COMBINED AES + AEGIS ARCHITECTURES FOR HIGH PERFORMANCE AND LIGHTWEIGHT SECURITY APPLICATIONS

A Thesis

by

Furkan Şahin

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Electrical and Electronics Engineering

Özyeğin University
August 2014

COMBINED AES + AEGIS ARCHITECTURES FOR HIGH PERFORMANCE AND LIGHTWEIGHT SECURITY APPLICATIONS

Approved by:

Assoc. Prof. H. Fatih Uğurdağ, Advisor
Department of Electrical and Electronics Engineering
Özyeğin University

Asst. Prof. Tolga Yalçın, Co-Advisor

University of Information Science and Technology, Macedonia

Assoc. Prof. Sezer Gören Uğurdağ
Department of Computer Engineering
Yeditepe University

Asst. Prof. Tankut Barış Aktemur
Department of Computer Science
Özyeğin University

Date Approved: 20 August 2014

Asst. Prof. Ali Özer Ercan
Department of Electrical and Electronics Engineering
Özyeğin University

To my family

ABSTRACT

Cryptography is going into everything, from bank cards to cell phones, cars, communication devices, cloud services, etc. There are many cryptographic algorithms to protect information from unauthorized accesses. The Advanced Encryption Standard (AES) is the most important block cipher today, since it is ratified as a standard by National Institute of Standards and Technology of the United States. It's proven security and reasonable resource usage makes it a right choice for almost all new applications. Several versions of AES have been implemented in both hardware and software with design targets varying from high-performance to lightweight. Since today's information security applications require both confidentiality and authentication, authenticated encryption (AE) has gained more importance. AES is considered by many cryptographers as the most appropriate choice for AE implementations. More recently, special AE schemes that utilize AES in its native form (or with minimal modifications) have emerged. While these modes claim better performance and resource usage, very few implementations exist to support these claims, yet. In this thesis, AES is combined with one of the most recent AE ciphers, namely AEGIS, in an effort to analyze the combined performance and resource usage of the two ciphers. This thesis proposes and implements in hardware two different architectures (i.e., high-performance and lightweight) for AES and AEGIS combined as well as AES alone, hence four architectures in total. This work is the first work to our knowledge to report a hardware implementation of AEGIS as well as it's combined implementation with AES, which reuses most hardware resources between AES and AEGIS. Implementation results obtained by using UMC 90nm low-leakage standard cell library and Cadence RTL Compiler are also reported and evaluated.

ÖZETÇE

Günümüzde kriptografi banka kartlarından telefonlara, arabalardan haberleşme araçlarına ve bulut hizmetlerine kadar pek çok alana girdi. Dijital bilgiyi yetkilendirilmemiş erişimlere karşıkorumak için bir çok şifreleme algoritması mevcuttur. Gelişmiş Şifreleme Standardı (AES) Amerikan Ulusal Standartlar ve Teknoloji Enstitüsü (U.S. NIST) tarafından standart şifreleme algoritması olarak onaylandıktan sonra en önemli blok şifreleyici olmuştur. AES'in kanıtlanmış güvenliği ve makul donanım kullanımı onu yeni bilgi güvenliği uygulamaları için mantıklı bir seçim yapmaktadır. Bugüne kadar, yüksek performanstan düşük alan kullanımına kadar pek çok tasarım yaklaşımını hedefleyen AES versiyonları donanımsal ve yazılımsal olarak gerçeklendi. Ancak günümüz bilgi güvenliği uygulamaları hem gizlilik hem de kimlik doğrulama gerektirdiği için, kimlik doğrulamalı şifreleme giderek daha çok önem kazanmaktadır. AES, bir çok kriptografi bilimcisi tarafından kimlik doğrulamalı şifrelemenin gerçeklenmesinde mantıklı bir seçim olarak düşünülmektedir. Son zamanlarda literatürde, AES algoritmasını olduğu gibi kullanan ya da AES'te küçük değişiklikler yaparak kullanan kimlik doğrulamalı şifreleme algoritmları çokça gözükmeye başlamıştır. Bu yeni kimlik doğrulamalı şifreleme algoritmalarının bazılarında daha iyi performans ve daha az kaynak kullanımı başarılabileceği iddia edilmektedir. Ancak bu algoritmalar henüz literatüre yeni girdiği için bu iddiaları destekleyen çok az donanım gerçeklemesi mevcuttur, hatta bazıları donanımsal olarak henüz gerçeklenmemiştir. Bu tezde AES'i, çok yakın zamanda literatüre giren AES tabanlı kimlik doğrulamalı şifreleme algoritması olan AEGIS ile aynı donanımda birleştiren mimariler tasarlanmştr. Buradaki amacımız iki algoritmanın tek bir devrede gerçeklenmesine örnek teşkil edecek bir mimari sunmak, birleşik performans ve donanım kaynağı kullanımını analiz etmek ve AEGIS'ın tasarımcılarının kaynak kullanımı konusundaki iddialarının tartışılabileceği sentez sonuçları elde etmektir. Ayrıca bilgilerimize göre literatürde AEGIS'ın donanımda gerçeklenmesini sunan bir çalışma henüz yapılmamıştır ve AEGIS ilk kez bu çalışmada donanımsal olarak gerçeklenmiştir. Bu tezde yüksek performans ve az donanım kaynağı kullanan olmak üzere iki değişik, birleşik AES + AEGIS donanım mimarisi sunulmuş ve bu devreler Verilog donanım tanımlama dilinde (Verilog HDL) yazmaç transfer seviyesinde (RTL) kodlanmıştır. Ayrıca sadece AES şifrelemesi yapan devreler de gerçeklenip, her iki birleşik mimarinin kaynak kullanımı bu devreler ile kendi kategorisinde kıyaslanmıştır. Devreler UMC 90nm standart hücre kütüphanesiyle ve Cadence RTL derleyicisi kullanılarak gerçeklenmiştir. Sentez sonuçları raporlanmış ve değerlendirilmiştir.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisors Assoc. Prof. H. Fatih Uğurdağ and Asst. Prof. Tolga Yalçın for their continuous support during my M.S. work, for their care, patience, immense knowledge and invaluable guidance. It would not have been possible to complete this thesis without their understanding.

Besides, I want to thank Assoc. Prof. Sezer Gören, Asst. Prof. Ali Özer Ercan and Asst. Prof. Tankut Barış Aktemur for taking part in my thesis committee and for dedicating their valuable time to my thesis. Their helpful comments and feedbacks improved the quality of this thesis.

Also, I express my warm thanks to all of nEMESysLab people who made my life enjoyable during my graduate studies. Especially, Fatih Temizkan, Bilgiday Yüce, Waqas Hussain, Mustafa Tosun, Ahmet Kakacak, Mert Kaya, Aydın Emre Güzel, Mehmet Polat Küntüz, Gökhan Güner, Abdullah Yıldız and Batuhan Yapanoğlu.

Finally, I am most grateful to my family for their love and endless support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Information security is gaining more importance everyday. Today, computation and communication devices are everywhere. People make monetary operations on their bank account via personal computers, mobile phones and tablets. Military and intelligence informations are mostly secret, so they must be kept secure also. Various cloud services also wide spread. People transfer and store their critical data on a cloud servers. Therefore it is very important and critical to guarantee information security, privacy and authenticity. Tool for the information security is cryptography.

The word cryptography originates from Greek words of *kryptos* (hidden, secret) and *graphein* (writing) [1]. Cryptography is defined as the science of protecting information by transforming it into an unintelligible format, called *cipher text*. Only those who possess the secret key can decipher the message back to *plain text*.

Cryptography is a new and growing field; it gained much importance with digital communication becoming widespread . However, it is also an old business, it's early examples dating back to about 2000 B.C. [2]. Ancient Egyptians used secret hieroglyphics. Hiding information has always been important for human beings. In the earlier times, simple cryptographic algorithms such as Caesar substitution was used. As time progress, cryptography evolved into today's modern cryptography. With increasing security needs, many cryptographic algorithms were developed, and they have been implemented in both hardware and software by researchers and engineers. Some of them are broken by attackers, some of them has proven secure and unbroken to this date. The most well-known one is the Advanced Encryption Standard (AES) [3]. It is approved by Federal Information Processing Standards in 2001. Since AES was introduced, many attackers have tried to break it, no one has been successful yet. The protection of a message involves both confidentiality and authenticity.

1.1 Motivation

The Advanced Encryption Standard (AES) is the most widely used symmetric cipher today. The AES block cipher is mandatory in several industry standards and is used in many commercial systems. The Internet security standard IPsec, TLS, the Wi-Fi encryption standard IEEE 802.11i, the SSH (Secure Shell), Skype and numerous security products around the world are among the commercial standards that include AES. To date, there are no attacks better than brute-force known against AES. AES is mainly designed and used to perform encryption and decryption. It can also be used with block cipher modes of operations to provide authentication as well. For example CCM mode (Counter with CBC-MAC), uses AES for encryption and authentication. So it turns into an authenticated encryption cipher [4]. Authenticated encryption (AE) ciphers are backbones of Internet Protocol Security [5]. Block ciphers with a hash function also an alternative for implementing AE cipher. However, none of the offered solutions are resource efficient. Some of them are worse throughput wise, since they runs AES twice, once for encryption and once for authentication. Some of them worse resource usage-wise, since they use a second module to perform hashing.

Recently, many specialized AE ciphers have been proposed. They can be categorized into three groups. First group introduced a completely new structure for AE. The second group uses hash functions in AE. The third group modifies the AES somehow to achieve authenticated encryption. AES-based Lightweight Authenticated Encryption (ALE) and AEGIS are two such examples of this type of ciphers [6] [7]. Cryptographers who proposed these AES-based AE ciphers also claim that these ciphers can be easily implemented by using an existing AES implementation, and resulting circuit will be resource efficient. However, since these type of ciphers introduced most recently, there is no implementation that prove these claims.

In this thesis, we choose AEGIS, the most recent AES-based AE proposal at the time we started to this work. We targeted to design architectures and implement them in hardware to examine the validity of such claims. In the end we propose combined architectures for both high performance and lightweight applications that users can switch between AES block cipher and AEGIS AE modes by using a simple switch.

1.2 Previous Work

AES has been implemented both on hardware and on software many times by researchers and engineers since it was introduced. High speed and high throughput hardware architectures introduced in [8] [9] [10] [11]. On the other hand low area architectures are introduced in [12] [13] [14] [15] [16]. Different approaches to increase speed and throughput are presented in literature. Look-up table (LUT) based FPGA implementations are presented in [17] [18]. High speed pipelined architectures are presented in [19] [20]. A sub-pipelined high speed very large scale integrated (VLSI) architecture is presented in [9]. Since AES-based AE ciphers are in infancy period, there is no hardware implementation for AEGIS yet. AEGIS designer claims, it can be easily implemented with a minimal effort by utilizing existed native AES hardware [7]. These claims are not proven yet.

1.3 Outline

After a brief introduction in this chapter, mathematical preliminaries are described in Chapter II. Most AES and AEGIS calculations are done in Galois Field (GF). Chapter III gives an overview of AES, operations and blocks of AES are described and an iterative standalone AES architecture is presented. An AEGIS overview is presented in Chapter IV. Functions and processing phases of AEGIS is described in that chapter. In Chapter V, our high performance combined AES + AEGIS architecture is presented. In Chapter VI, our lightweight combined AES + AEGIS architecture is presented. Finally, synthesis results and future work are given in Chapter VII.

CHAPTER II

MATHEMATICAL PRELIMINARIES

Since most AES calculations are done in GF(2^8), before describing internal functions of AES, it is necessary to give an overview and describe Galois Field (GF) operations.

2.1 Finite Field

In algebra, a field defined as a non-zero commutative ring that contains a multiplicative inverse for every non-zero element [21]. A finite field is a field that contains a finite number of elements. Sometimes it is also called as Galois field (to honor Evariste Galois) [22]. A finite field is a set that commutative addition, subtraction, multiplication and division operations have been defined on it. the number of elements in the field is called as the order of the field. The following theorem defines finite field and has a fundamental importance [2].

Theorem 2.1.1:

A field with order m only exists if m is a prime power, i.e., $m = p^n$, for some positive integer n and prime integer p. p is called the characteristic of the finite field.

2.2 GF(2^m) Extension Fields

The finite field in AES contains 256 (0 to 255) elements and it is represented as GF(2^8). AES treats every byte of internal data as an element of the finite field GF(2^8) and manipulates the data by performing some arithmetic operations in this field. If the order of a finite field is not a prime (e.g. 2^8) the addition and multiplication operations cannot be operated by addition and multiplication of integers modulo 2^8. If m > 1, such fields are called extension fields. There are different notations and different rules for performing arithmetic operations on extension fields [2].

In extension field GF(2^8) elements are represented as polynomials with coefficients in GF(2). Maximum degree of polynomials is m-1. In the field GF(2^8) each element A is represented as: $A(x) = a_7x^7 + ... + a_1x + a_0$, where $a_i \in GF(2) = 0, 1$

Totally there are 256 such polynomials in GF(2^8). Every polynomial can be stored as an 8-bit vector $A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$.

Let A and B be polynomials defined over GF(2^m):

$A = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0$

$B = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + ... + b_1x + b_0$

In the following sections, arithmetic operations in GF(2^8) are described.

2.3 GF(2^m) Addition

Addition is the simplest arithmetic operation in GF(2^m). The sum of A and B is:

$C = A + B = (a_{m-1} + b_{m-1})\, x^{m-1} + ... + (a_1 + b_1)\, x + a_0 + b_0$

Since a_i and b_i are defined over GF(2), addition is performed in modulo-2, which is the logical *xor* operation.

$C = (a_{m-1} \oplus b_{m-1})\, x^{m-1} + ... + (a_1 \oplus b_1)\, x + (a_0 \oplus b_0)$

2.4 GF(2^m) Multiplication

$C = A \times B = (a_{m-1}b_{m-1})\, x^{2m-2} + (a_{m-1}b_{m-2})\, x^{2m-3} + ... + (a_0b_{m-1})\, x^{m-1} + (a_0b_{m-2})\, x^{m-2} + ... + a_0b_0$

Since in GF(2^m) it is not allowed any element with power greater than x^{m-1}, an elimination procedure have to be operated. To understand that procedure, let's see it is for m = 8:

The irreducible polynomial for multiplication in GF(2^8) is $x^8 + x^4 + x^3 + x + 1$

Firstly calculate partial products:

Let $S_7 = M_7 = m_{77}x^{14} + m_{76}x^{13} + ...\ m_{70}x^7$

$$\begin{array}{lll}
 & b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 & \\
x & a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0 & \\
\hline
a_7b_7\ a_7b_6\ \ldots\ a_7b_0 & & \rightarrow M_7:\ m_{77}m_{76}m_{75}m_{74}m_{73}m_{72}m_{71}m_{70} \\
\quad \ldots\ldots\ldots\ldots & & \rightarrow M_6:\ m_{67}m_{66}m_{65}m_{64}m_{63}m_{62}m_{61}m_{60} \\
\qquad \ldots\ldots\ldots\ldots & & \rightarrow M_5:\ m_{57}m_{56}m_{55}m_{54}m_{53}m_{52}m_{51}m_{50} \\
\qquad\quad \ldots\ldots\ldots\ldots & & \rightarrow M_4:\ m_{47}m_{46}m_{45}m_{44}m_{43}m_{42}m_{41}m_{40} \\
 & \ldots\ldots\ldots\ldots & \rightarrow M_3:\ m_{37}m_{36}m_{35}m_{34}m_{33}m_{32}m_{31}m_{30} \\
 & \quad \ldots\ldots\ldots\ldots & \rightarrow M_2:\ m_{27}m_{26}m_{25}m_{24}m_{23}m_{22}m_{21}m_{20} \\
 & \qquad \ldots\ldots\ldots\ldots & \rightarrow M_1:\ m_{17}m_{16}m_{15}m_{14}m_{13}m_{12}m_{11}m_{10} \\
 & \qquad\quad a_0b_7\ a_0b_6\ \ldots\ a_0b_0 & \rightarrow M_0:\ m_{07}m_{06}m_{05}m_{04}m_{03}m_{02}m_{01}m_{00}
\end{array}$$

Rewriting S_7:

$S_7 = x^6\ (m_{77}x^8 + m_{76}x^7 + \ldots\ m_{70}x)$

Recall that $x^8 = x^4 + x^3 + x + 1$

$S_7 = x^6\ (m_{77}(x^4 + x^3 + x + 1) + m_{76}x^7 + \ldots\ m_{70}x)$

$S_7 = x^6\ (m_{76}x^7 + m_{75}x^6 + m_{74}x^5 + (m_{77} + m_{73})x^4 + (m_{77} + m_{72})x^3 + m_{71}\ x^2 + (m_{77} + m_{70})\ x + m_{77})$

$S_7 = s_{77}x^{13} + s_{76}x^{12} + s_{75}x^{11} + s_{74}x^{10} + s_{73}x^9 + s_{72}x^8 + s_{71}x^7 + s_{70}x^6$

Add M_6 to S_7 :

$S_6 = S_7 + M_6 = (s_{77} + m_{67})x^{13} + (s_{76} + m_{66})x^{12} + \ldots + (s_{70} + m_{60})x^6$

$S_6 = x^5\ ((s_{77} + m_{67})x^8 + (s_{76} + m_{66})x^7 + \ldots + (s_{70} + m_{60})x)$

Then substitute $x^4 + x^3 + x + 1$ to x^8 and repeat same procedures for S_6, S_5, S_4, S_3, S_2, S_1 to obtain $S_0 = s_{07}x^7 + s_{06}x^6 + s_{05}x^5 + s_{04}x^4 + s_{03}x^3 + s_{02}x^2 + s_{01}x + s_{00}$, which is the multiplication result in $GF(2^8)$.

2.5 $GF(2^m)$ Inversion

$GF(2^m)$ can be also represented as $GF((2^{m/2})^2)$. Then any symbol in $GF(2^8)$, A, can be represented as :

$A = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ $A = A_1y + A_0$, where A_1,

$A_0 \in GF(2^4)$

Then firstly apply a *isomorphic transform* to A to get $\hat{A}$ = I(A). The isomorphic transform is the multiplication of the vector A_{8x1} with a 8x8 transform matrix. Although it seems complex in terms of hardware, thanks to multiplication being AND operation and addition being XOR operation in GF(2), it is possible to end up with a very simple design.

The isomorphic transform matrix is given below:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$\hat{A} = \hat{A}_1 y + \hat{A}_0$ where $y^2 + y + w_0 = 0$

Then the inverse of $\hat{A}$ is:

$\hat{F} = \hat{A}^{-1} = \frac{\hat{A}_1}{\hat{A}_0(\hat{A}_0+\hat{A}_1)+w_0\hat{A}_1} y + \frac{\hat{A}_1+\hat{A}_0}{\hat{A}_0(\hat{A}_0+\hat{A}_1)+w_0\hat{A}_1}$

Then apply the inverse *isomorphic transform* to $\hat{F}$ to get F = A^{-1}. Inverse isomorphic transform is very similar to isomorphic transform, but 8x8 inverse transform matrix multiplied with input matrix. The inverse isomorphic transform matrix is given below:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

$F = I^{-1}(\hat{F}) = IIT\_matrix \times \hat{F}$

Block diagram of inversion in $GF(2^8)$ is given in Figure 1

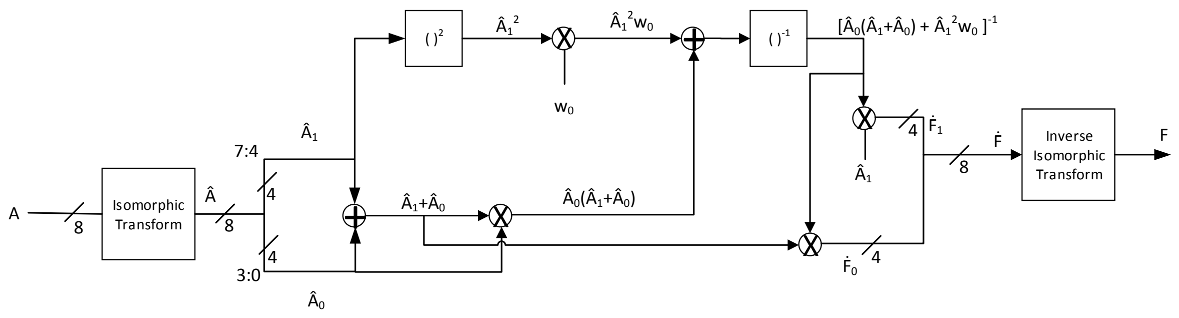

Figure 1: GF(2^8) inversion block diagram

CHAPTER III

AES OVERVIEW

The Advanced Encryption Standard (AES) declares a symmetric block cipher that can be used to protect data [23]. It is also known as Rijndael algorithm, since it is designed by two Belgian cryptographers Joan Daemen and Vincent Rijmen. After Data Encryption Standard (DES) indicate some weaknesses, the U.S. NIST called proposal for new cryptographic algorithm, which is Advanced Encryption Standard in 1997 [2]. After some evaluation rounds NIST announced that it had chosen Rijndael as the AES on October 2, 2000. In 2001, NIST declared the new AES and published it as a final standard (FIPS PUB 197).

In this chapter, AES is over viewed. For detailed information about AES, we refer to [23].

3.1 Notation and Conventions

In this section; inputs, outputs and state of AES are explained.

3.1.1 Inputs and Outputs

As shown in Figure 2 The input of AES is a sequence of 128 bits called as *plain text.* The output is also a sequence of 128 bits and it is called as *cipher text.* Cipher *key* is also another input, it could be 128, 192, 256 bits.

3.1.2 The State

The basic unit in processed in AES is byte, a sequence of 8 bits. The array of bytes is represented as $\mathrm{a_0a_1a_2}$ ... $\mathrm{a_n}$. The 128-bit input plain text block (in_bit_0 to in_bit_127) of AES firstly grouped to array of bytes like $\mathrm{in_0in_1}$ ... $\mathrm{a_{15}}$. Then it is

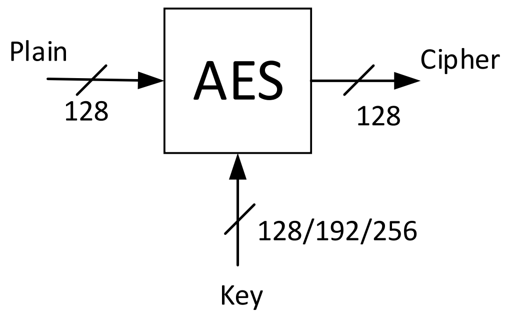

Figure 2: AES inputs and outputs

putted into a 4x4 array of bytes, called as ***state***. AES operates on this 4x4 state array.

The state goes through repetition of processing steps and finally cipher text is outputted in same fashion, 128-bit output (out_bit_0 to out_bit_127). These conventions are depicted in Figure 3.

3.2 AES Encryption Algorithm

The AES cipher converts input plain text into output cipher text after specified repetitions of round transformation. The pseudo code for cipher is given below. It is taken from FIPS PUB 197 [23]. The Nr is a generic number, representing number of rounds, depends on the key size. It is value is 10, 12 and 14 for key sizes of 128, 192, 256, respectively. The array w contains the key schedule.

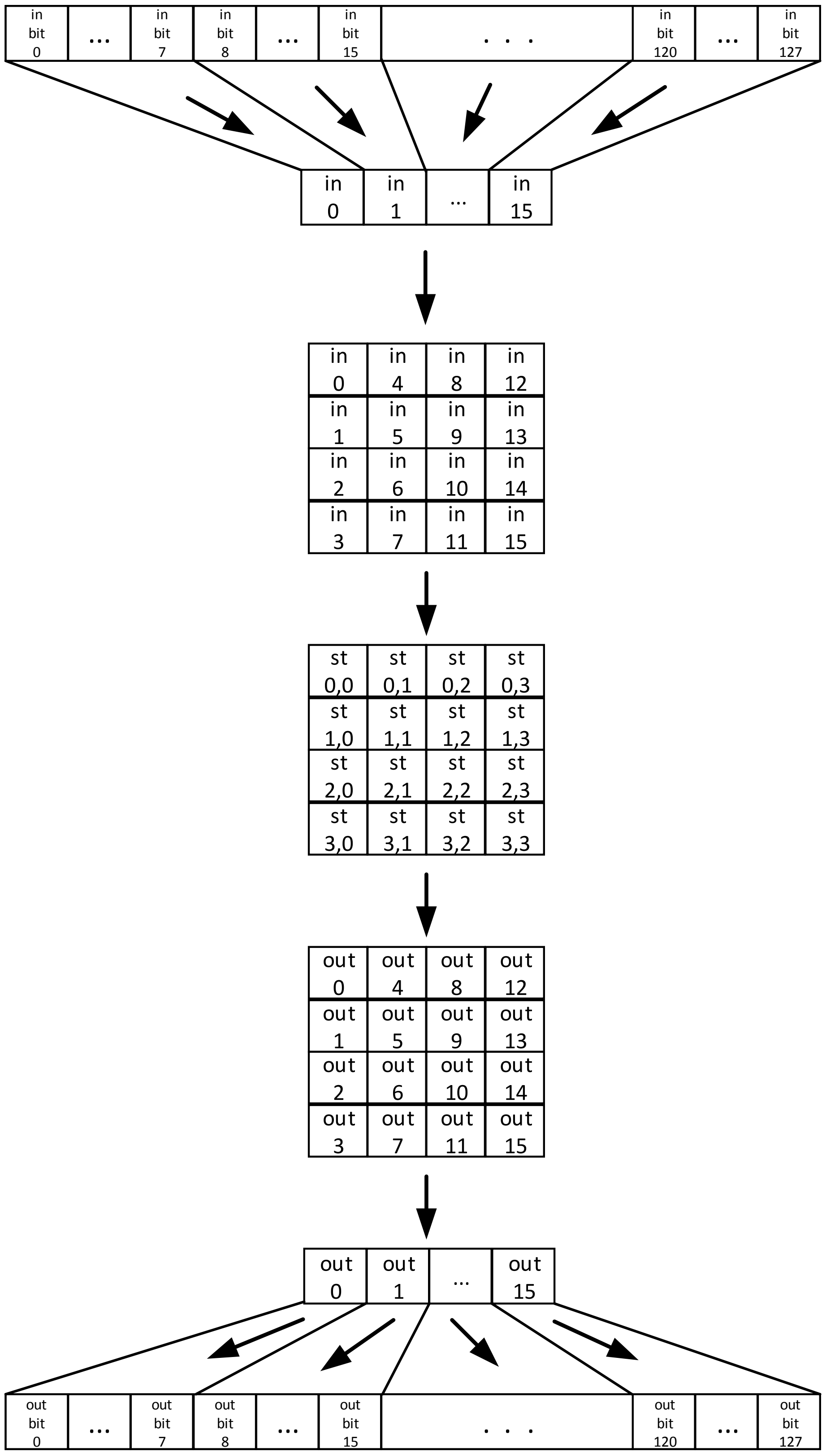

Figure 3: AES I/O and state conventions

```
Cipher (byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4, Nb]
    state = in
    AddRoundKey(state, w[0, Nb−1])
    for round = 1 step 1 to Nr−1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb−1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb−1])
    out = state
end
```

Table 1: AES key sizes and number of rounds (Nr)

key size	Nr
128 bits	10
192 bits	12
256 bits	14

Basically there are four transformations (AddRoundKey, SubBytes, ShiftRows, MixColums) in the algorithm. In the following sections, these transformations are described.

3.3 AddRoundKey

It is the simplest transformation in AES. The *subkey*, which derived from original encryption key, is added to *state* as shown in Figure 4. Addition is equivalent to bitwise XOR, since operations performed over finite fields. The *subkey* derived using key schedule and each *subkey* is the same size with the original key. The AES key scheduling algorithm also described in Section 3.7.

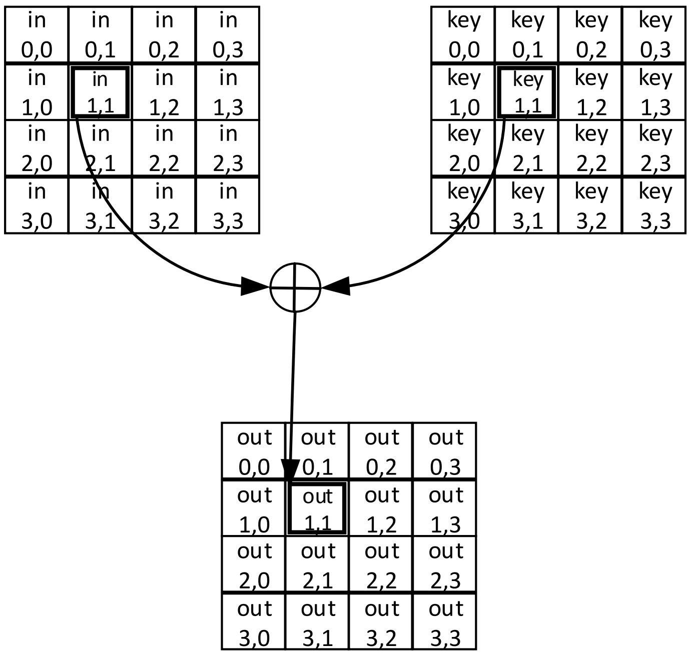

Figure 4: AES AddRoundKey transformation

3.4 SubBytes

The SubBytes transformation, also known as S-box, updates each input byte of state according to a substitution box. This is a non-linear transformation. The substitution box (S-box) is derived from following affine transformation.

$$\begin{bmatrix} out_0 \\ out_1 \\ out_2 \\ out_3 \\ out_4 \\ out_5 \\ out_6 \\ out_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} inv\_in_0 \\ inv\_in_1 \\ inv\_in_2 \\ inv\_in_3 \\ inv\_in_4 \\ inv\_in_5 \\ inv\_in_6 \\ inv\_in_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$[in_7 in_6 in_5 in_4 in_3 in_2 in_1 in_0]$ and $[out_7 out_6 out_5 out_4 out_3 out_2 out_1 out_0]$ are input and output bytes, respectively. $[inv\_in_7\ inv\_in_6\ inv\_in_5\ inv\_in_4\ inv\_in_3\ inv\_in_2\ inv\_in_1\ inv\_in_0]$ is the multiplicative inverse of the input byte. All arithmetic operations are performed on GF(2^8) with the irreducible polynomial p(x) = $x^8 + x^4 + x^3 + x + 1$.

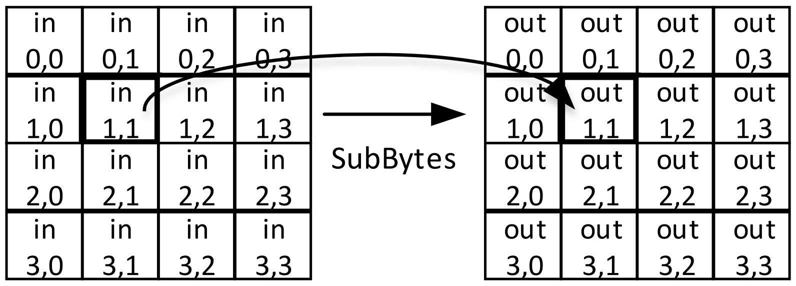

Figure 5: AES SubBytes transformation

3.5 ShiftRows

The ShiftRows transformation cyclically shifts the bytes in each rows as shown in Figure 6. The first row of state matrix is not changed, the second row of the state matrix is shifted by three bytes to the right, the third row is shifted by two bytes to the right and the fourth row is shifted by one byte to the right by the ShiftRows.

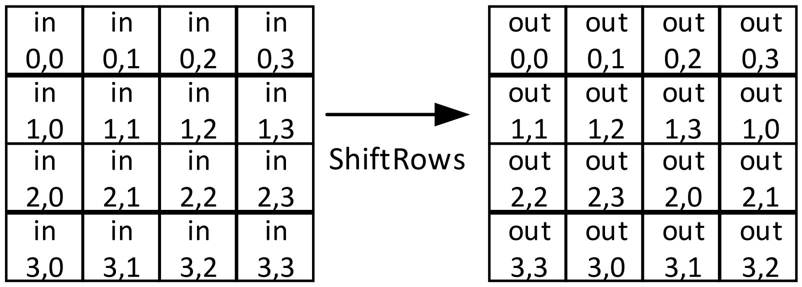

Figure 6: AES ShiftRows transformation

3.6 MixColumns

The MixColumns is a linear transformation which mixes each column of state matrix. Each 4-byte column is considered as a polynomial over GF(2^8) and multiplied modulo $x^4 + 1$ by a fixed constant 4x4 matrix as follows:

$$\begin{bmatrix} out_{0,\mathrm{c}} \\ out_{1,\mathrm{c}} \\ out_{2,\mathrm{c}} \\ out_{3,\mathrm{c}} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} in_{0,\mathrm{c}} \\ in_{1,\mathrm{c}} \\ in_{2,\mathrm{c}} \\ in_{3,\mathrm{c}} \end{bmatrix}$$

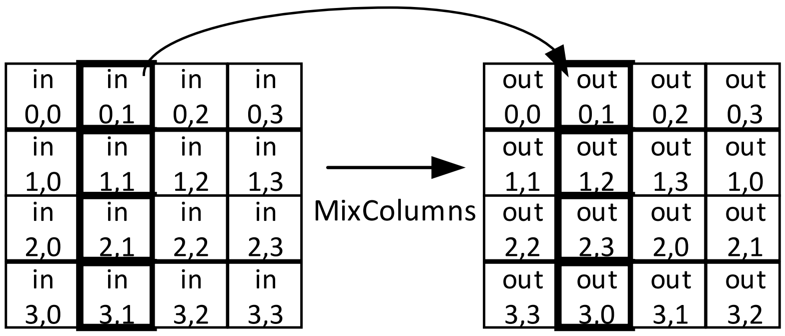

Figure 7: AES MixColumns transformation

3.7 Key Expansion

In AES algorithm, a key schedule is generated from original encryption key. The hey schedule consists of *subkeys*, each of them is used as a key in a round. Pseudo code of **Key Expansion**, which is taken from [23], given below. It is a generic *Key Expansion* code, which supports key sizes 128, 192 and 256. **w**[] is the output key schedule. **Nr** is the number of rounds, which is 10, 12 and 14 for key sizes of 128, 192 and 256, respectively. Nk is 4, 6 and 8 for key sizes of 128, 192 and 256, respectively. **SubWord** is 4-byte version of **Subbytes**. **RotWord** shifts bytes of a word to left by one byte. **Rcon[i]** contains the values given by [$\{02\}^{i-1}$,{00},{00},{00}], with x^{i-1} being powers of x in $GF(2^8)$.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i = 0
    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1)]
        temp = w[i−1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i−Nk] xor temp
        i = i + 1
    end while
end
```

CHAPTER IV

AEGIS OVERVIEW

AEGIS is a dedicated authenticated encryption algorithm, which is constructed from the AES encryption round function [7]. It is one of the proposed submission to CAESAR(Competition for Authenticated Encryption: Security, Applicability, and Robustness), one of the latest cryptographic competition, by Hongjun Wu and Bart Preneel [24]. According to the designers of AEGIS, the algorithm offer high levels of security. There are three version of AEGIS, which are AEGIS-128, AEGIS-256, and AEGIS-128L. Depending on AEGIS version 5, 6 and 8 **AES Round** operation performed in AEGIS-128, AEGIS-256, and AEGIS-128L, respectively. Intermediate cipher results are called **state** also in AEGIS, however, in contrast to AES, a state consists of 5, 6 or 8 16-byte data blocks in AEGIS-128, AEGIS-256, and AEGIS-128L, respectively. A function called *StateUpdate* performs 5, 6, 8 AES rounds on **state** in AEGIS-128, AEGIS-256, and AEGIS-128L, respectively. **Initialization, Processing The Authenticated Data, Encryption, and Finalization** are processing phases of AEGIS. Depending on which AEGIS algorithm performed and the length of data to be processed, each step also consists of different numbers of *StateUpdate* iterations. So, the number of AES rounds are not fixed, it depends on data.
In the following section, there are brief description of AEGIS-128 and it's phases. For a detailed explanation of AEGIS, we refer to [7].

4.1 Notations, Variables and Functions

In this section; notations, variables, constants and functions, which are used in AEGIS, are explained.

4.1.1 Notations

$\&$: bitwise AND	$\oplus$: bitwise XOR
$\|$: concatenation	$\lceil x \rceil$: ceiling operation.

4.1.2 Variables and Constants

P : plaintext

P_i : a 16-byte plaintext block

S_i : state at the beginning of i^{th} step

$S_{i,\,j}$: j^{th} 16-byte element of the state S_i

K_{128} : 128-bit key of AEGIS-128

AD : associated data

AD_i : a 16-byte associated data block

$adlen$: bit length of associated data with $0 \leq adlen < 2^{64}$

IV_{128} : 128-bit initialization vector of AEGIS-128

$const$: a 32-byte constant, which consists of Fibonacci sequence modulo 256, in hexadecimal format; $const$ = 00 ‖ 01 ‖ 01 ‖ 02 ‖ 03 ‖ 05 ‖ 08 ‖ 0d ‖ 15 ‖ 22 ‖ 37 ‖ 59 ‖ 90 ‖ e9 ‖ 79 ‖ 62 ‖ db ‖ 3d ‖ 18 ‖ 55 ‖ 6d ‖ c2 ‖ 2f ‖ f1 ‖ 20 ‖ 11 ‖ 31 ‖ 42 ‖ 73 ‖ b5 ‖ 28 ‖ dd.

$const_0$: first 16 bytes of $const$

$const_1$: last 16 bytes of $const$

C : ciphertext

C_i : a 16-byte ciphertext block

$msglen$: bit length of the plaintext/ciphertext with $0 \leq msglen < 2^{64}$

m_i : a 16-byte data block

T : authentication tag

t : bit length of the authentication tag with $64 \leq t \leq 128$

$u : u = \lceil \frac{adlen}{128} \rceil$

$v : u = \lceil \frac{msglen}{128} \rceil$

4.1.3 Functions

4.1.3.1 AES Round

The AES round function, which consists of *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*, is used in AEGIS.

AESRound(S, K): S and K are 16-byte state and round key, respectively.

4.1.3.2 StateUpdate

The *StateUpdate* function of AEGIS updates state S_i with performing 5, 6 and 8 *AESRound(S, m)* in AEGIS-128, AEGIS-256 and AEGIS-128L, respectively.

4.2 AEGIS-128

In AEGIS-128, key length and initialization vector (IV) are 128 bits. The associated data length and plain text length are less than 2^{64} bits. The authentication tag is less than or equal to 128 bits.

The $\boldsymbol{S_{i+1}} = \textbf{StateUpdate128(}\boldsymbol{S_i, m_i}\textbf{)}$ function of AEGIS-128 updates the 80-byte state S_i as follows:

$S_{i+1,\,0} = AESRound(S_{i,4},\, S_{i,0} \oplus m_i)$

$S_{i+1,\,1} = AESRound(S_{i,0},\, S_{i,1})$

$S_{i+1,\,2} = AESRound(S_{i,1},\, S_{i,2})$

$S_{i+1,\,3} = AESRound(S_{i,2},\, S_{i,3})$

$S_{i+1,\,4} = AESRound(S_{i,3},\, S_{i,4})$

4.3 The Initialization Phase of AEGIS-128

The initialization phase of AEGIS-128 consists of loading 128-bit initialization vector and key into the state and updating state 10 times as follows:

1. Load K_{128} and IV into state S_{-10}:

   $S_{-10,\ 0} = K_{128} \oplus IV_{128}$;

   $S_{-10,\ 1} = const_1$;

   $S_{-10,\ 2} = const_0$;

   $S_{-10,\ 3} = K_{128} \oplus const_0$;

   $S_{-10,\ 4} = K_{128} \oplus const_1$;

2. For i = -10 to -1, run S_{i+1} = StateUpdate128($S_i,\ m_i$) with $m_i = K_{128} \oplus IV_{128}$ for odd i's and $m_i = K_{128}$ for even i's.

4.4 The Authenticated Data Processing Phase of AEGIS-128

After initialization phase, state updated with using associated data AD.

For $i = 0$ to $\lceil \frac{adlen}{128} \rceil$ - 1 run S_{i+1} = StateUpdate128($S_i,\ AD_i$).

If $adlen = 0$, the state will not update. If the last block of AD is not full block, it should be padded with 0 bits.

4.5 The Encryption Phase of AEGIS-128

In this phase each P_i block is encrypted to C_i and meanwhile P_i is used to update state S_i. For $i = 0$ to v-1, perform encryption and state update as follows:

$C_i = P_i \oplus S_{u+i,\ 1} \oplus S_{u+i,\ 4} \oplus (S_{u+i,\ 2} \ \&\ S_{u+i,3})$

S_{u+i+1} = StateUpdate128($S_{u+i},\ P_i$)

where $u = \lceil \frac{adlen}{128} \rceil$ and $v = \lceil \frac{msglen}{128} \rceil$. If $msglen = 0$, there is no encryption and state update. If the last block of P is not full block, zero-padding should be done.

4.6 The Finalization Phase of AEGIS-128

In the finalization phase of AEGIS-128, an authentication tag is generated. Firstly state updated seven more times and than tag is generated from state as follows:

For $i = u{+}v$ to $u{+}v + 6$, perform state update.

$S_{i+1} = \text{StateUpdate128}(S_i,\ tmp)$

where $tmp = S_{u+v,3} \oplus (adlen \parallel msglen)$

Then the authentication tag is generated from the final state S_{u+v+7}:

$T = S_{u+v+7,0} \oplus S_{u+v+7,1} \oplus S_{u+v+7,2} \oplus S_{u+v+7,3} \oplus S_{u+v+7,4}$

CHAPTER V

HIGH PERFORMANCE AES + AEGIS ENCRYPTION CORE ARCHITECTURE

In this chapter, an iterative high performance standalone AES architecture and our combined high performance AES + AEGIS architecture are explained.

5.1 An Iterative Standalone AES Encryption Core Architecture

AES consists of round iterations, and the number of round iterations is 10, 12, 14 for the key sizes of 128, 192, 256, respectively. Each intermediate cipher result is called *state*. A *roundkey* for each round iteration is also generated from the encryption key. Each round consists of SubBytes, ShiftRows, and MixColumns operations performed on state, finally adding the state and *roundkey*, which is called AddRoundKey operation. However, the last round skips MixColumns.

Before introduce to combined AES + AEGIS architecture, it will be more explanatory to give and talk about standalone AES architecture, which is the point of departure. In Figure 8 standalone AES-128 architecture is given. All data paths and registers are 128-bit.

In the following subsections module of given architecture is explained briefly.

5.1.1 Input Logic

The input logic unit performs first AddRoundKey operation during load phase of AES. key_add_en control is enabled when loading plain text and key to registers, so the first AddRoundKey is performed meanwhile.

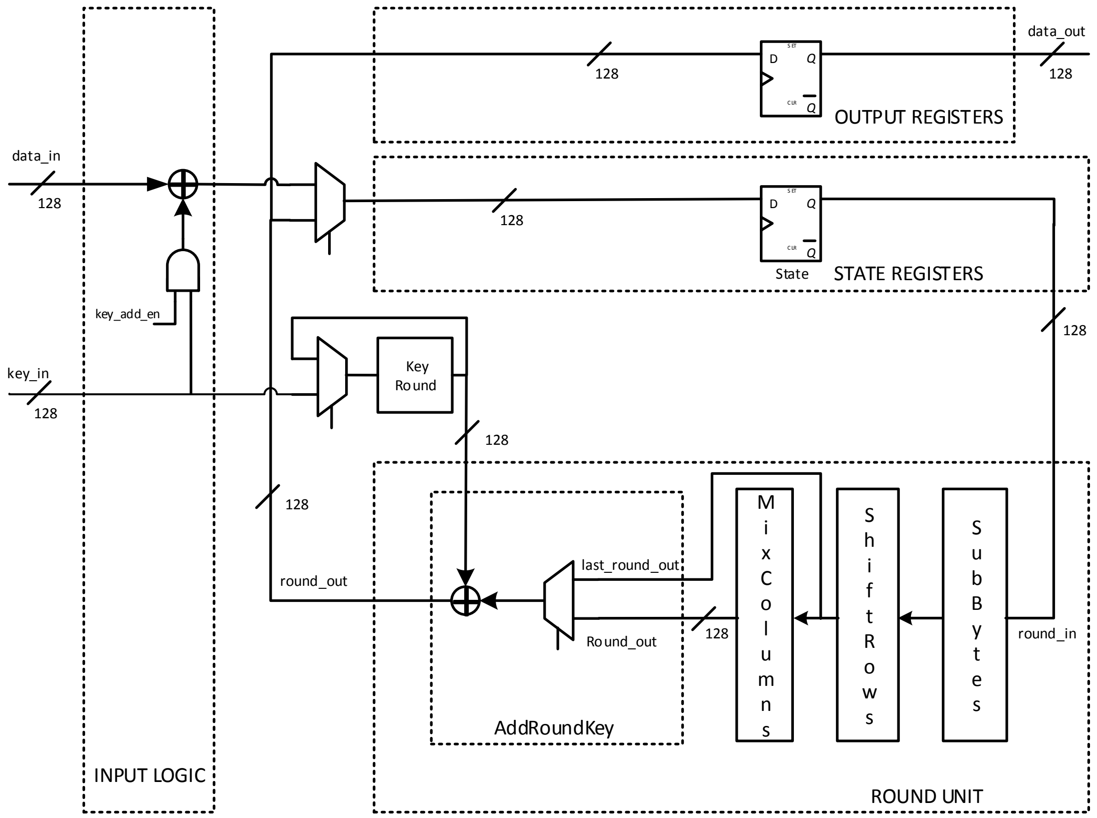

Figure 8: Standalone AES architecture

5.1.2 State Registers

A state of AES-128 stored in 128-bit register. At the beginning plain text is loaded to registers. Each cycle, current state is rounded and new state is produced and it is fed to registers as next state.

5.1.3 Round Unit

This unit performs the AES round operation. An AES round consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations. Since MixColumns is skipped at the last round, ShiftRows output is selected as second operand of AddRoundKey via a 2-to-1 multiplexer. All modules in round unit consists of combinational logics.

5.1.3.1 SubBytes Module

SubBytes hardware module in the round unit is given in Figure 9. Input and output are 128-bit width. Input is grouped into 16 bytes (B_0 to B_{15}) like shown in the Figure 9. Each byte is fed to an identical SubByte module and processed parallel. A SubByte module is shown in Figure 10.

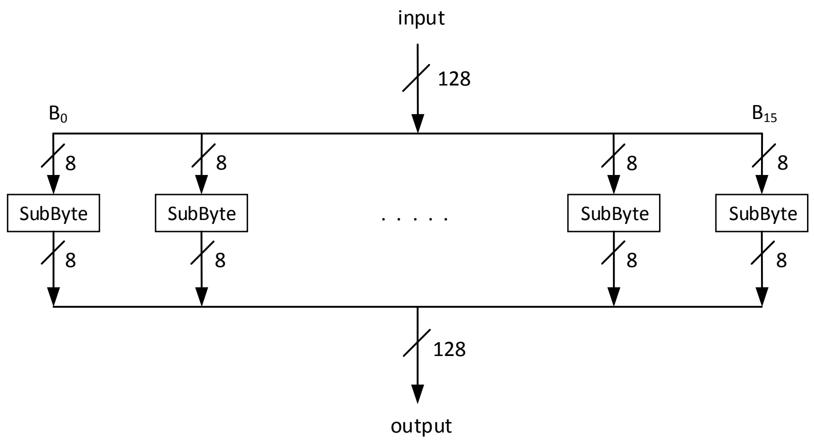

Figure 9: AES SubBytes module

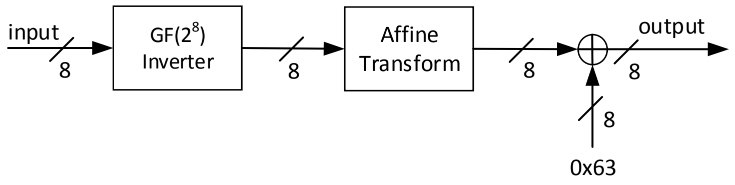

Figure 10: SubByte module

5.1.3.2 ShiftRows Module

The ShiftRows module is given in Figure 11. As mentioned in Section 3.5, it just shifts each row of the state a certain offset.

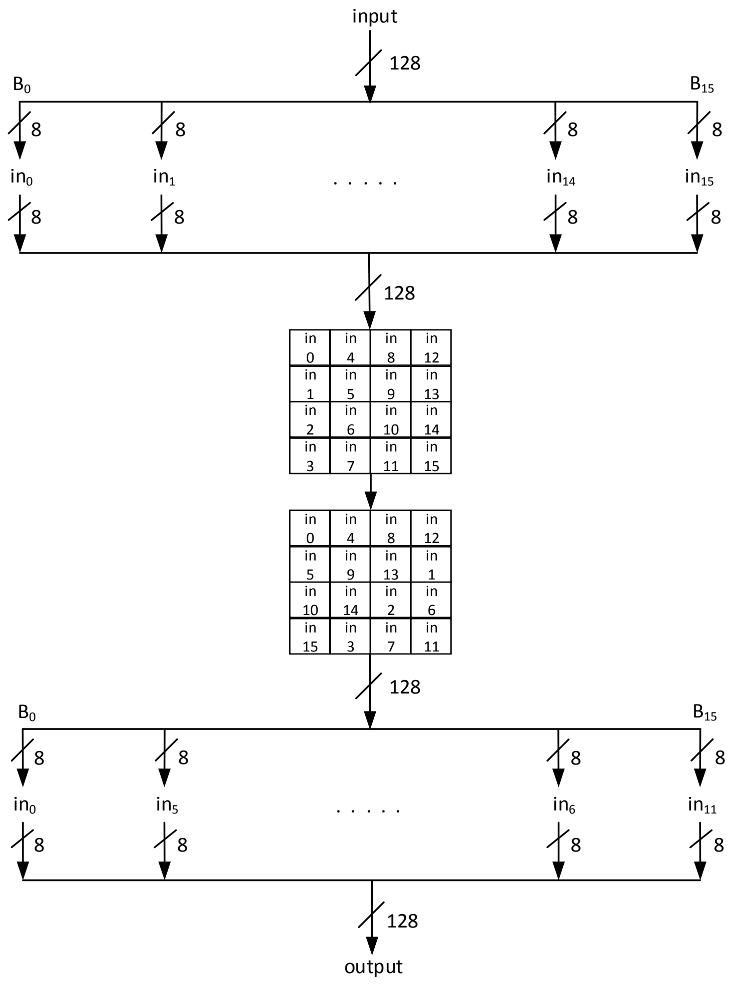

Figure 11: ShiftRows module

5.1.3.3 *MixColumns Module*

The MixColumn hardware module is shown in Figure 13. Firstly it groups 128-bit input to 32-bit columns (Col-0 to Col-3), then identical MixColumn modules are operates on these columns. Inside of a MixColumn module is given in in Figure 12. All arithmetic operations performed in GF(2^8).

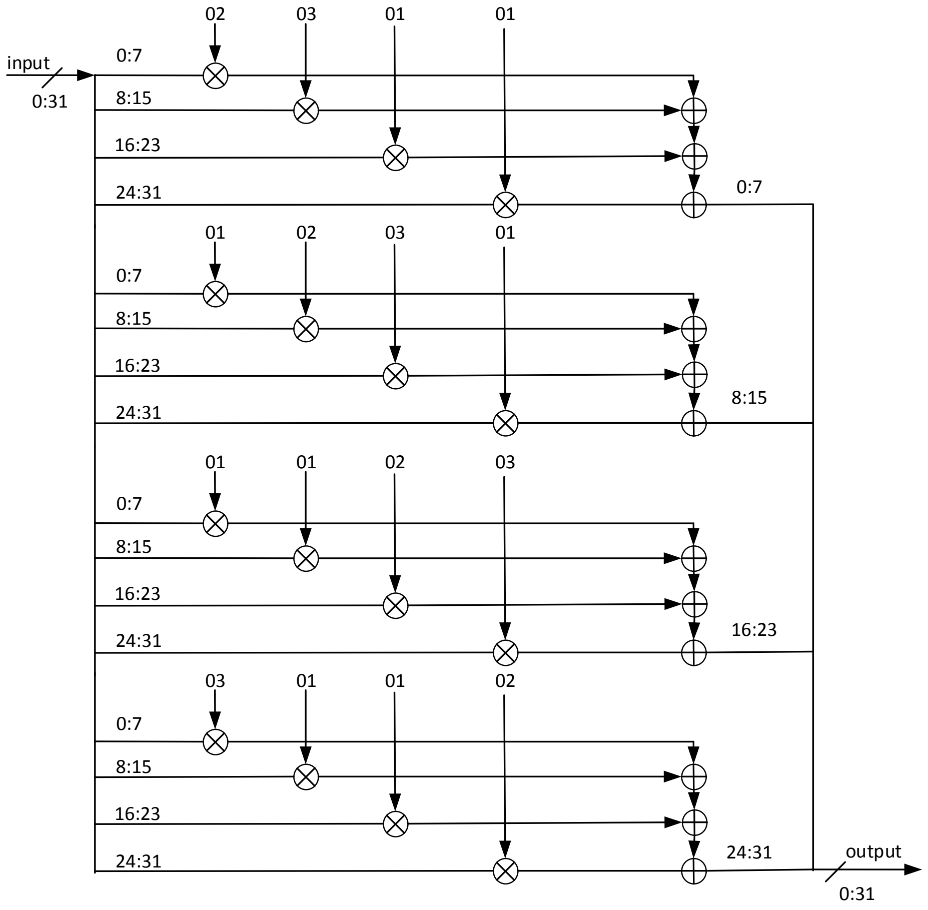

Figure 12: MixColumn module

5.1.4 KeyRound Module

The KeyRound module generates *subkeys* from encryption key. During load phase initial encryption key is load to a 128-bit register. Then each cycle a new round key is generated from current subkey. The KeyRound hardware module is given in Figure 14.**SubWord** is 4-byte version of the SubBytes, it consists of 4 parallel

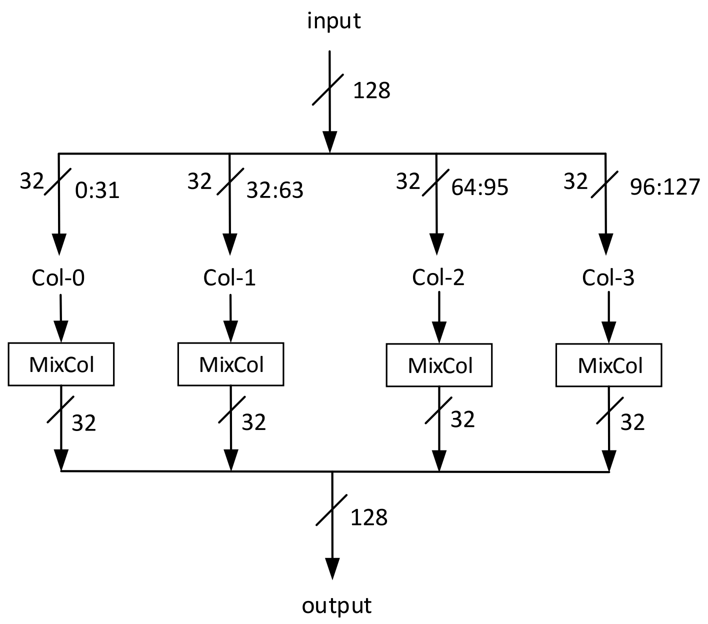

Figure 13: MixColumns module

SubByte modules as shown in Figure 15. **RotWord** shifts bytes of a word to left by one byte as shown in Figure 16. Initially Rcon(0) = 0x01000000, and every step it is updated like this: Rcon(i+1) = 2 $\otimes$ Rcon(i). The $\otimes$ symbol represents GF(2^8) multiplication.

5.1.5 Output Registers

After performing 10 round operations, the cipher text is outputted via 128-bit data_out register. Note that the last round skips MixColumn transformation. In this AES architecture a 128-bit plain text block is encrypted in every ten cycle period, and the next 128-bit plain text block can be fed to the core immediately.

5.2 Combined AES + AEGIS Architecture

In the previous section the standalone AES architecture is explained, which is our starting point for designing combined AES + AEGIS architecture for high performance applications. In this section, our combined AES + AEGIS architecture is explained.

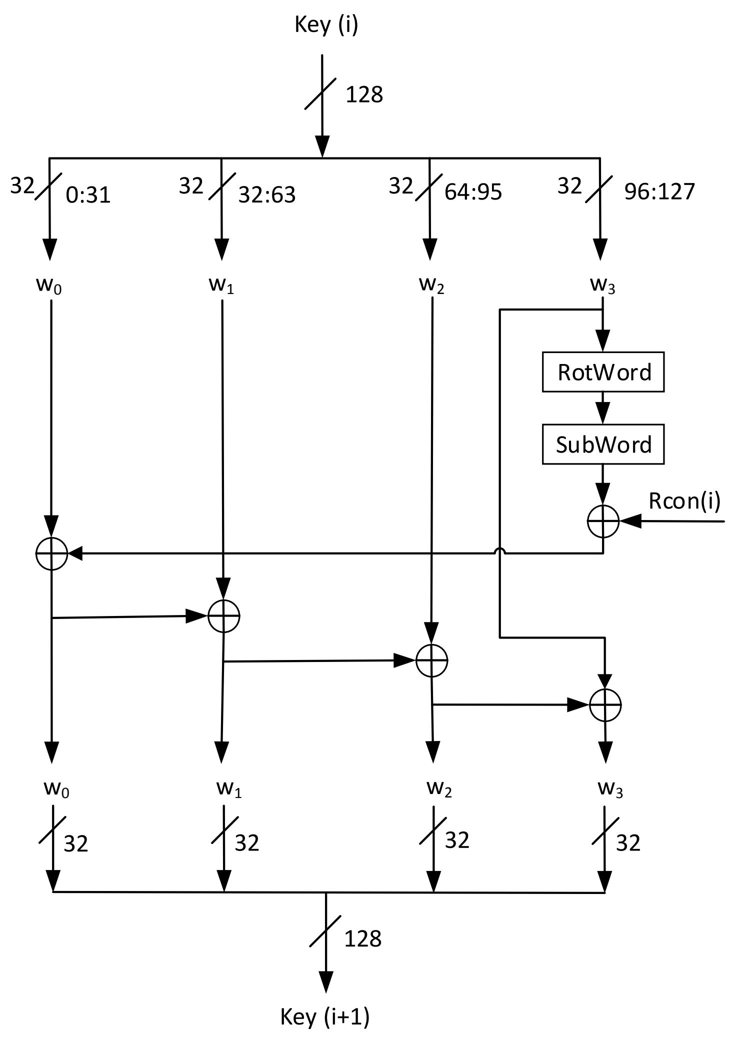

Figure 14: The KeyRound module

We examined the AEGIS and tried to design a new architecture that can perform both AES and AEGIS encryption. To make possible this we made some additions and modifications on standalone AES encryption core. More complex control logic is also designed to control the core. The architecture of our combined AES + AEGIS encryption core is depicted in Figure 17. This core works for 128-bit keys. All data-paths and registers in this design are 128-bit. The core consists of five components: input logic, state registers, round unit, AEGIS finalization-tmp unit, output logic. The core operates in five phases, which are ***load***, ***initialization***, ***associated data processing***, ***encryption*** **and** ***finalization***. Cycle counts for each phase is given

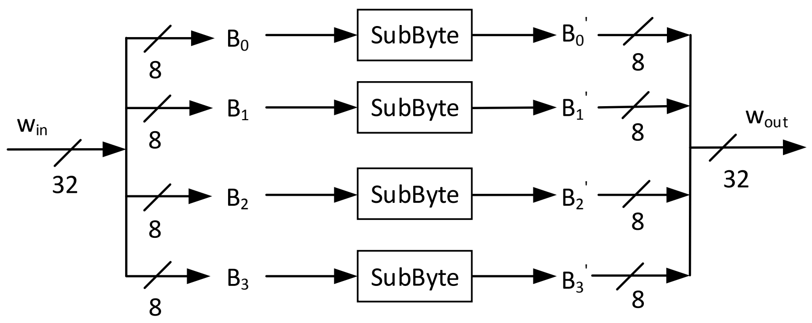

Figure 15: The SubWord module

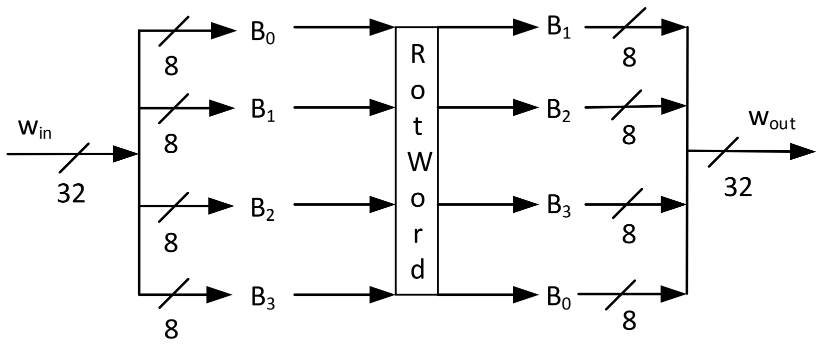

Figure 16: The RotWord module

in Table 2, where $u = \lceil \frac{adlen}{128} \rceil$ and $v = \lceil \frac{msglen}{128} \rceil$.

Table 2: Cycle counts for AEGIS-128 and AES-128 operation phases

Phases	AEGIS-128	AES-128
0: Load	5 cycles	1 cycle
1: Initialization	10×5 cycles	–
2: AD processing	$u \times 5$ cycles	–
3: Encryption	$v \times 5$ cycles	10 cycles
4: Finalization	7×5 cycles	–

5.2.1 Input Logic

The input logic unit performs key addition during load and initialization phases of AES and AEGIS, and input data selection for state registers.

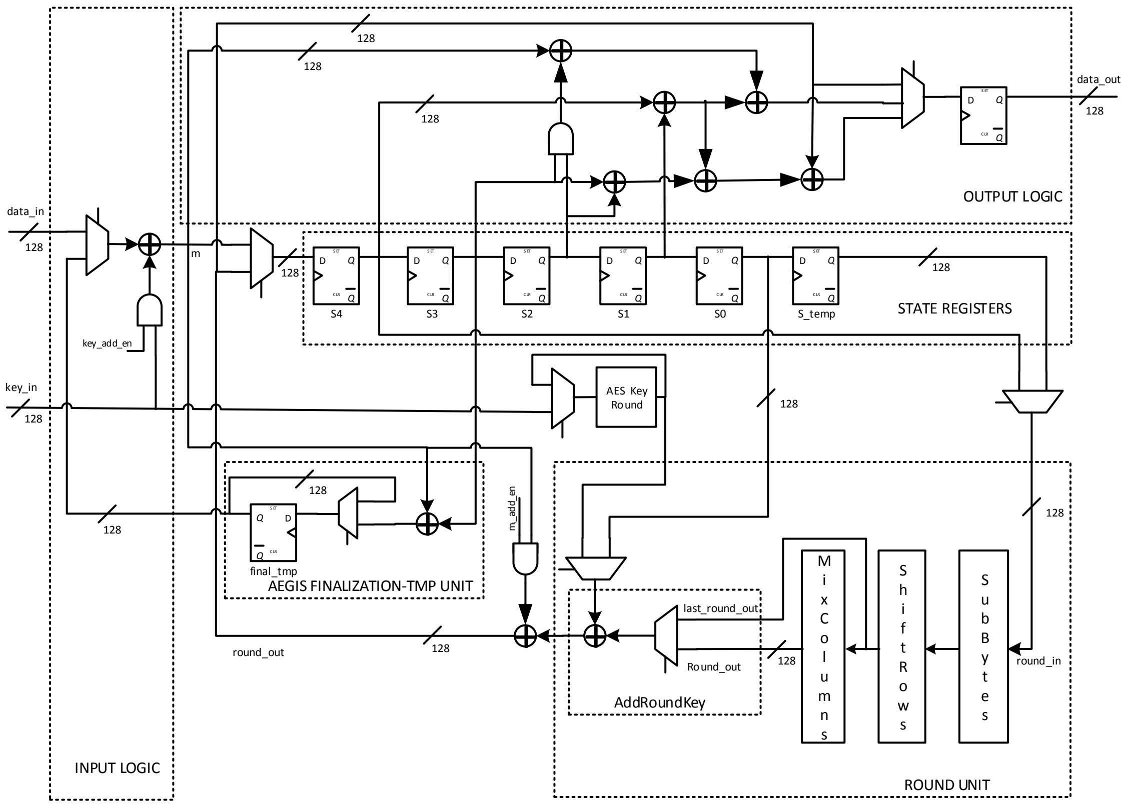

Figure 17: Combined AES + AEGIS encryption core architecture

5.2.1.1 data_in:

The data_in is used for plain text input in AES and initialization vector, const0, const1, associated data, plain text input in AEGIS. Depending on phase, state_update, cycle, AEGIS and AES which data block must be fed to data_in is shown in Table 3.

Table 3: data_in schedule for AES-128 and AEGIS-128 operation phases

phase	state_update	cycle	data_in (AEGIS)	data_in (AES)
0	- (load)	0	init. vector (IV)	plain text
		1	const1	–
		2	const0	–
		3	const0	–
		4	const1	–
1	-10, -8, -6, -4, -2	0	–	–
	-9, -7, -5, -3, -1	0	init. vector (IV)	–
		1-4	–	–
2	all	0	assoc. data (AD)	–
	all	1-4	–	–
3	all	0	plain text	–
	except last	1-4	–	–
	last	4	(adlen ‖ msglen)	–
4	all	all	–	–

5.2.1.2 key_in:

128-bit key input for both AES and AEGIS. In AES, initial encryption key is also fed to the AES KeyRound Unit.

5.2.1.3 m output:

Table 4 shows **m output** of the input logic unit depending on phase, state_update, cycle, AEGIS and AES.

Table 4: The m signal of input logic for AES-128 and AEGIS-128 operation phases

phase	state_update	cycle	m (AEGIS)	m (AES)
0	- (load)	0	key_in ⊕ data_in	key_in ⊕ data_in
		1	data_in	–
		2	data_in	–
		3	key_in ⊕ data_in	–
		4	key_in ⊕ data_in	–
1	-10, -8, -6, -4, -2	0	key_in	–
	-9, -7, -5, -3, -1	0	key_in ⊕ data_in	–
		1-4	–	–
2	all	0	data_in	–
	all	1-4	–	–
3	all	0	data_in	–
	except last	1-4	–	–
	last	4	data_in	–
4	all	0	finalization_tmp	–
	all	1-4	–	–

5.2.2 State Registers

This unit consists of six 16-byte shift registers. First five registers, which are S4, S3, S2, S1 and S0, store a state of AEGIS-128. S_temp is an additional 16-byte register for storing the previous S0. It is a necessity coming from the AEGIS-128 State_Update128 function. As stated in [7], AEGIS State_Update128 function rounds firstly $S_{i,4}$, then it rounds $S_{i,0}$. So we designed our AES + AEGIS encryption core as follows: In the first cycle of each State_Update128 operation, S4, which stores $S_{i,4}$, is fed into the Round Unit. Since $S_{i,0}$ must be rounded in the second cycle of State_Update128, it is shifted from S0 to S_temp at first cycle, and stored in S_temp. Except the first cycle, S_temp always contains the proper one fifth part of a state, which must be fed into Round Unit. Contents of registers, depending on the cycle, are given in Table 5. Register contents are shifted each cycle, and parts of a state is propagated through the S_temp register.

Table 5: State register contents for AES-AEGIS operation cycles

cycle	S4	S3	S2	S1	S0	S_temp
0	$\mathbf{S}_{i,4}$	$\mathrm{S}_{i,3}$	$\mathbf{S}_{i,2}$	$\mathrm{S}_{i,1}$	$\mathbf{S}_{i,0}$	$\mathrm{S}_{i-1,4}$
1	$\mathrm{S}_{i+1,0}$	$\mathbf{S}_{i,4}$	$\mathrm{S}_{i,3}$	$\mathbf{S}_{i,2}$	$\mathrm{S}_{i,1}$	$\mathbf{S}_{i,0}$
2	$\mathbf{S}_{i+1,1}$	$\mathrm{S}_{i+1,0}$	$\mathbf{S}_{i,4}$	$\mathrm{S}_{i,3}$	$\mathbf{S}_{i,2}$	$\mathrm{S}_{i,1}$
3	$\mathrm{S}_{i+1,2}$	$\mathbf{S}_{i+1,1}$	$\mathrm{S}_{i+1,0}$	$\mathbf{S}_{i,4}$	$\mathrm{S}_{i,3}$	$\mathbf{S}_{i,2}$
4	$\mathbf{S}_{i+1,3}$	$\mathrm{S}_{i+1,2}$	$\mathbf{S}_{i+1,1}$	$\mathrm{S}_{i+1,0}$	$\mathbf{S}_{i,4}$	$\mathrm{S}_{i,3}$

5.2.3 The Round Unit

The Round Unit performs the AES round operation. An AES round consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations. In the first cycle (cycle 0) of AEGIS State_Update128, S4 is rounded, whereas S_temp is rounded in all other cycles (cycle 1-4) of State_Update128. Depending on which encryption is performing, AES or AEGIS, subkey or S0 is added to round_out, respectively. It is switched via a 2-to-1 multiplexer as shown in Figure 17.

5.2.4 AEGIS Finalization-Tmp Unit

This unit computes and stores the 16-byte *tmp* value in the finalization step of the AEGIS-128. *tmp* is defined as $\mathrm{S}_{u+v,3} \oplus$ (adlen $\|$ msglen) in [7]. The $\mathrm{S}_{u+v,3}$ is stored in S3 register and the (adlen $\|$ msglen) is fed to the unit via data_in, where the $\|$ symbol represents concatenation. After *tmp* is computed, it is stored in 128-bit final_tmp register, which shown in Figure 17, and the same *tmp* used in all seven State_Update128 iterations of finalization phase.

5.2.5 Output Logic

The Output Logic unit consists of combinational logics as depicted in Figure 17. This unit performs computation of output values *cipher text* and *tag.* The proper output is selected by a multiplexer with respect to the performed encryption (AES or AEGIS) and output type (cipher text or tag). In the case of AEGIS, cipher text is output at the first cycle of each state_update in encryption phase (phase 3). For AES, there is no State_Update128 iteration. However, since ten round iterations of AES are correspond to two state_update of AEGIS (2 x 5 = 10 cycles), the last round of AES corresponds to last round of second state_update of phase 3 and cipher text is output after that round. data_out values are shown in Table 6.

Table 6: data_out schedule for AES and AEGIS

phase	state_update	cycle	data_out (AEGIS)	data_out (AES)
0	all	all	–	–
1	all	all	–	–
2	all	all	–	–
3	all	0	data_in ⊕ S1 ⊕ S4 ⊕ (S2 & S3)	–
	last	4	–	round_out
4	7	4	S4 ⊕ S3 ⊕ S2 ⊕ S1 ⊕ S0	–

CHAPTER VI

LIGHTWEIGHT AES + AEGIS ENCRYPTION CORE ARCHITECTURE

In this chapter, our lightweight AES + AEGIS architecture is introduced. For AES, several lightweight hardware implementations are reported in literature [13] [14] [15]. These are FPGA and ASIC implementations of AES with 8-bit datapaths. Based on our knowledge, the lowest power and lowest area implementation of AES encryption hardware core have been reported in [12], where 8-bit datapaths are employed and one AES round performed in 16 clock cycles. Only 128-bit keys are supported in that architecture. In that implementation, processing a single plain text block takes 176 cycle including loading and unloading. Since loading and unloading can be perform simultaneously with encryption, sequential encryption with same key takes 160 cycles. Combining reported lightweight AES architecture in [12] and our proposed architecture in Chapter 5, we proposed a new lightweight hardware core that performs both AES-128 and AEGIS-128 encryption based on a 1-bit selection input. Basically, we modified some parts of the design in [12] and added some new parts to make AEGIS-128 encryption possible. Our new lightweight design is a compact combination of the design in [12]] and our AES + AEGIS core design approach, which was presented in chapter 5. Our lightweight core also employs 8-bit datapaths, and the effective cycle count for one AES round is 16. Operation phases are same with high performance architecture presented in Chapter 5. Cycle counts for phases are given in Table 7. High-level architecture of our design is depicted in Figure 18. Modules of the architecture are detailed in following sections.

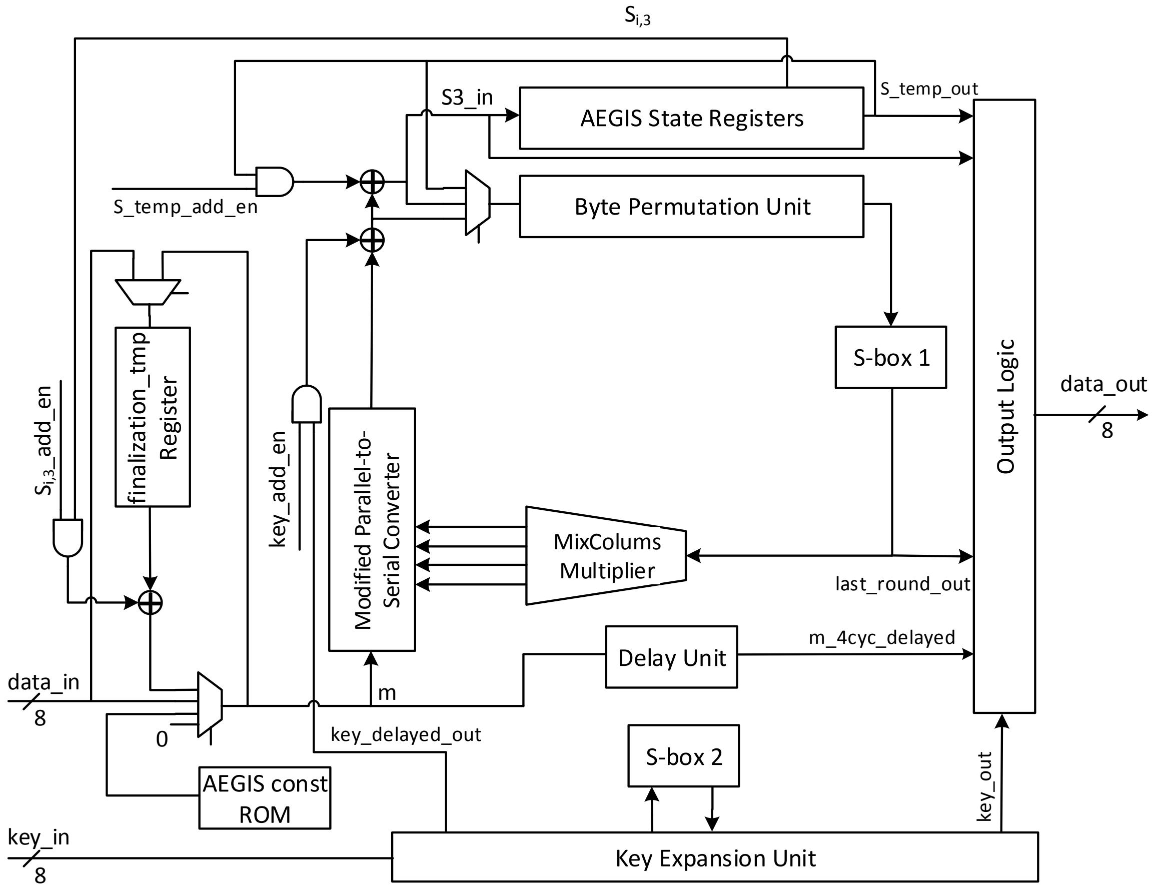

Figure 18: Lightweight AES + AEGIS encryption core architecture

Table 7: Cycle counts for lightweight AEGIS-128 and AES-128 operation phases

Phases	AEGIS-128	AES-128
0: Load	5×16 cycles	16 cycle
1: Initialization	$10 \times 16 \times 5$ cycles	–
2: AD processing	$u \times 16 \times 5$ cycles	–
3: Encryption	$v \times 16 \times 5$ cycles	10×16 cycles
4: Finalization	$7 \times 16 \times 5$ cycles	–

6.1 Top Level I/O Ports

Top level I/O ports of our design are data input, key input and data out. Initialization vector, (adlen ‖ msglen), associated data in AEGIS, plain text in both AES and AEGIS fed to the core as serial bytes via data input port. Encryption key is fed into the core via key input port. Authentication tag in AEGIS, ciphertext in both AES and AEGIS output via data out port. One AES round is performed in 16 clock cycles. Input schedule for data input port is given in Table 8.

Table 8: Input schedule for data_in

phase	state update	round	data input (AEGIS)	data input (AES)
0	- (load)	0	init. vector (IV)	plain text
		1-4	–	–
1	-10	0	(adlen ‖ msglen)	–
	-8, -6, -4, -2	all	–	–
	-9, -7, -5, -3, -1	0	init. vector (IV)	–
	all	1-4	–	–
2	all	0	assoc. data (AD)	–
	all	1-4	–	–
3	all (except last)	0	plain text	–
	all	1-4	–	–
4	all	all	–	–

6.2 S-box Module

The S-box module performs SubByte operation. This module same as the Sub-Byte module depicted in Figure 10 in Section 5.1.3.1. There are two identical S-box in our design same as in [12]. Sbox-1 used for state rounding and Sbox-2 is used for AES key rounding.

6.3 Byte Permutation Unit

The byte permutation unit performs ShiftRows operation and stores 12 bytes of a state. Remaining 4 bytes part of state are processed in other data paths registers. The hardware of byte permutation unit is depicted in Figure 19. It consists of 12 8-bit registers, 3 2-to-1 multiplexer and 1 4-to-1 multiplexer. Table 9 shows how input, contents of registers and output is changing during one round operation. It also explains how ShiftRows operation performed by this module. **b***i*'s and **N***i*'s represent ith byte of current state and next state, respectively. Boxed byte in each row is selected for output via 4-to-1 multiplexer. Bold bytes are coming from R0 register in previous row, which are not outputted yet.

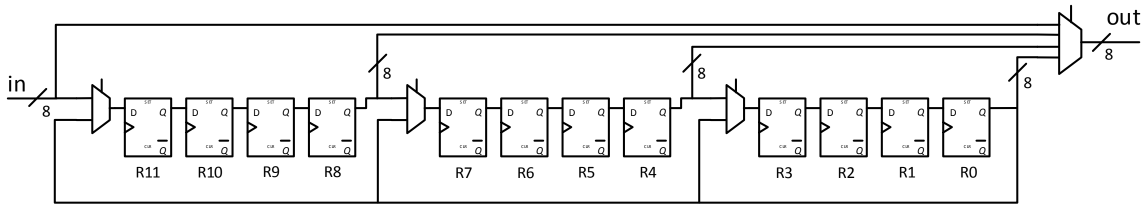

Figure 19: Byte permutation unit

6.4 MixColumns Multiplier

MixColumns multiplication performed by this module. A column of a state is fed to the unit byte by byte, so one column multiplication is completed in 4 clock cycle. A complete MixColumns transformation takes 16 cycle. The mixcolumns multiplier is depicted in Figure 20. MixColumns transformation for a column performed by

Table 9: Register and output schedule for byte permutation unit

cycle	in	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0	out
0	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	$\boxed{b0}$	b0
1	b13	b12	b11	b10	b9	b8	b7	b6	$\boxed{b5}$	b4	b3	b2	b1	b5
2	b14	b13	b12	b11	$\boxed{b10}$	b9	b8	b7	b6	**b1**	b4	b3	b2	b10
3	$\boxed{b15}$	b14	b13	b12	b11	**b2**	b9	b8	b7	b6	b1	b4	b3	b15
4	N0	**b3**	b14	b13	b12	b11	b2	b9	b8	b7	b6	b1	$\boxed{b4}$	b4
5	N1	N0	b3	b14	b13	b12	b11	b2	$\boxed{b9}$	b8	b7	b6	b1	b9
6	N2	N1	N0	b3	$\boxed{b14}$	b13	b12	b11	b2	**b1**	b8	b7	b6	b14
7	N3	N2	N1	N0	$\boxed{b3}$	**b6**	b13	b12	b11	b2	b1	b8	b7	b3
8	N4	N3	N2	N1	N0	**b7**	b6	b13	b12	b11	b2	b1	$\boxed{b8}$	b8
9	N5	N4	N3	N2	N1	N0	b7	b6	$\boxed{b13}$	b12	b11	b2	b1	b13
10	N6	N5	N4	N3	N2	N1	N0	b7	b6	**b1**	b12	b11	$\boxed{b2}$	b2
11	N7	N6	N5	N4	N3	N2	N1	N0	$\boxed{b7}$	b6	b1	b12	b11	b7
12	N8	N7	N6	N5	N4	N3	N2	N1	N0	**b11**	b6	b1	$\boxed{b12}$	b12
13	N9	N8	N7	N6	N5	N4	N3	N2	N1	N0	b11	b6	$\boxed{b1}$	b1
14	N10	N9	N8	N7	N6	N5	N4	N3	N2	N1	N0	b11	$\boxed{b6}$	b6
15	N11	N10	N9	N8	N7	N6	N5	N4	N3	N2	N1	N0	$\boxed{b11}$	b11

multiplication with coefficients, adding and cyclically shifting the intermediate results. Addition controlled with **add_en** signal. After completion of a column multiplication, the 4 bytes result is fed to the modified parallel-to-serial converter. Register contents for a column multiplication, which takes 4 cycle, is given in Table 10.

Table 10: MixColumns multiplier registers contents for a column multiplication

Registers	t_0	t_1	t_2	t_3
R0	in_0	$in_0 \oplus in_1$	$\{03\}in_0 \oplus in_1 \oplus in_2$	$\{02\}in_0 \oplus \{03\}in_1 \oplus in_2 \oplus in_3$
R1	in_0	$\{03\}in_0 \oplus in_1$	$\{02\}in_0 \oplus \{03\}in_1 \oplus in_2$	$in_0 \oplus \{02\}in_1 \oplus \{03\}in_2 \oplus in_3$
R2	$\{03\}in_0$	$\{02\}in_0 \oplus \{03\}in_1$	$in_0 \oplus \{02\}in_1 \oplus \{03\}in_2$	$in_0 \oplus in_1 \oplus \{02\}in_2 \oplus \{03\}in_3$
R3	$\{02\}in_0$	$in_0 \oplus \{02\}in_1$	$in_0 \oplus in_1 \oplus \{02\}in_2$	$\{03\}in_0 \oplus in_1 \oplus in_2 \oplus \{02\}in_3$

6.5 *Modified Parallel-to-Serial Converter*

The parallel-to-serial converter module presented in [12] is a very simple module. Basically, it is used for converting the parallel coming output of MixColumns

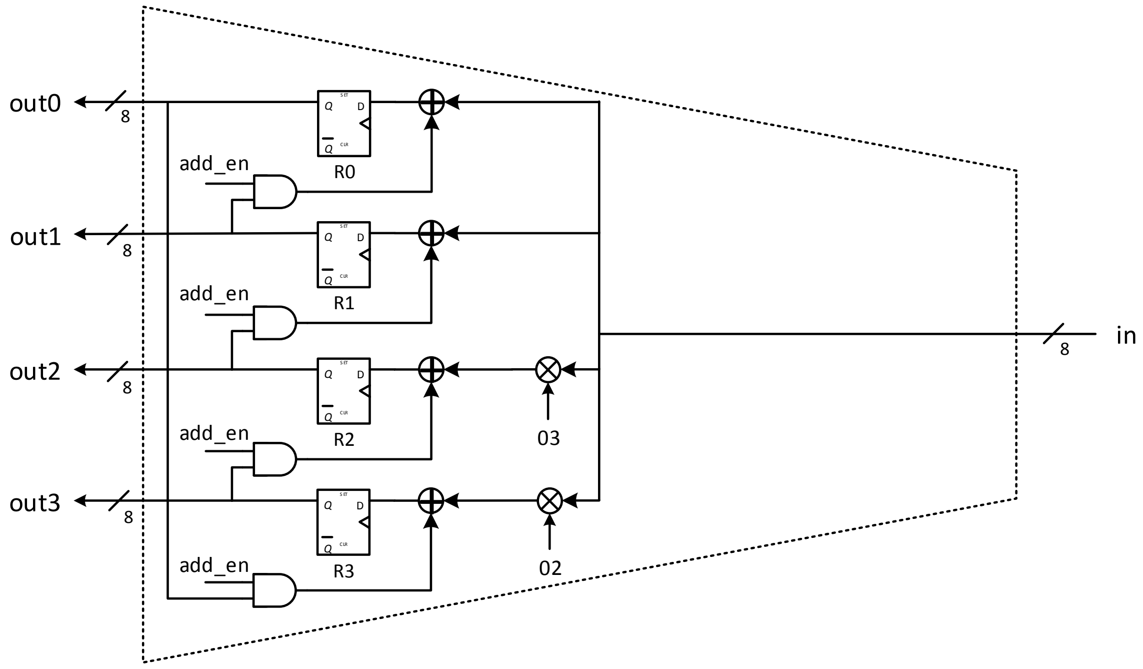

Figure 20: MixColumns multiplier

multiplier to serial bytes. It selects and shifts the data coming from the data_in port while data feeding to the data_in port. Our module is very similar to the parallel-to-serial converter presented in [12] with some modification. However, since our core supports also AEGIS-128 authenticated encryption, we made some modifications. In AEGIS, associated data, plain text, and finalization tmp is added to $S_{i,0}$ in proper state_update128 operation. Since the rounded part of an AEGIS state is transferred from MixColumns multiplier to parallel-to-serial converter, adding m to $S_{i,0}$ is performed by modified parallel-to-serial converter. The module is depicted in Figure 21.

6.6 AEGIS State Registers

This unit stores an AEGIS-128 state partially . S3, S2, S1 and S0 consists of 16 8-bit registers. At the beginning of each state update, S3, S2, S1, and S0 store $S_{i,3}$, $S_{i,2}$, $S_{i,1}$ and $S_{i,0}$, respectively. Each clock cycle, register contents are shifted and bytes of AEGIS state is propagated through S_temp_out. $S_{i,4}$ is stored in the byte permutation unit and the modified parallel-to-serial converter partially. These registers are used

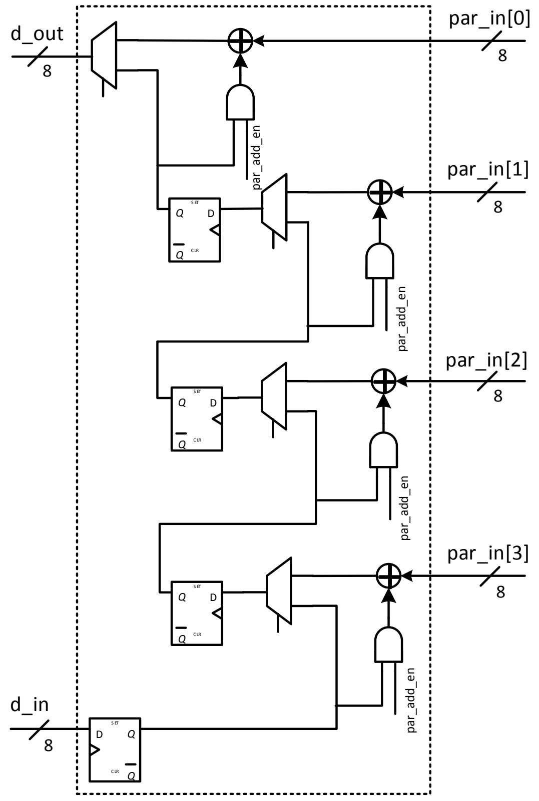

Figure 21: Modified parallel-to-serial converter

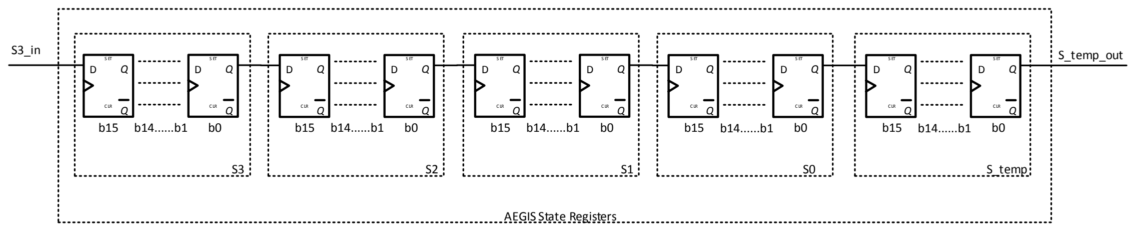

Figure 22: AEGIS state registers

for the same logical reason, which was mentioned in our high performance AES + AEGIS hardware encryption core in Chapter 5. The module depicted in Figure 22.

6.7 Key Expansion Unit

The key expansion unit generates *subkeys* from the encryption key in AES mode. The key expansion unit in our design is almost the same as the architecture presented in [12]. Since there is no key rounding in AEGIS, our key expansion unit works like a ring shift register when AEGIS is employed. Figure 23 shows key expansion unit

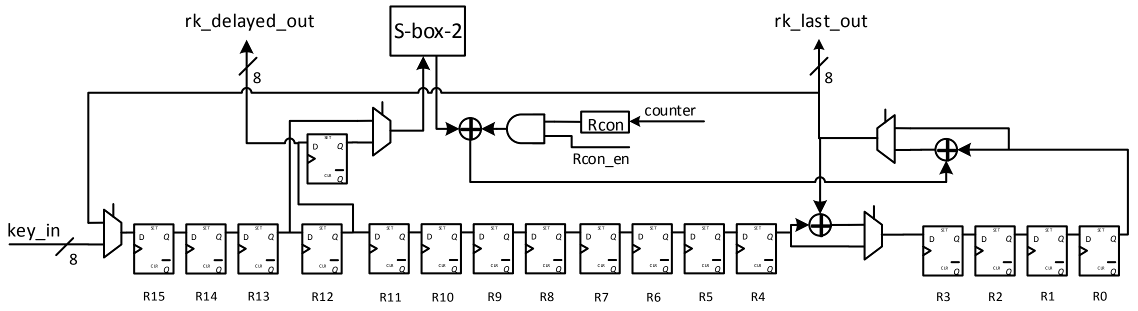

Figure 23: Key expansion unit

circuit details. During the load phase, initial encryption key is load to 16 8-bit register (R0 to R15). After the encryption key is loaded once into the unit, if the core is working in AES mode, the unit performs regular key rounding operation, otherwise it just shifts the register contents. Since it is the last register output fed into first register input in AEGIS mode, encryption key is shifted in a ring. The rk_delayed_out output is 4 cycle delayed output of key/subkey, which aligns the key with parallel-to-serial converter output delay caused by MixColumns multiplier. The rk_last_out is for last round in AES, which skips MixColumns multiplication. For more information about the key expansion unit, we refer to [12].

6.8 AEGIS const ROM

The AEGIS const ROM stores 32 bytes const value, which is the Fibonacci sequence module 256. First 16 bytes of it is called *const0* and the last 16 bytes is called *const1*. During the initialization phase of AEGIS, it is fed into the encryption core via multiplexer in the input and $S_{-10,\,1}$, $S_{-10,\,2}$, $S_{-10,\,3}$, $S_{-10,\,4}$ initialized like in Section 4.3.

6.9 AEGIS finalization_tmp Register

This unit stores 16-byte AEGIS finalization *tmp*, which is used during state updates in finalization phase. The *tmp* computed at the beginning of finalization phase, then it is stored and added to first round output of each state_update. Before the finalization phase, this unit stores (adlen ∥ msglen). As shown in Table 8, (adlen ∥ msglen) is fed to the core at the first round of phase 1. By using (adlen ∥ msglen), control module of core computes u and v, which are number of State_Update128 iterations in AD Processing phase (phase 2) and encryption phase (phase 3). In the first round of AEGIS finalization phase, to compute the finalization *tmp* $S_{i,3}$ is exored with (adlen ∥ msglen) and the result stored again in finalization *tmp* register unit. *tmp* is outputted and added to state first round of each state update in finalization phase of AEGIS.

6.10 Delay Unit

This unit consists of four 8-bits back to back registers. It just delays its input 4 clock cycles. Since the latency of MixColums Multiplier is 4 clock cycles, this delay unit aligns the input coming from data input and MixColumns Multiplier output for calculations in the output logic.

6.11 Output Logic

The output logic unit is designed to perform cipher text calculation in the encryption phase (phase 3) for both encryption algorithms and the AEGIS tag calculation in the finalization phase (phase 4). Data output schedule is given in Table 11. This unit same as the output logic in Section 5.2.5, but datapaths are 8-bit.

Table 11: Output schedule for data_out

phase	state update	round	data_out (AEGIS)	data_out (AES)
0	all	all	–	–
1	all	all	–	–
2	all	all	–	–
3	all	0	m_4cyc_delayed ⊕ S1 ⊕ S3_in ⊕ (S2 & S3)	–
	last	4	–	rnd_last_out ⊕ rk_last_out
4	7	4	S3_in ⊕ S3 ⊕ S2 ⊕ S1 ⊕ S0	–

CHAPTER VII

CONCLUSION

We describe both high performance architecture and lightweight architecture cores in Verilog HDL at register transfer level. Then synthesize our implementations using UMC 90 nm low-leakage standard cell library and Cadence RTL Compiler. Resulted cell counts, area and the number of gate equivalent (GE) are reported in Table 12.

Table 12: Synthesis results

Implementation	Frequency	Cells	Area	Gates	Cycles per block	Throughput
High Perf. AES	91 MHz	5606	40137	12885	11	1059 Mbps
High Perf. AES + AEGIS	91 MHz	8062	61525	19619	11 (AES)	1059 Mbps (AES)
Lightweight AES	100 KHz	1026	8805	2807	160	80 Kbps
Lightweight AES + AEGIS	100 KHz	3386	29293	9340	160 (AES)	80 Kbps (AES)

The high performance AES + AEGIS core occupies a total area of 19.6K GE, while standalone version occupies only 12.8K GE. Both of them can run up to a maximum frequency of 91 MHz. This corresponds to a maximum throughput of 1163 Mbps for AES mode. Since cycles per block in AEGIS depends on associated data length and plain text length, it is impossible to give a fixed throughput value. But, we assume both associated data length and plain text length are 128-bit, we can say that cycles per block are 100 for AEGIS.

Since the lightweight version was targeted for lightweight applications, we synthesized it for a fixed target frequency of 100 KHz. We did not test its highest frequency. At this frequency, it offers a throughput of 80 Kbps. The lightweight AES + AEGIS core occupies a total area of 9.3K GE, while standalone version occupies

2.8K GE. The area is tripled with respect to standalone version.

In high performance case, the area of the combined architecture is only 52.2% higher than that of a standalone AES module, while for the lightweight case, the area is tripled. For both cases, there is no loss in terms of throughput. These results support the claims of the designers of AES-based AE cipher schemes in general.

Our future work possibly involve implementing our architectures using other cell libraries as well as on various FPGA platforms, in order to verify our initial observations.

Another important future work could be adding power consumption figures for each module and analyzing them.

Furthermore, we are planning to investigate ways of integrating other AES-based AE schemes in our architecture with minimal additional resource usage.

Bibliography

[1] H. Liddell, *Greek - English Lexicon: Abridged from Liddell & Scott's Greek - English Lexicon*. Oxford University Press, 1984.

[2] C. Paar and J. Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.

[3] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin, Heidelberg, New York: Springer Verlag, 2002.

[4] "Formal Specification of the CCM Mode of Operation," 2005.

[5] "Information Technology - Security Techniques - Authenticated Encryption," 2009.

[6] A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser, "Lightweight AES-Based Authenticated Encryption," in *Proceedings of Fast Software Encryption (FSE)*, (Singapore), March 2013.

[7] H. Wu and B. Preneel, "AEGIS: A Fast Authenticated Encryption Algorithm." Cryptology ePrint Archive, Report 2013/695, 2013. `http://eprint.iacr.org/`.

[8] A. Brokalakis, A. Kakarountas, and C. Goutis, "A High-throughput Area Efficient FPGA Implementation of AES-128 Encryption," *IEEE Workshop on Signal Processing Systems Design & Implementation*, p. 116, 2005.

[9] X. Zhang and K. Parhi, "High-speed VLSI Architectures for the AES Algorithm.," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 957 – 967, September 2014.

[10] M. Liberatori, F. Otero, J. Bonadero, and J. Castineira, "AES-128 Cipher High Speed, Low Cost FPGA Implementation," *Southern Conference on Programmable Logic*, pp. 195–198, 2007.

[11] R. V. Kshirsagar and M. V. Vyawahare, "FPGA Implementation of High Speed VLSI Architectures for AES Algorithm," in *Proceedings of International Conference on Emerging Trends in Engineering and Technology (ICETET)*, pp. 239–242, IEEE, 2012.

[12] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and Implementation of Low-area and Low-power AES Encryption Hardware Core," in *Proceedings of The EUROMICRO Conference on Digital System Design (DSD)*, (Washington, DC, USA), pp. 577–583, IEEE Computer Society, 2006.

[13] T. Good and M. Benaissa, "AES on FPGA from the Fastest to the Smallest.," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, vol. 3659 of *Lecture Notes in Computer Science*, pp. 427–440, Springer, 2005.

[14] S. M. Farhan, S. A. Khan, and H. Jamal, "An 8-bit Systolic AES Architecture for Moderate Data Rate Applications," *Microprocess. Microsyst.*, vol. 33, pp. 221–231, May 2009.

[15] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen, "AES Implementation on A Grain of Sand," *IEEE Transactions on Information Security*, vol. 152, pp. 13 – 20, 2005.

[16] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong Authentication for RFID Systems Using the AES Algorithm," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 357 – 370, Springer, 2004.

[17] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists," *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 545–557, 2001.

[18] M. McLoone and J. V. McCanny, "Rijndael FPGA Implementations Utilising Look-Up Tables," *Journal of VLSI Signal Processing Systems*, vol. 34, pp. 261–275, July 2003.

[19] K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä, "A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, (New York, USA), pp. 207–215, ACM, 2003.

[20] N. Iyer, P. Anandmohan, D. Poornaiah, and V. Kulkarni, "High-throughput, Low-cost, Fully Pipelined Architecture for AES Crypto Chip," in *Proceedings of Annual IEEE India Conference*, p. 1, IEEE, 2006.

[21] D. A. R. Wallace, *Groups, Rings, and Fields.* Springer-Verlag, 1998.

[22] G. L. Mullen, *Handbook of Finite Fields.* CRC Press, 2013.

[23] N. I. of Standards and Technology, "Advanced Encryption Standard," *NIST FIPS Pub 197*, 2001.

[24] "Caesar: Competition for authenticated encryption: Security, applicability, and robustness," 2014. `http://competitions.cr.yp.to/caesar.html/`.

VITA

Furkan Şahin was born in Mersin. He received the BS degree in Electronics Engineering from Istanbul Technical University in 2010. He was an MSEE student and graduate assistant in nEMESysLab at Özyeğin University under the supervision of professors H. Fatih Uğurdağ and Tolga Yalçın between 2011 and 2014. He was partly supported by Vestek Electronics R&D, a subsidiary of Vestel, besides graduate assistantship at Özyeğin University. His research interests include cryptographic reconfigurable hardware design, computer arithmetic, embedded systems, machine vision and image processing.