Fachhochschule Dortmund
University of Applied Sciences and Arts

Faculty of Computer Science

Master Thesis

Design and FPGA implementation of a highly resource-efficient AES-256 encryption and decryption engine

In the master programme

Embedded Systems Engineering

Saul García Rodríguez

Supervisor:	Prof. Dr. Michael Karagounis
Second Advisor:	M.Sc. Felix Schneider
Place, date:	Dortmund, 24.10.2023

Statement of Authorship

I, Saul García Rodríguez, hereby certify that this research thesis has been written by me independently and without any unauthorized outside help. All sources and aids used are listed in the thesis in such a way that the nature and extent of their use are comprehensible.

The thesis has not yet been submitted in the same or a similar form for examination.

Dortmund, 24. October 2023

Signature

Abstract

Growing demand for security in a wide range of fields gives raise to research for more efficient and modern methods. Additionally, the increase of systems that are deployed on hardware requires security to be embedded in small area to protect intellectual property, hardware, and integrity and confidentially of sensible data. Therefore, in this work a design and FPGA implementation of a highly resource-efficient AES-256 encryption and decryption engine is presented, as well as its comparison with state-of-the-art designs. The design shows a reduction in the resources used due to its architecture to reuse hardware throughout all the processing. The design is implemented on a Xilinx Artix-7 FPGA.

Keywords: *Advance Encryption Standard (AES), encryption, decryption, Field Programmable Gate Array (FPGA), low-area, hardware security.*

Table of Contents

LIST OF FIGURES VIII

LIST OF TABLES XII

1 INTRODUCTION 1

2 MATHEMATICAL PRELIMINARIES 3

2.1 FINITE FIELDS 3

2.2 POLYNOMIALS OVER A FIELD 3

2.3 OPERATIONS ON SET OF POLYNOMIALS $GFpn$ 4

2.4 $GF28$ IN RIJNDAEL 10

2.4.1 MULTIPLICATION BY x 11

2.5 POLYNOMIALS WITH COEFFICIENTS IN $GF28$ 14

2.6 FUNCTIONS, TRANSFORMATIONS AND PERMUTATIONS 18

2.7 BLOCK CIPHERS 22

2.8 CORRELATION 24

2.9 DIFFERENCE PROPAGATION 24

3 SECURITY SOLUTIONS 25

3.1 RSA 25

3.2 CRYPTOGRAPHIC HASH FUNCTIONS 28

3.3 SHA-512 31

3.4 HMAC 35

4 XILINX FPGAS SECURITY SOLUTIONS 37

4.1 SPARTAN-3A/3AN/3A DSP FPGAS 39

4.1.1 ADVANCED DATA MANIPULATION 39

4.1.2 ADVANCED DATA MANIPULATION ON THE STORED CHECK CODE AND ALGORITHM CONTROL 40

4.1.3 ADDING FOURTH OUTPUT TO DE-MULTIPLEXER 41

4.2 7 SERIES FPGAS 41

4.2.1 BITSTREAM ENCRYPTION 41
4.2.2 BITSTREAM AUTHENTICATION 42
4.2.3 DEVICE IDENTIFIER/DNA 43
4.3 ULTRASCALE/ULTRASCALE+ FPGA 44
4.3.1 BITSTREAM CONFIDENTIALITY AND AUTHENTICATION (SYMMETRIC) 44
4.3.2 BITSTREAM AUTHENTICATION (ASYMMETRIC) 44

5 AES 46

5.1 ARCHITECTURE 47
5.1.1 ENCRYPTION 49
5.1.2 DECRYPTION 50
5.2 KEY SCHEDULE 51
5.2.1 KEY EXPANSION 52
5.2.2 ROUND KEY SELECTION 57
5.3 ADDROUNDKEY 58
5.4 SUBBYTES 59
5.5 INVSUBBYTES 60
5.6 SHIFTROWS 62
5.7 INVSHIFTROWS 63
5.8 MIXCOLUMNS 63
5.9 INVMIXCOLUMNS 64

6 DESIGN OF RIJNDAEL 66

6.1 GENERAL CRITERION 66
6.2 STRUCTURE SELECTION 67
6.3 TRANSFORMATIONS SELECTION 69
6.3.1 LOCAL NONLINEAR TRANSFORMATION $\boldsymbol{\gamma}$ 70
6.3.2 LINEAR MIXING TRANSFORMATION $\boldsymbol{\lambda}$ 73
6.3.3 LOCAL DIFFUSION TRANSFORMATION $\boldsymbol{\theta}$ 73
6.3.4 DISPERSION TRANSFORMATION $\boldsymbol{\pi}$ 75
6.4 ROUND TRANSFORMATION 76
6.5 NUMBER OF ROUNDS 78
6.6 KEY SCHEDULE 79

7 CONVENTIONAL AES-256 IMPLEMENTATION 80

7.1 APPLICATION OF STEPS ... 80
7.2 RESOURCE UTILIZATION ... 81
7.3 SCHEDULING ... 83
8 STATE-OF-THE-ART ... 87
8.1 AES ARCHITECTURE FOR SECURE ECG SIGNAL TRANSMISSION ... 87
8.2 IMPLEMENTATION AND COMPARATIVE ANALYSIS OF AES ... 88
8.3 AES IMPLEMENTATION ON XILINX FPGAS SUITABLE FOR FPGA BASED WBSNS ... 88
8.4 EFFICIENT AES IMPLEMENTATION USING XILINX SYSTEM GENERATOR ... 89
8.5 HIGH PERFORMANCE DATA ENCRYPTION WITH AES ... 90
8.6 HIGH THROUGHPUT AND FULLY PIPELINED IMPLEMENTATION OF AES-192 ... 92
8.7 IMPLEMENTATION OF AES ALGORITHM ON FPGA AND ON SOFTWARE ... 93
8.8 OPTIMIZATION AND IMPLEMENTATION OF AES ALGORITHM BASED ON FPGA ... 93
9 PROPOSED AES-256 IMPLEMENTATION ... 95
9.1 APPLICATION OF STEPS ... 95
9.2 RESOURCE UTILIZATION ... 97
9.3 SCHEDULING ... 98
10 AES256 ENCRYPTION IMPLEMENTATION ... 103
10.1 AES256 ENCRYPTION TOP MODULE ... 103
10.2 COUNTERS MODULE ... 106
10.3 KEY EXPANSION MODULE ... 106
10.3.1 ROUND CONSTANT MODULE ... 109
10.3.2 ROUND KEY MODULE ... 110
10.4 MIXCOLUMNS MODULE ... 111
10.4.1 MIX WORD MODULE ... 112
10.4.1.1 BYTEMIX MODULE ... 113
10.4.1.2 MULTIPLY BY 2 MODULE ... 114
10.4.1.3 MULTIPLY BY 3 MODULE ... 115
10.5 SUBBYTES MODULE ... 116
10.6 SHIFTROWS MODULE ... 117
11 AES256 DECRYPTION IMPLEMENTATION ... 118
11.1 AES256 DECRYPTION TOP MODULE ... 118
11.2 COUNTERS DEC MODULE ... 121

11.3 KEY EXPANSION DEC MODULE 121
11.3.1 ROUND KEY DEC MODULE 124
11.3.2 ROUND CONSTANT DEC MODULE 125
11.4 INV MIX COLUMNS MODULE 126
11.4.1 INV MIX WORD MODULE 126
11.4.1.1 INV BYTEMIX MODULE 127
11.4.1.2 MULTIPLY BY 2 MODULE 128
11.4.1.3 MULTIPLY BY 9 MODULE 128
11.4.1.4 MULTIPLY BY D MODULE 129
11.4.1.5 MULTIPLY BY B MODULE 130
11.4.1.6 MULTIPLY BY E MODULE 130
11.5 INV SUBBYTES MODULE 131
11.6 INV SHIFTROWS MODULE 132

12 SIMULATIONS 133

12.1 ENCRYPTION SIMULATION 133
12.1.1 KNOWN ANSWER TESTS 134
12.1.2 MONTE CARLO TEST 135
12.2 DECRYPTION SIMULATION 136
12.2.1 KNOWN ANSWER TESTS 136
12.2.2 MONTE CARLO TEST 137

13 TEST ON XILINX ARTIX-7 FPGA 139

13.1 GUI 139
13.2 ARCHITECTURE 141

14 RESULTS 144

14.1 STATE-OF-THE-ART ANALYSIS 144
14.2 RESULTS AND DISCUSSION 146

15 CONCLUSIONS 151

16 FUTURE WORK 152

BIBLIOGRAPHY XV

APPENDIX XXVII

APPENDIX A – GROUPS, RINGS AND FIELDS XXVIII

APPENDIX B – SUBSTITUTION TABLES XXX

APPENDIX C – AES-256 CODE FOR ENCRYPTION XXXV

APPENDIX D - AES-256 CODE FOR DECRYPTION LIV

APPENDIX E – TEST BENCH CODE AND RESULTS LXXVII

APPENDIX F – CODE FOR TESTING ON GUI CII

List of Figures

FIGURE 1 - EXAMPLE OF THE BOOLEAN PERMUTATION CALLED TRANSPOSITION. 19

FIGURE 2 - EXAMPLE OF TRANSPOSITION OF SUBSETS CONTAINING 2 BITS 20

FIGURE 3 - EXAMPLE OF BRICKLAYER TRANSFORMATION 21

FIGURE 4 - EXAMPLE OF AN ITERATIVE BOOLEAN TRANSFORMATION 21

FIGURE 5 – EXAMPLE OF ITERATIVE BLOCK CIPHER WITH THREE ROUND TRANSFORMATIONS 22

FIGURE 6 - EXAMPLE OF KEY-ALTERNATING BLOCK CIPHER WITH TWO ROUNDS 23

FIGURE 7 - MESSAGE AND HASH CODE BOTH ENCRYPTED. 28

FIGURE 8 - ENCRYPTION OF HASHED MESSAGE ONLY 29

FIGURE 9 - HASHING MESSAGE WITH SECRET VALUE 29

FIGURE 10 - CONFIDENTIALITY WITH ENCRYPTION AND AUTHENTICATION WITH SECRET VALUE 30

FIGURE 11 - HASH FUNCTION WITH DIGITAL SIGNATURE 30

FIGURE 12 - HASHED FUNCTION WITH DIGITAL SIGNATURE AND CONFIDENTIALITY 31

FIGURE 13 - MESSAGE PADDING FOR SHA-512 31

FIGURE 14 - SHA-512 ARCHITECTURE 33

FIGURE 15 – MESSAGE EXPANSION FOR SCHEDULING OF SHA-512 33

FIGURE 16 - HMAC ARCHITECTURE 36

FIGURE 17 - DATA MANIPULATION OF DEVICE DNA[24] 40

FIGURE 18 - DATA MANIPULATION ON STORED CHECK CODE [24] 40

FIGURE 19 - ADDING FOURTH OUTPUT TO DE-MULTIPLEXER. 41

FIGURE 20 - HASHED MESSAGE AUTHENTICATION OPERATION [22] 43

FIGURE 21 - 7 SERIES FPGA DNA_PORT DESIGN PRIMITIVE [25] 43

FIGURE 22 - BITSTREAM AUTHENTICATION USING RSA FOR DIGITAL SIGNATURE [23] 45

FIGURE 23 - AES INPUT AND OUTPUT BLOCK FOR ENCRYPTION AND DECRYPTION 47

FIGURE 24 - 128-BIT STATE ARRAY 48

FIGURE 25 - 256-BIT KEY ARRAY FOR THE AES-256 VARIANT 48

FIGURE 26 – AES ENCRYPTION STEPS 50

FIGURE 27 – AES DECRYPTION STEPS 51

FIGURE 28 - EXPANDEDKEY ARRAY W 53

FIGURE 29 - EXPANDEDKEY ARRAY W FOR AES-256 53

FIGURE 30 - KEY EXPANSION FOR FIRST Nk COLUMNS 54
FIGURE 31 – KEY EXPANSION IN AES-256 FOR THE FIRST Nk COLUMNS 54
FIGURE 32 - KEY EXPANSION FOR COLUMNS Wj, WHERE J IS MULTIPLE OF Nk 55
FIGURE 33 - KEY EXPANSION FOR COLUMN $W8$ FOR AES-256 55
FIGURE 34 - KEY EXPANSION FOR COLUMN Wj WHEN $j \bmod Nk = 4$ 56
FIGURE 35 - KEY EXPANSION FOR COLUMN $W12$ FOR AES-256 WHEN $j \bmod Nk = 4$ 56
FIGURE 36 - KEY EXPANSION FOR THE REST OF THE COLUMNS IN W 57
FIGURE 37 - KEY EXPANSION FOR COLUMN $W9$ FOR AES-256 57
FIGURE 38 - EXPANDEDKEY DIVIDED INTO ROUNDKEYS 58
FIGURE 39 - EXPANDEDKEY DIVIDED INTO ROUNDKEYS FOR AES-256 58
FIGURE 40 - ADDROUNDKEY STEP 58
FIGURE 41 - SUBBYTES STEP OPERATING ON INDIVIDUAL BYTES. 59
FIGURE 42 - INVSUBBYTES STEP OPERATING ON INDIVIDUAL BYTES 61
FIGURE 43 – SHIFTROWS OPERATING ON ROWS OF STATE. 62
FIGURE 44 - SHIFTROWS OPERATING ON ROWS OF STATE. 63
FIGURE 45 – MIXCOLUMN STEP OPERATING ON COLUMN OF STATE. 64
FIGURE 46 - INVMIXCOLUMN STEP OPERATING ON COLUMN OF STATE. 64
FIGURE 47 - EXAMPLE OF A $\gamma\lambda$ ROUND STRUCTURE 69
FIGURE 48 – ROUND TRANSFORMATION OF RIJNDAEL 77
FIGURE 49 – S-BOX INSTANCES FOR THE SUBBYTES STEP 82
FIGURE 50 - D-BOX INSTANCES FOR THE MIXCOLUMNS STEP 83
FIGURE 51 - MODIFIED ARCHITECTURE OF ROUND MODULE OF AES [75] 88
FIGURE 52 - ARCHITECTURE FOR AES IMPLEMENTATION USING BLOCK RAMS [78] 89
FIGURE 53 - PIPELINED ARCHITECTURE [79] 90
FIGURE 54 – DEEP-PIPELINED ARCHITECTURE OF AES [80] 91
FIGURE 55 – COMPLETE UNROLL DESIGN [80] 91
FIGURE 56 - FULLY-PIPELINED ARCHITECTURE FOR AES-192 WITH LOOP-UNROLLED TECHNIQUE [81] 92
FIGURE 57 - SUB-PIPELINED ARCHITECTURE FOR ROUND [81] 92
FIGURE 58 - PIPELINE INTERNAL ARCHITECTURE OF ROUND [83] 93
FIGURE 59 - PIPELINE INTERNAL ARCHITECTURE OF FINAL ROUND [83] 94
FIGURE 60 - FULLY UNROLLED INNER AND OUTER DOUBLE PIPELINED ARCHITECTURE [83] 94
FIGURE 61 – PROPOSED ARCHITECTURE OF AES-256 FOR ENCRYPTION 96
FIGURE 62 - PROPOSED ARCHITECTURE OF AES-256 FOR DECCRYPTION 96
FIGURE 63 – AES-256 STEPS FOR DECRYPTION 97
FIGURE 64 - S-BOX INSTANCES FOR THE *SUBBYTES* STEP 98
FIGURE 65 - D-BOX INSTANCES FOR THE MIX STEP 98
FIGURE 66 – SCHEDULING OF AES-256 FOR ENCRYPTION CONSIDERING INSTANCES USED BASED ON [84]. 100

FIGURE 67 – SCHEDULING OF AES-256 FOR ENCRYPTION WITH KEY EXPANSION IN REGARD TO INSTANCES BASED ON [84]. 101
FIGURE 68 - SCHEDULING OF AES-256 FOR DECRYPTION WITH KEY EXPANSION IN REGARD TO INSTANCES BASED ON [84]. 102
FIGURE 69 – AES256 TOP MODULE FOR ENCRYPTION 104
FIGURE 70 – AES-256 MODULES AND THEIR CONNECTIONS IN THE TOP MODULE FOR ENCRYPTION 105
FIGURE 71 – COUNTER MODULE 106
FIGURE 72 – MODULES INSTANTIATED IN KEY EXPANSION MODULE 108
FIGURE 73 – KEY EXPANSION MODULE 109
FIGURE 74 – ROUND CONSTANT MODULE 110
FIGURE 75 – ROUND KEY MODULE 111
FIGURE 76 – MIX COLUMNS MODULE 112
FIGURE 77 – MIX WORD MODULE 113
FIGURE 78 - BYTEMIX MODULE 114
FIGURE 79 – MULTIPLY BY 2 MODULE 115
FIGURE 80 - MULTIPLY BY 3 MODULE 115
FIGURE 81 – SUBBYTES MODULE 116
FIGURE 82 - S-BOX MODULE 116
FIGURE 83 – SHIFTROWS MODULE 117
FIGURE 84 – AES256 TOP MODULE FOR DECRYPTION 119
FIGURE 85 – AES-256 MODULES AND THEIR CONNECTIONS IN THE TOP MODULE FOR DECRYPTION 120
FIGURE 86 - COUNTERS DEC MODULE 121
FIGURE 87 - MODULES INSTANTIATED IN KEY EXPANSION DEC MODULE 123
FIGURE 88 - KEY EXPANSION DEC MODULE 124
FIGURE 89 - ROUND KEY DEC MODULE 125
FIGURE 90 - ROUND CONSTANT DEC MODULE 125
FIGURE 91 – INV MIX COLUMNS MODULE 126
FIGURE 92 – INV MIX WORD MODULE 127
FIGURE 93 – INV BYTEMIX MODULE 128
FIGURE 94 - MULTIPLY BY 2 MODULE 128
FIGURE 95 - MULTIPLY BY 9 MODULE 129
FIGURE 96 - MULTIPLY BY D MODULE 129
FIGURE 97 - MULTIPLY BY B MODULE 130
FIGURE 98 - MULTIPLY BY E MODULE 131
FIGURE 99 – INV SUBBYTES MODULE 131
FIGURE 100 – INV SHIFTROWS MODULE 132
FIGURE 101 – AESAVS KAT SIMULATION RESULTS FOR FIRST TWO TESTS FOR ENCRYPTION 134
FIGURE 102 – AESAVS KAT SIMULATION RESULTS FOR LAST TEST FOR ENCRYPTION 135

FIGURE 103 – MONTE CARLO TEST SIMULATION RESULTS FOR FIRST TWO KEYS FOR ENCRYPTION 136
FIGURE 104 - MONTE CARLO TEST SIMULATION RESULT FOR ENCRYPTION 136
FIGURE 105 - AESAVS KAT SIMULATION RESULTS FOR FIRST TWO TESTS FOR DECRYPTION 137
FIGURE 106 - AESAVS KAT SIMULATION RESULTS FOR LAST TEST FOR DECRYPTION 137
FIGURE 107 – MONTE CARLO TEST SIMULATION RESULTS FOR FIRST TWO KEYS FOR DECRYPTION 138
FIGURE 108 - MONTE CARLO TEST SIMULATION RESULT FOR DECRYPTION 138
FIGURE 109 - GRAPHICAL USER INTERFACE FOR VALIDATION OF AES-256 IMPLEMENTATION 141
FIGURE 110 - ARCHITECTURE TO TEST THE IMPLEMENTED CIPHER ON A XILINX FPGA 142
FIGURE 111 - STATE MACHINE TO RECEIVE, ENCRYPT OR DECRYPT DATA AND TRANSMIT DATA 143
FIGURE 112 - RESOURCE UTILIZATION [84] 146
FIGURE 113 - RESOURCE UTILIZATION FROM ENC [85] 147
FIGURE 114 - RESOURCE UTILIZATION FROM OUR DESIGN 147
FIGURE 115 – RESOURCE UTILIZATION REPORT FROM VIVADO FOR ENC [85] ON ARTIX-7 FPGA 147
FIGURE 116 - RESOURCE UTILIZATION REPORT FROM VIVADO FOR OUR ENCRYPTION DESIGN ON ARTIX-7 FPGA 148
FIGURE 117 - RESOURCE UTILIZATION REPORT FROM VIVADO FOR DEC [85] ON ARTIX-7 FPGA 148
FIGURE 118 - RESOURCE UTILIZATION REPORT FROM VIVADO FOR OUR DECRYPTION DESIGN ON ARTIX-7 FPGA 149
FIGURE 119 – DISTRIBUTED RAM USAGE IN DEC [85] 149
FIGURE 120 - SLICE LUTS AND SLICE REGISTERS USAGE FOR EXPANDED KEY IN OUR DECRYPTION DESIGN 149

List of Tables

TABLE 1 - INTERMEDIATE PRODUCTS USED FOR MULTIPLICATION OF POLYNOMIALS. 13
TABLE 2 - NUMBER OF ROUNDS Nr DEPENDING ON Nb (BLOCK LENGTH / 32) AND Nk (KEY LENGTH / 32) 49
TABLE 3 - ROUND CONSTANT RC TABLE 55
TABLE 4 - TABULAR REPRESENTATION OF RIJNDAEL S-BOX $SRDxy$ 60
TABLE 5 - TABULAR REPRESENTATION OF RIJNDAEL INVERSE S-BOX $SRD-1xy$ 61
TABLE 6 - SHIFT OFFSETS FOR DIFFERENT BLOCK LENGTHS 62
TABLE 7 – OFFSETS FOR DIFFERENT BLOCK LENGTHS 76
TABLE 8 - NUMBER OF ROUNDS Nr DEPENDING ON $Nb\ (block\ length/32)$ AND $Nk\ (key\ length/32)$ 78
TABLE 9 - SCHEDULING OF CONVENTIONAL IMPLEMENTATION OF AES-256 FOR ENCRYPTION 85
TABLE 10 - SCHEDULING OF CONVENTIONAL IMPLEMENTATION OF AES-256 FOR DECRYPTION 86
TABLE 11 - AES IMPLEMENTATIONS COMPARISON IN TERMS OF RESOURCES AND PERFORMANCE 145
TABLE 12 – AES IMPLEMENTATION COMPARISON BETWEEN OPEN-SOURCE DESIGNS 145
TABLE 13 - ENCRYPTION AND DECRYPTION DESIGNS FROM [85] 146
TABLE 14 - TABULAR REPRESENTATION OF S-BOX $SRDxy$ XXX
TABLE 15 - TABULAR REPRESENTATION OF THE INVERSE S-BOX $SRD-1xy$ XXXI
TABLE 16 - TABULAR REPRESENTATION OF THE AFFINE TRANSFORMATION $Aff8xy$ XXXII
TABLE 17 - TABULAR REPRESENTATION OF THE AFFINE TRANSFORMATION $Aff8-1xy$ XXXIII
TABLE 18 - TABULAR REPRESENTATION OF TRANSFORMATION $Inv8xy$ XXXIV

1 Introduction

The advance of technology allows a wide range of fields to implement modern solutions to common problems that were extremely complex to solve or demanded an excessive number of resources. These solutions often require processing large amount of sensible data that must be protected from malicious use. Moreover, deploying applications on hardware gives rise to modern threats, thus increasing the importance of reliable security solutions to provide confidentiality using as few resources as possible.

Not protecting the design properly may result in other companies learning from the design and developing new features or products from it, this is called reverse engineering. This process reduces initial investment for research and development and for continuous improvement. Competitors will also save time to release the product to market. From a military perspective the enemy will be able to produce counterattacks knowing beforehand what to expect.

If the attacker obtains the design and the intention is not to understand how it works but only its functional operation, the design can be copied and be released with a different name, this is known as cloning. The design does not need to be deeply analysed to be used for different applications. As a result of lack of understanding, further tests might be incomplete or incorrectly applied for validation. In safety-critical systems the consequences of using cloned designs may directly impact human lives.

Overbuilding is another threat that can impact a company. The unauthorized production of the identical device with the same name can saturate customer service without producing any revenue for the company.

The last threat to mention is tampering, which can be used to some degree in the concepts explained above. Tampering starts with the unauthorized access to the system and to the design to subsequently extract sensible data or to modify the functionality of the device.

Due to all the threats mentioned above, encryption of data has become a famous solution to provide confidentiality to sensible and private information. As a result, in 2001 the National Institute of Standards and Technology (NIST) defined the Advance Encryption Standard (AES) [1] to be used for encryption and decryption. Since then, researchers have proposed different designs and implementations for AES. This work presents a design and FPGA implementation of a highly-resource efficient encryption and decryption engine employing the Advance Encryption Standard (AES) with a 256-bit key.

In chapter 2, important mathematical concepts to understand AES are addressed, such as operations in Galois Fields and structures of ciphers. Chapter 3 focuses on algorithms that can be implemented to provide security. In chapter 4 techniques used by Xilinx to provide protection to designs implemented on their FPGAs are described. Chapter 5 explains the structure and operations of AES. The basis of the Rijndael design is explained in detail in chapter 6 to motivate each design decision. Chapter 7 explains the conventional implementation of the AES algorithm. Various implementations of AES designs on FPGAs from different authors are described in Chapter 8, whereas in chapter 9 the proposed implementation is explained. Chapter 10 explains the implementation of AES for encryption and chapter 11 the implementation for decryption. The simulations performed on the designs are described in chapter 12. Chapter 13 explains the test done on Xilinx Artix-7 FPGA for both encryption and decryption with the creation of a GUI. The results obtained from the designs are compared and discussed with other works in chapter 14. In chapter 15 the conclusions are discussed and in chapter 16 future work is proposed.

2 Mathematical Preliminaries

2.1 Finite Fields

A field with a finite number of elements is called *finite field*. The *order* of a finite filed is the number of elements contained in the field. A field is considered to be of order m if and only if $m = p^n$, where p is a prime integer and n any integer, in other words, if and only f m is a *prime power*. If this is the case, p is named the *characteristic* of the finite field. Finite fields with order p^n are denoted by $GF(p^n)$, whose elements can be represented by integers $0, 1, \dots\ p^n - 1$.

For finite fields with prime order $GF(p)$ with elements $0, 1, \dots\ p - 1$ the addition and multiplication operation can be represented by "integer addition modulo p" and "integer multiplication modulo p". Rijndael represents finite fields $GF(p^n)$ with $n > 1$ using polynomials over $GF(p)$ to achieve representing operations via modulo [2].

2.2 Polynomials over a Field

A polynomial over a field F is represented by the *indeterminate* x and the *coefficients* of the polynomial $b_j \in F$ as follows:

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_2x^2 + b_1x + b_0 \qquad (1)$$

The degree of the polynomial is defined by l if $b_j = 0, \forall j > l$, and l is the smallest number with this property. $F[x]$ is the representation of the set of polynomials over a field F. If the degree of the set is below l it can be represented

by $F[x]|_l$. It is important to remark the difference between the degree of the set of polynomials l and the order of the finite field m.

Example 2.1. Let field F be $GF(p)$ with order $m = p = 2$ and elements $0, 1$ and let $F[x]|_l$ be the set of polynomials over field $GF(2)$ with degree below $l = 8$. The last element in the set of polynomials $GF(2)|_8$ is the polynomial with all its coefficients equal to the last element in field $GF(2)$, which is element 1. The polynomial is then expressed as follows:

$$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Considering that the coefficients of the polynomial can be represented by a string of bits $b_7b_6b_5b_4b_3b_2b_1b_0$, the polynomials in the set $GF(2)|_8$ can be stored as bytes. The polynomial mentioned above corresponds to value FF in hexadecimal. Therefore, the set of polynomials $GF(p)|_n$ contains p^n elements.

Example 2.2. The byte 11 represented in hexadecimal can also be represented as the polynomial:

$$x^4 + 1.$$

The set of polynomials $GF(p)|_n$ can be used to do operations on bytes if the structure $(GF(p)|_n, +, \cdot)$ meets the requirements to be considered a field. In this case, the field can be denoted by $GF(p^n)$.

2.3 Operations on Set of Polynomials $GF(p)|_n$

2.3.1 $(GF(p)|_n, +)$ becoming an Abelian group

To consider structure $(GF(p)|_n, +, \cdot)$ a field, operations $(+, \cdot)$ on elements in the set $GF(p)|_n$ must be defined to satisfy the required properties (See Appendix A).

1. $(GF(p)|_n, +, \cdot)$ must be a commutative ring.

2. An inverse element in set F with respect to operation $(\cdot)$ exists for all elements of F, except for the neutral element of $(F, +)$ denoted by $\mathbf{0}$.

However, to meet property 1 operation $(+)$ must be defined first for elements in the set of polynomials $GF(p)|_n$, so that structure $(GF(p)|_n, +)$ can be considered an Abelian group.

Operation $(+)$ can be defined as addition of polynomials summing the coefficients with same powers of x:

$$c(x) = a(x) + b(x) \Leftrightarrow c_i = a_i + b_i, \qquad 0 \leq i < n \tag{2}$$

Let $a(x)$ and $b(x)$ be two elements in the set of polynomials $GF(p)|_n$. The addition of the polynomials results in $c(x)$.

$$a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2x^2 + a_1x + a_0,$$

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_2x^2 + b_1x + b_0,$$

$$c(x) = (a_{n-1} + b_{n-1})x^{n-1} + (a_{n-2} + b_{n-2})x^{n-2} + \cdots$$

$$+(a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0),$$

The result is another element in the set, since the maximum power of x in $c(x)$ is the maximum power in $a(x)$ and $b(x)$, so the operation satisfies the property of being closed (3).

$$\forall\, a(x), b(x) \in GF(p)|_n : a(x) + b(x) \in GF(p)|_n \tag{3}$$

Example 2.3. Let bytes 10011011 and 01101001 be represented as polynomials in set $GF(2)|_8$. The addition of these elements is performed as follows:

$$(x^7 + x^4 + x^3 + x + 1) + (x^6 + x^5 + x^3 + 1$$
$$= x^7 + x^6 + x^5 + x^4 + (1+1)x^3 + x + (1+1)$$
$$= x^7 + x^6 + x^5 + x^4 + x$$

Since the elements in field $GF(2)$ are 0 and 1 the addition of 1 and 1 equals 0. This operation can be implemented with a bitwise XOR of the coefficients of the polynomials stored as bits.

There must be a neutral element (4) denoted by $\mathbf{0}$ for the addition operation in structure $(GF(p)|_n, +)$.

$$\exists\, 0 \in GF(p)|_n, \forall\, a(x) \in GF(p)|_n : a(x) + 0 = a(x) \tag{4}$$

The neutral element $\mathbf{0}$ for the addition operation is defined as the polynomial with all coefficients equal to 0.

$$c(x) = a(x) + 0 \Leftrightarrow c_i = a_i, \qquad 0 \le i < n \tag{5}$$

Let $a(x)$ be an element in the set of polynomials $GF(p)|_n$ and $\mathbf{0}$ the neutral element for the addition operation.

$$a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2x^2 + a_1x + a_0,$$

$$0 = (0)x^{n-1} + (0)x^{n-2} + \cdots + (0)x^2 + (0)x + (0),$$

$$c(x) = (a_{n-1} + 0)x^{n-1} + (a_{n-2} + 0)x^{n-2} + \cdots + (a_2 + 0)x^2 + (a_1 + 0)x + (a_0 + 0)$$
$$= a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2x^2 + a_1x + a_0$$

There must be an inverse element for every element (6), except for the neutral element, for the addition operation in structure $(GF(p)|_n, +)$.

$$\forall\, a(x) \in GF(p)|_n, \exists\, b(x) \in GF(p)|_n : a(x) + b(x) = 0. \tag{6}$$

The inverse element for the addition operation is obtained replacing each coefficient by its inverse element in $GF(p)$.

$$0 = a(x) + b(x) \Leftrightarrow 0 = a_i + b_i, \qquad 0 \leq i < n \tag{7}$$

In this way, the inverse element for each coefficient can be defined by itself.

$$a_i = b_i, \qquad 0 \leq i < n$$

Example 2.4. Let byte $C3$ be represented as element $a(x)$ in set $GF(2)|_8$. Therefore, the inverse element $b(x)$ for the addition operation corresponds to $C3$.

$$a(x) = x^7 + x^6 + x + 1,$$

$$b(x) = x^7 + x^6 + x + 1,$$

$$\begin{aligned} a(x) + b(x) &= (x^7 + x^6 + x + 1) + (x^7 + x^6 + x + 1) \\ &= (1+1)x^7 + (1+1)x^6 + (1+1)x + (1+1) = 0 \end{aligned}$$

Under the definition of operation $(+)$ for structure $(GF(p)|_n, +)$, it can be concluded that $(+)$ is also associative (8) and commutative (9) on the polynomials in set $GF(p)|_n$.

$$\begin{gathered} \forall\, a(x), b(x), c(x) \in GF(p)|_n : \\ \big(a(x) + b(x)\big) + c(x) = a(x) + \big(b(x) + c(x)\big) \end{gathered} \tag{8}$$

$$\forall\, a(x), b(x) \in GF(p)|_n : a(x) + b(x) = b(x) + a(x) \tag{9}$$

Example 2.5. Let byte $9E$ be polynomial $a(x)$, byte CA be $b(x)$ and byte $2F$ be $c(x)$ contained in set $GF(2)|_8$.

$$\begin{aligned}
&\big(a(x) + b(x)\big) + c(x) \\
&\quad = \big((x^7 + x^4 + x^3 + x^2 + x) + (x^7 + x^6 + x^3 + x)\big) \\
&\quad + (x^5 + x^3 + x^2 + x + 1) \\
&\quad = \big((1 + 1)x^7 + x^6 + x^4 + (1 + 1)x^3 + x^2 + (1 + 1)x\big) \\
&\quad + (x^5 + x^3 + x^2 + x + 1) \\
&\quad = (1 + 1)x^7 + x^6 + x^5 + x^4 + (1 + 1 + 1)x^3 + (1 + 1)x^2 \\
&\quad + (1 + 1 + 1)x + 1 \\
&\quad = (x^7 + x^4 + x^3 + x^2 + x) \\
&\quad + (x^7 + x^6 + x^5 + (1 + 1)x^3 + x^2 + (1 + 1)x + 1) = \\
&\quad = a(x) + \big(b(x) + c(x)\big)
\end{aligned}$$

$$\begin{aligned}
a(x) + b(x) &= (x^7 + x^4 + x^3 + x^2 + x) + (x^7 + x^6 + x^3 + x) \\
&= (1 + 1)x^7 + x^6 + x^4 + (1 + 1)x^3 + x^2 + (1 + 1)x \\
&= (x^7 + x^6 + x^3 + x) + (x^7 + x^4 + x^3 + x^2 + x) = b(x) + a(x)
\end{aligned}$$

After the properties demonstrated for operation $(+)$, structure $(GF(p)|_n, +)$ qualifies as an Abelian group.

2.3.2 $(GF(p)|_n, +, \cdot)$ becoming a Commutative Ring

Operation $(\cdot)$ is defined as multiplication of polynomials and must be associative (10), commutative (11) and be related to operation $(+)$ by the law of distributive (12).

$$\begin{gathered}
\forall\, a(x), b(x), c(x) \in GF(p)|_n : \\
\big(a(x) \cdot b(x)\big) \cdot c(x) = a(x) \cdot \big(b(x) \cdot c(x)\big)
\end{gathered} \tag{10}$$

$$\forall\, a(x), b(x) \in GF(p)|_n : a(x) \cdot b(x) = b(x) \cdot a(x) \tag{11}$$

$$\begin{gathered}
\forall\, a(x), b(x), c(x) \in GF(p)|_n : \big(a(x) + b(x)\big) \cdot c(x) \\
= \big(a(x) \cdot c(x)\big) + \big(b(x) \cdot c(x)\big)
\end{gathered} \tag{12}$$

The neutral element for multiplication of polynomials is the polynomial of degree 0 with coefficient 1 for x^0.

$$\exists\, 1 \in GF(p)|_n, \forall\, a(x) \in GF(p)|_n : a(x) \cdot 1 = a(x) \tag{13}$$

A polynomial in set $GF(p)|_n$ is obtained as the result of multiplication of elements in the set, using a *reduction polynomial* $m(x)$ of degree $l = n$. Consequently, operation $(\cdot)$ is considered closed (14) on the elements in $GF(p)|_n$ and the result is then defined as the algebraic product of the polynomials in set $GF(p)|_n$ modulo $m(x)$.

$$\forall\, a(x), b(x) \in GF(p)|_n : a(x) \cdot b(x) \in GF(p)|_n \tag{14}$$

$$c(x) = a(x) \cdot b(x) \Leftrightarrow c(x) \equiv a(x) \times b(x) \pmod{m(x)} \tag{15}$$

After defining operations $(+)$ and $(\cdot)$, structure $(GF(p)|_n, +, \cdot)$ can be considered a commutative ring, since the properties of $(+)$ and $(\cdot)$ meet all the requirements.

2.3.3 $(GF(p)|_n, +, \cdot)$ becoming a Finitie Field $GF(p^n)$

Finally, to call structure $(GF(p)|_n, +, \cdot)$ a field, there must exist an inverse element (16) for operation $(\cdot)$.

$$\forall a(x) \in GF(p)|_n, \exists\, b(x) \in GF(p)|_n : a(x) \cdot b(x) = 1 \tag{16}$$

Using the extended Euclidean algorithm (17) polynomials $b(x)$ and $c(x)$ can be obtained, as well as the greatest common divisor for $a(x)$ and $m(x)$.

$$a(x) \times b(x) + m(x) \times c(x) = gcd\big(a(x), m(x)\big) \tag{17}$$

The greatest common divisor of $a(x)$ and $m(x)$ is equal to 1, if and only if $m(x)$ is an *irreducible* polynomial of degree n over $GF(p)|_n$, meaning that there are no polynomials $d(x)$ and $e(x)$ with coefficients in $GF(p)|_n$ and degree > 0

such that $m(x) = d(x) + e(x)$. (18) is obtained from the definition for operation $(\cdot)$ in (15) and after performing modular reduction (19).

$$a(x) \times b(x) + m(x) \times c(x) = 1 \pmod{m(x)} \tag{18}$$

$$a(x) \times b(x) = 1 \pmod{m(x)} \tag{19}$$

With the definitions given for operations $(+,\cdot)$ on elements in the set of polynomials $GF(p)|_n$, structure $(GF(p)|_n, +,\cdot)$ satisfies the conditions needed to be a field when the reduction polynomial $m(x)$ is irreducible. The field is denoted by $GF(p^n)$ containing p^n elements.

2.4 $GF(2^8)$ in Rijndael

The characteristic p of the finite field $GF(p^n)$ used in Rijndael is 2 with $n = 8$. Hence, the finite field used in Rijndael is $GF(2^8)$, which contains 2^8 elements.

The irreducible reduction polynomial $m(x)$ used in the specification of Rijndael [3], and, therefore, in AES [4] to define polynomial multiplication is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{20}$$

Example 2.6. Let bytes $D1$ and $6E$ be elements in $GF(2^8)$. The product of these polynomials is AD, which is also an element contained in $GF(2^8)$.

$$\begin{aligned}
&(x^7 + x^6 + x^4 + 1) \times (x^6 + x^5 + x^3 + x^2 + x) \\
&\qquad = (x^{13} + x^{12} + x^{10} + x^9 + x^8) + (x^{12} + x^{11} + x^9 + x^8 + x^7) \\
&\qquad + (x^{10} + x^9 + x^7 + x^6 + x^5) + (x^6 + x^5 + x^3 + x^2 + x) \\
&\qquad = (x^{13} + (1 + 1)x^{12} + x^{11} + (1 + 1)x^{10} + (1 + 1 + 1)x^9 + (1 + 1)x^8 \\
&\qquad + (1 + 1)x^7 + (1 + 1)x^6 + (1 + 1)x^5 + x^3 + x^2 + x) \\
&\qquad = (x^{13} + x^{11} + x^9 + x^3 + x^2 + x)
\end{aligned}$$

$$(x^7 + x^6 + x^4 + 1) \times (x^6 + x^5 + x^3 + x^2 + x) \\ \equiv x^7 + x^5 + x^3 + x^2 + 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

Modular reduction is achieved subtracting a multiple of $m(x)$, which in terms of an XOR is the same as the addition of a multiple of $m(x)$ to the result. The addition operation was previously defined as the XOR operation. After each reduction with a multiple of $m(x)$, the degree of the polynomial decreases.

$$(x^{13} + x^{11} + x^9 + x^3 + x^2 + x) + (x^8 + x^4 + x^3 + x + 1) \times (x^5) \\ = (x^{13} + x^{11} + x^9 + x^3 + x^2 + x) + (x^{13} + x^9 + x^8 + x^6 + x^5) \\ = x^{11} + x^8 + x^6 + x^5 + x^3 + x^2 + x$$

$$(x^{11} + x^8 + x^6 + x^5 + x^3 + x^2 + x) + (x^8 + x^4 + x^3 + x + 1) \times (x^3) \\ = (x^{11} + x^8 + x^6 + x^5 + x^3 + x^2 + x) + (x^{11} + x^7 + x^6 + x^4 + x^3) \\ = x^8 + x^7 + x^5 + x^4 + x^2 + x$$

$$(x^8 + x^7 + x^5 + x^4 + x^2 + x) + (x^8 + x^4 + x^3 + x + 1) = x^7 + x^5 + x^3 + x^2 + 1$$

The complete multiple of $m(x)$ can be used to reduce the polynomial in one operation.

$$(x^{13} + x^{11} + x^9 + x^3 + x^2 + x) + (x^8 + x^4 + x^3 + x + 1) \times (x^5 + x^3 + 1) \\ = (x^{13} + x^{11} + x^9 + x^3 + x^2 + x) \\ + (x^{13} + x^{11} + x^9 + x^7 + x^5 + x + 1) = x^7 + x^5 + x^3 + x^2 + 1$$

Finally, the product of $D1$ and $6E$ in $GF(2^8)$ is:

$$(x^7 + x^6 + x^4 + 1) \times (x^6 + x^5 + x^3 + x^2 + x) \\ \equiv x^7 + x^5 + x^3 + x^2 + 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

2.4.1 Multiplication by x

The approach to reduce the product of two polynomials used for the implementation of Rijndael is based on the multiplication of a polynomial with polynomial x modulo $m(x)$ [5].

$$\begin{aligned} a(x) = b(x) \times x \\ &= (b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) \\ &\times x \\ &= b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \\ &\equiv a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x \\ &+ a_0 \pmod{x^8 + x^4 + x^3 + x + 1} \end{aligned} \tag{21}$$

The reduction polynomial $m(x)$ is used when $b_7 = 1$. In this case, $m(x)$ is subtracted to the polynomial, which is the same as addition due to its implementation with an XOR operation.

Multiplication by x is the same as multiplying a byte by 2 due to the definitions of the finite field $GF(2^8)$. For this reason, multiplying polynomials by x is achieved at the byte level with a left shift and a conditional bitwise XOR with byte $m(x) = 1B$ depending on b_7.

Since the finite field satisfies the law of distributivity for addition and multiplication and polynomials can be formed through the addition of polynomials with one coefficient different than 0 $(1, x, x^2, x^3, x^4, x^5, x^6, x^7)$, multiplication on bytes can be implemented with the addition of intermediate products.

Example 2.7. Let byte 3A and byte 73 be elements in $GF(2^8)$. The product of the two polynomials is obtained using the law of distributivity adding intermediate products of byte 3A with polynomials with one coefficient different than 0.

$$\begin{aligned} 3A \cdot 73 = 3A \cdot (40 \oplus 20 \oplus 10 \oplus 02 \oplus 01) \\ = (3A \cdot 40) \oplus (3A \cdot 20) \oplus (3A \cdot 10) \oplus (3A \cdot 02) \oplus (3A \cdot 01) \end{aligned}$$

The operations to obtain the intermediate products for byte $3A$ are presented in hexadecimal format and represented as polynomials in Table 1.

Hexadecimal Format	Polynomial Representation
$3A \cdot 02 = 74$	$(x^5 + x^4 + x^3 + x) \times x = x^6 + x^5 + x^4 + x^2$

$3A \cdot 04 = (3A \cdot 02) \cdot 02$ $= 74 \cdot 02$ $= E8$	$(x^5 + x^4 + x^3 + x) \times x^2$ $= \big((x^5 + x^4 + x^3 + x) \times x\big) \times x$ $= (x^6 + x^5 + x^4 + x^2) \times x$ $= x^7 + x^6 + x^5 + x^3$
$3A \cdot 08 = (3A \cdot 04) \cdot 02$ $= E8 \cdot 02 = 1D0$ $\equiv CB \pmod{11B}$	$(x^5 + x^4 + x^3 + x) \times x^3$ $= \big((x^5 + x^4 + x^3 + x) \times x^2\big) \times x$ $= (x^7 + x^6 + x^5 + x^3) \times x$ $= x^8 + x^7 + x^6 + x^4$ $\equiv x^7 + x^6 + x^3 + x$ $+ 1 \pmod{x^8 + x^4 + x^3 + x + 1}$
$3A \cdot 10 = (3A \cdot 08) \cdot 02$ $\equiv CB \cdot 02 = 196$ $\equiv 8D \pmod{11B}$	$(x^5 + x^4 + x^3 + x) \times x^4$ $= \big((x^5 + x^4 + x^3 + x) \times x^3\big) \times x$ $\equiv (x^7 + x^6 + x^3 + x + 1) \times x$ $= x^8 + x^7 + x^4 + x^2 + x$ $\equiv x^7 + x^3 + x^2$ $+ 1 \pmod{x^8 + x^4 + x^3 + x + 1}$
$3A \cdot 20 = (3A \cdot 10) \cdot 02$ $\equiv 8D \cdot 02 = 11A$ $\equiv 01 \pmod{11B}$	$(x^5 + x^4 + x^3 + x) \times x^5$ $= \big((x^5 + x^4 + x^3 + x) \times x^4\big) \times x$ $\equiv (x^7 + x^3 + x^2 + 1) \times x$ $= x^8 + x^4 + x^3 + x$ $\equiv 1 \pmod{x^8 + x^4 + x^3 + x + 1}$
$3A \cdot 40 = (3A \cdot 20) \cdot 02$ $\equiv 01 \cdot 02$ $= 02 \pmod{11B}$	$(x^5 + x^4 + x^3 + x) \times x^6$ $= \big((x^5 + x^4 + x^3 + x) \times x^5\big) \times x$ $\equiv 1 \times x$ $= x \pmod{x^8 + x^4 + x^3 + x + 1}$

Table 1 - Intermediate products used for multiplication of polynomials.

Using the results from Table 1 the product of polynomials $3A$ and 73 is obtained.

$$\begin{aligned} 3A \cdot 73 &= 3A \cdot (40 \oplus 20 \oplus 10 \oplus 02 \oplus 01) \\ &= (3A \cdot 40) \oplus (3A \cdot 20) \oplus (3A \cdot 10) \oplus (3A \cdot 02) \oplus (3A \cdot 01) \\ &\equiv 02 \oplus 01 \oplus 8D \oplus 74 \oplus 3A = C0 \pmod{11B} \end{aligned}$$

$$
\begin{aligned}
(x^5 + x^4 + x^3 + x) \times (x^6 + x^5 + x^4 + x + 1) \\
&= \left((x^5 + x^4 + x^3 + x) \times x^6\right) + \left((x^5 + x^4 + x^3 + x) \times x^5\right) \\
&+ \left((x^5 + x^4 + x^3 + x) \times x^4\right) + \left((x^5 + x^4 + x^3 + x) \times x\right) \\
&+ \left((x^5 + x^4 + x^3 + x) \times 1\right) \\
&\equiv x + 1 + (x^7 + x^3 + x^2 + 1) + (x^6 + x^5 + x^4 + x^2) \\
&+ (x^5 + x^4 + x^3 + x) = x^7 + x^6 \pmod{11B}
\end{aligned}
$$

2.5 Polynomials with coefficients in $GF(2^8)$

Throughout the Rijndael algorithm not only multiplication of bytes is needed, but also multiplication of arrays of bytes in the form:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0, \tag{22}$$

where the coefficients are polynomials in $GF(2^8)$. This means, there are four bytes involved in polynomial $a(x)$, one four each coefficient.

Addition is defined the same way as addition of polynomials in $GF(2^8)$. Coefficients with the same power are added by means of a bitwise XOR (23). Compared to addition in $GF(2^8)$ where the operation XOR is carried out in single bits, in this case, addition is performed on bytes.

$$a(x) + b(x) = (a_3 + b_3)x^3 + (a_2 + b_2)x^2 + (a_1 + b_1)x + (a_0 + b_0) \tag{23}$$

Multiplication of polynomials in this form is described in (24). However, since the degree of the result can be greater than the required degree, which is three, it is necessary to use a reduction polynomial.

$$\begin{aligned} c(x) = a(x) \times b(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0) \times (b_3x^3 + b_2x^2 + b_1x + b_0) \\ &= (a_3 \times b_3)x^6 + \big((a_3 \times b_2) + (a_2 \times b_3)\big)x^5 \\ &+ \big((a_3 \times b_1) + (a_2 \times b_2) + (a_1 \times b_3)\big)x^4 \\ &+ \big((a_3 \times b_0) + (a_2 \times b_1) + (a_1 \times b_2) + (a_0 \times b_3)\big)x^3 \\ &+ \big((a_2 \times b_0) + (a_1 \times b_1) + (a_0 \times b_2)\big)x^2 \\ &+ \big((a_1 \times b_0) + (a_0 \times b_1)\big)x + (a_0 \times b_0) \\ &= c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \end{aligned} \tag{24}$$

The reduction polynomial specified for multiplication of polynomials with coefficients in $GF(2^8)$ in Rijndael [6] and AES [7] is presented in (25) and it is different than the one specified for multiplication in $GF(2^8)$ introduced in (20).

$$l(x) = x^4 + 1 \tag{25}$$

The result obtained from multiplication of polynomials in this form with the reduction polynomial $l(x)$ ensures that this modular product has a degree equal or less than three.

$$\begin{aligned} d(x) = a(x) \times b(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0) \times (b_3x^3 + b_2x^2 + b_1x + b_0) \\ &= (a_3 \times b_3)x^6 + \big((a_3 \times b_2) + (a_2 \times b_3)\big)x^5 \\ &+ \big((a_3 \times b_1) + (a_2 \times b_2) + (a_1 \times b_3)\big)x^4 \\ &+ \big((a_3 \times b_0) + (a_2 \times b_1) + (a_1 \times b_2) + (a_0 \times b_3)\big)x^3 \\ &+ \big((a_2 \times b_0) + (a_1 \times b_1) + (a_0 \times b_2)\big)x^2 \\ &+ \big((a_1 \times b_0) + (a_0 \times b_1)\big)x + (a_0 \times b_0) \\ &\equiv d_3x^3 + d_2x^2 + d_1x + d_0 \pmod{x^4 + 1} \end{aligned} \tag{26}$$

The reduction polynomial $l(x)$ is reducible, thus, there is no inverse element for each element that can be represented as a polynomial with the form defined in (22). Consequently, to ensure invertibility polynomials used to multiply other polynomials must have an inverse element.

Due to the characteristics of the reduction polynomial $l(x)$, the polynomial reduction can be expressed as:

$$C_i x^i \mathrm{mod}(x^4+1) = C_i x^{i\mathrm{mod}4}. \tag{27}$$

This expression is obtained subtracting a multiple of the reduction polynomial $l(x)$ to the polynomial, which from the definition for addition given in (23) is the same as adding a multiple of $l(x)$ to the polynomial.

$$\begin{aligned} C_i x^i \mathrm{mod}(x^4+1) \\ &= C_i x^i + C_i x^{i-4}(x^4+1) = C_i x^i + C_i x^{i-4+4} + C_i x^{i-4} \\ &= C_i x^i + C_i x^i + C_i x^{i-4} = C_i x^{i-4} = C_i x^{i\mathrm{mod}4} \end{aligned} \tag{28}$$

The coefficients d_i with $0 \leq i < 4$ for $d(x)$ in (26) obtained after the modular product consists of the addition of products of polynomials in $GF(2^8)$.

$$\begin{aligned} d(x) = a(x) \times b(x) \\ &= (a_3x^3 + a_2x^2 + a_1x + a_0) \times (b_3x^3 + b_2x^2 + b_1x + b_0) \\ &= (a_3 \times b_3)x^6 + \big((a_3 \times b_2) + (a_2 \times b_3)\big)x^5 \\ &+ \big((a_3 \times b_1) + (a_2 \times b_2) + (a_1 \times b_3)\big)x^4 \\ &+ \big((a_3 \times b_0) + (a_2 \times b_1) + (a_1 \times b_2) + (a_0 \times b_3)\big)x^3 \\ &+ \big((a_2 \times b_0) + (a_1 \times b_1) + (a_0 \times b_2)\big)x^2 \\ &+ \big((a_1 \times b_0) + (a_0 \times b_1)\big)x + (a_0 \times b_0) \\ &\equiv \big((a_3 \times b_0) + (a_2 \times b_1) + (a_1 \times b_2) + (a_0 \times b_3)\big)x^3 \\ &+ \big((a_2 \times b_0) + (a_1 \times b_1) + (a_0 \times b_2) + (a_3 \times b_3)\big)x^2 \\ &+ \big((a_1 \times b_0) + (a_0 \times b_1) + (a_3 \times b_2) + (a_2 \times b_3)\big)x \\ &+ \big((a_0 \times b_0) + (a_3 \times b_1) + (a_2 \times b_2) + (a_1 \times b_3)\big) \\ &\equiv d_3x^3 + d_2x^2 + d_1x + d_0 \quad \big(\mathrm{mod}(x^4+1)\big) \end{aligned} \tag{29}$$

Therefore,

$$\begin{aligned} d_0 &= (a_0 \times b_0) + (a_3 \times b_1) + (a_2 \times b_2) + (a_1 \times b_3) \\ d_1 &= (a_1 \times b_0) + (a_0 \times b_1) + (a_3 \times b_2) + (a_2 \times b_3) \\ d_2 &= (a_2 \times b_0) + (a_1 \times b_1) + (a_0 \times b_2) + (a_3 \times b_3) \\ d_3 &= (a_3 \times b_0) + (a_2 \times b_1) + (a_1 \times b_2) + (a_0 \times b_3) \end{aligned} \tag{30}$$

The modular multiplication can be represented in the form of a matrix.

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0\ a_3\ a_2\ a_1 \\ a_1\ a_0\ a_3\ a_2 \\ a_2\ a_1\ a_0\ a_3 \\ a_3\ a_2\ a_1\ a_0 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{31}$$

Since the reduction polynomial $l(x)$ is reducible, there is no inverse element for polynomials that can be divided by $(x^4 + 1)$, which is the irreducible factor of $l(x)$. Hence, for invertibility, AES specifies a four-term polynomial $a(x)$ along with its inverse element $a^{-1}(x)$ to be used throughout the algorithm, the coefficients are presented as bytes in hexadecimal format.

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{32}$$

$$a^{-1}\,(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\} \tag{33}$$

Verification of invertibility is achieved multiplying polynomials $a(x)$ and $a^{-1}\,(x)$.

$$\begin{aligned} d(x) = a(x) \times a^{-1}\,(x) & \\ &= (\{03\}x^3 + \{01\,\}x^2 + \{01\}x + \{02\}) \\ &\times (\{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}) \\ &\equiv d_3x^3 + d_2x^2 + d_1x + d_0\ = 1 \quad \left(\mathrm{mod}(x^4 + 1)\right) \end{aligned} \tag{34}$$

Represented in matrix form:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \cdot \begin{bmatrix} 0E \\ 09 \\ 0D \\ 0B \end{bmatrix} \tag{35}$$

Coefficients are thus obtained as follows:

$$
\begin{aligned}
d_0 &= (02 \cdot 0E) \oplus (03 \cdot 09) \oplus (01 \cdot 0D) \oplus (01 \cdot 0B) \\
&= 1C \oplus 1B \oplus 0D \oplus 0B = 1 \\
d_1 &= (01 \cdot 0E) \oplus (02 \cdot 09) \oplus (03 \cdot 0D) \oplus (01 \cdot 0B) \\
&= 0E \oplus 12 \oplus 17 \oplus 0B = 0 \\
d_2 &= (01 \cdot 0E) \oplus (01 \cdot 09) \oplus (02 \cdot 0D) \oplus (03 \cdot 0B) \\
&= 0E \oplus 09 \oplus 1A \oplus 21 = 0 \\
d_3 &= (03 \cdot 0E) \oplus (01 \cdot 09) \oplus (01 \cdot 0D) \oplus (02 \cdot 0B) \\
&= 12 \oplus 09 \oplus 0D \oplus 16 = 0
\end{aligned}
\tag{36}
$$

In conclusion, for any modular product $c(x)$ of a four-term polynomial $p(x)$ with polynomial $a(x)$ by using the inverse polynomial $a^{-1}(x)$ it is possible to obtain the four-term polynomial $p(x)$.

$$
\begin{aligned}
c(x) &= a(x) \times p(x) \\
p(x) &= a^{-1}(x) \times c(x)
\end{aligned}
\tag{37}
$$

2.6 Functions, Transformations and Permutations

The mapping of Boolean vectors to other Boolean vectors can be described using *Boolean functions*. In this case Boolean vectors do not require to have the same length.

$$b = \phi(a) \tag{38}$$

In cryptography, Boolean vectors are often represented as blocks called *states* as is the case in the Rijndael block cipher. Boolean functions that operate on a state have the same length in the input as in the output and are called *Boolean transformations* [8].

A Boolean transformation is called a *Boolean permutation* if it is invertible, in other words, if all the possible inputs are mapped one-to-one to all the possible

outputs [8]. An example of a Boolean permutation is changing the position of bits, this operation is called *transposition* [9]. The function can be represented as a permutation over the index space $p(i)$ as in (40). An example is given in Figure 1.

$$b = \pi(a) \tag{39}$$

$$b_i = a_{p(i)} \tag{40}$$

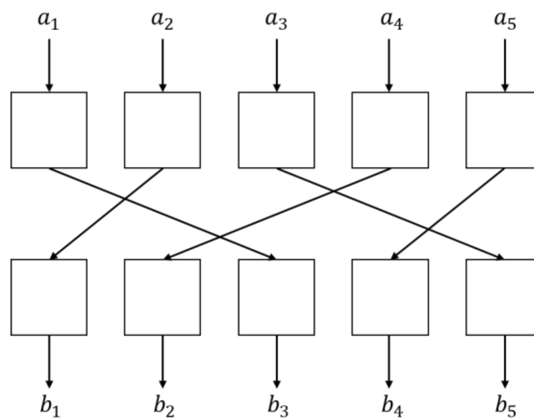

Figure 1 - Example of the Boolean permutation called transposition.

In Rijndael, the n_b bits in the state are grouped in n_t number of tuples with 8 bits each. A byte a_i within the state is specified by its index position $i \in I$ with the *index space* I defined as the set $\{1, \dots n_b\}$. A bit $a_{(i,j)}$ can be specified by the byte position within the state i and the bit position within a byte j.

If a subset of bits is moved without changing the position of bits within the subset and the subset is a byte, the operation is called *byte transposition* [9]. The index of the position of the byte can be defined as a permutation over the index space $p(i)$, whereas the index of the position of bits j remains the same.

$$b_{(i,j)} = a_{(p(i),j)} \tag{41}$$

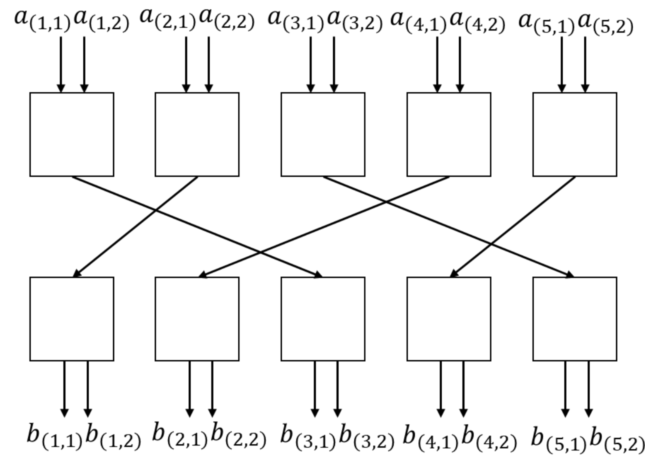

Figure 2 - Example of transposition of subsets containing 2 bits

When a function can be described as a set of Boolean functions being applied independently to individual subsets of a vector, this function is called a *bricklayer function* [10]. Nonlinear bricklayer functions are called *S-boxes* and linear ones *D-boxes*, where the letter D stands for diffusion. Analogous to the Boolean transformation, when a bricklayer function operates on a state it is called a *bricklayer transformation* [10]. A graphical illustration is given in Figure 3 and its mathematical representation in (42). The Boolean functions γ_i conforming the bricklayer transformation can be different to each other. If the transformation is invertible, it is called *bricklayer permutation* [10]. Rijndael uses a nonlinear bricklayer permutation (S-box) over bytes, and a linear bricklayer permutation (D-box) over columns formed by four bytes, where all the S-boxes are defined the same, as well as the D-boxes.

$$\left(b_{(i,1)}, b_{(i,2)}, \dots b_{(i,m)}\right) = \gamma_i\left(a_{(i,1)}, a_{(i,2)}, \dots a_{(i,m)}\right) \tag{42}$$

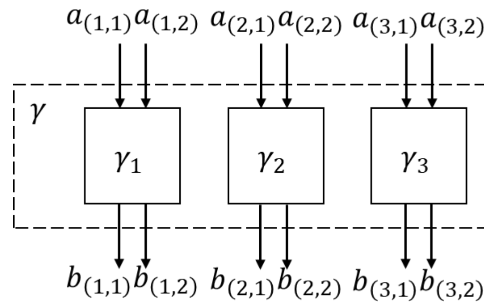

Figure 3 - Example of bricklayer transformation

A transformation β that is composed by a sequence of Boolean transformations ρ operating over a state a iteratively is called *iterative Boolean transformation* [11]. The sequence of transformations is represented in (43), where r is the number of iterations. The mathematical representation of intermediate states $a^{(i)}$ obtained after each transformation $\rho^{(i)}$ is shown in (44). A graphical example is presented in Figure 4. In order to call the transformation an *iterated Boolean permutation* the sequence of Boolean transformations must be Boolean permutations [11].

$$\beta = \rho^{(r)} \circ \ldots \circ \rho^{(2)} \circ \rho^{(1)} \tag{43}$$

$$a^{(i)} = \rho^{(i)}\left(a^{(i-1)}\right) \tag{44}$$

$$b = a^{(m)} = \beta(d) = \beta\left(a^{(0)}\right) \tag{45}$$

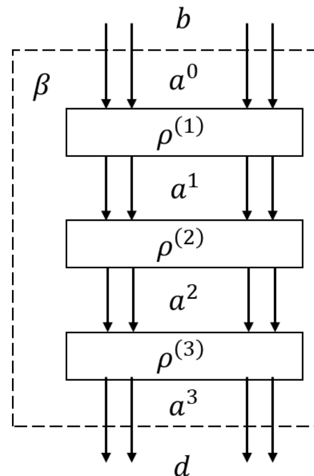

Figure 4 - Example of an iterative Boolean transformation

2.7 Block Ciphers

In cryptography, encryption algorithms are called *block ciphers* [12]. The block cipher applies over an n_b-bit state a set of Boolean permutations. Cipher designers focus on provide a high degree of security to avoid attacks taking advantage of the internal structure of the cipher, whereas providing an efficient implementation of the cipher on as many platforms as possible.

A block cipher B that consists of key-dependant Boolean permutations $\rho[k]$ operating iteratively on a state as presented in (46) is called *iterative block cipher* [13]. A graphical representation shown in Figure 5. The Boolean permutations for this sort of cipher blocks are called *round transformations* [13]. Each of them can be defined as a sequence of more than one different Boolean permutations. The keys used in each round are called *round keys* and are obtained from the cipher key k after a process of expansion. The initial key can be defined as the concatenation of all round keys as shown in (47).

$$B[k] = \rho^{(r)}\left[k^{(r)}\right] \circ \dots \circ \rho^{(1)} k\left[k^{(1)}\right] \tag{46}$$

$$k = k^{(r)} | \dots | k^{(2)} | k^{(1)} \tag{47}$$

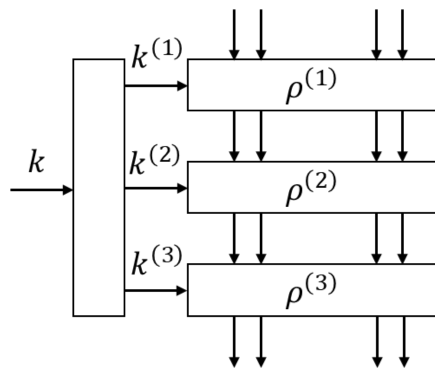

Figure 5 – Example of iterative block cipher with three round transformations

Furthermore, if only one round transformation ρ is defined and used iteratively in the iterative block cipher as shown in (48) the cipher is called an *iterated block cipher* [11].

$$B[k] = \rho\left[k^{(r)}\right] \circ \dots \circ \rho\left[k^{(1)}\right] \tag{48}$$

The key-dependant round transformations in the iterative block cipher can be formed by the alternation of key-independent round transformations $\rho^{(r)}$ and key additions $\sigma\left[k^{(r)}\right]$ as shown in (49). If this is the case and an XOR is used for a simple key addition, then the iterative block cipher is called a *key-alternating block cipher* [14]. An example is given in Figure 6.

$$B[k] = \sigma\left[k^{(r)}\right] \circ \rho^{(r)} \circ \dots \circ \sigma\left[k^{(1)}\right] \circ \rho^{(1)} \circ \sigma\left[k^{(0)}\right] \tag{49}$$

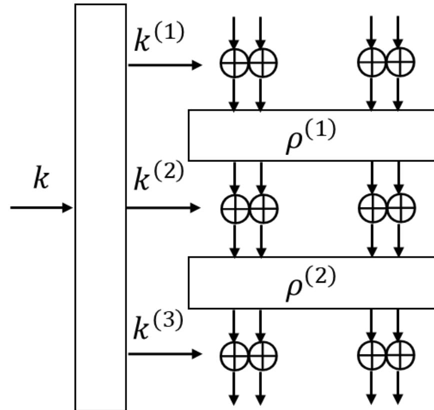

Figure 6 - Example of key-alternating block cipher with two rounds

Moreover, if in the key-alternated block cipher the round transformations ρ are the same, the cipher is called a *key-iterated block cipher* [14].

$$B[k] = \sigma\left[k^{(r)}\right] \circ \rho \circ \dots \circ \sigma\left[k^{(1)}\right] \circ \rho \circ \sigma\left[k^{(0)}\right] \tag{50}$$

2.8 Correlation

A cipher can achieve resistance against linear cryptanalysis if the cipher is designed is such a way that there is no large correlation over rounds. Boolean functions $f(a)$ and $g(a)$ are correlated if their correlation $C(f,g)$ is different from zero [15].

$$C(f,g) = 2 \times Probability\ of\ \big(f(a) = g(a)\big) - 1. \tag{51}$$

A correlation has its minimum value -1 when $f(a)$ is the complement of $g(a)$ as shown in (52) and its maximum value when $f(a) = g(a)$ as shown in (53).

$$C(f,g) = 2 \times Probability\ of\ \big(-g(a) = g(a)\big) - 1 = -1 \tag{52}$$

$$C(f,g) = 2 \times Probability\ of\ \big(g(a) = g(a)\big) - 1 = 1 \tag{53}$$

2.9 Difference Propagation

Let a' be the bitwise difference from n-bit vectors a and a^*, $b = h(a)$, $b^* = h(a^*)$ and $b' = b + b^*$. It can be said that the difference a' propagates to the difference b' through h if and only if (54) occurs [16].

$$h(a) + h(a^*) = b' \tag{54}$$

The probability that (54) occurs if a pair is chosen randomly from the set of all pairs (a, a^*) is called the *difference propagation probability* [16].

$$\mathrm{DP}^{(h)}(a', b') = 2^{-n} \sum_{a} \delta\big(b' + h(a + a') + h(a)\big) \tag{55}$$

3 Security Solutions

Plenty algorithms and methods have been developed though the years and some of them have gained popularity to secure data due to their ease of implementation, reliability, versatility in the sense that they can be implemented in a wide range of devices, and simplicity in the way they can easily be explained and understood.

For an overview of some important security solutions this chapter explains the RSA cryptographic algorithm, the Secure Hash Algorithm-2 (SHA-2) and finally the Hash-based Message Authentication Code (HMAC).

3.1 RSA

The RSA is a public key encryption or encryption algorithm, meaning that a pair of keys are used in the process of encryption and decryption. The key used for encryption is called the *public key,* because it does not require to be confidential, so anyone can encrypt a plaintext using this key. The other key is a confidential key called *private key* and without it the encrypted data cannot be decrypted.

The security of this algorithm relies on the fact that it is a trapdoor one-way permutation, which are functions with special properties [17]. Permutations are addressed in Section 2.6. Permutations of this kind can be computed efficiently only in one direction. In the RSA, this means that a plaintext can be efficiently encrypted. However, to invert the permutation and obtain the pre-processed value, the so-called *trapdoor* information is necessary. In the RSA, the trapdoor information is the private key, which is required for decryption. Without the private

key, obtaining the decrypted data results in great effort and resources. The mathematical explanation behind this algorithm is described below.

For encryption the *modulus* n and the *public exponent* e are required. To obtain n first two chosen prime numbers p and q are multiplied. After that, $\varphi(n)$ is obtained with this numbers.

$$
\begin{gathered}
n = p * q \\
\varphi(n) = (p-1)(q-1)
\end{gathered}
\tag{56}
$$

Then, e is calculated with $\varphi(n)$ and conditions shown in (57).

$$
\begin{gathered}
1 < e < \varphi(n) \\
gcd(e, \varphi(n)) = 1
\end{gathered}
\tag{57}
$$

Finally, plaintext m is encrypted as follows:

$$
\mathcal{E}_{n,e}(m) = m^e \; mod \, n, \;\; m < n. \tag{58}
$$

Example 3.1 Let the prime numbers p and q be $p = 5$ and $q = 13$. So n and $\varphi(n)$ are obtained as follows:

$$n = 5 * 13 = 65$$

$$\varphi(n) = (5-1)(13-1) = 4 * 12 = 48.$$

From these values, e is equal to 5 since:

$$1 < 5 < 48$$

and

$$gcd(5{,}48) = 1$$

If there is a plaintext m equal to 7, the encrypted plaintext is:

$$\mathcal{E}_{65,5}(7) = 7^5 \, mod \, 65 = 37$$

For decryption, d must be found so that:

$$e * d \, mod \, \varphi(n) = 1$$
$$gcd(d, \varphi(n)) = 1 \tag{59}$$

Finally, decryption is achieved with n and d.

$$\mathcal{D}_{n,d}(c) = c^d \, mod \, n \tag{60}$$

Example 3.2 To decrypting the result obtained in the previous example, d and the decrypted ciphertext are obtained as follows:

If

$$5 * d \, mod \, 48 = 1,$$
$$d = 29.$$

Thus,

$$\mathcal{D}_{n,d}(37) = 37^{29} \bmod 65 = 7$$

To decrypt a cipher text only with the private key, d would need to be found. Since e is known from the public key, $\varphi(n)$ is the only value required and it can be obtained with the two prime factors of n, which is also known from the public key. In conclusion, the only values that need to be found are the two prime factors of n, that are p and q. In Examples 3.1 and 3.2, the modulus n is a 2-digit value, however, NIST suggests a modulus n with length of minimum 2048 bits [18]. Consequently, the high number of computational resources to factor the product of two large primer numbers provides high security to the algorithm.

3.2 Cryptographic Hash Functions

Integrity of sensitive data must be secured preventing its modification including insertion or deletion of information. It is assumed that a hash function h is public and it must be a one-way function, in other words, it must be computationally infeasible to find M given the output of $h(M)$. The input of a hash function is a variable-length message, whereas the output is a fixed-length value.

In this first method, message M is hashed to obtain $h(M)$, both are encrypted and transferred. The receiver decrypts the ciphertext, hashes the message and authenticates it by hashing it and comparing the result with the hashed message sent. This method adds a layer to security, since if the key is compromised and the message is modified the adversary must also hash the message to succeed. A graphical illustration of this method is given in Figure 7.

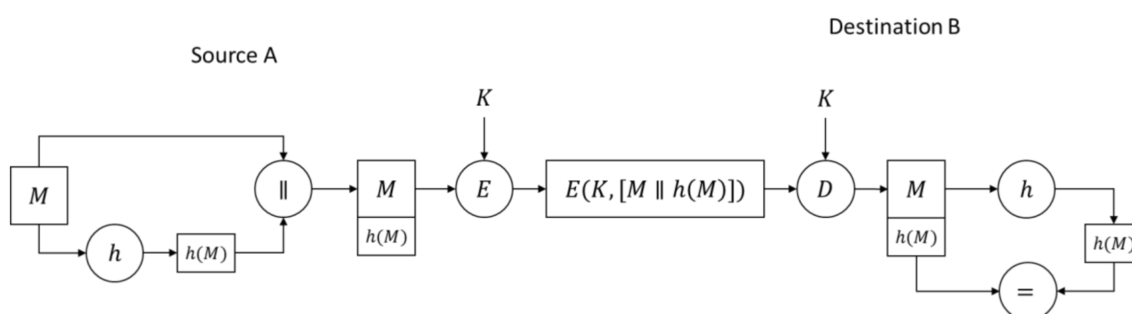

Figure 7 - Message and Hash Code both encrypted.

For some applications that required authentication of large amount of data, encryption of such data might imply higher resource costs. In this case, if the message does not require confidentiality, only the hashed message can be encrypted not only reducing time for encryption and decryption, but also reducing implementation costs. A graphical illustration of this method is given in Figure 8.

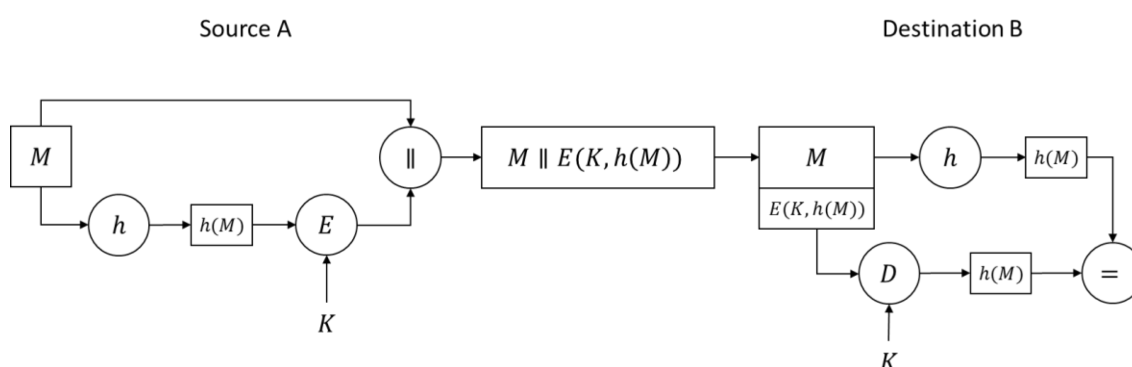

Figure 8 - Encryption of hashed message only

Another method, if confidentiality is not necessary and encryption requires to be avoided, is the addition of a secret value S to the message before this is hashed, hence, this value must be confidential. The receiver authenticates the message adding to it the secret value, hashing both values, and comparing the result with the received hashed value as presented in Figure 9.

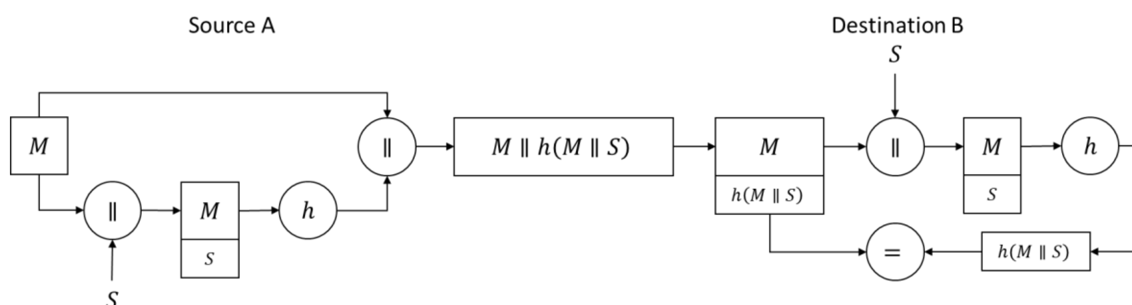

Figure 9 - Hashing message with secret value

If the application requires case confidentiality, symmetric encryption can be implemented to the previous method before the transmission of data. If this is

the case, the same key is used by the receiver to decrypt the data. The key must remain confidential. However, if the key is compromised the message still can be authenticated if the secret value remains confidential. A graphical illustration is provided in Figure 10.

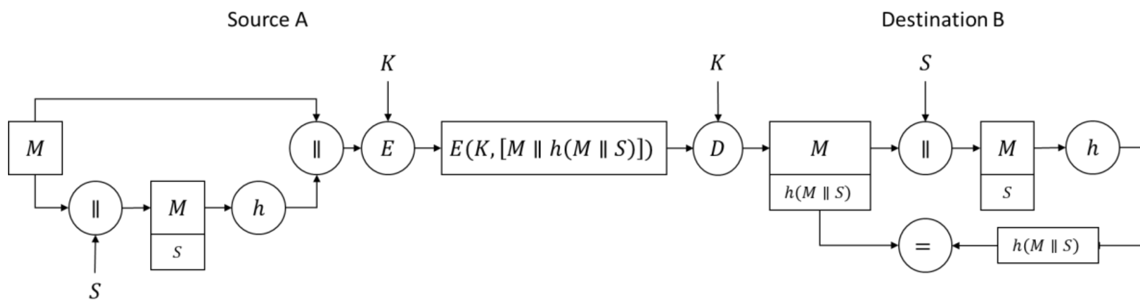

Figure 10 - Confidentiality with encryption and authentication with secret value

The hash functions can also be used to authenticate the message, as well as the source using a technique called *digital signature*. This technique is achieved by means of asymmetric encryption where a public and a private key are employed. This method encrypts the hashed message with the private key and anyone with the public key can decrypt the message and authenticate it. A graphical illustration is presented in Figure 11.

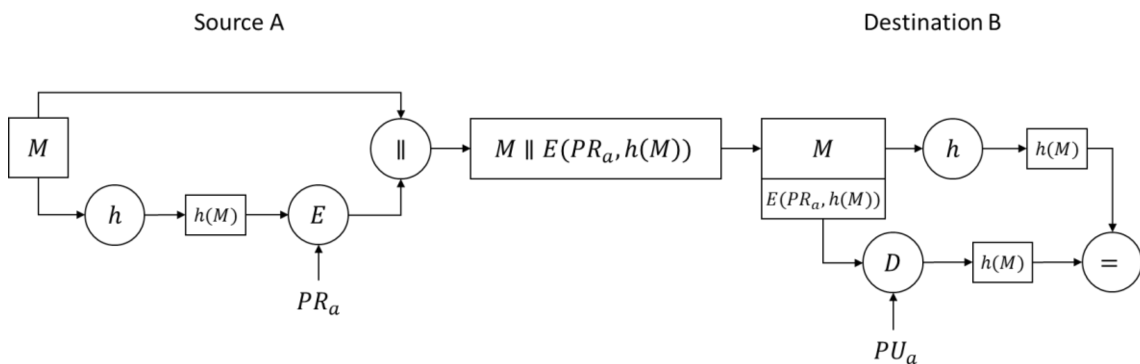

Figure 11 - Hash function with digital signature

Confidentiality can be added to the previous method if symmetric encryption is performed as shown in Figure 12.

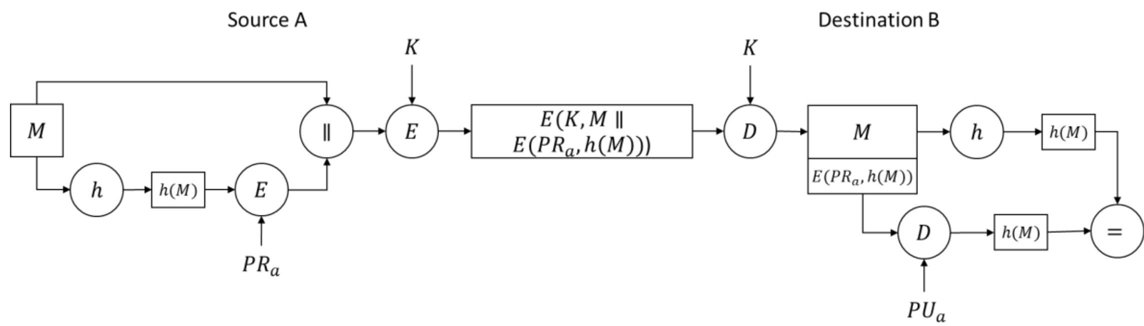

Figure 12 - Hashed function with digital signature and confidentiality

3.3 SHA-512

The Secure Hash Algorithm-512 (SHA-512) is a hash function member of the SHA-2 family which includes five more hash functions SHA-224, SHA-256, SHA-384, SHA-512/224, and SHA-512/256. The numbers in the name of the functions correspond to their output lengths, while the input length is a variable length message.

The first step in the hash function SHA-512 is the preparation of the data to obtain 1024-bit blocks from the message. If required, a defined padding is used as shown in Figure 13.

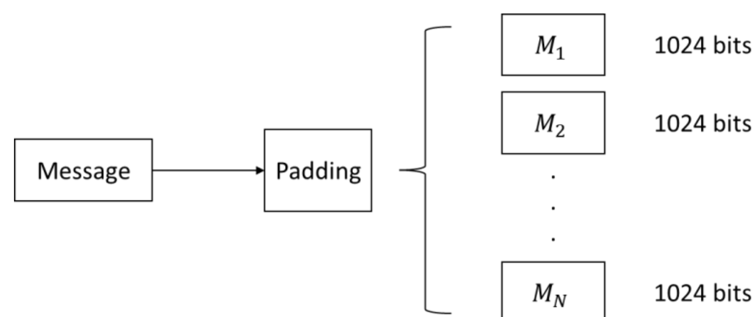

Figure 13 - Message padding for SHA-512

If the last block has length l different than 1024 bits, a padding value p must be found such that

$$(l + p) \bmod 1024 = 896. \tag{61}$$

With value p it is possible to fill the block with the initial value followed by a 1, $p - 1$ zeros and the hexadecimal value of the initial length l filling the 128 remaining bits.

Example 3.3 Let the last block has the data "abc", which in hexadecimal representation is 616263. The data has a length l of 24 bits, which in hexadecimal representation is 18. To meet equation (61), p must be equal to 872. Therefore, the final block is represented in binary as:

616263**8**000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 00000000000000**18**

Once the 1024-bit blocks are prepared, an initialization vector IV is required by the algorithm to be considered vector H_0. The algorithm structure is based on recursive calculations, where the output of each recursion is a new vector H_n. In every recursion, function f requires vector H_n and one of the blocks from the message. After computing function f, its output is added by means of an XOR to vector H_n. The last recursion on the last block leads to the last vector H_N. A graphical illustration of the SHA-512 is given in Figure 14.

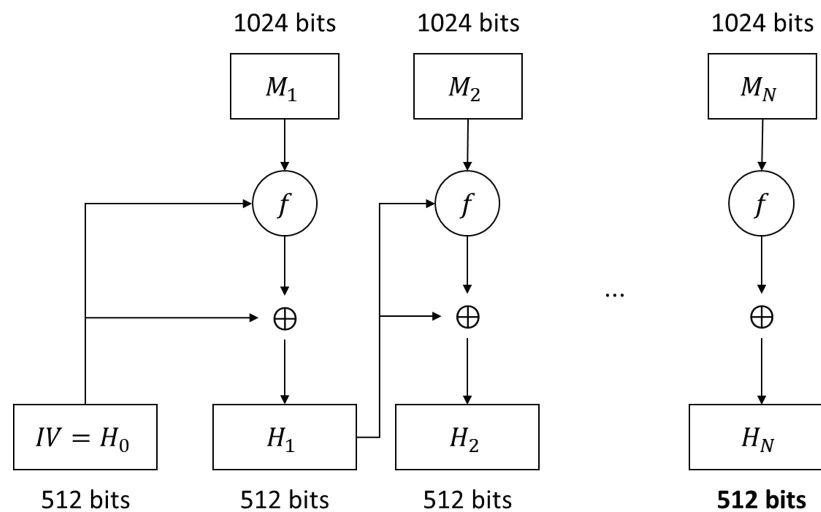

Figure 14 - SHA-512 Architecture

Function f is defined as a recursive application of specific operations. The same operations are performed 80 times. First, 80 64-bit words are required, one for each round. The first 16 words are mapped by the block data M_n as demonstrated in Figure 15.

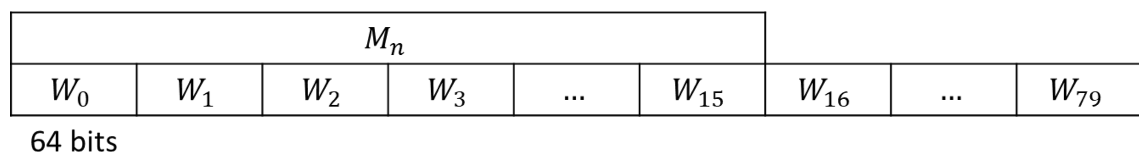

Figure 15 – Message expansion for scheduling of SHA-512

The new words are generated with (62), using (63) and (64). The differences between the modular addition (+) and the bitwise XOR (⊕) must be taken into consideration for the definition of the operations. The exponent w in the module is specified by the word length.

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \bmod 2^w, \quad 16 \leq i \leq 79 \tag{62}$$

$$\sigma_0(x) = ROTR^1(x) \oplus ROTR^8(x) \oplus SHR^7(x) \quad (63)$$

$$\sigma_1(x) = ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \quad (64)$$

Considering that H_n has a length of 512 bits, it can be decomposed into 8 64-bit elements A, B, C, D, E, F, G and H. These elements are recalculated in each of the 80 rounds of f for $0 \leq i \leq 79$ as follows:

- $H \leftarrow G$
- $G \leftarrow F$
- $F \leftarrow E$
- $E \leftarrow D + T_1 \; mod \; 2^w$
- $D \leftarrow C$
- $C \leftarrow B$
- $B \leftarrow A$
- $A \leftarrow T_1 + T_2 \; mod \; 2^w$

Values T_1 and T_2 are obtained with the following equations.

$$T_1 \leftarrow H + \sum\nolimits_1 (E) + f_{if}(E, F, G) + K_i + W_i \bmod 2^{64} \quad (65)$$

$$T_2 \leftarrow \sum\nolimits_0 (A) + f_{maj}(A, B, C) \bmod 2^{64} \quad (66)$$

$$\sum\nolimits_1 (E) = ROTR^{14}(E) \oplus ROTR^{18}(E) \oplus ROTR^{41}(E) \quad (67)$$

$$\sum\nolimits_0 (A) = ROTR^{28}(A) \oplus ROTR^{34}(A) \oplus ROTR^{39}(A) \quad (68)$$

$$f_{if}(E,F,G) = (E \text{ AND } F) \oplus (\text{NOT } E \text{ AND } G) \tag{69}$$

$$f_{maj}(A,B,C) = (A \text{ AND } B) \oplus (A \text{ AND } C) \oplus (B \text{ AND } C) \tag{70}$$

The round constants K_i for SHA-512 can be obtained from the following numbers as stated by NIST [19].

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c7123	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e37	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9beb	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

3.4 HMAC

Authentication of messages to protect integrity of data can be achieved implementing a message authentication code (MAC), which is also called Keyed-Hashed Message Authentication Code (HMAC) when used with cryptographic hash functions [20].

The HMAC requires a secret key on the input to generate an authentication code required to verify integrity of messages and the initialization vector. This algorithm operates the hash function two times, one after the other.

First, to create the authentication code using hash function SHA-512, the message must be padded to reach a length of a multiple of 1024 bits. Another 1024-bit block is concatenated to the padded message. To obtain this block, the value called *ipad* is added via a bitwise XOR with the key. Both values must have length of 1024 bits as well. Thus, the key is padded with zeros on the left to have this length and the *ipad* is equal to the 8-bit value 36 (represented in hexadecimal form) repeated $1024/8$ times to have the required length.

After that, the result of the concatenation is processed with the hash function using the initialization vector IV. The output of the hash function has a length of 512 bits and must be padded to have a length of 1024 bits. To this result the same process is followed with *opad* equal to the 8-bit value $5\mathrm{C}$ (represented in hexadecimal form) repeated $1024/8$ times to have the required length. The final output is a 512-bit value from the second hashed function. A graphical illustration is presented in Figure 16.

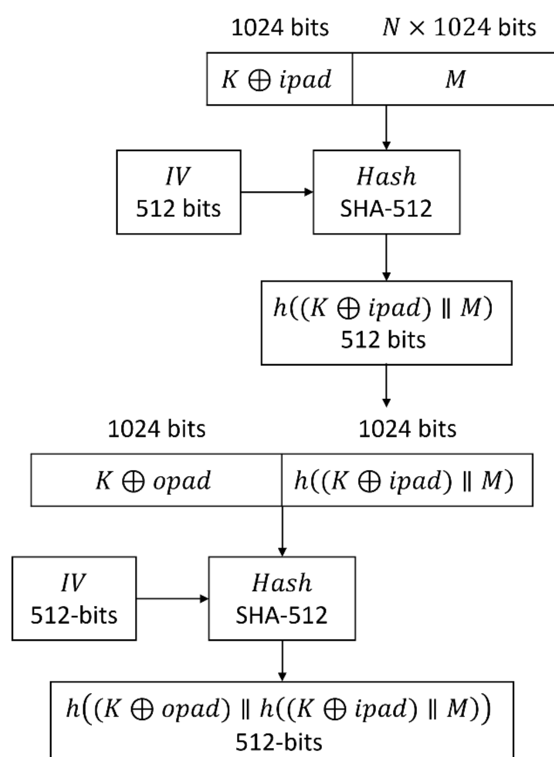

Figure 16 - HMAC architecture

4 Xilinx FPGAs Security Solutions

There are multiple ways in which an FPGA can be attacked [21]. To speed up testing of the design on the desired device security methods might be avoided during verification stage. If security is not addressed carefully after this stage, security holes might be left giving the attacker options to readback the configuration bitstream. The next step is to decode the bitstream from available formats to recover the configuration of the device. Once the bitstream has been extracted, substitution of sections of the bitstream is done to modify the behaviour of the system, this technique is known as spoofing. If the enemy wants to cause the system to malfunction a fault insertion can be done. Another attack is the side-channel attack where the adversary takes advantage of certain properties of the design or the device, such as, power or timing to compromise the system.

Through the years Xilinx has dedicated effort to add generation after generation new security features and has released a series of documents related to security concerns and detailed explanation on how to implement all these features.

In this chapter, Xilinx approach to protect designs for different generations of Xilinx FPGAs will be shown. The complexity of new features and the different security layouts that a designer can consider with modern FPGAs will become visible. Also techniques used in older FPGAs are explained which shows how Xilinx started to address anti-tapering (AT) solutions.

Xilinx offers two different security features for FPGAs: passive and active features [22] [23]. Passive features are all those features that can be implemented without modifying the original design. These solutions do not interfere with the functionality of the design and are already implemented into the

device or are part of the Vivado tool flow, but need to be enabled to provide the desired protection.

In contrast, active security is considered by Xilinx as all those features available for Xilinx FPGAs where the security is part of the design. It is more time-consuming for the designer contrary to selecting passive features, but it takes less time during the configuration of the device.

Even though passive solutions offer a simple implementation, they have their own challenges. One example of a passive solution is the encryption of the bitstream, where only one key to encrypt and decrypt is needed. This feature would need a battery when used with a battery-backed RAM (BBRAM) and key management, which is the most important element to decrypt the bitstream.

The selection of the best methods for a tampering resistant system depends on multiple factors. According to Peterson Ed [22] [23], the impact on the company if the design is tampered must be considered. Therefore, the more the value of the design for the company the more financial resources and effort should be invested to protect the design.

The degree of complexity of the solution selected also depends on the skills, investment, or patience the enemy possesses. It might not be required to invest much on security when the attacker does not have the skills to overcome every security method implemented on the FPGA. However, if the adversary is a well-funded government or an important competing company who is willing to spend significant resources and time to obtain the design, the security level must be the highest according to the possible alternatives.

Lastly, to make the best choice the design stage should be taken into consideration. Comparing several AT methods at the beginning of the project and defining the most suitable ones will result in a proper integration with the design. Additionally, the cost and time for integration will be lower contrasted to when these concerns are not addressed until later stages.

4.1 Spartan-3A/3AN/3A DSP FPGAs

Security solutions against reverse engineering, overbuilding, cloning, and tampering were considered in the architecture of the low-cost Xilinx Spartan-3A/3AN/3A DSP FPGAs. Some of the available methods to protect integrity or confidentiality of designs implemented in these FPGAs are explained in this section based on [24].

4.1.1 Advanced Data Manipulation

The FPGAs mentioned posses a default unique 64-bit device identifier containing a 57-bit value called *Device DNA*. Additional bits can be added to extend the device DNA to be used for security techniques. These additional bits can be stored in the user flash memory, as well as the *Stored Checked Code,* which is the value used to compare the output of the Security Algorithm as shown in Figure 17.

The values stored in the flash memory and the Device DNA could be easily discovered, however the manipulation of this data by a Security Algorithm and a sorter is confidential. The sorter is part of the design data, whereas the Device DNA and the Security Algorithm are contained in the Spartan-3A device. The sorter is defined by a de-multiplexer and a decoded counter used to control the de-multiplexer. After the Device DNA has been extended, the sorter selects the data to be used as input for the Security Algorithm and the not selected data is dumped into the proverbial bit bucket. If the result of the Security Algorithm does not match the Stored Check Code, the design can completely shut off, limit some functionalities, operate for a specific amount of time before stop functioning or lock Flash memory to all zeros to prevent repeated unauthorized access attempts.

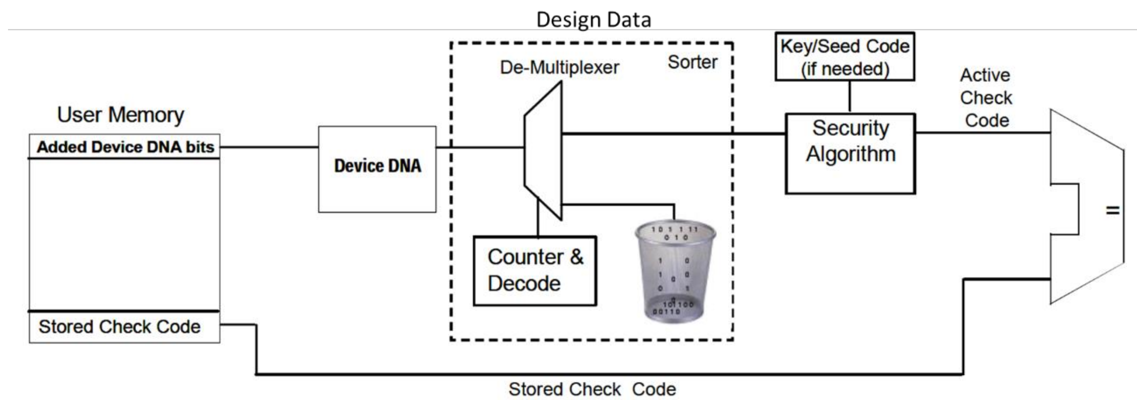

Figure 17 - Data manipulation of Device DNA[24]

4.1.2 Advanced Data Manipulation on the Stored Check Code and Algorithm Control

The data manipulation technique can be expanded by adding an additional output to the de-multiplexer, so that the attacker only sees the extended Device DNA being read by the FPGA. In this way, the attacker not only must reverse engineer the Security Algorithm but also it must find the Stored Check Code, and to do that first the trash bits must be reconstructed.

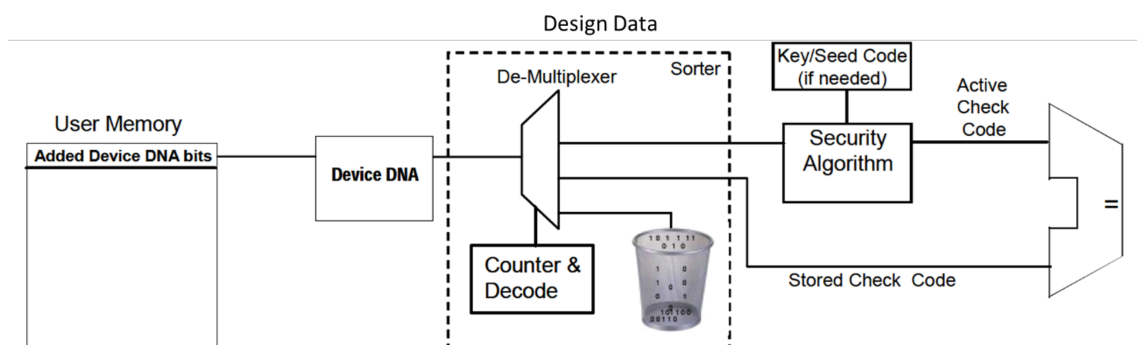

Figure 18 - Data manipulation on Stored Check Code [24]

4.1.3 Adding Fourth Output to De-Multiplexer

A more advance technique can be achieved by adding a fourth output to the de-multiplexer to modify the security key, seed value, or the internal structure of the Security Algorithm which introduces a higher layer of security [24] as shown in Figure 19.

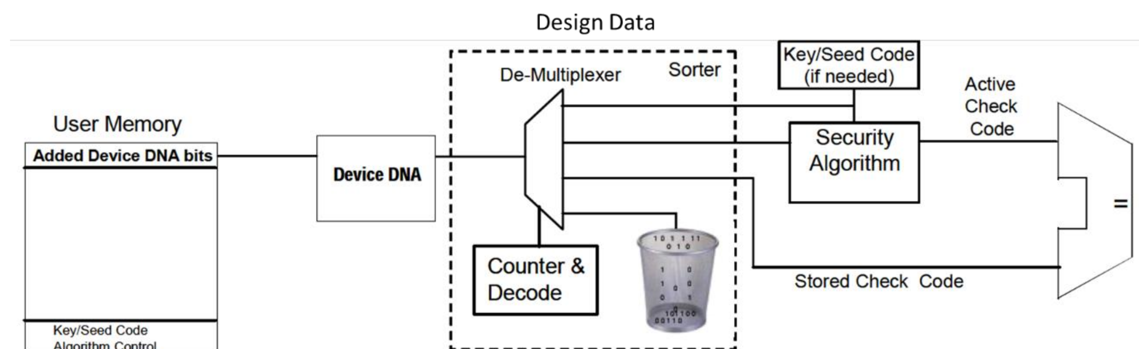

Figure 19 - Adding fourth output to de-multiplexer.

4.2 7 Series FPGAs

Due to higher security requirements enhanced methods to provide security in terms of confidentiality and authentication, compared to devices mentioned in previous sections, can be implemented on Xilinx 7 Series FPGAs. The methods explained for this Series are: bitstream encryption with the Advance Encryption Standard (AES), bitstream authentication using the Key-Hashed Message Authentication Code (HMAC) with a Secure Hash Algorithm (SHA), as well as the traditional Device DNA.

4.2.1 Bitstream Encryption

Important for designers is the confidentiality of their designs, as well as preventing using them in unauthorized devices. For this reason, Xilinx Vivado Design Suite encrypts the bitstream of the design before its deployment on Xilinx 7 series

devices [25]. Therefore, to configure the FPGA, an on-chip decryption engine is required to obtain the bitstream.

The 256-bit key required for encryption and decryption can be generated with the Xilinx Vivado tool or by the user, however, Xilinx recommends key definition by the user through random key generators. The key is either stored in the eFUSE or in a dedicated battery-backed RAM, which is a RAM externally connected to a battery.

Consequently, encrypted designs are unable to be loaded on devices without the correct key or the decryption engine. The encryption algorithm used by Xilinx Vivado is the Advanced Encryption Standard AES-256 in cipher block chaining mode of operation (CBC).

4.2.2 Bitstream Authentication

Encrypting the bitstream with the AES-256 algorithm provides confidentiality to the designs. Moreover, adding another layer of security through authentication of the bitstream prevents loading a tampered or corrupted bitstream into the FPGA.

For authentication, the SHA-256 defined in FIPS PUB-182-2 and an HMAC defined in FIPS PUB-198 published by NIST are employed [22]. Although the AES encryption key is loaded via JTAG, this is not the case for the HMAC key required for authentication. The key along with the configuration bitstream is hashed with the SHA-256. The output of the hash function called *digest* is added to the HMAC key and the configuration bitstream to be encrypted together by the Xilinx tool called BitGen. On the FPGA, the decryption engine decrypts the encrypted bitstream with the AES key stored either in the eFUSE or in the BBRAM to start the process of authentication as explained in Section 4.2.1. The HMAC key and the configuration bitstream are used by the hash function available on the FPGA to provide the digest. The result must be equal to the digest encrypted by BitGen. If both are the same, the start up commands on the FPGA are performed.

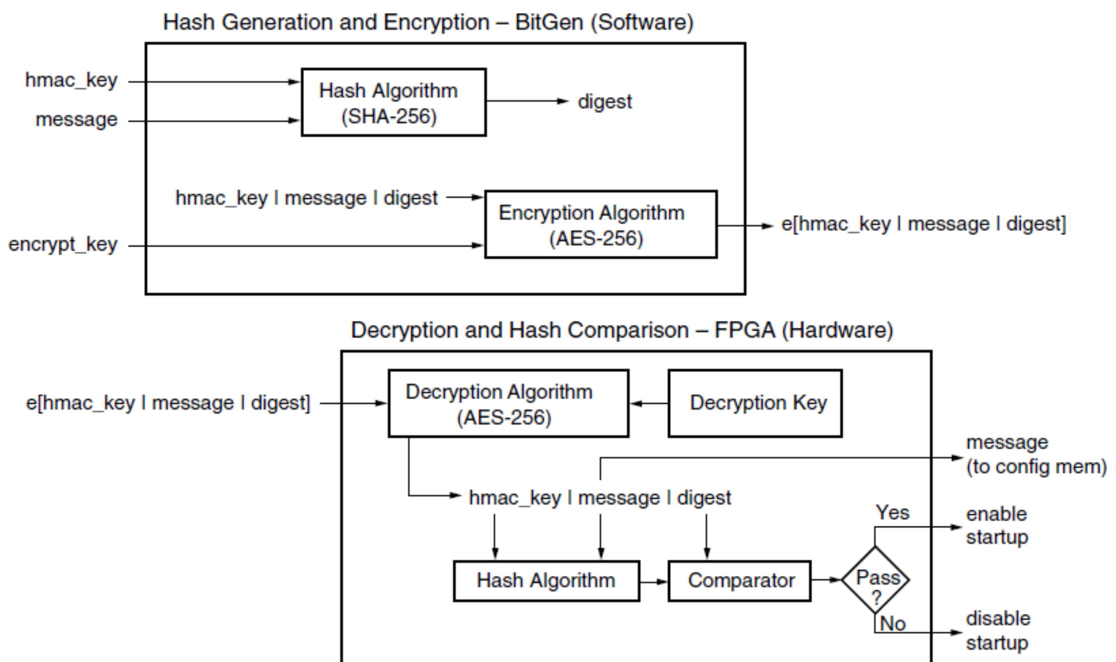

Figure 20 - Hashed message authentication operation [22]

4.2.3 Device Identifier/DNA

In the 7 series FPGAs, a Device DNA is permanently programmed by Xilinx into the FPGA in a similar way as devices mentioned in previous sections. This non-volatile identifier can be employed by security techniques to provide authentication since its value cannot be modified [25]. The Device DNA can be read using it access port design primitive shown Figure 21.

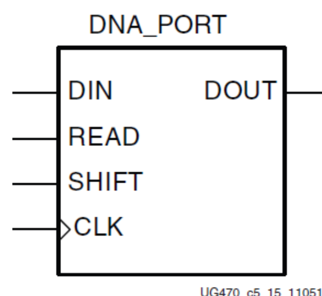

Figure 21 - 7 Series FPGA DNA_PORT design primitive [25]

4.3 UltraScale/UltraScale+ FPGA

The UltraScale/UltraScale+ FPGA family uses modern methos to provide security through confidentiality and authentication of the bitstream. Authentication can be achieved with or without the encryption of the bitstream. If confidentiality is required a symmetric encryption algorithm is employed, otherwise for authentication only an asymmetric algorithm is used.

4.3.1 Bitstream Confidentiality and Authentication (Symmetric)

Rather than using an HMAC key for authentication, this method uses the AES algorithm with the Galois/Counter Mode (GSM) mode of operation and a 256-bit key. This modern algorithm achieves confidentiality and authentication at the same time with a universal hash function inside AES-GCM without the requirement of any other specific key for the hash function [23].

4.3.2 Bitstream Authentication (Asymmetric)

If no confidentiality is needed, the asymmetric encryption algorithm RSA-2048 is used along with the hash function SHA-3 to generate a digital signature used for authentication [23]. Even though an encryption algorithm is employed, it is used for authentication purposes and not to add confidentiality. The digital signature does not require to store the private key in the FPGA since this key is used for encryption. Instead, only the public key is required for authentication on the FPGA. The public key is also authenticated on the device. As a result, to load the configuration bitstream the hashed key stored in the eFUSEs must be the same as the hashed public key contained in the RSA authenticated bitstream and also the result after using the public key to decrypt the digital signature contained in the RSA authenticated bitstream must be the same as the hashed configuration bitstream. A graphical illustration is given in Figure 22.

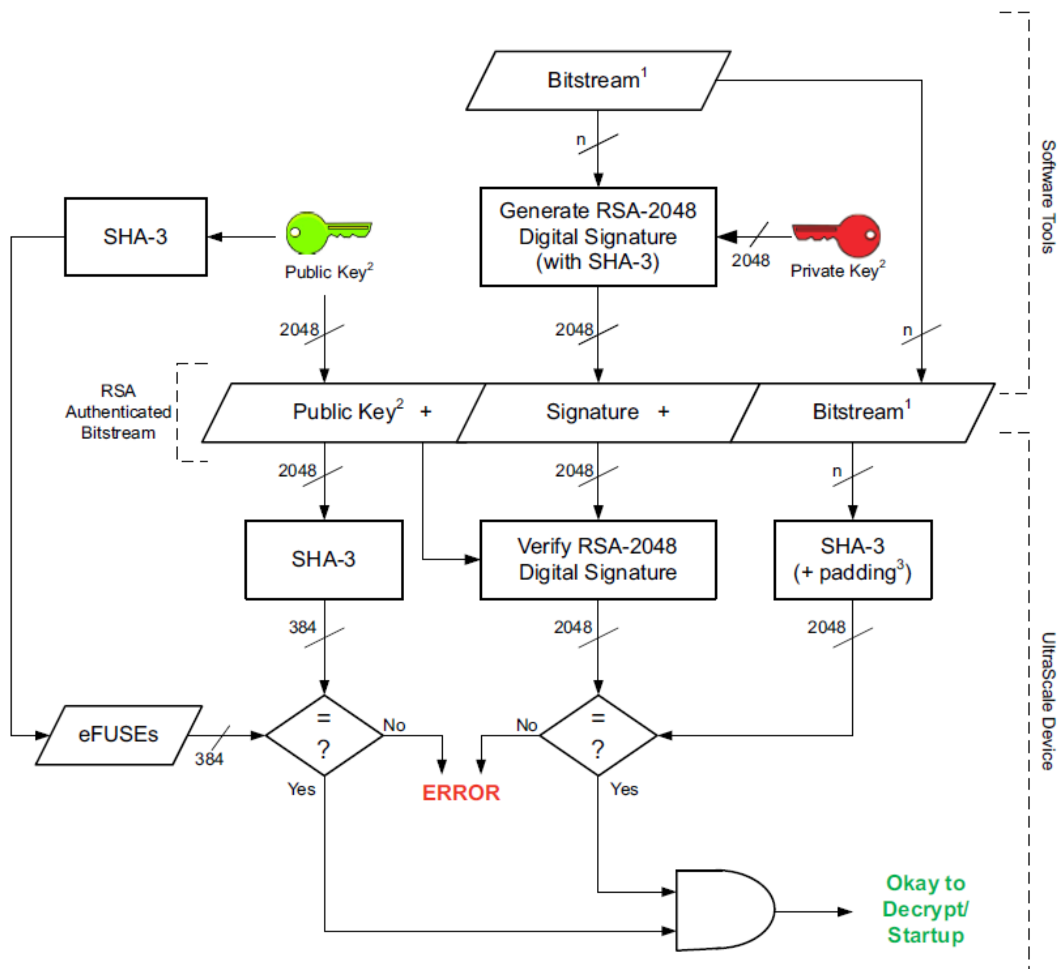

Figure 22 - Bitstream authentication using RSA for digital signature [23]

5 AES

Owing to the fact that an agency of the US Department of commerce called NIST decided to replace the Data Encryption Standard (DES) [26] and Triple DES, in January 1997 a competition was announced to select a new encryption standard that would be called the Advances Encryption Standard (AES) [27]. After 2 rounds the algorithm designed by the Belgian cryptographers Joan Daemen and Vincent Rijmen named as “Rijndael” was selected as the winner, and it was published by NIST as the Federal Information Processing Standard (FIPS) Publication 197 (FIPS 197) [1].

The Rijndael algorithm is considered a *block cipher* algorithm. A *block cipher* is an algorithm that for any given key *k* specifies an encryption algorithm to determine the *n*-bit ciphertext for a given *n*-bit plaintext, additionally it also specifies a decryption algorithm to determine the *n*-bit plaintext with a specified key *k* for a given *n*-bit ciphertext [28].

Even though the Rijndael algorithm was selected to become the AES standard, these algorithms cannot be considered the same. In Rijndael the block length and the key length can be specified to any multiple of 32 bits, from 128 bits to 256 bits [29]. However, AES reduces these specifications setting the block length only to 128 bits and fixing the key length to three possible values: 128, 192 or 256 bits. Therefore, the FIPS standard defines three variants for AES: “AES-128”, “AES-192” and “AES-256” depending on the key length [30].

5.1 Architecture

The AES algorithm can also be defined as a *symmetric cryptosystem* since encryption or decryption is computed using the same key [31]. *Symmetric key* is the term that can be given to a key used for both operations. For this reason, the key must be private and must be shared only by the two parties.

The term for the input differs for each operation as presented in Figure 23. The input block for encryption is called the *plaintext* and the output the *ciphertext* block, whereas for decryption the input block is the *ciphertext* and the output the *plaintext* [32].

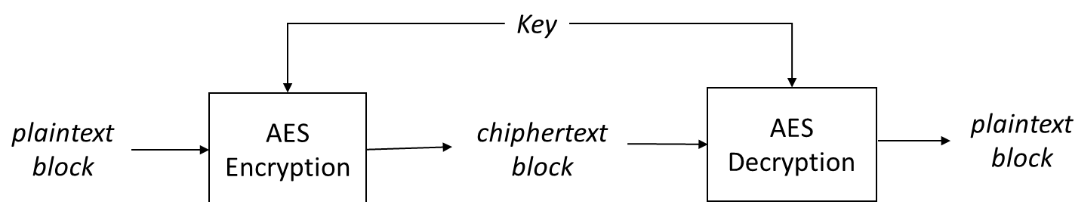

Figure 23 - AES input and output block for encryption and decryption

The 128-bit input block becomes the block named *state*. During the process of encryption and decryption a specific number of iterations called *round transformations* act over the state [33]. The operations contained in the round transformations, called *steps*, process the elements of the state changing the block in every operation, but maintaining the state size. The state can be expressed as an array, in which the number of columns N_b is defined by (71).

$$N_b = \frac{block\ length}{32} = \frac{128}{32} = 4 \tag{71}$$

Considering that each element in the state contains 8 bits and there must be four rows the length for each column is equal to 32. Therefore, the number of columns N_b is calculated dividing the block length, which for AES is 128 bits, by the length of each column, which is 32 bits. Finally, although the Rijndael

algorithm accepts different block lengths, the total number of columns for AES is always four. Figure 24 shows the 128-bit state array formed by a 4-by-4 matrix, in which every element is composed by 8 bits.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Figure 24 - 128-bit state array

The same logic is applied to obtain the number of columns of the key N_k as presented (72). AES supports only 128-, 196- and 256-bit key lengths [30]. Consequently, the number of columns for each variant is 4, 6 and 8 respectively.

$$N_k = \frac{key\ \ length}{32} \tag{72}$$

The key array $K[4][8]$ for the AES-256 variant is shown in Figure 25 and can be represented as a 4-by-8 matrix and each column expressed as $K_0, K_1 \dots K_7$.

K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$	$k_{0,6}$	$k_{0,7}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$	$k_{1,6}$	$k_{1,7}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$	$k_{2,6}$	$k_{2,7}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$	$k_{3,6}$	$k_{3,7}$

Figure 25 - 256-bit key array for the AES-256 variant

5.1.1 Encryption

The algorithm structure for encryption begins with the state block, which is initialized with the 128-bit plaintext block as presented in Figure 26. The first step called AddRoundKey operates on the state using a portion of the ExpandedKey called RoundKey, which is obtained from an operation called KeyExpansion. This operation takes the initial key and through a sequence of calculations expands the key into $N_r + 1$ RoundKeys, where N_r is the number of rounds. The size of the ExpandedKey formed by the generated RoundKeys is defined by the number of columns in the state N_b and in the key array N_k.

The number of rounds N_r for the Rijndael algorithm is obtained from Table 2 depending on the number of columns in the state N_b and in the key array N_k. Due to the fact that for AES N_b is equal to four, N_r can take the values 10 for AES-128, 12 for AES-192 and 14 for the AES-256 variant [34].

	N_b				
N_k	4	5	6	7	8
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Table 2 - Number of rounds N_r depending on N_b (block length / 32) and N_k (key length / 32)

The output of the operation AddRoundKey is the input to the round transformation called *Round*. The first step in *Round* is SubBytes, followed by ShiftRows, MixColumns and again AddRoundKey, which uses the next RoundKey available in the ExpandedKey. *Round* is iterated $N_r - 1$ times and in every iteration a different RoundKey is selected from the ExpandedKey to be used in the step AddRoundKey [35].

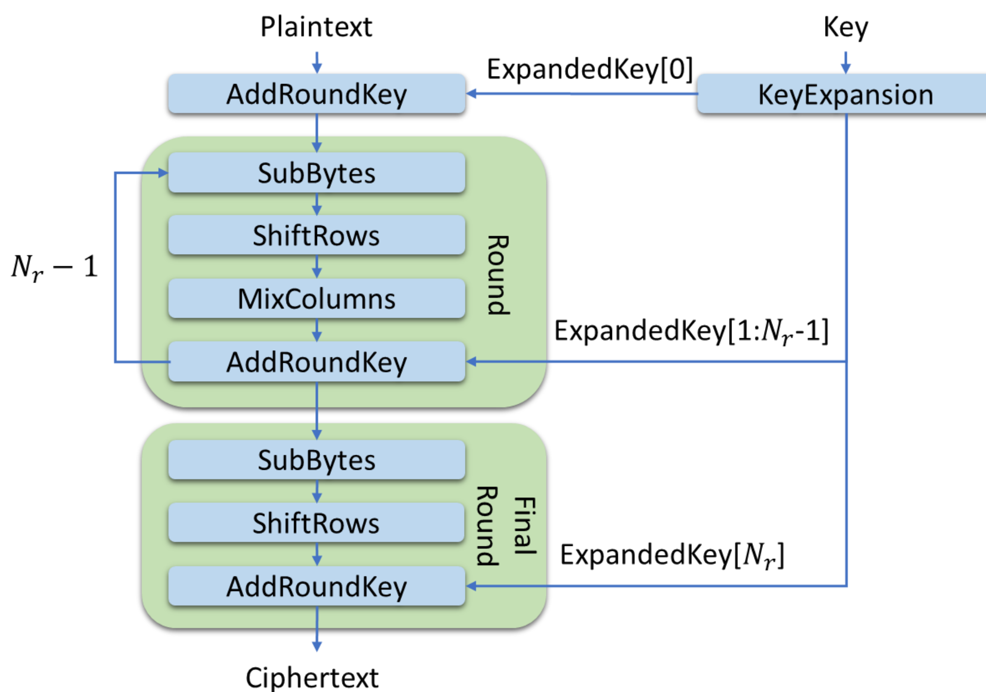

Figure 26 – AES Encryption steps

After all the iterations of round transformation Round the state obtained is the input for the round transformation called FinalRound, which is only performed once. FinalRound consists only of three operations as opposed to Round. Step MixColumns is not considered a step in FinalRound. The RoundKey used in this AddRoundKey step is the last one available in the ExpandedKey [33].

Finally, the encrypted data called *ciphertext* is the state block obtained after the last step in FinalRound. The size of the ciphertext block is the same as the plaintext block and therefore the state block.

5.1.2 Decryption

The decryption structure of a ciphertext block can be seen as the inverse operations of the encryption algorithm but in the opposite way as presented in Figure 27 [36]. Although the first operation for decryption is the same as for encryption (AddRoundKey), in decryption this operation belongs to a round

transformation (InvFinalRound) and the RoundKeys expressed as elements of the ExpandedKey are used backwards compared to the order used for encryption.

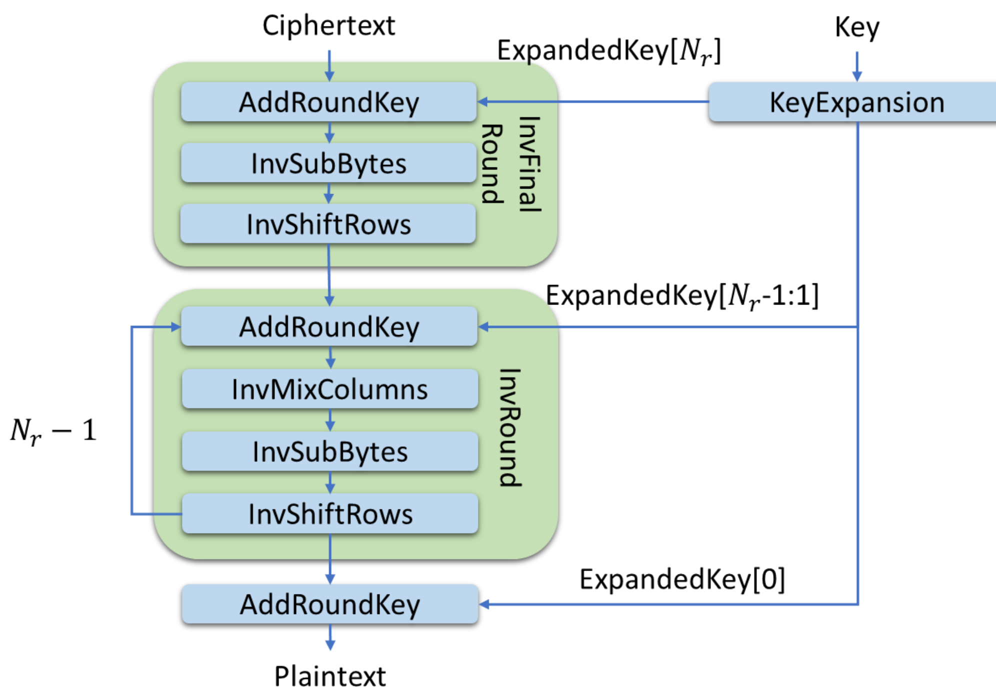

Figure 27 – AES Decryption Steps

In spite of the fact that the plaintext can be obtained inverting the order of the encryption algorithm and using the inverted operations (InvSubBytes, InvShiftRows and InvMixColumns), it is important to mention that the order of steps InvSubBytes and InvShiftRows is irrelevant, so the order does not need to be inverted [37].

5.2 Key Schedule

As explained in previous sections AddRoundKey requires a RoundKey to operate on the state and these RoundKeys are portions of the ExpandedKey, which is derived from the initial key. The process of expanding the key into a larger array and selecting the correct portion of the ExpandedKey to be used as RoundKey receives the name of key schedule [38].

5.2.1 Key Expansion

From the initial key new elements are calculated to create the ExpandedKey. The total number of bits can be determined by the number of times a RoundKey is used, in other words the number of times AddRoundKey is performed, this is $N_r + 1$ times. Considering that the size of a RoundKey must be equal to the size of the state block, the number of bits in the ExpandedKey can be expressed by (*73*) [39].

$$Number\ of\ bits\ in\ ExpandedKey = block\ length \times (N_r + \mathbf{1}) \tag{73}$$

Since AES operates on blocks of 128 bits, the number of bits in the ExpandedKey for AES-256, in which N_r is equal to 14, is shown in (74).

$$\begin{aligned} Number\ of\ bits\ in\ ExpandedKey\ for\ AES - 256 &= 128 \times (14 + \mathbf{1}) \\ &= 1{,}920\ bits \end{aligned} \tag{74}$$

The bits in the ExpandedKey can be represented as an array denoted by W as presented in Figure 28, where every element $w_{i,j}$ is composed by a byte. The number of rows in the array is four as it is in the state and RoundKeys. As a result, the number of columns can be determined by dividing the total number of bits by the 32 bits contained in every column W_j or considering the fact that $N_r + 1$ RoundKeys are required and each of them consisting of four columns, given the number of columns in the state. Thus, the size of the array can be represented as $W[4][N_b(N_r + 1)]$.

It is important to remark that column one W_1 is actually the second column in the array and not the first one, which is W_0. For this reason, the last column is denoted by $W_{N_b(N_r+1)-1}$.

W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9		$W_{N_b(N_r+1)-1}$
$w_{0,0}$	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{0,7}$	$w_{0,8}$	$w_{0,9}$		$w_{0,N_b(N_r+1)-1}$
$w_{1,0}$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$	$w_{1,7}$	$w_{1,8}$	$w_{1,9}$	…	$w_{1,N_b(N_r+1)-1}$
$w_{2,0}$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$	$w_{2,7}$	$w_{2,8}$	$w_{2,9}$		$w_{2,N_b(N_r+1)-1}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$	$w_{3,7}$	$w_{3,8}$	$w_{3,9}$		$w_{3,N_b(N_r+1)-1}$

Figure 28 - ExpandedKey array W

The size of the ExpandedKey for AES-256 can be represented as $W[4][60]$. The array is shown in Figure 29.

W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9		W_{58}	W_{59}
$w_{0,0}$	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{0,7}$	$w_{0,8}$	$w_{0,9}$		$w_{0,58}$	$w_{0,59}$
$w_{1,0}$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$	$w_{1,7}$	$w_{1,8}$	$w_{1,9}$	…	$w_{1,58}$	$w_{1,59}$
$w_{2,0}$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$	$w_{2,7}$	$w_{2,8}$	$w_{2,9}$		$w_{2,58}$	$w_{2,59}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$	$w_{3,7}$	$w_{3,8}$	$w_{3,9}$		$w_{3,58}$	$w_{3,59}$

Figure 29 - ExpandedKey array W for AES-256

The columns in the ExpandedKey are obtained in four different ways depending on the column to be calculated:

1. For first N_k columns: $\{W_0, W_1, \dots W_{N_k-1}\}$
2. For columns multiples of N_k: $\{W_{N_k}, W_{2N_k}, \dots W_{N_b(N_r+1)-4}\}$
3. For columns multiples of four that are not multiples of N_k, if $N_k > 6$: $\{W_{N_k+4}, W_{2N_K+4}, \dots W_{N_b(N_r+1)-8}\}$
4. For all the remaining columns: $\{W_{N_K+1}, W_{N_k+2}, \dots W_{N_b(N_r+1)-1}\}$

The first N_k columns in the *ExpandedKey* are simply filled with the initial key as presented in Figure 30.

W_0	W_1		W_{N_k-1}	W_{N_k}		$W_{N_b(N_r+1)-1}$
$w_{0,0}$	$w_{0,1}$		w_{0,N_k-1}	w_{0,N_k}		$w_{0,N_b(N_r+1)-1}$
$w_{1,0}$	$w_{1,1}$	…	w_{1,N_k-1}	w_{1,N_k}	…	$w_{1,N_b(N_r+1)-1}$
$w_{2,0}$	$w_{2,1}$		w_{2,N_k-1}	w_{2,N_k}		$w_{2,N_b(N_r+1)-1}$
$w_{3,0}$	$w_{3,1}$		w_{3,N_k-1}	w_{3,N_k}		$w_{3,N_b(N_r+1)-1}$

Figure 30 - Key expansion for first N_k columns

A representation of the $\mathrm{ExpandedKey}$ in AES-256 stressing the first N_k columns is shown in Figure 31.

W_0	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9		W_{59}
$w_{0,0}$	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{0,7}$	$w_{0,8}$	$w_{0,9}$		$w_{0,59}$
$w_{1,0}$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$w_{1,5}$	$w_{1,6}$	$w_{1,7}$	$w_{1,8}$	$w_{1,9}$	…	$w_{1,59}$
$w_{2,0}$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$w_{2,5}$	$w_{2,6}$	$w_{2,7}$	$w_{2,8}$	$w_{2,9}$		$w_{2,59}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$w_{3,5}$	$w_{3,6}$	$w_{3,7}$	$w_{3,8}$	$w_{3,9}$		$w_{3,59}$

Figure 31 – Key expansion in AES-256 for the first N_k columns

Columns W_j, where j is multiple of N_k are obtained adding the column N_k positions earlier W_{j-N_k} to the previous column W_{j-1} after using the S-Box with a cyclic rotation within the column and adding to the first row a round constant RC depending on the specific column to be calculated and N_k. A visual representation is exhibit in Figure 32.

W_{j-N_k}		$\mathrm{rot}(S_{RD}[W_{j-1}])$				W_j
$w_{0,j-N_k}$		$S_{RD}[w_{1,j-1}]$		$\mathrm{RC}[j/N_k]$		$w_{0,j}$
$w_{1,j-N_k}$	+	$S_{RD}[w_{2,j-1}]$	+	00	=	$w_{1,j}$
$w_{2,j-N_k}$		$S_{RD}[w_{3,j-1}]$		00		$w_{2,j}$
$w_{3,j-N_k}$		$S_{RD}[w_{0,j-1}]$		00		$w_{3,j}$

Figure 32 - Key expansion for columns W_j, where j is multiple of N_k

The RC values can be selected from Table 3.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathrm{RC}[i]$	00	01	02	04	08	10	20	40	80	1B	36	6C	D8	AB	4D

Table 3 - Round Constant RC table

The operation to obtain the first column multiple of N_k for AES-256 can be exemplified by Figure 33. The set of columns obtained using this computation for AES-256 is represented by $\{W_8, W_{16}, \dots W_{56}\}$.

W_0		$\mathrm{rot}(S_{RD}[W_7])$		$\mathrm{RC}[1]$		W_8
$w_{0,0}$		$S_{RD}[w_{1,7}]$		01		$w_{0,8}$
$w_{1,0}$	+	$S_{RD}[w_{2,7}]$	+	00	=	$w_{1,8}$
$w_{2,0}$		$S_{RD}[w_{3,7}]$		00		$w_{2,8}$
$w_{3,0}$		$S_{RD}[w_{0,7}]$		00		$w_{3,8}$

Figure 33 - Key expansion for column W_8 for AES-256

When $N_k > 6$ columns W_j, where j is multiple of four and not multiple of N_k, are obtained adding the column N_k positions earlier W_{j-N_k} and the previous column W_{j-1} after using the S-Box as demonstrated in Figure 34.

W_{j-N_k}		$S_{RD}[W_{j-1}]$		W_j
$w_{0,j-N_k}$		$S_{RD}[w_{0,j-1}]$		$w_{0,j}$
$w_{1,j-N_k}$	+	$S_{RD}[w_{1,j-1}]$	=	$w_{1,j}$
$w_{2,j-N_k}$		$S_{RD}[w_{2,j-1}]$		$w_{2,j}$
$w_{3,j-N_k}$		$S_{RD}[w_{3,j-1}]$		$w_{3,j}$

Figure 34 - Key expansion for column W_j when $j \ mod \ N_k = 4$

The operation to obtain the first column multiple of four and not of N_k for AES-256, where N_k is equal to eight, can be exemplified by Figure 35. The set of columns obtained using this computation for AES-256 is represented by $\{W_{12}, W_{20}, \dots W_{52}\}$.

W_4		$S_{RD}[W_{11}]$		W_{12}
$w_{0,4}$		$S_{RD}[w_{0,11}]$		$w_{0,12}$
$w_{1,4}$	+	$S_{RD}[w_{1,11}]$	=	$w_{1,12}$
$w_{2,4}$		$S_{RD}[w_{2,11}]$		$w_{2,12}$
$w_{3,4}$		$S_{RD}[w_{3,11}]$		$w_{3,12}$

Figure 35 - Key expansion for column W_{12} for AES-256 when $j \ mod \ N_k = 4$

For any other columns in the ExpandedKey array the operation to follow is shown in Figure 36. The columns required are the column N_k positions earlier W_{j-N_k} and the previous column W_{j-1}. Figure 37 shows the operation to obtain W_9 for AES-256.

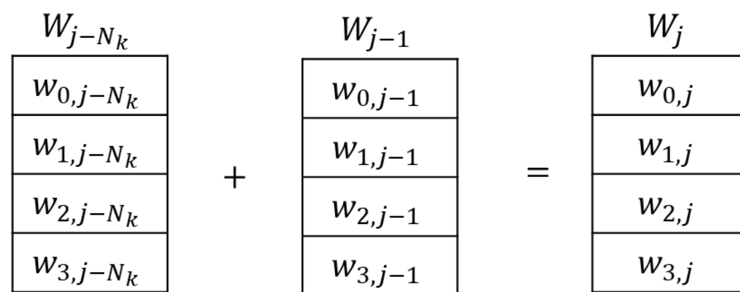

Figure 36 - Key expansion for the rest of the columns in W

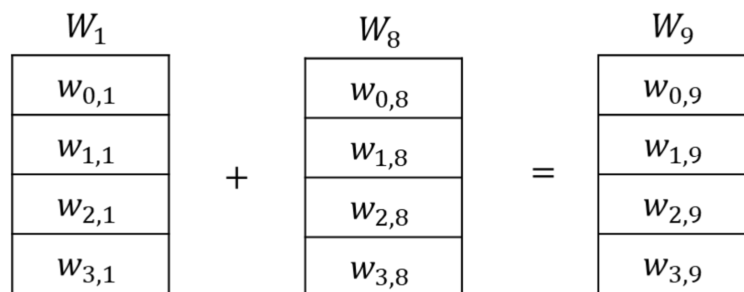

Figure 37 - Key expansion for column W_9 for AES-256

5.2.2 Round Key Selection

The ExpandedKey conformed by $N_b(N_r + 1)$ columns must be divided into $N_r + 1$ columns to be used as RoundKeys in the AddRoundKey step [40]. Therefore, every RoundKey consists of N_b columns and can be represented as a portion of the ExpendedKey. This is presented in (75). In (76), RoundKey i for AES is presented and in (77) the first RoundKey for AES.

$$ExpandedKey[i] = w[\cdot][N_b i] \parallel w[\cdot][N_b i + 1] \parallel \cdots \parallel w[\cdot][N_b(i + 1) - 1],\quad 0 \leq i \leq N_r \tag{75}$$

$$AES\ ExpandedKey[i] = w[\cdot][4i] \parallel w[\cdot][4i + 1] \parallel \cdots \parallel w[\cdot][4(i + 1) - 1],\quad 0 \leq i \leq N_r \tag{76}$$

$$AES\ ExpandedKey[0] = w[\cdot][0] \parallel w[\cdot][1] \parallel w[\cdot][2] \parallel w[\cdot][3] \tag{77}$$

A graphical representation of the ExpandedKey divided into RoundKeys is shown in Figure 38 and in Figure 39 the RoundKeys for AES.

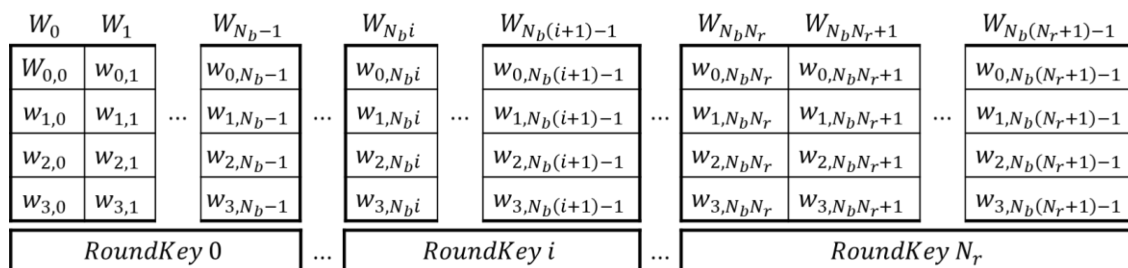

Figure 38 - ExpandedKey divided into RoundKeys

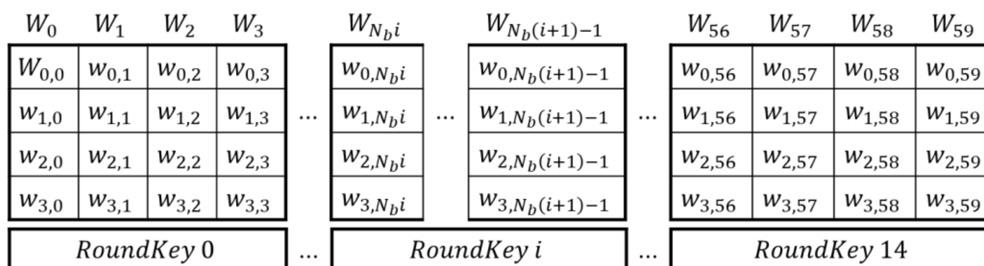

Figure 39 - ExpandedKey divided into RoundKeys for AES-256

5.3 AddRoundKey

In this operation a RoundKey selected from the ExpandedKey is added to the state via a bitwise XOR operation [41] as shown in Figure 40. The inverse of AddRoundKey is itself, for this reason, this operation is also used for decryption [42].

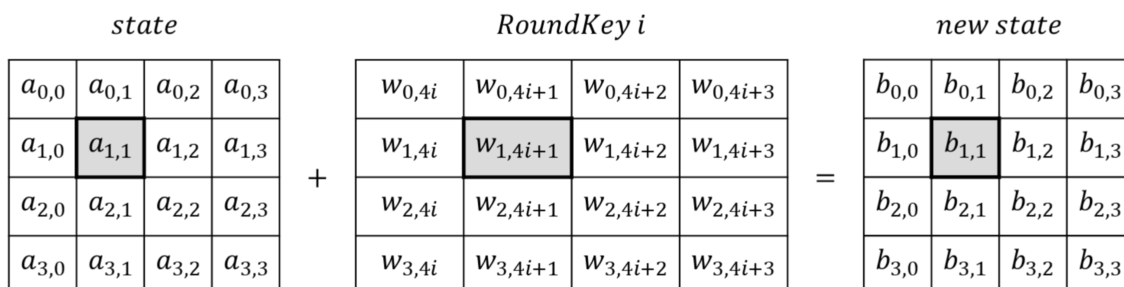

Figure 40 - AddRoundKey step

5.4 SubBytes

In this step every element within the state is substituted with a new element. The substitution is done using the S-box $\mathrm{S_{RD}}$ shown as a tabular representation in Table 4. The S-box maps one-to-one every possible value that a byte can have with a different byte. For this reason, the step is a bricklayer permutation [43]. A graphical representation of the $\mathrm{SubBytes}$ step is presented in Figure 41.

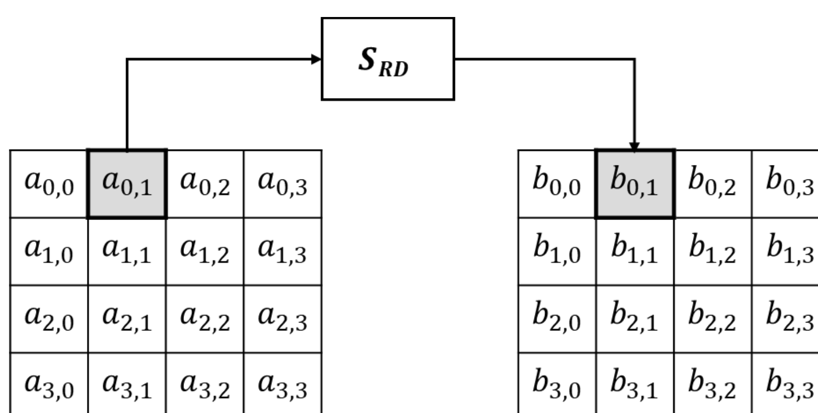

Figure 41 - $\mathrm{SubBytes}$ *step operating on individual bytes.*

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	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	AC	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	79
	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	AB	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 4 - Tabular representation of Rijndael S-box $S_{RD}(xy)$

Detailed description and explanation of the Rijndael S-box is given in Section 6.3.1.

5.5 InvSubBytes

The $\mathrm{InvSubBytes}$ is defined in the same manner as the $\mathrm{SubBytes}$ but using the inverse S-box $\mathrm{S}_{\mathrm{RD}}^{-1}$ shown in Table 5 [43]. Figure 42 presents the substitution of elements through the Rijndael inverse S-box transformation.

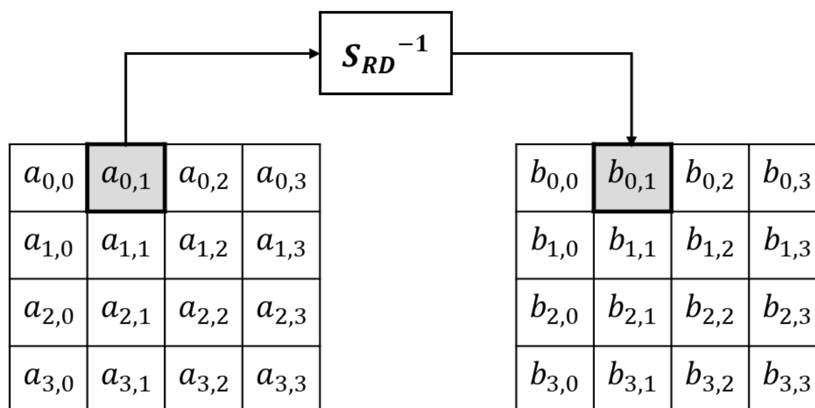

Figure 42 - InvSubBytes step operating on individual bytes.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	7B
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 5 - Tabular representation of Rijndael inverse S-box $S_{RD}^{-1}(xy)$

Detailed description and explanation of the Rijndael inverse S-box is given in Section 6.3.1.

5.6 ShiftRows

The step is defined as a byte transposition over the state. The rows of the state are cyclically shifted to the left over specific offsets as shown in Figure 43. The offset for each row depends on the number of columns of the state as presented in Table 6. Since AES is conformed by four columns, its offsets are emphasised in the table. Therefore, a byte a in row i and column j is moved to column $(j - C_i)\text{mod}N_b$ [44].

$$b_{i,(j-C_i\text{mod}N_b)} = a_{i,j} \tag{78}$$

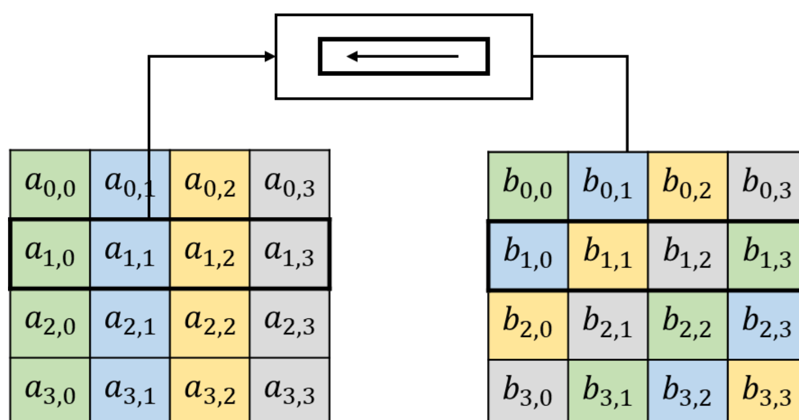

Figure 43 – ShiftRows operating on rows of state.

N_b	C_0	C_1	C_2	C_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Table 6 - Shift offsets for different block lengths

A detailed explanation regarding the design of the step is provided in Section 6.3.4.

5.7 InvShiftRows

The InvShiftRows step is the inverse byte transposition defined in ShiftRows as shown in Figure 44. The offsets used in the transformation are the same, however, rows are cyclically shifted to the right compared to the ShiftRows step [44]. Hence, the byte position is defined as

$$b_{i,(j+C_i \mathrm{mod} N_b)} = a_{i,j}. \tag{79}$$

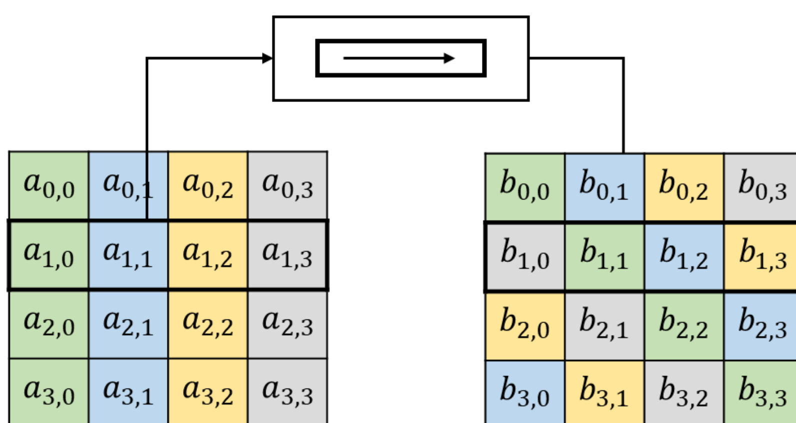

Figure 44 - ShiftRows operating on rows of state.

5.8 MixColumns

This step is defined as a bricklayer permutation operating independently on columns of the state. The columns at the output are obtained through modular multiplication between a fixed polynomial represented as a matrix and the columns at the input as shown in Figure 45.

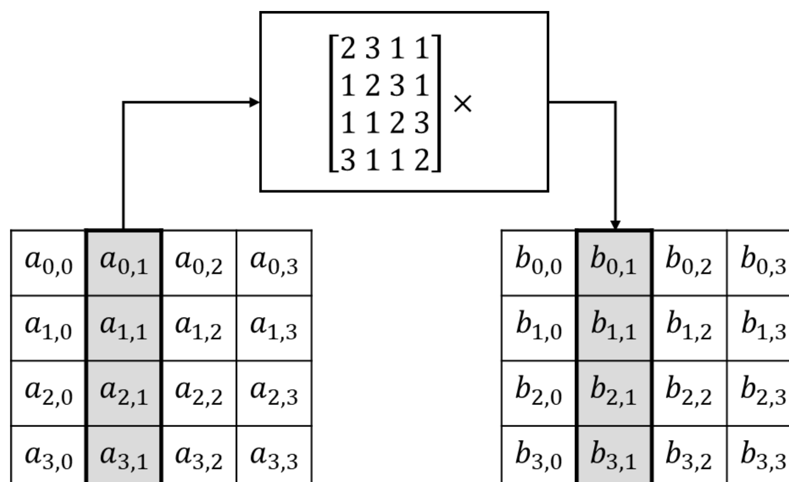

Figure 45 – MixColumn *step operating on column of state.*

Explanation about the fixed polynomial is given in Section 2.5 and explanation regarding the design of the step in Section 6.3.3.

5.9 InvMixColumns

The InvMixColumns step operates in the same manner as the MixColumns. However, the fixed polynomial used in this step is the inverse polynomial of the one used in the MixColumns step. A graphical ilustration is presentated in Figure 46.

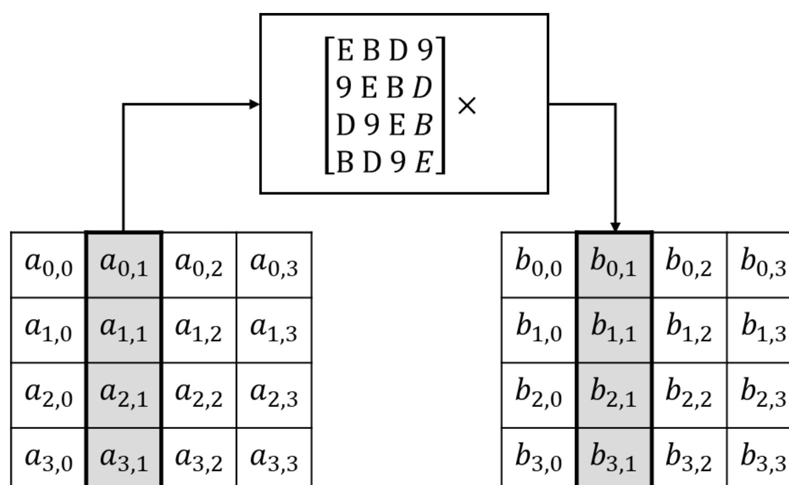

Figure 46 - InvMixColumn *step operating on column of state.*

6 Design of Rijndael

This chapter provides detailed information about the path followed by Joan Daemen and Vincent Rijmen to design the Rijndael block cipher. Topics such as the selection for the cipher structure, transformations to be used, operations to avoid, optimal number of rounds and security goals are addressed to provide greater insight into the design.

6.1 General Criterion

The important property when it comes to ciphers has to be security. Designers must be aware of the weaknesses that the internal structure of the cipher has, to avoid attacks taking advantage of it. When these cryptanalytic attacks are properly addressed the workload necessary to succeed would be the same one as for an exhaustive search of the key [45].

The design had to achieve that the same amount of effort and storage was used for all possible attacks for most possible block ciphers. To achieve this, approach to security was based on two concepts: the cipher had to be K-secure and hermetic. A cipher is called K-secure if there are no key-recovery attacks faster than exhaustive search of the key and if the cipher does not have particular symmetry properties or weak keys, therefore, if there are no successful related-key attacks. To make the cipher hermetic, the must be no advantages of attacking specific block ciphers with same block and key length [46]. This means, that the internal structure cannot be used against the cipher.

Since ciphers can be employed in a wide number of industries, the resources for its implementation must be considered, such as energy

consumption, speed and area required [47]. For a software application, the amount of working memory and memory to store the program is relevant. Moreover, these considerations must be valid on wide range of processors [48].

Depending on the implementation requirements the key might have to be generated more than once, or even for every time there is a new input. This is the reason why the agility to generate new expanded keys is important to be considered [49].

The designer must find the balance between efficiency and security. On the one hand, the use of more resources can lead to a more secure cipher. On the other hand, focusing only on efficiency creates vulnerabilities. The cipher should be designed to achieve maximum efficiency with a reasonable security margin.

To facilitate the implementation of Rijndael, the designers considered using as less operations as possible enough to satisfy the requirements above mentioned. These operations had to be easy to explain to avoid misunderstandings during implementation [50]. With this simplicity of specification, more people would find its implementation attractive and hence, after different implementations of the cipher by more people it would gain more credibility.

Ciphers also gain credibility through analysis against different attacks. In the same way, the cipher had to attract people for its analysis from a mathematical approach, so its properties to provide security had to be easy to demonstrate and understand [50].

6.2 Structure Selection

The designer's approach to achieve simplicity was through symmetry and the choice of operations. Iterating the same key-dependant round transformation except for the last round allows decryption to have the same structure as encryption [51]. A risk of having high symmetry on a cipher are slide attacks,

which exploits the property of symmetry in its internal structure. Consequently, the design had to find a way to avoid high symmetry throughout all the cipher. The measure taken was avoiding high symmetry in the key schedule [52].

The element considered as the one that gives strength to the cipher is the element that adds nonlinearity, which is an S-box [53]. However, linear steps such as diffusion among other properties within ciphers provide the resistance required against differential and linear cryptanalysis, which is the theoretical basis to analyse security in iterative clock ciphers [54]. The way in which these properties can be added is by first selecting the proper optimization approach, which also defines the number of rounds needed.

Optimization is performed on the worst-case behaviour of the cipher. The maximum input-output correlation defines the worst-case scenario for linear cryptanalysis, while for differential cryptanalysis the worst-case scenario is defined by the maximum differential propagation probability. Local optimization considers the worst-case behaviour of one round, whereas in global optimization a sequence of rounds is considered to determine the worst-case behaviour to [55].

A disadvantage of using local optimization is the expensive nonlinear functions required to improve worst-case behaviour. In Rijndael, this was achieved applying global optimization making use of symmetry and alignment properties, but more importantly a design approach called the *wide trail strategy* [55]. With this approach every component within the cipher has a concrete function and thus is defined with regard to nonlinearity and diffusion. In addition, nonlinear functions are not as costly as they are for local optimization. To simplify linear and differential cryptanalysis for optimization a key-alternating cipher structure was selected to be used for the cipher, as this structure allows analysis regardless of the key due to its simple addition through a bitwise XOR operation [56].

6.3 Transformations Selection

After selecting a key-alternating cipher structure as the structure for Rijndael the transformations in its rounds were chosen to combine resistance and efficiency. This was achieved with the design approach called the *wide trail strategy*, which defines the round as a sequence of two invertible steps [57].

The first transformation in the round transformation ρ is a local nonlinear transformation γ. The term local to define a transformation means that specific bits at the output depend only on specific bits at the input, while neighbouring bits at the input are only considered by neighbouring bits at the output. The second transformation is a linear mixing transformation that provides high diffusion [57]. The sequence of transformations is shown in (80) and a graphical illustration is given in Figure 47, where state a consists of elements $[a_1\ a_2\ a_3]$ with two bits each and after the round transformation a key addition $\sigma[k]$ is performed.

$$\rho = \lambda \circ \gamma \tag{80}$$

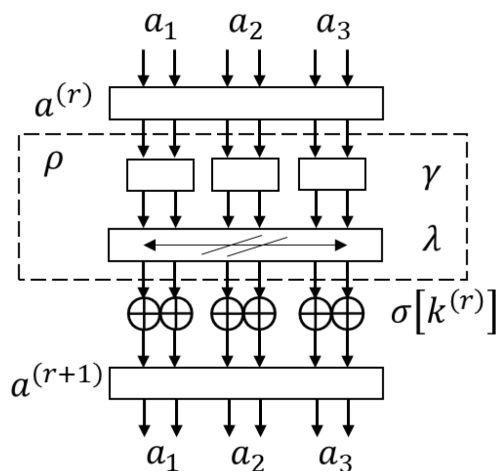

Figure 47 - Example of a $\gamma\lambda$ round structure

6.3.1 Local Nonlinear Transformation γ

The first transformation in the sequence of steps specified by the wide trail strategy is the local nonlinear transformation γ.

This transformation achieves nonlinearity using an invertible nonlinear substitution box called S-box. The transformation is defined as a bricklayer permutation that operates on induvial bytes, so it satisfies the property of being local.

$$b = \gamma(a) \Leftrightarrow b_i = S_\gamma(a_i) \tag{81}$$

Even though nonlinearity can be achieved using different S-boxes for each position byte, this approach is not required since no improvement of resistance against linear and differential cryptanalyst was demonstrated [57]. Moreover, it only increases the resource cost. Consequently, the same S-box is used for each byte position.

The S-box must satisfy nonlinearity to reduce as much as possible the input-output correlation and difference propagation probability in order to provide resistance against linear and differential cryptanalysis. The substitution box must also provide high degree of complexity in its algebraic expression to prevent interpolation attacks [43].

From a set of already defined S-boxes that satisfy nonlinearity the S-box consisting of transformation Inv_8 was selected. This transformation is described as the mapping of an element with its inverse in $GF(2^8)$ extended with 0 mapped to 0. However, it does not possess algebraic complexity. Therefore, the Rijndael S-box S_{RD} was built as a sequence of transformation Inv_8 and an invertible affine transformation Aff_8. Even though the affine transformation Aff_8 has no complexity in its algebraic expression, its combination with Inv_8 provides the required complexity [43].

$$\mathrm{S_{RD}} = \mathrm{Aff_8} \circ \mathrm{Inv_8} \tag{82}$$

To define the affine transformation $\mathrm{Aff_8}$ first two restrictions were imposed. The first restriction declares no fixed points, while the second one no opposite fixed points.

$$\mathrm{S_{RD}}[a] + a \neq 0, \quad \forall a \tag{83}$$

$$\mathrm{S_{RD}}[a] + a \neq FF, \quad \forall a \tag{84}$$

The affine transformation is defined as the addition in $GF(2^8)$ of a linearized polynomial $L(a)$ over $GF(2^8)$ and a constant q.

$$\mathrm{Aff_8}(a) = L(a) + q \tag{85}$$

A matrix representation of the affine transformation and its inverse transformation is presented in (86) and (87) respectively.

$$b = \mathrm{Aff_8}(a) \Leftrightarrow$$

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \\ 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{86}$$

$$x = \mathrm{Aff_8^{-1}}(y) \Leftrightarrow$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \\ 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{87}$$

Since the inverse transformation of transformation Inv_8 is itself, the Rijndael S-box S_{RD} and its inverse transformation $\mathrm{S}_{\mathrm{RD}}^{-1}$ can be represented as shown in (*88*) and (*89*) respectively.

$$\mathrm{S}_{\mathrm{RD}}[a] = \mathrm{Aff}_8\left(\mathrm{Inv}_8(a)\right) \tag{88}$$

$$\mathrm{S}_{\mathrm{RD}}^{-1}[a] = \mathrm{Inv}_8^{-1}\left(\mathrm{Aff}_8^{-1}(a)\right) = \mathrm{Inv}_8\left(\mathrm{Aff}_8^{-1}(a)\right) \tag{89}$$

In Rijndael, the step that applies the local nonlinear transformation γ using the substitution box S_{RD} over the state is called the $\mathrm{SubBytes}$ step is used for encryption. In contrast, the step that uses the inverse substitution box $\mathrm{S}_{\mathrm{RD}}^{-1}$ in the local nonlinear transformation γ for decryption is called the $\mathrm{InvSubBytes}$ step.

The application of the S-box to each byte position allows parallelism, as no order must be followed to substitute the bytes within the state. To facilitate analysis alignment during application of the transformation must be satisfied. In other words, bytes must contain only bits from same column and row within the state if the state is represented as a block. As a result of alignment, each bit is handled similarly compared to all other bits, thus adding symmetry to the transformation [58]. This imposes to the cipher the requirement that the block length must be multiple of the S-box, which is 8 bits [59].

The wide trail strategy focusses on the analysis of the *weight of trails*. The weight of trails is defined by the correlation and difference propagation over the round. The design approach goal is to improve the weight of trails increasing the minimum weight of trails. This can be achieved either using a large S-box, which requires higher implementation costs or designing the round avoiding the existence of trails with low weight [60].

To avoid using a larger S-box, diffusion can be added to the round increasing the minimum weight. However, bricklayer transformations add no diffusion. Therefore, neither the nonlinear transformation γ nor the key addition

provide diffusion to the round as they operate on individual bytes. For this reason, the linear mixing transformation λ is required [61].

6.3.2 Linear Mixing Transformation λ

Adding diffusion to the round is only possible through the linear mixing transformation λ increasing the *branch number*, which is the diffusion measure. Since the branch number is the same for a transformation and its inverse, it does not change for encryption and decryption adding symmetry to the design. Nevertheless, transformations with high branch number imply high implementation costs as well [62].

Rather than defining an expensive transformation in terms of resources, an efficient solution was designed taking advantage of propagation properties. This approach consists of building transformation λ as a sequence of two transformations: on the one hand, a transformation θ that provides high local diffusion through a linear bricklayer permutation on array bytes, and on the other, a transformation π that provides high dispersion by moving bytes that were closed to each other at the input to distant positions at the output [62].

6.3.3 Local Diffusion Transformation θ

This transformation is defined as a linear bricklayer permutation operating on columns, which are arrays of bytes. Furthermore, since it is a linear transformation, it can be defined using an n_ξ by n_ξ matrix M_ξ, where n_ξ is the number of elements in the columns, which is the same as the number of rows in the state if this is represented as a block [63].

$$b = \theta(a) \Leftrightarrow b_\xi = M_\xi a_\xi \qquad (90)$$

The columns are considered as polynomials over $GF(2^8)$. Hence, a fixed polynomial $c(x)$ used to multiply each column must be defined. The polynomial must have an inverse element in $GF(2^8)$, and its coefficients must be the simplest

possible to achieve high performance [64]. As explained in Section 2.5, the fixed polynomial $c(x)$ and its inverse is $d(x)$ are defined as

$$c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}, \quad (91)$$

$$d(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}. \quad (92)$$

A matrix representation of the multiplication of a column with the fixed polynomial $c(x)$ and its inverse $d(x)$ over $GF(2^8)$ is shown in (93) and (94).

$$b(x) = c(x) \times a(x) \Leftrightarrow$$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (93)$$

$$a(x) = d(x) \times b(x) \Leftrightarrow$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0\mathrm{E}\ 0\mathrm{B}\ 0\mathrm{D}\ 09 \\ 09\ 0\mathrm{E}\ 0\mathrm{B}\ 0\mathrm{D} \\ 0\mathrm{D}\ 09\ 0\mathrm{E}\ 0\mathrm{B} \\ 0\mathrm{B}\ 0\mathrm{D}\ 09\ 0\mathrm{E} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \quad (94)$$

Therefore, since the transformation is linear, it can also be described as a D-box operating on individual columns. The restriction imposed to the D-box matrix to be circular, as shown in (95), defines the number of bytes in each column to four elements, which is the same as the number of rows. In addition, imposing alignment between bytes and columns adds symmetry to the transformation and restricts the length of the block length to be multiple of the column size, which is 4 bytes or 32 bits. The D-Box also allows parallelism in its operation over each column [58].

$$a_{i,j} = a_{0,j-i\mathrm{mod}4} \quad (95)$$

In Rijndael, the step that operates on the state through this transformation is called the MixColumns step, which is used for encryption, whereas the step that uses the inverse transformation is called the InvMixColumns step, which is used for decryption.

6.3.4 Dispersion Transformation π

More diffusion can be achieved without necessarily increasing the branch number and thus implementation costs as well. Rather than incrementing the branch number, the *column branch number*, which is the branch number with respect to columns, can be increased providing inter-column diffusion, if the transformation does not operate in independent columns [65]. Furthermore, combining the branch number with the column branch number affects the weight of trails [66].

Consequently, the dispersion transformation π is defined as a transposition of bytes in order to move the elements in each column to different columns [67].

$$b = \pi(a) \Leftrightarrow b_i = a_{p(i)} \tag{96}$$

Since in Section 6.3.2 the number of rows was already specified as four by the D-box to satisfy the circular matrix restriction, the minimum number of columns must be equal to the number of rows to ensure all bytes within the column are distributed over different columns. If this occurs, it can be said that the transformation π is *diffusion optimal* [67].

Thus, to make the transformation diffusion optimal different offsets were specified for the transposition of bytes. The criterion of simplicity was applied to select the different offsets for each row. For this reason, one offset is equal to zero. Although the offsets could be assigned arbitrarily to the rows, their simplest values were selected considering resistance against saturation and truncated differential attacks [44]. The offsets are defined in Table 7.

N_b	C_0	C_1	C_2	C_3
4	0	1	2	3
5	0	1	2	3
6	0	1	2	3
7	0	1	2	4
8	0	1	3	4

Table 7 – Offsets for different block lengths

In Rijndael, this transformation is used as the $\mathrm{ShiftRows}$ step for encryption, and its inverse transformation is used as the $\mathrm{InvShiftRows}$ step for decryption.

6.4 Round Transformation

The round transformation ρ was defined by the local nonlinear transformation γ and the linear mixing transformation λ, which consists of the dispersion transformation π and the local diffusion transformation π [68].

$$\rho = \lambda \circ \gamma = \theta \circ \pi \circ \gamma \quad (97)$$

A graphical illustration of the round transformation followed by the key addition is given in Figure 48.

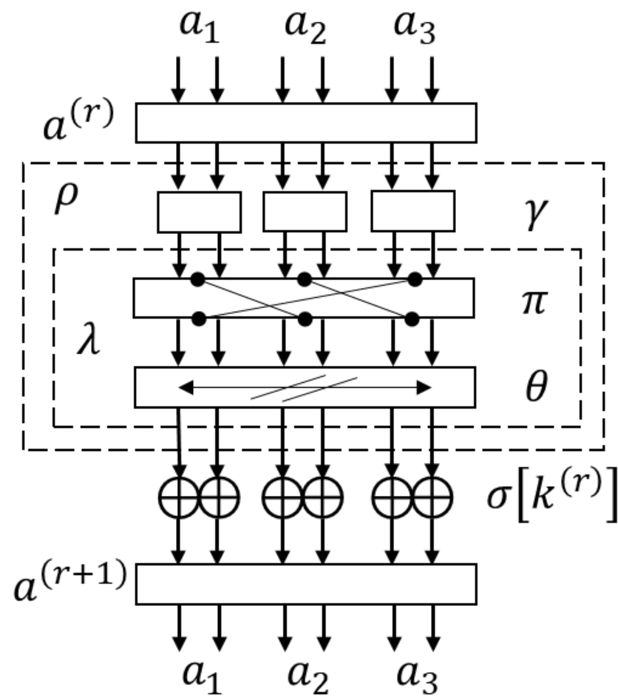

Figure 48 – Round transformation of Rijndael

Applying alignment and symmetry in the transformations that form the circular process allows for adaptable sequencing of steps. Since the nonlinear step γ operates on individual bytes, it can commute with the transposition step π. Furthermore, the transposition and the mixing step can also commute with the key addition if the same transformation is applied to the key [59]. This property is valid for any linear transformation A with state x and key k [37].

$$A(x + k) = A(x) + a(k) \tag{98}$$

The XOR operation for the key addition, modular multiplication with constants for the $\mathrm{MixColumns}$ step and addition operation in $GF(2^8)$ were selected to be used by the transformations. Arithmetic operations were excluded, as they might not take full advantage of some processors and because implementation of multiplication involves larger number of gates. Additionally, carry propagation complicates the implementation of countermeasures against differential power analysis [69].

6.5 Number of rounds

If the number of cipher rounds increases, the resistance against cryptanalytic attacks also increases. To determine the number of rounds, first the number of rounds, for which shortcut attacks show more efficiency than an exhaustive search of the key, had to be identified. Once the minimum number of rounds to prevent shortcut attacks was known, four more rounds were added to provide a security margin. It can be said that *full diffusion* was added at the beginning and at the end of the round, since two rounds are sufficient to provide full diffusion. In other words, after two rounds, one state bit is likely to affect half of the state bits. The security margin provides resistance against linear and differential cryptanalysis, as well as truncated differential attacks, saturation attacks and the impossible-differential attack [70].

To minimize efficiency in shortcut attacks, for every additional column in the key, consisting of 32 bits, the number of rounds increases by one. Considering the key length in the number of rounds provides protection against partially known keys attacks and related-key attacks. Moreover, the number of rounds increases by one for every additional column in the block as well, because diffusion is added after three rounds for block lengths above 128 bits. In this way, patterns at the output with respect to the input are reduced [70].

Table 8 shows the number of rounds the cipher must have depending on the number of columns in the block and the number of columns in the key.

	N_b				
N_k	4	5	6	7	8
4	10	11	12	13	14
5	11	11	12	13	14
6	12	12	12	13	14
7	13	13	13	13	14
8	14	14	14	14	14

Table 8 - Number of rounds N_r depending on N_b $(block\ length/32)$ and N_k $(key\ length/32)$

6.6 Key Schedule

The key schedule is formulated to introduce asymmetry into the cipher via round constants, nonlinearity through the use of the S-box, and efficient diffusion during key expansion. This approach protects the cipher against slide attacks and provides resistance against related-key attacks [71].

The resources utilized and the simplicity of its operations during the key expansion were designed to minimize costs in implementation and thus provide high key agility, which is related to time, power, and memory costs [72].

The expansion of the key enables independence from the block length as its design relies on the key length, which is mapped to the expanded key during the initial stage of key expansion. In contrast, the round key selection step depends on the columns of the state, since it selects the same number of bits from the expanded key. However, it does not depend to the key length [73].

Thanks to the key expansion's design approach, round keys can be generated on-the-fly. This means they are generated individually as needed instead of all at once before using the first key. This approach can be used for both encryption and decryption. As a consequence, only memory with size of the key is needed to expand the key [74].

7 Conventional AES-256 Implementation

In this chapter the characteristics of a conventional implementation of the Advance Encryption Standard – 256 are explained, such as the application of the steps, instances used, and scheduling for the encryption and decryption. This approach also applies for the implementation of the other variants, these are AES-128 and AES-192.

7.1 Application of steps

The conventional sequence of steps for encryption as well as for decryption begins with the expansion of the cipher key. First, the cipher key is mapped to the array of the expanded key and the following columns are obtained with operations on previous columns. The conventional approach proposes expanding the full key before the first RoundKey is added in the AddRoundKey step, which is the first step in both the encryption and decryption process.

Once the ExpandedKey is complete, stored, and ready to use the AddRoundKey step operates over the state. In encryption, the AddRoundKey step is not considered in the Round transformation in comparison with decryption, where it is a step within the InvRound transformation as explained in Section 5.1.

Due to the cipher design approach, the property of symmetry allows parallelism in the application of each step over the elements within the state as explained in Section 6.3. In other words, all the element in the state can be processed at the same time if the required resources exist.

7.2 Resource Utilization

To apply the transformations in each step over all the elements of the state at once as the conventional implementation proposes, first the analysis of resources must be determined.

The steps of interest are the steps using an S-box, D-box and their inverse, The steps using these transformations are the SubBytes, InvSubBytes, MixColumns and InvMixColumns step, as well as for the KeyExpansion. Since all the steps excluding the KeyExpansion are defined as bricklayer permutations, the transformations operate independently over elements of the state. The S-Box operates independently over bytes, whereas the D-Box over its columns.

If a bricklayer permutation requires the complete state to be computed at once, the number of transformations necessary to operate over the state is equal to the number of elements within the state. For the SubBytes step, the number of S-box instances required to compute all the bytes at the same time is equal to the number of bytes within the state, which is 16 bytes as shown in Figure 49. In contrast, the number of D-box instances required to compute all the columns at the same time for the MixColumns step is equal to the number of columns within the state, which is 4 as presenented in Figure 50.

If different S-box instances are used for the KeyExpansion, the number of S-box instances increases by 8 for AES-256. Consequently, the total number of S-box instances for AES-256 is equal to 24 instances.

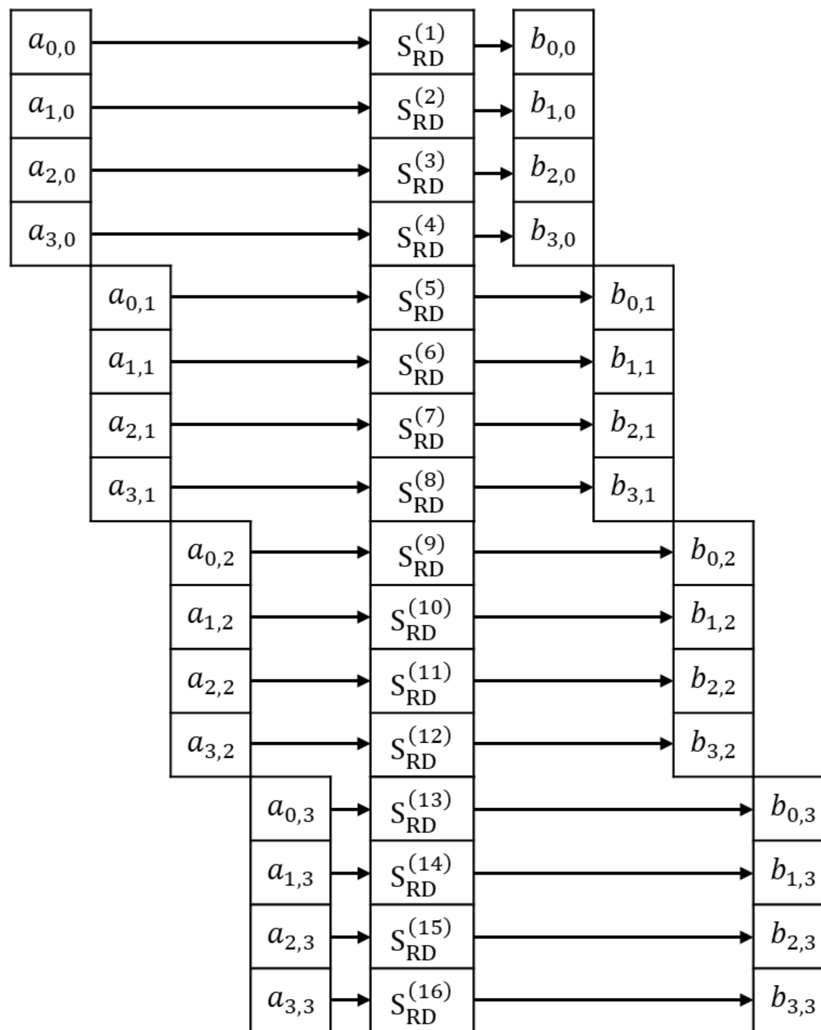

Figure 49 – S-box instances for the SubBytes *step*

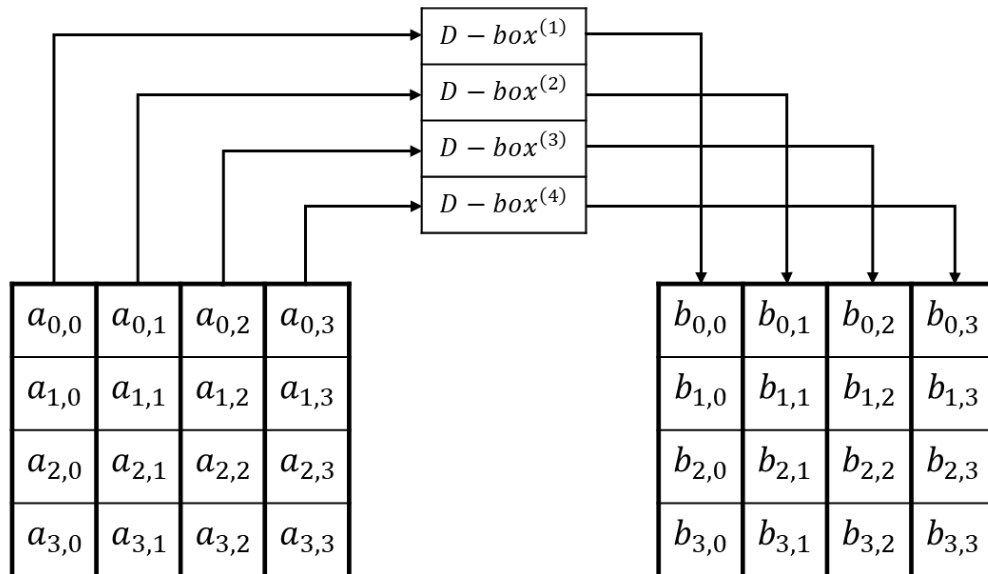

Figure 50 - D-box instances for the MixColumns *step*

7.3 Scheduling

The KeyExpansion for AES-256 is computed iteratively for every 8 columns after the cipher key has been mapped to the first 8 columns of the ExpandedKey. The same operations are employed using 8 S-box instances for every iteration. Since 7 iterations are required, the KeyExpansion takes 8 cycles to be computed considering the mapping of the cipher key. After the KeyExpansion, the encryption or decryption steps can be performed.

The total number of cycles can be calculated taking into account the following considerations:

1. The Round and InvRound transformations consist of four steps and are iterated 13 times.

2. The FinalRound and InvFinalRound are conformed by three steps and are computed once.

3. An additional AddRoundKey step operates on the state at the beginning and at the end in encryption and decryption respectively.

4. Each step takes one cycle.

Excluding KeyExpansion the cipher takes 56 cycles for encryption or decryption. Accordingly, the total number of cycles is obtained adding the number of cycles taken by the KeyExpansion and by the steps for encryption or decryption. In conclusion, the total number of cycles is equal to 64 cycles.

Encryption and decryption scheduling is shown in

Process	Step	Cycle
KeyExpansion	Mapping of Cipher Key to ExpandedKey	1
	Iteration 1	2
	...	...
	Iteration 7	8
	AddRoundKey	9
Round 1	SubBytes	10
	ShiftRows	11
	MixColumns	12
	AddRoundKey	13
Round i	...	...
Round 13	SubBytes	58
	ShiftRows	59
	MixColumns	60
	AddRoundKey	61
Round14 (FinalRound)	SubBytes	62
	ShiftRows	63
	AddRoundKey	64

Table 9 and

Process	Step	Cycle
KeyExpansion	Mapping of Cipher Key to ExpandedKey	1
	Iteration 1	2
	...	...
	Iteration 7	8
Round 1 (InvFinalRound)	AddRoundKey	9
	InvSubBytes	10
	InvShiftRows	11
Round 2	AddRoundKey	12
	InvMixColumns	13
	InvSubBytes	14
	InvShiftRows	15
Round i	...	...
Round 14	AddRoundKey	60
	InvMixColumns	61
	InvSubBytes	62
	InvShiftRows	63
	AddRoundKey	64

Table 10 respectively.

Process	Step	Cycle
KeyExpansion	Mapping of Cipher Key to ExpandedKey	1
	Iteration 1	2
	…	…
	Iteration 7	8
	AddRoundKey	9
Round 1	SubBytes	10
	ShiftRows	11
	MixColumns	12
	AddRoundKey	13
Round i	…	…
Round 13	SubBytes	58
	ShiftRows	59
	MixColumns	60
	AddRoundKey	61
Round14 (FinalRound)	SubBytes	62
	ShiftRows	63
	AddRoundKey	64

Table 9 - Scheduling of conventional implementation of AES-256 for encryption

Process	Step	Cycle
KeyExpansion	Mapping of Cipher Key to ExpandedKey	1
	Iteration 1	2
	…	…
	Iteration 7	8
Round 1 (InvFinalRound)	AddRoundKey	9
	InvSubBytes	10
	InvShiftRows	11
Round 2	AddRoundKey	12
	InvMixColumns	13
	InvSubBytes	14
	InvShiftRows	15
Round i	…	…
Round 14	AddRoundKey	60
	InvMixColumns	61
	InvSubBytes	62
	InvShiftRows	63
	AddRoundKey	64

Table 10 - Scheduling of conventional implementation of AES-256 for decryption

8 State-of-the-art

Analysis and approaches fin the implementation of the Advanced Encryption Standard (AES) has been an interesting topic of study not only in the cryptography field, but also in many industries that require security embedded in hardware.

8.1 AES Architecture for Secure ECG Signal Transmission

AES implementations on FPGAs has been used in the medical field, such as work [75], that implements AES-128 for encryption and decryption on a Xilinx ZedBoard Zynq™−7000 FPGA to secure Electrocardiogram signals. ECG signals are considered a biometric identification since two persons cannot emit the same ECG signals. Consequently, malicious intentional manipulation of this signals might lead to damage to the user´s integrity. Additionally, by law it is required to transfer the ECG signals encrypted [76].

Rather than implementing all the rounds in a fully pipelined fashion, a folded architecture was implemented, where only one set of operations was reused reducing area. A graphical illustration is presented in Figure 51.

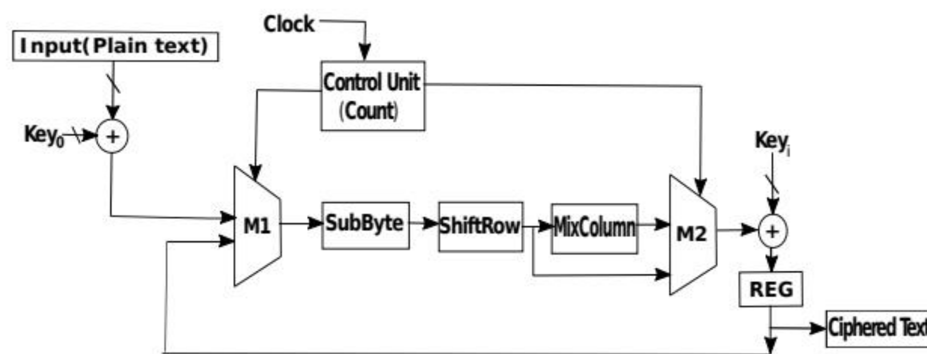

Figure 51 - Modified Architecture of Round Module of AES [75]

8.2 Implementation and Comparative Analysis of AES

The work described in [77] implements AES-128 on a Xilinx FPGA Artix-7. The implementation approach in this work is the conventional one with no further modifications. Additionally, it is presented a comparative analysis of its implementation with other works.

8.3 AES implementation on Xilinx FPGAs suitable for FPGA based WBSNs

Confidentiality in individual´s physiological data was also addressed in work [78] with the implementation of AES-128 on the Xilinx Artix-7 Virtex-7, Virtex-6, Virtex-4 and Spartan-6 FPGAs for wireless body sensor networks (WBSN). The approach taken in this work was the utilization of the dedicated Block RAM resources provided by the FPGAs for the S-box look up table. This approach results in a higher efficiency on the Artix-7 FPGA in terms of speed, area, and power compared to the other devices. The architecture followed for this implementation is given in Figure 52.

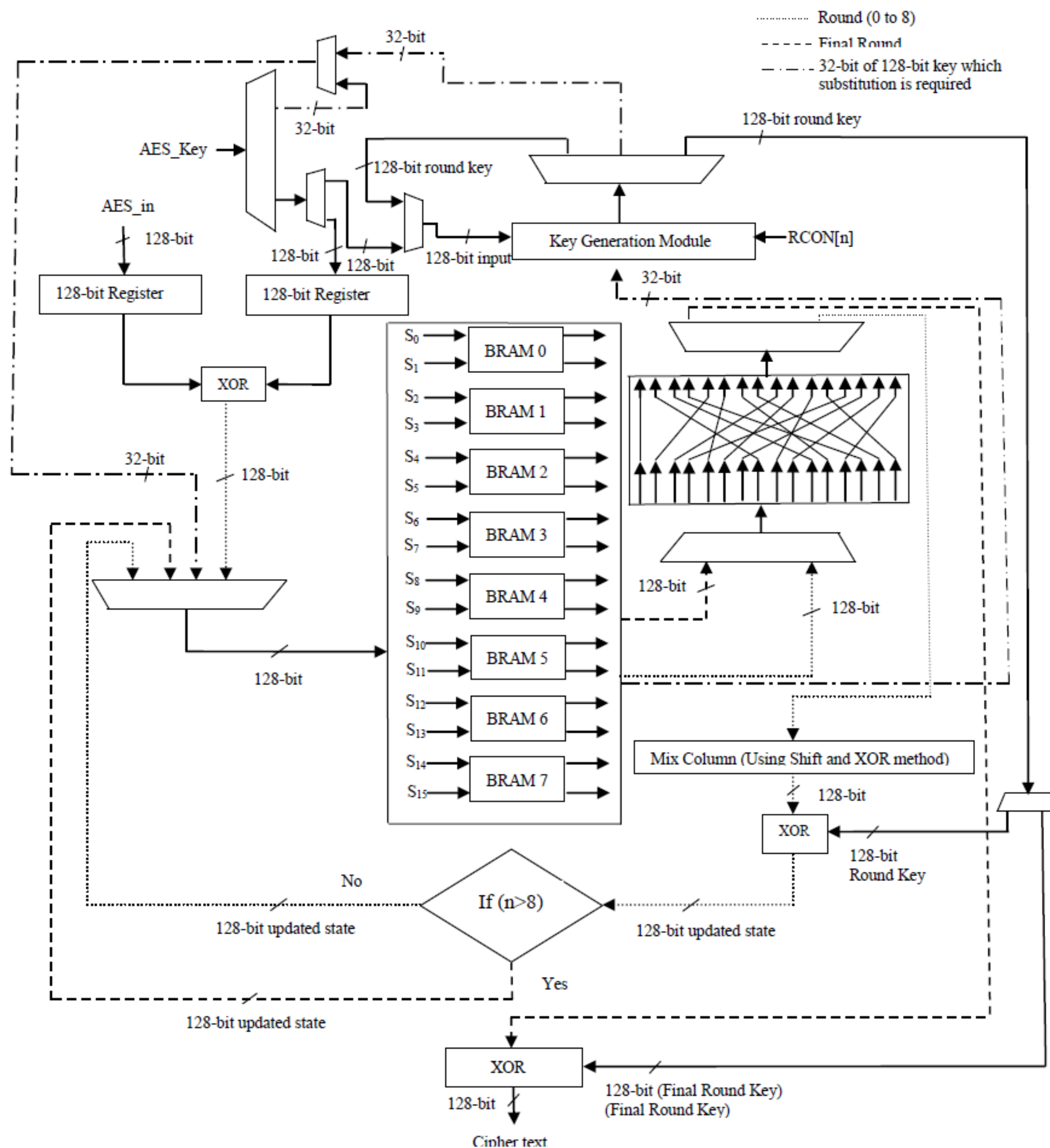

Figure 52 - Architecture for AES implementation using Block RAMS [78]

8.4 Efficient AES Implementation Using Xilinx System Generator

Reducing time costs during implementation of AES was addressed in [79] using High Level Language (HLL) through Xilinx System Generator for MATLAB, which provides flexibility in design. However, with this approach the low-level

design is ignored. For implementation of the S-box for the SubBytes step, BRAM resources were used. The design was implemented on a Xilinx Virtex-6 FPGA. The architecture followed by this work is a full-pipelined architecture as shown in Figure 53. Hardware resources were not reused during encryption, so 10 rounds were implemented.

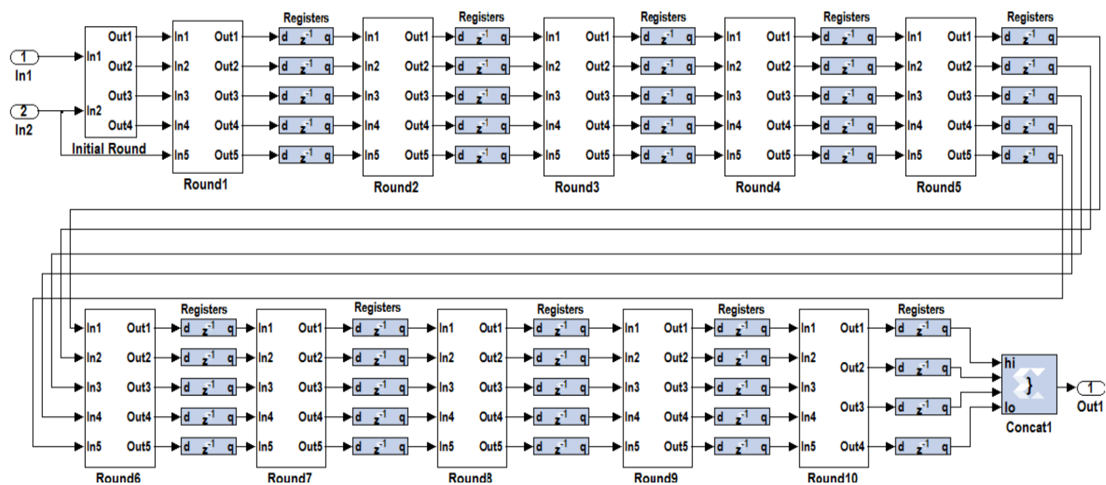

Figure 53 - Pipelined architecture [79]

8.5 High Performance Data Encryption with AES

Big Data encryption also represents a topic of interest in implementations of AES as demonstrated in [80]. In this work, speed is the main concern since large amount of data needs to be encrypted. To achieve high throughput, a deep pipeline and full expansion technology was implemented on the Xilinx xc7k325tffg676-2l chip. In the same way as in previous work [79], this work uses independent hardware for each round as shown in Figure 54. Additionally, a complete unroll design was proposed, implementing 16 S-boxes for each round to prevent access congestion, 16 registers for the intermediate states, and 16 registers for each round key used as shown in Figure 55. This approach

increases the resources used, but most importantly increases the number of bits processed in every cycle.

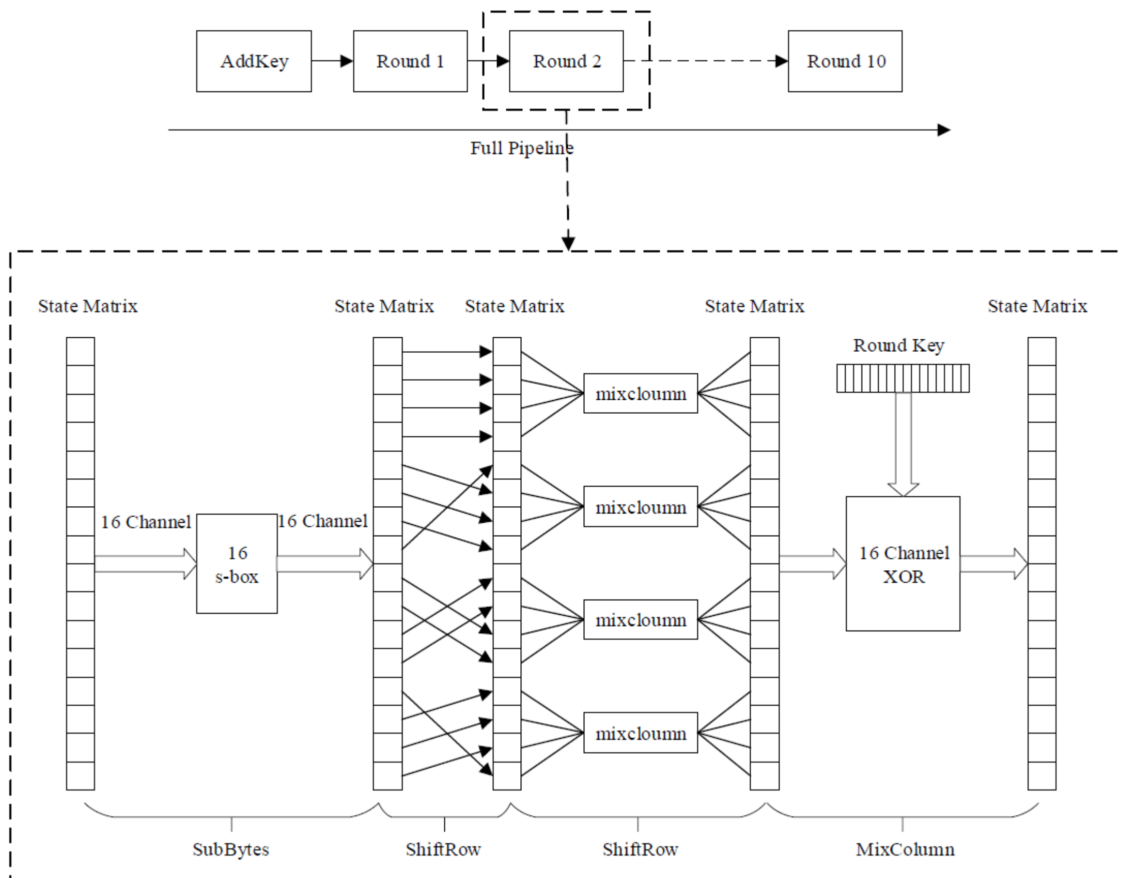

Figure 54 – Deep-pipelined architecture of AES [80]

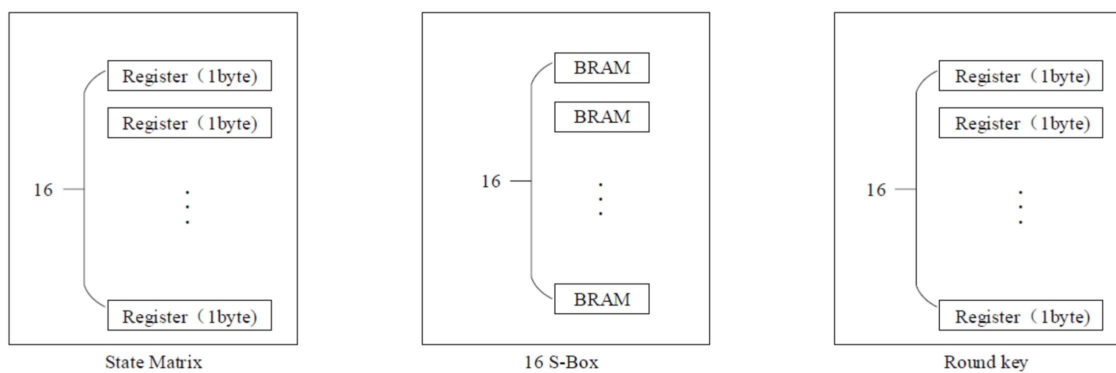

Figure 55 – Complete unroll design [80]

8.6 High Throughput and Fully Pipelined Implementation of AES-192

A similar approach was implemented in [81] in which loop-unrolling, fully pipelining, and sub-pipelining techniques were implemented to achieve high throughput. The loop-unrolling technique was employed to prevent a loop implementation that reuses the same hardware for all rounds. This approach results in a full-pipelined architecture. A graphical illustration of this approach is given in Figure 56. Moreover, in every round the steps are implemented in a sub-pipelined fashion as shown in Figure 57. The design was implemented on a Xilinx Virtex-7 Defense-Grade FPGA.

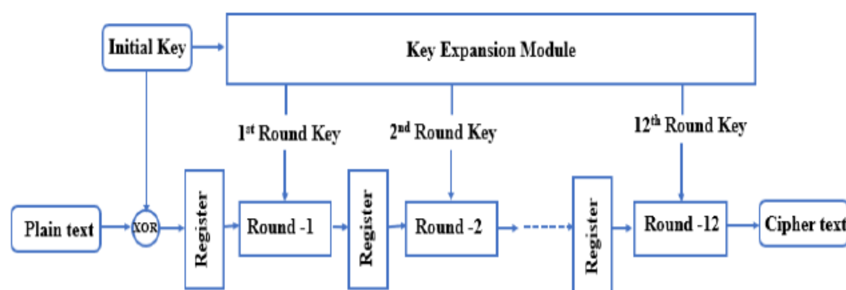

Figure 56 - Fully-pipelined architecture for AES-192 with loop-unrolled technique [81]

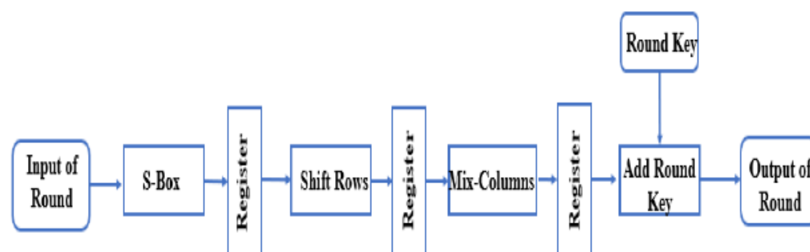

Figure 57 - Sub-pipelined architecture for round [81]

8.7 Implementation of AES Algorithm on FPGA and on software

For some applications implementation on both FPGAs and software might be possible. For this reason, in work [82] a comparison between the AES implementation on a Xilinx Artix-7 FPGA and on software show the propagation delay for both implementations. On FPGA 40.26 ns and 80.36 ns were required for Encryption and Decryption respectively. In contrast, 5ms for encryption and decryption were required when implementing the design on software.

8.8 Optimization and Implementation of AES Algorithm Based on FPGA

Higher throughput using the minimum amount of resources was achieved optimizing the conventional implementation in work [83]. Rather than processing the steps inside the round independently, four look up tables for the rounds called T tables and four for the final round called S tables were implemented. These tables combine all the steps and are presented in Figure 58 and Figure 59 respectively. Additionally, three tables for modular multiplications for the mixing step was employed. These tables are stored in dual port ROM structures.

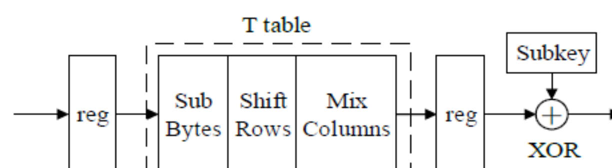

Figure 58 - Pipeline internal architecture of round [83]

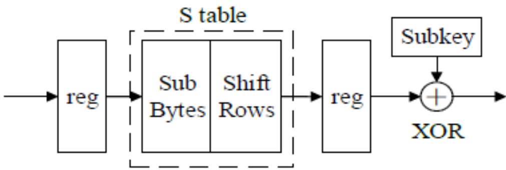

Figure 59 - Pipeline internal architecture of final round [83]

The design was implemented as a fully unrolled inner and outer double pipelined architecture. No loops reusing hardware were implemented neither inside nor outside the rounds. Furthermore, the key expansion is performed during encryption generating the round keys when required and it also uses a fully unrolled outer-round pipelined architecture. A graphical illustration is given in Figure 60. The design was implemented on a Xilinx Virtex-6 FPGA.

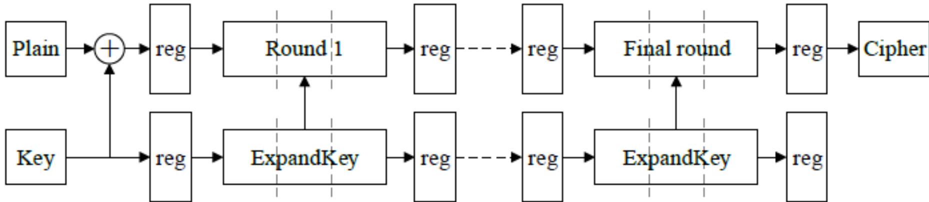

Figure 60 - Fully unrolled inner and outer double pipelined architecture [83]

Even though the works described in this chapter have implemented AES using different approaches in terms of architecture, most of them mainly focus on reaching high speeds. However, for applications where area is the primary concern, none of the works above mentioned achieved what the design approach implemented in [84] has achieved in terms of resources used. Its detailed description is presented in chapter 9.

9 Proposed AES-256 Implementation

In this chapter, a different approach for the implementation of the AES-256 variant is described as presented in work [84]. The comparison in terms of the sequence of steps, resource costs, and scheduling for encryption and decryption between this approach and the conventional implementation explained in Chapter 7 is addressed.

9.1 Application of steps

The conventional implementation computes the full KeyExpansion before the first AddRoundKey step. In contrast, the proposed implementation suggests the generation of each RoundKey as they are required. In this way, the first AddRoundKey step is computed adding the RoundKey that is a part of the cipher key. Therefore, no operations are necessary for KeyExpansion in either the first or second AddRoundKey as the last columns of the cipher key are used as the RoundKey.

The application of the steps over the state differs with the conventional approach as well. Rather than computing the steps over all the elements of the state at the same time, this approach computes the steps each column at a time.

Additionally, [84] proposes combining the MixColumns step with the AddRoundKey step calling it the Mix step and the combination of AddRoundKey with the InvMixColumns step calling it InvMix. The Mix and InvMix steps depend on the round iteration, as they do not consider the MixColumns step for round 0

and 14. Consequently, the structure of the cipher can be represented in the same way for both encryption and decryption considering that the substitution of bytes commutes with the byte transposition as shown in Figure 61 and Figure 62.

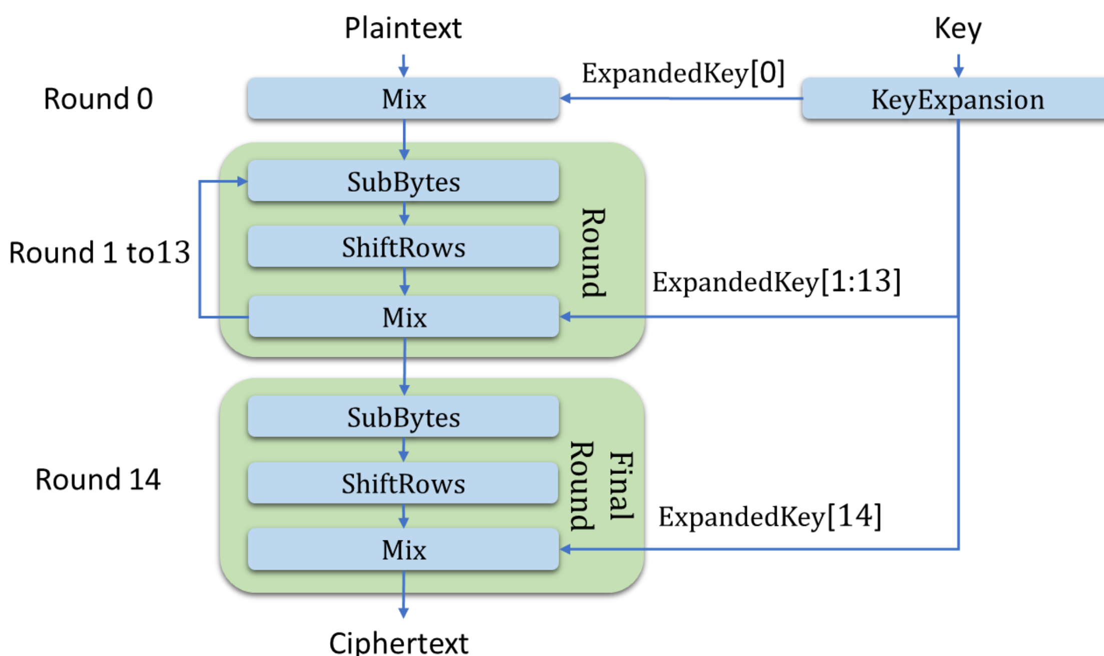

Figure 61 – Proposed architecture of AES-256 for encryption

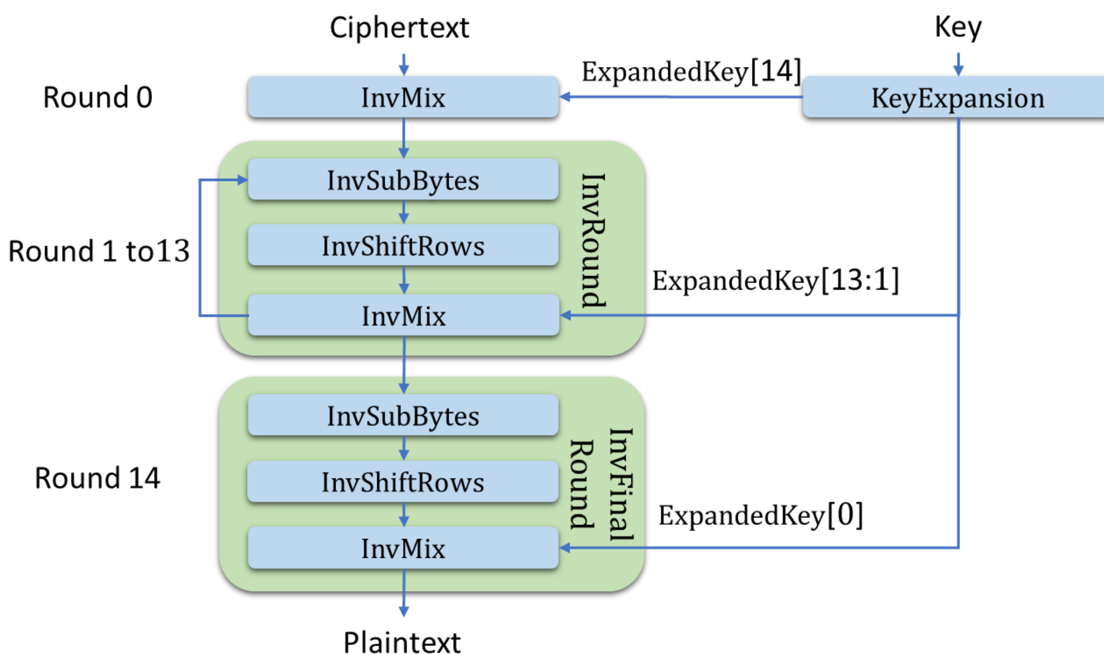

Figure 62 - Proposed architecture of AES-256 for deccryption

For comparison, a graphical illustration of the structure of AES-256 for decryption is provided in Figure 63.

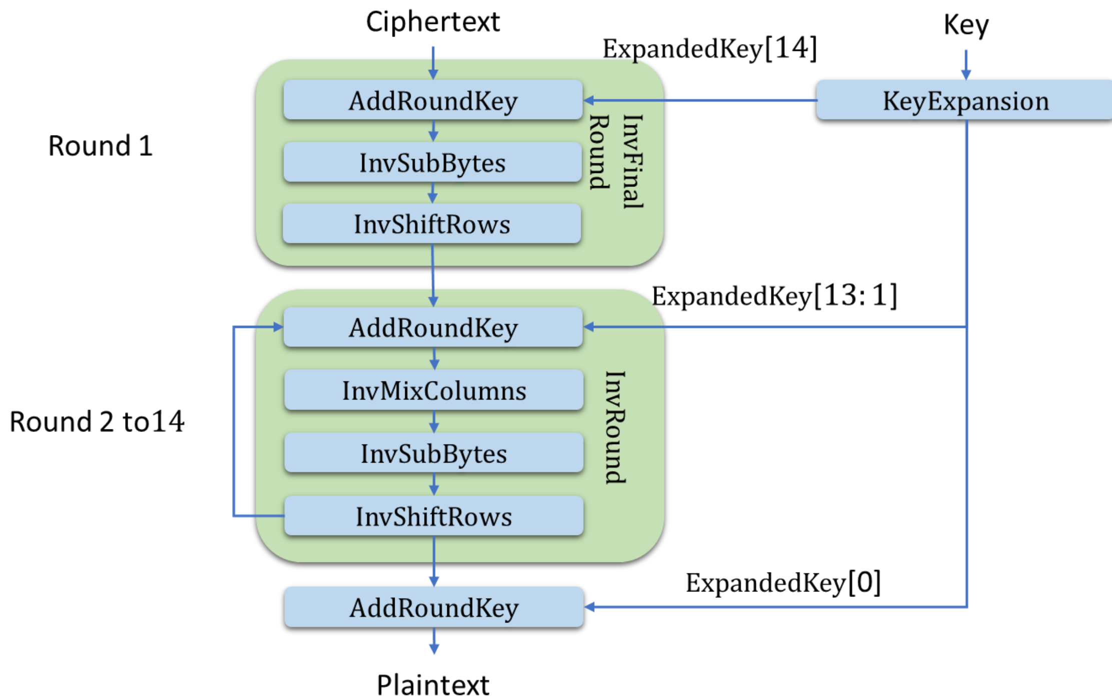

Figure 63 – AES-256 steps for decryption

9.2 Resource Utilization

This approach focuses on reusing S-box and D-box instances, as well as their inverse instances. A reduction of these instances is achieved changing the way the bricklayer permutations operate on the state. If the permutations act over individual columns in different cycles, the number of instances is divided by 4 for these steps.

Figure 64 presents the application of the SubBytes step with only 4 S-box instances for all the state if only a column is computed at a time, while Figure 65 demonstrates that only one D-box instance is needed for the MixColumn step, which in this approach is part of the Mix step.

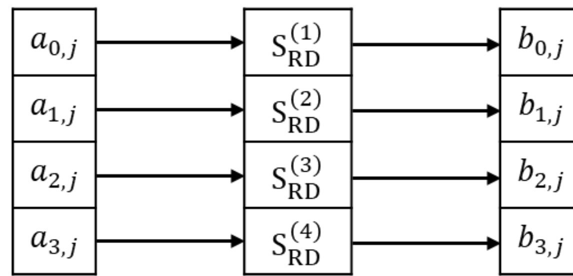

Figure 64 - S-box instances for the *SubBytes* step

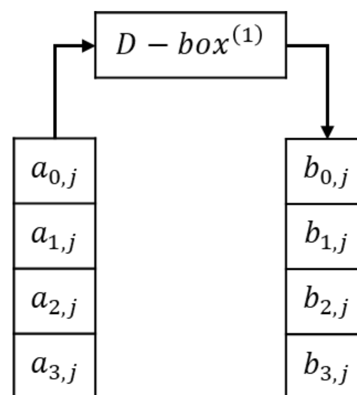

Figure 65 - D-box instances for the Mix *step*

Furthermore, if the KeyExpansion is implemented in such a way it reuses the same 4 S-box instances used for the SubBytes step, no more instances are required. Consequently, only 4 S-box instances and 1 D-box instance are used though all the encryption or decryption process.

9.3 Scheduling

For a clear demonstration Figure 66 shows the columns processed by the different steps emphasising each round with different colours. In cycle 1, the D-box instance is used by the Mix step to operate on the first column of the state. Since this operation occurs in round 0, the Mix step only adds the RoundKey

which is part of the cipher key. For that reason, no cycles to generate the RoundKey are required before the addition of the key.

In cycle 2, the S-box instance is used by the SubBytes step to operate over the first column of the state, which has already been computed by the Mix step in the previous cycle. Besides, the second column is computed by the Mix step to be used in the next cycle in the SubBytes step.

After the first column is processed by the SubBytes step in cycle 2, the bytes in the column are moved to different columns of the state through the ShiftRows step.

After the last column is processed by the SubBytes step in cycle 5, all the bytes in the first column are ready to be used by the Mix step in the next cycle if the byte transposition for this column does not require an extra cycle. Therefore, the ShiftRows step for the last column and the Mix step for the first column occurs in the same cycle.

Finally, for the last round the Mix step behaves like the AddRoundKey step, since the MixColumns step is not required. After the Mix step operates over the last round in cycle 74, the state becomes the encrypted block, also called ciphertext.

S-box	ShiftRows	D-box	Cycle
		Mix_0	1
Sub_0		Mix_1	2
Sub_1	Shift_0	Mix_2	3
Sub_2	Shift_1	Mix_3	4
Sub_3	Shift_2		5
	Shift_3	Mix_0	6
Sub_0		Mix_1	7
Sub_1	Shift_0	Mix_2	8
Sub_2	Shift_1	Mix_3	9
Sub_3	Shift_2		10
	Shift_3	Mix_0	11
...	...	...	...
	Shift_3	Mix_0	71
		Mix_1	72
		Mix_2	73
		Mix_3	74

Figure 66 – Scheduling of AES-256 for encryption considering instances used based on [84].

In contrast to the conventional implementation, in this approach the KeyExpansion is performed as the RoundKeys are required. Hence, the RoundKey generation only employs 4 S-box instances and can reuse the instances used by the SubByte step if these are used in cycles with instances available as shown in Figure 67.

S-box	ShiftRows	D-box	Cycle
		Mix_0	1
Sub_0		Mix_1	2
Sub_1	Shift_0	Mix_2	3
Sub_2	Shift_1	Mix_3	4
Sub_3	Shift_2		5
RoundKey_2	Shift_3	Mix_0	6
Sub_0		Mix_1	7
Sub_1	Shift_0	Mix_2	8
Sub_2	Shift_1	Mix_3	9
Sub_3	Shift_2		10
RoundKey_3	Shift_3	Mix_0	11
...	...	...	...
RoundKey_14	Shift_3	Mix_0	66
Sub_0		Mix_1	67
Sub_1	Shift_0	Mix_2	68
Sub_2	Shift_1	Mix_3	69
Sub_3	Shift_2		70
	Shift_3	Mix_0	71
		Mix_1	72
		Mix_2	73
		Mix_3	74

Figure 67 – Scheduling of AES-256 for encryption with key expansion in regard to instances based on [84].

Unfortunately, the same implementation for decryption is not possible because the RoundKeys are used backwards in the InvMix step. In other words, the first AddRoundKey step requires the last RoundKey in the ExpandedKey. Thus, the full KeyExpansion must be computed before using the RoundKey for round 0 as presented in Figure 68.

Inverse S-box	InvShiftRows	Inverse D-box	Cycle
			1
…	…	…	…
RoundKey_2			5
…	…	…	…
RoundKey_13			27
			28
RoundKey_14			29
			30
		InvMix_0	31
InvSub_0		InvMix_1	32
InvSub_1	InvShift_0	InvMix_2	33
InvSub_2	InvShift_1	InvMix_3	34
InvSub_3	InvShift_2		35
	InvShift_3	InvMix_0	36
InvSub_0		InvMix_1	37
InvSub_1	InvShift_0	InvMix_2	38
…	…	…	…
	InvShift_3	InvMix_0	101
		InvMix_1	102
		InvMix_2	103
		InvMix_3	104

Figure 68 - Scheduling of AES-256 for decryption with key expansion in regard to instances based on [84].

10 AES256 Encryption Implementation

In this chapter, the implementation of the AES-256 cipher for encryption is described, including the interaction between its internal modules. The importance and functionality of each of the modules is addressed to explain the relation that each module has with the AES algorithm.

10.1 AES256 Encryption Top Module

A top module was designed to synchronize the steps defined for encryption. The steps considered for the implementation are those provided by AES, except for the Mix step explained in detailed in Section 9.1, which combines the $\mathrm{MixColumns}$ step with the $\mathrm{AddRoundKey}$ step.

The input signals of the top module are the clock signal, an asynchronous low active reset, a start signal, the key, and the plaintext to encrypt. The output signals are the ciphertext and a ready signal. A graphical illustration of the top module inputs and outputs is given in Figure 69.

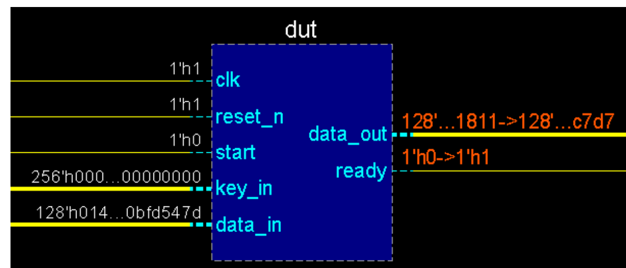

Figure 69 – AES256 Top module for encryption

The modules instantiated in the top module are:

- Counter Module
- Key Expansion Module
- MixColumns
- SubBytes Module
- ShiftRows Module

A block diagram showing the modules and their connections is presented in Figure 70.

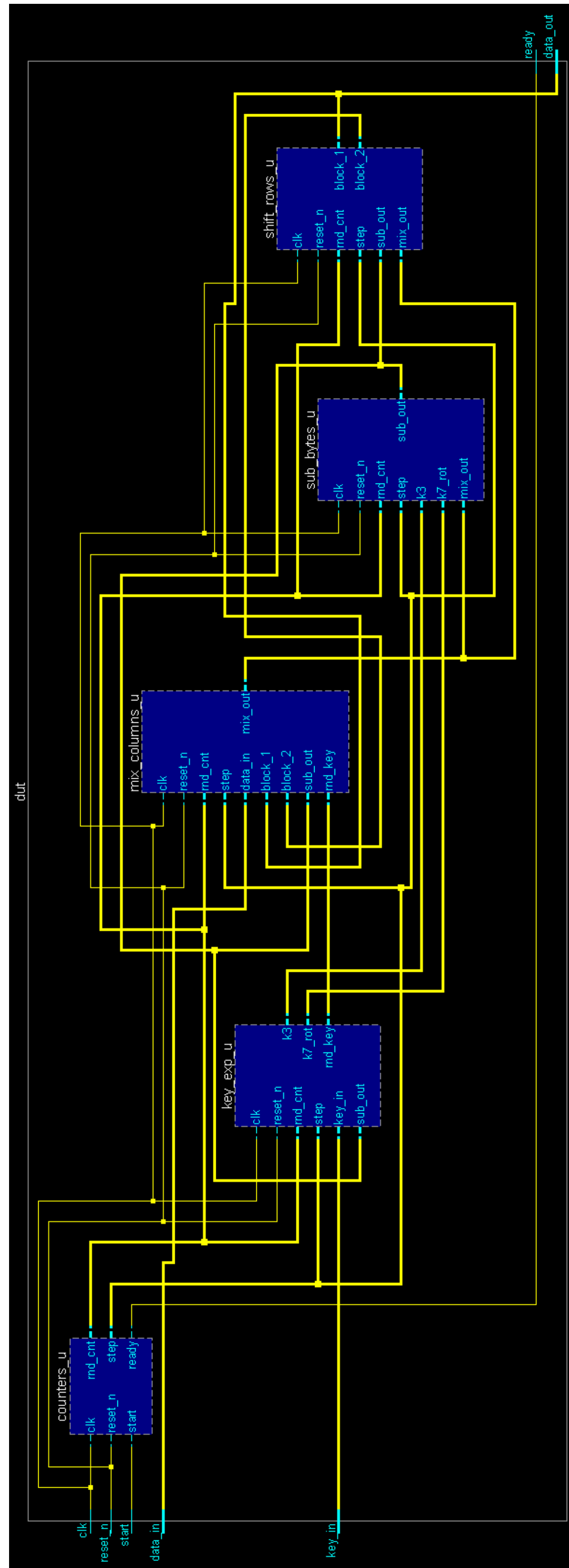

Figure 70 – AES-256 modules and their connections in the top module for encryption

10.2 Counters Module

In AES 15 rounds are required to encrypt a plaintext with a key and each round can be divided into 5 steps. The Counter module defines the number of the round and the step to synchronize all the other modules. To define the number of the round a counter was used, whereas a finite state machine was implemented to define the step. The first transition in the finite state machine is triggered by the start signal. The transition to the next step is triggered by a clock event since every step lasts only one clock cycle. When the encryption process is completed, a ready signal is set to high.

The input signals of the Counter module are the clock signal, an asynchronous low active reset, and the start signal. The output signals are the number of the round, the step, and the ready signal. A graphical illustration of the Counter module inputs and outputs is given in Figure 71.

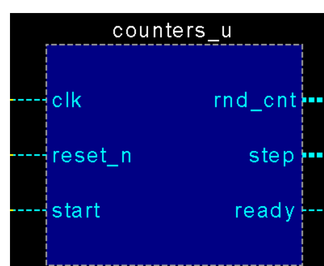

Figure 71 – Counter module

10.3 Key Expansion Module

For encryption, RoundKeys are required, these are obtained through the KeyExpansion step. In round 0 and step 0, 256-bit registers are used to store the cipher key defined as input. No expansion is required for RoundKey 0 and RoundKey 1, since these are already stored elements of the cipher key. In step 1, the next required RoundKeys are calculated using the results from the 4 S-box

instances. Finally, the $\mathrm{RoundKey}$ reuses the registers used to store the initial key in step 4 before the $\mathrm{RoundKey}$ is required.

The modules included in the Key Expansion Module are:

- Round Key Module
- Round Constant Module

A block diagram showing the modules and their connections is presented in Figure 72.

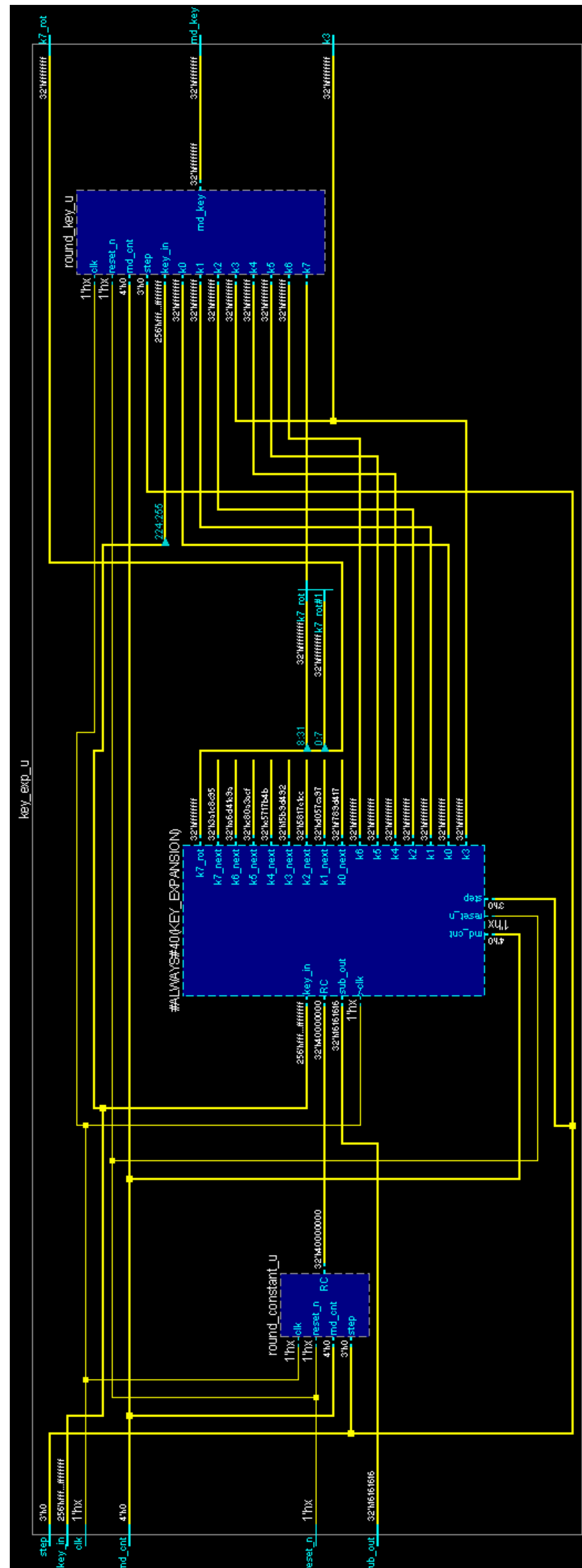

Figure 72 – Modules instantiated in Key Expansion Module

The input signals of the Key Expansion module are the clock signal, an asynchronous low active reset, the number of the round, the step, the key, and the results from the S-box instances. The output signals are the columns required to be substituted, and the RoundKey. A graphical illustration of the Key Expansion module inputs and outputs is given in Figure 73.

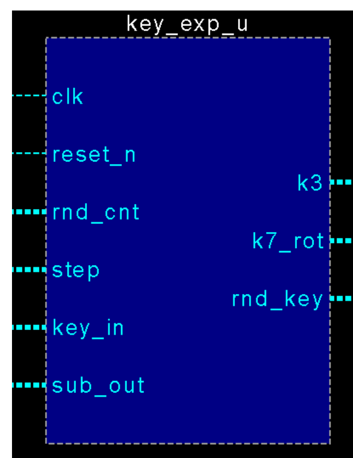

Figure 73 – Key Expansion module

10.3.1 Round Constant Module

To calculate some RoundKeys a RoundConstant is required. Depending on the round, a RoundConstant is calculated in step 0. The new RoundConstant is obtained shifting the previous one to the left beginning with a byte with value 1.

The input signals of the Round Constant module are the clock signal, an asynchronous low active reset, the number of the round and the step. The only output signal is the RoundConstant. A graphical illustration of the Round Constant module inputs and outputs is given in Figure 74.

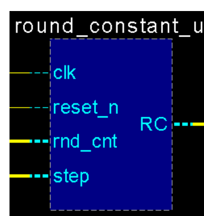

Figure 74 – Round Constant module

10.3.2 Round Key Module

This module selects the correct RoundKey from the registers used to store the RoundKeys. Not all the RoundKey is used in one step since the operations in this implementation approach operate on individual 32-bit columns rather than on the complete 128-bit state. For this reason, only one column of the RoundKey is used for addition with a column of the state per clock cycle.

The input signals of the Round Key module are the clock signal, an asynchronous low active reset, the number of the round, the step, the key, and the columns used to expand the key. The only output signal is the RoundKey. A graphical illustration of the Round Key module inputs and outputs is given in Figure 75.

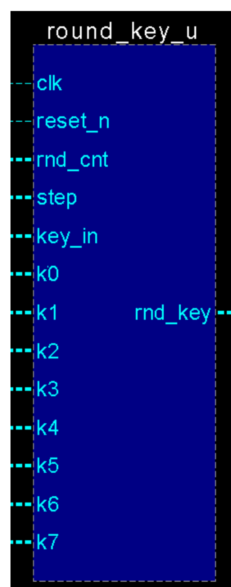

Figure 75 – Round Key module

10.4 MixColumns Module

In this module, the column of the state to be processed is selected, depending on the specific step and round. The four columns of the state are selected from step 0 to step 3 respectively. In round 0, the columns are taken from the input data of the top module. After round 0, the columns are taken from the registers where the state is stored. The columns are processed by the Mix Word module defined inside the MixColumns Module and the resulting processed columns are defined as the outputs of both modules.

The output of the MixColumns module is used as the input of the SubBytes module. The results from the last round are the columns of the final state, which is the output of the top module. These results are stored in the registers defined in the ShiftRows module, where the state is stored.

The input signals of the Mix Columns module are the clock signal, an asynchronous low active reset, the number of the round, the step, the plaintext, the output of the ShiftRows module, the output of the SubBytes module, and the

RoundKey. The only output signal is the processed column. A graphical illustration of the Mix Columns module inputs and outputs is given in Figure 76.

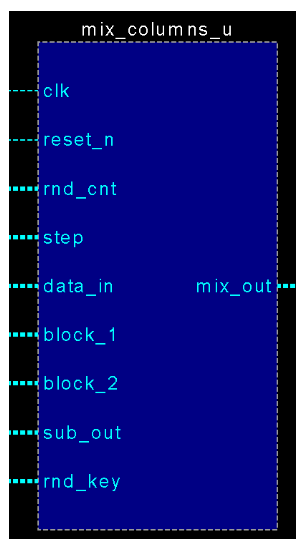

Figure 76 – Mix Columns module

10.4.1 Mix Word Module

This module operates the Mix step on columns of the state previously selected by the MixColumns module. The Mix step combines the AddRoundKey step with the MixColumns step, for this reason the RoundKey is required by the module. The Mix step only adds the RoundKey to the column if the round number is equal to 0 or 14, on the contrary, it also performs the MixColumns step over the column of the state before the addition of the RoundKey.

The addition of the key is done by means of a bitwise XOR operation. The MixColumns step for a column is done obtaining individually all the bytes in the resulting column in parallel using four instances of the ByteMix module. The output of each instance results in a byte of the processed column.

The input signals of the Mix Word module are the number of the round, the RoundKey and the column to be processed. The only output signal is the

processed column. A graphical illustration of the Mix Word module inputs and outputs is given in Figure 77.

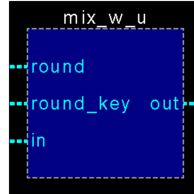

Figure 77 – Mix Word module

10.4.1.1 ByteMix Module

The multiplication of a column with a circular matrix can be performed using four instances of the ByteMix module, one for each byte in the output, since the same operations must be followed to obtain an element in the column as shown in (99).

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2(a_0) + 3(a_1) + a_2 + a_3 \\ a_0 + 2(a_1) + 3(a_2) + a_3 \\ a_0 + a_1 + 2(a_2) + 3(a_3) \\ 3(a_0) + a_1 + a_2 + 2(a_3) \end{bmatrix} = \begin{bmatrix} 2(a_0) + 3(a_1) + a_2 + a_3 \\ 2(a_1) + 3(a_2) + a_3 + a_0 \\ 2(a_2) + 3(a_3) + a_0 + a_1 \\ 2(a_3) + 3(a_0) + a_1 + a_2 \end{bmatrix} \tag{99}$$

The additions are achieved through bitwise XOR operations and the multiplications with more modules instantiated in this module, which is a module to multiply by 2 and a module to multiply by 3.

The input signals of the ByteMix module are the four bytes in the column. The only output signal is a byte, which composes the column. A graphical illustration of the ByteMix module inputs and outputs is given in Figure 78.

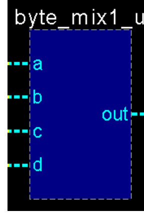

Figure 78 - ByteMix module

10.4.1.2 Multiply by 2 Module

Multiplication by 2 is the basis to multiply by any other number required for AES. This operation is a modular multiplication in $GF(2^8)$, which can be implemented with a bitwise XOR operation and conditional operators.

Multiplying a polynomial by 2 in $GF(2^8)$ is equal to multiplying it by x as explained in Section 2.4.1. If $a(x)$ is a polynomial in $GF(2^8)$ and $b(x)$ the multiplication of $a(x)$ with x.

$$\begin{aligned} b(x) = a(x) \times x \\ &= (a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \\ &\times x \\ &= a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 \\ &+ a_0x \end{aligned} \tag{100}$$

If $a_7 = 1$, the polynomial must be reduced with the reduction polynomial $m(x)$ defined for Rijndael to be an element in $GF(2^8)$. This reduction is achieved adding $m(x)$ to $b(x)$.

$$\begin{aligned} b(x) \equiv (a_7x^8 + a_6x^7 + a_5x^6 + a_4x^5 + a_3x^4 + a_2x^3 + a_1x^2 + a_0x) \\ &+ (x^8 + x^4 + x^3 + x + 1) \\ &\equiv (a_7 + 1)x^8 + a_6x^7 + a_5x^6 + a_4x^5 + (a_3 + 1)x^4 \\ &+ (a_2 + 1)x^3 + a_1x^2 + (a_0 + 1)x + 1 \\ &= b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \end{aligned} \tag{101}$$

In conclusion, $b_i = a_{i-1}$ for $5 \leq i \leq 7$ regardless of the value of a_7. If $a_7 = 1$, b_i for $1 \leq i \leq 4$ is obtained adding polynomial $a_3x^4 + a_2x^3 + a_1x^2 + a_0x$ to $x^4 + x^3 + x$. Finally, if $a_7 = 1$, then $b_0 = 1$ otherwise $b_0 = 0$.

The input signal of the Multiply by 2 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 2 module inputs and outputs is given in Figure 79.

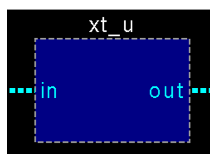

Figure 79 – Multiply by 2 module

10.4.1.3 Multiply by 3 Module

Multiplication of polynomial $a(x)$ by 3 in $GF(2^8)$ can be implemented as explained in Section 2.4.1 and shown in (102).

$$b(x) = a(x) \times (x + 1) = (a(x) \times x) + a(x) \tag{102}$$

This module uses the Multiply by 2 module to add to its output the input polynomial. Addition is done with a bitwise XOR.

The input signal of the Multiply by 3 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 3 module inputs and outputs is given in Figure 80.

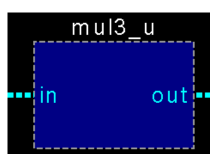

Figure 80 - Multiply by 3 module

10.5 SubBytes Module

This module selects the data required to be substituted by the S-box. The S-box module is instantiated inside this module. Depending on the step and round, data for the KeyExpansion or the output of the MixColumns module can be selected. The S-box module contains the S-box used for the SubBytes step and for the KeyExpansion. The S-box implementation is based on work [85].

The input signals of the SubBytes module are the clock signal, an asynchronous low active reset, the number of the round, the step, columns of the expanded key to be substituted and the output of the MixColumns module. The only output signal is the processed column. A graphical illustration of the SubBytes module inputs and outputs is given in Figure 81.

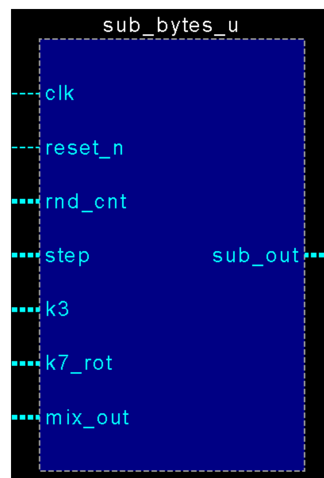

Figure 81 – SubBytes module

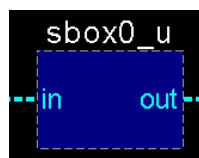

Figure 82 - S-box module

10.6 ShiftRows Module

This module computes the byte transposition of the SubBytes step. It also stores the output of the SubBytes module in registers in a specific order to perform the byte transposition over each column of the state in one cycle. The elements in the registers are used by the MixColumns module.

The input signals of the SubBytes module are the clock signal, an asynchronous low active reset, the number of the round, the step, the output of the SubBytes module and the output of the MixColumns module. The only output signals are the states after the transposition. A graphical illustration of the ShiftRows module inputs and outputs is given in Figure 83.

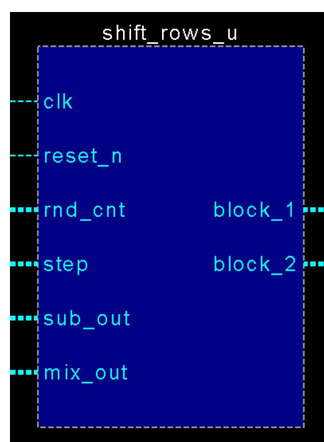

Figure 83 – ShiftRows module

11 AES256 Decryption Implementation

In this chapter the FPGA implementation of the AES256 cipher for decryption is addressed. The modules used for the implementation, their interaction with other modules and their function are described. The structure of the chapter starts with the top module and after that the modules in it. The description of submodules in every module is also included in this chapter. The modules used for decryption have a strong similarity with the modules used for encryption. Consequently, the description of the modules focuses on its differences with its counterpart used for encryption.

11.1 AES256 Decryption Top Module

A top module containing the main modules was implemented to add modularity to the design. This module instantiates the main modules for their interaction with each other and with the input and output signals of the top module. Apart from that, it does not provide additional functionalities.

The input signals of the top module are the clock signal, an asynchronous low active reset, the start signal, the cipher key and the ciphertext to decrypt. The output signals are the decrypted data and the ready signal. A graphical illustration of the top module inputs and outputs is given in Figure 84.

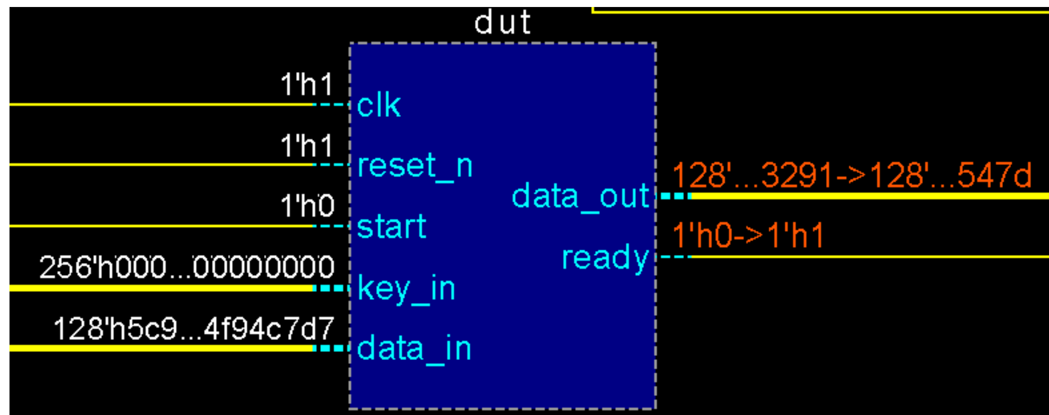

Figure 84 – AES256 Top module for decryption

The submodules instantiated in the top module are:

- Counters Dec module
- Key Expansion Dec module
- InvMixColumns module
- InvSubBytes module
- InvShiftRows module

A block diagram showing the modules and their connections is presented in Figure 85.

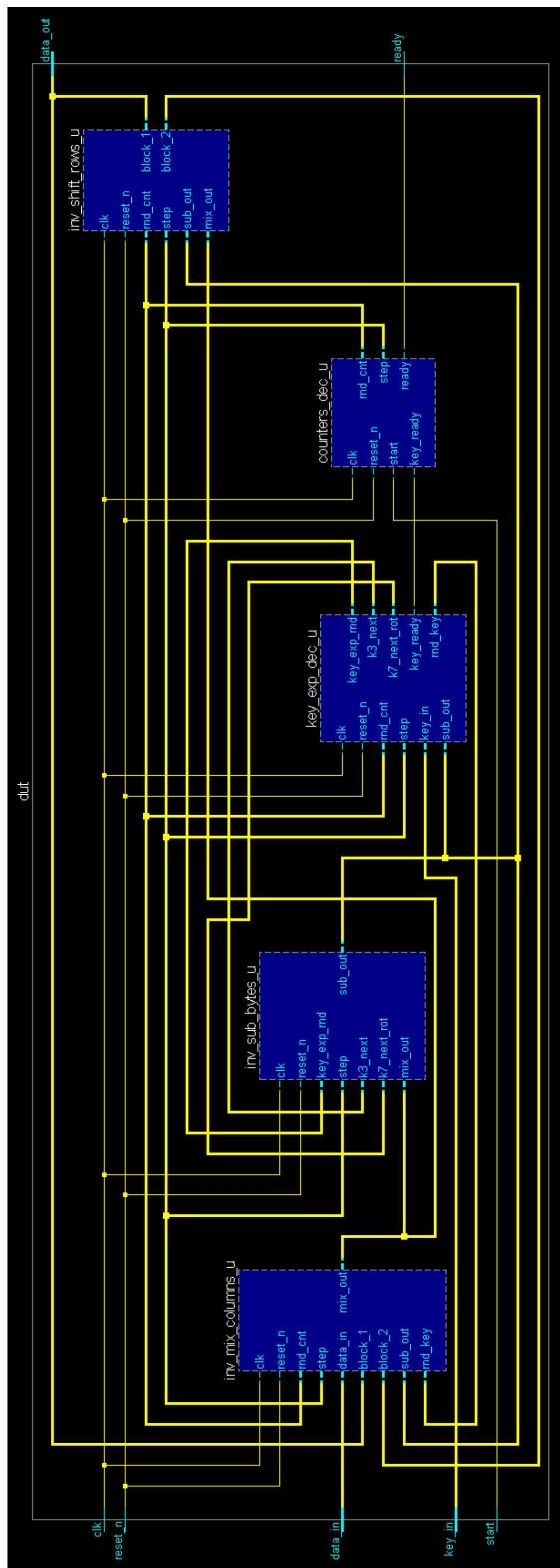

Figure 85 – AES-256 modules and their connections in the top module for decryption

11.2 Counters Dec Module

Compared to decryption, this module extends the finite state machine to expand the key completely before the beginning of decryption. After that, the 15 rounds with the 5 steps each are performed. When the counters responsible for the rounds and steps reach the value defined for the last cycle, the ciphertext is decrypted and the ready signal is set to high.

The input signals of the Counter Dec module are the clock signal, an asynchronous low active reset, the start signal and the signal to notify that the key is expanded. The output signals are the number of the round, the step, and the ready signal. A graphical illustration of the Counter Dec module inputs and outputs is given in Figure 86.

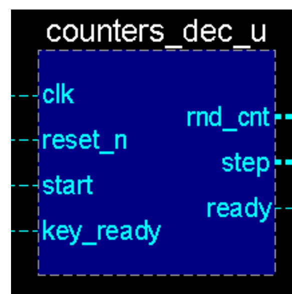

Figure 86 - Counters Dec module

11.3 Key Expansion Dec Module

Since the key is expanded at the beginning for decryption, a different implementation for the key expansion was employed. Rather than generating one new RoundKey in every round, which is equal to 5 steps or 5 clock cycles, in this module a new RoundKey is generated every 2 clock cycles.

The modules included in the Key Expansion Dec Module are:

- Round Key Dec Module
- Round Constant Dec Module

A block diagram showing the modules and their connections is presented in Figure 87.

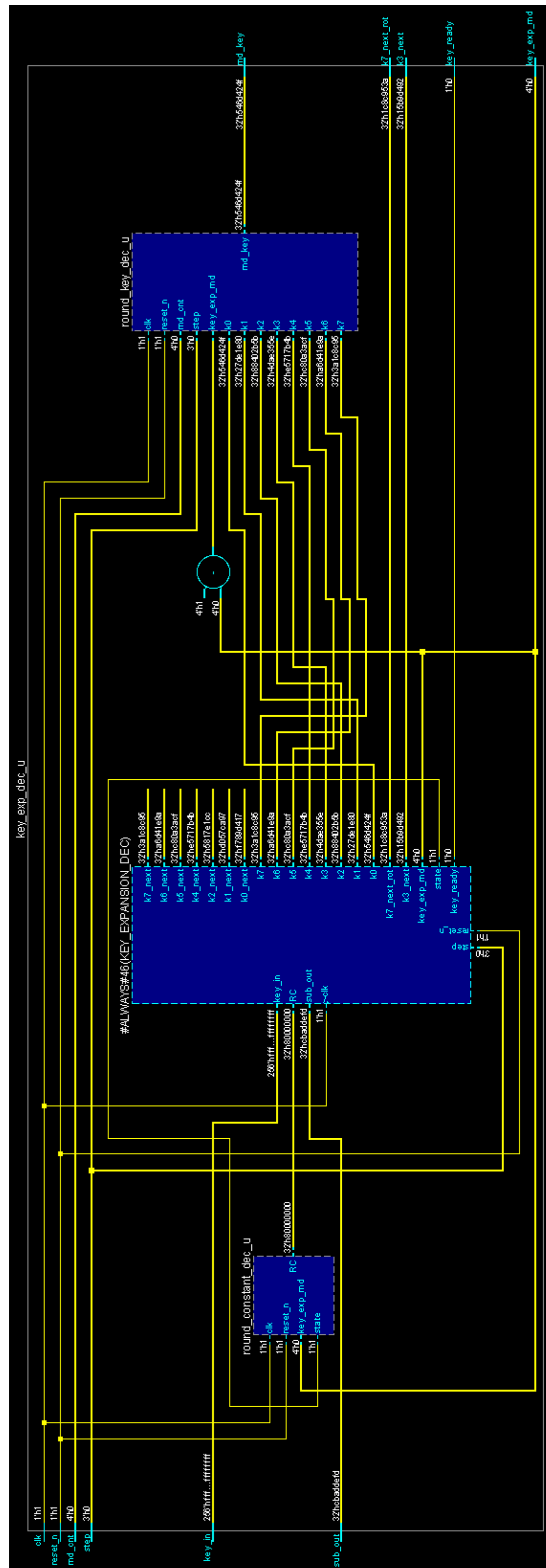

Figure 87 - Modules instantiated in Key Expansion Dec Module

The input signals of the Key Expansion Dec module are the clock signal, an asynchronous low active reset, the number of the round, the step, the key, and the results from S-box instances. The output signals are the number of the round in the KeyExpansion, columns required to be substituted, the ready signal of the KeyExpansion and the RoundKey. A graphical illustration of the Key Expansion Dec module inputs and outputs is given in Figure 88.

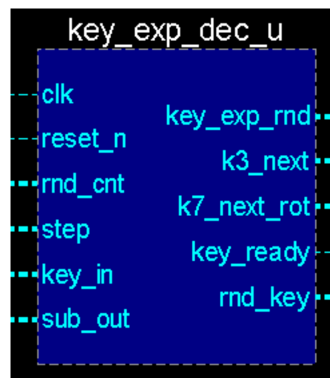

Figure 88 - Key Expansion Dec module

11.3.1 Round Key Dec Module

This module apart from selecting the required RoundKey in the same way encryption does, it stores all the expanded key in registers defined by the number of the round for KeyExpansion.

The input signals of the Round Key Dec module are the clock signal, an asynchronous low active reset, the number of the round, the step, the key, the round of KeyExpansion, and the columns that were used to expand the key. The only output signal is the RoundKey. A graphical illustration of the Round Key Dec module inputs and outputs is given in Figure 89.

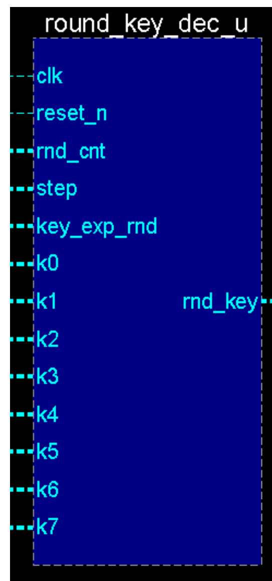

Figure 89 - Round Key Dec module

11.3.2 Round Constant Dec Module

The round constant is calculated with the round and the step of the KeyExpansion, compared to the module for encryption which uses the round and the step of the encryption.

The input signals of the Round Constant Dec module are the clock signal, an asynchronous low active reset, the number of the KeyExpansion round and the step of the KeyExpansion. The only output signal is the RoundConstant. A graphical illustration of the Round Constant Dec module inputs and outputs is given in Figure 90.

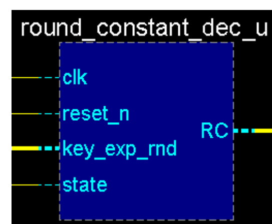

Figure 90 - Round Constant Dec Module

11.4 Inv Mix Columns Module

This module contains the same functions as the Mix Columns module. The only difference is the submodules instantiated in it which operate over the columns of the state the InvMix step.

The input signals of the Inv Mix Columns module are the clock signal, an asynchronous low active reset, the number of the round, the step, the plaintext, the output of the Inv ShiftRows module, the output of the Inv SubBytes module, and the RoundKey. The only output signal is the processed column. A graphical illustration of the Inv Mix Columns module inputs and outputs is given in Figure 91.

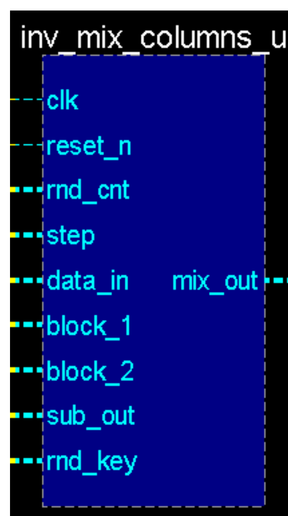

Figure 91 – Inv Mix Columns module

11.4.1 Inv Mix Word Module

The InvMix step is performed in this module. It is important to remark the order of the steps that conforms the Mix and InvMix step. The Mix step computes the

MixColumns step before the AddRoundKey step, while in the InvMix step the first step performed is the AddRoundKey step and after that the InvMixColumns step. Apart form that differences the operations are implemented in the same way. In both steps, the mixing step is ignored for round 0 and round 14.

The input signals of the Inv Mix Word module are the number of the round, the RoundKey and the column to be processed. The only output signal is the processed column. A graphical illustration of the Inv Mix Word module inputs and outputs is given in Figure 92.

Figure 92 – Inv Mix Word module

11.4.1.1 Inv ByteMix Module

As in encryption, four instances of the module for modular multiplication are required. The fixed polynomial used for the InvMixColumns step is the inverse of the one used for the MixColumns step. The same module can be used to obtain the four output bytes of the matrix multiplication reordering its inputs as shown in (103).

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} E\ B\ D\ 9 \\ 9\ E\ B\ D \\ D\ 9\ E\ B \\ B\ D\ 9\ E \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} E(a_0) + B(a_1) + D(a_2) + 9(a_3) \\ 9(a_0) + E(a_1) + B(a_2) + D(a_3) \\ D(a_0) + 9(a_1) + E(a_2) + B(a_3) \\ B(a_0) + D(a_1) + 9(a_2) + E(a_3) \end{bmatrix}$$
$$= \begin{bmatrix} E(a_0) + B(a_1) + D(a_2) + 9(a_3) \\ E(a_1) + B(a_2) + D(a_3) + 9(a_0) \\ E(a_2) + B(a_3) + D(a_0) + 9(a_1) \\ E(a_3) + B(a_0) + D(a_1) + 9(a_2) \end{bmatrix} \quad (103)$$

The input signals of the Inv ByteMix module are the four bytes in the column. The only output signal is a byte, which forms the column. A graphical illustration of the Inv ByteMix module inputs and outputs is given in Figure 93.

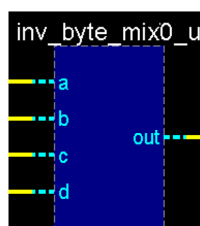

Figure 93 – Inv ByteMix module

11.4.1.2 Multiply by 2 Module

This module is the exact same module used for encryption. No difference between them exists. The input signal of the Multiply by 2 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 2 module inputs and outputs is given in Figure 94.

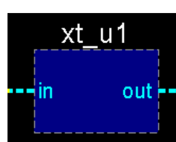

Figure 94 - Multiply by 2 module

11.4.1.3 Multiply by 9 Module

Modular multiplication of a polynomial by 9 is achieved using the module for modular multiplication by 2 and chaining their inputs with outputs as shown in (104).

$$b(x) = a(x) \times (x^3 + 1) = \big((a(x) \times x) \times x\big) \times x + a(x) \tag{104}$$

The input signal of the Multiply by 9 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 9 module inputs and outputs is given in Figure 95.

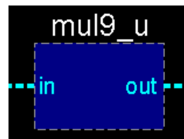

Figure 95 - Multiply by 9 module

11.4.1.4 Multiply by D Module

In the same way as modular multiplication by 9, this module is implanted chaining inputs and outputs of the Multiply by 2 module as shown in (105).

$$\begin{aligned} b(x) &= a(x) \times (x^3 + x^2 + 1) \\ &= \big((a(x) \times x) \times x\big) \times x + (a(x) \times x) \times x + a(x) \end{aligned} \tag{105}$$

The input signal of the Multiply by D module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by D module inputs and outputs is given in Figure 96.

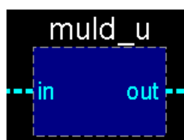

Figure 96 - Multiply by D module

11.4.1.5 Multiply by B Module

Modular multiplication by B is also implemented using the Multiply by 2 module as shown in (106).

$$b(x) = a(x) \times (x^3 + x + 1) = \big((a(x) \times x) \times x\big) \times x + a(x) \times x + a(x) \qquad (106)$$

The input signal of the Multiply by 2 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 2 module inputs and outputs is given in Figure 97.

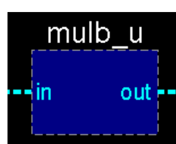

Figure 97 - Multiply by B module

11.4.1.6 Multiply by E Module

Modular multiplication by E is implemented as shown in (110).

$$b(x) = a(x) \times (x^3 + x + 1) = \big((a(x) \times x) \times x\big) \times x + a(x) \times x + a(x) \qquad (107)$$

The input signal of the Multiply by 2 module is the byte to be processed. The only output signal is the processed byte. A graphical illustration of the Multiply by 2 module inputs and outputs is given in Figure 98.

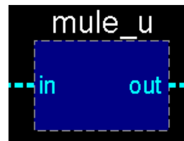

Figure 98 - Multiply by E module

11.5 Inv SubBytes Module

Since the key is expanded before the process of decryption, rather than using the number of the round to use the S-box instances as for encryption, the module uses the number of round of the KeyExpansion.

The input signals of the Inv SubBytes module are the clock signal, an asynchronous low active reset, the round in the expanded key, the step, columns of the expanded key to be substituted and the output of the Inv MixColumns module. The only output signal is the processed column. A graphical illustration of the Inv SubBytes module inputs and outputs is given in Figure 99.

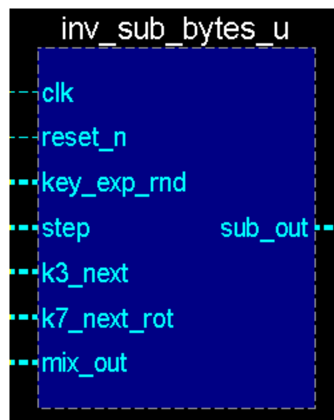

Figure 99 – Inv SubBytes module

11.6 Inv ShiftRows Module

The transposition of bytes is done storing the output bytes of the Inv SubBytes module in the corresponding registers. The only difference between this module and its inverse module is the position of the bytes in the registers.

The input signals of the Inv SubBytes module are the clock signal, an asynchronous low active reset, the number of the round, the step, the output of the Inv SubBytes module and the output of the Inv MixColumns module. The only output signals are the states after transposition. A graphical illustration of the Inv ShiftRows module inputs and outputs is given in Figure 100.

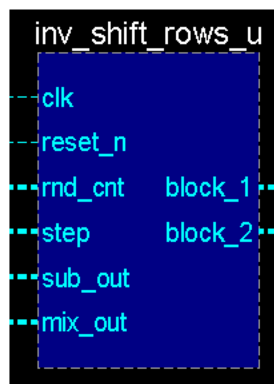

Figure 100 – Inv ShiftRows module

12 Simulations

In this chapter the steps that were followed to validate the implemented cipher for encryption, as well as for decryption are described. The generated simulations for validation are based on The Advance Encryption Standard Algorithm Validation Suite (AESAVS) [86].

AESAVS specifies Known Answer Tests and the Monte Carlo Test for encryption and decryption. NIST provides for these tests the AES Known Answer Test (KAT) Vectors, as well as AES Monte Carlo Test (MCT) Sample Vectors to validate results in simulations [87].

The vectors used by the simulations were taken from the NIST files without initialization vectors, those vectors are used for different modes of operation. The modes of operation are out of the scope of this work. Therefore, the mode that does not require initialization vectors is the Electronic Codebook (ECB) mode. This mode only requires a key and the plaintext or ciphertext depending on the cipher.

12.1 Encryption Simulation

Two different simulations were run to validate encryption in the cipher. The first test consists of 405 encryptions using the KAT vectors for encryption included in the AES KAT Vectors files related to the ECB mode of operation. The second test is a Monte Carlo Test that encrypts 100,000 different pairs of keys with plaintexts.

12.1.1 Known Answer Tests

The KAT files specified for the ECB mode of operation provides in total 405 different sets of vectors. Each set is composed by three elements: a cipher key, a plaintext, and the expected ciphertext after encryption.

The simulation was designed to read the pair of keys and plaintexts from files, encrypt the plaintext with the key and compare the obtained ciphertext with the expected ciphertext from a file. The results of the simulation are saved in a file displaying the index of the test beginning in zero, the cipher key, the plaintext, the obtained ciphertext, the expected ciphertext and the test result, which is either "Passed" or "Failed" as shown in Figure 101.

```
Test :              0
Key :                   0000000000000000000000000000000000000000000000000000000000000000
Plaintext :             014730f80ac625fe84f026c60bfd547d
Obtained Ciphertext :   5c9d844ed46f9885085e5d6a4f94c7d7
Expected Ciphertext :   5c9d844ed46f9885085e5d6a4f94c7d7
Test Result : PASSED

Test :              1
Key :                   0000000000000000000000000000000000000000000000000000000000000000
Plaintext :             0b24af36193ce4665f2825d7b4749c98
Obtained Ciphertext :   a9ff75bd7cf6613d3731c77c3b6d0c04
Expected Ciphertext :   a9ff75bd7cf6613d3731c77c3b6d0c04
Test Result : PASSED
```

Figure 101 – AESAVS KAT simulation results for first two tests for encryption

The simulation stops the tests if in one of the cases the obtained ciphertext does not match the expected ciphertext. If all the cases pass the test, at the end of the results file a comment is displayed notifying that the simulation is done and its result is equal to "Passed" as shown in Figure 102.

```
Test :          404
Key :                   ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Plaintext :             00000000000000000000000000000000
Obtained Ciphertext :   4bf85f1b5d54adbc307b0a048389adcb
Expected Ciphertext :   4bf85f1b5d54adbc307b0a048389adcb
Test Result : PASSED

###   Simulation DONE   ###
Simulation Result : PASSED
```

Figure 102 – AESAVS KAT simulation results for last test for encryption

12.1.2 Monte Carlo Test

The Monte Carlo Test runs 100,000 different encryptions. The test uses the same key provided by NIST for the first 1,000 cases and every 1,000 cases the key is updated. The key is updated 100 times, using 1,000 different plaintexts for each key.

In the first case, the ciphertext obtained from the plaintext provided by NIST is used as the plaintext for the next case using the same key to obtain a new ciphertext. This process chains the outputs to the inputs.

After the first 1,000 encryptions the last two ciphertexts are concatenated to create a new 256-bit vector. This vector is added with a bitwise XOR to the cipher key used for the 1,000 cases to update the value cipher key to be used for the next 1,000 cases. Taking into account that each key encrypts 1,000 plaintexts, and there are 100 cipher keys, the total number of cases is 100,000.

As in the KAT, the results after each 1,000 cases are saved in a file. Figure 103 demonstrates the chaining of the obtained ciphertext with the next plaintext. In contrast to the KAT, the expected ciphertext is only displayed at the end of the Monte Carlo Test as shown in Figure 104, since it only needs to fail once in the 100,000 cases to provide an unexpected result.

```
Test :           0
Key :          f9e8389f5b80712e3886cc1fa2d28a3b8c9cd88a2d4a54c6aa86ce0fef944be0
Plaintext :    b379777f9050e2a818f2940cbbd9aba4
Ciphertext :   6893ebaf0a1fccc704326529fdfb60db

Test :           1
Key :          db9ea5a2284fa17fb63e13bf891c8e42e40f332527559801aeb4ab26126f2b3b
Plaintext :    6893ebaf0a1fccc704326529fdfb60db
Ciphertext :   f3c78a5e85e5439bf26d5818718157d6
```

Figure 103 – Monte Carlo Test simulation results for first two keys for encryption

```
Test :          99
Key :          312c5b43263c1af8d1e35c0f24d1004386ee1cc0100fb3adfb7107e3f4eaff5e
Plaintext :    5c8e622ddbd32ee79c17572e8b3ee61c
Ciphertext :   c5d2cb3d5b7ff0e23e308967ee074825

###   MONTE CARLO TEST DONE   ###
Obtained Ciphertext :   c5d2cb3d5b7ff0e23e308967ee074825
Expected Ciphertext :   c5d2cb3d5b7ff0e23e308967ee074825

Monte Carlo Test : PASSED
```

Figure 104 - Monte Carlo Test simulation result for encryption

12.2 Decryption Simulation

Similar simulations were designed to validate decryption using the same files provided by NIST. For these simulations the vectors used as inputs to the cipher were the ciphertexts, while the expected results are the plaintext vectors.

12.2.1 Known Answer Tests

Figure 105 demonstrates the modification in the selection of vectors for the input and output for each decryption. The obtained result after decryption is the plaintext contrary to encryption. When the results in all case match the expected plaintexts, the file generated by the simulation displays at the end the simulation result as presented in Figure 106. If one of the cases fails, the simulation stops,

preventing the next cases from being tested, in the same way that the encryption simulation does.

```
Test :           0
Key :                  0000000000000000000000000000000000000000000000000000000000000000
Ciphertext :           5c9d844ed46f9885085e5d6a4f94c7d7
Obtained Plaintext:    014730f80ac625fe84f026c60bfd547d
Expected Plaintext :   014730f80ac625fe84f026c60bfd547d
Test Result : PASSED

Test :           1
Key :                  0000000000000000000000000000000000000000000000000000000000000000
Ciphertext :           a9ff75bd7cf6613d3731c77c3b6d0c04
Obtained Plaintext:    0b24af36193ce4665f2825d7b4749c98
Expected Plaintext :   0b24af36193ce4665f2825d7b4749c98
Test Result : PASSED
```

Figure 105 - AESAVS KAT simulation results for first two tests for decryption

```
Test :         404
Key :                  ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Ciphertext :           4bf85f1b5d54adbc307b0a048389adcb
Obtained Plaintext:    00000000000000000000000000000000
Expected Plaintext :   00000000000000000000000000000000
Test Result : PASSED

###   Simulation DONE   ###
Simulation Result : PASSED
```

Figure 106 - AESAVS KAT simulation results for last test for decryption

12.2.2 Monte Carlo Test

The Monte Carlo Test for decryption follows the same process as the encryption simulation. However, a new cipher key and ciphertext is provided as shown in Figure 107.

The plaintext obtained after the last encryption is compared with the plaintext vector provided by NIST. The expected vector is only displayed at the end of the file generated by the simulation as demonstrated in Figure 108.

```
Test :              0
Key :             2b09ba39b834062b9e93f48373b8dd018dedf1e5ba1b8af831ebbacbc92a2643
Ciphertext :      89649bd0115f30bd87856761022 3a59d
Plaintext :       1f9b9b213f1884fa98b62dd6639fd33b

Test :              1
Key :             58ac71619fdc3ac73a17f285319e1cd492766ac485030e02a95d971daab5f578
Ciphertext :      1f9b9b213f1884fa98b62dd6639fd33b
Plaintext :       aecd334ef8fb0c51b6896ae065d8be28
```

Figure 107 – Monte Carlo Test simulation results for first two keys for decryption

```
Test :             99
Key :             9977c985745bc33954a2ce898bc8febdaa2cf3bb4210e6a72ad1ec65a4e54889
Ciphertext :      c83e20e18f2b1457788954b49fd84307
Plaintext :       e3d3868f578caf34e36445bf14cefc68

###   MONTE CARLO TEST DONE   ###
Obtained Plaintext :    e3d3868f578caf34e36445bf14cefc68
Expected Plaintext :    e3d3868f578caf34e36445bf14cefc68

Monte Carlo Test : PASSED
```

Figure 108 - Monte Carlo Test simulation result for decryption

The Verilog code to run the simulations, as well as more results are provided in Appendix E.

13 Test on Xilinx Artix-7 FPGA

In this chapter, the test design to observe the operation of the cipher on a Xilinx FPGA is explained. Furthermore, the structure and functions of the Graphical User Interface created to interact with the Xilinx board is addressed, as well as the external module required for data transmission. The description of a required top module that combines both functions encryption and decryption in one design is also considered in this section.

13.1 GUI

A Graphical User Interface was developed to send and receive data from the Xilinx board. The tool used to create the application was Qt Creator, due to its simplicity to design GUIs using C++. Qt Creator is a development environment which provides several C++ classes, as well as extensive documentation and support.

The created GUI includes functionalities to handle data and interact with the serial port of the PC. The GUI can:

1. Identify the connection of the Xilinx board to the serial port, so it can be detected even if it is connected to a different port.
2. Open and close the serial port connected to the Xilinx board.
3. Select one of the 405 different KAT.
4. Select the mode of behaviour of the cipher (encryption or decryption).

5. Transmit the array of bytes containing the *mode*, *key*, and *text* to encrypt or decrypt.

6. Receive the encrypted or decrypted data.

The first step to test the cipher with the GUI is connecting the board to the PC, after that, the user must press the button "*Open Port"* to open the serial port. After selecting a KAT case the respective key, plaintext and ciphertext are displayed. These vectors are obtained from a set of vectors stored in external files. There is a file for the 405 keys, a file storing the plaintext vectors and a file with ciphertext vectors.

The user must select the *mode* of the cipher to define the input and the expected output of the cipher. For instance, if the encryption option is selected as the *mode*, the *text* to send is displayed as the "*plaintext*" and the vector with the expected value is defined as the "*expected ciphertext*", whereas if the decryption option is selected, the text to send is defined as the "*ciphertext*" and the vector with the expected value as the "*expected plaintext*".

Finally, the user must press the button "*Send*" to transmit all the data to the board. The application then receives the data transmitted from the board and displays this information as the *obtained plaintext* or *obtained ciphertext* depending on the *mode*. The application compares the obtained and the expected value to display a test result. If both vectors have the same value, it is displayed "*PASSED*", otherwise it displays "*FAILED*" as presented in Figure 109.

Figure 109 - Graphical User Interface for validation of AES-256 implementation

13.2 Architecture

To receive and transmit data, a Pmod USBUART interface board was connected to the FPGA Pmod ports and a top module was implemented containing modules for asynchronous serial communication data.

The top module is conformed by four modules: a module to receive data from the PC called *serial_rx*, a module to transmit data to the PC called *serial_tx* and the encryption and decryption modules called *aes256_enc* and *aes256_dec* respectively as presented in Figure 110.

The interaction between the PC and FPGA begins with the GUI sending an array of bytes though the serial port. The vector is conformed by the *mode* of the cipher, a *key,* and a *text* to be processed. The *mode* defines the behaviour of the cipher. If the byte is equal to 00, the cipher encrypts the *text* with the *key* provided, otherwise it decrypts the text. The *text* is called the *plaintext* when it is required to be encrypted, for decryption the input *text* is called the *ciphertext.*

The serial receiver module called *serial_rx* receives the array and sorts each byte in the corresponding register. Three registers are defined to store each element in the array. The *mode* byte is stored in an 8-bit register, the *key* in a 256-bit register and the *text* in a 128-bit register. Both encryption and decryption modules read the registers where the *key* and *text* are stored, and depending on the value stored in the *mode* register the top module sends a start signal either to the encryption or the decryption module. Depending on the *mode* register the top module decides which output to select between the encryption and the decryption module. Finally, the selected output is sent using the serial transmitter module *serial_tx*. A graphical representation of the system architecture is given in Figure 110.

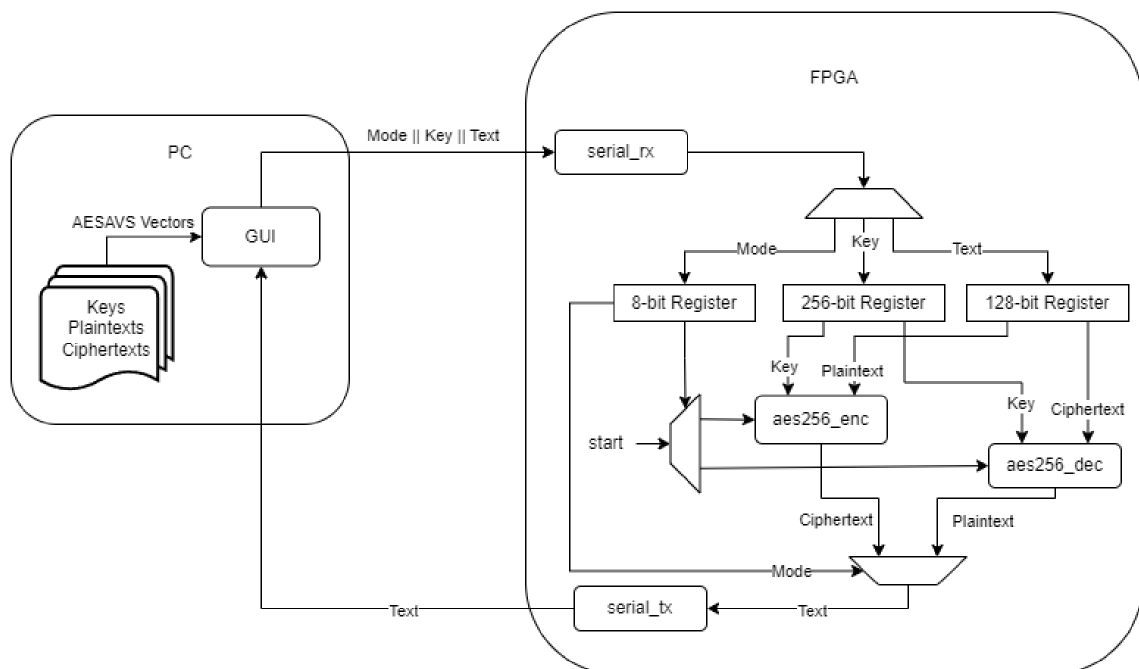

Figure 110 - Architecture to test the implemented cipher on a Xilinx FPGA

A state machine with four states was designed to control the sequence of the tasks. The first state *RX_DATA* waits until all the bytes sent by the PC are received. Every time the *serial_rx* module receives a byte, this is stored in a register. A counter determines if all the bytes have been received. When the counter reaches its maximum value, the last byte received is stored and the signal *rx_done* is set to high triggering a transition to the *AES* state. In this state, depending on the *mode*, the *aes256_enc* or the *aes256_dec* module ciphers the *text* with the *key*. When the *text* has been encrypted or decrypted the corresponding module set the signal *aes_ready* to high triggering the transition to the next state called *TX_DATA*. In this state, a byte from the output of one of the cipher modules is selected by a counter. The transition to the next state called *TX_BUSY* is triggered by the next clock event. In this state, the *serial_tx* module transmits the byte to the PC through the Pmod interface. After the byte has been sent, the state transitions to the previous state to select a new byte to be sent. The counter reaches its maximum value, when all the bytes have been sent. In this case, the signal *tx_done* is set to high triggering a transition to the *RX_DATA*.

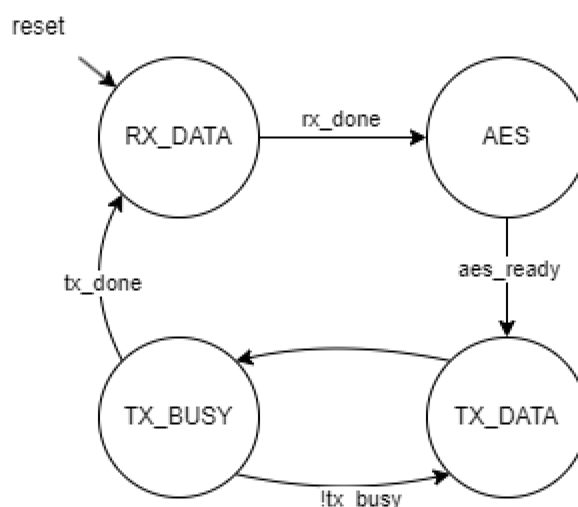

Figure 111 - State Machine to receive, encrypt or decrypt data and transmit data.

14 Results

In this chapter, results obtained from the design implemented are discussed compared to works from other sources, such as designs presented in Section 8, as well as open-source designs.

14.1 State-of-the-Art analysis

Researchers have implemented different AES designs on FPGAs through the years, being Xilinx FPGAs the most used for comparison with other implementations. AES designs depend on the length of the key and most of designs focus on the implementation of AES with a 128-bit key since the designs can be easily expanded to keys with different length. However, some works have implemented designs for 192- or 256-bit keys. These designs are expected to take more clock cycles to provide the processed data and use more resources depending on its design approach.

A comparison of the results obtained from designs presented in Section 8 is shown in Table 11. This analysis compares resources used such as slice LUTs, slice registers and slices, maximum frequency reached in MHz, clock cycles and throughput in Gbps. Some characteristics for some designs are not provided.

Design	Device	Key length	Slice LUTs	Slice Registers	Slices	Max. Freq. (MHz)	Clock Cycles	Through-put (Gbps)
[77]	Artix-7	128	4115	3987	x	x	x	x
[78]	Artix-7	128	x	x	359 (+ 8 BRAM)	311.72	59	x
[79]	Virtex-6	128	x	x	1002 (+ 50 BRAM)	254.453	x	x
[80]	Kintex-7	128	19312	8311	x (+ 42 BRAM)	x	2758	17.80
[81]	Virtex-7	192	43673	48000	x	428.996	x	54.52
[83]	Virtex-6	128	3028	8925	2252 (+244 BRAM)	470.998	31	60.29
[84]	Virtex-7	256	1814	836	x	161	74	0.278

Table 11 - AES implementations comparison in terms of resources and performance

Additionally, a comparison between open-core designs demonstrating the number and percentage of slice LUTs and slice registers required is presented in

Table 12. These designs were all synthetized using Xilinx Vivado Suite after selecting a Xilinx Artix-7 FPGA as the target device. Some designs differ to each other in key length or functions have i.e., [85] and [88] contained both encryption and decryption functions and accept more than one key length.

Design	Enc/Dec	Key length	Slice LUTs	LUT Util%	Slice Registers	FF Util%
[85]	Enc/Dec	128/192/256	2756	2.05	1538	0.57
[88]	Enc/Dec	128/256	3327	2.47	2990	1.11
[89]	Enc	128	1690	1.26	1242	0.46
[90]	Enc	128	9719	7.22	3712	1.30
[91]	Enc	128	4402	3.27	9519	3.54

Table 12 – AES implementation comparison between open-source designs

More details were observed modifiying design [85] to split the design into one for encryption and another for decryption both using a 256-bit key. The results of both new designs are presented in Table 13.

Design	Enc/Dec	Key length	Slice LUTs	LUT Util%	Slice Registers	FF Util%
Enc [85]	Enc	256	1717	1.28	1117	0.41
Dec [85]	Dec	256	2114	1.57	1117	0.41

Table 13 - Encryption and decryption designs from [85]

14.2 Results and Discussion

The resource utilization between [84], [85] and the design from this work is compared in this section. The reason [84] was selected for analysis is that our implementation is based on its design approach as explained in Section 9, whereas design [85] was selected due to its key length and its functionality for encryption and decryption. The results obtained from these designs and ours are shown in Figure 112, Figure 113 and Figure 114 respectively.

Name	Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	Slice (75900)	LUT as Logic (303600)	LUT Flip Flop Pairs (303600)	Bonded IOB (600)
N AES_TOP	1814	836	128	785	1814	434	515

Figure 112 - Resource utilization [84]

Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)	BUFGCTRL (32)
aes	1717	1117	520	1

Figure 113 - Resource utilization from Enc [85]

Name	Slice LUTs (134600)	Slice Registers (269200)	Bonded IOB (400)	BUFGCTRL (32)
aes256_enc	1010	881	516	1

Figure 114 - Resource utilization from our design

The resource utilization reports generated by Xilinx Vivado for Enc [85] and our design are presented in Figure 115 and Figure 116 respectively, where the use of distributed RAM on Enc [85] to store the expanded key is shown.

```
+----------------------------+------+-----------+-------+
|         Site Type          | Used | Available | Util% |
+----------------------------+------+-----------+-------+
| Slice LUTs*                | 1717 |    134600 |  1.28 |
|   LUT as Logic             | 1629 |    134600 |  1.21 |
|   LUT as Memory            |   88 |     46200 |  0.19 |
|     LUT as Distributed RAM |   88 |           |       |
|     LUT as Shift Register  |    0 |           |       |
| Slice Registers            | 1117 |    269200 |  0.41 |
|   Register as Flip Flop    | 1117 |    269200 |  0.41 |
|   Register as Latch        |    0 |    269200 |  0.00 |
| F7 Muxes                   |    0 |     67300 |  0.00 |
| F8 Muxes                   |    0 |     33650 |  0.00 |
+----------------------------+------+-----------+-------+
```

Figure 115 – Resource utilization report from Vivado for Enc [85] on Artix-7 FPGA

Site Type	Used	Available	Util%
Slice LUTs*	1010	134600	0.75
LUT as Logic	1010	134600	0.75
LUT as Memory	0	46200	0.00
Slice Registers	881	269200	0.33
Register as Flip Flop	881	269200	0.33
Register as Latch	0	269200	0.00
F7 Muxes	0	67300	0.00
F8 Muxes	0	33650	0.00

Figure 116 - Resource utilization report from Vivado for our encryption design on Artix-7 FPGA

To compare decryption, the resource utilization reports generated by Xilinx Vivado for Dec [85] and our design are presented in .

Site Type	Used	Util%
Slice LUTs*	2114	1.57
LUT as Logic	2026	1.51
LUT as Memory	88	0.19
LUT as Distributed RAM	88	
LUT as Shift Register	0	
Slice Registers	1117	0.41
Register as Flip Flop	1117	0.41
Register as Latch	0	0.00
F7 Muxes	0	0.00
F8 Muxes	0	0.00

Figure 117 - Resource utilization report from Vivado for Dec [85] on Artix-7 FPGA

```
+------------------------+------+-------+
|       Site Type        | Used | Util% |
+------------------------+------+-------+
| Slice LUTs*            | 1735 |  1.29 |
|   LUT as Logic         | 1735 |  1.29 |
|   LUT as Memory        |    0 |  0.00 |
| Slice Registers        | 2850 |  1.06 |
|   Register as Flip Flop | 2850 |  1.06 |
|   Register as Latch    |    0 |  0.00 |
| F7 Muxes               |   64 |  0.10 |
| F8 Muxes               |    0 |  0.00 |
+------------------------+------+-------+
```

Figure 118 - Resource utilization report from Vivado for our decryption design on Artix-7 FPGA

In these reports, an increase in resources in our decryption design is observed compared to the encryption design since the expanded key must be stored. Furthermore, a constraint preventing memory usage was specified increasing the number of slice LUTs and slice registers due to the storage of the expanded key. The resources used for key expansion in both designs are shown in Figure 119 and Figure 120.

Name	Slice LUTs (134600)	Slice Registers (269200)
aes	2114	1117
u_ram_1 (xram_16x64_15)	284	0
u_ram_0 (xram_16x64)	284	0

Figure 119 – Distributed RAM usage in Dec [85]

Name	Slice LUTs (134600)	Slice Registers (269200)	F7 Muxes (67300)
aes256_dec	1735	2850	64
key_exp_dec_u (key_exp_dec)	1107	2511	64
round_key_dec_u (round_key_dec)	771	1952	64
round_constant_dec_u (round_consta	29	8	0

Figure 120 - Slice LUTs and slice registers usage for expanded key in our decryption design

15 Conclusions

In this work a different implementation approach was implemented to reduce the number of required instances for the KeyExpansion, SubBytes and MixColumns step as proposed in [84]. In this approach more than one step operates in the same clock cycle on 32 bits of the state, contrary to the conventional approach that operates on the 128-bit state one step per clock cycle. This approach demonstrates a reduction in area compared to the conventional approach. However, decryption takes more cycles than encryption since the same approach for KeyExpansion cannot be implemented due to the order in which the RoundKeys are used. Furthermore, decryption requires more area than encryption since the complete ExpandedKey must be stored compared to encryption which reuses the registers used by the RoundKeys every time a new one is calculated.

16 Future Work

A different approach to reduce the area required for decryption can be implemented, if first the ExpandedKey is computed reusing the same registers to store the RoundKeys in the same way as for encryption. From the last RoundKeys the ExpandedKey can be computed backwards as the new RoundKeys are required. This computation can reuse the S-box instances available in certain clock cycles as for encryption. This approach is not intended to reduce the number of clock cycles required to decrypt data, but to reduce the implementation area.

The designs here presented can be extended to be used for encryption or decryption of sets of blocks depending on the modes of operation described in [92].

Furthermore, several applications can make use of this designs to provide security to the hardware, such as bitstream protection to add confidentiality to the IP designs programmed into the FPGA. This can be achieved encrypting the bitstream before being programmed into the device and decrypting it in the device with a key stored previously in the device.

Bibliography

[1] National Institute of Standards and Technology, „Announcing the ADVANCED ENCRYPTION STANDARD (AES),“ Federal Information Processing Standards (FIPS) 197, Washington, D.C., 2001.

[2] J. Daemen und V. Rijmen, „Fields with a Finite Number of Elements,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 13.

[3] J. Daemen und V. Rijmen, „Polynomials and Bytes,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 16.

[4] National Institute of Standards and Technology, „Multiplication,“ in *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, pp. 10-11.

[5] National Institute of Standards and Technology, „Multiplication by x,“ in *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, pp. 11-12.

[6] J. Daemen und V. Rijmen, „Polynomials and Columns,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 16-17.

[7] National Institute of Standards and Technology, „Polynomials with Coefficients in GF(2^8),“ in *Announcing the ADVANCED ENCRYPTION*

STANDARD (AES), Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, pp. 12-13.

[8] J. Daemen und V. Rijmen, „Boolean Functions,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 22-23.

[9] J. Daemen und V. Rijmen, „Transpositions,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 24-25.

[10] J. Daemen und V. Rijmen, „Bricklayer Functions,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 25-26.

[11] J. Daemen und V. Rijmen, „Iterative Boolean Transformations,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 26.

[12] J. Daemen und V. Rijmen, „Block Cipher,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 26-27.

[13] J. Daemen und V. Rijmen, „Iterative Block Ciphers,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 27-28.

[14] J. Daemen und V. Rijmen, „Key-Alternating Block Ciphers,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 28-29.

[15] J. Daemen und V. Rijmen, „Correlation,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 91-92.

[16] J. Daemen und V. Rijmen, „Difference Propagation," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, 55, 2020, pp. 115-116.

[17] D. Pointcheval, „RSA Public-Key Encryption," in *Encyclopedia of Cryptography and Security*, T. H. C. A. van und S. Jajodia, Hrsg., New York, Springer, 2011, pp. 1069-1072.

[18] National Institute of Standards and Technology, „Recommendation for Key Management Part 3: Application-Specific Key Management Guidance," Washington, D.C., 2015.

[19] National Institute of Standards and Technology, „Secure Hash Standard (SHS)," FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS PUB) 180-4., Washington, D.C., 2015.

[20] National Institute of Standards and Technology, „The Keyed-Hash Message Authentication Code (HMAC)," FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS PUB) 198-1, Washington, D.C., 2008.

[21] S. McNeil, „Solving Today's Design Security Concerns (WP365)," Xilinx, 30 July 2012. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp365_Solving_Security_Concerns. [Zugriff am 04 March 2023].

[22] E. Peterson, „Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs Application Note (XAPP1084)," Xilinx, 13 June 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/xapp1084_tamp_resist_dsgns. [Zugriff am 20 March 2023].

[23] E. Peterson, „Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs Application Note (XAPP1098)," Xilinx, 25 March 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/xapp1098-tamper-resist-designs. [Zugriff am 20 March 2023].

[24] G. Crow, „Advanced Security Schemes for Spartan-3A/3AN/3A DSP FPGAs (WP267),“ Xilinx, 15 August 2007. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp267. [Zugriff am 18 June 2023].

[25] Xilinx, „7 Series FPGAs Configuration User Guide (UG470),“ Xilinx, 1 Febreuary 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug470_7Series_Config. [Zugriff am 15 June 2023].

[26] National Institute of Standards and Technology, „Data Encryption Standard,“ Federal Information Processing Standards Publications (FIPS PUB) 46-3 , Washington, D.C., 1999.

[27] J. Daemen und V. Rijmen, „The Advanced Encryption Standard Process,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 1.

[28] L. R. Knudsen, „Block Ciphers,“ in *van Tilborg, H.C.A., Jajodia, S. (eds) Encyclopedia of Cryptography and Security*, Springer, Boston, MA, 2011, p. 152.

[29] J. Daemen und V. Rijmen, „Specification of Rijndael,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 31.

[30] National Institute of Standards and Technology, „Introduction,“ in *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, p. 5.

[31] B. Kaliski, „Symmetric Cryptosystem,“ in *van Tilborg, H.C.A., Jajodia, S. (eds) Encyclopedia of Cryptography and Security*, Springer, Boston, MA, 2011, p. 1271.

[32] J. Daemen und V. Rijmen, „Input and Output for Encryption and Decryption,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 31-32.

[33] J. Daemen und V. Rijmen, „The Round Transformation," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 33-34.

[34] National Institute of Standards and Technology, „Algorithm Specification," in *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, p. 14.

[35] National Institute of Standards and Technology, „Cipher," in *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*, Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, p. 14.

[36] J. Daemen und V. Rijmen, „Decryption," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 46.

[37] J. Daemen und V. Rijmen, „Algebraic Properties," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 48.

[38] J. Daemen und V. Rijmen, „Rijndael," in *van Tilborg, H.C.A., Jajodia, S. (eds) Encyclopedia of Cryptography and Security*, Springer, Boston, MA, 2011, p. 1049.

[39] J. Daemen und V. Rijmen, „Key Schedule," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 44.

[40] J. Daemen und V. Rijmen, „Selection," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 45.

[41] National Institute of Standards and Technology, „AddRoundKey () Transformation," in *Announcing the ADVANCED ENCRYPTION*

STANDARD (AES), Washington, D.C., Federal Information Processing Standards (FIPS) 197, 2001, p. 18.

[42] J. Daemen und V. Rijmen, „The Key Addition," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 41.

[43] J. Daemen und V. Rijmen, „The SubBytes Step," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 34-37.

[44] J. Daemen und V. Rijmen, „The ShiftRows Step," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 37-38.

[45] J. Daemen und V. Rijmen, „Security," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 65.

[46] J. Daemen und V. Rijmen, „Security Goals," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 73-74.

[47] J. Daemen und V. Rijmen, „Efficiency," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 65.

[48] J. Daemen und V. Rijmen, „Versatility," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 66.

[49] J. Daemen und V. Rijmen, „Key Agility," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 66.

[50] J. Daemen und V. Rijmen, „Simplicity,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 66-67.

[51] J. Daemen und V. Rijmen, „The Equivalent Decryption Algorithm,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 48-51.

[52] J. Daemen und V. Rijmen, „Symmetry Across the Rounds,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 67-68.

[53] J. Daemen und V. Rijmen, „Nonlinearity and Diffusion Criteria,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 76.

[54] J. Daemen und V. Rijmen, „Resistance Against Differential and Linear Cryptanalysis,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 76-77.

[55] J. Daemen und V. Rijmen, „Local Versus Global Optimization,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 77-78.

[56] J. Daemen und V. Rijmen, „Key-Alternating Cipher Structure,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 79.

[57] J. Daemen und V. Rijmen, „The Wide Trail Strategy,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 128.

[58] J. Daemen und V. Rijmen, „Symmetry Within the Round Transformation,“ in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 68-69.

[59] J. Daemen und V. Rijmen, „Additional Benefits of Symmetry," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 70-71.

[60] J. Daemen und V. Rijmen, „Weight of a Trail," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 131-132.

[61] J. Daemen und V. Rijmen, „Branch Numbers and Two-Round Trails," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 133-135.

[62] J. Daemen und V. Rijmen, „Diffusion," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 132-133.

[63] J. Daemen und V. Rijmen, „The Diffusion Step θ," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 136-138.

[64] J. Daemen und V. Rijmen, „The MixColumns Step," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 39-41.

[65] J. Daemen und V. Rijmen, „The Linear Step Θ," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 138.

[66] J. Daemen und V. Rijmen, „A Lower Bound on the Byte Weight of Four-Round Trails," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 138-139.

[67] J. Daemen und V. Rijmen, „An Efficient Construction for Θ," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 139-140.

[68] J. Daemen und V. Rijmen, „A Key-Iterated Structure," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 140-142.

[69] J. Daemen und V. Rijmen, „Arithmetic Operations," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 71-72.

[70] J. Daemen und V. Rijmen, „The Number of Rounds," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 42-43.

[71] J. Daemen und V. Rijmen, „Key Schedule," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 43-46.

[72] J. Daemen und V. Rijmen, „The Cost of the Key Expansion," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 81.

[73] J. Daemen und V. Rijmen, „Key Expansion and Key Selection," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, p. 80.

[74] J. Daemen und V. Rijmen, „A Recursive Key Expansion," in *The Design of Rijndael: The Advanced Encryption Standard (AES)*, Springer, Berlin, Heidelberg, 2020, pp. 81-82.

[75] M. Janveja, B. Paul, G. Trivedi, G. Vijayakanthi, A. Agrawal, P. Jan und Z. Němec, „Design of Efficient AES Architecture for Secure ECG Signal Transmission for Low-power IoT Applications," in *2020 30th International Conference Radioelektronika (RADIOELEKTRONIKA)*, Bratislava, Slovakia, 2020.

[76] V. Garg und J. Brewer, „Telemedicine security: a systematic review," in *Journal of diabetes science and technology*, 2011, p. 768–777.

[77] K. Kumar, K. R. Ramkumar und A. Kaur, „A Design Implementation and Comparative Analysis of Advanced Encryption Standard (AES) Algorithm on FPGA," in *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, Noida, India, 2020.

[78] M. Rao, T. Newe und I. Grout, „AES implementation on Xilinx FPGAs suitable for FPGA based WBSNs," in *2015 9th International Conference on Sensing Technology (ICST)*, Auckland, New Zealand, 2015.

[79] S. M. Umar Talha, M. Asif, H. Hussain, A. Asghar und H. Ameen, „Efficient advance encryption standard (AES) implementation on FPGA using Xilinx system generator," in *2016 6th International Conference on Intelligent and Advanced Systems (ICIAS)*, Kuala Lumpur, Malaysia, 2016.

[80] S. Chen, W. Hu und Z. Li, „High Performance Data Encryption with AES Implementation on FPGA," in *2019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, Washington, DC, USA, 2019.

[81] M. S. Abdul-Karim, K. H. Rahouma und K. Nasr, „High Throughput and Fully Pipelined FPGA Implementation of AES-192 Algorithm," in *2020 International Conference on Innovative Trends in Communication and Computer Engineering (ITCE)*, Aswan, Egypt, 2020.

[82] J. Sunil, S. H. S, S. B. K und S. Santhameena, „Implementation of AES Algorithm on FPGA and on software," in *2020 IEEE International Conference for Innovation in Technology (INOCON)*, Bangluru, India, 2020.

[83] X. Zhang, M. Li und J. Hu, „Optimization and Implementation of AES Algorithm Based on FPGA," in *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, Chengdu, China, 2018.

[84] M. Gunasekaran, K. Rahul und S. Yachareni, „Virtex 7 FPGA Implementation of 256 Bit Key AES Algorithm with Key Schedule and Sub

Bytes Block Optimization," in *2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, Toronto, ON, Canada, 2021.

[85] M. Litochevsky und L. Dongjun, „AES_Hightroughput_Lowarea," 1 January 2011. [Online]. Available: https://github.com/freecores/aes_highthroughput_lowarea/blob/master/verilog/rtl/sbox.v. [Zugriff am 25 04 18].

[86] L. E. Bassham III , „The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)," National Institute of Standards and Technology, 2002.

[87] Information Technology Laboratory, „Cryptographic Algorithm Validation Program CAVP," National Institute of Standards and Technology, 16 03 2023. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers. [Zugriff am 18 04 2023].

[88] J. Strömbergson und O. Kindgren, „aes," 7 February 2023. [Online]. Available: https://github.com/secworks/aes. [Zugriff am 15 March 2023].

[89] Ahmad, „AES," 2 Ocrober 2022. [Online]. Available: https://github.com/ahegazy/aes. [Zugriff am 15 March 2023].

[90] „AES-Core-engine-," 27 February 2021. [Online]. Available: https://github.com/Gourav0486/AES-Core-engine-. [Zugriff am 15 March 2023].

[91] A. Salah, „AES-128 pipelined encryption," 6 September 2023. [Online]. Available: https://github.com/freecores/aes-128_pipelined_encryption. [Zugriff am 15 March 2023].

[92] M. Dworkin, „Recommendation for Block 2001 Edition Cipher Modes of Operation: Methods and Techniques," National Institute of Standards and Technology, 2001.

[93] M. Smerdon, „Security Solutions Using Spartan-3 Generation FPGAs (WP266),“ Xilinx, 22 April 2008. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp266. [Zugriff am 04 March 2023].

Appendix

Appendix A – Groups, Rings and Fields

Definition A.1. The structure $(G, +)$ conformed by set G and an arbitrary operation $(+)$ defined on its elements is considered an *Abelian group* if and only if the operation satisfies the following properties for the elements of G:

1. Closed: $$\forall\, a, b \in G : a + b \in G \tag{108}$$
2. Associative: $$\forall\, a, b, c \in G : (a + b) + c = a + (b + c) \tag{109}$$
3. Commutative: $$\forall\, a, b \in G : a + b = b + a \tag{110}$$
4. Neutral element: $$\exists\, 0 \in G, \forall\, a \in G : a + 0 = a \tag{111}$$
5. Inverse elements: $$\forall\, a \in G, \exists\, b \in G : a + b = 0 \tag{112}$$

Definition A.2. The structure $(R, +, \cdot)$ conformed by set R and two operations $(+, \cdot)$ defined on its elements is considered a *ring* if and only if:

1. Structure $(R, +)$ conformed by set R and operation $(+)$ meets the requirements to be considered an Abelian group.
2. Operation $(\cdot)$ in structure $(R, \cdot)$ is closed, and associative on its elements.
3. Operation $(+, \cdot)$ satisfy:

Distributivity: $$\forall\, a, b, c \in R : (a + b) \cdot c = (a \cdot c) + (b \cdot c) \tag{113}$$

If operation $(\cdot)$ also satisfies the property of commutativity, ring $(R, +, \cdot)$ is called *commutative ring*.

Definition A.3. The structure $(F, +, \cdot)$ conformed by set F and two operations $(+, \cdot)$ defined on its elements is considered a *field* if and only if:

1. $(F, +, \cdot)$ is a commutative ring.
2. Exists an inverse element in set F with respect to operation $(\cdot)$ for all elements of F, except for the neutral element of $(F, +)$ denoted by $\mathbf{0}$.

Appendix B – Substitution Tables

In this section tabular representation is given for the Rijndael S-box $\mathrm{S_{RD}}$, inverse Rijndael S-box $\mathrm{S_{RD}^{-1}}$, affine transformation $\mathrm{Aff_8}$, inverse affine transformation $\mathrm{Aff_8^{-1}}$ and the transformation $\mathrm{Inv_8}$ that maps an element with its inverse in $GF(2^8)$. The relationship between all the transformations is given by

$$\mathrm{S_{RD}}[a] = \mathrm{Aff_8}\big(\mathrm{Inv_8}(a)\big) \quad (114)$$

$$\mathrm{S_{RD}^{-1}}[a] = \mathrm{Inv_8}\big(\mathrm{Aff_8^{-1}}(a)\big). \quad (115)$$

B.1 Rijndael S-box S_{RD}

Tabular representation of the Rijndael substitution box $\mathrm{S_{RD}}$ used in the $\mathrm{SubBytes}$ step for encryption of a plaintext is presented in Table 14.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	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	AC	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	79
	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	AB	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 14 - Tabular representation of S-box $S_{RD}(xy)$

B.2 Rijndael inverse S-box $S_{\mathbf{RD}}^{-1}$

Table 15 presents a tabular representation of the Rijndael inverse substitution box $\mathrm{S_{RD}^{-1}}$ used in the $\mathrm{InvSubBytes}$ step for decryption of a ciphertext.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	7B
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 15 - Tabular representation of the inverse S-box $S_{RD}^{-1}(xy)$

B.3 Affine Transformation $\mathbf{Aff_8}$

Tabular representation of the affine transformation Aff_8 used to obtain the Rijndael substitution box S_{RD} is presented in Table 16.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	5D	42	1F	00	21	3E	9B	84	A5	BA	E7	F8	D9	C6
	1	92	8D	AC	B3	EE	F1	D0	CF	6A	75	54	4B	16	09	28	37
	2	80	9F	BE	A1	FC	E3	C2	DD	78	67	46	59	04	1B	3A	25
	3	71	6E	4F	50	0D	12	33	2C	89	96	B7	A8	F5	EA	CB	D4
	4	A4	BB	9A	85	D8	67	E6	F9	5C	43	62	7D	20	3F	1E	01
	5	55	4A	6B	74	29	36	17	08	AD	B2	93	8C	D1	CE	EF	F0
	6	47	58	79	66	3B	24	05	1A	BF	A0	81	9E	C3	DC	FD	E2
	7	B6	A9	88	97	CA	D5	F4	EB	4E	51	70	6F	32	2D	0C	13
	8	EC	F3	D2	CD	90	8F	AE	B1	14	0B	2A	35	68	77	56	49
	9	1D	02	23	3C	61	7E	5F	40	E5	FA	DB	C4	99	86	A7	B8
	A	0F	10	31	2E	73	6C	4D	52	F7	E8	C9	D6	8B	94	B5	AA
	B	FE	E1	C0	DF	82	9D	BC	A3	06	19	38	27	7A	65	44	5B
	C	2B	34	15	0A	57	48	69	76	D3	CC	ED	F2	AF	B0	91	8E
	D	DA	C5	E4	FB	A6	B9	98	87	22	3D	1C	03	5E	41	60	7F
	E	C8	D7	F6	E9	B4	AB	8A	95	30	2F	0E	11	4C	53	72	6D
	F	39	26	07	18	45	5A	7B	64	C1	DE	FF	E0	BD	A2	83	9C

Table 16 - Tabular representation of the affine transformation $Aff_8(xy)$

B.4 Inverse Affine Transformation $\mathbf{Aff_8^{-1}}$

Table 17 presents a tabular representation of the inverse affine transformation Aff_8^{-1} to obtain the Rijndael inverse substitution box S_{RD}^{-1}.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	05	4F	91	DB	2C	66	B8	F2	57	1D	C3	89	7E	34	EA	A0
	1	A1	EB	35	7F	88	C2	1C	56	F3	B9	67	2D	DA	90	4E	04
	2	4C	06	D8	92	65	2F	F1	BB	1E	54	8A	C0	37	7D	A3	E9
	3	E8	A2	7C	36	C1	8B	55	1F	BA	F0	2E	64	93	D9	07	4D
	4	97	DD	03	49	BE	F4	2A	60	C5	8F	51	1B	EC	A6	78	32
	5	33	79	A7	ED	1A	50	8E	C4	61	2B	F5	BF	48	02	DC	96
	6	DE	94	4A	00	F7	BD	63	29	8C	C6	18	52	A5	EF	31	7B
	7	7A	30	EE	A4	53	19	C7	8D	28	62	BC	F6	01	4B	95	DF
	8	20	6A	B4	FE	09	43	9D	D7	72	38	E6	AC	5B	11	CF	85
	9	84	CE	10	5A	AD	E7	39	73	D6	9C	42	08	FF	B5	6B	21
	A	69	23	FD	B7	40	0A	D4	9E	3B	71	AF	E5	12	58	86	CC
	B	CD	87	59	13	E4	AE	70	3A	9F	D5	0B	41	B6	FC	22	68
	C	B2	F8	26	6C	9B	D1	0F	45	E0	AA	74	3E	C9	83	5D	17
	D	16	5C	82	C8	3F	75	AB	E1	44	0E	D0	9A	6D	27	F9	B3
	E	FB	B1	6F	25	D2	98	46	0C	A9	E3	3D	77	80	CA	14	5E
	F	5F	15	CB	81	76	3C	E2	A8	0D	47	99	D3	24	6E	B0	FA

Table 17 - Tabular representation of the affine transformation $Aff_8^{-1}(xy)$

B.5 Inverse Element in $GF(2^8)$ Transformation $\mathbf{Inv_8}$

Tabular representation of the transformation Inv_8, which maps elements with its inverse in $GF(2^8)$ used to obtain the Rijndael substitution box S_{RD}, is presented in Table 18.

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	00	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
	2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
	3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
	4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
	5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
	6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
x	7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
	F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

Table 18 - Tabular representation of transformation $Inv_8(xy)$

Appendix C – AES-256 code for encryption

C.1 Top Module

```
`timescale 1ns / 1ps

// ---------- ADVANCED ENCRYPTION STANDARD --- AES-256 --- ENCRYPTION ----------

module aes256_enc  (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,
                    ready
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global async negative edge reset
input               start;              // start pulse
input       [255:0] key_in;             // 256-bit key
input       [127:0] data_in;            // 128-bit block size of plaintext
output wire [127:0] data_out;           // 128-bit ciphertext and also as storage block
output wire         ready;              // output ready


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [3:0]       rnd_cnt;            // round counter [0 to 14]
wire    [2:0]       step;               // step counter [0 to 4]
wire    [31:0]      mix_out;            // processed column after mix operation
wire    [31:0]      sub_out;            // processed column after substitution bytes operation
wire    [127:0]     block_2;            // storage block for shift rows operation
wire    [31:0]      rnd_key;            // round key for add round key operation
wire    [31:0]      k3, k7_rot;         // word 3 and word 7 used for key expansion


// ---------- MODULES INSTANTIATION ----------

// ---------- ROUND AND STEP COUNTERS ----------

counters        counters_u     (
                                clk,
                                reset_n,
                                start,
                                rnd_cnt,    // output
                                step,       // output
                                ready       // output
                                );
```

```
// ---------- MIX COLUMNS UNIT ----------

mix_columns     mix_columns_u  (
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                data_in,
                                data_out,
                                block_2,
                                sub_out,
                                rnd_key,
                                mix_out     // output
                                );

// ---------- SUB BYTES UNIT ----------

sub_bytes       sub_bytes_u    (
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                k3, k7_rot,
                                mix_out,
                                sub_out      // output
                                );

// ---------- SHIFT ROWS UNIT ----------

shift_rows      shift_rows_u   (
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                sub_out,
                                mix_out,
                                data_out,   // output
                                block_2     // output
                                );

// ---------- KEY EXPANSION UNIT ----------

key_exp         key_exp_u      (
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                key_in,
                                sub_out,
                                k3, k7_rot, // output
                                rnd_key     // output
                                );


endmodule
```

C.2 Counters Module

```
`timescale 1ns / 1ps

// ---------- ROUND AND STEP COUNTERS ----------

module counters    (
                    clk,
                    reset_n,
                    start,
                    rnd_cnt,            // output
                    step,               // output
                    ready               // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input               start;              // start encryption
output reg  [3:0]   rnd_cnt;            // round counter [0 to 14]
output reg  [2:0]   step;               // step counter [0 to 4]
output reg          ready;              // encryption done


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter           max_rnd = 4'd14;    // AES-256 requires 14 rounds (plus round 0)
parameter           max_steps = 4'd4;   // every round consists on 5 steps (0 to 4)

reg [2:0]           current_state, next_state;
parameter           IDLE = 3'd6,
                    S0 = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                    S4 = 3'd4,
                    DONE = 3'd5;


// ---------- STATE MACHINE ----------

always @ (posedge clk or negedge reset_n)
  begin: STATE_MEMORY

    if (!reset_n)   current_state <= IDLE;
    else            current_state <= next_state;

  end


always @ (current_state or start or ready or reset_n)
  begin: NEXT_STATE_LOGIC

    case (current_state)
      IDLE :    next_state = (start) ? S1 : IDLE;   // start step counter with start
      S0 :      next_state = (ready) ? DONE : S1;   // to DONE if ready, else continue encryption
      S1 :      next_state = S2;
      S2 :      next_state = S3;
      S3 :      next_state = S4;
      S4 :      next_state = S0;                    // resets after step = 4
      DONE:     next_state = (start) ? S1 : DONE;   // it can start from DONE state
      default : next_state = IDLE;
    endcase

  end
```

```
always @ (current_state or rnd_cnt)
  begin: OUTPUT_LOGIC

    case (current_state)
      IDLE :  begin
                step = 3'd0;
                ready = 1'b0;
              end

      S0 :    begin
                step = 3'd0;
                if (rnd_cnt == 0)   ready = 1'b1;   // when round counter finishes and step = 0
                else                ready = 1'b0;
              end

      S1 :    begin
                step = 3'd1;
                ready = 1'b0;
              end

      S2 :    begin
                step = 3'd2;
                ready = 1'b0;
              end

      S3 :    begin
                step = 3'd3;
                ready = 1'b0;
              end

      S4 :    begin
                step = 3'd4;
                ready = 1'b0;
              end

      DONE :  begin
                step = 3'd0;
                ready = 1'b1;       // Encryption DONE
              end

      default : begin
                  step = 3'd0;
                  ready = 1'b0;
                end
    endcase

  end

always @ (posedge clk or negedge reset_n)
  begin

    if (!reset_n)
      begin
        rnd_cnt <= 4'b0;                            // reset of round counter
      end
    else
      if(current_state == S3)                       // round counter only when step = 3
        if (rnd_cnt == max_rnd) rnd_cnt <= 4'b0;    // reset when round = 14
        else                    rnd_cnt <= rnd_cnt + 1;

  end


endmodule
```

C.3 Key Expansion Module

```
`timescale 1ns / 1ps

// ---------- KEY EXPANSION ----------

module key_exp     (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    key_in,
                    sub_out,
                    k3,k7_rot,
                    rnd_key
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [255:0] key_in;             // 256-bit key
input       [31:0]  sub_out;            // processed column after substitution bytes operation
output reg  [31:0]  k3,k7_rot;         // word 3 and word 7 rotated used for key expansion
output wire [31:0]  rnd_key;            // round key for add round key operation


// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [31:0]      k0_next, k1_next, k2_next, k3_next; // words used for first part of key expansion
reg     [31:0]      k4_next, k5_next, k6_next, k7_next; // words used for second part of key expansion
reg     [31:0]      k0, k1, k2;         // words used for first part of key expansion
reg     [31:0]      k4, k5, k6;         // words used for second part of key expansion
wire    [31:0]      RC;                 // round constant for key expansion


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: KEY_EXPANSION

    if (!reset_n)                       // reset words registers
      begin
        k0 <= 31'b0;
        k1 <= 31'b0;
        k2 <= 31'b0;
        k3 <= 31'b0;
        k4 <= 31'b0;
        k5 <= 31'b0;
        k6 <= 31'b0;
        k7_rot <= 31'b0;
        k0_next <= 31'b0;
        k1_next <= 31'b0;
        k2_next <= 31'b0;
        k3_next <= 31'b0;
        k4_next <= 31'b0;
        k5_next <= 31'b0;
        k6_next <= 31'b0;
        k7_next <= 31'b0;
      end
```

```
    else
        case (step)
                                        // round 0
          0:  if (rnd_cnt == 0)
                begin                   // last column of key_in is k7
                  {k0,k1,k2,k3,k4,k5,k6,{k7_rot[7:0],k7_rot[31:8]}} <= key_in;
                end


          1:  begin                     // First Part
                                        // when j mod Nk = 0, (j = 8,16...56)
                k0_next = k0 ^ sub_out ^ RC;
                k1_next = k0_next ^ k1;
                k2_next = k1_next ^ k2;
                k3_next = k2_next ^ k3;
                                        // Second Part
                k4_next = sub_out ^ k4; // when j mod Nk = 4, (j = 12,20...52)
                k5_next = k4_next ^ k5;
                k6_next = k5_next ^ k6;
                k7_next = k6_next ^ {k7_rot[7:0],k7_rot[31:8]}; // k7 NOT rotated
              end

          4:  begin                     // 2,4...12 for Mix_0 in 2,4...14
                if (rnd_cnt[0] == 0 && rnd_cnt > 1)
                  begin
                    k0 <= k0_next;      // update key columns for first part
                    k1 <= k1_next;
                    k2 <= k2_next;
                    k3 <= k3_next;
                  end                   // 3,5...13 for Mix_0 in 3,5...13
                if (rnd_cnt[0] == 1 && rnd_cnt > 2)
                  begin
                    k4 <= k4_next;      // update key columns for second part
                    k5 <= k5_next;
                    k6 <= k6_next;
                    k7_rot <= {k7_next[23:0],k7_next[31:24]};
                  end
              end

        endcase

  end


round_constant  round_constant_u   (
                                    clk,
                                    reset_n,
                                    rnd_cnt,
                                    step,
                                    RC          // output
                                    );

round_key       round_key_u        (
                                    clk,
                                    reset_n,
                                    rnd_cnt,
                                    step,
                                    key_in[255:224],
                                    k0,k1,k2,k3,
                                    k4,k5,k6,{k7_rot[7:0],k7_rot[31:8]},
                                    rnd_key     // output
                                    );

endmodule
```

C.4 Round Constant Module

```
`timescale 1ns / 1ps

/*
            Round Constant
        for K2  K4  K6  K8 K10 K12 K14
    i       1   2   3   4   5   6   7
    -----------------------------------
    RC[i]  01  02  04  08  10  20  40

*/

// ---------- ROUND CONSTANT ----------

module round_constant  (
                        clk,
                        reset_n,
                        rnd_cnt,
                        step,
                        RC                  // output
                        );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
output reg  [31:0]  RC;                 // round constant for key expansion


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: ROUND_CONSTANT

    if (!reset_n) RC <= 32'h0;          // reset RC register
                                        // rounds 1,3...13
    else if (step == 0 && rnd_cnt[0] != 0)
                                        // RC[i] = RC[i-1] << 1
        RC <= (rnd_cnt == 1) ? 32'h01000000 : {RC[30:0],1'b0};

  end


endmodule
```

C.5 Round Key Module

```
`timescale 1ns / 1ps

// ---------- ROUND KEY ----------

module round_key   (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    key_in,
                    k0,k1,k2,k3,
                    k4,k5,k6,k7,
                    rnd_key             // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [31:0]  key_in;             // 256-bit key
input       [31:0]  k0, k1, k2, k3;     // words used for first part of key expansion
input       [31:0]  k4, k5, k6, k7;     // words used for second part of key expansion
output reg  [31:0]  rnd_key;            // round key for add round key operation


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: ROUND_KEY

    if (!reset_n) rnd_key <= 32'h0;     // reset round_key register

    else
        case (step)
                                        // w0 for Mix_0 in round 0 (key_in[255:224])
          0:  rnd_key <= (rnd_cnt == 0) ? key_in:
                                        // for Mix_0 in 2,4...14 : 1,3...13
                        (rnd_cnt[0] == 0) ? k0 : k4;
                                        // for Mix_1 in 2,4...14 : 1,3...13
          1:  rnd_key <= (rnd_cnt[0] == 0) ? k1 : k5;
                                        // for Mix_2 in 2,4...14 : 1,3...13
          2:  rnd_key <= (rnd_cnt[0] == 0) ? k2 : k6;
                                        // for Mix_3 in 2,4...14 : 1,3...13
          3:  rnd_key <= (rnd_cnt[0] == 0) ? k3 : k7;

        endcase

  end


endmodule
```

C.6 MixColumns Module

```
`timescale 1ns / 1ps

// ---------- MIX COLUMNS ----------

module mix_columns (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    data_in,
                    block_1,
                    block_2,
                    sub_out,
                    rnd_key,
                    mix_out             // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [127:0] data_in;            // 128-bit plaintext
input       [127:0] block_1, block_2;   // storage blocks for shift rows operation
input       [31:0]  sub_out;            // processed column after sub bytes operation
input       [31:0]  rnd_key;            // round key for add round key operation
output wire [31:0]  mix_out;            // processed column after mix operation


// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [3:0]       rnd_cnt_mix;        // round counter used as input for mix column unit
reg     [31:0]      mix_in;             // column as input for mix operation


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: MIX_COLUMNS

    if (!reset_n)
      begin
        mix_in <= 32'b0;      // reset of mix_in register
        rnd_cnt_mix <= 4'b0;
      end

    else
      begin
        rnd_cnt_mix <= rnd_cnt;         // syncrhonize to use correct round counter along with mix_in
        case (step)
          0:  begin                     // Mix_0
                mix_in <= (rnd_cnt == 0) ? data_in[127:96]:                           // round 0
                          (rnd_cnt[0] != 1'b0) ? {block_1[127:104],sub_out[7:0]}:   // rounds 1,3...13
                                                 {block_2[127:104],sub_out[7:0]};   // rounds 2,4...14
              end                       // sub_out[7:0] is the last elemtent for first column
                                        // already considering shift rows after sub bytes

          1:  begin                     // Mix_1
                mix_in <= (rnd_cnt == 0) ? data_in[95:64]:          // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[95:64]:    // rounds 1,3...13
                                                 block_2[95:64];    // rounds 2,4...14
              end
```

```
          2:  begin                     // Mix_2
                mix_in <= (rnd_cnt == 0) ? data_in[63:32]:          // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[63:32]:    // rounds 1,3...13
                                                 block_2[63:32];    // rounds 2,4...14
              end

          3:  begin                     // Mix_3
                mix_in <= (rnd_cnt == 0) ? data_in[31:0]:           // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[31:0]:     // rounds 1,3...13
                                                 block_2[31:0];     // rounds 2,4...14
              end


        endcase
      end
  end


// ---------- MIX OPERATION BY WORD UNIT ----------

mix_w       mix_w_u    (
                        .round      (rnd_cnt_mix),      // round counter as condition to mix column
operation
                        .round_key  (rnd_key),          // round key for add round key operation
                        .in         (mix_in),           // column as input for mix operation
                        .out        (mix_out)           // processed column after mix operation
                        );


endmodule
```

C.7 Mix Word Module

```
// ---------- MIX COLUMNS FOR ENC WORD ----------

module mix_w    ( round, round_key, in, out );


// ---------- MODULE INTERFACE ----------

input       [3:0]   round;
input       [31:0]  round_key;
input       [31:0]  in;
output      [31:0]  out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       byte0_in, byte1_in, byte2_in, byte3_in;
wire    [7:0]       byte0_out, byte1_out, byte2_out, byte3_out;


// ---------- MODULE IMPLEMENTATION ----------

assign byte0_in[7:0] = in[31:24];
assign byte1_in[7:0] = in[23:16];
assign byte2_in[7:0] = in[15:8];
assign byte3_in[7:0] = in[7:0];

assign out = (round == 0 || round == 14) ? in ^ round_key : {byte0_out, byte1_out, byte2_out,
byte3_out} ^ round_key;


// ---------- MODULES INSTANTIATION ----------

byte_mix byte_mix0_u (.a(byte0_in), .b(byte1_in), .c(byte2_in), .d(byte3_in), .out(byte0_out));

byte_mix byte_mix1_u (.a(byte1_in), .b(byte2_in), .c(byte3_in), .d(byte0_in), .out(byte1_out));

byte_mix byte_mix2_u (.a(byte2_in), .b(byte3_in), .c(byte0_in), .d(byte1_in), .out(byte2_out));

byte_mix byte_mix3_u (.a(byte3_in), .b(byte0_in), .c(byte1_in), .d(byte2_in), .out(byte3_out));


endmodule
```

C.8 ByteMix Module

```
// ---------- BYTE MIX COLUMNS ----------

module byte_mix (   a, b, c, d, out );

// ---------- MODULE INTERFACE ----------

input       [7:0]   a, b, c, d;
output      [7:0]   out;

// ---------- MODULE REGISTERS AND SIGNALS ----------

wire [7:0] mul2, mul3;

// ---------- MODULE IMPLEMENTATION ----------

assign out = mul2 ^ mul3 ^ c ^ d;

// ---------- MODULES INSTANTIATION ----------

xtimes      xt_u       (    .in(a), .out(mul2)  );

MUL3        mul3_u     (    .in(b), .out(mul3)  );

endmodule
```

C.9 Multiply by 3 Module

```
// ---------- MULTIPLY BY 3 ----------

module MUL3     ( in, out );

// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;

// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       xt;

// ---------- MODULE IMPLEMENTATION ----------

assign out = xt ^ in;

// ---------- MODULES INSTANTIATION ----------

xtimes  xt_u    ( .in(in), .out(xt) );

endmodule
```

C.10 Multiply by 2 Module

```
// ---------- MULTIPLY BY 2 ----------

/*
   2 = 0010 = x ;   v = 1011 0010 = x7 + x5 + x4 + x ;   P = 'h11B = 1 0001 1011 = x8 + x4 + x3 + x +
1
   v*2 = v*x = (x7 + x5 + x4 + x)x = x8 + x6 + x5 + x2;
   v*2(modP) = (x8 + x6 + x5 + x2) + (x8 + x4 + x3 + x + 1) = x6 + x5 + x4 + x3 + x2 + x + 1 = 0111
1111
*/

module xtimes    ( in, out );


// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [3:0]       xt;


// ---------- MODULE IMPLEMENTATION ----------

// same as: xt = in[7] ? 1101 : 0000;
assign xt[3] = in[7];
assign xt[2] = in[7];
assign xt[1] = 1'b0;
assign xt[0] = in[7];

// if in = 1011 0010, in * 2 (mod P)= 1 0110 0100
//                                xor 1 0001 1011 = 011 1111 1
// same as: {in[6:4], xt ^ in[3:0], in[7]} =        011,1111,1

assign out[7:5] = in[6:4];
assign out[4:1] = xt[3:0] ^ in[3:0];  // 1101 ^ 0010 = 1111
assign out[0]   = in[7];

endmodule
```

C.11 SubBytes Module

```
`timescale 1ns / 1ps

// ---------- SUB BYTES ----------

module sub_bytes   (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    k3, k7_rot,
                    mix_out,
                    sub_out              // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;               // global clock
input               reset_n;           // global negative active reset
input        [3:0]  rnd_cnt;           // round counter [0 to 14]
input        [2:0]  step;              // step counter [0 to 4]
input        [31:0] k3, k7_rot;        // word 3 and word 7 rotated used for key expansion
input        [31:0] mix_out;           // processed column after mix operation
output wire  [31:0] sub_out;           // processed column after sub bytes operation

// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [31:0]      sub_in;            // column as input for S-Box


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: SUB_BYTES

    if (!reset_n) sub_in <= 32'b0;     // reset sub_in register

    else if (step == 0)                // Key expansion in every step 0
                                       // K2,K4...K14 : K3,K5...K13
      sub_in <= (rnd_cnt[0] == 1) ? {k7_rot} : k3[31:0]; // rounds 1 to 13
      //Rotate to use as input for S-Box when j mod Nk = 0, j = 8,16...56

    else                               // Sub_0, Sub_1, Sub_2, Sub_3
      sub_in <= mix_out;               // steps 1,2,3 and 4

  end


// ---------- S-BOX UNITS ----------

sbox            sbox0_u        (
                                sub_in[31:24],
                                sub_out[31:24]
                                );

sbox            sbox1_u        (
                                sub_in[23:16],
                                sub_out[23:16]
                                );

sbox            sbox2_u        (
                                sub_in[15:8],
                                sub_out[15:8]
                                );

sbox            sbox3_u        (
                                sub_in[7:0],
                                sub_out[7:0]
                                );

endmodule
```

C.12 ShiftRows Module

```
`timescale 1ns / 1ps

/*
            Initial Matrix
    127:120 | 95:88 |  63:56  |  31:24
    119:112 | 87:80 |  55:48  |  23:16
    111:104 | 79:72 |  47:40  |  15:8
    103:96  | 71:64 |  39:32  |   7:0

            After Shift Rows
    127:120 |  95:88 |  63:56  |  31:24
     87:80  |  55:48 |  23:16  | 119:112
     47:40  |  15:8  | 111:104 |  79:72
      7:0   | 103:96 |  71:64  |  39:32

*/

// ---------- SHIFT ROWS and OUTPUT ----------

module shift_rows  (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    sub_out,
                    mix_out,
                    block_1,            // output
                    block_2             // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [31:0]  sub_out;            // processed column after substitution bytes operation
input       [31:0]  mix_out;            // processed column after mix operation
output reg  [127:0] block_1;            // storage block used after first shift rows operation and to
store final output(data_out)
output reg  [127:0] block_2;            // storage block used after second shift rows operation


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: SHIFT_ROWS

    if (!reset_n)                       // reset blocks registers
      begin
        block_1 <= 128'b0;
        block_2 <= 128'b0;
      end

    else
        case (step)

          0:  begin                         // Shift_3
                if(rnd_cnt[0] != 1'b0)      // for rounds 1,3...13
                  {block_1[31:24],block_1[55:48],block_1[79:72],block_1[103:96]} <= sub_out;
                else                        // for rounds 2,4...14
                  {block_2[31:24],block_2[55:48],block_2[79:72],block_2[103:96]} <= sub_out;
              end
                                            // data_out first column
          1:  if(rnd_cnt == 4'd14) block_1[127:96] <= mix_out;
```

```
         2:  begin
               if(rnd_cnt == 4'd14)        // data_out second column
                 block_1[95:64] <= mix_out;
               else
                 begin                     // Shift_0
                   if(rnd_cnt[0] == 1'b0)  // for rounds 0,2...12
                     {block_1[127:120],block_1[23:16],block_1[47:40],block_1[71:64]} <= sub_out;
                   else                    // for rounds 1,3..13
                     {block_2[127:120],block_2[23:16],block_2[47:40],block_2[71:64]} <= sub_out;
                 end
             end

         3:  begin
               if(rnd_cnt == 4'd14)        // data_out third column
               block_1[63:32] <= mix_out;
               else
                 begin                     // Shift_1
                   if(rnd_cnt[0] == 1'b0)  // for rounds 0,2...12
                     {block_1[95:88],block_1[119:112],block_1[15:8],block_1[39:32]} <= sub_out;
                   else                    // for rounds 1,3..13
                     {block_2[95:88],block_2[119:112],block_2[15:8],block_2[39:32]} <= sub_out;
                 end
             end

         4:  begin
               if(rnd_cnt == 4'd0)         // data_out fourth column
                 block_1[31:0] <= mix_out;
               else
                 begin                     // Shift_2
                   if(rnd_cnt[0] == 1'b1)  // for rounds 0,2...12
                     {block_1[63:56],block_1[87:80],block_1[111:104],block_1[7:0]} <= sub_out;
                   else                    // for rounds 1,3..13
                     {block_2[63:56],block_2[87:80],block_2[111:104],block_2[7:0]} <= sub_out;
                 end
             end

       endcase

  end

endmodule
```

C.13 S-box Module

```
module sbox(
        in,
        out);
input   [7:0]   in;
output  [7:0]   out;

wire [7:0] first_matrix_out,first_matrix_in,last_matrix_out_enc, last_matrix_out_dec;
wire [3:0] p,q,p2,q2,sumpq,sump2q2,inv_sump2q2,p_new,q_new,mulpq,q2B;
wire [7:0]  first_matrix_out_L;
wire [3:0]  p_new_L,q_new_L;

// GF(256) to GF(16) transformation
assign first_matrix_in[7:0] = in[7:0];
assign first_matrix_out[7:0] = GF256_TO_GF16(first_matrix_in[7:0]);

assign p[3:0] = first_matrix_out_L[3:0];
assign q[3:0] = first_matrix_out_L[7:4];
assign p2[3:0] = SQUARE(p[3:0]);
assign q2[3:0] = SQUARE(q[3:0]);
//p+q
assign sumpq[3:0] = p[3:0] ^ q[3:0];
//p*q
assign mulpq[3:0] = MUL(p[3:0],q[3:0]);
//q2B calculation
assign q2B[0]=q2[1]^q2[2]^q2[3];
assign q2B[1]=q2[0]^q2[1];
assign q2B[2]=q2[0]^q2[1]^q2[2];
assign q2B[3]=q2[0]^q2[1]^q2[2]^q2[3];
//p2+p*q+q2B
assign sump2q2[3:0] = q2B[3:0] ^ mulpq[3:0] ^ p2[3:0];
// inverse p2+pq+q2B
assign inv_sump2q2[3:0] = INVERSE(sump2q2[3:0]);
// results
assign p_new[3:0] = MUL(sumpq[3:0],inv_sump2q2[3:0]);
assign q_new[3:0] = MUL(q[3:0],inv_sump2q2[3:0]);

assign {p_new_L[3:0],q_new_L[3:0]} = {p_new[3:0],q_new[3:0]};

// GF(16) to GF(256) transformation
assign last_matrix_out_dec[7:0] = GF16_TO_GF256(p_new_L[3:0],q_new_L[3:0]);
assign last_matrix_out_enc[7:0] = AFFINE(last_matrix_out_dec[7:0]);
assign out[7:0] = last_matrix_out_enc[7:0];
```

```
/*****************************************************************************/
// Functions
/*****************************************************************************/

// convert GF(256) to GF(16)
function [7:0] GF256_TO_GF16;
input [7:0] data;
reg a,b,c;
begin
        a = data[1]^data[7];
        b = data[5]^data[7];
        c = data[4]^data[6];
        GF256_TO_GF16[0] = c^data[0]^data[5];
        GF256_TO_GF16[1] = data[1]^data[2];
        GF256_TO_GF16[2] = a;
        GF256_TO_GF16[3] = data[2]^data[4];
        GF256_TO_GF16[4] = c^data[5];
        GF256_TO_GF16[5] = a^c;
        GF256_TO_GF16[6] = b^data[2]^data[3];
        GF256_TO_GF16[7] = b;
end
endfunction

// squre
function [3:0] SQUARE;
input [3:0] data;
begin
        SQUARE[0] = data[0]^data[2];
        SQUARE[1] = data[2];
        SQUARE[2] = data[1]^data[3];
        SQUARE[3] = data[3];
end
endfunction

// inverse
function [3:0] INVERSE;
input [3:0] data;
reg a;
begin
        a=data[1]^data[2]^data[3]^(data[1]&data[2]&data[3]);
        INVERSE[0]=a^data[0]^(data[0]&data[2])^(data[1]&data[2])^(data[0]&data[1]&data[2]);
        INVERSE[1]=(data[0]&data[1])^(data[0]&data[2])^(data[1]&data[2])^data[3]^
                (data[1]&data[3])^(data[0]&data[1]&data[3]);
        INVERSE[2]=(data[0]&data[1])^data[2]^(data[0]&data[2])^data[3]^
                (data[0]&data[3])^(data[0]&data[2]&data[3]);
        INVERSE[3]=a^(data[0]&data[3])^(data[1]&data[3])^(data[2]&data[3]);
end
endfunction

// multiply
function [3:0] MUL;
input [3:0] d1,d2;
reg a,b;
begin
        a=d1[0]^d1[3];
        b=d1[2]^d1[3];

        MUL[0]=(d1[0]&d2[0])^(d1[3]&d2[1])^(d1[2]&d2[2])^(d1[1]&d2[3]);
        MUL[1]=(d1[1]&d2[0])^(a&d2[1])^(b&d2[2])^((d1[1]^d1[2])&d2[3]);
        MUL[2]=(d1[2]&d2[0])^(d1[1]&d2[1])^(a&d2[2])^(b&d2[3]);
        MUL[3]=(d1[3]&d2[0])^(d1[2]&d2[1])^(d1[1]&d2[2])^(a&d2[3]);
end
endfunction
```

```
// GF16 to GF256 transform
function [7:0] GF16_TO_GF256;
input [3:0] p,q;
reg a,b;
begin
        a=p[1]^q[3];
        b=q[0]^q[1];

        GF16_TO_GF256[0]=p[0]^q[0];
        GF16_TO_GF256[1]=b^q[3];
        GF16_TO_GF256[2]=a^b;
        GF16_TO_GF256[3]=b^p[1]^q[2];
        GF16_TO_GF256[4]=a^b^p[3];
        GF16_TO_GF256[5]=b^p[2];
        GF16_TO_GF256[6]=a^p[2]^p[3]^q[0];
        GF16_TO_GF256[7]=b^p[2]^q[3];
end
endfunction

// affine transformation
function [7:0] AFFINE;
input [7:0] data;
begin
        //affine trasformation
        AFFINE[0]=(!data[0])^data[4]^data[5]^data[6]^data[7];
        AFFINE[1]=(!data[0])^data[1]^data[5]^data[6]^data[7];
        AFFINE[2]=data[0]^data[1]^data[2]^data[6]^data[7];
        AFFINE[3]=data[0]^data[1]^data[2]^data[3]^data[7];
        AFFINE[4]=data[0]^data[1]^data[2]^data[3]^data[4];
        AFFINE[5]=(!data[1])^data[2]^data[3]^data[4]^data[5];
        AFFINE[6]=(!data[2])^data[3]^data[4]^data[5]^data[6];
        AFFINE[7]=data[3]^data[4]^data[5]^data[6]^data[7];
end
endfunction

// inverse affine transformation
function [7:0] INV_AFFINE;
input [7:0] data;
reg a,b,c,d;
begin
        a=data[0]^data[5];
        b=data[1]^data[4];
        c=data[2]^data[7];
        d=data[3]^data[6];
        INV_AFFINE[0]=(!data[5])^c;
        INV_AFFINE[1]=data[0]^d;
        INV_AFFINE[2]=(!data[7])^b;
        INV_AFFINE[3]=data[2]^a;
        INV_AFFINE[4]=data[1]^d;
        INV_AFFINE[5]=data[4]^c;
        INV_AFFINE[6]=data[3]^a;
        INV_AFFINE[7]=data[6]^b;
end
endfunction
endmodule

// based on https://github.com/freecores/aes_highthroughput_lowarea/blob/master/verilog/rtl/sbox.v
```

Appendix D - AES-256 code for decryption

D.1 Top Module

```
`timescale 1ns / 1ps

// ---------- ADVANCED ENCRYPTION STANDARD --- AES-256 --- DECRYPTION ----------

module aes256_dec  (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,
                    ready
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global async negative edge reset
input               start;              // start pulse
input       [255:0] key_in;             // 256-bit key
input       [127:0] data_in;            // 128-bit block size of ciphertext
output wire [127:0] data_out;           // 128-bit plaintext and also as storage block
output wire         ready;              // output ready


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [3:0]       rnd_cnt;            // round counter [0 to 14]
wire    [2:0]       step;               // step counter [0 to 4]
wire    [31:0]      mix_out;            // processed column after inv mix operation
wire    [31:0]      sub_out;            // processed column after inv substitution bytes operation
wire    [3:0]        key_exp_rnd;        // round for key expansion
wire    [127:0]     block_2;            // storage block for inv shift rows operation
wire    [31:0]      rnd_key;            // round key for add round key operation
wire    [31:0]      k3_next,k7_next_rot;// word 3 and word 7 used for key expansion
wire                 key_ready;          // key is expanded


// ---------- MODULES INSTANTIATION ----------

// ---------- ROUND AND STEP COUNTERS ----------

counters_dec    counters_dec_u (
                                 clk,
                                reset_n,
                                start,
                                 key_ready,
                                rnd_cnt,    // output
                                step,       // output
                                ready       // output
                                );
```

```
// ---------- INV MIX COLUMNS UNIT ----------

inv_mix_columns inv_mix_columns_u(
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                data_in,
                                data_out,
                                block_2,
                                sub_out,
                                rnd_key,
                                mix_out     // output
                                );

// ---------- INV SUB BYTES UNIT ----------

inv_sub_bytes   inv_sub_bytes_u(
                                clk,
                                reset_n,
                                key_exp_rnd,
                                step,
                                k3_next,k7_next_rot,
                                mix_out,
                                sub_out      // output
                                );

// ---------- INV SHIFT ROWS UNIT ----------

inv_shift_rows  inv_shift_rows_u(
                                clk,
                                reset_n,
                                rnd_cnt,
                                step,
                                sub_out,
                                mix_out,
                                data_out,   // output
                                block_2     // output
                                );

// ---------- KEY EXPANSION UNIT ----------

key_exp_dec     key_exp_dec_u  (
                                clk,
                                reset_n,
                                 rnd_cnt,
                                step,
                                key_in,
                                sub_out,
                                 key_exp_rnd,// output
                                k3_next,k7_next_rot, // output
                                 key_ready,  // output
                                rnd_key     // output
                                );


endmodule
```

D.2 Counters Module

```
`timescale 1ns / 1ps

// ---------- ROUND AND STEP COUNTERS for DEC ----------

module counters_dec (
                    clk,
                    reset_n,
                    start,
                     key_ready,
                    rnd_cnt,            // output
                    step,               // output
                    ready               // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input               start;              // start decryption
input                key_ready;          // key is expanded
output reg  [3:0]   rnd_cnt;            // round counter [0 to 14]
output reg  [2:0]   step;               // step counter [0 to 4]
output reg          ready;              // deccryption done


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter           max_rnd = 4'd14;    // AES-256 requires 14 rounds (plus round 0)
parameter           max_steps = 4'd4;   // every round consists on 5 steps (0 to 4)

reg [3:0]           current_state, next_state;
parameter           IDLE = 4'd6,
                     K_EXP = 4'd7,
                     K_DONE = 4'd8,
                    S0 = 4'd0,
                    S1 = 4'd1,
                    S2 = 4'd2,
                    S3 = 4'd3,
                    S4 = 4'd4,
                    DONE = 4'd5;


// ---------- STATE MACHINE ----------

always @ (posedge clk or negedge reset_n)
  begin: STATE_MEMORY_DEC

    if (!reset_n)   current_state <= IDLE;
    else            current_state <= next_state;

  end


always @ (current_state or start or key_ready or ready)
  begin: NEXT_STATE_LOGIC_DEC

    case (current_state)
      IDLE :    next_state = (start) ? K_EXP : IDLE;// start step counter with start
      K_EXP :    next_state = (key_ready) ? K_DONE : K_EXP; //to K_DONE when key is expanded
      K_DONE :   next_state = S1;
      S0 :      next_state = (ready) ? DONE : S1;   // to DONE if ready, else continue encryption
      S1 :      next_state = S2;
      S2 :      next_state = S3;
      S3 :      next_state = S4;
      S4 :      next_state = S0;                    // resets after step = 4
      DONE:     next_state = (start) ? K_EXP : DONE;   // it can start from DONE state*/
      default : next_state = IDLE;
    endcase

  end
```

```
always @ (current_state or rnd_cnt)
  begin: OUTPUT_LOGIC_DEC

    case (current_state)
      IDLE :  begin
                step = 3'd0;
                ready = 1'b0;
              end

      K_EXP : begin
                step = 3'd5;
                ready = 1'b0;
              end

      K_DONE :begin
                step = 3'd0;
                ready = 1'b0;
              end

      S0 :    begin
                step = 3'd0;
                if (rnd_cnt == 0)        ready = 1'b1;   // when round counter finishes and step = 0
                else                     ready = 1'b0;
              end

      S1 :    begin
                step = 3'd1;
                ready = 1'b0;
              end

      S2 :    begin
                step = 3'd2;
                ready = 1'b0;
              end

      S3 :    begin
                step = 3'd3;
                ready = 1'b0;
              end

      S4 :    begin
                step = 3'd4;
                ready = 1'b0;
              end

      DONE :  begin
                step = 3'd0;
                ready = 1'b1;       // Decryption DONE
              end

      default : begin
                  step = 3'd0;
                  ready = 1'b0;
                end
    endcase

  end

always @ (posedge clk or negedge reset_n)
  begin: ROUND_COUNTER_DEC

    if (!reset_n)
      begin
        rnd_cnt <= 4'b0;                            // reset of round counter
      end
    else
      if(current_state == S3)                       // round counter only when step = 3
        if (rnd_cnt == max_rnd) rnd_cnt <= 4'b0;    // reset when round = 14
        else                    rnd_cnt <= rnd_cnt + 1;

  end


endmodule
```

D.3 Key Expansion Dec Module

```
`timescale 1ns / 1ps

// ---------- KEY EXPANSION DEC ----------

module key_exp_dec (
                    clk,
                    reset_n,
                    rnd_cnt,
                     step,
                    key_in,
                    sub_out,
                     key_exp_rnd, // output
                    k3_next,k7_next_rot,// output
                     key_ready,          // output
                    rnd_key             // output
                    );


// ---------- MODULE INTERFACE ----------

input              clk;                // global clock
input              reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;           // round counter [0 to 14]
input       [2:0]   step;              // step counter [0 to 4]
input       [255:0] key_in;            // 256-bit key
input       [31:0]  sub_out;           // processed column after inv substitution bytes operation
output reg  [3:0]   key_exp_rnd;       // round for key expansion
output reg  [31:0]  k3_next,k7_next_rot;// word 3 and word 7 rotated used for key expansion
output reg           key_ready;         // key is expanded
output wire [31:0]  rnd_key;           // round key for add round key operation


// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [31:0]      k0_next, k1_next, k2_next;   // words used for first part of key expansion
reg     [31:0]      k4_next, k5_next, k6_next, k7_next;  // words used for second part of key
expansion
reg     [31:0]      k0, k1, k2, k3;              // words used for first part of key expansion
reg     [31:0]      k4, k5, k6, k7;              // words used for second part of key expansion
reg                  state;             // 2 state reg, key expansion is done in 2 cycle clocks
wire    [31:0]      RC;                // round constant for key expansion



// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: KEY_EXPANSION_DEC

    if (!reset_n)                      // reset words registers
      begin
        key_ready <= 1'b0;
        key_exp_rnd <= 4'b0;
        state <= 1'b0;
        k0 <= 31'b0;
        k1 <= 31'b0;
        k2 <= 31'b0;
        k3 <= 31'b0;
        k4 <= 31'b0;
        k5 <= 31'b0;
        k6 <= 31'b0;
        k7 <= 31'b0;
        k0_next <= 31'b0;
        k1_next <= 31'b0;
        k2_next <= 31'b0;
        k3_next <= 31'b0;
        k4_next <= 31'b0;
        k5_next <= 31'b0;
        k6_next <= 31'b0;
        k7_next <= 31'b0;
        k7_next_rot <= 31'b0;
      end
```

```
    else if (step == 5)
      begin             // EXPAND KEY

        if (state == 0)
          begin

            if (key_exp_rnd == 0)
              //{k0,k1,k2,k3,k4,k5,k6,{k7_rot[7:0],k7_rot[31:8]}} <= key_in;
              {k0,k1,k2,k3,k4,k5,k6,k7} <= key_in;

                        // 2,4...14
            if (key_exp_rnd[0] == 0 && key_exp_rnd > 1)
              begin
                k0 <= k0_next;   // update key columns for first part
                k1 <= k1_next;
                k2 <= k2_next;
                k3 <= k3_next;
              end
                        // 3,5...13
            if (key_exp_rnd[0] == 1 && key_exp_rnd > 2)
              begin
                k4 <= k4_next;   // update key columns for second part
                k5 <= k5_next;
                k6 <= k6_next;
                k7 <= k7_next;
              end

            key_exp_rnd <= key_exp_rnd + 1;

          end

        else
          begin
                        // First Part: when j mod Nk = 0, (j = 8,16...56)
            k0_next = k0 ^ sub_out ^ RC; // sub_out comes from k7_next_rot
            k1_next = k0_next ^ k1;
            k2_next = k1_next ^ k2;
            k3_next = k2_next ^ k3;

                        // Second Part: when j mod Nk = 4, (j = 12,20...52)
            k4_next = sub_out ^ k4;             // sub_out from k3_next
            k5_next = k4_next ^ k5;
            k6_next = k5_next ^ k6;
            k7_next = k6_next ^ k7;
            k7_next_rot = (key_exp_rnd < 2) ? {k7[23:0],k7[31:24]} : {k7_next[23:0],k7_next[31:24]};

            if (key_exp_rnd == 15)
              key_ready <= 1'b1;

          end

          state <= ~state;

        end

    else
      begin
        key_ready <= 1'b0;
        key_exp_rnd <= 4'b0;
      end

  end
```

```
// ---------- MODULES INSTANTIATION ----------

// ---------- ROUND CONSTANT UNIT ----------

round_constant_dec round_constant_dec_u (
                                    clk,
                                    reset_n,
                                    key_exp_rnd,
                                     state,
                                    RC          // output
                                    );

// ---------- ROUND KEY UNIT ---------

round_key_dec    round_key_dec_u     (
                                    clk,
                                    reset_n,
                                    rnd_cnt,
                                    step,
                                     key_exp_rnd - 4'b1,  // key_exp_rnd calculating already next key
                                    k0,k1,k2,k3,
                                    k4,k5,k6,k7,
                                    rnd_key     // output
                                    );

endmodule
```

D.4 Round Constant Dec Module

```
`timescale 1ns / 1ps

/*
            Round Constant
 for round  2   4   6   8  10  12  14
 --------------------------------------
    i       1   2   3   4   5   6   7
 --------------------------------------
    RC[i]  01  02  04  08  10  20  40

*/

// ---------- ROUND CONSTANT DEC ----------

module round_constant_dec (
                        clk,
                        reset_n,
                        key_exp_rnd,
                         state,
                        RC               // output
                        );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   key_exp_rnd;        // round for key expansion
input                state;              // 2 state reg, key expansion is done in 2 cycle clocks
output reg  [31:0]  RC;                 // round constant for key expansion


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: ROUND_CONSTANT_DEC

    if (!reset_n) RC <= 32'h0;          // reset RC register
                                        // in rounds 1,3...13 and in 2nd cycle for rounds 2,4...14
    else if (key_exp_rnd[0] != 0 && state)
                                        // RC[i] = RC[i-1] << 1
        RC <= (key_exp_rnd == 1) ? 32'h01000000 : {RC[30:0],1'b0};

  end


endmodule
```

D.5 Round Key Dec Module

```
`timescale 1ns / 1ps

// ---------- ROUND KEY DEC ----------

module round_key_dec (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                     key_exp_rnd,
                    k0,k1,k2,k3,
                    k4,k5,k6,k7,
                    rnd_key             // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14] for decryption
input       [2:0]   step;               // step counter [0 to 4]
input       [3:0]   key_exp_rnd; // round for key expansion
input       [31:0]  k0, k1, k2, k3;     // words used for first part of key expansion
input       [31:0]  k4, k5, k6, k7;     // words used for second part of key expansion
output reg  [31:0]  rnd_key;            // round key for add round key operation


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter           max_rnd = 4'd14;    // AES-256 requires 14 rounds (plus round 0)
reg     [63:0]       key_mem0 [14:0];     // array for round keys for k0, k1, k4, k5
reg     [63:0]       key_mem1 [14:0];     // array for round keys for k2, k3, k6, k7


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: GET_ROUND_KEY

    if (!reset_n)
        rnd_key <= 32'h0;     // reset round_key register

    else
        case (step)
                                         // k0 or k4 for Inv_Mix_0
          0:  rnd_key <= key_mem0[max_rnd - rnd_cnt][63:32];
                                        // k1 or k5 for Inv_Mix_1
          1:  rnd_key <= key_mem0[max_rnd - rnd_cnt][31:0];
                                        // k2 or k6 for Inv_Mix_2
          2:  rnd_key <= key_mem1[max_rnd - rnd_cnt][63:32];
                                        // k3 or k7 for Inv_Mix_3
          3:  rnd_key <= key_mem1[max_rnd - rnd_cnt][31:0];

        endcase

  end
```

```
always @ (posedge clk or negedge reset_n)
  begin: SET_ROUND_KEY

    if (!reset_n)
      begin
        key_mem0[0] <= 64'b0;  // reset round_key register
        key_mem0[1] <= 64'b0;
        key_mem0[2] <= 64'b0;
        key_mem0[3] <= 64'b0;
        key_mem0[4] <= 64'b0;
        key_mem0[5] <= 64'b0;
        key_mem0[6] <= 64'b0;
        key_mem0[7] <= 64'b0;
        key_mem0[8] <= 64'b0;
        key_mem0[9] <= 64'b0;
        key_mem0[10] <= 64'b0;
        key_mem0[11] <= 64'b0;
        key_mem0[12] <= 64'b0;
        key_mem0[13] <= 64'b0;
        key_mem0[14] <= 64'b0;
        key_mem1[0] <= 64'b0;
        key_mem1[1] <= 64'b0;
        key_mem1[2] <= 64'b0;
        key_mem1[3] <= 64'b0;
        key_mem1[4] <= 64'b0;
        key_mem1[5] <= 64'b0;
        key_mem1[6] <= 64'b0;
        key_mem1[7] <= 64'b0;
        key_mem1[8] <= 64'b0;
        key_mem1[9] <= 64'b0;
        key_mem1[10] <= 64'b0;
        key_mem1[11] <= 64'b0;
        key_mem1[12] <= 64'b0;
        key_mem1[13] <= 64'b0;
        key_mem1[14] <= 64'b0;
      end

    else if (step == 3'd5)
                  begin
                    key_mem0[key_exp_rnd][63:0] <= (key_exp_rnd[0] == 0) ? {k0,k1} : {k4,k5};
                    key_mem1[key_exp_rnd][63:0] <= (key_exp_rnd[0] == 0) ? {k2,k3} : {k6,k7};
                  end

  end


endmodule
```

D.6 Inv Mix Columns Module

```
`timescale 1ns / 1ps

// ---------- MIX COLUMNS ----------

module inv_mix_columns (
                        clk,
                        reset_n,
                        rnd_cnt,
                        step,
                        data_in,
                        block_1,
                        block_2,
                        sub_out,
                        rnd_key,
                        mix_out         // output
                        );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [127:0] data_in;            // 128-bit ciphertext
input       [127:0] block_1, block_2;   // storage blocks for inv shift rows operation
input       [31:0]  sub_out;            // processed column after inv sub bytes operation
input       [31:0]  rnd_key;            // round key for add round key operation
output wire [31:0]  mix_out;            // processed column after inv mix operation


// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [3:0]       rnd_cnt_mix;        // round counter used as input for inv mix column unit
reg     [31:0]      mix_in;             // column as input for inv mix operation


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: INV_MIX_COLUMNS

    if (!reset_n)
      begin
        mix_in <= 32'b0;        // reset of mix_in register
        rnd_cnt_mix <= 4'b0;
      end

    else
      begin
        rnd_cnt_mix <= rnd_cnt;         // syncrhonize to use correct round counter along with mix_in
        case (step)
          0:  begin                     // Inv_Mix_0
                mix_in <= (rnd_cnt == 0) ? data_in[127:96]:                          // round 0
                          (rnd_cnt[0] != 1'b0) ? {block_1[127:120],sub_out[23:16],block_1[111:96]}:
// rounds 1,3...13
                                                 {block_2[127:120],sub_out[23:16],block_2[111:96]};
// rounds 2,4...14
              end                       // sub_out[23:16] is the last elemtent for first column
                                        // already considering inv shift rows after inv sub bytes

          1:  begin                     // Inv_Mix_1
                mix_in <= (rnd_cnt == 0) ? data_in[95:64]:          // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[95:64]:    // rounds 1,3...13
                                                 block_2[95:64];    // rounds 2,4...14
              end
```

```
          2:  begin                     // Inv_Mix_2
                mix_in <= (rnd_cnt == 0) ? data_in[63:32]:          // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[63:32]:    // rounds 1,3...13
                                                 block_2[63:32];    // rounds 2,4...14
              end

          3:  begin                     // Inv_Mix_3
                mix_in <= (rnd_cnt == 0) ? data_in[31:0]:           // round 0
                          (rnd_cnt[0] != 1'b0) ? block_1[31:0]:     // rounds 1,3...13
                                                 block_2[31:0];     // rounds 2,4...14
              end


        endcase
      end
  end


// ---------- INV MIX OPERATION BY WORD UNIT ----------

inv_mix_w   inv_mix_w_u (
                        .round      (rnd_cnt_mix),      // round counter as condition to inv mix
column operation
                        .round_key  (rnd_key),          // round key for add round key operation
                        .in         (mix_in),           // column as input for inv mix operation
                        .out        (mix_out)           // processed column after inv mix operation
                        );


endmodule
```

D.7 Inv Mix Word Module

```
// ---------- MIX COLUMNS FOR DEC WORD ----------

module inv_mix_w    ( round, round_key, in, out );


// ---------- MODULE INTERFACE ----------

input       [3:0]   round;
input       [31:0]  round_key;
input       [31:0]  in;
output      [31:0]  out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       byte0_in, byte1_in, byte2_in, byte3_in;
wire    [7:0]       byte0_out, byte1_out, byte2_out, byte3_out;


// ---------- MODULE IMPLEMENTATION ----------

assign byte0_in[7:0] = in[31:24] ^ round_key[31:24];
assign byte1_in[7:0] = in[23:16] ^ round_key[23:16];
assign byte2_in[7:0] = in[15:8] ^ round_key[15:8];
assign byte3_in[7:0] = in[7:0] ^ round_key[7:0];

assign out = (round == 0 || round == 14) ? in ^ round_key : {byte0_out, byte1_out, byte2_out,
byte3_out};


// ---------- MODULES INSTANTIATION ----------

inv_byte_mix inv_byte_mix0_u (.a(byte0_in), .b(byte1_in), .c(byte2_in), .d(byte3_in),
.out(byte0_out));

inv_byte_mix inv_byte_mix1_u (.a(byte1_in), .b(byte2_in), .c(byte3_in), .d(byte0_in),
.out(byte1_out));

inv_byte_mix inv_byte_mix2_u (.a(byte2_in), .b(byte3_in), .c(byte0_in), .d(byte1_in),
.out(byte2_out));

inv_byte_mix inv_byte_mix3_u (.a(byte3_in), .b(byte0_in), .c(byte1_in), .d(byte2_in),
.out(byte3_out));


endmodule
```

D.8 Inv ByteMix Module

```
// ---------- INV BYTE MIX COLUMNS ----------

module inv_byte_mix (   a, b, c, d, out );

// ---------- MODULE INTERFACE ----------

input       [7:0]   a, b, c, d;
output      [7:0]   out;

// ---------- MODULE REGISTERS AND SIGNALS ----------

wire [7:0] mule, mulb, muld, mul9;

// ---------- MODULE IMPLEMENTATION ----------

assign out = mule ^ mulb ^ muld ^ mul9;

// ---------- MODULES INSTANTIATION ----------

MULE        mule_u     (    .in(a), .out(mule)  );

MULB        mulb_u     (    .in(b), .out(mulb)  );

MULD        muld_u     (    .in(c), .out(muld)  );

MUL9        mul9_u     (    .in(d), .out(mul9)  );

endmodule
```

D.9 Multiply by 9 Module

```
// ---------- MULTIPLY BY 9 ----------

module MUL9     ( in, out );

// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;

// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       xt1, xt2, xt3;

// ---------- MODULE IMPLEMENTATION ----------

assign out = xt3 ^ in;

// ---------- MODULES INSTANTIATION ----------

xtimes  xt_u1   ( .in(in), .out(xt1) );
xtimes  xt_u2   ( .in(xt1), .out(xt2) );
xtimes  xt_u3   ( .in(xt2), .out(xt3) );

endmodule
```

D.10 Multiply by D Module

```
// ---------- MULTIPLY BY D ----------

module MULD     ( in, out );


// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       xt1, xt2, xt3;


// ---------- MODULE IMPLEMENTATION ----------

assign out = xt3 ^ xt2 ^ in;


// ---------- MODULES INSTANTIATION ----------

xtimes  xt_u1   ( .in(in), .out(xt1) );
xtimes  xt_u2   ( .in(xt1), .out(xt2) );
xtimes  xt_u3   ( .in(xt2), .out(xt3) );


endmodule
```

D.11 Multiply by B Module

```
// ---------- MULTIPLY BY B ----------

module MULB     ( in, out );


// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       xt1, xt2, xt3;


// ---------- MODULE IMPLEMENTATION ----------

assign out = xt3 ^ xt1 ^ in;


// ---------- MODULES INSTANTIATION ----------

xtimes  xt_u1   ( .in(in), .out(xt1) );
xtimes  xt_u2   ( .in(xt1), .out(xt2) );
xtimes  xt_u3   ( .in(xt2), .out(xt3) );


endmodule
```

D.12 Multiply by E Module

```
// ---------- MULTIPLY BY E ----------

/*
   e = 1110 = x3 + x2 + x, v = 1011 0010 = x7 + x5 + x4 + x
   v*e(mod P) = x(x(x(x7 + x5 + x4 + x)modP)modP)modP + x(x(x7 + x5 + x4 + x)modP)modP + x(x7 + x5 +
x4 + x)modP)modP)modP
*/

module MULE     ( in, out );


// ---------- MODULE INTERFACE ----------

input       [7:0]       in;
output      [7:0]       out;


// ---------- MODULE REGISTERS AND SIGNALS ----------

wire    [7:0]       xt1, xt2, xt3;


// ---------- MODULE IMPLEMENTATION ----------

assign out = xt3 ^ xt2 ^ xt1;


// ---------- MODULES INSTANTIATION ----------

xtimes  xt_u1    ( .in(in), .out(xt1) );
xtimes  xt_u2    ( .in(xt1), .out(xt2) );
xtimes  xt_u3    ( .in(xt2), .out(xt3) );
```

D.13 Inv SubBytes Module

```
`timescale 1ns / 1ps

// ---------- INV SUB BYTES ----------

module inv_sub_bytes (
                    clk,
                    reset_n,
                    key_exp_rnd,
                    step,
                    k3_next,k7_next_rot,
                    mix_out,
                    sub_out              // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative active reset
input       [3:0]   key_exp_rnd;        // round to calculate for key expansion
input       [2:0]   step;               // step counter [0 to 4]
input       [31:0]  k3_next, k7_next_rot;// word 3 and word 7 rotated used for key expansion
input       [31:0]  mix_out;            // processed column after inv mix operation
output wire [31:0]  sub_out;            // processed column after inv sub bytes or sub bytes operation

// ---------- MODULE REGISTERS AND SIGNALS ----------

reg     [31:0]      sub_in;             // column as input for S-Box
reg                  ende;               // set encrpytion or decryption mode for S-Box


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: INV_SUB_BYTES

    if (!reset_n)
       begin                             // reset sub_in and ende register
         sub_in <= 32'b0;
         ende <= 1'b0;
       end

    else if (step == 5)                 // Key expansion
      begin                             // for round 2,4...14 : 3,5...13
        sub_in <= (key_exp_rnd[0] == 1) ? {k7_next_rot} : k3_next[31:0]; // rounds 1 to 13
        //Rotate to use as input for S-Box when j mod Nk = 0, j = 8,16...56
        ende <= 1'b0;                    // encryption mode for key expansion
      end
    else
      begin                              // Inv_Sub_0, Inv_Sub_1, Inv_Sub_2, Inv_Sub_3
        ende <= 1'b1;                    // decryption mode for inv substitution bytes operation
        sub_in <= mix_out;              // steps 1,2,3 and 4
      end

  end
```

```
// ---------- S-BOX UNITS ----------

sbox          sbox0_u       (
                             sub_in[31:24],
                              ende,
                             sub_out[31:24]
                             );

sbox          sbox1_u       (
                             sub_in[23:16],
                              ende,
                             sub_out[23:16]
                             );

sbox          sbox2_u       (
                             sub_in[15:8],
                              ende,
                             sub_out[15:8]
                             );

sbox          sbox3_u       (
                             sub_in[7:0],
                              ende,
                             sub_out[7:0]
                             );

endmodule
```

D.14 Inv ShiftRows Module

```
`timescale 1ns / 1ps

/*
            Initial Matrix
    127:120 |  95:88  |  63:56  |  31:24
    119:112 |  87:80  |  55:48  |  23:16
    111:104 |  79:72  |  47:40  |  15:8
    103:96  |  71:64  |  39:32  |   7:0

            After Shift Rows
    127:120 |  95:88  |  63:56  |  31:24
     23:16  | 119:112 |  87:80  |  55:48
     47:40  |  15:8   | 111:104 |  79:72
     71:64  |  39:32  |   7:0   | 103:96

*/

// ---------- INV SHIFT ROWS and OUTPUT ----------

module inv_shift_rows (
                    clk,
                    reset_n,
                    rnd_cnt,
                    step,
                    sub_out,
                    mix_out,
                    block_1,            // output
                    block_2             // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk;                // global clock
input               reset_n;            // global negative edge reset
input       [3:0]   rnd_cnt;            // round counter [0 to 14]
input       [2:0]   step;               // step counter [0 to 4]
input       [31:0]  sub_out;            // processed column after inv substitution bytes operation
input       [31:0]  mix_out;            // processed column after inv mix operation
output reg  [127:0] block_1;            // storage block used after first inv shift rows operation and
to store final output(data_out)
output reg  [127:0] block_2;            // storage block used after second inv shift rows operation


// ---------- MODULE IMPLEMENTATION ----------

always @ (posedge clk or negedge reset_n)
  begin: INV_SHIFT_ROWS

    if (!reset_n)                       // reset blocks registers
      begin
        block_1 <= 128'b0;
        block_2 <= 128'b0;
      end

    else
        case (step)

          0:  begin                         // Inv_Shift_3
                if(rnd_cnt[0] != 1'b0)      // for rounds 1,3...13
                  {block_1[31:24],block_1[119:112],block_1[79:72],block_1[39:32]} <= sub_out;
                else                        // for rounds 2,4...14
                  {block_2[31:24],block_2[119:112],block_2[79:72],block_2[39:32]} <= sub_out;
              end
                                            // data_out first column
          1:  if(rnd_cnt == 4'd14) block_1[127:96] <= mix_out;
```

```
    else
        case (step)

          0:  begin                         // Inv_Shift_3
                if(rnd_cnt[0] != 1'b0)      // for rounds 1,3...13
                  {block_1[31:24],block_1[119:112],block_1[79:72],block_1[39:32]} <= sub_out;
                else                        // for rounds 2,4...14
                  {block_2[31:24],block_2[119:112],block_2[79:72],block_2[39:32]} <= sub_out;
              end
                                            // data_out first column
          1:  if(rnd_cnt == 4'd14) block_1[127:96] <= mix_out;

          2:  begin
                if(rnd_cnt == 4'd14)        // data_out second column
                  block_1[95:64] <= mix_out;
                else
                  begin                     // Inv_Shift_0
                    if(rnd_cnt[0] == 1'b0)  // for rounds 0,2...12
                      {block_1[127:120],block_1[87:80],block_1[47:40],block_1[7:0]} <= sub_out;
                    else                    // for rounds 1,3..13
                      {block_2[127:120],block_2[87:80],block_2[47:40],block_2[7:0]} <= sub_out;
                  end
              end

          3:  begin
                if(rnd_cnt == 4'd14)        // data_out third column
                block_1[63:32] <= mix_out;
                else
                  begin                     // Inv_Shift_1
                    if(rnd_cnt[0] == 1'b0)  // for rounds 0,2...12
                      {block_1[95:88],block_1[55:48],block_1[15:8],block_1[103:96]} <= sub_out;
                    else                    // for rounds 1,3..13
                      {block_2[95:88],block_2[55:48],block_2[15:8],block_2[103:96]} <= sub_out;
                  end
              end

          4:  begin
                if(rnd_cnt == 4'd0)         // data_out fourth column
                  block_1[31:0] <= mix_out;
                else
                  begin                     // Inv_Shift_2
                    if(rnd_cnt[0] == 1'b1)  // for rounds 0,2...12
                      {block_1[63:56],block_1[23:16],block_1[111:104],block_1[71:64]} <= sub_out;
                    else                    // for rounds 1,3..13
                      {block_2[63:56],block_2[23:16],block_2[111:104],block_2[71:64]} <= sub_out;
                  end
              end

        endcase

  end


endmodule
```

D.15 S-box dec Module

```
module sbox_dec(
        in,
        ende,
        out);

input   [7:0]   in;
input           ende;  //0: encryption;  1: decryption
output  [7:0]   out;

wire [7:0] first_matrix_out,first_matrix_in,last_matrix_out_enc,last_matrix_out_dec;
wire [3:0] p,q,p2,q2,sumpq,sump2q2,inv_sump2q2,p_new,q_new,mulpq,q2B;
wire [7:0]  first_matrix_out_L;
wire [3:0]  p_new_L,q_new_L;

// GF(256) to GF(16) transformation
assign first_matrix_in[7:0] = ende ? INV_AFFINE(in[7:0]): in[7:0];
assign first_matrix_out[7:0] = GF256_TO_GF16(first_matrix_in[7:0]);

/*
// pipeline 1
always @ (posedge clk or posedge reset)
begin
        if (reset)
                first_matrix_out_L[7:0] <= 8'b0;
        else if (enable)
                first_matrix_out_L[7:0] <= first_matrix_out[7:0];
end
*/
assign first_matrix_out_L[7:0] = first_matrix_out[7:0];
assign p[3:0] = first_matrix_out_L[3:0];
assign q[3:0] = first_matrix_out_L[7:4];
assign p2[3:0] = SQUARE(p[3:0]);
assign q2[3:0] = SQUARE(q[3:0]);
//p+q
assign sumpq[3:0] = p[3:0] ^ q[3:0];
//p*q
assign mulpq[3:0] = MUL(p[3:0],q[3:0]);
//q2B calculation
assign q2B[0]=q2[1]^q2[2]^q2[3];
assign q2B[1]=q2[0]^q2[1];
assign q2B[2]=q2[0]^q2[1]^q2[2];
assign q2B[3]=q2[0]^q2[1]^q2[2]^q2[3];
//p2+p*q+q2B
assign sump2q2[3:0] = q2B[3:0] ^ mulpq[3:0] ^ p2[3:0];
// inverse p2+pq+q2B
assign inv_sump2q2[3:0] = INVERSE(sump2q2[3:0]);
// results
assign p_new[3:0] = MUL(sumpq[3:0],inv_sump2q2[3:0]);
assign q_new[3:0] = MUL(q[3:0],inv_sump2q2[3:0]);

/*
// pipeline 2
always @ (posedge clk or posedge reset)
begin
        if (reset)
                {p_new_L[3:0],q_new_L[3:0]} <= 8'b0;
        else if (enable)
                {p_new_L[3:0],q_new_L[3:0]} <= {p_new[3:0],q_new[3:0]};
end
*/
assign {p_new_L[3:0],q_new_L[3:0]} = {p_new[3:0],q_new[3:0]};

// GF(16) to GF(256) transformation
assign last_matrix_out_dec[7:0] = GF16_TO_GF256(p_new_L[3:0],q_new_L[3:0]);
assign last_matrix_out_enc[7:0] = AFFINE(last_matrix_out_dec[7:0]);
//assign en_dout[7:0] = last_matrix_out_enc[7:0];
//assign de_dout[7:0] = last_matrix_out_dec[7:0];
assign out[7:0] = ende ?  last_matrix_out_dec[7:0] : last_matrix_out_enc[7:0];
```

```
/****************************************************************************/
// Functions
/****************************************************************************/

// convert GF(256) to GF(16)
function [7:0] GF256_TO_GF16;
input [7:0] data;
reg a,b,c;
begin
        a = data[1]^data[7];
        b = data[5]^data[7];
        c = data[4]^data[6];
        GF256_TO_GF16[0] = c^data[0]^data[5];
        GF256_TO_GF16[1] = data[1]^data[2];
        GF256_TO_GF16[2] = a;
        GF256_TO_GF16[3] = data[2]^data[4];
        GF256_TO_GF16[4] = c^data[5];
        GF256_TO_GF16[5] = a^c;
        GF256_TO_GF16[6] = b^data[2]^data[3];
        GF256_TO_GF16[7] = b;
end
endfunction

// squre
function [3:0] SQUARE;
input [3:0] data;
begin
        SQUARE[0] = data[0]^data[2];
        SQUARE[1] = data[2];
        SQUARE[2] = data[1]^data[3];
        SQUARE[3] = data[3];
end
endfunction

// inverse
function [3:0] INVERSE;
input [3:0] data;
reg a;
begin
        a=data[1]^data[2]^data[3]^(data[1]&data[2]&data[3]);
        INVERSE[0]=a^data[0]^(data[0]&data[2])^(data[1]&data[2])^(data[0]&data[1]&data[2]);
        INVERSE[1]=(data[0]&data[1])^(data[0]&data[2])^(data[1]&data[2])^data[3]^
                (data[1]&data[3])^(data[0]&data[1]&data[3]);
        INVERSE[2]=(data[0]&data[1])^data[2]^(data[0]&data[2])^data[3]^
                (data[0]&data[3])^(data[0]&data[2]&data[3]);
        INVERSE[3]=a^(data[0]&data[3])^(data[1]&data[3])^(data[2]&data[3]);
end
endfunction

// multiply
function [3:0] MUL;
input [3:0] d1,d2;
reg a,b;
begin
        a=d1[0]^d1[3];
        b=d1[2]^d1[3];

        MUL[0]=(d1[0]&d2[0])^(d1[3]&d2[1])^(d1[2]&d2[2])^(d1[1]&d2[3]);
        MUL[1]=(d1[1]&d2[0])^(a&d2[1])^(b&d2[2])^((d1[1]^d1[2])&d2[3]);
        MUL[2]=(d1[2]&d2[0])^(d1[1]&d2[1])^(a&d2[2])^(b&d2[3]);
        MUL[3]=(d1[3]&d2[0])^(d1[2]&d2[1])^(d1[1]&d2[2])^(a&d2[3]);
end
endfunction
```

```
// GF16 to GF256 transform
function [7:0] GF16_TO_GF256;
input [3:0] p,q;
reg a,b;
begin
        a=p[1]^q[3];
        b=q[0]^q[1];

        GF16_TO_GF256[0]=p[0]^q[0];
        GF16_TO_GF256[1]=b^q[3];
        GF16_TO_GF256[2]=a^b;
        GF16_TO_GF256[3]=b^p[1]^q[2];
        GF16_TO_GF256[4]=a^b^p[3];
        GF16_TO_GF256[5]=b^p[2];
        GF16_TO_GF256[6]=a^p[2]^p[3]^q[0];
        GF16_TO_GF256[7]=b^p[2]^q[3];
end
endfunction

// affine transformation
function [7:0] AFFINE;
input [7:0] data;
begin
        //affine trasformation
        AFFINE[0]=(!data[0])^data[4]^data[5]^data[6]^data[7];
        AFFINE[1]=(!data[0])^data[1]^data[5]^data[6]^data[7];
        AFFINE[2]=data[0]^data[1]^data[2]^data[6]^data[7];
        AFFINE[3]=data[0]^data[1]^data[2]^data[3]^data[7];
        AFFINE[4]=data[0]^data[1]^data[2]^data[3]^data[4];
        AFFINE[5]=(!data[1])^data[2]^data[3]^data[4]^data[5];
        AFFINE[6]=(!data[2])^data[3]^data[4]^data[5]^data[6];
        AFFINE[7]=data[3]^data[4]^data[5]^data[6]^data[7];
end
endfunction

// inverse affine transformation
function [7:0] INV_AFFINE;
input [7:0] data;
reg a,b,c,d;
begin
        a=data[0]^data[5];
        b=data[1]^data[4];
        c=data[2]^data[7];
        d=data[3]^data[6];
        INV_AFFINE[0]=(!data[5])^c;
        INV_AFFINE[1]=data[0]^d;
        INV_AFFINE[2]=(!data[7])^b;
        INV_AFFINE[3]=data[2]^a;
        INV_AFFINE[4]=data[1]^d;
        INV_AFFINE[5]=data[4]^c;
        INV_AFFINE[6]=data[3]^a;
        INV_AFFINE[7]=data[6]^b;
end
endfunction
endmodule

// based on https://github.com/freecores/aes_highthroughput_lowarea/blob/master/verilog/rtl/sbox.v
```

Appendix E – Test bench code and results

Verilog code is provided in this section as test benches to validate the cipher encryption and decryption functions following the procedures specified by The Advanced Encryption Standard Algorithm Validation Suite (AESAVS) in addition with vectors provided by NIST [87]. Besides, simulation results for the first 10 and last 10 cases are presented as well. The Known Answer Tests (KAT) and the Monte Carlo Test are the tests computed in the simulation.

E.1 KAT test for encryption

E.1.1 Verilog Code

```
`timescale 1ns / 10ps

// ---------- TEST BENCH --- AES-256 --- ENCRYPTION ----------

module aes256_enc_tb();


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter       tests = 405;    // number of tests

reg [255:0]     key_vectors         [tests-1:0];    // input key test vectors
reg [127:0]     plaintext_vectors   [tests-1:0];    // input data test vectors
reg [127:0]     ciphertext_vectors  [tests-1:0];    // expected outputs test vectors

reg             error_det;      // error detector
integer         j;              // loop integer
integer         FILE;           // file to store records of validation

// ---------- UUT INTERFACE REGISTERS AND SIGNALS ----------

reg             clk;            // global clock
reg             reset_n;        // global async negative edge reset
reg             start;          // start pulse
reg  [255:0]    key_in;         // 256-bit key
reg  [127:0]    data_in;        // 128-bit block size of plaintext
wire [127:0]    data_out;       // 128-bit ciphertext
wire            ready;          // output ready


// ---------- UNIT UNDERT TEST ----------

aes256_enc  dut    (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,       // output
                    ready           // output
                    );


// ---------- EVENTS ----------

event reset_enable;     // async negative edge reset pulse
event reset_done;       // reset is done
event finish_sim;       // finish simulation


// ---------- AESAVS Known Answer Tests (KAT) VECTORS ----------
// https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers

initial
  begin
                        // read KAT vectors from files
    $readmemh("key_vectors.txt",        key_vectors);           // 256bit keys
    $readmemh("plaintext_vectors.txt",  plaintext_vectors);     // 128bit plaintexts
    $readmemh("ciphertext_vectors.txt", ciphertext_vectors);    // 128bit expected ciphertexts

    FILE = $fopen("AES256_enc_validation.txt");                  // export results to file

  end
```

```
// ---------- GLOBAL CLOCK GENERATOR ----------

initial     clk = 1'b1;
always          #10 clk = ~clk;


// ---------- DEFAULT INPUT VALUES ----------

initial
  begin
        start = 1'b0;
        key_in = 256'b0;
        data_in = 128'b0;
        error_det = 1'b0;
  end


// ---------- GLOBAL ASYNC NEGATIVE EDGE RESET GENERATOR ----------

always @(reset_enable)
  begin
               reset_n = 1'b0;
        #15    reset_n = 1'b1;
        -> reset_done;
  end


// ---------- MAIN TEST BENCH PROCESS CONTROL ----------

initial
  begin

    -> reset_enable;                // reset pulse
    @(reset_done);                  // wait until reset is done

    for(j=0; j < tests; j=j+1)
      begin

        key_in = key_vectors[j];        // key input value
        data_in = plaintext_vectors[j]; // plaintext input vaule

        start = 1'b1;                  // start pulse
        @ (posedge clk);
        start = 1'b0;
        @ (posedge clk);
        @(ready);                       // wait until encryption ready

        $fdisplay(FILE, "Test :\t%d", j);       // display results
        $fdisplay(FILE, "Key :\t\t\t%64h", key_vectors[j]);
        $fdisplay(FILE, "Plaintext :\t\t%32h", plaintext_vectors[j]);
        $fdisplay(FILE, "Obtained Ciphertext :\t%32h", data_out);
        $fdisplay(FILE, "Expected Ciphertext :\t%32h", ciphertext_vectors[j]);

        if(data_out == ciphertext_vectors[j])   // obtained ciphertext as expected
                $fdisplay(FILE, "Test Result : PASSED\n");
        else
          begin                            // obtained ciphertext different than expected
            $fdisplay(FILE, "Test Result : FAILED at TIME %d\n", $time);
            error_det = 1;
            -> finish_sim;
          end

        if(j == tests-1) -> finish_sim;         // finish simulation after last test

        @(posedge clk);

      end

end
```

```
// ---------- SIMULATION RESULT ----------

always @(finish_sim)
  begin

    $fdisplay(FILE, "###   Simulation DONE   ###");
    $display("###   Simulation DONE   ###");

    if (error_det == 0)         // No error detected during entire simulation
      begin
        $fdisplay(FILE, "Simulation Result : PASSED\n");
        $display("Simulation Result : PASSED\n");
      end
    else                        // Error detected
      begin
        $fdisplay(FILE, "Simulation Result : FAILED\n");
        $display("Simulation Result : FAILED\n");
      end

    repeat(5)@ (posedge clk);
    $finish;

  end

endmodule
```

E.1.2 Test Results

Test : 0
Key :
0000000000000000000000000000000000000000000000000000000000000000
Plaintext : 014730f80ac625fe84f026c60bfd547d
Obtained Ciphertext : 5c9d844ed46f9885085e5d6a4f94c7d7
Expected Ciphertext : 5c9d844ed46f9885085e5d6a4f94c7d7
Test Result : PASSED

Test : 1
Key :
0000000000000000000000000000000000000000000000000000000000000000
Plaintext : 0b24af36193ce4665f2825d7b4749c98
Obtained Ciphertext : a9ff75bd7cf6613d3731c77c3b6d0c04
Expected Ciphertext : a9ff75bd7cf6613d3731c77c3b6d0c04
Test Result : PASSED

Test : 2
Key :
0000000000000000000000000000000000000000000000000000000000000000
Plaintext : 761c1fe41a18acf20d241650611d90f1
Obtained Ciphertext : 623a52fcea5d443e48d9181ab32c7421
Expected Ciphertext : 623a52fcea5d443e48d9181ab32c7421
Test Result : PASSED

Test : 3
Key :
0000000000000000000000000000000000000000000000000000000000000000
Plaintext : 8a560769d605868ad80d819bdba03771
Obtained Ciphertext : 38f2c7ae10612415d27ca190d27da8b4
Expected Ciphertext : 38f2c7ae10612415d27ca190d27da8b4
Test Result : PASSED

Test : 4
Key :
0000000000000000000000000000000000000000000000000000000000000000
Plaintext : 91fbef2d15a97816060bee1feaa49afe
Obtained Ciphertext : 1bc704f1bce135ceb810341b216d7abe
Expected Ciphertext : 1bc704f1bce135ceb810341b216d7abe
Test Result : PASSED

Test : 5
Key :
c47b0294dbbbee0fec4757f22ffeee3587ca4730c3d33b691df38bab076bc558
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 46f2fb342d6f0ab477476fc501242c5f
Expected Ciphertext : 46f2fb342d6f0ab477476fc501242c5f
Test Result : PASSED

Test : 6
Key :
28d46cffa158533194214a91e712fc2b45b518076675affd910edeca5f41ac64
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 4bf3b0a69aeb6657794f2901b1440ad4
Expected Ciphertext : 4bf3b0a69aeb6657794f2901b1440ad4
Test Result : PASSED

Test : 7
Key :
c1cc358b449909a19436cfbb3f852ef8bcb5ed12ac7058325f56e6099aab1a1c
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 352065272169abf9856843927d0674fd
Expected Ciphertext : 352065272169abf9856843927d0674fd
Test Result : PASSED

Test : 8
Key :
984ca75f4ee8d706f46c2d98c0bf4a45f5b00d791c2dfeb191b5ed8e420fd627
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 4307456a9e67813b452e15fa8fffe398
Expected Ciphertext : 4307456a9e67813b452e15fa8fffe398
Test Result : PASSED

Test : 9
Key : b43d08a447ac8609baa-
dae4ff12918b9f68fc1653f1269222f123981ded7a92f
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 4663446607354989477a5c6f0f007ef4
Expected Ciphertext : 4663446607354989477a5c6f0f007ef4
Test Result : PASSED

Test : 395
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe00
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 5b40ff4ec9be536ba23035fa4f06064c
Expected Ciphertext : 5b40ff4ec9be536ba23035fa4f06064c
Test Result : PASSED

Test : 396
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 60eb5af8416b257149372194e8b88749
Expected Ciphertext : 60eb5af8416b257149372194e8b88749
Test Result : PASSED

Test : 397
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff80
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 2f005a8aed8a361c92e440c15520cbd1
Expected Ciphertext : 2f005a8aed8a361c92e440c15520cbd1
Test Result : PASSED

Test : 398
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc0
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 7b03627611678a997717578807a800e2
Expected Ciphertext : 7b03627611678a997717578807a800e2
Test Result : PASSED

Test : 399
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : cf78618f74f6f3696e0a4779b90b5a77
Expected Ciphertext : cf78618f74f6f3696e0a4779b90b5a77
Test Result : PASSED

Test : 400
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff0
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 03720371a04962eaea0a852e69972858
Expected Ciphertext : 03720371a04962eaea0a852e69972858
Test Result : PASSED

Test : 401
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff8
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 1f8a8133aa8ccf70e2bd3285831ca6b7
Expected Ciphertext : 1f8a8133aa8ccf70e2bd3285831ca6b7
Test Result : PASSED

Test : 402
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 27936bd27fb1468fc8b48bc483321725
Expected Ciphertext : 27936bd27fb1468fc8b48bc483321725
Test Result : PASSED

Test : 403
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe

Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : b07d4f3e2cd2ef2eb545980754dfea0f
Expected Ciphertext : b07d4f3e2cd2ef2eb545980754dfea0f
Test Result : PASSED

Test : 404
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Plaintext : 00000000000000000000000000000000
Obtained Ciphertext : 4bf85f1b5d54adbc307b0a048389adcb
Expected Ciphertext : 4bf85f1b5d54adbc307b0a048389adcb
Test Result : PASSED

Simulation DONE
Simulation Result : PASSED

E.2 Monte Carlo Test for encryption

E.2.1 Verilog Code

```
`timescale 1ns / 10ps

// ---------- TEST BENCH --- AES-256 --- ENCRYPTION ----------

module aes256_mct_enc_tb();


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter       key_tests = 100;        // number of key tests
parameter       iterations = 1000;      // number of iterations per key

                // AESAVS Monte Carlo Test Vectors
// https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers
parameter       key_mct_vector =
256'hf9e8389f5b80712e3886cc1fa2d28a3b8c9cd88a2d4a54c6aa86ce0fef944be0;
parameter       plaintext_mct_vector = 128'hb379777f9050e2a818f2940cbbd9aba4;
parameter       ciphertext_mct_vector = 128'hc5d2cb3d5b7ff0e23e308967ee074825;

reg [255:0]     key [key_tests-1:0];    // keys generated in each test
reg [127:0]     pt  [iterations-1:0];   // plaintexts generated in each test
reg [127:0]     ct  [iterations-1:0];   // ciphertext generated in each test

reg             error_det;      // error detector
integer         j;              // key loop integer
integer         i;              // iteration loop integer
integer         FILE;           // file to store records of validation

// ---------- UUT INTERFACE REGISTERS AND SIGNALS ----------

reg             clk;            // global clock
reg             reset_n;        // global async negative edge reset
reg             start;          // start pulse
reg  [255:0]    key_in;         // 256-bit key
reg  [127:0]    data_in;        // 128-bit block size of plaintext
wire [127:0]    data_out;       // 128-bit ciphertext
wire            ready;          // output ready


// ---------- UNIT UNDERT TEST ----------

aes256_enc  dut    (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,       // output
                    ready           // output
                    );


// ---------- EVENTS ----------

event reset_enable;     // async negative edge reset pulse
event reset_done;       // reset is done
event start_pulse;      // start pulse
event mct_done;         // Monte Carlo Test done
```

```
// ---------- GLOBAL CLOCK GENERATOR ----------

initial     clk = 1'b1;
always          #10 clk = ~clk;


// ---------- INITIAL VALUES ----------

initial
  begin
        start = 1'b0;              // initial input values
        key_in = 256'b0;
        data_in = 128'b0;

    FILE = $fopen("AES256_MCT_enc_validation.txt"); // export results to file

    key[0] = key_mct_vector;    // initial test values
    pt[0] = plaintext_mct_vector;
  end


// ---------- GLOBAL ASYNC NEGATIVE EDGE RESET GENERATOR ----------

always @(reset_enable)
  begin
               reset_n = 1'b0;
        #15    reset_n = 1'b1;
        -> reset_done;
  end


// ---------- MAIN TEST BENCH PROCESS CONTROL ----------

initial
  begin

    -> reset_enable;           // reset pulse
    @(reset_done);             // wait until reset is done

    for(j=0; j < key_tests; j=j+1)      // loop to test 100 different keys
      begin

        $fdisplay(FILE, "Test :\t%d", j);               // display to file test number
        $fdisplay(FILE, "Key :\t\t%64h", key[j]);       // display to file key value
        $fdisplay(FILE, "Plaintext :\t%32h", pt[0]);    // display to file data value

        for(i=0; i < iterations; i=i+1) // loop to test 1000 different plaintexts for each key
          begin

            key_in = key[j];            // key input value
            data_in = pt[i];            // plaintext input value

            start = 1'b1;                // start pulse
            @ (posedge clk);
            start = 1'b0;
            @ (posedge clk);
            @(ready)                    // wait until encryption ready

            ct[i] = data_out;           // stores output ciphertext
            pt[i+1] = ct[i];            // obtained ciphertext becomes plaintext for next iteration

            @ (posedge clk);

          end
```

```
                // next key generated from current key and the last 2 cipherkey
            key[j+1] = key[j] ^ {ct[iterations-2],ct[iterations-1]};
                // initial plaintext for next test will be the last cipherkey of current test
            pt[0] = ct[iterations-1];
                // display to file last cipherkey of current test
            $fdisplay(FILE, "Ciphertext :\t%32h\n", ct[iterations-1]);

        end

        -> mct_done;      // Monte Carlo Test done

end


// ---------- SIMULATION RESULTS ----------

always @(mct_done)
  begin
            // display to file results
    $fdisplay(FILE, "###   MONTE CARLO TEST DONE   ###");
    $fdisplay(FILE, "Obtained Ciphertext :\t%32h", ct[iterations-1]);
    $fdisplay(FILE, "Expected Ciphertext :\t%32h\n", ciphertext_mct_vector);
            // display to console results
    $display("###   MONTE CARLO TEST DONE   ###");
    $display("Obtained Ciphertext :\t%32h", ct[iterations-1]);
    $display("Expected Ciphertext :\t%32h\n", ciphertext_mct_vector);

    if(ct[iterations-1] != ciphertext_mct_vector)       // obtained ciphertext different than expected
      begin
        $fdisplay(FILE, "Monte Carlo Test : FAILED");
        $display("Monte Carlo Test : FAILED");
      end
    else
      begin                                             // obtained ciphertext as expected
        $fdisplay(FILE, "Monte Carlo Test : PASSED");
        $display("Monte Carlo Test : PASSED");
      end

    @ (posedge clk);
    $finish;

  end

endmodule
```

E.2.2 Test Results

Test : 0
Key : f9e8389f5b80712e3886cc1fa2d28a3b8c9cd88a2d4a54c6aa86ce0fef944be0
Plaintext : b379777f9050e2a818f2940cbbd9aba4
Ciphertext : 6893ebaf0a1fccc704326529fdfb60db

Test : 1
Key : db9ea5a2284fa17fb63e13bf891c8e42e40f332527559801aeb4ab26126f2b3b
Plaintext : 6893ebaf0a1fccc704326529fdfb60db
Ciphertext : f3c78a5e85e5439bf26d5818718157d6

Test : 2
Key : 7099ed88e82744228a5303ae2ef6c0d017c8b97ba2b0db9a5cd9f33e63ee7ced
Plaintext : f3c78a5e85e5439bf26d5818718157d6
Ciphertext : 2326b958b00b3050697eedb08cc20504

Test : 3
Key : 5e9e65ea96e78dd4fb78ea1184f6ebde34ee002312bbebca35a71e8eef2c79e9
Plaintext : 2326b958b00b3050697eedb08cc20504
Ciphertext : ec4332d5e3cebd3e0f5fc51452f4560d

Test : 4
Key : 33acf1cafc822646dc869e905bd26f9ad8ad32f6f17556f43af8db9abdd82fe4
Plaintext : ec4332d5e3cebd3e0f5fc51452f4560d
Ciphertext : 5da58b5ef2076340d555f861c3449a77

Test : 5
Key : eb0ae85c1b44d5db4729d268f49be2a08508b9a8037235b4efad23fb7e9cb593
Plaintext : 5da58b5ef2076340d555f861c3449a77
Ciphertext : 307d50c18a0b6a08402ff131d72cb7ec

Test : 6
Key : fac93b561a9b6a0e809d71ecdb980afab575e96989795fbcaf82d2caa9b0027f
Plaintext : 307d50c18a0b6a08402ff131d72cb7ec
Ciphertext : 92c34165a2963e77e05e2d6fc2d931d5

Test : 7
Key : a0559e41d58af36174a67246df87541b27b6a80c2bef61cb4fdcffa56b6933aa
Plaintext : 92c34165a2963e77e05e2d6fc2d931d5
Ciphertext : cb33d519a1fdb1d5fbb185c47870c1ed

Test : 8
Key : e48824d6c2251d3a27f38fb543c31fc1ec857d158a12d01eb46d7a611319f247
Plaintext : cb33d519a1fdb1d5fbb185c47870c1ed
Ciphertext : 78fb452f384c8f870e572890588f3728

Test : 9
Key : 7a33440ad7c69d583355c745e5c88c47947e383ab25e5f99ba3a52f14b96c56f
Plaintext : 78fb452f384c8f870e572890588f3728
Ciphertext : 12375e02a8bbc84b00feaab54a66db43

Test : 90
Key : 6d7f0f7584162a1fa4dd6764548f355af22f775429bbc776848a586a75ce6f76
Plaintext : 805be62789549ce6af74966467f41135
Ciphertext : ab6001c6c4c56e8ca393c5fd173505ba

Test : 91
Key : d8ecc39ac1d00c53216f6e64e826a7a9594f7692ed7ea9fa27199d9762fb6acc
Plaintext : ab6001c6c4c56e8ca393c5fd173505ba
Ciphertext : 3ba3673f4f495dd1541d47c22b7921c5

Test : 92
Key : 493108f91caedf714652149a2b2030fe62ec11ada237f42b7304da5549824b09
Plaintext : 3ba3673f4f495dd1541d47c22b7921c5
Ciphertext : b24fe17cdc5c8cfa4260c38691b57bfa

Test : 93
Key : 446af6dd5f58755aeaa0a1226d8c584fd0a3f0d17e6b78d1316419d3d83730f3
Plaintext : b24fe17cdc5c8cfa4260c38691b57bfa
Ciphertext : 86d999a63b96f6c9d9aaf3be6202977b

Test : 94
Key : 65f92d4e1723d5e58aeb350c79df28de567a697745fd8e18e8ceea6dba35a788
Plaintext : 86d999a63b96f6c9d9aaf3be6202977b
Ciphertext : c4712aa733f9737f91e4ed61609e02f1

Test : 95
Key : 915cee6af4ea95623f7122acda5e9040920b43d07604fd67792a070cdaaba579
Plaintext : c4712aa733f9737f91e4ed61609e02f1
Ciphertext : 0e8c1a77b280f4c753682768fd6f3b23

Test : 96
Key : 8e9bb2887fe60d42db4d827f00ba68ff9c8759a7c48409a02a42206427c49e5a
Plaintext : 0e8c1a77b280f4c753682768fd6f3b23
Ciphertext : dccb684d47c480cc1317dcaa451234c0

Test : 97
Key : cbf85a6645469e5df882fe840776b6aa404c31ea8340896c3955fcce62d6aa9a
Plaintext : dccb684d47c480cc1317dcaa451234c0
Ciphertext : 9a2c4f07489c14265e33ac031d02b3d8

Test : 98
Key : 3ea3c33d7439ab3c478c01907f13cda7da607eedcbdc9d4a676650cd7fd41942
Plaintext : 9a2c4f07489c14265e33ac031d02b3d8
Ciphertext : 5c8e622ddbd32ee79c17572e8b3ee61c

Test : 99
Key : 312c5b43263c1af8d1e35c0f24d1004386ee1cc0100fb3adfb7107e3f4eaff5e
Plaintext : 5c8e622ddbd32ee79c17572e8b3ee61c
Ciphertext : c5d2cb3d5b7ff0e23e308967ee074825

MONTE CARLO TEST DONE
Obtained Ciphertext : c5d2cb3d5b7ff0e23e308967ee074825
Expected Ciphertext : c5d2cb3d5b7ff0e23e308967ee074825

Monte Carlo Test : PASSED

E.3 KAT test for decryption

E.3.1 Verilog Code

```
`timescale 1ns / 10ps

// ---------- TEST BENCH --- AES-256 --- DECRYPTION ----------

module aes256_dec_tb();


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter       tests = 405;    // number of tests

reg [255:0]     key_vectors         [tests-1:0];    // input key test vectors
reg [127:0]     ciphertext_vectors   [tests-1:0];    // input data test vectors
reg [127:0]     plaintext_vectors  [tests-1:0];    // expected outputs test vectors

reg             error_det;      // error detector
integer         j;              // loop integer
integer         FILE;           // file to store records of validation

// ---------- UUT INTERFACE REGISTERS AND SIGNALS ----------

reg             clk;            // global clock
reg             reset_n;        // global async negative edge reset
reg             start;          // start pulse
reg  [255:0]    key_in;         // 256-bit key
reg  [127:0]    data_in;        // 128-bit block size of ciphertext
wire [127:0]    data_out;       // 128-bit plaintext
wire            ready;          // output ready


// ---------- UNIT UNDERT TEST ----------

aes256_dec  dut    (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,       // output
                    ready           // output
                    );


// ---------- EVENTS ----------

event reset_enable;     // async negative edge reset pulse
event reset_done;       // reset is done
event start_pulse;      // start pulse
event finish_sim;       // finish simulation


// ---------- AESAVS Known Answer Tests (KAT) VECTORS ----------
// https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers

initial
  begin                 // read KAT vectors from files
    $readmemh("key_vectors.txt",        key_vectors);           // 256bit keys
    $readmemh("ciphertext_vectors.txt", ciphertext_vectors);    // 128bit ciphertexts
    $readmemh("plaintext_vectors.txt",  plaintext_vectors);     // 128bit expected plaintexts
  end
```

```
// ---------- GLOBAL CLOCK GENERATOR ----------

initial         clk = 1'b1;
always          #10 clk = ~clk;


// ---------- DEFAULT INPUT VALUES ----------

initial
  begin
        start = 1'b0;
        key_in = 256'b0;
        data_in = 128'b0;
        error_det = 1'b0;
  end


// ---------- GLOBAL ASYNC NEGATIVE EDGE RESET GENERATOR ----------

always @(reset_enable)
  begin
               reset_n = 1'b0;
        #15    reset_n = 1'b1;
        -> reset_done;
  end


// ---------- MAIN TEST BENCH PROCESS CONTROL ----------

initial
  begin

    -> reset_enable;                // reset pulse
    @(reset_done);                  // wait until reset is done

    FILE = $fopen("AES256_dec_validation.txt");

    for(j=0; j < tests; j=j+1)
      begin

        key_in = key_vectors[j];        // key input value
        data_in = ciphertext_vectors[j]; // ciphertext input vaule

        start = 1'b1;                  // start pulse
        @ (posedge clk);
        start = 1'b0;
        @ (posedge clk);
        @(ready);                       // wait until decryption ready

        $fdisplay(FILE, "Test :\t%d", j);       // display results
        $fdisplay(FILE, "Key :\t\t\t%64h", key_vectors[j]);
        $fdisplay(FILE, "Ciphertext :\t\t%32h", ciphertext_vectors[j]);
        $fdisplay(FILE, "Obtained Plaintext:\t%32h", data_out);
        $fdisplay(FILE, "Expected Plaintext :\t%32h", plaintext_vectors[j]);
        if(data_out == plaintext_vectors[j])   // obtained plaintext as expected
                $fdisplay(FILE, "Test Result : PASSED\n");
        else
          begin                                // obtained plaintext different than expected
            $fdisplay(FILE, "Test Result : FAILED at TIME %d\n", $time);
            error_det = 1;
            -> finish_sim;
          end

        if(j == tests-1) -> finish_sim;         // finish simulation after last last

        @(posedge clk);

      end

end
```

```
// ---------- SIMULATION RESULT ----------

always @(finish_sim)
  begin

    $fdisplay(FILE, "###   Simulation DONE   ###");
    $display("###   Simulation DONE   ###");

    if (error_det == 0)         // No error detected during entire simulation
      begin
        $fdisplay(FILE, "Simulation Result : PASSED\n");
        $display("Simulation Result : PASSED\n");
      end
    else                        // Error detected
      begin
        $fdisplay(FILE, "Simulation Result : FAILED\n");
        $display("Simulation Result : FAILED\n");
      end

    repeat(5)@ (posedge clk);
    $finish;

  end

endmodule
```

E.3.2 Test Results

Test : 0
Key :
0000000000000000000000000000000000000000000000000000000000000000
Ciphertext : 5c9d844ed46f9885085e5d6a4f94c7d7
Obtained Plaintext: 014730f80ac625fe84f026c60bfd547d
Expected Plaintext : 014730f80ac625fe84f026c60bfd547d
Test Result : PASSED

Test : 1
Key :
0000000000000000000000000000000000000000000000000000000000000000
Ciphertext : a9ff75bd7cf6613d3731c77c3b6d0c04
Obtained Plaintext: 0b24af36193ce4665f2825d7b4749c98
Expected Plaintext : 0b24af36193ce4665f2825d7b4749c98
Test Result : PASSED

Test : 2
Key :
0000000000000000000000000000000000000000000000000000000000000000
Ciphertext : 623a52fcea5d443e48d9181ab32c7421
Obtained Plaintext: 761c1fe41a18acf20d241650611d90f1
Expected Plaintext : 761c1fe41a18acf20d241650611d90f1
Test Result : PASSED

Test : 3
Key :
0000000000000000000000000000000000000000000000000000000000000000
Ciphertext : 38f2c7ae10612415d27ca190d27da8b4
Obtained Plaintext: 8a560769d605868ad80d819bdba03771
Expected Plaintext : 8a560769d605868ad80d819bdba03771
Test Result : PASSED

Test : 4
Key :
0000000000000000000000000000000000000000000000000000000000000000
Ciphertext : 1bc704f1bce135ceb810341b216d7abe
Obtained Plaintext: 91fbef2d15a97816060bee1feaa49afe
Expected Plaintext : 91fbef2d15a97816060bee1feaa49afe
Test Result : PASSED

Test : 5
Key :
c47b0294dbbbee0fec4757f22ffeee3587ca4730c3d33b691df38bab076bc558
Ciphertext : 46f2fb342d6f0ab477476fc501242c5f
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 6
Key :
28d46cffa158533194214a91e712fc2b45b518076675affd910edeca5f41ac64
Ciphertext : 4bf3b0a69aeb6657794f2901b1440ad4
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 7
Key :
c1cc358b449909a19436cfbb3f852ef8bcb5ed12ac7058325f56e6099aab1a1c
Ciphertext : 352065272169abf9856843927d0674fd
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 8
Key :
984ca75f4ee8d706f46c2d98c0bf4a45f5b00d791c2dfeb191b5ed8e420fd627
Ciphertext : 4307456a9e67813b452e15fa8fffe398
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 9
Key : b43d08a447ac8609baa-
dae4ff12918b9f68fc1653f1269222f123981ded7a92f
Ciphertext : 4663446607354989477a5c6f0f007ef4
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 395
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe00
Ciphertext : 5b40ff4ec9be536ba23035fa4f06064c
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 396
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff00
Ciphertext : 60eb5af8416b257149372194e8b88749
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 397
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff80
Ciphertext : 2f005a8aed8a361c92e440c15520cbd1
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 398
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc0
Ciphertext : 7b03627611678a997717578807a800e2
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 399
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe0
Ciphertext : cf78618f74f6f3696e0a4779b90b5a77
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 400
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff0
Ciphertext : 03720371a04962eaea0a852e69972858
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 401
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff8
Ciphertext : 1f8a8133aa8ccf70e2bd3285831ca6b7
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 402
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc
Ciphertext : 27936bd27fb1468fc8b48bc483321725
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 403
Key : fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe

Ciphertext : b07d4f3e2cd2ef2eb545980754dfea0f
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Test : 404
Key : ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
Ciphertext : 4bf85f1b5d54adbc307b0a048389adcb
Obtained Plaintext: 00000000000000000000000000000000
Expected Plaintext : 00000000000000000000000000000000
Test Result : PASSED

Simulation DONE
Simulation Result : PASSED

E.4 Monte Carlo Test for decryption

E.4.1 Verilog Code

```
`timescale 1ns / 10ps

// ---------- TEST BENCH --- AES-256 --- DECRYPTION ----------

module aes256_mct_dec_tb();


// ---------- MODULE REGISTERS AND SIGNALS ----------

parameter       key_tests = 100;        // number of key tests
parameter       iterations = 1000;      // number of iterations per key

                // AESAVS Monte Carlo Test Vectors
// https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/block-ciphers
parameter       key_mct_vector =
256'h2b09ba39b834062b9e93f48373b8dd018dedf1e5ba1b8af831ebbacbc92a2643;
parameter       ciphertext_mct_vector = 128'h89649bd0115f30bd878567610223a59d;
parameter       plaintext_mct_vector = 128'he3d3868f578caf34e36445bf14cefc68;

reg [255:0]     key [key_tests-1:0];    // keys generated in each test
reg [127:0]     ct  [iterations-1:0];   // ciphertext generated in each test
reg [127:0]     pt  [iterations-1:0];   // plaintexts generated in each test

reg             error_det;      // error detector
integer         j;              // key loop integer
integer         i;              // iteration loop integer
integer         FILE;           // file to store records of validation

// ---------- UUT INTERFACE REGISTERS AND SIGNALS ----------

reg             clk;            // global clock
reg             reset_n;        // global async negative edge reset
reg             start;          // start pulse
reg  [255:0]    key_in;         // 256-bit key
reg  [127:0]    data_in;        // 128-bit block size of ciphertext
wire [127:0]    data_out;       // 128-bit plaintext
wire            ready;          // output ready


// ---------- UNIT UNDERT TEST ----------

aes256_dec  dut    (
                    clk,
                    reset_n,
                    start,
                    key_in,
                    data_in,
                    data_out,       // output
                    ready           // output
                    );
```

```
// ---------- EVENTS ----------

event reset_enable;     // async negative edge reset pulse
event reset_done;       // reset is done
event start_pulse;      // start pulse
event mct_done;         // Monte Carlo Test done


// ---------- GLOBAL CLOCK GENERATOR ----------

initial     clk = 1'b1;
always  #10 clk = ~clk;


// ---------- INITIAL VALUES ----------

initial
  begin
        start = 1'b0;              // initial input values
        key_in = 256'b0;
        data_in = 128'b0;

        FILE = $fopen("AES256_MCT_dec_validation.txt"); // export results to file

        key[0] = key_mct_vector;    // initial test values
        ct[0] = ciphertext_mct_vector;
  end


// ---------- GLOBAL ASYNC NEGATIVE EDGE RESET GENERATOR ----------

always @(reset_enable)
  begin
               reset_n = 1'b0;
        #15    reset_n = 1'b1;
        -> reset_done;
  end

// ---------- MAIN TEST BENCH PROCESS CONTROL ----------

initial
  begin

    -> reset_enable;            // reset pulse
    @(reset_done);              // wait until reset is done

    for(j=0; j < key_tests; j=j+1)      // loop to test 100 different keys
      begin

        $fdisplay(FILE, "Test :\t%d", j);               // display to file test number
        $fdisplay(FILE, "Key :\t\t%64h", key[j]);       // display to file key value
        $fdisplay(FILE, "Ciphertext :\t%32h", ct[0]);    // display to file data value

        for(i=0; i < iterations; i=i+1) // loop to test 1000 different ciphertexts for each key
          begin

            key_in = key[j];            // key input value
            data_in = ct[i];            // ciphertext input value

            start = 1'b1;                // start pulse
            @ (posedge clk);
            start = 1'b0;
            @ (posedge clk);
            @(ready)                    // wait until decryption ready

            pt[i] = data_out;           // stores output plaintext
            ct[i+1] = pt[i];            // obtained ciphertext becomes ciphertext for next iteration

            @ (posedge clk);

          end
```

```
          key[j+1] = key[j] ^ {pt[iterations-2],pt[iterations-1]};      // next key generated from
current key and the last 2 plaintexts
          ct[0] = pt[iterations-1];     // initial ciphertext for next test will be the last plaintext
of current test

          $fdisplay(FILE, "Plaintext :\t%32h\n", pt[iterations-1]);    // display to file last
plaintext of current test

      end

      -> mct_done;      // Monte Carlo Test done

end


// ---------- SIMULATION RESULTS ----------

always @(mct_done)
  begin
            // display to file results
    $fdisplay(FILE, "###   MONTE CARLO TEST DONE   ###");
    $fdisplay(FILE, "Obtained Plaintext :\t%32h", pt[iterations-1]);
    $fdisplay(FILE, "Expected Plaintext :\t%32h\n", plaintext_mct_vector);
            // display to console results
    $display("###   MONTE CARLO TEST DONE   ###");
    $display("Obtained Plaintext :\t%32h", pt[iterations-1]);
    $display("Expected Plaintext :\t%32h\n", plaintext_mct_vector);

    if(pt[iterations-1] != plaintext_mct_vector)       // obtained plaintext different than expected
      begin
        $fdisplay(FILE, "Monte Carlo Test : FAILED");
        $display("Monte Carlo Test : FAILED");
      end
    else
      begin                                             // obtained plaintext as expected
        $fdisplay(FILE, "Monte Carlo Test : PASSED");
        $display("Monte Carlo Test : PASSED");
      end

    @ (posedge clk);
    $finish;

  end

endmodule
```

E.4.2 Test Results

Test : 0
Key : 2b09ba39b834062b9e93f48373b8dd018dedf1e5ba1b8af831ebbacbc92a2643
Ciphertext : 89649bd0115f30bd878567610223a59d
Plaintext : 1f9b9b213f1884fa98b62dd6639fd33b

Test : 1
Key : 58ac71619fdc3ac73a17f285319e1cd492766ac485030e02a95d971daab5f578
Ciphertext : 1f9b9b213f1884fa98b62dd6639fd33b
Plaintext : aecd334ef8fb0c51b6896ae065d8be28

Test : 2
Key : f6e3cca2cd628c10625c62cf08b385743cbb598a7df802531fd4fdfdcf6d4b50
Ciphertext : aecd334ef8fb0c51b6896ae065d8be28
Plaintext : f1938dd245c055e9c380336ff8450d9d

Test : 3
Key : a9ae3bd7d454f19d69289875ff009d16cd28d458383857badc54ce92372846cd
Ciphertext : f1938dd245c055e9c380336ff8450d9d
Plaintext : 42d5c4a13b748800ebfe0f67781dcff1

Test : 4
Key : e3bc6232865f0476149be7162ef8fc4b8ffd10f9034cdfba37aac1f54f35893c
Ciphertext : 42d5c4a13b748800ebfe0f67781dcff1
Plaintext : 469968a00226f0aae7acfda02b2ce0ae

Test : 5
Key : 451fbfba7f09b625540822fac8c8e30bc9647859016a2f10d0063c5564196992
Ciphertext : 469968a00226f0aae7acfda02b2ce0ae
Plaintext : eab144d6f80ccfd2fae95d16784718ac

Test : 6
Key : f53e21e46a0ec97c980d49d6f4b81ec423d53c8ff966e0c22aef61431c5e713e
Ciphertext : eab144d6f80ccfd2fae95d16784718ac
Plaintext : e04b91c3f084d733d3d0c1c7c152695a

Test : 7
Key : 5fb3a2cbdbe6971fcd345961bdcba5f6c39ead4c09e237f1f93fa084dd0c1864
Ciphertext : e04b91c3f084d733d3d0c1c7c152695a
Plaintext : e47db9b8c7fcc9459fa0a7fc84047b5b

Test : 8
Key : 6b145436c5ae0dcb2477f94f5bdb037927e314f4ce1efeb4669f07785908633f
Ciphertext : e47db9b8c7fcc9459fa0a7fc84047b5b
Plaintext : f617d26b55da999d65b2d236358b2e60

Test : 9
Key : 126752da3d7f9a0ffc1a46ccaa8a3925d1f4c69f9bc46729032dd54e6c834d5f
Ciphertext : f617d26b55da999d65b2d236358b2e60
Plaintext : 066ffea799ad5f09d03cb868deb1591e

Test : 90
Key : c4bde4a0c5f80329be6dd515f5bf6dabd805dd0a9ed853a94ba02087bae9596e
Ciphertext : 1515ef6a25cf3943eadadc36a029194a
Plaintext : 7b4ad946dfb59c80b1dc5cedb2fa87a0

Test : 91
Key : 80b47f9126b0b1f5ab408bb9ea5f349aa34f044c416dcf29fa7c7c6a0813dece
Ciphertext : 7b4ad946dfb59c80b1dc5cedb2fa87a0
Plaintext : fa10a53e14cde5ef1e9a8a5692847aa5

Test : 92
Key : 86a43939c8cd243f7ff2a9658524373b595fa17255a02ac6e4e6f63c9a97a46b
Ciphertext : fa10a53e14cde5ef1e9a8a5692847aa5
Plaintext : 4c9bed07308abcd0b09265b5efbe301c

Test : 93
Key : 74a8f02a15285153e0ae160fbddcf6bf15c44c75652a9616547493897529477
Ciphertext : 4c9bed07308abcd0b09265b5efbe301c
Plaintext : 75b7496c42809a3a592acbd8e069269e

Test : 94
Key : 293b40a2f70cc2c4262554663fb6f0986073051927aa0c2c0d5e58519540b2e9
Ciphertext : 75b7496c42809a3a592acbd8e069269e
Plaintext : 2dbc01185af7e084d90578468b6b10ef

Test : 95
Key : b04d4f4eb708eef86c5b6b8e08a665084dcf04017d5deca8d45b20171e2ba206
Ciphertext : 2dbc01185af7e084d90578468b6b10ef
Plaintext : c95b3a9f689c9dce4995c24f72dd5162

Test : 96
Key : 7cb9f7211815e0cb48b71286f84f80a184943e9e15c171669dcee2586cf6f364
Ciphertext : c95b3a9f689c9dce4995c24f72dd5162
Plaintext : 616e55c3bf113e2c18cae3c61b7eb7d1

Test : 97
Key : b3cdb46cf92aa0b96e87212bc650d5e5e5fa6b5daad04f4a8504019e778844b5
Ciphertext : 616e55c3bf113e2c18cae3c61b7eb7d1
Plaintext : 87e8b80767ebbdbad75cb94f4cb54f3b

Test : 98
Key : 512c2a3821eb53af613141c71e1076656212d35acd3bf2f05258b8d13b3d0b8e
Ciphertext : 87e8b80767ebbdbad75cb94f4cb54f3b
Plaintext : c83e20e18f2b1457788954b49fd84307

Test : 99
Key : 9977c985745bc33954a2ce898bc8febdaa2cf3bb4210e6a72ad1ec65a4e54889
Ciphertext : c83e20e18f2b1457788954b49fd84307
Plaintext : e3d3868f578caf34e36445bf14cefc68

MONTE CARLO TEST DONE
Obtained Plaintext : e3d3868f578caf34e36445bf14cefc68
Expected Plaintext : e3d3868f578caf34e36445bf14cefc68

Monte Carlo Test : PASSED

Appendix F – Code for testing on GUI

F.1 Verilog Code for Top Module

```
`timescale 1ns / 1ps

// ---------- AES256 UART ----------

module aes256_uart  (
                    clk100,
                    reset_n,
                    data_in,
                    LED,                 // output
                    data_out            // output
                    );


// ---------- MODULE INTERFACE ----------

input               clk100;         // global clock
input               reset_n;        // global negative edge reset
input               data_in;        // input data bits for RX
output wire [3:0]   LED;            // output from counter for RX
output wire         data_out;       // output data bits from TX


// ---------- MODULE REGISTERS AND SIGNALS ----------

  // 10 MHz clock
  // Want to interface to 115200 baud UART
  // 10000000 / 115200 = 87 Clocks Per Bit.
reg             clk;        // 10MHz
reg [3:0]       cnt_clk;

wire            rx;
wire            rx_ready;
wire [7:0]      rx_byte;

reg  [7:0]      tx_byte;
reg             tx_start;
wire            tx_busy;
wire            tx;

reg             aes_start;
wire            aes_start_enc;
wire            aes_start_dec;
reg  [7:0]      enc_dec;
reg  [255:0]    key;
reg  [127:0]    text_in;
wire [127:0]    text_out;
wire [127:0]    text_out_enc;
wire [127:0]    text_out_dec;
wire            aes_ready;
wire            aes_ready_enc;
wire            aes_ready_dec;

wire            rx_done;
parameter       num_bytes_rx = 49;    // number of bytes
parameter       num_bytes_tx = 16;    // number of bytes
reg  [5:0]      cnt_rx;              // loop integer to send data
reg  [4:0]      cnt_tx;

reg [2:0]           current_state;
parameter           RX_DATA = 6'd0,
                    AES = 6'd1,
                    TX_DATA = 6'd2,
                    TX_BUSY = 6'd3;
```

```
// ---------- STATE MACHINE ----------

always @(posedge clk100 or negedge reset_n)
  begin: CNT_CLK
    if(!reset_n)
      cnt_clk <= 0;
    else if(cnt_clk == 9)
      cnt_clk <= 0;
    else
      cnt_clk <= cnt_clk + 1;
  end

always @(posedge clk100 or negedge reset_n)
  begin: CLK_GEN
    if(!reset_n)
        clk <= 0;
    else if(cnt_clk < 5)
        clk <= 1;
    else
        clk <= 0;
  end


always @ (posedge clk or negedge reset_n)
  begin: STATE_MEMORY

    if (!reset_n)   current_state <= RX_DATA;
    else
      begin
        case (current_state)
          RX_DATA : current_state <= (rx_done) ? AES : RX_DATA;
          AES :     current_state <= (aes_ready) ? TX_DATA : AES;
          TX_DATA : current_state <= TX_BUSY;
          TX_BUSY : begin
                      if (!tx_busy && !tx_start)
                        current_state <= TX_DATA;
                      else if (cnt_tx == num_bytes_tx)
                        current_state <= RX_DATA;
                      else
                        current_state <= TX_BUSY;
                    end

          default : current_state <= RX_DATA;
        endcase
      end
  end

always @ (posedge clk or negedge reset_n)
  begin: RX

    if (!reset_n)
      begin
        enc_dec <= 0;
        key <= 0;
        text_in <= 0;
        aes_start <= 0;
      end
```

```
    else if (current_state == RX_DATA && rx_ready)
        case (cnt_rx)
            0:  enc_dec <= rx_byte;
            1:  key[7:0] <= rx_byte;
            2:  key[15:8] <= rx_byte;
            3:  key[23:16] <= rx_byte;
            4:  key[31:24] <= rx_byte;
            5:  key[39:32] <= rx_byte;
            6:  key[47:40] <= rx_byte;
            7:  key[55:48] <= rx_byte;
            8:  key[63:56] <= rx_byte;
            9:  key[71:64] <= rx_byte;
            10: key[79:72] <= rx_byte;
            11: key[87:80] <= rx_byte;
            12: key[95:88] <= rx_byte;
            13: key[103:96] <= rx_byte;
            14: key[111:104] <= rx_byte;
            15: key[119:112] <= rx_byte;
            16: key[127:120] <= rx_byte;
            17: key[135:128] <= rx_byte;
            18: key[143:136] <= rx_byte;
            19: key[151:144] <= rx_byte;
            20: key[159:152] <= rx_byte;
            21: key[167:160] <= rx_byte;
            22: key[175:168] <= rx_byte;
            23: key[183:176] <= rx_byte;
            24: key[191:184] <= rx_byte;
            25: key[199:192] <= rx_byte;
            26: key[207:200] <= rx_byte;
            27: key[215:208] <= rx_byte;
            28: key[223:216] <= rx_byte;
            29: key[231:224] <= rx_byte;
            30: key[239:232] <= rx_byte;
            31: key[247:240] <= rx_byte;
            32: key[255:248] <= rx_byte;

            33: text_in[7:0] <= rx_byte;
            34: text_in[15:8] <= rx_byte;
            35: text_in[23:16] <= rx_byte;
            36: text_in[31:24] <= rx_byte;
            37: text_in[39:32] <= rx_byte;
            38: text_in[47:40] <= rx_byte;
            39: text_in[55:48] <= rx_byte;
            40: text_in[63:56] <= rx_byte;
            41: text_in[71:64] <= rx_byte;
            42: text_in[79:72] <= rx_byte;
            43: text_in[87:80] <= rx_byte;
            44: text_in[95:88] <= rx_byte;
            45: text_in[103:96] <= rx_byte;
            46: text_in[111:104] <= rx_byte;
            47: text_in[119:112] <= rx_byte;
            48: begin
                  text_in[127:120] <= rx_byte;
                  aes_start <= 1;
                end

        endcase
    else if (current_state == AES)
        aes_start <= 0;

  end
```

```
always @ (posedge clk or negedge reset_n)
  begin: TX

    if (!reset_n)
      begin
        tx_byte <= 0;
        tx_start <= 1'b0;
      end
    else if (current_state == TX_DATA)
      begin
        tx_byte <= (cnt_tx == 0)  ? text_out  [7:0]  :
                   (cnt_tx == 1)  ? text_out [15:8]  :
                   (cnt_tx == 2)  ? text_out [23:16] :
                   (cnt_tx == 3)  ? text_out [31:24] :
                   (cnt_tx == 4)  ? text_out [39:32] :
                   (cnt_tx == 5)  ? text_out [47:40] :
                   (cnt_tx == 6)  ? text_out [55:48] :
                   (cnt_tx == 7)  ? text_out [63:56] :
                   (cnt_tx == 8)  ? text_out [71:64] :
                   (cnt_tx == 9)  ? text_out [79:72] :
                   (cnt_tx == 10) ? text_out [87:80] :
                   (cnt_tx == 11) ? text_out [95:88] :
                   (cnt_tx == 12) ? text_out[103:96] :
                   (cnt_tx == 13) ? text_out[111:104]:
                   (cnt_tx == 14) ? text_out[119:112]:
                   (cnt_tx == 15) ? text_out[127:120]: 8'b0;

        tx_start <= 1'b1;
      end
    else if (current_state == TX_BUSY)
        tx_start <= 1'b0;

  end


always @ (posedge clk or negedge reset_n)
  begin: CNT_RX

    if (!reset_n)
        cnt_rx <= 0;
    else if (rx_ready)
        cnt_rx <= cnt_rx + 1;
    else if (cnt_rx == num_bytes_rx)
        cnt_rx <= 0;

  end


always @ (posedge clk or negedge reset_n)
  begin: CNT_TX

    if (!reset_n)
           cnt_tx <= 0;
    else if (current_state==TX_DATA)
           cnt_tx <= cnt_tx + 1;
    else if (cnt_tx == num_bytes_tx && current_state==TX_BUSY)
           cnt_tx <= 0;

  end

assign aes_start_enc = (enc_dec == 0) ? aes_start : 0;
assign aes_start_dec = (enc_dec != 0) ? aes_start : 0;
assign text_out = (enc_dec == 0) ? text_out_enc : text_out_dec;
assign aes_ready = (enc_dec == 0) ? aes_ready_enc : aes_ready_dec;

assign rx = data_in;
assign rx_done = (cnt_rx == num_bytes_rx) ? 1'b1 : 1'b0;
assign data_out = tx;
assign LED = cnt_rx;
```

```
// ---------- MODULES INSTANTIATION ----------

aes256_enc  aes256_enc_u    (
                            .clk(clk),
                            .reset_n(reset_n),
                            .start(aes_start_enc),
                            .key_in(key),
                            .data_in(text_in),
                            .data_out(text_out_enc),    // output
                            .ready(aes_ready_enc)       // output
                            );

aes256_dec  aes256_dec_u    (
                            .clk(clk),
                            .reset_n(reset_n),
                            .start(aes_start_dec),
                            .key_in(key),
                            .data_in(text_in),
                            .data_out(text_out_dec),    // output
                            .ready(aes_ready_dec)       // output
                            );

serial_rx   serial_rx_u     (
                            .clk(clk),
                            .rst(!reset_n),
                            .rx(rx),
                            .new_data(rx_ready),    // output
                            .data(rx_byte)          // output
                            );

serial_tx   serial_tx_u     (
                            .clk(clk),
                            .rst(!reset_n),
                            .tx_block(1'b0),
                            .data(tx_byte),
                            .new_data(tx_start),
                            .busy(tx_busy),     // output
                            .tx(tx)             // output
                            );

endmodule
```

F.2 C++ Code for Qt GUI

F.2.1 Source Code

```
#include "serialcom.h"
#include <QSerialPort>
#include <QSerialPortInfo>
#include <QFile>
#include <QDebug>
#include <QByteArray>

SerialCom::SerialCom()
{
    read2 = false;
    num_bytes = 0;
    connect(&serial_port, SIGNAL(readyRead()), this, SLOT(read_serial()));

    QFile key_file("key_vectors.txt");
    if (!key_file.open(QIODevice::ReadOnly | QIODevice::Text)){
        qDebug() << "Error Opening File.";
        return;
    }

    QTextStream key_in(&key_file);
    int lines = 0;
    while (!key_in.atEnd()) {
       QString line = key_in.readLine();
       key_vectors[lines] = line;
       lines++;
    }

    QFile plaintext_file("plaintext_vectors.txt");
    if (!plaintext_file.open(QIODevice::ReadOnly | QIODevice::Text)){
        qDebug() << "Error Opening File.";
        return;
    }

    QTextStream plaintext_in(&plaintext_file);
    lines = 0;
    while (!plaintext_in.atEnd()) {
       QString line = plaintext_in.readLine();
       plaintext_vectors[lines] = line;
       lines++;
    }

    QFile ciphertext_file("ciphertext_vectors.txt");
    if (!ciphertext_file.open(QIODevice::ReadOnly | QIODevice::Text)){
        qDebug() << "Error Opening File.";
        return;
    }

    QTextStream ciphertext_in(&ciphertext_file);
    lines = 0;
    while (!ciphertext_in.atEnd()) {
       QString line = ciphertext_in.readLine();
       ciphertext_vectors[lines] = line;
       lines++;
    }

}
```

```
void SerialCom::open(){
    QString USB_Port;

    foreach(const QSerialPortInfo &serialPortInfo, QSerialPortInfo::availablePorts()){

        if((serialPortInfo.vendorIdentifier()==1027) && (serialPortInfo.productIdentifier()==24577)){
        USB_Port = serialPortInfo.portName();
        }/*
        qDebug() << serialPortInfo.portName();
        qDebug() << serialPortInfo.vendorIdentifier();
        qDebug() << serialPortInfo.productIdentifier();*/
    }

    serial_port.setPortName(USB_Port);
    serial_port.setBaudRate(QSerialPort::Baud115200);
    serial_port.setDataBits(QSerialPort::Data8);
    serial_port.setParity(QSerialPort::NoParity);
    serial_port.setStopBits(QSerialPort::OneStop);
    serial_port.setFlowControl(QSerialPort::NoFlowControl);
    bool error_sp = serial_port.open(QIODevice::ReadWrite);

    if(error_sp == false){ qDebug() << "Error trying to open port"; }
    else qDebug() << "Port Opened";

}

void SerialCom::close(){
    serial_port.close();
    qDebug() << "Port Closed";
}

void SerialCom::read_serial(){

    QByteArray  read_buf;
    QString  exp_data_out;
    int num_bytes_2;

    data_out.resize(16);

    buf.resize(16);

    read_buf = serial_port.read(20);
    num_bytes_2 = read_buf.size();
    read_buf.resize(num_bytes_2);

    //qDebug() << "Buf: " << read_buf.toHex();
    //qDebug() << "Read: " << num_bytes_2;

    for(int i=0; i < num_bytes_2; i++){ buf[i+num_bytes] = read_buf[i]; }
    num_bytes += num_bytes_2;
    //qDebug() << "Acum: " << num_bytes;
```

```
    if(num_bytes == 16){
        num_bytes = 0;
        for(int i=0; i < 16; i++){
            data_out[i] = buf[15-i];
        }
        qDebug() << "Total Buf: " << data_out.toHex();
        if(enc_dec){
            exp_data_out = ciphertext_vectors[test_num-1];
        }
        else{
            exp_data_out = plaintext_vectors[test_num-1];
        }
        if(data_out.toHex() == exp_data_out){
            result = "PASSED";
        }
        else{
            result = "FAILED";
        }

        emit dataReceived();
    }

}

void SerialCom::sendMsg(int test, bool en_de){

    enc_dec = en_de;
    test_num = test;
    QString datain;
    QString str_key_datain;
    QString sliced_str;
    QByteArray key_datain;
    key_datain.resize(49);
    int hexByte;
    bool ok;

    QString key = key_vectors[test_num-1];     // Bits: 256, Bytes: 32, Nibbles:64
    if(enc_dec){ // ENC
        datain = plaintext_vectors[test_num-1];  // Bits: 128, Bytes: 16, Nibbles:32
        // Key + Data_in: Bits: 384, Bytes: 48, Nibbles:96
        // Enc_Dec + Key + Data_in: Bits: 392, Bytes: 49, Nibbles:98

        str_key_datain = datain.append(key).append("00");
    }
    else{   // DEC
        datain = ciphertext_vectors[test_num-1];
        str_key_datain = datain.append(key).append("01");;
    }

    for(int i=0; i < 49; i++){
        sliced_str = str_key_datain.sliced(i*2,2);
        hexByte = sliced_str.toInt(&ok, 16);
        key_datain[48-i] = hexByte;
    }

    int n = serial_port.write(key_datain,49);
    if(n == -1){ qDebug() << "Writting Fail"; }
    qDebug() << "Msg Sent";
    //qDebug() << key_datain.toHex();

}
```

F.2.2 Header File

```
#ifndef SERIALCOM_H
#define SERIALCOM_H

#include <QSerialPort>
#include <QString>
#include <QObject>

class SerialCom: public QObject
{
    Q_OBJECT

public:
    SerialCom();
    void open();
    void close();
    void sendMsg(int test, bool en_de);

    QString key_vectors[405];
    QString plaintext_vectors[405];
    QString ciphertext_vectors[405];
    QByteArray data_out;
    QString result;

private:
    QSerialPort serial_port;
    QByteArray  buf;
    unsigned char read_buf_ch[15];
    bool read2;
    int num_bytes = 0;
    int test_num;
    int enc_dec;

signals:
    void dataReceived();

private slots:
    void read_serial();
};

#endif // SERIALCOM_H
```