 7.7, a general diagram presenting the main blocks of the architecture is presented. Figure 7.7 shows the interconnection among the main modules and the main modules are the multiplier arrangement MulDM, the Adder, and the array z, which processes the for loop and the distributed memories.

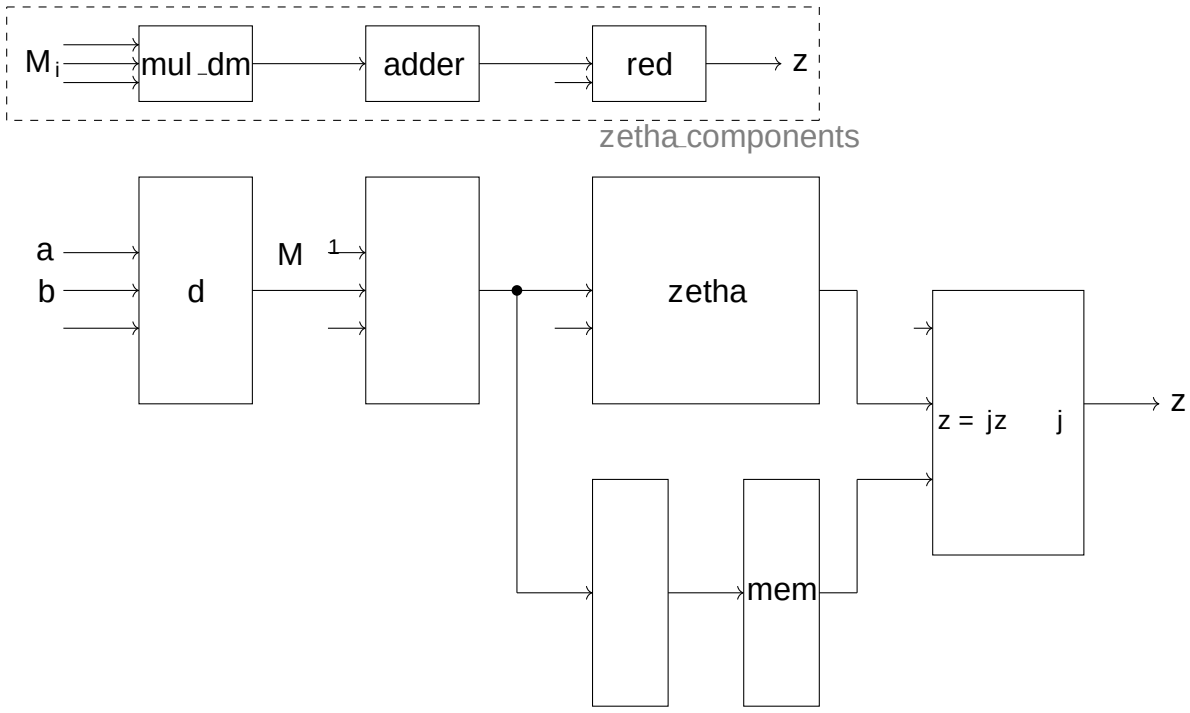

Figure 7.7: General architecture performing the modular reduction of Algorithms 19 and 22

Figure 7.7 implements Algorithm 19. In Figure 7.7, one first receive as input the parameter $d = a_i b_i \bmod b$: Then, our implementation operates, $a_i b_i \bmod (2^w b_i)$ with $i \in \{0, \ldots, \ell - 1\}$.

This way, the result in RNS representation d; is composed of ℓ 34-bit words. After that, the multiplication of Step 5 of Algorithm 19 is performed, i.e., each 34-bit word of each operand is processed by the module MulDM. Since the block MulDM performs the RNS multiplication of ℓ words in six clock cycles. once that the resulting $_j$ of Step 6 have been computed in component, it follows the computation of their addition performed into component. This produces the value of step 12. This can be processed concurrently with Step 11. Taking advantage of the non-dependency of these steps, zeta can process all the required multiplications using the corresponding value $_i$ in 11 clock cycles. Here we have an array of components of zeta (shown in a dashed box) that perform all the operations as a parallel matrix. This step is performed as a matrix processing of rows internally composed of all components shown in the dashed square in Figure 7.7 in an array of rows; each row processes the multiplication for each one M_i then here it is necessary then once the result outs from RNSModule to the adder tree the output from the adder is at least of $w + 1$ bits then another reduction allows to deliver a w bits word for each row into the matrix component named z.

In Step 12, an independent Adder component computed the parameter in parallel. Also, the pre-computed tables have been queried in such a way that when is ready, so are the pre-computed values ready to be subtracted in Step 13 with the value z_j for $j = 0; 1; : : : ; \ell$: The result is stored in a register considering that the operands are processed from the least signi cant bits to the most signi cant bits. This process takes ℓ clock cycles. Finally, the whole design takes sixty clock cycles to process the RNS operation $a \cdot b \bmod p$:

7.5.2 Montgomery Implementation

This Section describes the implementation of Algorithm 22. The hardware modules are the ones presented in §7.4. The main design enforced in our implementation was to try to exploit the pipeline available in the RNSModule module as much as possible.

7.6 Results

This section reports the results obtained from the hardware implementation of the RNS eld multiplier.

7.6.1 RNS word multiplier with reduction

The most important component of our architecture is the RNS word multiplier with reduction shown in Figure 7.3. We present the hardware performance of the three designs proposed for performing the RNS word multipliers.

Proposal	FPGA	Freq	Time ns	FF	DSPs	Pipeline
Mult a)	Kintex-7	159.49 Mhz	6.27 ns	125	7	1
	Virtex-7	185.18 Mhz	5.4 ns	91	7	1
Mult b)	Kintex-7	307.62 Mhz	3.25 ns	32	8	6
	Virtex-7	396.82 Mhz	2.52 ns	32	8	6
Mult c)	Kintex-7	303.03 Mhz	3.3 ns	16	11	5
	Virtex-7	400 Mhz	2.5 ns	16	11	5

Table 7.1: Comparative table for the three proposed RNS word multipliers with reduction

Table 7.1 reports the number of resources required by the three proposed designs shown in Figure 7.3. Due to compatibility reasons, we have chosen the design of Figure 7.3b). This design option is compatible with most FPGA families of devices. At the same time, this design achieves a maximum clock frequency similar to the one associated with the design option of Figure 7.3c). However, it requires one extra clock cycle. In the case of the design of Figure 7.3a), despite its relatively low maximum clock frequency (around 160 MHz up to 186 MHz), it has a latency of just one clock cycle. However, using this RNS word multiplier design implies a general slowdown of the architecture. Because of that, it was not considered any further.

The implementation of 7.3c) yields the best performance. However, the Ultrascale family of FPGA devices ranks among the most expensive.

Table 7.2 presents the time and area complexity for both RNS reduction algorithms studied in this chapter.

One can see that 5229, 3243, and 373 LUTs, slices, and DSP48 blocks are required,

respectively. Let us recall that the RNS reduction Algorithm 19 requires just an RNS basis of length $2n+3$; where n is the size in 34-bit words of the prime modulus p: Since we were dealing with 256-bit primes, the length of the RNS basis for this scenario is of nine-teen moduli.

It can be seen that the resources required by the Montgomery reduction procedure shown in Algorithm 22 are lesser than the ones required by Algorithm 19. 3,682, 3,673, 204 LUTs, and DSP48 slices are required in this case. The Montgomery reduction algorithm has a smaller area complexity than Algorithm 19. Let us recall that in the case of the Montgomery reduction procedure of Algorithm 19, one requires two RNS bases, each one of length $n+2$ moduli. Since we were dealing with 512-bit primes, the length of each of the two RNS bases for this scenario is seventeen moduli.

	Jeljeli	Montgomery
Frequency (MHz)	149	156
Clock cycles	1	1
Latency	28	35
Total time by product	187.88ns/ 6.71ns	224.35ns/ 6.41ns
Device	Virtex7	Virtex7
Area (slices)	2354	3726
DSP	3968	2528
FF	8025	7570

Table 7.2: Total resources of both implementations

For a 256-bit multiplication, the amount of memory required by the pre-computed values of Algorithm 22 can be outlined as follows. The pre-computed vectors and Tables are,

$$|M|_{m_i'}; |M^2|_{m_i}; |(-M)|_{m_i'}; |(-M^2)|_{m_i}$$

This amounts to something close to 10K bits, as shown in Table 7.3.
The clock cycle count and maximum clock frequency for the implementation of the modular reduction Algorithms 22 and 19 for the targeted FPGA devices of the Xilinx

	word x size	bits
$jM_i\ ^1j$	8 x 34	272
$jM_i^{\ 0\ 1}j_{m_i}$	8 x 34	272
$jM\ ^1j_{m_i^0}$	8 x 34	272
$jPj_{m_i^0}$	8 x 34	272
$jM_ij_{m_j^0}$	(8 x 34) x 8	2176
$jM_i^0j_{m_j}$	(8 x 34) x 8	2176
$j\ (\ M)j_{m_i^0}$	(8 x 34) x 8	2176
$j\ (\ M^0)j_{m_i}$	(8 x 34) x 8	2176
Total		9792
$jjPj_{m_i}\ \ jM_i\ ^1j_{m_i}j_{m_i}$	8 x 34	272
$jjM\ ^1j_{m_i^0}\ \ jM_i^{\ 0\ 1}j_{m_i^0}j_{m_i^0}$	8 x 34	272
$jjM\ ^1j_{m_i^0}\ \ jM_i^{\ 0\ 1}j_{m_i^0}\ \ Pj_{m_i^0}j_{m_i^0}$	8 x 34	272
$jjM\ ^1j_{m_i^0}\ \ Pj_{m_i^0}j_{m_i^0}$	8 x 34	272
$jM_i^0j_{m_j}$	(8 x 34) x 8	8704
$jM_ij_{m_j^0}$	(8 x 34) x 8	8704
$j\ (\ M)j_{m_i^0}$	(8 x 34) x 8	8704
$j\ (\ M^0)j_{m_i}$	(8 x 34) x 8	8704
Total		9792

Table 7.3: Comparative table with total memory required

Kintex-7 and Virtex-7 families.

7.6.2 Discussion and comparison

Here we compare against related work. The reference work [34] where the cox-rower architecture was introduced. As mentioned, the rower units process the data in parallel in this architecture. In contrast, the cox unit is an accumulator that distributes the carries to the corresponding rower units.

ˆ We compare against the work reported in [53]. This is mainly based on a module called \inner Montogomery ALU". They use an eight-stage pipeline.

They implemented their ALU using di erent word sizes ranging from 15 to 17 bits, achieving a maximum frequency of 400 MHz using a 10-stage pipeline.

ˆ In the work by [17], they used a similar ALU unit organized on a six-stage pipeline. However, this design strongly in uences the ox-rower architecture.

For a number ` co-primes, the number of DSPs used by the basic RNS is rns( mod) module. The levels that the adder will have is N = 1 + d log$_3$ $\frac{`}{2}$e, and the number of DSPs needed for the adder is stage_add = d $\frac{`}{2}$ + N log$_3$( $\frac{`}{2}$) 1)e 1.

the \mulDM" component consisting of two arrays of ` components with muldm = (2 ` rnsmod) and the array will need a z = ` ((` rnsmod) + stage_add + 1). And the nal subtraction needs last_sub = (` 2). Therefore, the DSPs needed are total = z + stage_add + muldm + last$_s$ub.

As an example, a 512-bit operand in RNS representation will need ll = d $\frac{512}{34}$e = 16 34-bit words, assuming the basic multiplier design RNSModule its uses 8 DSPs (design b)) and it will have rnsmod = 8, N = 1 + d log$_3$ $\frac{`}{2}$d = 4 levels of the inverted tree for the sum that calculates to .

It also needs an stage_add = d $\frac{`}{2}$ + ( N log$_3$( $\frac{`}{2}$) 1)e 1 = 27 DSPs needed to carry out the sum, and the component \MulDM" that carries out the calculation of the value d from is muldm = 2 ` rnsmod = 560 DSPs, to process the matrix from step 11 of the algorithm 19. The addition and reduction are carried out for each row.

We have z = ` (` rnsmod + stage_add + 1) = 10780 DSPs, nally, the nal subtraction of step 12 is processed in last_sub = ` 2 = 70 DSPs. Now the total is obtained by adding all the previous values; that is, total = z + stage_add + muldm + last $_s$ub = 11437 DSP48 needed to implement the Jeljeli reduction algorithm will be added.

Chapter 8

Hardware accelerator for the elliptic curve ECC25519

This chapter shows a different focus on multiplier implementation as a further approximation of how to improve the architecture proposed in chapter 7. This alternative proposal explores two slopes; the first uses the schoolbook method for multiplicating two large operands of 255 bits, and the second uses the Karatusba multiplication method. In this work, we multiplicate two integers on 255 bit for the elliptic curve ECC 25519 using the two methods mentioned above on a Xilinx FPGA and taking advantage of the DSPs available in the FPGA. Therefore each design has a different focus from the previous RNS proposal from chapter 7.

In the paper, [14], Daniel J. Bernstein introduces and analyzes the elliptic curve Curve25519, in which each user has a 32-byte secret key and public key of the same length. As a low-level view: the function of the curve is F_p restricted x-coordinate scalar multiplication over $E(F_{p^2})$, with p the prime number $2^{255} - 19$ seen as the hexadecimal number 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFED and E the elliptic curve $y^2 = x^3 + 486662x^2 + x$. The paper also reports computer implementations from 2006 and earlier processors. The design and implementation proposed by Bernstein focus on designing and implementing an elliptic curve Diffie-Hellman, having special care in the choices to obtain a better performance and avoiding the wrong decisions can destroy it.

We base our proposal on this vision, with the main difference in the implementation because we use an FPGA and (Digital Signal Processors) DSPs to perform the multiplication only and later implement our design with a control circuit to achieve a

Diffie-Hellman protocol entirely.

8.1 Karatsuba Proposal

This section shows a diamond adder proposal and a critical path for implementing the Karatsuba algorithm presented in chapter 2 section 2.3.2. The design has dedicated DSPs included in almost all modern FPGAs, and hereafter we give the required equations:

For this case, we need $R = b^n$, $d = 2n$ with $B = (B_{d-1}, ..., B_0)_b$ and $A = (A_{d-1}, ..., A_0)_b$ two integers consist of d words obtained when we divide the two integers B, A into smaller words with a fixed length as required, for the Karatsuba design we use two operands know as most significant word and less significant word, then we use the following equation.

$$B \times A = (B_1 \times A_1)R^2 + ((B_0 + B_1)(A_0 + A_1) - B_1A_1 - B_0A_0)R + B_0A_0 \quad (8.1)$$

With the Karatsuba equation 8.1 and the explanation from chapter 2 section 2.3.2, we can divide the most and less significant words into smaller operands and later process them using the schoolbook method presented in chapter 2 section 2.3.2. Hence, the Karatuba design uses the multiplier embedded into the DSPs as a basis. Therefore we can process the $m \times n$ individual multiplications in parallel with one clock cycle, and figure 8.1 shows the results as a diamond array.

We only use one level of Karatsuba to process both integers of 255 bits. Furthermore, we divided both inputs into two words of 128 bits, each denoted by B_H, A_H and B_L, A_L. The diamond and its subindex reference the addition of the results from all individual multiplications $(B_H \times A_H), (B_L \times A_L)$. Additionally, the word size used for this design has a length of $w = 17$ bits using a symmetric configuration of the DSP, allowing inputs of 18×18 bits with an output result of $P = 36$ bits.

The diamond has 15 levels, and we process them in parallel, exploiting the possibility of using the DSP adder option with two or three inputs simultaneously; Thus, we get figure 8.2, which shows the same diamond but with the difference as each "box" illustrates the processing of two or three inputs, and the operations performed are shown in the following equations 8.2 and 8.3.

$$DSP_2 + C + CONCAT \quad (8.2)$$

	$B_0 \times A_7$	
	$B_1 \times A_6$	
$B_2 \times A_7$	$B_2 \times A_5$	$B_0 \times A_5$
$B_3 \times A_6$	$B_3 \times A_4$	

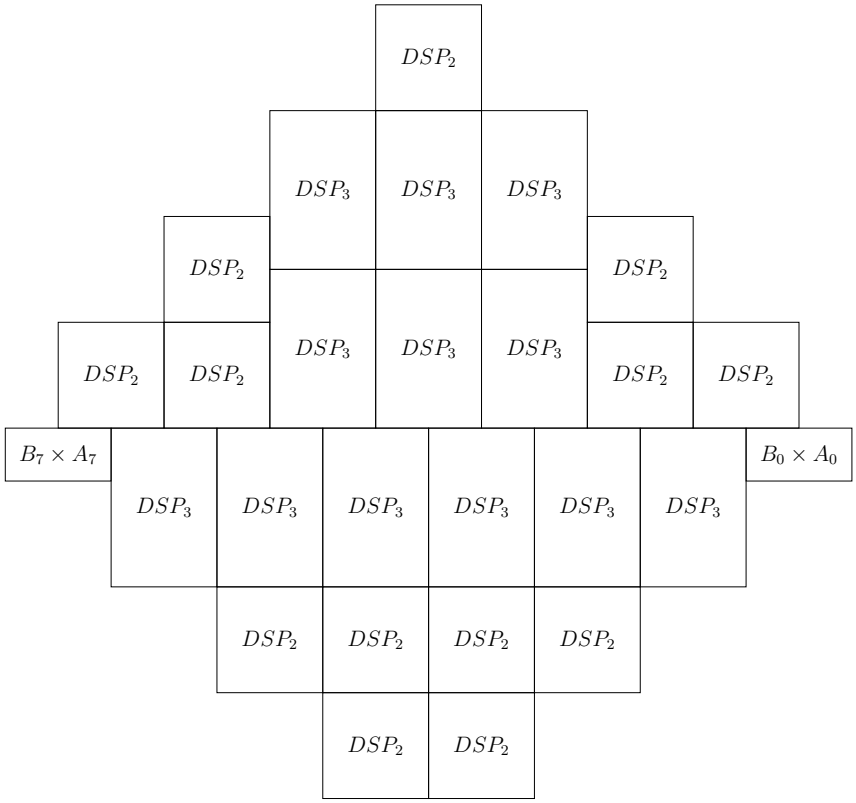

Figure 8.2: Karatsuba diamond with DSPs

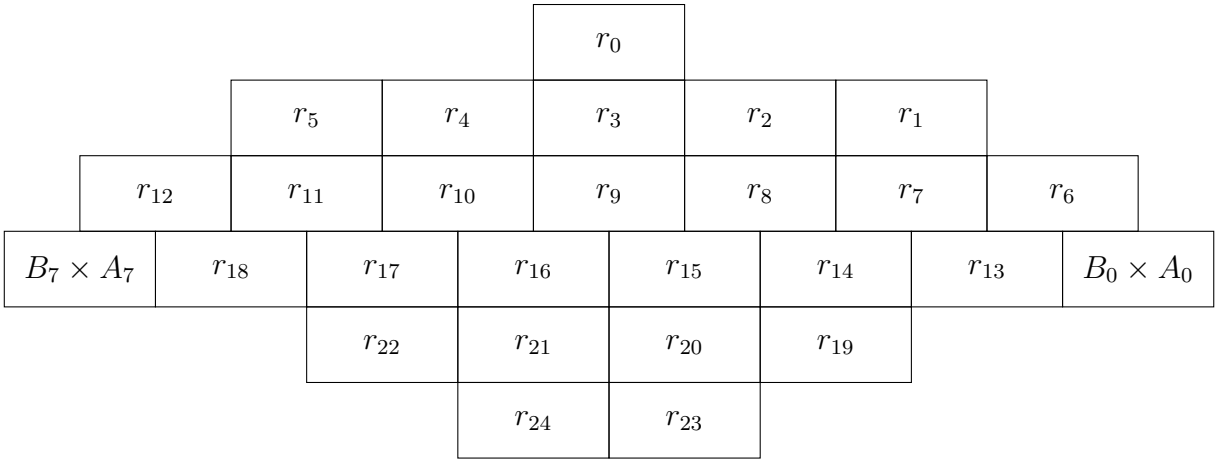

Figure 8.3: Karatsuba diamond results

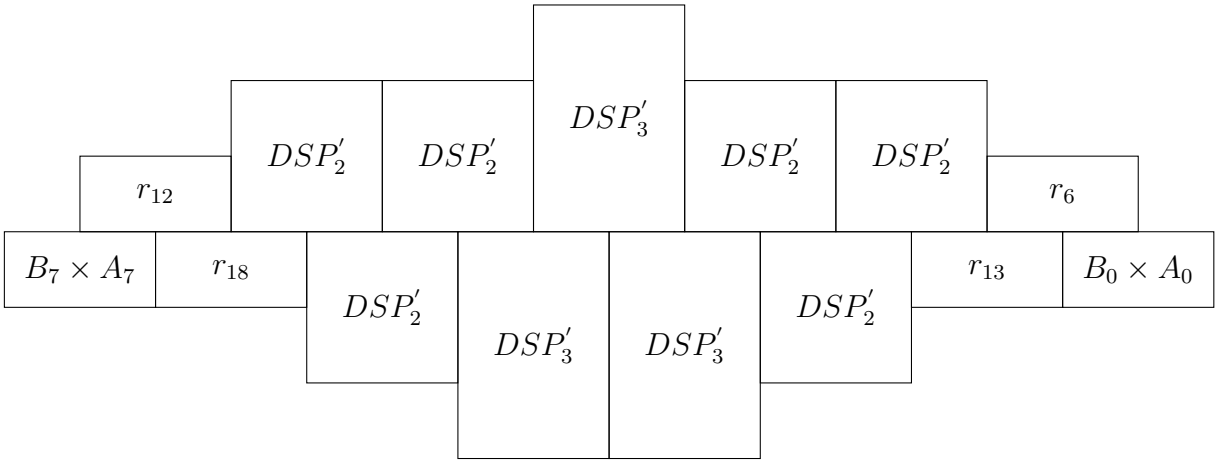

Figure 8.4: Karatsuba diamond results from DSPs

results from the multiplications $B_0 \times A_0$ get added with the less significant bits from the result r_6 with a left shift of w bits for each word in the diamond.

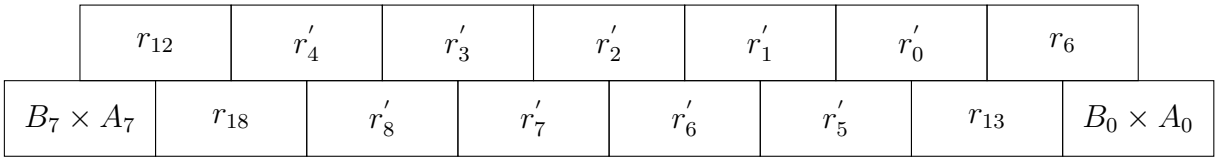

Figure 8.5: Diamond of the second level of DSPs results

Figure 8.6 shows the carry due to the previous additions in red boxes. Once more, adding those carry becomes an easy task because all results because each word has a fixed size of 34-bits, but in the additions, we use 48-bit length additions, so the carry has upto 5 bits length, i.e., our additions has 39 bits length of whom 5-bits belong to the carry.

The Karatsuba multiplier architecture proposed using DSPs shown in figure 8.7

Figure 8.6: Last addition for Karatsuba diamond adder with carry

shows all components developed to implement the algorithm. This way, we can choose where to activate the required registers and the pipeline size. We perform the multiplications of $B_H \times A_H$ and $B_L \times A_L$ in parallel, and the multiplier dedicated has the arithmetic operations as follows: one single multiplication followed by five levels of additions for the diamond as previously presented in this section.

The proposal performs the additions $B_H + B_L$ and $A_H + A_L$ in parallel in only two levels in the first level, adds the words ad pairs, and in the second level, adds the carry. Hence the results become available. The multiplication of both begins with a dedicated multiplier to offer its pipeline for these components.

To perform the subtraction of $(B_H \times A_H)$ and $(B_L \times A_L)$ to get the result of the middle of the final result, we added both results using a similar component as the used in the additions of $B_H + B_L$ and $A_H + A_L$ but dedicated to performing the required additions. The results show the basic operations needed to complete the subtraction. This component performs the following equation 8.4:

$$((B_H + B_L)(A_H + A_L) - (B_H \times A_H) + (B_L \times A_L)) \tag{8.4}$$

Once all results become available, the architecture adds the three intermediate results. In this way, we use the diamond approach, which has three levels performing two additions. For example, this component has the following three inputs.

$$(B_H \times A_H) \tag{8.5}$$

$$(B_L \times A_L)((B_H + B_L)(A_H + A_L) - (B_H \times A_H) + (B_L \times A_L)) \tag{8.6}$$

We already have the Karatsuba algorithm's result; the last adder's output gets divided into two words of 256 bits each (remembering that $|B| = |A| = 256$ bits). The final result has 512 bits, and to begin the reduction process, we deliver it divided into two words. Therefore, the reduction uses DSPs only to perform the following procedure:

$$(B \times A) \bmod 2^{255} - 19 \tag{8.7}$$

We perform the reduction process by multiplying the upper 256 bits of the result with the value in hexadecimal $0x13$, as shown in the following equation 8.8

$$H \times 0x13 \tag{8.8}$$

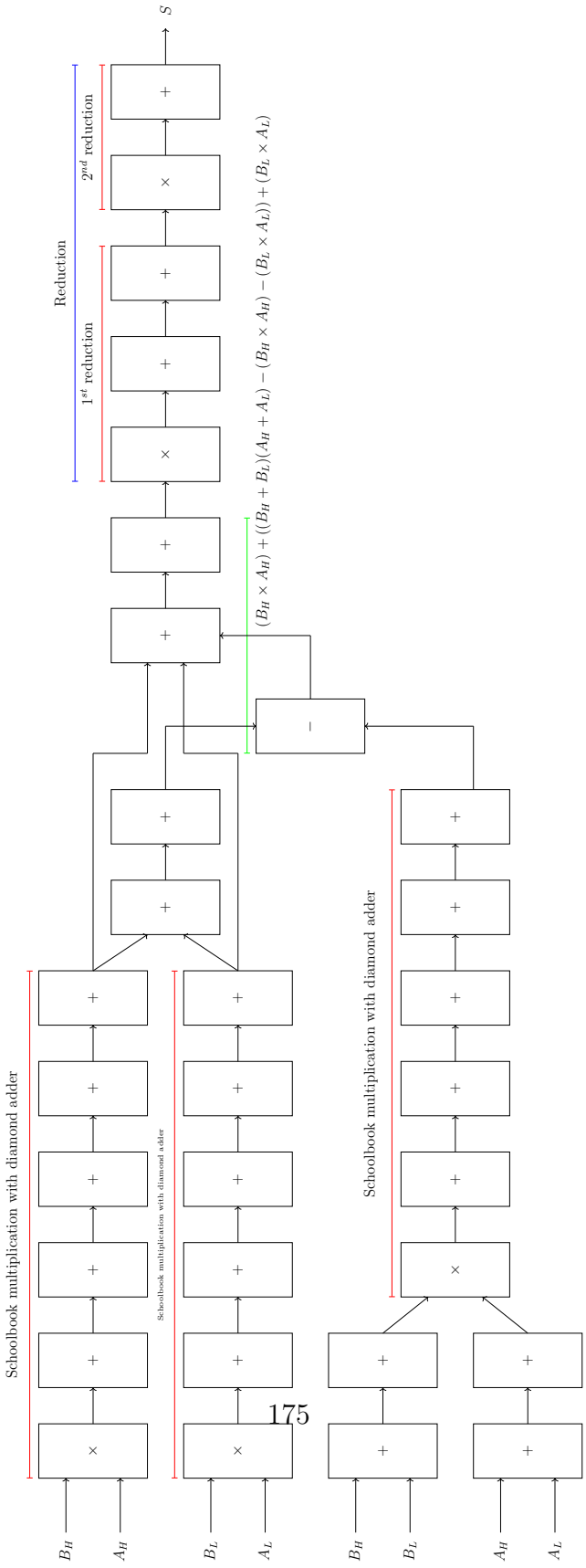

Figure 8.7: Karatsuba multiplier critical path

As previously performed at the beginning of the implementation, we shrink the word H into n smaller words of $w = 17$ bits, each multiplied by $0x13$ (19 decimal). Once the result becomes available, we add it to each word of 17 bits from the lower 256 bits from the Karatsuba multiplication. We show the complete procedure in the following equation 8.9.

$$R_i = L_i + (H_i \times 0x13) \tag{8.9}$$

The architecture in figure 8.7 shows that the multiplier outputs go directly to two additions. The first addition performs $L_i + (H_i \times 0x13)$, and in the following addition, we add all carry results from the addition process. Finally, each carry gets added to the word $i + 1$.

Finally, in the second reduction process, the extra bits from the first reduction and, once again, we perform the same procedure as in the first reduction process but with only one addition rather than two as in the first reduction. The architecture delivers the result via the signal $|S| = 25$ bits. Moreover, the architecture and its components use DSPs only, leaving the LUTs from the FPGA available to perform procedures less expensive furthermore, this architecture has 14 individual arithmetic operations to perform a complete Karatsuba multiplication with reduction.

8.2 Schoolbook proposal

This multiplier becomes a simple way to multiply two integer operands. Also, each component and designed architecture uses DSPs only. The basic multiplier merely multiplies all small words of each operand, i.e., the first step consists of dividing each operand into smaller words with $B = A = 255$-bits into words $w = 17$ bits, splitting the operands results on 15 terms of 17-bit length for both operands B and A. Then we multiply them with the advantage of the process of each multiplication parallelly; hence there are $15 \times 15 = 225$ DSPs deployed in parallel, performing each multiplication in one clock cycle.

Once all multiplications have finished, each partial result becomes available to perform the addition differently; we choose to complete the additions of all partial results by stages using the diamond method. This method includes rearranging all partial results as a diamond and performing fewer additions than the schoolbook method. In our proposal, the diamond has 225 partial results, and there are many levels, and the results need to handle high signal traffic.

We divide all 225 partial results into three smaller diamonds with 75 partial results, and each sees figure 8.8. The three smaller diamonds are processed in parallel to achieve better performance by reducing the number of levels in the diamond. Each diamond has ten levels of partial results, and the partial result from each has a 340-bit length.

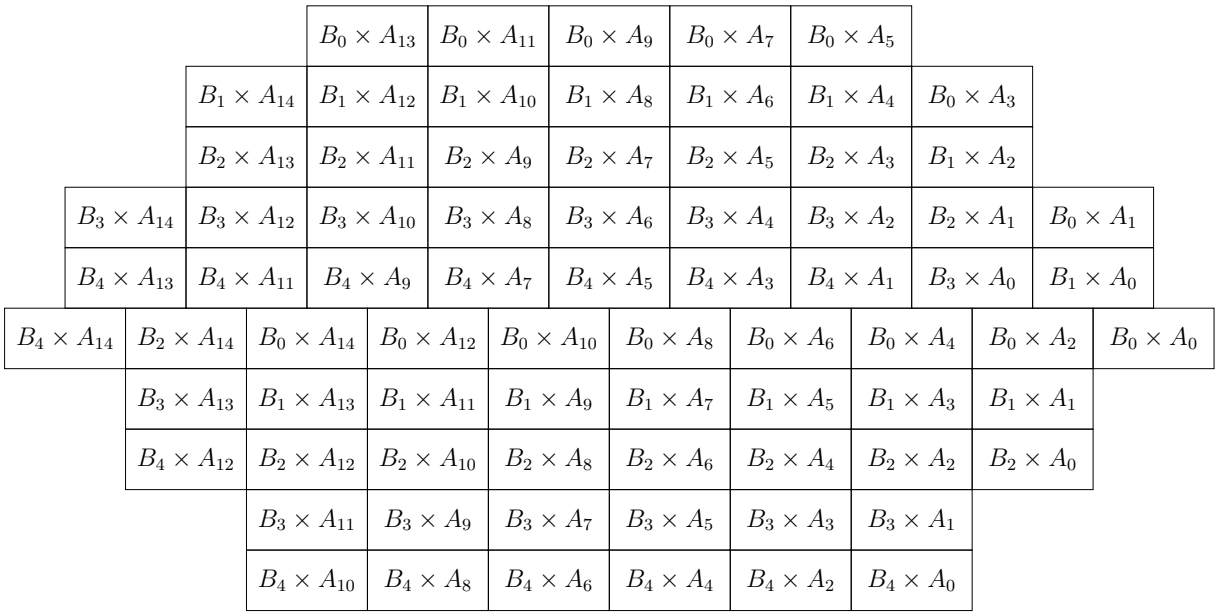

Figure 8.8: Diamond proposal for schoolbook method

Take advantage of the adder implemented in the DSP with 48-bit inputs and up to three input addition simultaneously. Figure 8.9 shows the configuration of each DSP with $DSP_{2,3}$, which can add two or three inputs of 48-bit in one clock cycle, respectively. Here the subindex indicates the number of inputs to add by each DSP. Therefore figure 8.9 shows the DSPs used to add two or three partial results. The number of DSPs needed to perform all additions required by the first stage of additions is equal to 13 DSPs configured to add three operands at a time and 17 DSPs configured to add two operands at a time.

Figure 8.9: Diamond with DSPs

Also, the two operands $B_4 \times A14$, and $B_0 \times A_0$ remain untouched for later processing. Meanwhile, figure 8.10 shows the resultant diamond. This diamond has all results from each addition performed by each DSP from the figure 8.9. Each result becomes arranged as another diamond with a label r_i for each result.

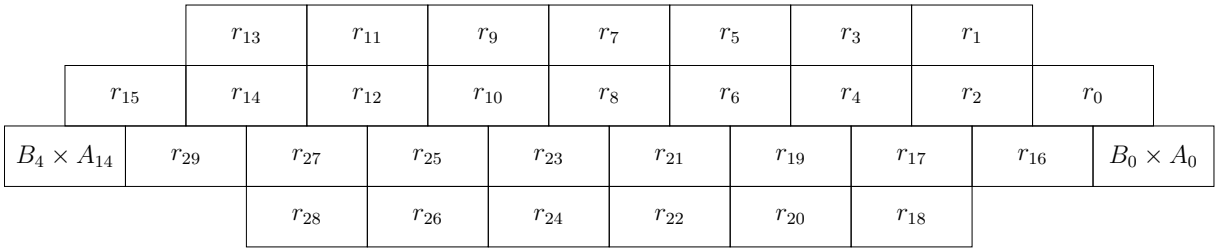

Figure 8.10: Diamond of results from the first level of additions

Following the same structure again, another group of additions is performed by a different set of DSPs presented in figure 8.11. The second level of additions has the label DSP'_2, and as previously mentioned, the subindex indicates the number of operands to add. In this case, each DSP adds only two inputs, leaving the following signals unmodified $B_4 \times A_{14}$, $B_0 \times A_0$, r_0, r_{15}, r_{16} and r_{29}.

In the next clock cycle, the results of the additions become available, and figure 8.12 shows the last level of additions; each result is labeled with r_i' to depict the previously obtained results. Finally, the diamond has two "large operands" to add a third time with all results as depicted in the figure.

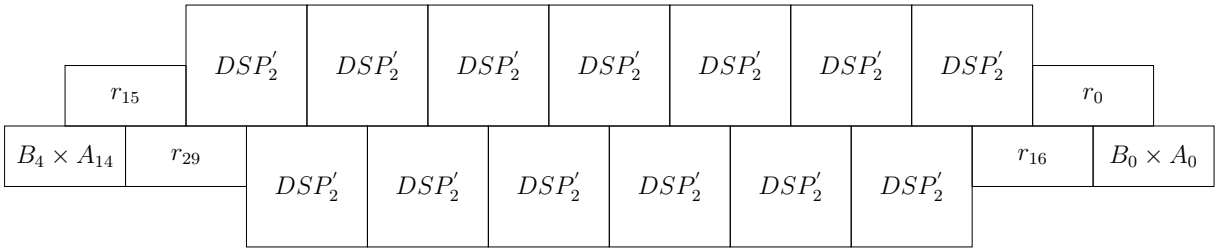

Figure 8.11: Diamond results on the first level of additions on DSPs

Figure 8.13 shows each arithmetic carry available after the three sum levels. To process each carry in parallel is necessary an extra addition to process each arithmetic carry from r_0'' to r_8'', and each will have at least one bit to four bits of length. The sum inputs are the carry from the result r_i'' and the word r_{i+1}''.

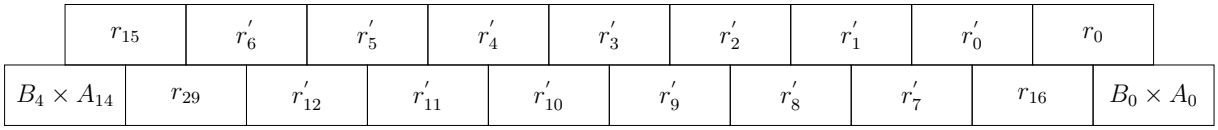

Figure 8.12: Diamond results from the second level of additions

To finish the additions stage, figure 8.14 shows the results from each diamond labeled with Di_j, for $i = 1, 2, 3$ and $j = 0, ...9$ and each intermediate result has w bits length. As the figure shows, each $Di_{,j}$ becomes arranged to create a new diamond with the results of the three smaller diamonds. The addition of all signals has 510 bits length; therefore, it performs the additions in one clock cycle. Consequently, we can divide the result into two large operands labeled (H, L) with $H = L = 255$ bits.

Figure 8.13: Carry addition for the inner diamond in schoolbook

Figure 8.15 shows the architecture of the schoolbook multiplier. The architecture has 12 simple operations conducted in sequential boxes; the first box calculates all words,

i.e., all 15 x 15 individual multiplications of the operands (B, A) in parallel. Once all results become available, the next step consists of sending those results to the three diamonds to add all intermediate results in parallel.

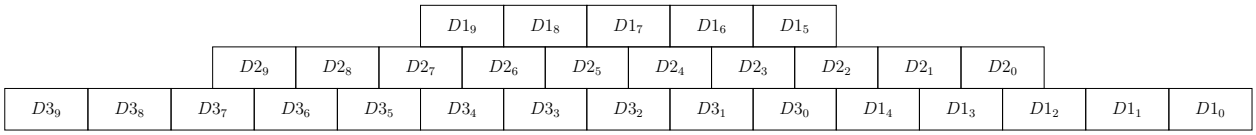

Figure 8.14: Addition of the three inner diamonds to get the final result

Therefore, the figure shows a label with the legend "Diamond_inner". This diamond represents the data path used in each smaller diamond processed in parallel. Once the three diamonds finish, each result goes to the next stage of additions, which adds the three intermediate results from the "Diamond_inner" with only two additions.

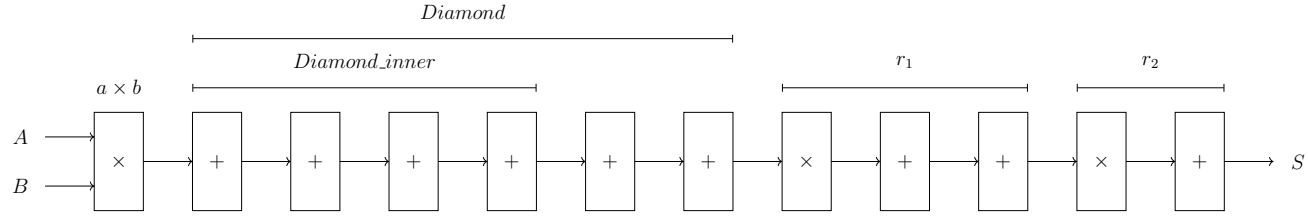

Figure 8.15: Critical path schoolbook method

As the figure 8.15 shows the operations with the label "Diamond," we use only six levels of additions to process all the 225 individual results and get the result of performing only the multiplication of $B \times A$.

Finally, with the result of the multiplication of $B \times A$, the next step consists in performing a fast reduction to obtain a result of 255 bits rather than a 510 bits length result. As shown in figure 8.15, the reduction operation has two stages. The first stage has three functions: multiplication and two additions with the label "r_1".

This stage picks up the 510-bit result from the diamond divided as two inputs of 255-bits; the two operands have the label H_i and L_i to recognize which belongs to the most significant part and the less significant part, respectively. The reduction process consists of multiplying and adding both results. The first step divides both operands (H, L) into smaller words, in this case, $w = 17$-bits, then begins the multiplication of the H_i words with the integer represented as $0x13$ in hexadecimal. Once

multiplied, begins the process of adding each result to its corresponding lower word L_i; the operation presented in 8.9 describes the general process.

As previously mentioned, multiplying and adding returns a result with a carry which needs another reduction to obtain the result. Finally, the second stage of reduction r_2 gets the most significant bits over the first 255-bits and divides them by words, then multiply them by $0x13$ again and adds them to each of the L_i of the second stage of the reduction. When the operation $B \times A \bmod 2^{255} - 19$ finishes, the circuit delivers the result via the signal "S" with 255 bits of length.

8.3 RNS

Figure 8.16 shows the components used in an RNS architecture among all the operations needed to perform the multiplication with a reduction in RNS. For example, the multiplier designed for the RNS Jeljeli algorithm presented in chapter 7 algorithm 19. The figure has 31 individual operations. However, the sequence follows the next steps even with that division level.

1. Compute the value of γ in step 6 of the algorithm. To compute this step, a component specially designed to calculate $d_j = |b_j \times a_j|_{\mu_j}$ once done, proceed to multiply the result with M_j^{-1} with a reduction stage in this process. Figure 8.16 shows this component presented in the first row of the figure has two multiplications and five sets of reduction denoted by the r_i.

2. Once calculated γ begins processing the matrix from step 11 in the algorithm, we can process all the matrix in parallel. Also, it has DSPs dedicated to having the freedom to enable or disable the internal registers as required for the pipeline. The component designed for this stage has one multiplication stage and five sets of reduction at each row of the matrix needs. Therefore an inverted addition tree with three steps. Once processed the data a signal label with z_j has the data vector, which needs a subtraction of α.

   Therefore at the same time we process the matrix, step 12 of the algorithm gets processed too. This step calculated the α needed to look up the table with the pre-computation values to get the correct values necessary for the subtraction in step 13 of the algorithm.

3. Finally in, step 13 executes the operation $z_j - \alpha$ with its respective reduction, therefore, α aims at a memory address. The memory already has the

precomputed values for all possible α needed by the prime number, and each query consumes only one clock cycle. When the subtraction "Last_Sub" has finished, the output from the component "+" has the result of the procedure $B \times A \bmod P$ in the RNS domain.

The RNS architecture has 33 words for each operand and an RNS base with 33 co-prime numbers. Also, have a high resource consumption against the other two implementations, with a lower work frequency and pipeline bigger than the other two proposals. The reasons mentioned before are the main reason to dismiss the RNS proposal and architecture. Therefore it does not achieve high speed, and the only way to implement the architecture is to use FPGAs with a high quantity of resources, which are prohibitive costs. The increased resources make it impossible to compare the two previous architectures fairly.

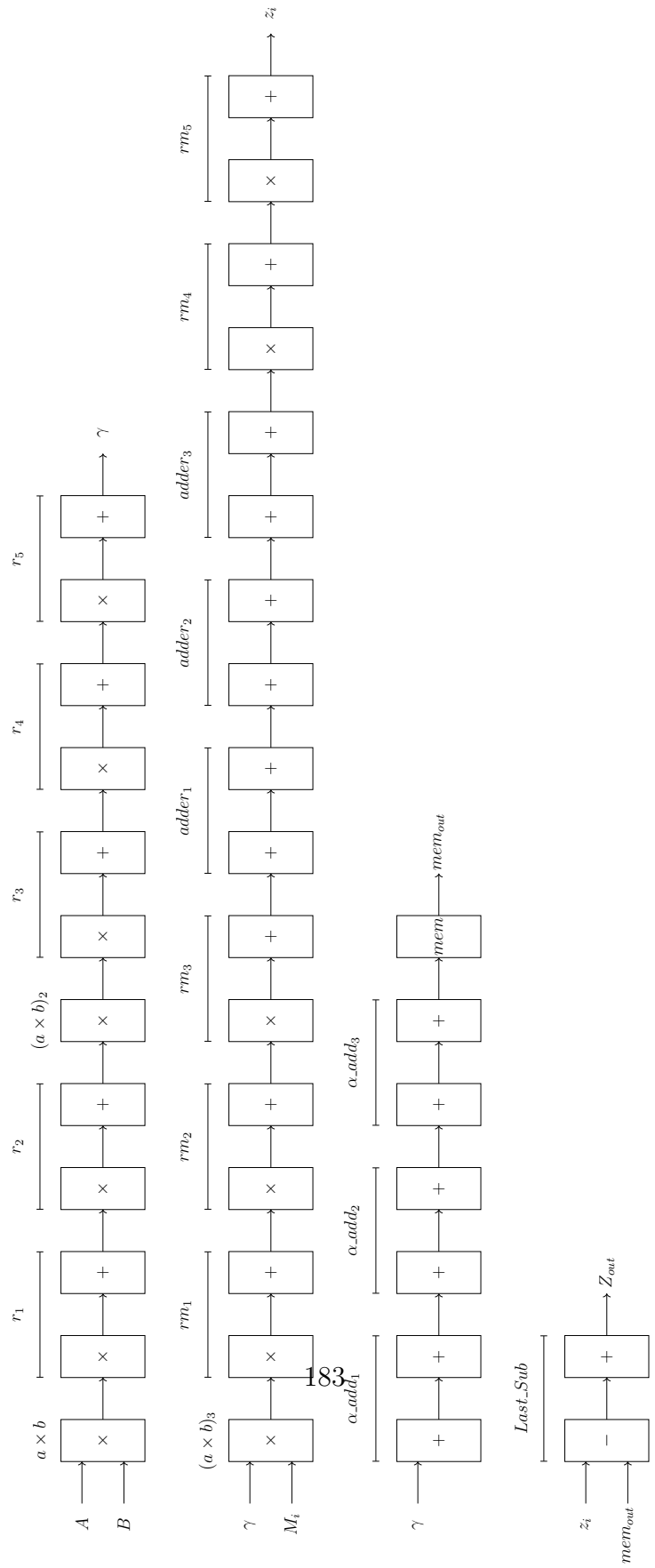

Figure 8.16: Critical path RNS proposal

8.4 Results

We show the results from the proposed architectures and their implementation for the Karatsuba and Schoolbook algorithms. The following tables show the resources used for each component's implementation in three different FPGAs that belong to the Xilinx *Virtex-7* family and its variants *ultrascale* and *ultrascale+*.

The main difference between the three FPGAs is the scope to use, i.e., Virtex-7 has high performance vs. the other families as Spartan because its oriented to portable radars, networking, and *ASIC* (Application-Specific Integrated Circuit) prototyping. The Virtex-7 *ultrascale* focus on enterprise industry applications, usually used in large-scale processes, can emulate and implement ASIC designs, and performs better than the Virtex-7 family. Finally, the Virtex-7 *ultrascale+* brings the highest clock speeds since it has dedicated routing between registers achieving higher clock frequencies. This family focuses on high-performance computing.

The three FPGAs used are:

- Virtex-7 : xc7v585tffg1761-3 with the following specs
  - 364200 LUTs.
  - 1260 DSPs.
  - 850 user I/O.
- Virtex-7 *ultrascale* : xcvu080-ffva2104-3-e with the following specs
  - 445712 LUTs.
  - 672 DSPs.
  - 832 user I/O.
- Virtex-7 *ultrascale+* : xcvu5p-flva2104-3-e with the following specs
  - 600577 LUTs.
  - 3474 DSPs.
  - 832 user I/O.

Tables 8.1 and 8.2 show the resources used by the Karatsuba and Schoolbook algorithms. We can see that both proposals have similar resource consumption, and both

designs have a 0 Slices and LUTs consumption. Hence the main difference between both architectures is the critical path from Karatsuba against the SchoolBook. We can see this in figures 8.7 and 8.15, respectively.

Device	DSPs	FF	LUTS	Slices	Freq	Component
xc7v585tffg1761-3 VIRTEX-7	450	0	0	0	43.78 MHz	Karatsuba
xcvu080-ffva2104-3-e VIRTEX-7 ultrascale	450	0	0	0	47.619	Karatsuba
xcvu5p-flva2104-3-e VIRTEX-7 ultrascale+	450	0	0	0	82.645	Karatsuba

Table 8.1: Karatsuba multiplication with reduction, resources, and speed

We have a 37 DSPs difference in consumption between both proposals. Karatuba consumes fewer DSPs but achieves lower speed and has a critical path more considerable than the schoolbook architecture. This second proposal has a smaller critical path and higher performance.

Device	DSPs	FF	LUTS	Slices	Freq MHz	Component
xc7v585tffg1761-3 VIRTEX-7	487	0	0	0	90.909	SchoolBook
xcvu080-ffva2104-3-e VIRTEX-7 ultrascale	487	0	0	0	106.383	SchoolBook
xcvu5p-flva2104-3-e VIRTEX-7 ultrascale+	487	0	0	0	197.044	SchoolBook

Table 8.2: SchoolBook multiplication with reduction, resources, and speed

Comparing the three FPGAs, we can see that the Virtex-7 has a performance of 90 MHz for the SchoolBook implementation, but the Karatsuba architecture only achieves almost half of the speed. On the other hand, in the Virtex-7 ultra-scale, we can see that the performance is nearly the same and only gain a few MHz in clock frequency for both implementations.

Finally, the Virtex-7 ultrascale+, which aims for high-performance computing, delivers almost twice the speed of a regular Virtex-7. In addition, the tables show that the three architectures have the same resource consumption for the three FPGAs but with different frequencies.

Device	DSPs	FF	LUTS	Slices	Freq MHz	Component
xc7v585tffg1761-3 VIRTEX-7	21	0	0	0	133.333	Adder
xcvu080-ffva2104-3-e VIRTEX-7 ultrascale	21	0	0	0	210.526	Adder
xcvu5p-flva2104-3-e VIRTEX-7 ultrascale+	21	0	0	0	322.581	Adder

Table 8.3: Adder with reduction, resources, and speed

Table 8.3 shows the required resources and speed achieved by the same three FPGAs, but the component implemented is an adder with a modular reduction with two registers. As shown in the table, this adder requires 21 DSPs and, in the Virtex-7, achieves a speed of 133 MHz. The Virtex-7 ultra-scale delivers twice the speed, and the Virtex-7 ultrascale+ delivers almost 3× of the speed of the regular Virtex-7.

The last component of this architecture is subtraction with modular reduction. Table 8.4 shows the results of the three implementations. It requires 12 DSPs and achieves a 108 MHz clock frequency on the Virtex-7. Implementing the Virtex-7 ultrascale+ achieves more than twice the regular Virtex-7. Therefore, this component uses only one LUT, also shown as a Slice by the Xilinx tool. The reason is to broadcast a carry with a one-bit *XOR* gate to deliver the correct result.

Device	DSPs	FF	LUTS	Slices	Freq MHz	Component
xc7v585tffg1761-3 VIRTEX-7	12	0	1	1	108.108	Subtraction
xcvu080-ffva2104-3-e VIRTEX-7 ultrascale	12	0	1	1	145.455	Subtraction
xcvu5p-flva2104-3-e VIRTEX-7 ultrascale+	12	0	1	1	216.216	Subtraction

Table 8.4: Subtraction with reduction, resources, and speed

The three architectures proposed and each component designed uses DSPs only, so each delivers different clock speeds for the FPGAs. Furthermore, even when the FPGAs belong to the same family Virtex-7, the manufacturing company implements restrictions on the design and speed for each subcategory in the same family.

8.5 Summary

This chapter shows an alternative multiplier with reduction using three different algorithms, therefore, shows extra essential components required to perform integer arithmetic with individual results for each proposal. The three architectures demonstrated in this chapter fall short of a control unit.

Part IV

Summary

Chapter 9

Conclusions

This thesis shows the relationship between software and hardware and how a software implementation can lead to different hardware architectures. Therefore, we show that hardware-oriented proposals can execute faster on software with constrained resources. As a result, both focuses can perform better in their environment, but they can run into different architectures and perform as quickly as the target device runs.

The following conclusion comprehends each work of this thesis individually.

9.1 Lightweight authenticated encryption with associated data

The Lightweight algorithms implemented show that the standardization process and the devices with constrained resources can offer various security services. However, in some cases, the development of a dedicated architecture faces restrictions like resource and power consumption due to the target embedded systems usually running with a battery. Therefore, lightweight cryptography still has a lot of research to fulfill the security needs of new lightweight applications and constrained devices.

9.2 Speedy Block cipher on ARM-M4 with Bitslice

Speedy implementation in software shows the possibility of using different techniques to implement a hardware-oriented algorithm in software capable of running on constrained devices. Also, with acceptable performance for encryption only, using intrinsic instructions from the target device can enhance the performance. However,

a disadvantage of this solution is that the implementation only runs with devices that share the same instruction set.

9.3 A DSP-based FPGA design and implementation of a fast RNS multiplier

The proposed architecture performs well when it uses a large amount of DPS, but it focuses on a general-purpose multiplier, i.e., it can receive any prime number of 512 bits. Hence, the reduction process becomes large and inefficient when using some primes. Therefore, the basic units used for the multiplication with reduction among the inverted adder tree create a significant amount of data traffic into the FPGA, reducing the speed or the implementation in hardware.

9.4 Hardware accelerator for the elliptic curve ECC25519

The multipliers proposed for the ECC 2255-19 demonstrate when large amounts of unique resources are available as multipliers and adders embedded in DSPs, allowing the designer to combine both to create an architecture based on fundamentals with a performance as good, even better than newer concepts. This result means that depending on the target, the designer can select a traditional algorithm or more recent proposals considering the device and its resources.

Chapter 10

Future Work

This section presents the future work available for each of our research works. The following list shows the possible future research valuable to the cryptography area.

- Lightweight authenticated encryption with associated data:
  - To find applications with reduced security needs for participant algorithms from NIST lightweight standardization process from the second and final rounds.
  - Side channel attacks can help the implementers to design countermeasures for the algorithms implemented, like constant time.
  - The bitslice approach can enhance the performance of lightweight algorithms.
- Speedy Block cipher on ARM-M4 with Bitslice:
  - Research side channel attacks vulnerabilities to hardware proposal.
  - To implement constant time to Speedy algorithm.
  - Improve the Keyscheduling.
- A DSP-based FPGA design and implementation of a fast RNS multiplier:
  - Search a basis from words smaller than 34 bits.
  - Implement a “full size” multiplier into newer FPGAs to see their behavior.
  - Implement countermeasures against side-channel attacks.
- Hardware accelerator for the elliptic curve ECC25519:

- Finalize the control unit for the multiplier.
- Implement a complete scheme like ECDSA.
- Implement countermeasures against side-channel attacks.

This future work can change with time, but this is future work with a high probability of success.

Bibliography

[1] Vipindev Adat and B. B. Gupta. Security in Internet of Things: issues, challenges, taxonomy, and architecture, 2017.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[3] Samuel Antão, Jean Claude Bajard, and Leonel Sousa. RNS-based elliptic curve point multiplication for massive parallel architectures. *Computer Journal*, 2012.

[4] Jean Claude Bajard, Julien Eynard, and Nabil Merkiche. Montgomery reduction within the context of residue number system arithmetic. *Journal of Cryptographic Engineering*, 2018.

[5] Jean Claude Bajard and Laurent Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 2004.

[6] Jean Claude Bajard and Nabil Merkiche. Double level montgomery cox-rower architecture, new bounds. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[7] Gustavo Banegas, Ricardo Custodio, and Daniel Panario. A new class of irreducible pentanomials for polynomial based multipliers in binary fields. Cryptology ePrint Archive, Paper 2018/554, 2018. https://eprint.iacr.org/2018/554.

[8] Subhadeep Banik, Andrey Bogdanov, Atul Luykx, and Elmar Tischhauser. SUNDAE: small universal deterministic authenticated encryption for the internet of things. *IACR Trans. Symmetric Cryptol.*, 2018(3):1–35, 2018.

[9] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: A small present. *Cryptographic Hardware and Embedded Systems-CHES*, pages 25–28, 2017.

[10] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017.

[11] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 175:1–175:6. ACM, 2015.

[12] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Annual international cryptology conference*, pages 1–15. Springer, 1996.

[13] Daniel Bernstein. Multidigit modular multiplication with the explicit chinese remainder theorem, 2001.

[14] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[15] René Beuchat, Florian Depraz, Andrea Guerrieri, and Kashani Sahand. *Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers*. arm Education Media, 2021.

[16] Arghya Bhattacharjee1, Eik List, Cuauhtemoc Mancillas López, and Mridul Nandi. The Oribatida Family of Lightweight Authenticated Encryption Schemes, 2019. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/oribatida-spec-round2.pdf.

[17] Karim Bigou and Arnaud Tisserand. Single base modular multiplication for efficient hardware RNS implementations of ECC. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[18] Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *Fast Software Encryption*, pages 260–272, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[19] Dobre Blazhevski, Adrijan Bozhinovski, Biljana Stojchevska, and Veno Pachovski. Modes of operation of the aes algorithm. 2013.

[20] A Bogdanov, L R Knudsen, G Leander, C Paar, A Poschmann, M.J.B Robshaw, Y Seurin, and C Vikkelsoe. PRESENT : An Ultra-Lightweight Block Cipher. *Springer Berlin Heidelberg*, pages 450–466, 2007.

[21] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçın. Prince - a low-latency block cipher for pervasive computing applications (full version). Cryptology ePrint Archive, Paper 2012/529, 2012. https://eprint.iacr.org/2012/529.

[22] Joppe W. Bos and Simon J. Friedberger. Faster modular arithmetic for isogeny based crypto on embedded devices. Cryptology ePrint Archive, Paper 2018/792, 2018. https://eprint.iacr.org/2018/792.

[23] Christina Boura, Nicolas David, Rachelle Heim Boissier, and Maria Naya-Plasencia. Better steady than speedy: Full break of speedy-7-192. Cryptology ePrint Archive, Paper 2022/1351, 2022. https://eprint.iacr.org/2022/1351.

[24] S. Brown. Fpga architectural research: a survey. *IEEE Design & Test of Computers*, 13(4):9–15, 1996.

[25] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. Elastic-tweak: A framework for short tweak tweakable block cipher. Cryptology ePrint Archive, Report 2019/440, 2019. https://eprint.iacr.org/2019/440.

[26] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. Estate. Lightweight Cryptography, Round 2 Candidates, 2019. https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/estate-authenticated-encryption-mode-lwc2019.pdf.

[27] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas Lopez, Mridul Nandi, and Yu Sasaki. Lotus-aead and locus-aead. Lightweight Cryptography, Round 2 Candidates, 2019. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/lotus-aead-and-locus-aead-spec.pdf.

[28] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas-López, Mridul Nandi, and Yu Sasaki. INT-RUP secure lightweight parallel AE modes. *IACR Trans. Symmetric Cryptol.*, 2019(4):81–118, 2019. https://dblp.org/rec/journals/tosc/ChakrabortiDJMN19.bib.

[29] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, Cuauhtemoc Mancillas-López, Mridul Nandi, and Yu Sasaki. ESTATE: A lightweight and low energy authenticated encryption mode. *IACR Trans. Symmetric Cryptol.*, 2020(S1):350–389, 2020. https://dblp.org/rec/journals/tosc/ChakrabortiDJMN20.bib.

[30] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):218–241, 2018.

[31] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2018/931, 2018. https://eprint.iacr.org/2018/931.

[32] Ray C.C. Cheung, Sylvain Duquesne, Junfeng Fan, Nicolas Guillermin, Ingrid Verbauwhede, and Gavin Xiaoxu Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011.

[33] P. Chow, Soon Ong Seo, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja. The design of an sram-based field-programmable gate array. i. architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):191–197, 1999.

[34] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of elliptic and hyperelliptic curve cryptography*. CRC press, 2005.

[35] Marius Cornea. Intel avx-512 instructions and their use in the implementation of math functions. *Intel Corporation*, pages 1–20, 2015.

[36] J. M. Couveignes. Computing a square root for the number field sieve. In Lenstra H. W. Lenstra A. K., editor, *The development of the number field sieve*, Lecture Notes in Mathematics, pages 90–97, 1993.

[37] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[38] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[39] Zhibin Dai and D.K. Banerji. Routability prediction for field programmable gate arrays with a routing hierarchy. In *16th International Conference on VLSI Design, 2003. Proceedings.*, pages 85–90, 2003.

[40] Nir Drucker and Shay Gueron. Fast modular squaring with avx512ifma. Cryptology ePrint Archive, Paper 2018/335, 2018. https://eprint.iacr.org/2018/335.

[41] Morris Dworkin. Recommendations for block cipher modes of operation, methods and techniques, 2001.

[42] Morris J Dworkin, Elaine B Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. Advanced encryption standard (aes). 2001.

[43] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. On optimal hyperuniversal and rearrangeable switch box designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(12):1637–1649, 2003.

[44] Hongbing Fan, Jiping Liu, Yu-Liang Wu, and Chak-Chung Cheung. The exact channel density and compound design for generic universal switch blocks. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2):19–es, apr 2007.

[45] Martin Feldhofer and Christian Rechberger. A case against currently used hash functions in rfid protocols. In *On the move to meaningful internet systems 2006: OTM 2006 workshops*, pages 372–381. Springer, 2006.

[46] N Fips. Announcing the ADVANCED ENCRYPTION STANDARD (AES). *Byte*, 2009(12):8–12, 2001.

[47] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed rsa key generation for semi-honest and malicious adversaries. Cryptology ePrint Archive, Paper 2018/577, 2018. https://eprint.iacr.org/2018/577.

[48] Konstantinos Fysarakis, George Hatzivasilis, Ioannis Askoxylakis, and Charalampos Manifavas. Rt-spdm: real-time security, privacy and dependability management of heterogeneous systems. In *International Conference on Human Aspects of Information Security, Privacy, and Trust*, pages 619–630. Springer, 2015.

[49] Konstantinos Fysarakis, George Hatzivasilis, Charalampos Manifavas, and Ioannis Papaefstathiou. Rtvmf: A secure real-time vehicle management framework. *IEEE Pervasive Computing*, 15(1):22–30, 2016.

[50] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. Fair and comprehensive methodology for comparing hardware performance of fourteen round two sha-3 candidates using fpgas. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 264–278. Springer, 2010.

[51] Kris Gaj, Ekawat Homsirikamol, Marcin Rogawski, Rabia Shahid, and Malik Umar Sharif. Comprehensive evaluation of high-speed and medium-speed implementations of five sha-3 finalists using xilinx and altera fpgas. Cryptology ePrint Archive, Paper 2012/368, 2012. https://eprint.iacr.org/2012/368.

[52] Filippo Gandino, Fabrizio Lamberti, Gianluca Paravati, Jean Claude Bajard, and Paolo Montuschi. An algorithmic and architectural study on montgomery exponentiation in RNS. *IEEE Transactions on Computers*, 2012.

[53] Benoît Gérard, Jean Gabriel Kammerer, and Nabil Merkiche. Contributions to the Design of Residue Number System Architectures. In *Proceedings - Symposium on Computer Arithmetic*, 2015.

[54] Nicolas Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over Fp. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010.

[55] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[56] George Hatzivasilis, Ioannis Papaefstathiou, Charalampos Manifavas, and Ioannis Askoxylakis. Lightweight password hashing scheme for embedded systems. In *IFIP International Conference on Information Security Theory and Practice*, pages 260–270. Springer, 2015.

[57] Michael Healy, Thomas Newe, and Elfed Lewis. Wireless sensor node hardware: A review. In *Proceedings of IEEE Sensors*, pages 621–624, 2008.

[58] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Comparing hardware performance of fourteen round two sha-3 candidates using fpgas. Cryptology ePrint Archive, Paper 2010/445, 2010. https://eprint.iacr.org/2010/445.

[59] Malik Imran, Muhammad Rashid, and Imran Shafi. Lopez dahab based elliptic crypto processor (ecp) over gf(2163) for low-area applications on fpga. In *2018 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6, 2018.

[60] Dworkin Morris J. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, NIST, Gaithersburg, MD, United States, 2001.

[61] Dworkin Morris J. Sp 800-38b. recommendation for block cipher modes of operation: The cmac mode for authentication. Technical report, NIST, Gaithersburg, MD, United States, 2005.

[62] Shruti Jaiswal and Daya Gupta. *Security Requirements for Internet of Things (IoT)*, pages 419–427. Springer Singapore, Singapore, 2017.

[63] Hamza Jeljeli. *Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis*. PhD thesis, Inria Nancy, Grand Est, LORIA ALGO Department of Algorithms, Computation, Image and Geometry, available at: https://hal.inria.fr/tel-01178931, 2015.

[64] Hamza Jeljeli. Accelerating iterative SpMV for the discrete logarithm problem using GPUs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[65] Qi Jing, Athanasios V. Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. Security of the internet of things: perspectives and challenges. *Wireless Networks*, 20(8):2481–2501, Nov 2014.

[66] Daemen Joan and Rijmen Vincent. The advanced encryption standard, 2001.

[67] Kimmo Järvinen. Optimized fpga-based elliptic curve cryptography processor for high-speed applications. *Integration*, 44(4):270–279, 2011. Hardware Architectures for Algebra, Cryptology and Number Theory.

[68] Christoforos Kachris, Stephan Wong, and Stamatis Vassiliadis. Design and performance evaluation of an adaptive fpga for network applications. *Microelectronics Journal*, 40(7):1103–1110, 2009. Mixed-Technology Testing Rapid System Prototyping.

[69] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, and Kris Gaj. Hardware api for lightweight cryptography. Tech. Report Oct 2019, 2019. https://cryptography.gmu.edu/athena/LWC/LWC_HW_API.pdf.

[70] Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, and Kris Gaj. Hardware api for lightweight cryptography. *URL https://cryptography. gmu. edu/athena/index. php*, pages 1–26, 2019.

[71] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[72] Anatolii Karatsuba and Yuri Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[73] Anatolii Alexeevich Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 211:169–183, 1995.

[74] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2000.

[75] Shinichi Kawamura, Yuichi Komano, Hideo Shimizu, and Tomoko Yonemura. RNS montgomery reduction algorithms using quadratic residuosity. *Journal of Cryptographic Engineering*, September 2018.

[76] Hyunjun Kim, Kyungbae Jang, Gyeongju Song, Minjoo Sim, Siwoo Eum, Hyunji Kim, Hyeokdong Kwon, Wai-Kong Lee, and Hwajeong Seo. Speedy on cortex–m3: Efficient software implementation of speedy on arm cortex–m3. Cryptology ePrint Archive, Paper 2021/1212, 2021. https://eprint.iacr.org/2021/1212.

[77] Mateusz Komorkiewicz, Krzysztof Turek, Pawel Skruch, Tomasz Kryjak, and Marek Gorgon. Fpga-based hardware-in-the-loop environment using video injection concept for camera-based systems in automotive applications. In *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 183–190, 2016.

[78] Thanikodi Manoj Kumar, Kasarla Satish Reddy, Stefano Rinaldi, Bidare Divakarachari Parameshachari, and Kavitha Arunachalam. A low area high speed fpga implementation of aes architecture for cryptography application. *Electronics*, 10(16), 2021.

[79] Yen-Tai Lai and Ping-Tsung Wang. Hierarchical interconnection structures for field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(2):186–196, 1997.

[80] Gregor Leander, Thorben Moos, Amir Moradi, and Shahram Rasoolzadeh. The speedy family of block ciphers - engineering an ultra low-latency cipher from gate level for secure processor architectures. Cryptology ePrint Archive, Paper 2021/960, 2021. https://eprint.iacr.org/2021/960.

[81] Chris Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23, 2011.

[82] Javier Lopez, Rodrigo Roman, and Cristina Alcaraz. Analysis of Security Threats , Requirements , Technologies and Standards in Wireless Sensor Networks. *Foundations of Security Analysis and Design V*, 5705:289–338, 2009.

[83] Lauren May, Lyta Penna, and Andrew Clark. An implementation of bitsliced des on the pentium mmxtm processor. In E. P. Dawson, A. Clark, and Colin Boyd, editors, *Information Security and Privacy*, pages 112–122, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[84] Kerry McKay, Lawrence Bassham, Meltem Sönmez Turan, and Nicky Mouha. Reposrt on lightweight cryptography. 2017.

[85] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[86] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[87] Kazuhiko Minematsu. Aes-otr v3.1, cited September 2020.

[88] Bassam Jamil Mohd, Thaier Hayajneh, Zaid Abu Khalaf, Ahmad Yousef, and Khalil Mustafa. Modeling and optimization of the lightweight hight block cipher design with fpga implementation. *Security and Communication Networks*, 9(13):2200–2216, 2016.

[89] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[90] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 1985.

[91] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[92] Peter L. Montgomery and Robert D. Silverman. An fft extension to the $p-1$ factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990.

[93] Peter Lawrence Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization*. PhD thesis, University of California at Los Angeles, USA, 1992.

[94] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.

[95] Ivan Müller, Edison Pignaton De Freitas, Altamiro Amadeu Susin, and Carlos Eduardo Pereira. Namimote: A low-cost sensor node for wireless sensor networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7469 LNCS, pages 391–400, 2012.

[96] Hanae Nozaki, Masahiko Motoyama, Atsushi Shimbo, and Shinichi Kawamura. Implementation of RSA algorithm based on RNS montgomery multiplication. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001.

[97] Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, Nareli Cruz Cortés, and Francisco Rodríguez-Henríquez. Implementation of RSA signatures on GPU and CPU architectures. *IEEE Access*, 8:9928–9941, 2020.

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

[99] J. Britto Pari and D. Vaithiyanathan. An optimized fpga implementation of dct architecture for image and video processing applications. In *2019 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET)*, pages 186–191, 2019.

[100] Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, 2004.

[101] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, May 1995.

[102] Karl C. Posch and Reinhard Posch. Modulo reduction in residue number systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):449–454, 1995.

[103] Axel York Poschmann. Lightweight cryptography: cryptographic engineering for a pervasive world. In *PH. D. THESIS*. Citeseer, 2009.

[104] FIPS Pub. Data encryption standard (des). *FIPS PUB*, pages 46–3, 1999.

[105] Behnaz Rezvani, Flora Coleman, Sachin Sachin, and William Diehl. Hardware implementations of nist lightweight cryptographic candidates: A first look. Cryptology ePrint Archive, Report 2019/824, 2019. https://eprint.iacr.org/2019/824.

[106] Juan J. Rodríguez-Andina, María D. Valdés-Peña, and María J. Moure. Advanced features and industrial applications of fpgas—a review. *IEEE Transactions on Industrial Informatics*, 11(4):853–864, 2015.

[107] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[108] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[109] Gaël Rouvroy, François-Xavier Standaert, Jean-Jacques Quisquater, and Jean-Didier Legat. Compact and efficient encryption/decryption module for FPGA implementation of the AES rijndael very well suited for small embedded applications. In *International Conference on Information Technology: Coding and Computing (ITCC'04), Volume 2, April 5-7, 2004, Las Vegas, Nevada, USA*, pages 583–587. IEEE Computer Society, 2004.

[110] Shreekant Sajjanar, Suraj K. Mankani, Prasad R. Dongrekar, Naman S. Kumar, Mohana, and H.V. Ravish Aradhya. Implementation of real time moving object detection and tracking on fpga for video surveillance applications. In *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pages 289–295, 2016.

[111] D. M. Schinianakis, A. P. Kakarountas, and T. Stouraitis. A new approach to Elliptic Curve Cryptography: An RNS architecture. In *Proceedings of the Mediterranean Electrotechnical Conference - MELECON*, 2006.

[112] Dimitrios M. Schinianakis, Apostolos P. Fournaris, Harris E. Michail, Atharoutas P. Kakarountas, and Thanos Stouraitis. An RNS implementation of an Fp elliptic curve point multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2009.

[113] J. Schwemmlein, K. C. Posch, and R. Posch. RNS-modulo reduction upon a restricted base value set and its applicability to RSA cryptography. *Computers and Security*, 1998.

[114] Gueron Shay, Jha Ashwin, and Nandi Mridul. COMET : COunter Mode Encryption whith authentication Tag, 2019. https://csrc.nist.gov/Projects/lightweight-cryptography/round-2-candidates.

[115] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Çağdaş Çalık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. Status report on the second round of the nist lightweight cryptography standardization process, 2021.

[116] Frank Stajano. Security for ubiquitous computing. In *International Conference on Information Security and Cryptology*, pages 2–2. Springer, 2004.

[117] STMicroelectronics. Rm0351 reference manual stm32l4x5 and stm32l4x6 advanced arm-based 32-bit mcus, 2022.

[118] STMicroelectronics. Stm32l4a6xg ultra-low-power arm cortex-m4 32-bit mcu+fpu, 100dmips, 1mb flash, 320kb sram, usb otg fs, audio, aes+hash, ext. smps, 2022.

[119] STMicroelectronics. Um1924 user manual stm32 crypto library, 2022.

[120] STMicroelectronics. Um2179 user manual stm32 nucleo-144 boards, 2022.

[121] Hui Suo, Jiafu Wan, Caifeng Zou, and Jianqi Liu. Security in the internet of things: A review. In *Proceedings - 2012 International Conference on Computer Science and Electronics Engineering, ICCSEE 2012*, volume 3, pages 648–651, 2012.

[122] Ruhma Tahir, Muhammad Younas Javed, and Ahmad Raza Cheema. Rabbit-mac: Lightweight authenticated encryption in wireless sensor networks. In *Information and Automation, 2008. ICIA 2008. International Conference on*, pages 573–577. IEEE, 2008.

[123] Brunel Happi Tietche, Olivier Romain, Bruce Denby, and Francois De Dieuleveult. Fpga-based simultaneous multichannel fm broadcast receiver for audio indexing applications in consumer electronics scenarios. *IEEE Transactions on Consumer Electronics*, 58(4):1153–1161, 2012.

[124] Meltem Sönmez Turan, Kerry A McKay, Çagdas Çalik, Donghoon Chang, Lawrence Bassham, et al. Status report on the first round of the nist lightweight cryptography standardization process. *National Institute of Standards and Technology, Gaithersburg, MD, NIST Interagency/Internal Rep.(NISTIR)*, 2019.

[125] Jim Vanderbauwhede and Jeremy Singer. *Operating Systems Foundations with Linux on the Raspberry Pi: Textbook*. arm Education Media, 2019.

[126] Serge Vaudenay. *A classical introduction to cryptography: Applications for communications security*. Springer Science & Business Media, 2006.

[127] Rolf H. Weber. Internet of Things – New security and privacy challenges. *Computer Law & Security Review*, 26(1):23–30, 2010.

[128] Yang Xiao, Hsiao-Hwa Chen, Xiaojiang Du, and Mohsen Guizani. Stream-based cipher feedback mode in wireless error channel. *IEEE Transactions on Wireless Communications*, 8(2):622–626, 2009.

[129] Inc. Xilinx. 7 series dsp48e1 slice, 2018.

[130] Inx. Xilinx. 7 series fpgas configurable logic block, 2016.

[131] Inx. Xilinx. Virtex-7 family overview, 2020.

[132] Inx. Xilinx. Ultrascale architecture dsp slice, 2021.

[133] Inx. Xilinx. 7 series fpgas configuration user guide, 2022.

[134] Xiaohui Xu. Study on security problems and key technologies of the internet of things. In *Proceedings - 2013 International Conference on Computational and Information Sciences, ICCIS 2013*, pages 407–410, 2013.

[135] Gavin Xiaoxu Yao, Junfeng Fan, Ray C.C. Cheung, and Ingrid Verbauwhede. Faster pairing coprocessor architecture. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013.

[136] Xueying Zhang, Howard M Heys, and Cheng Li. Energy efficiency of encryption schemes applied to wireless sensor networks. *Security and Communication Networks*, 5(7):789–808, 2012.

[137] Kai Zhao and Lina Ge. A survey on the internet of things security. In *Proceedings - 9th International Conference on Computational Intelligence and Security, CIS 2013*, pages 663–667, 2013.