A RECONFIGURABLE AND SCALABLE EFFICIENT ARCHITECTURE FOR AES

KE LI
Bachelor of Science, University of Electronic Science and Technology of China, 2003

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

For my family, who offered me unconditional love and support throughout the course of this thesis.

Abstract

A new 32-bit reconfigurable FPGA implementation of AES algorithm is presented in this thesis. It employs a single round architecture to minimize the hardware cost. The combinational logic implementation of S-Box ensures the suitability for non-Block RAMs (BRAMs) FPGA devices. Fully composite field $GF((2^4)^2)$ based encryption and keyschedule lead to the lower hardware complexity and convenience for the efficient subpipelining. For the first time, a subpipelined on-the-fly keyschedule over composite field $GF((2^4)^2)$ is applied for the all standard key sizes (128-, 192-, 256-bit). The proposed architecture achieves a throughput of 805.82Mbits/s using 523 slices with a ratio throughput/slice of 1.54Mbps/Slice on Xilinx Virtex2 XC2V2000 ff896 device.

Acknowledgments

I would like to express many thanks to my supervisor Dr. Hua Li, for his invaluable advice and ideas on the research and also for his devotion of time to me during this program. His support and expertise resolved many hurdles that I encountered throughout the research.

I am also grateful to other committee members Dr. Howard Cheng and Dr. Gongbing Shan for their advice.

Finally, I would like thank my parents for their support of me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cryptography is of importance in digital communications systems. The security aspects of many applications such as Automated Teller Machines (ATMs), e-commerce, internet bank services depend on various cryptographic schemes.

A symmetric-key cryptography algorithm, Data Encryption Standard (DES), has been the encryption standard since 1977. It has been widely used and no attack better than the brute force search has been discovered. But its 56-bit key size has been criticized since its inception. 3DES with triple key size of DES offers higher security but it is inefficient in software, because DES was primarily designed for hardware implementations [30].

In 2001, the National Institute of Standards and Technology (NIST) announced the approval of the Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), FIPS-197 [20]. This standard specifies the Rijndael algorithm [7] as an FIPS-approved symmetric encryption algorithm that may be used by U.S. government organizations (and others) to protect sensitive information.

As a replacement of DES, AES is presently widely used in both software and hardware implementations. Hardware approaches are attractive because it provides better throughput as well as higher physical security. Besides, the byte-wise arithmetic in AES gives hardware approaches more convenience. There are mainly two categories of hardware implementations: Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA). Compared with ASIC, FPGA becomes more and more popular because of its scalability, re-programmability and obvious advantage on time-to-market [19].

1.1 Motivation

The standard announced by NIST [20] indicates that AES is a block cipher with 128-bit block size and 128-, 192-, 256-bit key sizes. These three key sizes are specified for various security levels. The capability to deal with all key sizes makes reconfigurability an important feature of AES implementations.

Numerous FPGA [5, 9, 22, 23, 34] and ASIC [2, 25, 28] implementations of the AES have been presented and evaluated. To date, most implementations feature high speed and high cost suitable for high-end applications only. Fully unrolled scheme makes a convenient platform for pipelining technology to get efficient area cost and high throughput by unfolding all the ten (128-bit key) rounds on the device, which is applied in literature [8, 9, 11, 17, 34].

The issue of secure communication in computing restricted environments, such as Personal Digital Assistants (PDAs), wireless devices, and many other embedded devices, has become more important recently. In order to apply AES in these devices, the AES implementations must be cost efficient. An opposite approach to fully unrolled scheme is to implement a single round unit on hardware [1, 2, 26, 27, 31]. When no further optimization effort is made, a block of data needs ten (128-bit key) cycles to go through encryption. The economic area cost is obtained by sacrificing the speed.

In this thesis, a compact design of AES with low hardware cost and adequate throughput is proposed and implemented in a non-BRAM FPGA.

1.2 32-bit Subpipelined Architecture

The following list summarizes the major contributions in this thesis.

- **32-bit Single Round Unit**: By extending one cycle's job to ten cycles (128-bit key), single round unit requires approximately 1/10 hardware area as fully unrolled scheme; by chopping a block data (128-bit) to four words, theoretically, a 32-bit single round unit costs 1/40 hardware area as the common 128-bit fully unrolled scheme as in [8, 9, 11, 17, 34]. Nevertheless, when 32-bit datapath is used, the shiftrows transformations can not be simply implemented by rewiring. We use the column fashion shiftrows which naturally cuts one round unit into four substages.

- **Complete Composite Field Based AES**: In a non-BRAM design, combinational logic is the approach used for subbytes, also known as S-Box. It is the most costly transformation in AES, in both time and area aspects. Rijmen [24] suggested an alternative approach to calculate multiplicative inverses in S-Box. Since then, the relevant research has proved that the composite field $GF((2^4)^2)$ based arithmetic provides the least gate count and the shortest critical path for calculating multiplicative inverse of a byte, which is the key step in S-Box. This conversion involves an isomorphic map function before and after inversion in each round. As in [9, 11, 28, 31, 34], when key size is 128-bit, it needs ten map functions for each block (128-bit) from finite field to composite field and ten inverse map functions for encryption. If key generator, which also has S-Boxes, is included, another ten mappings and ten inverse mappings are needed. To save the overhead caused by mapping, our design converts the whole AES algorithm from $GF(2^8)$ to $GF((2^4)^2)$, which needs only one forward mapping before the initial round and one backward mapping after the final round. Only one forward mapping is needed for the keyschedule.

- **Subpipelined On-the-fly Keyschedule and Encryptor**: On-the-fly keyschedule supports instant key changing. The previous works of [1, 2, 4, 9, 12, 14, 17, 33, 26, 28, 34] applied the on-the-fly key generator, but only [1, 2, 17] integrate on-the-fly keyschedule for all three key sizes (128, 192, 256-bit). These three designs employ subpipelining to optimize throughput/area ratio. However, none of them uses it in keyschedule. When pipelining and on-the-fly keyschedule are both employed in an AES implementation, the keyschedule must be synchronized with the cipher because they share the same clock. The designs in [34, 26] made a subpipelined keyschedule, but they only support 128-bit key size.

1.3 Thesis Outline

Chapter 2 introduces the mathematical background of finite fields which are relevant to AES. We also present the definition of composite fields in this chapter.

Chapter 3 gives an overview of AES standard. We focus on encryption and keyschedule in this thesis. However, the complete AES standard, including decryption, is presented in this chapter.

Chapter 4 describes the proposed architecture in detail. The formulas for the non-trivial transformations in field $GF((2^4)^2)$ are presented. The keyschedules for three key sizes are demonstrated in figures.

Chapter 5 presents the implementation and compares the proposed architecture with the previous designs.

Chapter 6 provides conclusion of the design.

Chapter 2

Mathematical Background

This chapter introduces the mathematical background of AES. ***Finite Fields***, also referred to as ***Galois Fields***, is the arithmetic basis of AES. The Rijndael algorithm [7] is derived from the finite field $GF(2^8)$. C. Paar [21] demonstrated that by decomposing field $GF(2^8)$ into composite field $GF((2^4)^2)$, we can make hardware implementations consuming less area. The following sections introduce the relevant properties and definitions in finite field $GF(2^8)$ and composite field $GF((2^4)^2)$. All statements are given without proof, but they are referred to the appropriate literature.

2.1 Finite Fields

This section introduces the definition of finite fields, followed by the basic AES mathematical representations and operations over finite field $GF(2^8)$. We start with the definition of *group*.

Definition 2.1 [21] A set G together with a binary operation $G \times G \longrightarrow G$ is called a ***group*** if the following condition are satisfied:

- The binary operation is associative: $(a \circ b) \circ c = a \circ (b \circ c)$, for all $a, b, c \in G$;
- There is an identity element $e \in G$ such that $a \circ e = e \circ a$, for all $a \in G$;
- For any element $a \in G$, there exists an inverse element $a' \in G$ such that $a \circ a' = a' \circ a = e$.

If a *group* satisfies the additional condition that $a \circ b = b \circ a$, for all $a, b \in G$, the *group* is *commutative* or *abelian*.

Definition 2.2 [29] Let F be a set of elements on which two binary operations, called addition "+" and multiplication "$\cdot$", are defined. The set F together with the two binary operation $+$ and $\cdot$ is a ***field*** if the following conditions are satisfied:

- F is a *commutative group* under addition +. The identity element with respect to addition is called the *zero element* or the *additive identity* of F and is dentoed by 0;
- The set of nonzero elements in F is a *commutative group* under multiplication $\cdot$. The *identity element* with respect to multiplication is called the *unit element* or the *multiplicative identity* of F and is denoted by 1;
- Multiplication is distributive over addition; that is, for any three elements a, b and c in F: $a \cdot (b+c) = a \cdot b + a \cdot c$.

Fields with a finite number of elements are called ***Finite*** or ***Galois Fields***, denoted as $GF(q)$. Here, q is the number of field elements, which is also the ***order*** of $GF(q)$. The ***extension field*** is of order q^m and is denoted by $GF(q^m)$ [21], which can be constructed by an ***irreducible polynomial*** $P(x)$ [29] of degree m over $GF(q)$. The field $GF(q)$ is a ***subfield*** of $GF(q^m)$ [16]. Every element of field $GF(q^m)$ can be represented as polynomial with a maximum degree of $m-1$ over $GF(q)$, which is the residue modulo $P(x)$. Hence $P(x)$ determines the arithmetic operations in field $GF(q^m)$.

2.1.1 AES Arithmetic over Field $GF(2^8)$

AES is built on the specific finite field $GF(q^m)$, when $q = 2, m = 8$. $GF(2^8)$ is an extension field of $GF(2)$. We use the same notations and conventions as the AES specification in [20], except the multiplication denotation of two elements in $GF(2^8)$. Instead of using $\bullet$, we use $\otimes$, for a consistency with the figures in the subsequent chapters. The basic unit for processing in the AES algorithm is a *byte*. Each 8-bit sequence of input, output, states, cipherkey or roundkeys is treated as a single entity.

A. 3 Notations of An Element

1. Binary notation: A concatenation of 8 individual bits. The bit value is 0 or 1.

$$\{a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0\}$$

2. Polynomial notation: Because $GF(2^8)$ is the extension field of $GF(2)$, its element can be represented as a polynomial over $GF(2)$ (Equation (2.1)). Bit a_i is the coefficients of the polynomial with the value of 0 or 1.

$$a(x) = \sum_{i=0}^{7} a_i x^i = a_7 x^7 + a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \qquad (2.1)$$

3. Hexadecimal notation: $\{AB\}$, A denotes $a_7 a_6 a_5 a_4$ in hexadecimal representation, B denotes $a_3 a_2 a_1 a_0$ in hexadecimal representation.

For example, $\{01100011\}$ (binary notation) can be represented as $x^6 + x^5 + x + 1$ (polynomial notation) and $\{63\}$ (hexadecimal notation).

B. Addition

The addition of two elements in $GF(2^8)$ is adding their corresponding polynomial coefficients modulo 2, which is the XOR-operation denoted by $\oplus$. For $a(x), b(x) \in GF(2^8)$ ($a(x)$ is in Equations (2.1); $b(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0$), it can be implemented by Equation (2.2)

$$a(x) \oplus b(x) = \sum_{i=0}^{7} a_i x^i \oplus \sum_{i=0}^{7} b_i x^i = \sum_{i=0}^{7} (a_i \oplus b_i) x^i \qquad (2.2)$$

C. Multiplication

For $a(x), b(x) \in GF(2^8)$, $\otimes$ is the multiplication operation in $GF(2^8)$, $\times$ is the normal polynomial multiplication.

Polynomial (2.3) is the irreducible polynomial used in AES. The multiplication of $a(x)$ and $b(x)$ is done by multiplying these two polynomials followed by a modular reduction over $m(x)$ (Equation (2.4)). The modular reduction is made to ensure that the result is an element in $GF(2^8)$.

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.3}$$

Given $q(x) \in GF(2^8)$, $q(x) = q_7x^7 + q_6x^6 + q_5x^5 + q_4x^4 + q_3x^3 + q_2x^2 + q_1x + q_0$, we have:

$$q(x) = a(x) \otimes b(x) = (a(x) \times b(x))\ mod\ m(x) \tag{2.4}$$

FIPS gives an efficient method to do multiplication in $GF(2^8)$ in [20]. It uses the multiplication by x, which is denoted as $xtimes(a(x))$. Given: $t(x) \in GF(2^8)$, $t(x) = t_7x^7 + t_6x^6 + t_5x^5 + t_4x^4 + t_3x^3 + t_2x^2 + t_1x + t_0$, we have:

$$\begin{aligned}
&t(x) = xtimes(a(x)) = (a(x) \times x)\ mod\ m(x) \\
&-------------------- \\
&t_0 = a_7,\ t_1 = a_0 \oplus a_7,\ t_2 = a_1,\ t_3 = a_2 \oplus a_7 \\
&t_4 = a_3 \oplus a_7,\ t_5 = a_4,\ t_6 = a_5,\ t_7 = a_6
\end{aligned} \tag{2.5}$$

In Equation (2.5), $t(x)$ is the multiplication result of $a(x)$ and x in $GF(2^8)$. It is calculated by multiplying $a(x)$ with x, followed by the modular reduction over $m(x)$. Based on Equations (2.5), we can use Equation (2.6) to conduct the multiplication in $GF(2^8)$

(Equation (2.4)).

$$q(x) = \sum_{i=0}^{7} P_i(x) \times b_i$$
$$-------------------- \qquad (2.6)$$
$$P_i(x) = xtimes(P_{i-1}(x)) \quad (P_0(x) = a(x))$$

In Equation (2.6), the partial multiplications ($P_i(x)$) is performed first, followed by adding the corresponding coefficients. Bit b_i is the coefficient in $b(x)$, which are 0 or 1.

D. Multiplicative Inverses

$$\forall a \in GF(2^8) \setminus \{0\} : a \otimes a^{-1} = \{1\} \qquad (2.7)$$

a^{-1} is the multiplicative inverse of a in $GF(2^8)$. A popular algorithm for inversion is the Extended Euclidean Algorithm [18], but it is not suitable for hardware implementation because of its high hardware complexity.

2.2 Composite Fields

Two Galois Fields of the same order are isomorphic, but they may have different hardware complexity which depends on the representations of their field elements. Green and Taylor [10] introduced a certain type of extension fields called *composite field*, which can simplify field operations in AES arithmetic.

Definition 2.4

We call two pairs

$$\{GF(2^n), Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i, q_i \in GF(2)\}$$
$$\{GF((2^n)^m), P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i, p_i \in GF(2)\}$$

a ***composite field*** if

- $GF(2^n)$ is constructed from $GF(2)$ by $Q(y)$;
- $GF((2^n)^m)$ is constructed from $GF(2^n)$ by $P(x)$.

Composite field is denoted by $GF((2^n)^m)$. A composite field $GF((2^n)^m)$ is isomorphic to the field $GF(2^k),\ k = nm$ [15].

2.2.1 AES Arithmetic over Composite Field $GF((2^4)^2)$

The specific composite field used in this thesis is $GF((2^4)^2)$, which is isomorphic to field $GF(2^8)$ ($k = 8, n = 4, m = 2$). Taking field $GF(2^8)$ as a quadratic extension of the field $GF(2^4)$, an element $a \in GF(2^8)$ is represented as a linear polynomial with coefficients in $GF(2^4)$.

A. Notation

Wolkerstorfer et al. introduced a ***two-term polynomial*** in [32], which is the representation of $GF((2^4)^2)$ used in the thesis.

$$a \cong a_h x + a_l,\ a \in GF(2^8),\ a_h, a_l \in GF(2^4) \tag{2.8}$$

The two-term polynomial $a_h x + a_l$ is an isomorphic representation of a. Hence, all mathematical operations applied to elements of $GF(2^8)$ can also be computed in this representation.

B. Addition

Adding the corresponding coefficients.

$$(a_h x + a_l) \oplus (b_h x + b_l) = (a_h \oplus b_h)x + (a_l \oplus b_l) \tag{2.9}$$

C. Multiplication

There are two irreducible polynomials needed for the two-term polynomial multiplication: $n(x)$ (Equations (2.10)) and $m(x)$ (Equations (2.11)).

$$n(x) = x^2 + \{1\}x + \{E\} \quad (\{E\}\ denotes\ \text{"1110"}) \tag{2.10}$$

$$m(x) = x^4 + x + 1 \tag{2.11}$$

Equation (2.10) is used to reduce the result to a two-term polynomial. The coefficients of $n(x)$ are written in hexadecimal notation which are elements in $GF(2^4)$ (Section (2.1.1)). Multiplication of two-term polynomials is denoted by $\otimes$. Normal polynomials multiplication is denoted by $\times$. Multiplying two two-term polynomials, followed by a modular reduction over $n(x)$, is described by Equations (2.12).

$$(a_h x + a_l) \otimes (b_h x + b_l) = ((a_h x + a_l) \times (b_h x + b_l))\ mod\ n(x) \tag{2.12}$$

Equation (2.11) is used to ensure that, the result of multiplication in subfield $GF(2^4)$ (Equation (2.13)), where $(a'(x), b'(x) \in GF(2^4))$, is an element of $GF(2^4)$.

$$a'(x) \otimes b'(x) = (a'(x) \times b'(x))\ mod\ m(x) \tag{2.13}$$

These two irreducible polynomials $n(x)$ and $m(x)$ are chosen by Wolkerstorfer et al. [32] to optimize the arithmetic.

D. Multiplicative Inverses

A multiplication of a two-term polynomial with its inverse yields the 1-element of the field $GF((2^4)^2)$

$$(a_h x + a_l) \otimes (a'_h x + a'_l) = \{0\}x + \{1\} \tag{2.14}$$

where $a_h, a_l, a'_h, a'_l \in GF(2^4)$.

$$(a_h x + a_l)^{-1} = (a'_h x + a'_l) = (a_h \otimes d)x + (a_h \oplus a_l) \otimes d \tag{2.15}$$

where $d = ((a_h^2 \otimes \{E\}) \oplus (a_h \otimes a_l) \oplus a_l^2)^{-1} = ((a_h^2 \otimes \{e\}) \oplus ((a_h \oplus a_l) \otimes a_l))^{-1}$ ($\oplus$ is addition in $GF(2^4)$; $\otimes$ is multiplication in $GF(2^4)$).

This multiplicative inversion equation is proposed by Wolkerstorfer et al. [32]. Reconfiguration of this equation can provide good quality for subpipeling. We will explain this in Section (4.4.2).

Chapter 3

AES Algorithm

This chapter introduces the AES algorithm presented by NIST in 2001 [20].

The AES algorithm, also known as the Rijndael algorithm is the encryption standard designed by two Belgian cryptographers John Daemen and Vincent Rijmen [7]. AES is a symmetric-key cipher where both the encryptor and decryptor use the same key. It is an iterative algorithm. Each iteration is called a round. According to NIST, AES is a symmetric block cipher with block size of 128-bit and three key sizes of (128-, 192-, or 256-bit). The AES parameters depend on the key size (Table (3.1)):

- Nk is the number of 32-bit words comprising the cipher key;
- Nb is the number of 32-bit words comprising a data block, which is four in AES standard;
- Nr is the number of rounds which is 10, 12 or 14 for AES-128, AES-192 and AES-256, respectively.

The internal operations of AES are performed on a 4×4 matrix of bytes, termed the ***state*** (Figure (3.1)). An individual byte of the state is referred as $S_{r,c}$ (r represents the row

Table 3.1: Key-Block-Round Combinations [20]

	Key Length (Nk words)	Block Size (Nb words)	Number of Rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

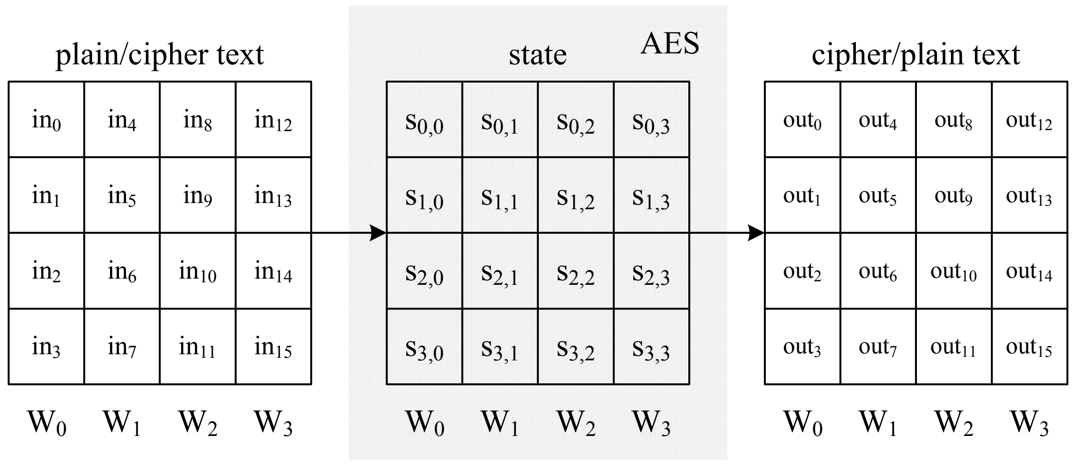


Figure 3.1: State array input and output

number and c represents the column number: $0 \leq r < 4, 0 \leq c < 4$). A word W_i $(0 \leq i < 4)$ consists of the four bytes of column i.

AES runs iteratively on four transformations (inv-/subbytes, inv-/shiftrows, inv-/mixcolumns and addroundkey) with different sequence in encryption and decryption. Figure (3.2) illustrates the basic architecture of AES. In the initial round ($r = 0$), only addroundkey is performed; in the final round ($r = Nr$), it skips inv-/mixcolumns. The keyschedule module expands cipherkey to $(Nr + 1) \times 4$ words of roundkeys. Each round applies a unique 128-bit roundkey in the addroundkey operation.

3.1 Subbytes and Invsubbytes

Inv-/subbytes is the only non-linear transformation in AES which is also called S-Box.

A. Subbytes – Uses an S-Box to perform a non-linear byte-by-byte substitution of the state.

S-Box is a 16×16 matrix containing all possible 256 8-bit values.

Consider a byte $\{x_7x_6x_5x_4x_3x_2x_1x_0\}$. Subbytes transformation has two steps:

1. $\{x'_7x'_6x'_5x'_4x'_3x'_2x'_1x'_0\}$ is its multiplicative inverse in $GF(2^8)$ field, modulo the irre-

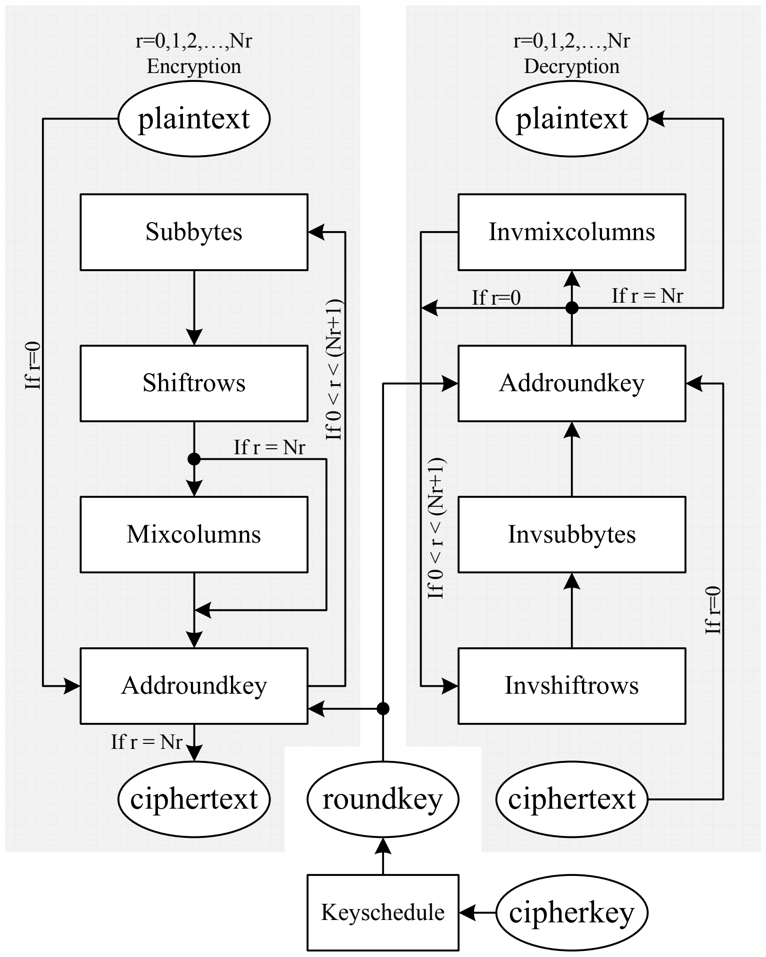


Figure 3.2: AES architecture

ducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$; $\{00000000\}$'s multiplicative inverse in $GF(2^8)$ field is itself;

2. An affine transformation over $GF(2)$ (Equation (3.1)) is conducted on the inverse, which is the result of the first step.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ x'_5 \\ x'_6 \\ x'_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{3.1}$$

Figure (3.3) shows the S-Box diagram:

$$\begin{tikzcd} \{x_7x_6x_5x_4x_3x_2x_1x_0\} \arrow[r] & \boxed{\text{S-Box}} \arrow[r] & \{y_7y_6y_5y_4y_3y_2y_1y_0\} \end{tikzcd}$$

Figure 3.3: AES S-box

B. Invsubbytes – Uses an inverse S-Box (IS-Box) to perform a non-linear byte-by-byte substitution of the state.

Considering a byte $\{y_7y_6y_5y_4y_3y_2y_1y_0\}$. Inverse subbytes transformation has two steps:

1. The inverse affine transformation over $GF(2)$ (Equation (3.2)) is performed first

$$\begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ x'_5 \\ x'_6 \\ x'_7 \end{bmatrix} = \begin{bmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.2}$$

2. $\{x_7x_6x_5x_4x_3x_2x_1x_0\}$ is the multiplicative inverse of $\{x'_7x'_6x'_5x'_4x'_3x'_2x'_1x'_0\}$ in $GF(2^8)$ field, modulo the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$; $\{00000000\}$ is mapped onto itself.

Figure (3.4) shows the IS-Box diagram:

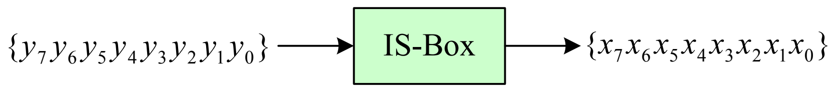

$\{y_7y_6y_5y_4y_3y_2y_1y_0\}$
IS-Box
$\{x_7x_6x_5x_4x_3x_2x_1x_0\}$

Figure 3.4: AES IS-box

3.2 Shiftrows and Invshiftrows

This transformation circularly shifts each row of the state to the left on encryption or to the right on decryption. The top row of the state is denoted as $row(0)$ and the bottom row is denoted as $row(3)$. The shift offset of each row corresponds to the row number.

A. Shiftrows – Each row of the state is left shifted cyclically a certain number of bytes.

Performs i-byte circular left shift to $row(i) (i = 0, 1, 2, 3)$. Figure (3.5) illustrates the shiftrows operation.

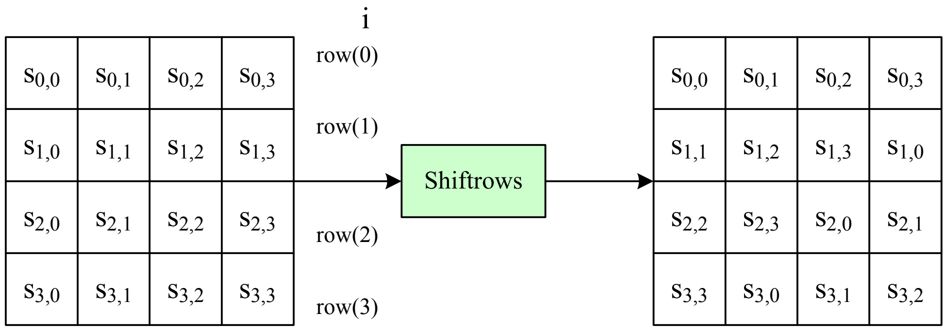

i
row(0)
row(1)
row(2)
row(3)
Shiftrows
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 3.5: AES Shiftrows

B. Invshiftrows – Each row of the state is right shifted cyclically a certain number of bytes.

Performs i-byte circular right shift to $row(i)(i = 0,1,2,3)$. Figure (3.6) illustrates the invshiftrows operation.

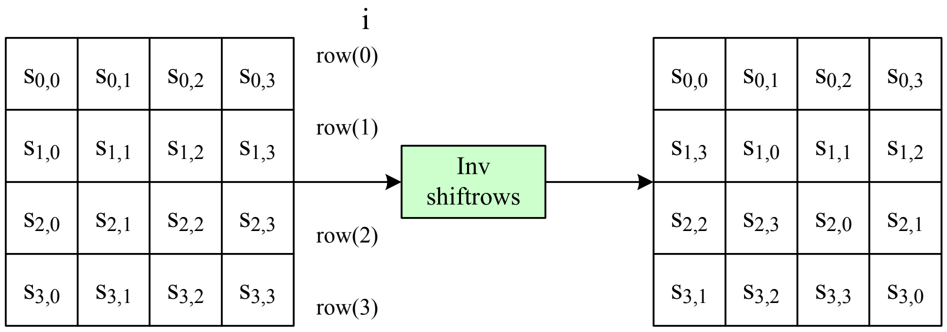

i
row(0)
row(1)
row(2)
row(3)
Inv shiftrows
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,3}$ $s_{1,0}$ $s_{1,1}$ $s_{1,2}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,1}$ $s_{3,2}$ $s_{3,3}$ $s_{3,0}$

Figure 3.6: AES Invshiftrows

3.3 Mixcolumns and Invmixcolumns

This transformation treats each column of the state as a four-term polynomial over $GF(2^8)$ and transforms each column to a new one by multiplying it with a constant polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ modulo $x^4 + 1$. The inverse mixcolumns operation is a multiplication of each column with $b(x) = a^{-1}(x) = \{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$

modulo x^4+1.

A. MixColumns – Left multiplies the state with a mixcolumns matrix.

Mixcolumns transformation gives each byte of a column a new value based on all four bytes in that column. In matrix form, the mixcolumns can be expressed as:

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

B. Invmixcolumns – Left multiplies the state with a invmixcolumns matrix.

In matrix form, the invmixcolumns can be expressed as:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{3.4}$$

3.4 Addroundkey

The addroundkey is a simple logical XOR of the current state with a roundkey which is generated by the keyschedule.

Addroundkey – The state is XORed with the 128-bit roundkey (Equation (3.5)).

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \oplus roundkey = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \tag{3.5}$$

3.5 Keyschedule

Keyschedule – Derives roundkeys from the cipherkey. It consists of two steps:

1. Key Expansion - Uses the AES Key Expansion Algorithm (Figure (3.7)) to generate $4 \times (N_r + 1)$ words of roundkeys ($W_0, W_1, ..., W_{4(N_r+1)-1}$). The cipherkey is divided into Nk words used as the first Nk roundkeys. Keyschedule repeats to generate the rest roundkeys.

2. Roundkey Selection - The first 4 roundkeys are the first 4 words, the second 4 roundkeys are the second 4 words, etc. Each roundkey has 128 bits: $roundkey(i) = (W_{4i}, W_{4i+1}, W_{4i+2}, W_{4i+3})$.

Figure (3.8) shows keyschedule's architecture which generates roundkeys for AES-128, AES-192 and AES-256.

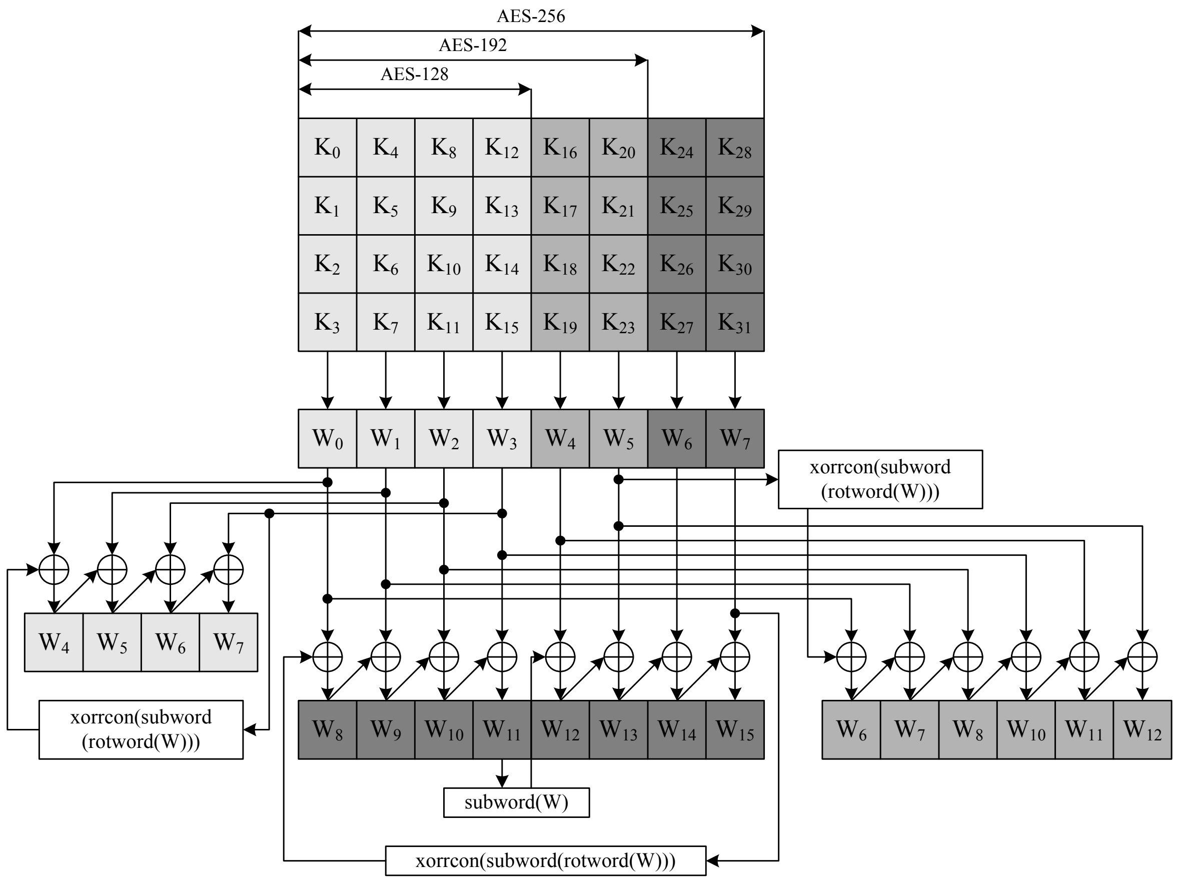
- **Rotword**: One-byte circular left shift on a word. For example, word (a,b,c,d) becomes (b,c,d,a).

- **Subword**: Using S-Box to perform a byte substitution on each byte.

- **Xorrcon**: XORing with a round constant $rcon[j], j = 1,2,\cdots,Nr$.
  $rcon[j] = (RC[j], 0, 0, 0)$, with $RC[1] = 1, RC[j] = 2 \cdot RC[j-1]$ and with multiplication defined over the field $GF(2^8)$.

```
////////////////////////////////////////////////////////////
//Input: key[4*Nk] (Cipherkey)
//Output: w[4*(Nr+1)] (Nr+1 roundkeys)
//Nk and Nr is specified in Table (3.1)
////////////////////////////////////////////////////////////

KeyExpansion(byte key[4*Nk], word w[4*(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
```

Figure 3.7: Pseudo Code for Key Expansion [20]



AES-256
AES-192
AES-128
K_0 K_4 K_8 K_{12} K_{16} K_{20} K_{24} K_{28}
K_1 K_5 K_9 K_{13} K_{17} K_{21} K_{25} K_{29}
K_2 K_6 K_{10} K_{14} K_{18} K_{22} K_{26} K_{30}
K_3 K_7 K_{11} K_{15} K_{19} K_{23} K_{27} K_{31}
W_0 W_1 W_2 W_3 W_4 W_5 W_6 W_7
xorrcon(subword (rotword(W)))
W_4 W_5 W_6 W_7
xorrcon(subword (rotword(W)))
W_8 W_9 W_{10} W_{11} W_{12} W_{13} W_{14} W_{15}
subword(W)
xorrcon(subword(rotword(W)))
W_6 W_7 W_8 W_{10} W_{11} W_{12}

Figure 3.8: AES Keyschedule

Chapter 4

Reconfigurable and Compact Architecture of the AES

In this chapter, the reconfigurable and compact AES architecture is proposed. We introduce the contributions in detail, followed by the four transformations (shiftrows, subbytes, mixcolumns and addroundkey). The three keyschedules with different key sizes (128-, 192-, 256-bit) are explained individually.

4.1 32-bit Single Round Unit

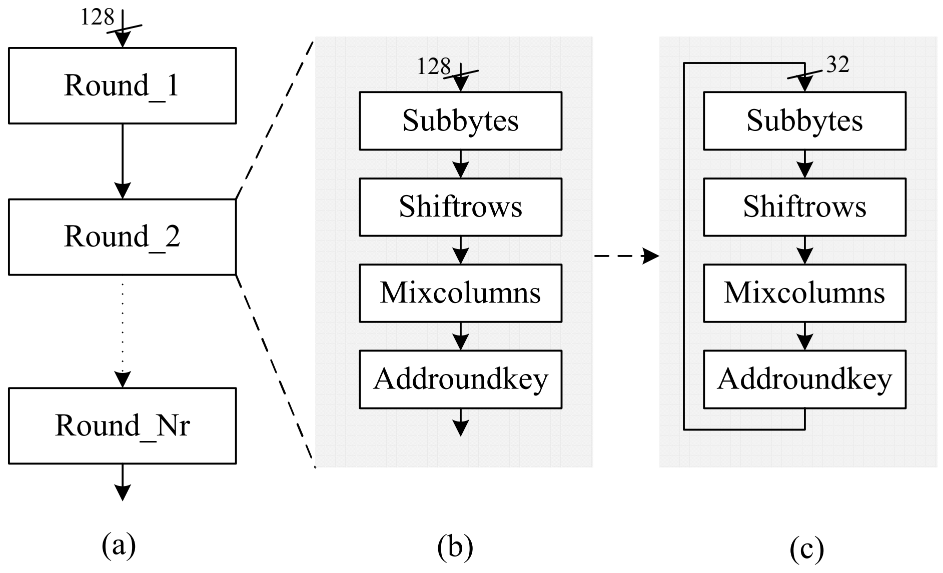


Figure 4.1: Unfolded Architecture(a) - Single Round Unit(b) - 32-bit Single Round Unit(c)

Roll unfolded architecture (Figure (4.1(a))) is widely used to achieve high speed. It processes several blocks of data during one clock cycle by implementing more than one round units on the hardware. The more round units the architecture implements, the higher the hardware cost. The opposite scheme, which is called the *single round unit* architecture (Figure (4.1(b))), can be applied to simplify the hardware complexity. Instead of unfolding all the round units in devices, it implements a single round unit which costs approximately $1/Nr$ area as the unfolded scheme by sacrificing the speed (Figure (4.1(a))).

Both Figure ((4.1(a)) and ((4.1(b)) use 128-bit data path. Sticking to the goal of making a compact design, we propose a 32-bit single round unit (Figure (4.1 (c))). It needs four iterations to perform a round on a block (128-bit). This 32-bit data path scheme saves about 75% hardware, compared with the 128-bit single round unit (Figure (4.1 (b))).

4.2 Full Composite Field Encryptor and Keyschedule

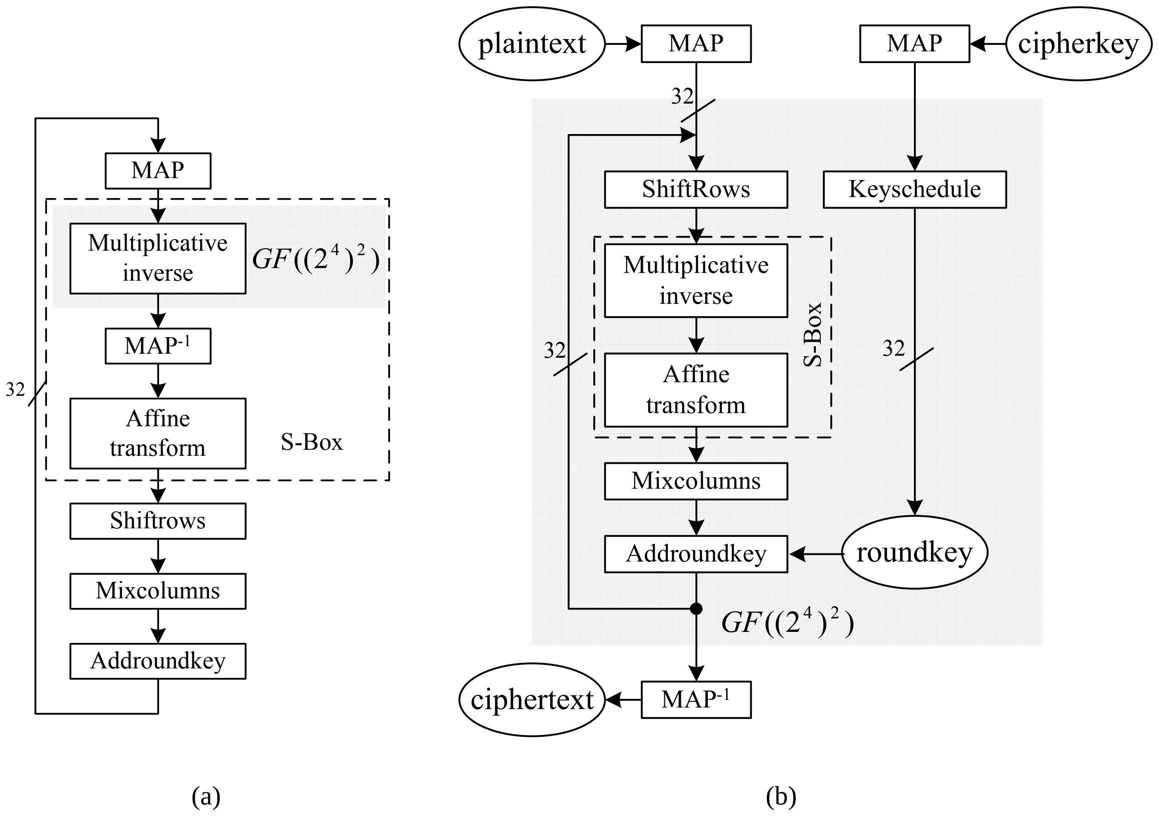


Figure 4.2: Partial Composite Field (a)- Full Composite Field (b)

Many high-end FPGA devices possess Block-RAMs (BRAMs) which are efficient for the implementation of S-Box. S-Box, also referred as subbytes, is the key part in both encryptor and keyschedule modules. However, these BRAM-based designs cannot be implemented in the low-cost devices which do not have BRAMs. An alternative approach for S-Box implementation is using combinational logic. But this method may lead to high hardware complexity because of the mathematic operations of AES over finite field $GF(2^8)$.

The key step of S-Box is calculating multiplicative inverse of each byte (Section (3.1)). Since the introduction of composite field $GF((2^4)^2)$ based S-Box, numerous research [9, 11, 28, 31, 34] has investigated the calculation of the multiplicative inverses over $GF((2^4)^2)$, instead of $GF(2^8)$, to decrease hardware complexity (Figure (4.2(a))). In Figure (4.2), the arithmetic in the shadow area is performed over field $GF((2^4)^2)$. Figure (4.2(a)) shows that the architecture implements only multiplicative inverse in $GF((2^4)^2)$. The architectures in [33, 22] extend the field $GF((2^4)^2)$ to affine transformation which makes all S-Box block operations performed in $GF((2^4)^2)$. By decomposing these operations from $GF(2^8)$ to its subfield $GF(2^4)$, the hardware complexity is decreased.

As in Figure (4.2(a)), in each round before S-Box, it needs an isomorphic mapping function (*MAP*) to convert a representation from $GF(2^8)$ to $GF((2^4)^2)$; to convert inversely after, it needs the inverse mapping (MAP^{-1}). If key size is 128 bits, it applies the S-Box to the plaintext and the cipherkey ten times, which means that it needs 20 *MAPs* and 20 $MAP^{-1}s$ for the encryption of 128-bit data. In [32], for every byte, *MAP* costs 11 XOR gates with 2 gates in critical path; MAP^{-1} costs 15 XOR gates with 3 gates in critical path. *MAP* and MAP^{-1} together cost 33.3% in critical path and 21% gates in total for the subbytes transformation.

In order to reduce the cost of *MAP* and MAP^{-1} as much as possible, we propose the complete composite field approach (Figure (4.2(b))). The $GF((2^4)^2)$ field covers both encryptor and keyschedule. As illustrated in Figure (4.2(b)), one *MAP* and one MAP^{-1} are applied in encryption, one MAP^{-1} is applied in keyschedule. This is a constant overhead which is not affected by the round count. No matter what the key size is, the cost of mapping is the same.

The isomorphic mapping functions between field $GF(2^8)$ and field $GF((2^4)^2)$ are determined by the irreducible polynomials of field $GF(2^8)$ (Equation (2.3)) and field $GF((2^4)^2)$ (Equations (2.10) and (2.11)). We use the mapping formulas in [32] to conduct the transition of representations between $GF(2^8)$ and $GF((2^4)^2)$:

$$\begin{array}{l}
a_h x + a_l = MAP(a), \quad a_h, a_l \in GF(2^4), \quad a \in GF(2^8) \\
-------------------------------- \\
a_A = a_1 \oplus a_7, \quad a_B = a_5 \oplus a_7, \quad a_C = a_4 \oplus a_6 \\
a_{l0} = a_C \oplus a_0 \oplus a_5, \quad a_{l1} = a_1 \oplus a_2, \quad a_{l2} = a_A, \quad a_{l3} = a_2 \oplus a_4 \\
a_{h0} = a_C \oplus a_5, \quad a_{h1} = a_A \oplus a_C, \quad a_{h2} = a_B \oplus a_2 \oplus a_3, \quad a_{h3} = a_B
\end{array} \tag{4.1}$$

In Equation (4.1), a is an element in field $GF(2^8)$. $MAP(a)$ convert a to its isomorphic element in $GF((2^4)^2)$, which is represented as $a_h x + a_l$.

$$\begin{array}{l}
a = MAP^{-1}(a_h x + a_l), \quad a \in GF(2^8), \quad a_h, a_l \in GF(2^4) \\
-------------------------------- \\
a_A = a_{l1} \oplus a_{h3}, \quad a_B = a_{h0} \oplus a_{h1} \\
a_0 = a_{l0} \oplus a_{h0}, \quad a_1 = a_B \oplus a_{h3}, \quad a_2 = a_A \oplus a_B \\
a_3 = a_B \oplus a_{l1} \oplus a_{h2}, \quad a_4 = a_A \oplus a_B \oplus a_{l3}, \quad a_5 = a_B \oplus a_{l2} \\
a_6 = a_A \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}, \quad a_7 = a_B \oplus a_{l2} \oplus a_{h3}
\end{array} \tag{4.2}$$

In Equation (4.2), $a_h x + a_l$ in an element in field $GF((2^4)^2)$. $MAP^{-1}(a_h x + a_l)$ convert $a_h x + a_l$ to its isomorphic element in $GF(2^8)$, which is represented as a.

4.3 Subpipelined Encryptor and Keyschedule

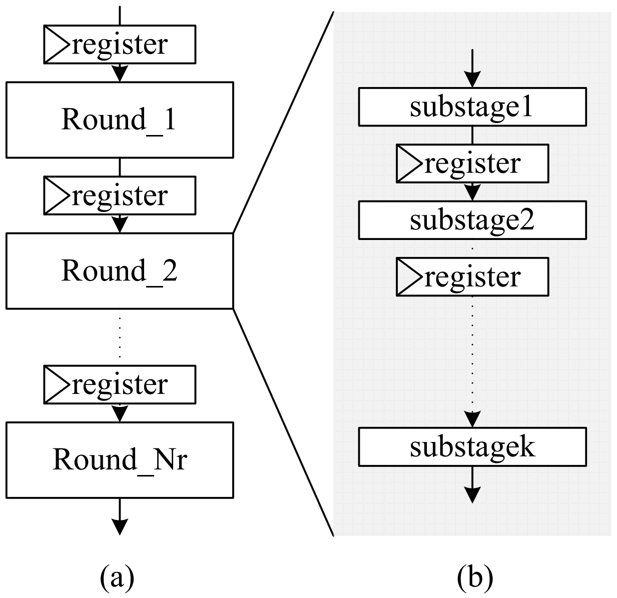


Figure 4.3: Pipelining (a) and Subpipelining (b)

The technique of pipelining is applied in the AES designs to optimize speed/area ratio in [1, 2, 8, 9, 11, 17, 33, 26, 27, 31, 34]. By inserting registers among combinational logic, multiple blocks are processed simultaneously. The frequency is determined by the maximum delay between two registers. When the maximum delay between two registers is decreased, the frequency is increased.

Figure (4.3(a)) is the fully unrolled pipelining architecture, which includes two steps. First, unfold the *Nr* round units on the device; second, insert registers between each round unit. In this case, the maximum delay is the period of one round which contains four transformations.

By cutting one round unit into more substages, we can further improve the frequency. This technology is called subpipelining [34]. Figure (4.3(b)) gives an example where registers are placed both between and inside each round unit. The frequency is determined by the maximum delay of a substage. In this thesis, we propose a single round subpipelined architecture, where one round unit is implemented and subpipelined into eight substages.

To generate the roundkeys, we design an on-the-fly keyschedule, which generates a 32-bit roundkey at each clock cycle. The encryption unit and the key expansion unit share the same clock which leads to the fact that the general frequency is determined by the maximum delay in both units. Hence, the substage balance of keyschedule is as important as in encryptor. We propose a new subpipelined keyschedule on composite field for all standard key sizes. The most costly part of keyschedule is still the S-Box. We divide it into the same substages as in encryptor.

4.4 Double-Block Subpipelined Architecture

An equivalent decryptor along with the AES was introduced in FIPS [20], where the same architecture can be used in both encryption and decryption. Figure 5.7 in [30] illustrates the equivalent inverse cipher. It makes use of the fact that the order of subbytes and shiftrows can be exchanged because subbytes changes the value of each byte individually while shiftrows only rearranges their positions. So it changes the order of invshiftrows and invsubbytes, and add an extra step to conduct invmixcolumns on each roundkey. We can also change the sequence of shiftrows and subbytes in encryptor to obtain the same result. In this design, we put shiftrows before subbytes.

Figure (4.4) illustrates the proposed encryption architecture. The eight 32-bit registers (four in shiftrows, three in subbytes and one between subbytes and mixcolumns) are used to cut one round unit into eight substages, which leads to an eight clock cycles initial delay to generate the first 32-bit ciphertext. $clk\_counter$ in Figure (4.4) is a clock register counter generated in keyschedule. It is repeating from 0 to $8Nr+7$ (Table (4.1)) and is used to synchronize encryptor and keyschedule.

Table 4.1: AES Encryption Sequence

clk_counter	0	1	2	3	4	5	6	7
plaintext	PA(0)	PA(1)	PA(2)	PA(3)	PB(0)	PB(1)	PB(2)	PB(3)
cipherkey	KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)
outcome(0)	OA(0)	OA(1)	OA(2)	OA(3)	OB(0)	OB(1)	OB(2)	OB(3)

(a)

clk_counter	8	9	10	11	12	13	14	15
input(1)	OA(0)	OA(1)	OA(2)	OA(3)	OB(0)	OB(1)	OB(2)	OB(3)
roundkey(1)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)	KB(6)	KB(7)
outcome(1)	OA(4)	OA(5)	OA(6)	OA(7)	OB(4)	OB(5)	OB(6)	OB(7)

(b)

•

•

•

clk_counter	8Nr	8Nr+1	8Nr+2	8Nr+3	8Nr+4	8Nr+5	8Nr+6	8Nr+7
input(Nr)	OA(4Nr-4))	OA(4Nr-3)	OA(4Nr-2)	OA(4Nr-1)	OB(4Nr-4)	OB(4Nr-3)	OB(4Nr-2)	OB(4Nr-1)
roundkey(Nr)	KA(4Nr)	KA(4Nr+1)	KA(4Nr+2)	KA(4Nr+3)	KB(4Nr)	KB(4Nr+1)	KB(4Nr+2)	KB(4Nr+3)
ciphertext	CA(0)	CA(1)	CA(2)	CA(3)	CB(0)	CB(1)	CB(2)	CB(3)

(c)

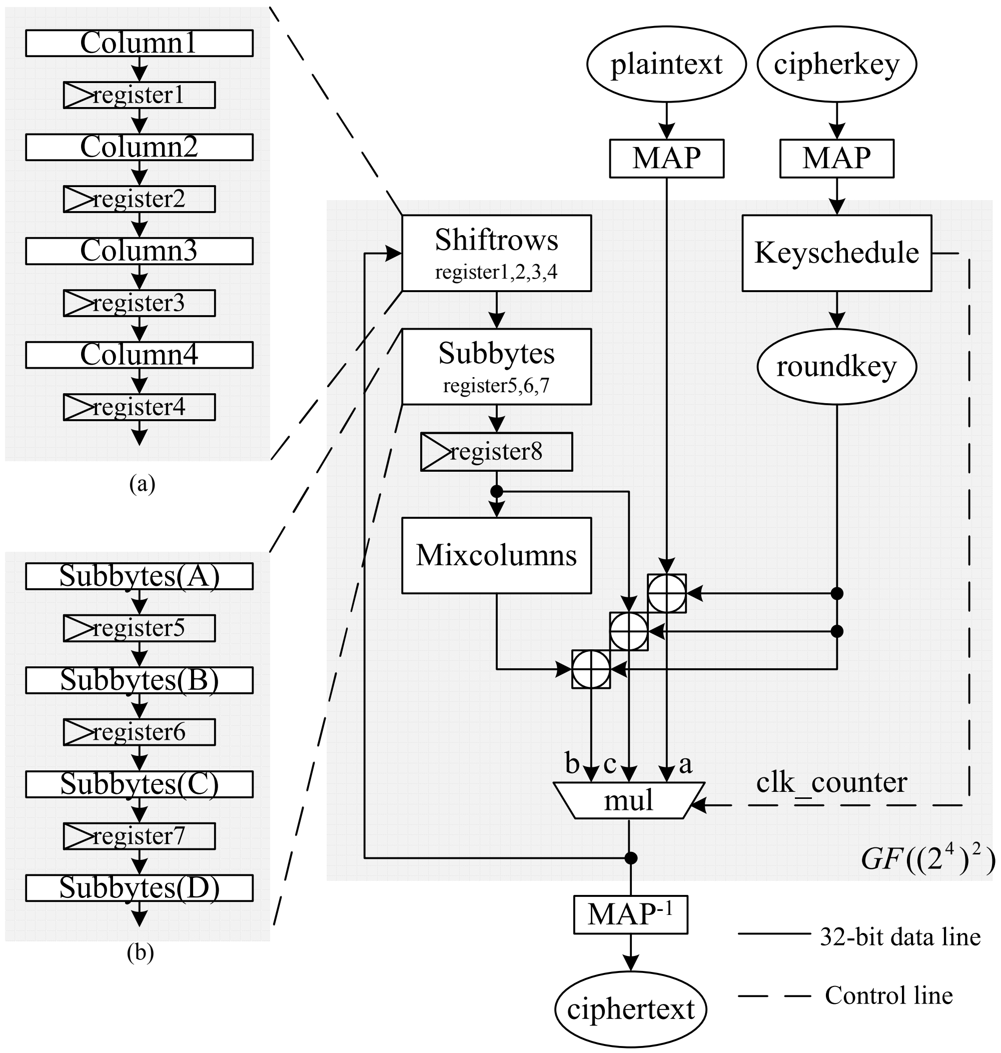


Figure 4.4: AES Encryption Architecture

We use a double-block (block *A* and *B*) data flow to avoid the eight clock cycles initial delay. Table (4.1(a)) illustrates the data sequence of the initial round ($Nr = 0$).

$\{PA(0),\ PA(1),\ PA(2),\ PA(3)\}$: 128-bit plaintext of block *A*

$\{PB(0),\ PB(1),\ PB(2),\ PB(3)\}$: 128-bit plaintext of block *B*

They are put into AES during the first eight clock cycles and then processed alternately.

$\{KA(0),\ KA(1),\ KA(2),\ KA(3)\}$: cipherkey for block *A*

$\{KB(0),\ KB(1),\ KB(2),\ KB(3)\}$: cipherkey for block *B*

Because in the initial round, the encryption involves only addroundkey, which is the simple XOR operation, and the according roundkey is the MAPed cipherkey, the operation in this round is not delayed by registers. Hence, the outcome of the initial round (outcome(0)) is produced from the very beginning.

$\{OA(0),\, OA(1),\, OA(2),\, OA(3)\}$: outcome of round 0 for block A

$\{OB(0),\, OB(1),\, OB(2),\, OB(3)\}$: outcome of round 0 for block B

Table (4.1(b)) is for round 1, which goes through the eight substages. At the eighth clock cycle, $OA(0)$ finishes the eight substages and XORes the the according roundkey ($KA(4)$) to generate the outcome ($OA(4)$) for block A, so as block B.

Table (4.1(c)) is for the last round Nr.

$\{CA(0),\, CA(1),\, CA(2),\, CA(3)\}$: ciphertext for block A

$\{CB(0),\, CB(1),\, CB(2),\, CB(3)\}$: ciphertext for block B

Now we explain the 3-to-1 multiplexer (*mul*) controlled by the *clk_counter*:

- **Case a:** In initial round, where $0 \leq clk\_counter < 8$, 128-bit plaintext is MAPed into $GF((2^4)^2)$ and XORed with the according roundkey in four clock cycles, 32 bits at each clock cycle. The result is the outcome of the initial round ($Nr = 0$) which is the input of the second round;

- **Case b:** In normal rounds, where $8 \leq clk\_counter < Nr \times 8$, the outcome of mix-columns XORs with the according roundkey to produce the outcome of this round.

- **Case c:** The last round, where $Nr \times 8 \leq clk\_counter < (Nr+1) \times 8$, the transformation mixcolumns is skipped. The result of subbytes is added with its roundkey.

Finally, the outcome of the last round goes through MAP^{-1} to generate the ciphertext.

Table 4.2: Four Control Signals

	$Counter\_W_0$	$Counter\_W_1$	$Counter\_W_2$	$Counter\_W_3$
W_0	1	0	0	0
W_1	0	1	0	0
W_2	0	0	1	0
W_3	0	0	0	1

4.4.1 Column Fashion Shiftrows

This subsection proposes the column fashion shiftrows (Figure (4.5)). It includes 16 8-bit registers ($Row0\_Col0$, $Row0\_Col1$, ... , $Row3\_Col3$) and three *2 to 1* multiplexers ($M1$, $M2$ and $M3$). Both input and output of shiftrows is a state. Each column is a word (W_0, W_1, W_2 and W_3), which includes four bytes. Every clock cycle it processes a 32-bit word (one column of a state), so four clock cycles are needed to produce a 128-bit state. The first 3 clock cycles are initial clock cycles, so the first word is shifted out at the 4th clock cycle. Figure (4.6) shows how it works in the first eight clock cycles. R_{00}, R_{01}, ... and R_{33} stand for registers $Row0\_Col0$, $Row0\_Col1$, ... and $Row3\_Col3$. Each row shows their values at each clock cycle($clk0$, $clk1$, ... and $clk7$). We will explain the shadow area and black border in the following text.

Four counters ($Counter\_W_0$, $Counter\_W_1$, $Counter\_W_2$ and $Counnter\_W_3$) control the registers and the multiplexers. Table (4.2) shows how these signals are generated.

When the first word (W_0) of a state is shifted in, $Counter\_W_0 = 1$;

When the second word (W_1) of a state is shifted in, $Counter\_W_1 = 1$;

When the third word (W_2) of a state is shifted in, $Counter\_W_2 = 1$;

When the forth word (W_3) of a state is shifted in, $Counter\_W_3 = 1$.

Certain registers are controlled by special enable signals ($Enable\_row1\_col3$, $Enable\_row2\_col23$ and $Enable\_row3\_col123$), others use the general enable signal, which

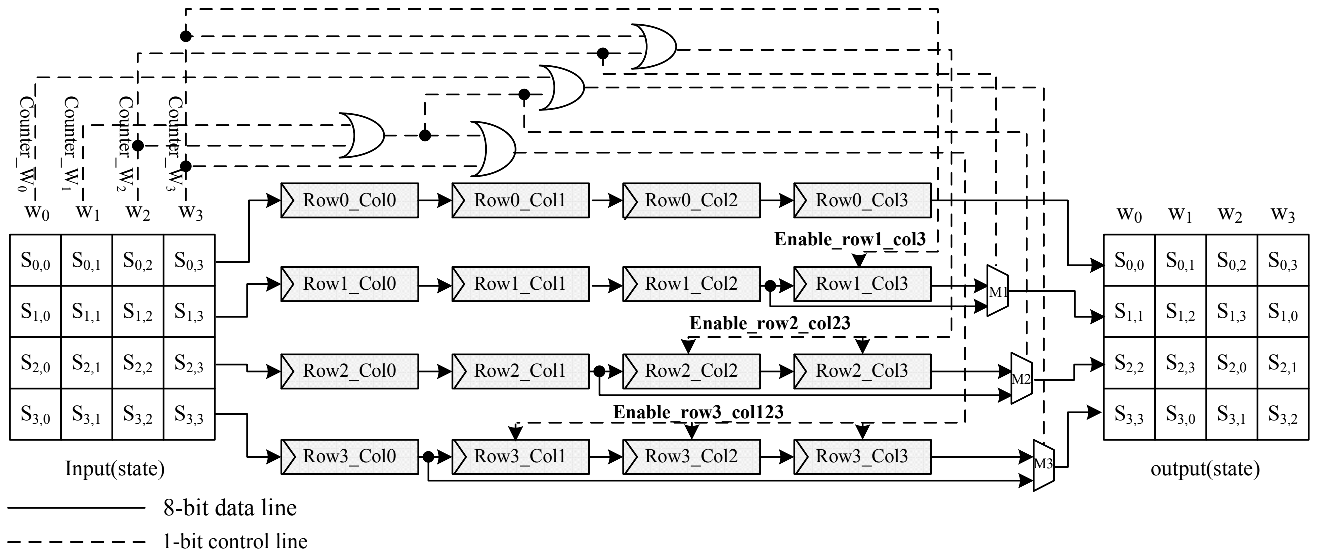


Figure 4.5: Column Fashion Shiftrows

is not shown.

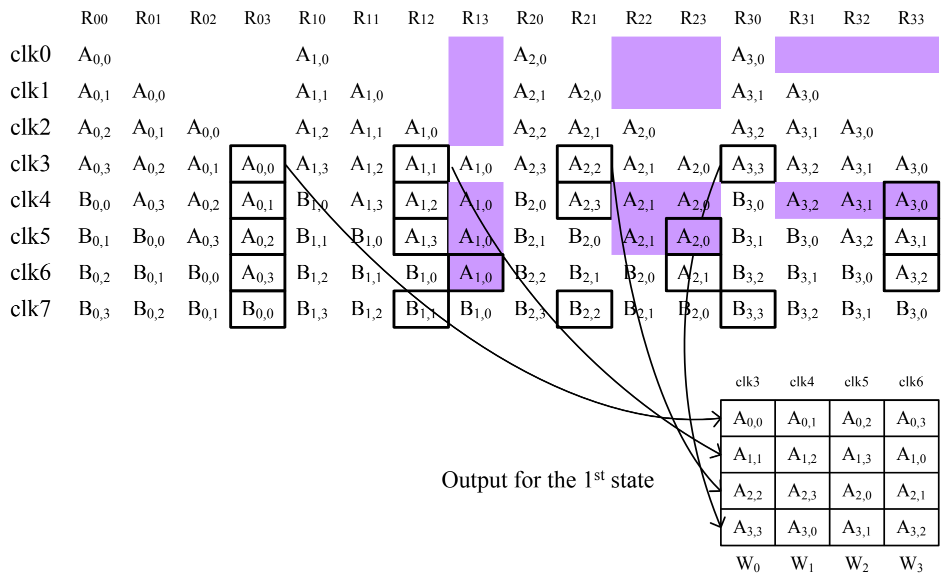


Figure 4.6: Two States' Arrangement in Shiftrows Registers

Enable $Enable\_row1\_col3$ ($Enable\_row1\_col3 = Counter\_W_3$) controls register $Row1\_Col3$. This enable signal is negative when clock is $clk0$, $clk1$, $clk2$, $clk4$, $clk5$, $clk6$, etc. $Row1\_Col3$ does not work during these clock cycles, which corresponds to the shadow areas of the column $R13$ in Figure (4.6);

Enable $Enable\_row2\_col23$ ($Enable\_row2\_col23 = Counter\_W_2 \vee Ccounter\_W_3$) controls registers $Row2\_Col2$ and $Row2\_Col3$. This enable signal is negative when clock is $clk0$, $clk1$, $clk4$, $clk5$, etc. $Row2\_Col2$ and $Row2\_Col3$ do not work during these clock cycles, which corresponds to the shadow area of the columns $R22$ and $R23$ in Figure (4.6);

Enable $Enable\_row3\_col123$ ($Enable\_row3\_col123 = Counter\_W_1 \vee Counter\_W_2 \vee Counter\_W_3$) controls registers $Row3\_Col1$, $Row3\_Col2$ and $Row3\_Col3$. This enable signal is negative when clock is $clk0$, $clk4$, etc. $Row3\_Col1$, $Row3\_Col2$ and $Row3\_Col3$ do not work during these clock cycles, which corresponds to the shadow area of the columns $R31$, $R32$ and $R33$ in Figure (4.6).

For each input word:

The input (a state) of shiftrows is the MAPed ciphertext if it is the initial round; otherwise it is the outcome of the previous round. Each word of the state (W_0, W_1, W_2 and W_3) is shifted into the first column of the registers ($Row0\_Col0$, $Row1\_Col0$, $Row2\_Col0$ and $Row3\_Col0$) at each clock cycle.

For each output word:

- 1st byte is shifted out from $Row0\_Col3$, which corresponds to the black border area of column R_{03} in Figure (4.6);

- 2nd byte is shifted out from $Row1\_Col3$ if ($Counter\_W_2$) is active, otherwise from $Row1\_Col2$, which corresponds to the black border area of columns R_{12} and R_{13} in Figure (4.6);

- 3rd byte is shifted out from $Row2\_Col3$ if ($Counter\_W_1$ or $Counter\_W_2$) is active, otherwise from $Row2\_Col1$, which corresponds to the black border area of columns R_{21} and R_{23} in Figure (4.6);

- 4th byte is shifted out from $Row3\_Col3$ if ($Counter\_W_0$ or $Counter\_W_1$ or $Counter\_W_2$) is active, otherwise from $Row3\_Col0$, which corresponds to the black border area of columns R_{30} and R_{33} in Figure (4.6).

Figure (4.6) takes two states A and B (Figure (4.7)) as the input of shiftrows. During the firsts eight clock cycles, each word of state A and B is shifted into the first column of registers (R_{00}, R_{10}, R_{20} and R_{30}) one after another. The first three clock cycles are the initial cycles with no output.

At $clk3$, the first column of state A is generated from registers (R_{03}, R_{12}, R_{21} and R_{30});

At $clk4$, the second column of state A is generated from registers (R_{03}, R_{12}, R_{21} and R_{33});

At $clk5$, the third column of state A is generated from registers (R_{03}, R_{12}, R_{23} and R_{33});

State A

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
W_0	W_1	W_2	W_3

State B

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$
W_0	W_1	W_2	W_3

Figure 4.7: Input of Shiftrows in Figure (4.6)

At *clk*6, the forth column of state *A* is generated from registers (R_{03}, R_{13}, R_{23} and R_{33}). The first output state is shown in the right down corner of the Figure (4.6).

4.4.2 Subpipelined Subbytes

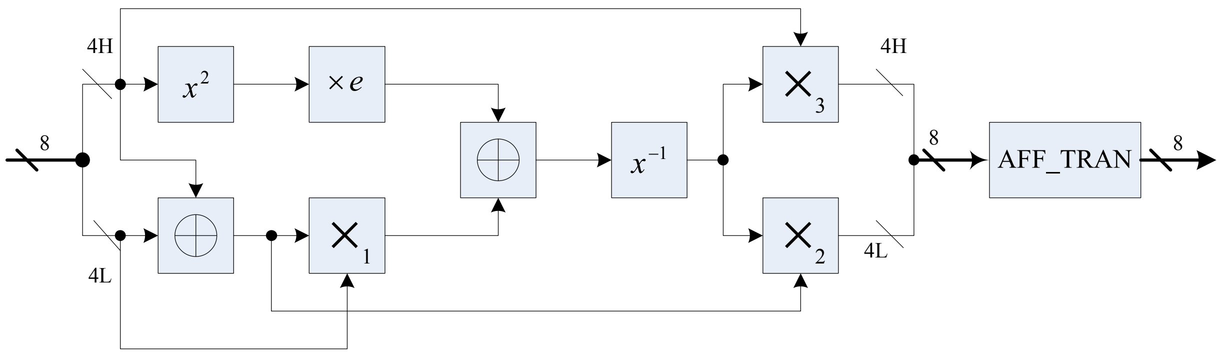


Figure 4.8: Subbytes in composite field $GF(2^4)$[34]

The key step of subbytes is the calculation of the multiplicative inverse. Figure (4.8) illustrates the architecture of subbytes used in [34], which applies Equation (2.15). As shown in this figure, it uses multiplication in $GF(2^4)$ three times. In order to distinguish the multipliers, we indicate them as $\times_1$, $\times_2$, $\times_3$. It also needs one inversion (x^{-1}), one constant multiplier with $\{E\}$ ($\times e$), $\{E\}$ is in hexadecimal notation, which is '1110' in binary notation), one squarer (x^2) and two 4-bit XORs ($\oplus$). These arithmetic operations are over field $GF(2^4)$.

Considering $x,y,z \in GF(2^4)$, x, y and z are represented in binary notation where $x = \{x_3x_2x_1x_0\}$, $y = \{y_3y_2y_1y_0\}$, $z = \{z_3z_2z_1z_0\}$. Let a, b, c, d, e and f are 1-bit value, which equals to 0 or 1. $\oplus$ stands for XOR-operation. x_0y_1 means $x_0 \wedge y_1$.

The following Equations (4.3), (4.4), (4.5) and (4.6) are used to calculate squaring, constant multiplication with {E}, multiplication and multiplicative inverse [32].

$$\begin{aligned}
&y = x^2\\
&-----------------------------\\
&y_0 = x_0 \oplus x_2, \quad y_1 = x_2\\
&y_2 = x_1 \oplus x_3, \quad y_3 = x_3
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
&y = x \times \{E\}\\
&-----------------------------\\
&a = x_0 \oplus x_1, \quad b = x_2 \oplus x_3\\
&y_0 = x_1 \oplus b, \quad y_1 = a\\
&y_2 = a \oplus x_2, \quad y_3 = a \oplus b
\end{aligned} \tag{4.4}$$

$$
\begin{aligned}
&z = x \times y \\
&------------------------------- \\
&a = x_0 \oplus x_3, \quad b = x_2 \oplus x_3, \quad c = x_1 \oplus x_2 \\
&z_0 = x_0y_0 \oplus x_3y_1 \oplus x_2y_2 \oplus x_1y_3 \\
&z_1 = x_1y_0 \oplus ay_1 \oplus by_2 \oplus cy_3 \\
&z_2 = x_2y_0 \oplus x_1y_1 \oplus ay_2 \oplus by_3 \\
&z_3 = x_3y_0 \oplus x_2y_1 \oplus x_1y_2 \oplus ay_3
\end{aligned}
\tag{4.5}
$$

$$
\begin{aligned}
&y = x^{-1} \\
&------------------------------- \\
&a = x_1 \oplus x_2 \oplus x_3 \oplus x_1x_2x_3 \\
&y_0 = a \oplus x_0 \oplus x_0x_2 \oplus x_1x_2 \oplus x_0x_1x_2 \\
&y_1 = x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \oplus x_3 \oplus x_1x_3 \oplus x_0x_1x_3 \\
&y_2 = x_0x_1 \oplus x_2 \oplus x_0x_2 \oplus x_3 \oplus x_0x_3 \oplus x_0x_2x_3 \\
&y_3 = a \oplus x_0x_3 \oplus x_1x_3 \oplus x_2x_3
\end{aligned}
\tag{4.6}
$$

As illustrated in Figure (4.4), subbytes should be cut into four substages. The key to an efficient subpipelining technology is to balance the delays of these substages. Previous research [34] calculate the delay of an individual substage by counting the gates in critical path.

Xilinx ISE provides synthesis tool to yield the maximum combinational delay of an entity. A more straightforward method to achieve the optimal balance is to cut subbytes in different manners and use this synthesis tool to measure the delay of each substage (an entity). The most even delays of these substages stand for the optimal balanced substages arrangement.

Based on our experiments, Equation (4.6) is not suitable for this 4-substage subbytes.

With this equation, the substage including x^{-1} yields the longest delay, hence decreasing this substage's delay can increase the general frequency. We derive a new Equation (4.7) from Equation (4.6) to reduce the delay caused by x^{-1}. Equation (4.7)is derived in three steps:

1. In Equation (4.6), replace a by its expression, we have:

   $y_0 = x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_0x_2 \oplus x_1x_2 \oplus x_0x_1x_2 \oplus x_1x_2x_3$

   $y_1 = x_0x_1 \oplus x_0x_2 \oplus x_1x_2 \oplus x_3 \oplus x_1x_3 \oplus x_0x_1x_3$

   $y_2 = x_0x_1 \oplus x_2 \oplus x_0x_2 \oplus x_3 \oplus x_0x_3 \oplus x_0x_2x_3$

   $y_3 = x_1 \oplus x_2 \oplus x_3 \oplus x_1x_2x_3 \oplus x_0x_3 \oplus x_1x_3 \oplus x_2x_3$

2. The expressions in step 1 can be equally changed to:

   $y_0 = x_1 \oplus x_2 \oplus x_1x_2 \oplus x_0x_2 \oplus (x_0 \oplus x_3)(1 \oplus x_1x_2)$

   $y_1 = x_1x_2 \oplus x_0x_2 \oplus x_0x_1 \oplus x_3(1 \oplus x_1 \oplus x_0x_1)$

   $y_2 = x_2 \oplus x_0x_2 \oplus x_0x_1 \oplus x_3(1 \oplus x_0 \oplus x_0x_2)$

   $y_3 = x_1 \oplus x_2 \oplus x_3(1 \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_1x_2)$

3. Let $a = x_1x_2,\ b = x_0x_2,\ c = x_0x_1,\ d = x_1 \oplus x_2,\ e = 1 \oplus a$ and $f = b \oplus c$, we have:

$$
\begin{aligned}
& y = x^{-1} \\
& ------------------------------ \\
& a = x_1x_2, \quad b = x_0x_2, \quad c = x_0x_1, \quad d = x_1 \oplus x_2 \\
& e = 1 \oplus a, \quad f = b \oplus c \\
& y_0 = a \oplus b \oplus d \oplus (x_0 \oplus x_3)e \\
& y_1 = a \oplus f \oplus x_3(x_1 \oplus 1 \oplus c) \\
& y_2 = f \oplus x_2 \oplus x_3(b \oplus 1 \oplus x_0) \\
& y_3 = d \oplus x_3(e \oplus x_0 \oplus d)
\end{aligned}
\tag{4.7}
$$

According to Equation (4.7), we design the circuit Figure (4.9) to perform x^{-1} over $GF(2^4)$.

Besides multiplicative inversion, other expensive operations in Figure (4.8) are the three multiplications ($\times_1$, $\times_2$ and $\times_3$). In order to decrease the maximum delay caused by multiplication, we separate each multiplication into two steps and put each step in different substages. The registers between each substage store the result of the first step of multiplication and pass it to the second step. We decompose these three multipliers into two different manners (*AB-type* and *MN-type*) to achieve the best balance.

AB-type: Equation (4.8) is derived from Equation (4.5). p_0, p_1, ..., p_{15} are 1-bit values, which represents one AND term in Equation (4.5). $Step\ A$ calculates the value of all the terms; $Step\ B$ conducts XOR of every four values to generate z_0, z_1, z_2 and z_3. A register is inserted between *Step A* and *Step B* to store $p_0, p_1, ..., p_{15}$. $\times_1$ in Figure (4.8) is separated in this way, as $\times_{1A}$ and $\times_{1B}$ in Figure (4.9);

$$\begin{aligned}
&z = x \times y \ (AB-type) \\
&----------Step\ A------------------ \\
&a = x_0 \oplus x_3, \quad b = x_2 \oplus x_3, \quad c = x_1 \oplus x_2 \\
&p_0 = x_0 y_0, \quad p_1 = x_3 y_1, \quad p_2 = x_2 y_2, \quad p_3 = x_1 y_3 \\
&p_4 = x_1 y_0, \quad p_5 = a y_1, \quad p_6 = b y_2, \quad p_7 = c y_3 \\
&p_8 = x_2 y_0, \quad p_9 = x_1 y_1, \quad p_{10} = a y_2, \quad p_{11} = b y_3 \\
&p_{12} = x_3 y_0, \quad p_{12} = x_2 y_1, \quad p_{14} = x_1 y_2, \quad p_{15} = a y_3 \\
&----------Step\ B------------------ \\
&z_0 = p_0 \oplus p_1 \oplus p_2 \oplus p_3 \\
&z_1 = p_4 \oplus p_5 \oplus p_6 \oplus p_7 \\
&z_2 = p_8 \oplus p_9 \oplus p_{10} \oplus p_{11} \\
&z_3 = p_{12} \oplus p_{13} \oplus p_{14} \oplus p_{15}
\end{aligned} \tag{4.8}$$

MN-type: Equation (4.9) is also derived form Equation (4.5). $Step\ M$ creates the value of a, b and c; $Step\ N$ finishes the rest of Equation (4.5). A register is inserted between $Step\ M$ and $Step\ N$ to store a, b, c. $\times_2$ and $\times_3$ in Figure (4.8) are separated in this way, as $\times_{2M}$ and $\times_{2N}$, $\times_{3M}$ and $\times_{3N}$ in Figure (4.9).

$$\begin{aligned}
&z = x \times y\ (MN - type)\\
&- - - - - - - - - - Step\ M - - - - - - - - - - - - - - - - - -\\
&a = x_0 \oplus x_3, \quad b = x_2 \oplus x_3, \quad c = x_1 \oplus x_2\\
&- - - - - - - - - - Step\ N - - - - - - - - - - - - - - - - - -\\
&z_0 = x_0 y_0 \oplus x_3 y_1 \oplus x_2 y_2 \oplus x_1 y_3\\
&z_1 = x_1 y_0 \oplus a y_1 \oplus b y_2 \oplus c y_3\\
&z_2 = x_2 y_0 \oplus x_1 y_1 \oplus a y_2 \oplus b y_3\\
&z_3 = x_3 y_0 \oplus x_2 y_1 \oplus x_1 y_2 \oplus a y_3
\end{aligned} \tag{4.9}$$

The last operation in subbytes is the affine transformation. We derive Equation (4.16) to do the affine transformation, based on Equation (3.1), Equation (4.1) and Equation (4.2). First, we change the format of Equation (4.1) and Equation (4.2).

Consider $p \in GF((2^4)^2)$, $q \in GF(2^8)$:

$$p = \{p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0\}$$
$$q = \{q_7 q_6 q_5 q_4 q_3 q_2 q_1 q_0\}$$

For Equation (4.1):

1. In expression of a_{l0},...,a_{h3}, replace a_A, a_B and a_C by their expression

   $a_{l0} = a_4 \oplus a_6 \oplus a_0 \oplus a_5$

   $a_{l1} = a_1 \oplus a_2$

   $a_{l2} = a_1 \oplus a_7$

$a_{l3} = a_2 \oplus a_4$

$a_{h0} = a_4 \oplus a_6 \oplus a_5$

$a_{h1} = a_1 \oplus a_7 \oplus a_4 \oplus a_6$

$a_{h2} = a_5 \oplus a_7 \oplus a_2 \oplus a_3$

$a_{h3} = a_5 \oplus a_7$

2. Let p replace $a_hx + a_l$, q replace a, we have Equation (4.10)

$$\begin{aligned}
&p = MAP(q), \;\; p \in GF((2^4)^2), \;\; q \in GF(2^8) \\
&------------------------------- \\
&p_0 = q_0 \oplus q_4 \oplus q_5 \oplus q_6 \\
&p_1 = q_1 \oplus q_2 \\
&p_2 = q_1 \oplus q_7 \\
&p_3 = q_2 \oplus q_4 \\
&p_4 = q_4 \oplus q_5 \oplus q_6 \\
&p_5 = q_1 \oplus q_4 \oplus q_6 \oplus q_7 \\
&p_6 = q_2 \oplus q_3 \oplus q_5 \oplus q_7 \\
&p_7 = q_5 \oplus q_7
\end{aligned} \tag{4.10}$$

The same steps for Equation (4.2):

1. In expression of a_0,...,a_7, replace a_A and a_B by their expression

$a_0 = a_{l0} \oplus a_{h0}$

$a_1 = a_{h0} \oplus a_{h1} \oplus a_{h3}$

$a_2 = a_{l1} \oplus a_{h3} \oplus a_{h0} \oplus a_{h1}$

$a_3 = a_{h0} \oplus a_{h1} \oplus a_{l1} \oplus a_{h2}$

$a_4 = a_{l1} \oplus a_{h3} \oplus a_{h0} \oplus a_{h1} \oplus a_{l3}$

$a_5 = a_{h0} \oplus a_{h1} \oplus a_{l2}$

$a_6 = a_{l1} \oplus a_{h3} \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}$

$a_7 = a_{h0} \oplus a_{h1} \oplus a_{l2} \oplus a_{h3}$

2. Let q replace a, p replace $a_h x + a_l$, we have Equation (4.11)

$$
\begin{aligned}
&q = MAP^{-1}(p), \;\; q \in GF(2^8), \;\; p \in GF((2^4)^2) \\
&-------------------------------- \\
&q_0 = p_0 \oplus p_4 \\
&q_1 = p_4 \oplus p_5 \oplus p_7 \\
&q_2 = p_1 \oplus p_4 \oplus p_5 \oplus p_7 \\
&q_3 = p_1 \oplus p_4 \oplus p_5 \oplus p_6 \\
&q_4 = p_1 \oplus p_3 \oplus p_4 \oplus p_5 \oplus p_7 \\
&q_5 = p_2 \oplus p_4 \oplus p_5 \\
&q_6 = p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_7 \\
&q_7 = p_2 \oplus p_4 \oplus p_5 \oplus p_7
\end{aligned}
\tag{4.11}
$$

Now we use Equation (3.1), Equation (4.10) and Equation (4.11) to derive Equation (4.16).

Let $x', \; y$ be the elements in $GF(2^8)$:

$$
\begin{aligned}
x' &= \{x'_7 x'_6 x'_5 x'_4 x'_3 x'_2 x'_1 x'_0\} \\
y &= \{y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0\}
\end{aligned}
$$

According to Equation (3.1), we have:

$$
\begin{aligned}
y_0 &= x'_0 \oplus x'_4 \oplus x'_5 \oplus x'_6 \oplus x'_7 \oplus 1 \\
y_1 &= x'_0 \oplus x'_1 \oplus x'_5 \oplus x'_6 \oplus x'_7 \oplus 1 \\
y_2 &= x'_0 \oplus x'_1 \oplus x'_2 \oplus x'_6 \oplus x'_7 \\
y_3 &= x'_0 \oplus x'_1 \oplus x'_2 \oplus x'_3 \oplus x'_7 \\
y_4 &= x'_0 \oplus x'_1 \oplus x'_2 \oplus x'_3 \oplus x'_4 \\
y_5 &= x'_1 \oplus x'_2 \oplus x'_3 \oplus x'_4 \oplus x'_5 \oplus 1 \\
y_6 &= x'_2 \oplus x'_3 \oplus x'_4 \oplus x'_5 \oplus x'_6 \oplus 1 \\
y_7 &= x'_3 \oplus x'_4 \oplus x'_5 \oplus x'_6 \oplus x'_7
\end{aligned}
\tag{4.12}
$$

In the following, we convert the result of y to the field $GF((2^4)^2)$, and use the $GF((2^4)^2)$ format to represent x'. Thus, we can derive the affine transformation in $GF((2^4)^2)$.

1. We let w to represent y in $GF((2^4)^2)$ (w is one element in $GF((2^4)^2)$). According to Equation (4.10) (Map from $GF(2^8)$ to $GF((2^4)^2)$):

$$
\begin{aligned}
w_0 &= y_0 \oplus y_4 \oplus y_5 \oplus y_6 \\
w_1 &= y_1 \oplus y_2 \\
w_2 &= y_1 \oplus y_7 \\
w_3 &= y_2 \oplus y_4 \\
w_4 &= y_4 \oplus y_5 \oplus y_6 \\
w_5 &= y_1 \oplus y_4 \oplus y_6 \oplus y_7 \\
w_6 &= y_2 \oplus y_3 \oplus y_5 \oplus y_7 \\
w_7 &= y_5 \oplus y_7
\end{aligned}
\tag{4.13}
$$

2. Next, we use $GF((2^4)^2)$ format to represent x' in Equation 4.12. Let z be the $GF((2^4)^2)$ format of x'. From Equation 4.11, we have:

$$\begin{aligned}
x'_0 &= z_0 \oplus z_4 \\
x'_1 &= z_4 \oplus z_5 \oplus z_7 \\
x'_2 &= z_1 \oplus z_4 \oplus z_5 \oplus z_7 \\
x'_3 &= z_1 \oplus z_4 \oplus z_5 \oplus z_6 \\
x'_4 &= z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7 \\
x'_5 &= z_2 \oplus z_4 \oplus z_5 \\
x'_6 &= z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_7 \\
x'_7 &= z_2 \oplus z_4 \oplus z_5 \oplus z_7
\end{aligned} \tag{4.14}$$

3. Now, we replace y with its $GF((2^4)^2)$ format (w), and replace x' with its $GF((2^4)^2)$ format (z):

   $w_0 = y_0 \oplus y_4 \oplus y_5 \oplus y_6$

   $= (x'_0 \oplus x'_4 \oplus x'_5 \oplus x'_6 \oplus x'_7 \oplus 1) \oplus (x'_0 \oplus x'_1 \oplus x'_2 \oplus x'_3 \oplus x'_4) \oplus (x'_1 \oplus x'_2 \oplus x'_3 \oplus x'_4 \oplus x'_5 \oplus 1) \oplus (x'_2 \oplus x'_3 \oplus x'_4 \oplus x'_5 \oplus x'_6 \oplus 1)$ (By Equation (4.12)

   $= x'_2 \oplus x'_3 \oplus x'_5 \oplus x'_7 \oplus 1$

   $= (z_1 \oplus z_4 \oplus z_5 \oplus z_7) \oplus (z_1 \oplus z_4 \oplus z_5 \oplus z_6) \oplus (z_2 \oplus z_4 \oplus z_5) \oplus (z_2 \oplus z_4 \oplus z_5 \oplus z_7) \oplus 1$ (By Equation (4.14))

   $= z_6 \oplus 1 = \overline{z_6}$

   In the same way, we can get:

$$
\begin{aligned}
w_0 &= \overline{z_6} \\
w_1 &= \overline{z_1 \oplus z_2 \oplus z_7} \\
w_2 &= \overline{z_0 \oplus z_5 \oplus z_6 \oplus z_3} \\
w_3 &= z_1 \oplus z_5 \oplus z_6 \oplus z_7 \\
w_4 &= z_0 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \\
w_5 &= z_1 \oplus z_5 \oplus z_6 \\
w_6 &= \overline{z_2 \oplus z_6 \oplus z_7} \\
w_7 &= \overline{z_3 \oplus z_5}
\end{aligned} \tag{4.15}
$$

4. Finally, for the consistency of the other equations in this thesis, we replace w by y, z by x ($x, y \in GF((2^4)^2)$). Let $a = x_5 \oplus x_6 \oplus x_7$, we have:

$$
\begin{aligned}
& y = AFF\_TRAN(x) \\
& -------------------------------- \\
& a = x_5 \oplus x_6 \oplus x_7 \\
& y_0 = \overline{x_6}, \quad y_1 = \overline{x_1 \oplus x_2 \oplus x_7} \\
& y_2 = \overline{x_0 \oplus x_3 \oplus x_5 \oplus x_6}, \quad y_3 = x_1 \oplus a \\
& y_4 = x_0 \oplus x_2 \oplus x_4 \oplus a, \quad y_5 = x_1 \oplus x_5 \oplus x_6 \\
& y_6 = \overline{x_2 \oplus x_6 \oplus x_7}, \quad y_7 = \overline{x_3 \oplus x_5}
\end{aligned} \tag{4.16}
$$

Figure (4.9) describes the proposed subpipelined architecture of subbytes in $GF((2^4)^2)$. The following symbols represent the equation for each arithmetic block in Figure (4.9), except the $\oplus$, which is a simple 4-bit XOR operation. The dashed lines in Figure (4.9) stand for the registers.

x^2 —- Equation (4.3)[32]	$\times e$ —- Equation (4.4)[32]
$\times_{1A}$ —- Equation (4.8) Step A	$\times_{1B}$ —- Equation (4.8) Step B
$\times_{2M}$ and $\times_{3M}$ —- Equation (4.9) Step M	$\times_{2N}$ and $\times_{3N}$ —- Equation (4.9) Step N
x^{-1} —- Equation (4.7)	AFF_TRAN —- Equation (4.16)

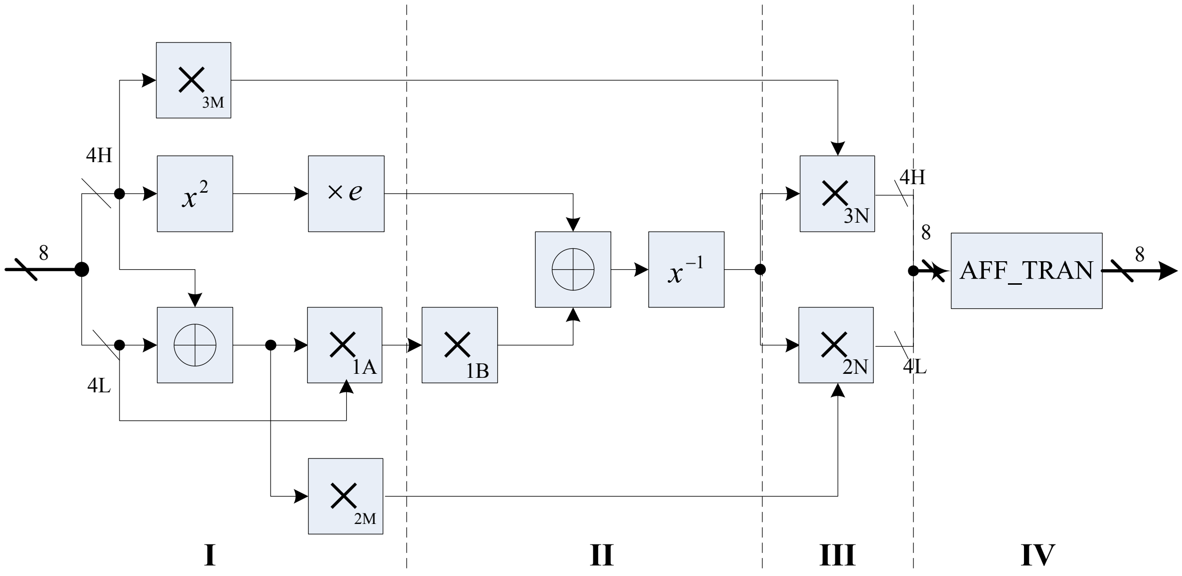


Figure 4.9: Pipelined Subbytes in composite field $GF((2^4)^2)$

Table (4.3) shows the time (ns) and area (slices) cost of each substage (I, II, III, IV in Figure (4.9)) when it runs on different FPGA devices. We cut an AES round unit into 8 substages with the maximum delay determined by part II in subbytes.

Table 4.3: Path Delays and Number of Slices for Spartan2E and Virtex2

Delay(ns):Slices	I	II	III	IV
Spartan2E	10.955:69	11.083:27	10.225:55	10.025:18
Virtex2	7.052:69	7.752:27	6.925:55	6.677:18

4.4.3 *Mixcolumns on* $GF((2^4)^2)$

Mixcolumns is another transformation which involves mathematic operations on $GF((2^4)^2)$. We derive the equations to perform mixcolumns in composite field in this subsection.

Subsection (3.3) describes mixcolumn in finite field $GF(2^8)$. Since $GF((2^4)^2)$ is an isomorphic field to $GF(2^8)$, and in $GF((2^4)^2)$, $\{02\}$ is mapped to $\{26\}$, $\{03\}$ is mapped to $\{27\}$, $\{01\}$ is still $\{01\}$, Equation (3.3) can be mapped directly to Equation (4.17).

$$\begin{bmatrix} 26 & 27 & 01 & 01 \\ 01 & 26 & 27 & 01 \\ 01 & 01 & 26 & 27 \\ 27 & 01 & 01 & 26 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} \tag{4.17}$$

Observing that in $GF((2^4)^2)$, $\{27\} = \{26\} \oplus \{01\}$, Equation (4.17) is equal to Equation (4.18), where $j = 0, 1, 2, 3$:

$$\begin{aligned} S'_{0,j} &= \{26\} \times (S_{0,j} \oplus S_{1,j}) \oplus S_{1,j} \oplus S_{2,j} \oplus S_{3,j} \\ S'_{1,j} &= \{26\} \times (S_{1,j} \oplus S_{2,j}) \oplus S_{0,j} \oplus S_{2,j} \oplus S_{3,j} \\ S'_{2,j} &= \{26\} \times (S_{2,j} \oplus S_{3,j}) \oplus S_{0,j} \oplus S_{1,j} \oplus S_{3,j} \\ S'_{3,j} &= \{26\} \times (S_{0,j} \oplus S_{3,j}) \oplus S_{0,j} \oplus S_{1,j} \oplus S_{2,j} \end{aligned} \tag{4.18}$$

Equation (4.18) presents the mixcolumn transformation of one column of a state. We implement the mixcolumn transformation as the structure in Figure (4.10).

In the following, we derive Equation (4.22) to calculate $x \times 26$ in $GF((2^4)^2)$. That is, we represent the results of $x \times \{02\}$ in $GF((2^4)^2)$:

1. Let, $x,\ y \in GF(2^8)$, using Equation (2.5) to calculate $y = x \times \{02\}$.

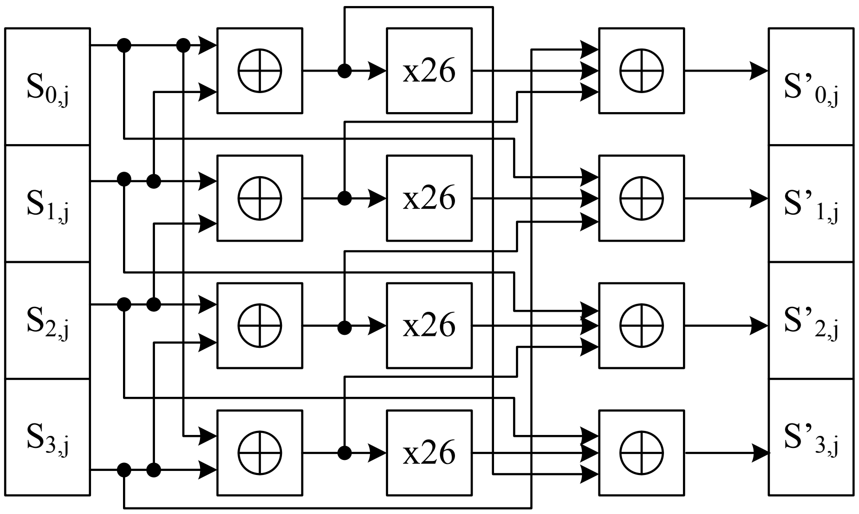


Figure 4.10: $GF((2^4)^2)$ Based Mixcolumns

$$\begin{aligned} &y_0 = x_7,\ y_1 = x_0 \oplus x_7,\ y_2 = x_1 \\ &y_3 = x_2 \oplus x_7,\ y_4 = x_3 \oplus x_7,\ y_5 = x_4 \\ &y_6 = x_5,\ y_7 = x_6 \end{aligned} \tag{4.19}$$

2. Convert y to the field element in $GF((2^4)^2)$. Let w to represent y in $GF((2^4)^2)$ (w is one element in $GF((2^4)^2)$). We have the same equation as Equation (4.13).

3. Next, we use $GF((2^4)^2)$ format to represent x. Let z be the $GF((2^4)^2)$ format of x. z is one element in $GF((2^4)^2)$. By Equation (4.11), we have:

$$\begin{aligned}
x_0 &= z_0 \oplus z_4 \\
x_1 &= z_4 \oplus z_5 \oplus z_7 \\
x_2 &= z_1 \oplus z_4 \oplus z_5 \oplus z_7 \\
x_3 &= z_1 \oplus z_4 \oplus z_5 \oplus z_6 \\
x_4 &= z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7 \\
x_5 &= z_2 \oplus z_4 \oplus z_5 \\
x_6 &= z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_7 \\
x_7 &= z_2 \oplus z_4 \oplus z_5 \oplus z_7
\end{aligned} \tag{4.20}$$

4. We replace x and y with their corresponding $GF((2^4)^2)$ format, z and w, we have:

   $w_0 = y_0 \oplus y_4 \oplus y_5 \oplus y_6$ (By Equation (4.13))

   $= (x_7) \oplus (x_3 \oplus x_7) \oplus (x_4) \oplus (x_5)$ (By Equation (4.19))

   $= x_3 \oplus x_4 \oplus x_5$

   $= (z_1 \oplus z_4 \oplus z_5 \oplus z_6) \oplus (z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7) \oplus (z_2 \oplus z_4 \oplus z_5)$ (By Equation (4.20))

   $= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7$

   By the same method, we derive:

$$\begin{aligned}
w_0 &= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \\
w_1 &= z_0 \oplus z_2 \oplus z_4 \\
w_2 &= z_0 \oplus z_1 \oplus z_3 \oplus z_4 \oplus z_5 \\
w_3 &= z_1 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_6 \\
w_4 &= z_3 \oplus z_6 \\
w_5 &= z_0 \oplus z_3 \oplus z_6 \oplus z_7 \\
w_6 &= z_1 \oplus z_4 \oplus z_7 \\
w_7 &= z_2 \oplus z_5
\end{aligned} \tag{4.21}$$

5. To be consistent, we replace z with x, and replace w with y ($x, y \in GF((2^4)^2)$). In addition, in order to calculate the mixcolumns operations efficiently, we store the intermediate results. Let $a = x_2 \oplus x_4)$, $b = x_3 \oplus x_6 \oplus x_7$, $c = x_1 \oplus x_5$, we have:

$$\begin{aligned}
& y = x \otimes 26, \; x, y \in GF((2^4)^2) \\
& ------------------------------- \\
& a = x_2 \oplus x_4, \quad b = x_3 \oplus x_6 \oplus x_7, \quad c = x_1 \oplus x_5 \\
& y_0 = a \oplus b \oplus x_5, \quad y_1 = a \oplus x_0 \\
& y_2 = c \oplus x_0 \oplus x_3 \oplus x_4, \quad y_3 = c \oplus a \oplus x_6 \\
& y_4 = x_3 \oplus x_6, \quad y_5 = b \oplus x_0 \\
& y_6 = x_1 \oplus x_4 \oplus x_7, \quad y_7 = x_2 \oplus x_5
\end{aligned} \tag{4.22}$$

This mixcolumns architecture (Figure (4.10)) is a 32-bit parallel combinational logic. When synthesized on Virtex2 XC2V2000, it costs 28 1-bit 2-input XOR gates, 44 1-bit 3-input XOR gates, four 1-bit 4-input XOR gates and four 8-bit 2-input XOR gates. The maximum combinational path delay is 7.922ns.

In the above section, we have designed all the modules in AES over field $GF((2^4)^2)$. In this way, each byte of the data needs only one MAP before the initial round and one inverse MAP after the last round.

4.4.4 Subpipelined Keyschedule

There are two approaches to implement keyschedule: (1) pre-calculated keyschedule and (2) on-the-fly keyschedule. In the pre-calculated keyschedule, the $(Nr + 1)$ 128-bit roundkeys are generated before the encryption or decryption begins and stored in the memory. The addroundkey operation accesses the roundkeys by referring the corresponding address in the memory. The advantage of this approach is that the keyschedule only needs to be performed once; however, the drawbacks include:

1. The $(Nr+1)$ roundkeys cost $(Nr+1)\times 128$ bits memory space;

2. The cipherkey cannot change frequently. Every time it changes, the roundkeys must be recalculated.

In this thesis, we propose a new 32-bit on-the-fly keyschedule in composite field ($GF((2^4)^2)$) with 128-, 192-, 256-bit key sizes, where each 128-bit roundkey is generated at every four clock cyles (32-bit at each clock). This is suitable for our 32-bit encryption architecture.

Table (4.1) shows the 32-bit roundkeys at each clock cycle. The following list explains this table for the three key sizes.

- When key size=128 bits, Nr=10, it generates 11 128-bit roundkeys for both block A and B from cycles 0 to 87.

  The roundkeys for block A:

  roundkey[0]={KA(0), KA(1), KA(2), KA(3)}

  roundkey[1]={KA(4), KA(5), KA(6), KA(7)}

  ......

  roundkey[10]={KA(40), KA(41), KA(42), KA(43)}

  The roundkeys for block B:

  roundkey[0]={KB(0), KB(1), KB(2) KB(3)}

  roundkey[1]={KB(4), KB(5), KB(6), KB(7)}

  ......

  roundkey[10]={KB(40), KB(41), KB(42), KB(43)}

- When key size=192, Nr=12, it generates 13 roundkeys for both block A and B from cycles 0 to 103.

The roundkeys for block A:

roundkey[0]={KA(0), KA(1), KA(2), KA(3)}

roundkey[1]={KA(4), KA(5), KA(6), KA(7)}

......

roundkey[12]={KA(48), KA(49), KA(50), KA(51)}

The roundkeys for block B:

roundkey[0]={KB(0), KB(1), KB(2), KB(3)}

roundkey[1]={KB(4), KB(5), KB(6), KB(7)}

......

roundkey[12]={KB(48), KB(49), KB(50), KB(51)}

- When key size=256, Nr=14, it generates 15 roundkeys for both block A and B from cycles 0 to 119.

  The roundkeys for block A:

  roundkey[0]={KA(0), KA(1), KA(2), KA(3)}

  roundkey[1]={KA(4), KA(5), KA(6), KA(7)}

  ......

  roundkey[14]={KA(56), KA(57), KA(58), KA(59)}

  The roundkeys for block B:

  roundkey[0]={KB(0), KB(1), KB(2), KB(3)}

  roundkey[1]={KB(4), KB(5), KB(6), KB(7)}

  ......

  roundkey[14]={KB(56), KB(57), KB(58), KB(59)}

Because we are using the on-the-fly keyschedule, keyschedule and encryptor are sharing the same clock, which means the general frequency is determined by the maximum delay in both keyschedule and encryptor modules. To achieve an efficient pipelining, proper division in keyschedule is as important as in encryptor. We know that subword is the most costly part in keyschedule. In order to make the same maximum delay in both modules, we implement subword in the same way as subbytes in encryptor.

In keyschedule module, rotword rearranges the position of each byte without changing its value, hence the sequence of rotword and subword can be changed. We do the subword operation before rotword to save one multiplexer in keyschedule 256.

All mathematic operations in keyschedule are transformed into field $GF((2^4)^2)$. Subword shares the same structure as in subbytes. Xorrcon is a simples XOR operation with a round constant, which is initially $\{01\}$ and multiplied by $\{02\}$ each *keyschedule round*. Keyschedule round is defined in this way. It begins when $clk\_counter = 0$. If key size is 128, keyschedule round cycle is four; if key size is 192, keyschedule round cycle is six; if key size is 256, keyschedule round cycle is eight. As explained in Subsection (4.4.3), in $GF((2^4)^2)$, $\{01\}$ is still $\{01\}$, $\{02\}$ is mapped to $\{26\}$. We can use Equation (4.22) to generate round constant for each keyschedule round in $GF((2^4)^2)$.

This keyschedule has three key size options: $Key128$, $Key192$ and $Key256$. In the following section, we discuss the generation of roundkeys in details for these three key size options. In the rest of the chapter, $roundkey_{32}$ stands for 32-bit roundkey for each clock cycle, $roundkey$ stands for 128-bit roundkey for a round of AES.

Key128

When key size is 128 bits, the encryptor round count is ten. Two blocks A and B need 22 $roundkeys$. Figure (4.11) illustrates the architecture of keyschedule when key size is 128 bits.

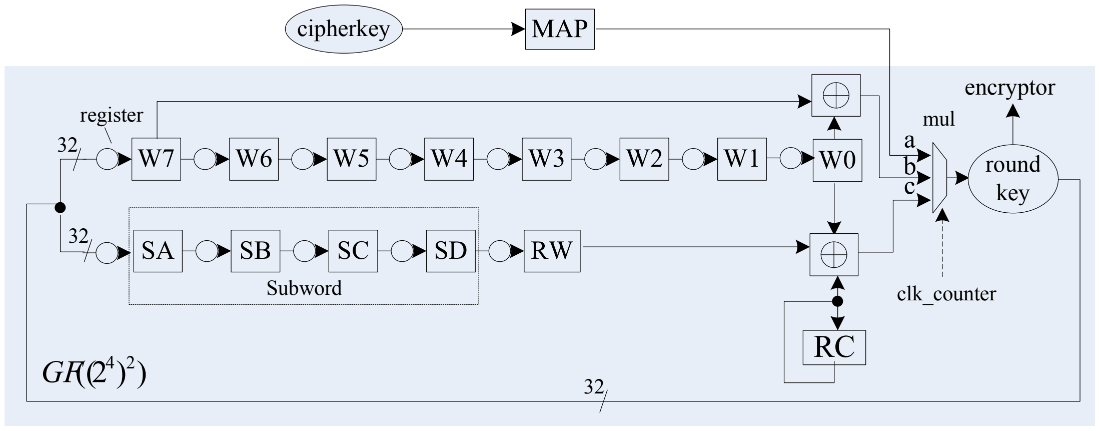


Figure 4.11: Architecture of Keyschedule 128

In our design, the first step is to map (MAP) cipherkey from $GF(2^8)$ to $GF((2^4)^2)$. After that, it performs its isomorphic functions in $GF((2^4)^2)$. The output of keyschedule are $roundkey_{32}s$ represented in $GF((2^4)^2)$. They are the exact format required in encryption where the message blocks are also represented in $GF((2^4)^2)$, hence no inverse MAP follows roundkey.

In Figure (4.11), W7, ..., W0 are 32-bit words separated by eight registers, which are used to store the previous eight $roundkey_{32}s$. SA, SB, SC and SD are the results of the 4 parts of subword. We place three registers among the four substages in subword, same as in Figure (4.9). RW is the outcome of rotword. RC generates the round constant for xorrcon in $GF((2^4)^2)$. mul is a 3-to-1 multiplexer controlled by $clk\_counter$, which is the same signal as in Figure (4.4) and Table (4.1). There are three different cases (a, b, c) to generate the current $roundkey_{32}$. Table (4.4) explains the value of each register in Figure (4.11) during the first 15 clock cycles. In this table, the row titled *mul* stands for the multiplexer in Figure (4.11). *a*, *b* and *c* are the three cases. In the following expressions, $roundkey_{32}[i]$ stands for the cell of this table with row ID of $roundkey_{32}$ and column ID of $clk\_counter = i$.

Case a: $clk\_counter < 8$ (initial round):

$roundkey_{32}[clk\_counter] = MAP(cipherkey[clk\_counter]);$

Case b: $clk\_counter >= 8$ and $clk\_counter\ mod 4 \neq 0$:

$roundkey_{32}[clk\_counter] = roundkey_{32}[clk\_counter-1] \oplus roundkey_{32}[clk\_counter-8]$

$roundkey_{32}[clk\_counter-1]$ is stored in $W7$

$roundkey_{32}[clk\_counter-8]$ is stored in $W0$;

Case c: $clk\_counter >= 8$ and $clk\_counter\ mod\ 4 = 0$:

$roundkey_{32}[clk\_counter] = rotword(subword(roundkey_{32}[clk\_counter-5]))$

$\oplus roundkey_{32}[clk\_counter-8] \oplus Rcon$

$rotword(subword(roundkey_{32}[clk\_counter-5]))$ is stored in RW

$roundkey_{32}[clk\_counter-8]$ is stored in $W0$.

We give examples for each case.

- When $clk\_counter = 0$, the first $roundkey_{32}$ is MAPed from the first word of cipherkey. $roundkey_{32}[0] = KA(0)$. (**Case a**)

- when $clk\_counter = 1$, the second $roundkey_{32}$ is MAPed from the second word of cipherkey. $roundkey_{32}[1] = KA(1)$. KA(0) is moved to W7. In the mean time, KA(0) finished the first part of subword and is stored in SA. (**Case a**)

  ......

- when $clk\_counter = 8$, KA(3) is moved in to RW, which means $KA(3) = rotword(subword(KA(3)))$. Now $roundkey_{32}[8] = KA(4) = KA(3) \oplus KA(0) \oplus Rcon$. (**Case c**)

Table 4.4: Key128 Roundkey Sequence

cipherkey	KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)								
clk_counter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
mul	a								c	b			c	b		
roundkey32	KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)	KB(6)	KB(7)
w7		KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)	KB(6)
w6			KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)
w5				KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)
w4					KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)
w3						KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)
w2							KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)
w1								KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)
w0									KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)
SA		KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)	KB(6)
SB			KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)
SC				KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)
SD					KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)
RW						KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)

- when $clk\_counter = 9$, KA(4) is put into W7 and SA (after finishes the first part of subword). $roundkey_{32}[9] = KA(5) = KA(4) \oplus KA(1)$. **(Case b)**

  ......

Key192

When key size is 192 bits, the encryptor round count is 12. Block A and block B need 104 $roundkey_{32}$s. Cipherkey size does not affect the function entities. So it shares the same subword, rotword and xorrcon as in Figure (4.11). However, due to key size, the structure becomess more complex. When key size is 192, the keyschedule round cycle is six while the encryptor cycle is still four. This cycle difference requires extra treatment for the input of subword. We can see in Figure (4.11) that, when key size is 128 bits, the input of subword is the roundkey. But when key size is 192 bits, the input of subword is classified into three cases. We use multiplexer $mul1$ in Figure (4.12) to choose the input from case x, y and z.

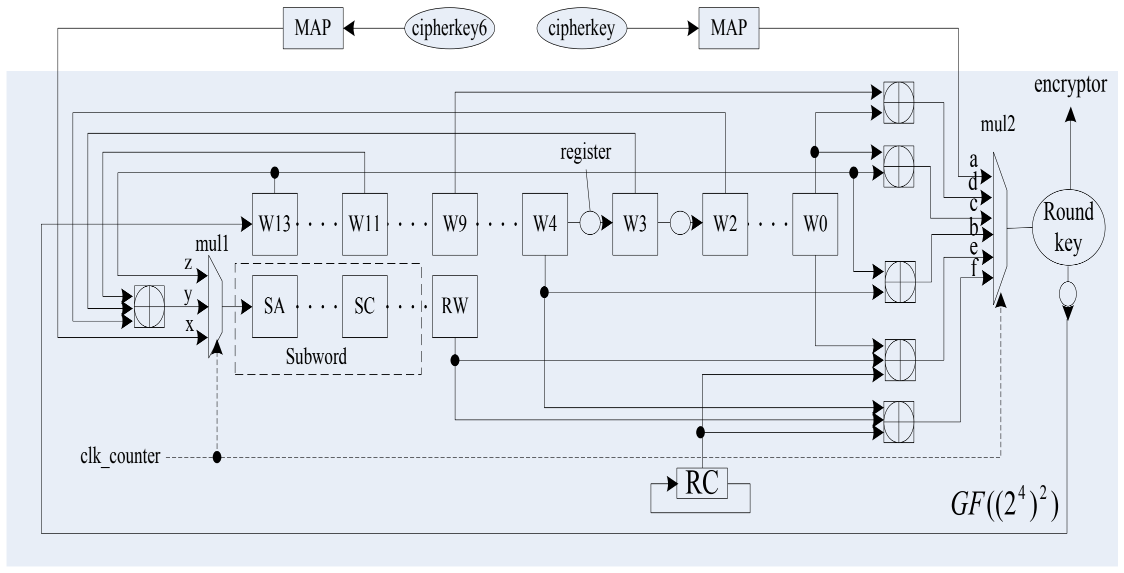


Figure 4.12: Architecture of Keyschedule 192

Table 4.5: Key192 Roundkey Sequence

cipherkey	KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)			KB(4)	KB(5)					
cipherkey6							KA(5)				KB(5)								
mul2	a										f	b	a		d	c	e	b	
mul1							x				x						z	y	
clk_counter	0	1	2	3	4	5	6	7	8	9	10	11	12	13	16	17	24	30	31
roundkey32	KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(5)	KA(8)	KA(9)	KA(12)	KB(14)	KB(15)
w13		KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(4)	KB(7)	KA(8)	KB(11)	KB(13)	KB(14)
w12			KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KA(7)	KB(6)	KB(7)	KB(10)	KB(12)	KB(13)
w11				KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KA(6)	KB(5)	KB(6)	KB(9)	KA(15)	KB(12)
w10					KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(5)	KB(4)	KB(5)	KB(8)	KA(14)	KA(15)
w9						KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(4)	KA(7)	KB(4)	KA(11)	KA(13)	KA(14)
w8							KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KB(3)	KA(6)	KA(7)	KA(10)	KA(12)	KA(13)
w7								KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KB(2)	KA(5)	KA(6)	KA(9)	KB(11)	KA(12)
w6									KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(1)	KA(4)	KA(5)	KA(8)	KB(10)	KB(11)
w5										KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KB(3)	KA(4)	KB(7)	KB(9)	KB(10)
w4											KA(0)	KA(1)	KA(2)	KA(3)	KB(2)	KB(3)	KB(6)	KB(8)	KB(9)
w3												KA(0)	KA(1)	KA(2)	KB(1)	KB(2)	KB(5)	KA(11)	KB(8)
w2													KA(0)	KA(1)	KB(0)	KB(1)	KB(4)	KA(10)	KA(11)
w1														KA(0)	KA(3)	KB(0)	KA(7)	KA(9)	KA(10)
w0															KA(2)	KA(3)	KA(6)	KA(8)	KA(9)
SA							KA(5)				KB(5)						KB(11)	KA(17)	
SB								KA(5)				KB(5)							KA(17)
SC									KA(5)				KB(5)						
SD										KA(5)				KB(5)					
RW											KA(5)						KA(11)		

Table (4.5) shows the value of each register in Figure (4.12) during the first 32 clock cycles. Row $mul1$ and $mul2$ correspond to these two multiplexers. In the following section, we explain the two multiplexers ($mul1$ and $mul2$) in Figure (4.12).

1. Multiplexer 1 ($mul1$)

   Case x: ($clk\_counter = 6$ or 10) : $SA = MAP(cipherkey6)$; (Cipherkey6 is the sixth 32-bit of the 192-bit cipherkey. We can see from Table (4.5) that, when $clk\_counter = 10$, $KA(5)$ must finish subword and rotword, so that we can produce $KA(6)$, where $KA(6) = KA(5) \oplus rotword(subword(KA(5))) \oplus Rcon$. Because it needs five clock cycles to complete $rotword(subword(KA(5))), KA(5)$ must be shifted into SA when $clk\_counter = 6$. That's why we need cipherkey6 to provide $KA(5)$.)

   Case y: ($clk\_counter$ mod $24 = 6$ or 10) and ($clk\_counter > 23$) : $SA = W11 \oplus W2 \oplus W3$; (One example is when $clk\_counter = 30$, $KA(17)$ needs to be shifted into SA. Because $KA(17)$ is not stored in any register, we need to calculate it from the existing data. $KA(17) = KA(16) \oplus KA(11) = KA(15) \oplus KA(10) \oplus KA(11)$. KA(15), KA(10) and KA(11) are stored in register W11, W2 and W3, respectively.)

   Case z: ($clk\_counter$ mod $24 = 0$ or 20) and ($clk\_counter \neq 0$) : $SA = W13$;

   This is why we need multiplexer $mul1$ to differentiate three cases for the input of subword when key size is 192 bits.

2. Multiplexer 2 ($mul2$)

   - Generate $roundkey_{32}$s from cipherkey directly, KA(i), KB(i), i = 0,...,5

Case a: $(clk\_counter < 10\ or = 12, 13) : roundkey_{32}[clk\_counter] = MAP(cipherkey$ $[clk\_counter])$ (Because $roundkey_{32}$ is generated when it is needed in encryption, the arrangement of row $cipherkey$ in Table (4.5) is determined by encryptor)

- Generate $roundkey_{32}$s KA(i), KB(i), $i \geq 8$ and $i\ mod\ 6 \neq 0$ (Because of the round cycle difference between keyschedule and encryptor, we need to classify it into three sub-cases, based on the value ($clk\_counter\ mod\ 4$). Table (4.5) shows these three sub-cases (b, c and d), where $roundkey_{32}$s are generated by the formula $KA/B(i) = KA/B(i-1) \oplus KA/B(i-6)$.)

Case b: ($clk\_counter\ mod\ 4 = 3\ or\ 0$) and ($clk\_counter > 7$) :

$roundkey_{32}[clk\_counter] = roundkey_{32}[clk\_counter-1] \oplus roundkey_{32}[clk\_counter-10]$;

$roundkey_{32}[clk\_counter-1]$ is stored in register W13;

$roundkey_{32}[clk\_counter-10]$ is stored in register W4;

Case c: ($clk\_counter\ mod\ 4 = 2$) and ($clk\_counter > 7$) :

$roundkey_{32}[clk\_counter] = roundkey_{32}[clk\_counter-1] \oplus roundkey_{32}[clk\_counter-14]$;

$roundkey_{32}[clk\_counter-1]$ is stored in register W13;

$roundkey_{32}[clk\_counter-14]$ is stored in register W0;

Case d: ($clk\_counter\ mod\ 4 = 1$) and ($clk\_counter\ mod\ 24 > 7$) :

$roundkey_{32}[clk\_counter] = roundkey_{32}[clk\_counter-5] \oplus roundkey_{32}[clk\_counter-14]$;

$roundkey_{32}[clk\_counter-5]$ is stored in register W9;

$roundkey_{32}[clk\_counter-14]$ is stored in register W0;

- Generate $roundkey_{32}$s KA(i), KB(i), $i \geq 8$ and $i\ mod\ 6 = 0$ (The sub-cases are caused by the same reason as the above case. The following two sub-cases are based on the formula $KA/B(i) = rotword(subword(KA/B(i-1))) \oplus KA/B(i-6) \oplus Rcon$)

Case e: ($clk\_counter \ mod \ 24 = 0 \ or \ 4$) and ($clk\_counter > 7$) :

$roundkey_{32}[clk\_counter] = rotword(subword(roundkey_{32}[clk\_counter-1]))$
$\oplus roundkey_{32}[clk\_counter-14] \oplus RC$;

$rotword(subword(roundkey_{32}[clk\_counter-1]))$ is stored in register RW;

$roundkey_{32}[clk\_counter-14]$ is stored in register W0;

Case f: ($clk\_counter \ mod \ 24 = 10 \ or \ 14$) :

$roundkey_{32}[clk\_counter] = rotword(subword(roundkey_{32}[clk\_counter-1]))$
$\oplus roundkey_{32}[clk\_counter-10] \oplus RC$;

$rotword(subword(roundkey_{32}[clk\_counter-1]))$ is stored in register RW;

$roundkey_{32}[clk\_counter-10]$ is stored in register W4;

Table (4.5) lists instances for each case for both $mul1$ and $mul2$ during the first 32 clock cycles.

- When $clk\_counter = 0$, $roundkey_{32}[0] = KA(0)$, which is MAPed from cipherkey(**Case a**);

  ......

- When $clk\_counter = 6$, $roundkey_{32}[6] = KB(2)$, which is MAPed from cipherkey(**Case a**). $SA = KA(5)$, where KA(5) is MAPed from cipherkey6 and shifted into SA after finished subword's first part(**Case x**);

  ......

- When $clk\_counter = 10$, $roundkey_{32}[10] = KA(6) = rotword(subword(KA(5))) \oplus KA(0) \oplus Rcon$ (**Case f**);

- When $clk\_counter = 11$, $roundkey_{32}[11] = KA(7) = KA(6) \oplus KA(1)$ (**Case b**);

  ......

- When $clk\_counter = 16$, $roundkey_{32}[16] = KA(8) = KA(7) \oplus KA(2)$ (**Case d**);

- When $clk\_counter = 17$, $roundkey_{32}[17] = KA(9) = KA(8) \oplus KA(3)$ (**Case c**);

  ......

- When $clk\_counter = 24$, $roundkey_{32}[24] = KA(12) = rotword(subword(KA(11))) \oplus KA(6) \oplus Rcon$ (**Case e**). $SA = KB(11)$, where KB(11) is shifted into SA after finished subword's first part (**Case z**);

  ......

- When $clk\_counter = 30$, $SA = KA(17) = KA(16) \oplus KA(11) = (KA(15) \oplus KA(10)) \oplus KA(11)$ (**Case y**);

  ......

Key256

Keyschedule 256 is slightly different from keyschedule 128. The keyschedule round cycle is eight clock cycles. As shown in Figure (4.13):

There are four different cases to generate $roundkey_{32}$s:

Case a: $(clk\_counter < 16)$:

$roundkey_{32}[clk\_counter] = MAP(cipherkey[clk\_counter]);$

Case b: $(clk\_counter \geq 16)$ and $(clk\_counter\ mod\ 4 \neq 0)$:

$roundkey_{32}[clk\_counter] = roundkey_{32}[clk\_counter - 1] \oplus roundkey_{32}[clk\_counter - 16];$

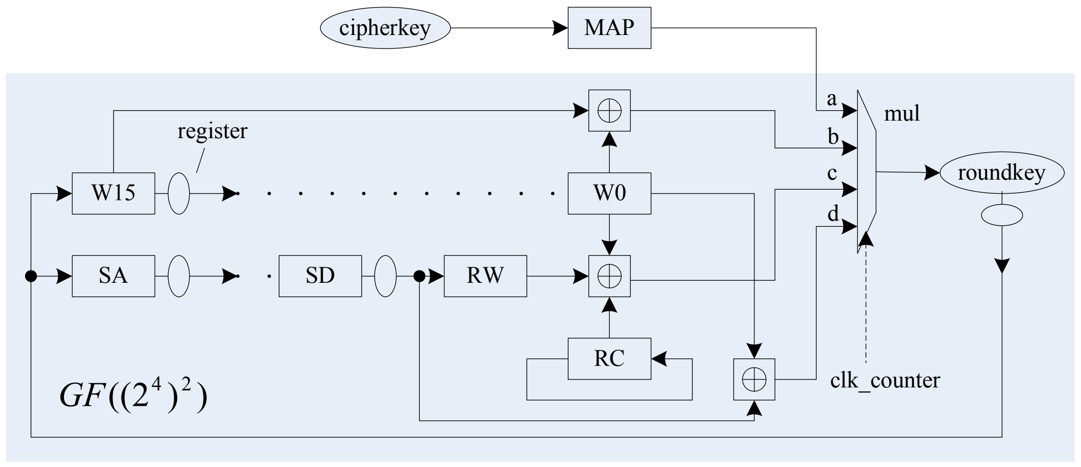


Figure 4.13: Architecture of Keyschedule 256

$roundkey_{32}[clk\_counter-1]$ is stored in register W15;

$roundkey_{32}[clk\_counter-16]$ is stored in register W0;

Case c: $(clk\_counter \geq 16)$ and $(clk\_counter\ mod\ 8 = 0)$:

$roundkey_{32}[clk\_counter] = rotword(subword(roundkey_{32}[clk\_counter-5]))RW$

$\oplus roundkey_{32}[clk\_counter-16] \oplus RC$;

$rotword(subword(roundkey_{32}[clk\_counter-5]))$ is stored in register RW;

$roundkey_{32}[clk\_counter-16]$ is stored in register W0;

Case d: $(clk\_counter \geq 16)$ and $(clk\_counter\ mod\ 4 = 0)$ and $(clk\_counter\ mod\ 8 \neq 0)$:

$roundkey_{32}[clk\_counter] = subword(roundkey_{32}[clk\_counter-5]) \oplus roundkey_{32}[clk\_counter-16]$;

$subword(roundkey_{32}[clk\_counter-5])$ is stored in register RW;

$roundkey_{32}[clk\_counter-16]$ is stored in register W0.

This is why we change the sequence of subword and rotword. Puting rotword before subword saves one multiplexer when key size is 256 bits.

Table (4.4.4) gives instances for each case.

- When $clk\_counter = 0$, $roundkey_{32}[0] = KA(0)$, where KA(0) is MAPed from cipherkey (**Case a**);

  ......

- When $clk\_counter = 16$, $roundkey_{32}[16] = KA(8) = rotword(subword(KA(7))) \oplus KA(0) \oplus Rcon$ (**Case c**);

- When $clk\_counter = 17$, $roundkey_{32}[17] = KA(9) = KA(8) \oplus KA(1)$ (**Case b**);

  ......

- When $clk\_counter = 24$, $roundkey_{32}[12] = KA(12) = subword(KA(11)) \oplus KA(4)$ (**Case d**);

  ......

Table 4.6: Key256 Roundkey Sequence

cipherkey	KA(0)		KB(7)							
mul		a		c	b			c	d	d
clk_reg	0		15	16	17	18	19	20	24	28
roundkey32	KA(0)		KB(7)	KA(8)	KA(9)	KA(10)	KA(11)	KB(8)	KA(12)	KB(12)
w15			KB(6)	KB(7)	KA(8)	KA(9)	KA(10)	KA(11)	KB(11)	KA(15)
w14			KB(5)	KB(6)	KB(7)	KA(8)	KA(9)	KA(10)	KB(10)	KA(14)
w13			KB(4)	KB(5)	KB(6)	KB(7)	KA(8)	KA(9)	KB(9)	KA(13)
w12			KA(7)	KB(4)	KB(5)	KB(6)	KB(7)	KA(8)	KB(8)	KA(12)
w11			KA(6)	KA(7)	Kb(4)	KB(5)	KB(6)	KB(7)	KA(11)	KB(11)
w0				KA(0)	KA(1)	KA(2)	KA(3)	KB(0)	KA(4)	KB(4)
SA			KB(6)	KB(7)	KA(8)	KA(9)	KA(10)	KA(11)	KB(11)	KA(15)
SB			KB(5)	KB(6)	KB(7)	KA(8)	KA9	KA(10)	KB(10)	KA(14)
SC			KB(4)	KB(5)	KB(6)	KB(7)	KA(8)	KA(9)	KB(9)	KA(13)
SD			KA(7)	KB(4)	KB(5)	KB(6)	KB(7)	KA(8)	KB(8)	KA(12)
RW			KA(6)	KA(7)	KB(4)	KB(5)	KB(6)	KB(7)	*KA(11)*	*KB(11)*

Chapter 5

Implementation Performance And Comparison

Literature regarding hardware implementation of AES have been published. The comparison tables listed in the literatures are synthesized by various design tools on different FPGA chips. Although the difficulty of comparison about FPGA implementations was reported, there is still no proved measure to get a real fair comparison among different architectures. Even for the devices from the same company (Xilinx), different families use different technology which leads to different frequency. For example, a slice in Virtex 5 has four LUTs (Look Up Tables) instead of two in previous families [6], which leads to different area cost (number of slice).

Since AES standard includes encryption, decryption and keyschedule with three key sizes, it is up to the designers to choose which function they would like to realize. Obviously, more functions need more resource. Hence it is reasonable to compare architectures providing similar functions.

In this chapter, we first classify previous AES architectures into different categories and then use tables to compare their performance.

1. **Encryption and Decryption:** AES architectures include encryption and decryption units. In [1, 3, 5, 8, 9, 17, 33, 22, 27, 28], they provide functions for both encryption and decryption. As a symmetric algorithm, encryption and decryption share same units. With the parameter indicated by the user, it executes encryption or decryption exclusively. Some other AES architectures only focus on encryption [2, 4, 11, 12, 25, 26, 31].

2. **Key Sizes:** AES uses data size of 128 bits but offers three key sizes (128, 192 and 256 bits). 128-bit is the most common choice in the reported designs [3, 4, 5, 9, 12,

33, 26, 27, 28, 34]. However, as reconfigurability is one of most important factors for FPGA implementations, options for all three key sizes are included in a number of designs [1, 2, 17, 22].

3. **Key Expansion:** The keyschedule in AES generates roundkeys for each round. The roundkeys can be previously calculated and stored in memory [1, 2, 3, 5, 22, 27]. This method results in an acceptable initial delay when the data size is relatively large compared with the key size. A more flexible approach is the on-the-fly keyschedule [4, 9, 12, 17, 33, 26, 28, 34] which conducts an on-line calculation of roundkeys for each 128-bit data block. On-the-fly keyschedule affects the general frequency as both the data unit and key unit share the same clock, especially when it is employed for all the three key sizes. There are also some architectures that do not include keyschedule [8, 11, 13, 31].

4. **BRAM based S-Box and combinational logic based S-Box:** Different approaches for S-Box implementation have obvious impact on AES performance. BRAM based approaches [5, 8, 13, 17, 26, 27] are preferred when low area cost is required. It saves the slices required in combinational logic based approach. Hence it is not reasonable to compare the ratio of throughput/slice between BRAM-based S-Box and combinational logic based S-Box [1, 2, 9, 11, 12, 13, 33, 22, 25, 28, 31, 34]. Good et al. [9] used a term (32bits/slice) to convert number of BRAMs to number of slices required to implement the equivalent distributed memory. But, the estimates vary between 8 and 32 bits/slice depending on the functionality required. In this thesis, we only compare our design throughput/slice with non-BRAM implementations.

The above four categories summarize the majors factors affecting the performance in hardware implementation of AES. Table 5.1 compares the performance of the architectures

Table 5.1: Comparisons of BRAMs Based AES Architecture

Design	Device	Frequency (MHz)	Slices	BRAMs	Throughput (Mbps)
Samanta et al. [27]	VIRTEX2 2V6000	76.699	1051	11	111.56
Chodowiec [8]	VIRTEX XCV1000	95	12600	80	12100
Chodowiec et al. [5]	SPARTAN2 XC2S30	60	222	3	166
Chang et al. [4]	SPARTAN2 XC2S30	38.50	200	2	38
Saggese et al. [26]	VIRTEXE XCV2000E	142	648	10	1820
McLoone et al. [17]	VIRTEXE XCV3200E	54.35	2222	100	6956

using BRAMs. Table 5.2 compares the architectures without BRAMs. Table 5.3 summarizes the functions provided by these architectures.

Among the architectures using BRAMs, Chodowiec [8] employed fully unrolled subpipelining achieving the highest throughput with the largest resource cost. Recently, Chodowiec et al. made a compact design costing 222 slices in a Spartan2 device offering a throughput of 166Mbps [5].

In our proposed architecture, we do not use BRAM. In Table 5.2, it can be seen that Good et al. achieves the highest throughput of 25.107 Gbps on Spartan3 XC3S2000. It employs fully parallel loop unrolled architecture which calculates multiplicative inverse of each byte over composite field $GF((2^4)^2)$. It also gets the frequency of 196.1MHz. But it only deals with 128-bit key size and costs 17425 slices. Another fully unrolled architecture is proposed by Zhang et al. [34]. This design used number-of-gates-in-critical-path to place the pipeline cuts. It subpipelines a round into seven substages and achieves 21.556 Gbps with the throughput/slice ratio of 1.956.

Compared with the previous architectures, our design focuses on the low cost, non-

Table 5.2: Comparisons of Non-BRAMs Architectures

Design	Device	Frequency (MHz)	Area (Slices)	Throughput (Mbps)	Mbps/ Slice
Good et al. [9]	SPARTAN3 XC3S2000	196.1	17425	25107	1.441
Zhang et al. [34]	VIRTEXE XCV1000E	168.4	11022	21560	1.956
Jarvinen et al. [12]	VIRTEX XC2V2000	139.1	10750	17800	1.656
Mucci et al. [11]	VIRTEX2P XV2VP20	169.1	9446	21640	2.291
Lemsitzer et al. [13]	VIRTEX4 FX100	110	7300	3500	0.479
Bulens et al. [3]	SPARTAN3	150	1800	1700	0.944
Standaert et al. [31]	VIRTEXE XCV1000E	167	1767	2085	1.180
Pramstaller et al. [22]	VIRTEXE XCV1000E	161	1125	215	0.191
Our Desisgn	VIRTEX2 XC2V2000E	277.4	523	807	1.543
Alam et al. [1]	VIRTEXE XCV1000E	135	510	432	0.847

BRAM implementations. There were not many literatures in the low-cost AES designs. Pramstaller et al. proposed a compact design costing 1125 slices in [22]. Its pre-calculate key generator can deal with three key sizes. Standaert et al. [31] made a single encryption architecture with 1767 slices which provides Gbps-level throughput. Alam et al. [1] reported a design including encryption, decryption and on-the-fly keyschedule for 3 key sizes, which achieves 432 Mbps with the frequency of 135MHz.

Compared with similar previous works, our proposed low-cost and efficient AES architecture only uses 523 slices, and achieves the throughput of 806Mbps when implemented in Virtex 2 XCV2V2000. The throughput/area ratio is 1.543, which is relatively high in low-cost designs ($<$ 2000 slices). The proposed design can be efficiently applied in computing-resources restricted environments, such as wireless devices and embedded devices.

Table 5.3: Comparisons of AES Architectures Functions

Design	Encryption	Decryption	KeySchedule	KeySize	BRAMs
Samanta [27]	●	●	Pre-Calculate	128	●
Chodowiec [8]	●	●			●
Chodowiec et al. [5]	●	●	Pre-Calculate	128	●
Satoh et al. [28]	●	●	On-The-Fly	128	
Hodjat et al. [11]	●				
Jarvinen et al. [12]	●		On-The-Fly	128	
Good et al. [9]	●	●	On-The-Fly	128	
Zhang et al. [34]	●		On-The-Fly	128	
Chang et al. [4]	●		On-The-Fly	128	●
Pramstaller et al. [22]	●	●	Pre-Calculate	128/192/256	
Saggese et al. [26]	●		On-The-Fly	128	●
Standaert et al. [31]	●				
McLoone et al. [17]	●	●	On-The-Fly	128/192/256	●
Lemsitzer et al. [13]	●				
Bulens et al. [3]	●	●	Pre-Calculate	128	●
Alam et al. [1]	●		On-The-Fly	128/192/256	
Our Design	●		On-The-Fly	128/192/256	

Chapter 6

Conclusion

AES is an important and popular cryptographic algorithm to secure the information and data transmission. In this thesis, we propose a compact reconfigurable FPGA architecture for the AES implementation.

The 32-bit single round unit design results in low area cost, which makes it suitable for low-end devices. The combinational logic approach of AES implementation eliminates the need for BRAMs. Full composite field ($GF((2^4)^2)$) based design decreases hardware complexity of arithmetic operations in AES. We apply subpipelining technology in both encryptor and keyschedule modules to optimize the speed/area ratio, which achieves 1.543Mbps/Slice in Virtex 2 XCV2V2000. Besides, the capability to deal with three key sizes makes our design an efficient reconfigurable architecture of AES.

The throughput of our proposed design achieves 805.8Mbps. It requires less than a quarter of the resources of a Xilinx Spartan2 FPGA, which is one of the smallest FPGA devices. The performance comparison indicates that the proposed AES architecture achieves higher throughput than previous compact designs.

FIPS standard [20] provides an equivalent inverse cipher which switches the sequence of the four transformations in decryption round so that the encryption and decryption can share the same functions, such as the multiplicative inversion in subbytes. In our design, the encryption conducts shiftrows before subbytes. When implementing the equivalent inverse cipher, it only needs to switch the relative sequence of inv-mixcolumns and addroundkey. The positions of inv-shiftrows and inv-subbytes are not changed. The proposed design can be easily modified into an equivalent cipher.

In conclusion, the proposed compact and reconfigurable AES architecture has high throughput and low area cost, which is very useful in the computing restricted environment and wireless devices.

Bibliography

[1] Monjur Alam, Santosh Ghosh, Dipanwita RoyChowdhury, and Indranil Sengupta. Single Chip Encryptor/Decryptor Core Implementation of AES Algorithm. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*, pages 693–698, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Monjur Alam, Sonai Ray, Debdeep Mukhopadhayay, Santosh Ghosh, Dipanwita Roy-Chowdhury, and Indranil Sengupta. An Area Optimized Reconfigurable Encryptor for AES-Rijndael. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1116–1121, San Jose, CA, USA, 2007. EDA Consortium.

[3] Philippe Bulens, Francois-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gael Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In *Progress in Cryptology - AfricaCrypt 2008*, pages 16 – 26. Springer, 2008.

[4] Chi-Jeng Chang, Chi-Wu Huang, Hung-Yun Tai, and Mao-Yuan Lin. 8-bit AES Implementation in FPGA by Multiplexing 32-bit AES Operation. In *ISDPE '07: Proceedings of the The First International Symposium on Data, Privacy, and E-Commerce*, pages 505–507, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Pawel Chodowiec and Kris Gaj. Very Compact FPGA Implementation of the AES Algorithm. In *CHES*, pages 319–333, 2003.

[6] Adrian Cosoroaba. Achieve Higher Performance with Virtex-5 FPGAs. Xilinx, Inc. Available at `http://china.xilinx.com/publications/xcellonline/xcell_59/xc_pdf/p016-018_59-consoroba.pdf`.

[7] J. Daemen and V. Rijmen. AES Proposal: Rijndael. Technical report, National Institute of Standards and Technology (NIST). Available at `http://www.nic.funet.fi/pub/crypt/cryptography/symmetric/aes/nist/Rijndael.pdf`.

[8] Kris Gaj and Pawel Chodowiec. Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. In *AES Candidate Conference*, pages 40–54, 2000.

[9] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2005.

[10] D.H. Green and I.S. Taylor. Irreducible Polynomials over Composite Galois Fields and Their Applications in Coding Techniques. pages 935–939, September 1974.

[11] Alireza Hodjat and Ingrid Verbauwhede. A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 308–309, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Kimmo U. Järvinen, Matti T. Tommiska, and Jorma O. Skyttä. A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 207–215, New York, NY, USA, 2003. ACM.

[13] Stefan Lemsitzer, Johannes Wolkerstorfer, Norbert Felber, and Matthias Braendli. Multi-gigabit GCM-AES Architecture Optimized for FPGAs. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 227–238. Springer, 2007.

[14] Liberatori, M. Otero, F. Bonadero, J.C. Castineira, J.UNMDP, and Mar del Plata. AES-128 Cipher. High Speed, Low Cost FPGA Implementation. pages 195–198, Mar del Plata, 2007. IEEE Computer Society.

[15] Rudolf Lidl and Harald Niederreiter. *Finite Fields (Encyclopedia of Mathematics and its Applications)*. Addison-Wesley, 1983.

[16] Robert J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Pub, 1987.

[17] Máire McLoone and John V. McCanny. High Performance Single-Chip FPGA Rijndael Algorithm Implementations. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 65–76, London, UK, 2001. Springer-Verlag.

[18] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[19] Mike Nelson. Why You Should Use FPGAs in Data Security. Xilinx is an Ideal Platform for Data Security Applications. Storage and Servers. Vertical Markets. Xilinx, Inc. Available at `http://www.xilinx.com/publications/xcellonline/xcell_57/xc_pdf/p054-057_57-secure.pdf`.

[20] NIST. Announcing the ADVANCED ENCRYPTION STANDARD (AES). Available at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[21] Christof Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics – University of Essen, 1994.

[22] Norbert Pramstaller, Stefan Mangard, Sandra Dominikus, and Johannes Wolkerstorfer. Efficient AES Implementations on ASICs and FPGAs. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *AES Conference*, volume 3373 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.

[23] Norbert Pramstaller and Johannes Wolkerstorfer. A Universal and Efficient AES Co-processor for Field Programmable Logic Arrays. 3203/2004:565–574, 2004.

[24] Vincent Rijmen. Efficient Implementation of the Rijndael S-box. Available at `http://www.comms.scitech.susx.ac.uk/fft/crypto/rijndael-sbox.pdf`.

[25] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 171–184, London, UK, 2001. Springer-Verlag.

[26] Giacinto Paolo Saggese, Antonino Mazzeo, Nicola Mazzocca, and Antonio G. M. Strollo. An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm. In *FPL*, pages 292–302, 2003.

[27] Sounak Samanta. FPGA Implementation of AES Encryption and Decryption. Sardar Vallabhbhai National Institute of Technology, Surat. Available at `http://www.design-reuse.com/articles/13981/fpga-implementation-of-aes-encryption-and-decryption.html`.

[28] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 239–254, London, UK, 2001. Springer-Verlag.

[29] Lin Shu and Costello Daniel J. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 1983.

[30] William Stallings. *Cryptography and Network Security-Principles and Practices (Fourth Edition)*. Pearson Prentice hall, 2006.

[31] François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater, and Jean-Didier Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In *CHES*, pages 334–350, 2003.

[32] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. An ASIC Implementation of the AES SBoxes. In *CT-RSA '02: Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, pages 67–78, London, UK, 2002. Springer-Verlag.

[33] Namin Yu and H.M. Heys. Investigation of Compact Hardware Implementation of the Advanced Encryption Standard. pages 1069– 1072, 2005.

[34] Xinmiao Zhang and Keshab K. Parhi. High-speed VLSI architectures for the AES algorithm. *IEEE Trans. Very Large Scale Integr. Syst.*, 12(9):957–967, 2004.