國立交通大學

電機與控制工程學系

碩 士 論 文

具有可變係數之 AES 加解密器

之矽智產設計與晶片實現

IP-based design and chip implementation of the AES

coprocessor with configurable parameters

研究生：白宗堯

指導教授：吳炳飛　教授

中 華 民 國 九 十 五 年 七 月

具有可變係數之 AES 加解密器之矽智產設計與晶片實現

IP-based design and chip implementation of the AES coprocessor with configurable parameter

研究生：白宗堯　　Student：Tsung-Yao Pai

指導教授：吳炳飛 教授　　Advisor：Prof. Bing-Fei Wu

國立交通大學

電機與控制工程學系

碩士論文

A Thesis

Submitted to Department of Electrical and Control Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Electrical and Control Engineering

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

具有可變係數之 AES 加解密器
之矽智產設計與晶片實現

學生： 白宗堯　　指導教授：吳炳飛　教授

國立交通大學電機與控制工程學系(研究所)碩士班

摘　　要

在此篇論文中，我們提出了一個具有可變係數的 AES (Configurable AES)加解密器，使得在不同的 $m(x)$、$c(x)$ 和 affine transformation 係數選擇之下，可產生多變的 AES 演算法，藉以進一步的提高系統的安全性。並由於我們所提出之硬體實現仍能具有高效能的表現，讓此構想可以跟到上網路傳輸速度的進步，運到用 Gigabit 的光纖與乙太網路安全晶片上。在規格上，除了可調變係數之外，並支援 128, 192, 256-bit 三種金鑰長度以及 ECB, CBC 兩種加密模式。對於加解密過程中所需之金鑰，我們也提出了一種可同步計算金鑰的電路，而不需使用額外的記憶體來儲存金鑰。此外，為了降低硬體成本和提升效率，我們採取 Composite Field Arithmetic 運算來實現演算法的核心 S-Box 部分，並將架構下的矩陣乘法運算合而為一以縮短運算時間。最後，以強調重複利用的矽智產方式(IP-based)實現，並遵守 AMBA AHB Slave 傳輸協定，以助於未來在系統面的開發。在本論文的成果方面，此 Configurable-AES 加解密器以 UMC 0.18μm CMOS 製程實現，擁有約 81K 的 gate counts，在最高處理速度下，對於 128/192/256 三種不同金鑰長度下，分別可達到 3.2Gbps、2.67 Gbps 和 2.29 Gbps。

IP-based design and chip implementation of the AES coprocessor with configurable parameter

Student : Tsung-Yao Pai Advisor : Prof. Bing-Fei Wu

Department of Electrical and Control Engineering

National Chiao Tung University

ABSTRACT

In this paper, we implement a configurable AES (C-AES) coprocessor, which supports all specified key lengths, such as 128, 192, and 256 bits, and both the ECB and CBC operation modes. The round keys for encryption and decryption are generated on the fly without any internal memory. Specifically, it provides the flexibility to change the parameters of each transformations, such as the irreducible polynomial, the affine matrix, the affine constant, and the row vector of the matrix used in *MixColumns*(). These parameters are online changeable, i.e., they are also the inputs of the circuit. For increasing the speed, an optimized combination is presented in the proposed architecture. By using basis conversion and composite field in *SubBytes*(), and pre-calculating the values of every power of *xtime*() of constants in *MixColumns*(), the matrix multiplications in *SubBytes*() and *MixColumns*() can be integrated into a new transformation to reduce the computation path. Furthermore, all arithmetic components are also reused for the encryption and the decryption data paths. The proposed design has been implemented using a UMC 0.18μm CMOS technology. The throughput is about 3.2Gbps for 128-bit keys, 2.67Gbps for 192-bit keys, and 2.29Gbps for 256-bit keys, respectively. The total gate count is about 81K. This work provides a customized AES cipher to let users change parameters; therefore, it can be utilized in the applications requiring customized security, .e.g., the virtual private networks (VPN).

致謝

首先，要感謝我的指導教授 吳炳飛博士二年來的教導，感謝您提供豐沛的研究資源與良好的學習環境，在設備與開發工具可以說是應有盡有，讓我懂得如何開發設計，以符合市場與業界的需求。另外，也要感謝我的大學專題指導教授張孟洲博士，是您帶領我進入數位 IC 設計的領域，啟蒙我做研究該有的 sense，您認真謙虛的態度，是我學習的榜樣。

此外，最要感謝的人，就是曾經帶領過我的顏志旭學長，林重甫學長，彭信元學長，在你們的教導之下，都讓我學到很多不同領域的東西，使的我在各方面，都有顯著的成長。

當然，也要感謝實驗室的夥伴們，學長晏阡、培恭、俊傑，學姊映伶，同學元馨、岑偉、小熊、子萱、ppj、皓昱，學弟秉宗、敏偉、晉源，有你們的陪伴，讓研究和生活都充滿了樂趣。

最後，感謝我親愛的父母及家人，有你們在背後辛苦的付出和支持，今天才能順利完成研究所學業，這一切都絕不是光靠我一個人能做得到的，謝謝你們。

Publication & Award

發表論文	■Chih-Hsu Yen, Tsung-Yao Pai, and Bing-Fei Wu, “The Implementations of the Reconfigurable Rijndael Algorithm with Throughput of 4.9Gbps,” *Proceedings of the 16th VLSI Design/CAD Symposium*, Aug. 2005.
得獎紀錄	■2005, 第七屆矽智產SIP設計競賽 「佳作」 可參數化之高安全度 Rijndael加密演算法

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

Due to the growth of applications in Internet and wireless communication, more and more users require the security measures and devices for protecting the data, which users transmit over the channels. Since nobody can guarantee that the information will not be stolen over open communication channels, it is a general way to encrypt the information before they are transmitted into the channels. There are many cryptosystem developed in the past. According to the key type, the cryptosystem can be classified into two type systems, such as the symmetric-key and asymmetric-key cryptosystem. The concept of cryptosystem is illustrated in Fig. 1.1. The plaintext, which will be sent in the transmitter, will be encrypted with the cipher key to generate the ciphertext, and the ciphertext, like a random number, is transmitted in the insecure channel. Finally, the ciphertext will be received in the receiver and be decrypted with the cipher key to recover the plain text.

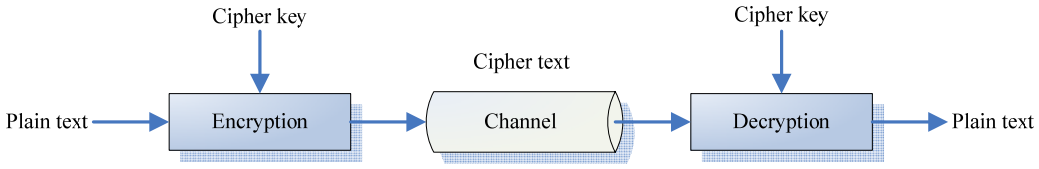


Fig. 1.1 The concept of cryptosystem.

The symmetric-key cryptosystem, such as DES, AES [1], and 3DES [2], uses an identical key to encrypt the message text and to decrypt the cipher text. The asymmetric-key cryptosystem, such as RSA and Elliptic Curve algorithms [3], uses a different key for encryption and decryption. Different from the asymmetric-key cryptography, the structure of the symmetric-key cryptography is simple. Usually, it consists of a block cipher, and by executing it iteratively, the encrypted data is

generated. The block cipher can be divided into two parts, the nonlinear and linear operations. These operations use the ways of substitution and permutation to cause the diffusion and confusion on data, and make the data difficult to be attacked. Because the architecture of the symmetric-key cryptography is simple, the cryptography can encrypt or decrypt data at high speed and is more suitable for the condition that has a large amount of data to be processed.

In early years, DES algorithm, approved in 1977, was a widespread method for this cryptosystem. However, the computer or other calculating machine has become more and more powerful in recent years, and DES algorithm is not strong enough. In order to replace the DES algorithm, the Advanced Encryption Standard (AES) is developed by National Institute of Standards and Technology (NIST). And finally NIST was announced that it has selected **Rijndael** to propose for the AES on November 26, 2001 and became effective on May 26, 2002.

1.2 Motivation

With the rapid advance in the communication technology, the use of networks and communication facilities for transmitting information between people, companies or countries has been implanted deeply in our real life. Network processing becomes an emerging problem that needs to be dealt with in the computer system. The ability to properly serve heavy traffic on Internet through network equipments is now provided by a fast network processing chip. The security of communications, originally a problem of government, military or privileged organizations, becomes one of the major concerns among individuals and corporations. There is an increasing demand in network processing, including the security processing.

Therefore, the goal of our design is providing a security processor that not only supports customized security requirement but also has high throughput to cooperate with fast network processing chip. In Barkan and Bihamn's [4] research, they pointed out that random selecting a dual cipher is desired during a connection. If all data in a connection are encrypted by several dual ciphers is possible, a more secure connection can be established by Rijndael. In other words, the coefficients of irreducible polynomial $m(x)$, *MixColumns* row vector $c(x)$, and affine transformation can be replaced by other values such that various encryption algorithms can be

obtained easily. However, AES algorithm with configurable coefficients will cause more complexity of implementation, and unsuitable low throughput for high-speed Ethernet. Thus, we propose the circuit design of the configurable AES algorithm to provide throughput over gigabit per seconds, so it can be implemented in high-speed network services for virtual private network (VPN) application.

However, not all the combinations can generate secure block ciphers against existing attacks. Several design criteria must be satisfied to ensure the selected tuple can generate proper *SubBytes*(), and all the inverse function of the four transformations can be found. The cryptanalysis of the configurable AES (C-AES) algorithm is beyond the scope of our works. Here, only the circuit design of a suitable architecture is considered.

1.3 Organization

This thesis is organized as follow. AES algorithm is described in Chapter 2. The hardware strategy to reduce the area and critical path in our cipher engine is discussed in Chapter 3, and the implementation of the 3-in-1 key generator to cooperate with the cipher engine is proposed in Chapter 4. Moreover, the top-level architecture of our C-AES coprocessor is shown in Chapter 5. In Chapter 6, the design methodology and verification based on intellectual property (IP) reuse are introduced, and the experimental results and comparison are also given. In Chapter 7, the conclusion of this thesis and the future work are listed.

Chapter 2
AES Algorithm

2.1 Algorithm Specification

AES algorithm, defined by NIST of the United States, has been widely accepted for replacing DES as the new symmetric encryption algorithm [5]. Originally NIST invited proposals for new algorithms for the AES in 1997. Among the 15 preliminary candidates, MARS, RC6, Rijndael [6], Serpent and Twofish were announced as the finalist candidates in 1999 for further evaluation. Finally in 2000, Rijndael was selected as AES algorithm. Actually, AES algorithm adopted Rijndael with the data block of length 128 bits and the cipher key of length 128, 192, or 256 bits only. It is an efficient algorithm for both hardware and software implementation. A basic pseudo code of AES encryption is depicted in Fig. 2.1.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])

   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

Fig. 2.1 Pseudo code of AES encryption.

Given a cipher input block of length 128 bit, composed by 16 bytes, are mapped onto the elements of a 4×4 array, called the *State* [5], in the order $a_{00}, a_{10}, a_{20}, a_{30}, a_{01}, a_{11}, \ldots, a_{03}, a_{13}, a_{23}, a_{33}$. As demonstrated in Fig. 2.2, the

algorithm implements four transformations that operate on elements, rows and columns of the array respectively. After an initial round key addition, a round function consisting of four transformations, ***SubBytes*(), *ShiftRows*(), *MixColumns*()** *and* ***AddRoundKey*(),** is applied to the *State* array. The round function is performed 10 times iteratively for 128-bit key, 12 times for 192-bit key and 14 times for 256-bit key. In the last round, *MixColumns*() is not applied. Four basic transformations of the AES algorithm are described briefly as follows [6]:

1. ***SubBytes*()** transformation, also called S-Box, is a non-linear byte substitution that operates independently on each byte of the State. Given an element of the *State* array, $a_{ij}, 0 \le i, j \le 3$, it is treated as the element in $GF(2^8)$ with the irreducible polynomial *m*(*x*). The *SubBytes*() transformation performs an inverse mapping of a_{ij} first followed by an affine transformation. The *SubBytes*() can be expressed as the following equation:

$$b_{ij} = (Affine(x) \cdot a_{ij}^{-1}) \oplus const(x)$$

where *Affine*(*x*), *const*(*x*) are two polynomial in *GF*(2) with the degree less than 8. In AES algorithm,

$$p(x) = x^8 + x^4 + x^3 + x + 1$$

or {11B} in hexadecimal representation, and

$$Affine(x) = x^4 + x^3 + x^2 + x + 1 \ = \{1\text{F}\}$$

$$const(x) = x^6 + x^5 + x + 1 \ = \{63\}.$$

respectively. For the inverse of *SubBytes*() transformation, it can be obtained by the inverse of the affine transformation followed by taking the multiplicative inverse in $GF(2^8)$, i.e.,

$$Inv\_Affine(x) = x^6 + x^3 + x \ = \{4\text{A}\}$$

$$Inv\_const(x) = x^2 + 1 \ = \{05\}.$$

2. ***ShiftRows*()** transformaion is simply a cyclic shifting operation on the rows of the *State* with different numbers of bytes (offsets). In the State array, Row 0 ($a_{00}, a_{01}, a_{02}, a_{03}$) is not shifted, Row 1 ($a_{10}, a_{11}, a_{12}, a_{13}$) is left shifted over 1 byte, Row 2 ($a_{20}, a_{21}, a_{22}, a_{23}$) is left shifted over 2 bytes and Row 3 ($a_{30}, a_{31}, a_{32}, a_{33}$) is left shifted over 3 bytes. The inverse of *ShiftRows*() is simply the cyclic right

shifting the Row 1, Row 2 and Row 3 over 1, 2 and 3 bytes respectively.

3. ***MixColumns*()** transformation is the operation that considers the column of *State* as polynomials over $GF(2^8)$, and performs the multiplication modulo (x^4+1) with a fixed polynomial $c(x)$. Let $a_j(x) = a_{0j} + a_{1j}x + a_{2j}x^2 + a_{3j}x^3$ be a polynomial with coefficients being the elements of the *j-th* columns of the *State* array. Let $c(x) = c_0 + c_1x + c_2x^2 + c_3x^3$ be a polynomial with coefficient $c_i \in GF(2^8), 0 \le i \le 3$. The matrix multiplication of *MixColumns*() transformation can be expressed as the implementation of each column by $c(x)$, i.e.,

$$b_j(x) = a_j(x) \cdot c(x) \mod (x^4+1), \quad 0 \le j \le 3.$$

in AES algorithm, $c(x)$ is defined as $\{02\} + \{01\}x + \{01\}x^2 + \{03\}x^3$. It can also be written as the following matrix multiplication.

$$\begin{bmatrix} b_{0j} \\ b_{1j} \\ b_{2j} \\ b_{3j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_{0j} \\ a_{1j} \\ a_{2j} \\ a_{3j} \end{bmatrix}$$

The inverse of *MixColumns*() transformation is similarly by multiplying each column with a specific multiplication polynomial $d(x)$, which is defined by

$$c(x) \cdot d(x) = 01.$$

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

4. ***AddRoundKey*()** transformation is simply an XOR operation that adds a round key to the State in each iteration, where the round keys are generated from the key expansion procedure.

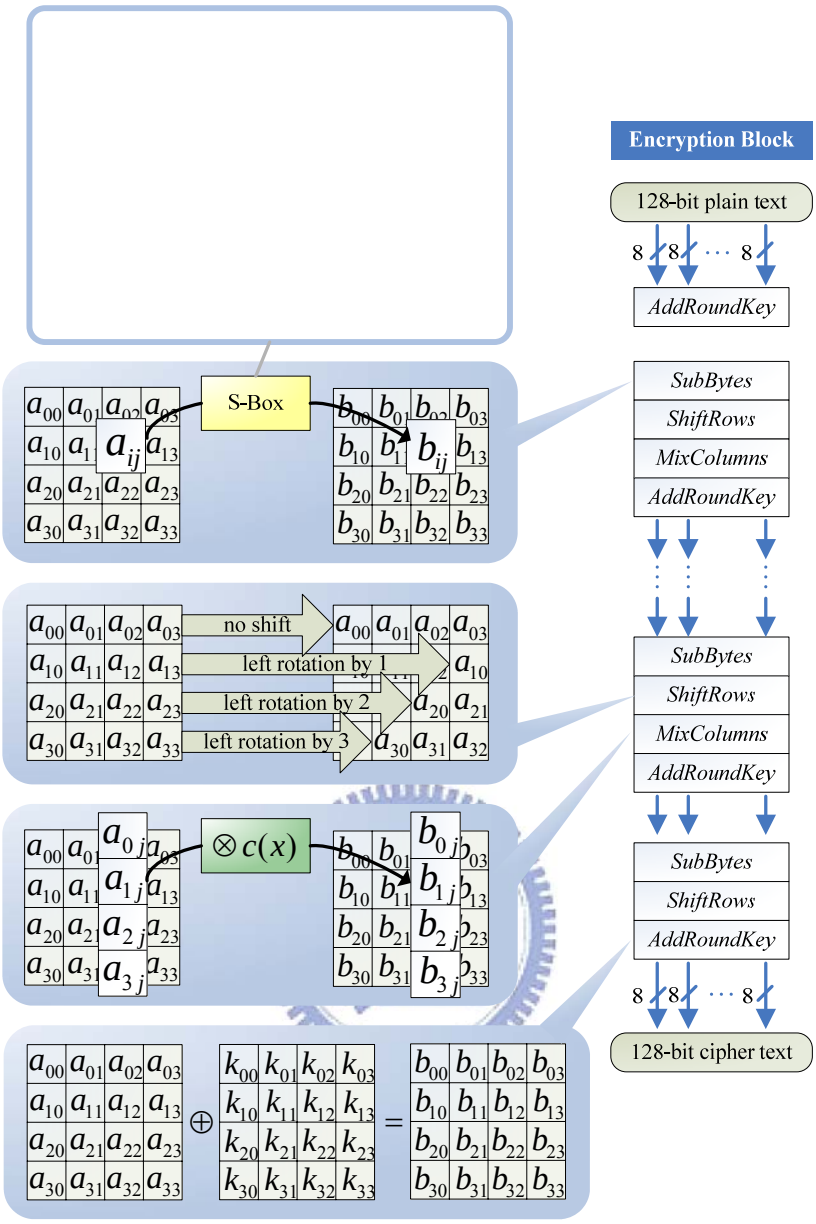


Fig. 2.2 The encryption procedure of AES algorithm.

The decryption procedure of the AES is basically the inverse of each of the transformation (*InvSubBytes*(),*InvShiftRows*(),*InvMixColumns*(), and *AddRoundKey*()) in reverse order.

The key expansion procedure in AES algorithm is used to calculate the round key for every *AddRoundKey*() transformation. Basic procedure of the key expansion is shown in Fig. 2.3. According to the selected key size, N_k is 4 for 128-bit key, 6 for 192-bit key and 8 for 256-bit key. Each W_i is a 32-bit word. The first N_k words (W_i) are identical to the initial key, while the rest of the round keys are expanded

iteratively by *SubBytes*() transformation and cyclic byte rotation. The *SubWord*() is a function that return a 4-byte word where each byte is the result of *SubBytes*() transformation to the byte at the corresponding position in the input word. *RotWord*() performs a cyclic left rotation of a given word by 8 bits. *Rcon*(*x*) is a constant composed by 4 bytes, $\{Rc_i,\{00\}.\{00\},\{00\}\}$, where $Rc_i = x^i$ is the field element in $GF(2^8)$ with polynomial *m*(*x*).

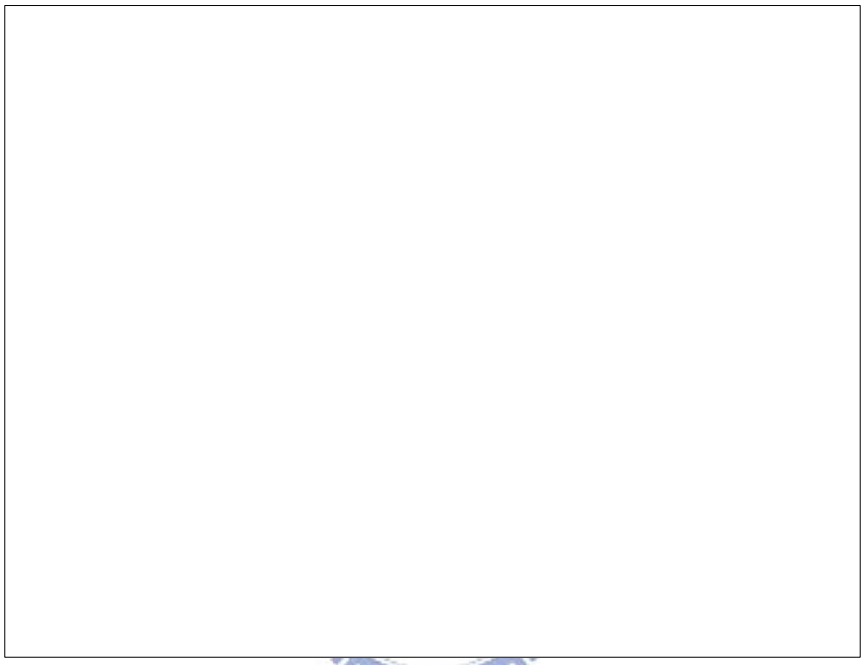


Fig. 2.3 Pseudo code of key expansion.

2.2 Block Cipher Modes of Operation

In cryptography, a block cipher operates on blocks of fixed length, often 64 or 128 bits. To encrypt longer messages, several modes of operation, such as Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB) may be used. In the following, ECB and CBC modes, which can be supported by our C-AES coprocessor, are introduced.

1. **Electronic Codebook mode (ECB)**

   When this cipher mode is used, each block is encrypted individually. No feedback is used. This means any blocks of plaintext that are identical and are either in the same message, or in a different message that is encrypted with the same key, will be transformed into identical ciphertext blocks. If the plaintext to

be encrypted contains substantial repetition, then it is feasible for the ciphertext to be broken one block at a time. Furthermore, it is possible for an unscrupulous person to substitute and exchange individual blocks without detection. The encryption procedure in ECB mode is described in Fig. 2.4.

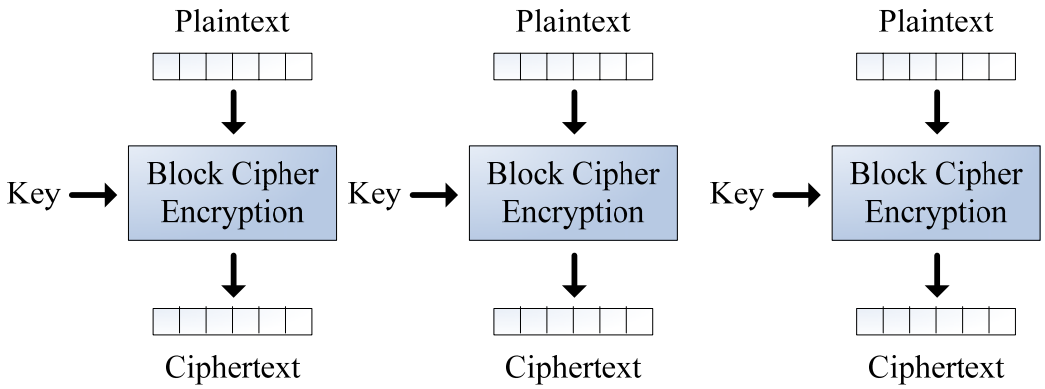


Fig. 2.4 The encryption procedure in ECB mode.

2. **Cipher Block Chaining mode (CBC)**

This cipher mode introduces feedback. Before each plaintext block is encrypted, it is combined with the ciphertext of the previous block by a bitwise XOR. This ensures that even if the plaintext contains many identical blocks, they will each encrypt to a different ciphertext block. As Fig. 2.5 shown, the initialization vector (IV) is combined with the first plaintext block by a bitwise XOR before the block is encrypted.

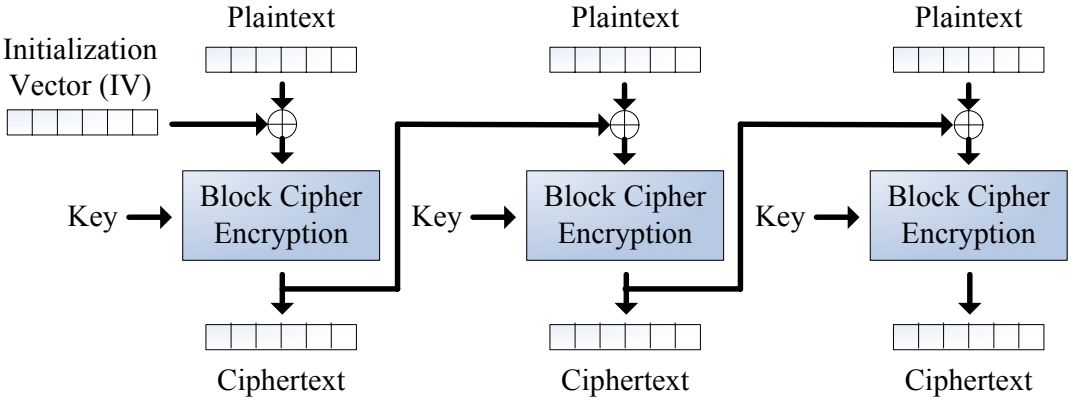


Fig. 2.5 the encryption procedure in CBC mode.

Chapter 3
Hardware-Reduction Strategy for C-AES

In general, the parameters of each transformation in the original AES algorithm are constants, so the optimization methods for hardware implementation of the configurable AES algorithm will be based on different consideration from previous works. The design of *SubBytes*() and *MixColumns*() transformations which provide the configurability and excellent trade-off between silicon area and performance will become the key point to be evaluated especially.

3.1 Previous Work

In AES proposal[5] , the authors describe the cipher Rijndael and treat the implementation aspects of the cipher and its inverse. They concentrate on the implementation in software on 8-bit processors, typical for current Smart Cards and on 32-bit processors, typical for PCs. The several performance comparisons of these implementations in software are estimated.

However, hardware implementations of AES algorithm compare to software implementations. They provide more physical security as well as higher speed. Since there is a need to perform data encryption on high-speed network services, the operation speed is very important. Many architecture optimization approaches are employed to speed up the hardware implementations. According to the approaches used to implement the *SubBytes*() transformations (also known as the S-Box) , we can divide these into two kinds : look-up table (LUT) based designs and non-LUT based designs.

The traditional LUT methodology is well suited to implement the complex and slow operations. Especially, it is cost-effective for the field programmable gate arrays

(FPGAs) [7][8][9][10][11][12]. In particular, several approaches merge the *SubBytes*() and *MixColumns*() transformation into a single LUT for an additional speedup [13][14][15][16]. The high speeds can be achieved by a 10-stage fully pipelined LUT based Rijndael encryption design [17]. However, the encryption and decryption processes need implementing as separate LUTs, and these approaches lead to high area requirements.

Non-LUT approaches employ the combinational logic only to implement the multiplicative inverse and the affine transformation of S-Box. Since the inversions in *Galois Field* $GF(2^8)$ have high hardware complexities, the field elements of $GF(2^8)$ are mapped to the elements in some isomorphic composite fields, in which the field operations can be implemented by lower cost subfield operations. Compared to the LUT-based approach, the composite field arithmetic has cost-benefit for the semi-custom application specific integrated circuit (ASIC) implementations. The approaches based on this idea can be found in [18][19][20][21]. In particular, Authors of [22][23] have evaluated the sub-pipelined architecture based on optimum speed-area ratio in non-feedback modes.

3.2 S-Box Optimization

Since our goal is to propose a configurable AES coprocessor. If the LUT-based approach is used to implement the S-Box, any change of the *Affine matrix, const*(*x*) and *m*(*x*) will require a replacement for the S-Box values. For example, if we use ROM-based LUT, it needs another 256×8-bit ROM to store one set of the S-Box values. It is unacceptable area requirement to support parameter configurability; else if we use RAM-based LUT to transfer S-Box values, either to re-compute these values on chip or off chip will consume a long configuration time. Therefore, we select the composite field arithmetic approach to implement S-Box. Since it only requires 2×16 8-bit matrix multiplier to provide the configurability of *Affine matrix, const*(*x*) and *m*(*x*). The area requirement can be reduced to an acceptable area, and the critical path also can be modified by combining the data path of sub-functions. In the following sections, two techniques, composite field arithmetic and combination of *SubBytes*() and *MixColumns*(), for hardware-reduction strategy will be introduced.

3.2.1 Composite Field Arithmetic

We call two pairs $\{ GF(2^n) , Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i , q_i \in GF(2) \}$ and $\{ GF((2^n)^m) , P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i , p_i \in GF(2^n) \}$ a composite field [26], if

- $GF(2^n)$ is constructed by $Q(y)$, which is an irreducible polynomial of degree *n* over $GF(2)$;
- $GF((2^n)^m)$ is constructed by $P(x)$, which is an irreducible polynomial of degree *m* over $GF(2^n)$.

Moreover, the composite field $GF((2^n)^m)$ is isomorphic to the field $GF(2^k)$ for *k=nm.* According the investigation of a lot of fields [23], the following irreducible polynomials are selected to extend the composite field of $GF(2^8)$ in our design.

$$\begin{cases} GF(2^4): & q_0(x) = x^4 + x + 1 \\ GF((2^4)^2): & q_1(x) = x^2 + x + \omega \quad (\omega = \{1001\}) \end{cases} \tag{3.1}$$

Additionally, the composite fields can be built iteratively from the lower order fields. As shown in [19] , the composite field of $GF(2^8)$ also can be extended under the polynomial basis using these irreducible polynomials:

$$\begin{cases} GF(2^2): & p_0(x) = x^2 + x + 1 \\ GF((2^2)^2): & p_1(x) = x^2 + x + \phi \quad (\phi = \{10\}_2) \\ GF(((2^2)^2)^2): & p_2(x) = x^2 + x + \lambda \quad (\lambda = \{1100\}_2) \end{cases} \tag{3.2}$$

Fig. 3.1 shows the outline of the S-Box implementation by using the composite field arithmetic. The multiplicative inversion over a field A is the most costly operation. The following 3 steps are adopted to implement this operation.

1. Map all elements of the field A to a composite field B, using an isomorphism function δ.
2. Compute the multiplicative inverses over the field B.
3. Re-map the computation results to A, using the function δ^{-1}.

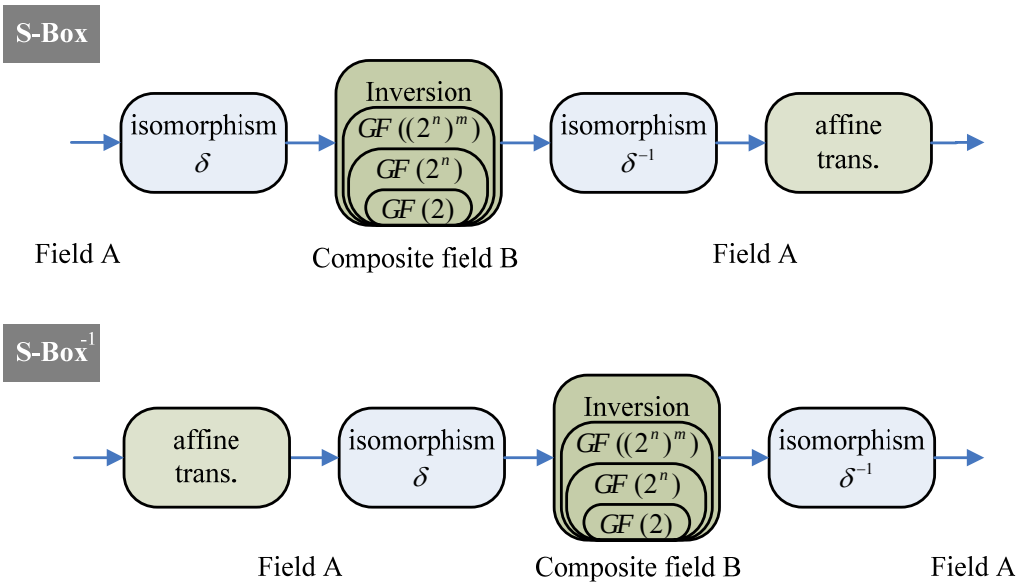


Fig. 3.1 The outline of the S-Box implementation

3.2.2 Isomorphism Functions and Basis Transformation

The isomorphism function is the transformation matrix to map elements of $GF(2^k)$ to $GF((2^n)^m)$. The method for generating the transformation matrix can be found in [19][27][28] for the condition where the field polynomials are primitive polynomials. Although, the polynomial $x^8+x^4+x^3+x+1$ {11B} used in the AES algorithm is an irreducible polynomial but is not primitive. The exhaustive-search -based algorithm in [28] can be used to find the transformation in this case, and the primitive irreducible polynomial $p(x)=x^8+x^4+x^3+x^2+1$ {11D} is the better choice to be the basis in the composite field [19][23][29].

The δ and δ^{-1} matrices which map $GF(2^8)$ into $GF((2^4)^2)$ and $GF((2^4)^2)$ into $GF(2^8)$ based on the field polynomial in (3.1) are as below.

$$\delta=\begin{bmatrix}1&1&0&0&0&0&1&0\\0&1&0&0&1&0&1&0\\0&1&1&1&1&0&0&1\\0&1&1&0&0&0&1&1\\0&1&1&1&0&1&0&1\\0&0&1&1&0&1&0&1\\0&1&1&1&1&0&1&1\\0&0&0&0&0&1&0&1\end{bmatrix}\qquad \delta^{-1}=\begin{bmatrix}1&0&1&0&1&1&1&0\\0&0&0&0&1&1&0&0\\0&1&1&1&1&0&0&1\\0&1&1&1&1&1&0&0\\0&1&1&0&1&1&1&0\\0&1&0&0&0&1&1&0\\0&0&1&0&0&0&1&0\\0&1&0&0&0&1&1&1\end{bmatrix} \tag{3.3}$$

The δ and δ^{-1} matrices which map $GF(2^8)$ into $GF(((2^2)^2)^2)$ and $GF(((2^2)^2)^2)$ into $GF(2^8)$ based on the field polynomial in (3.2) are as below. The least significant bits are in the upper left corner.

$$\delta = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad \delta^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.4}$$

However, in order to support the configurability of the irreducible polynomial $m(x)$ in the S-Box and to use the previous isomorphism functions directly, it is necessary to perform the change of basis on the common $GF(2^8)$ field. Based on the algorithms in [30][31], they proposed the efficiently operation to calculate the change-of-basis matrix from Basis B_1 to B_0 on the common field degree. Therefore we can convert our field element which modulo another $m(x)$ into the basis used in the isomorphism functions.

For example, if we suppose that B_0 is the polynomial basis modulo $m_0(x) = x^8 + x^4 + x^3 + x + 1$ {11B}, and B_1 is the polynomial basis modulo $m_1(x) = x^8 + x^7 + x^6 + x^5 + x^2 + 1$ {1F5}, which is another irreducible polynomial $m(x)$. According the arithmetic operations in [30][31], the change-of-basis from B_1 to B_0 is Γ and the inverse matrix from B_0 to B_1 is Γ^{-1}.

$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad \Gamma^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Therefore, the modification of the S-Box implementation is shown in Fig. 3.2. As the irreducible polynomial $m(x)$ is changed, The δ and δ^{-1} are replaced with $(\delta \cdot \Gamma)$ and $(\Gamma^{-1} \cdot \delta^{-1})$. We define the $\delta' = \delta \cdot \Gamma$ and $\delta'^{-1} = \Gamma^{-1} \cdot \delta^{-1}$ as the new isomorphism functions for configurable S-Box in the following section. Because the affine transformation and the isomorphism are all linear operation, it is possible to merge them together to reduce the path delay. Thus, the values of $(\delta' \cdot A)$ and $(A \cdot \delta'^{-1})$ can be computed before the encryption or the decryption operations. In fact, we process the parameter initialization when the input interface receives the parameter data concurrently. The parameter initialization will be described in the Chapter 5. Moreover, we can reuse the inversion over the composite field for different $m(x)$, *Affine matrix* and *constant*(x) easily with the help of parameter initialization.

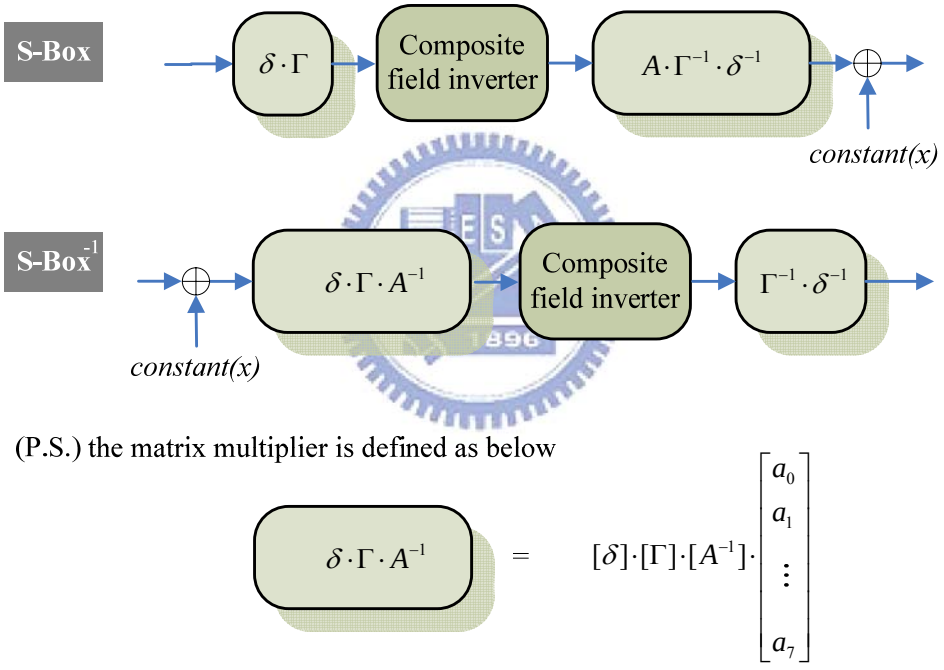


Fig. 3.2 The outline of the configurable S-Box implementation.

3.2.2 Multiplicative Inversion over the Composite Field

For the composite field $GF((2^m)^n)$, computing the multiplicative inverses can be done as a combination of operations over the subfields $GF(2^m)$, using the Extended Euclidean Algorithm described in [32]. Taking our proposed implementation as an example, in the composite field $GF((2^4)^2)$ using the

irreducible polynomials (3.1), an element can be expressed as $S(x) = s_h x + s_l$, where $s_h, s_l \in GF(2^4)$, and x is the root of $q_1(x)$. The multiplicative inverse of $s_h x + s_l$ modulo $q_1(x)$ is equivalent to $B(x)$ which satisfying the follow equation [33]:

$$A(x)q_1(x) + B(x)S(x) = 1 \tag{3.6}$$

Such $A(x)$ and $B(x)$ can be found by using the extended Euclidean algorithm. Firstly, we need to rewrite $q_1(x)$ in the form of

$$q_1(x) = Q(x)S(x) + R(x) \tag{3.7}$$

$Q(x)$ and $R(x)$ are the quotient and remainder polynomial of dividing $q_1(x)$ by $S(x)$. By long division it can be derived as follow:

$$Q(x) = s_h^{-1}x + (1 + s_h^{-1}s_l)s_h^{-1} \tag{3.8}$$

$$R(x) = \omega + (1 + s_h^{-1}s_l)s_h^{-1}s_l \tag{3.9}$$

Substituting (3.8) and (3.9) into (3.6) and multiplying s_h^2 to both sides of the equation, it follows that

$$s_h^2 q_1(x) = (s_h x + (s_h + s_l))S(x) + (s_h^2\omega + s_h s_l + s_l^2) \tag{3.10}$$

Multiplying $\Theta = (s_h^2\omega + s_h s_l + s_l^2)^{-1}$ to both sides of (3.10), we get

$$\Theta s_h^2 q_1(x) = \Theta(s_h x + (s_h + s_l))S(x) + 1 \tag{3.11}$$

Since addition and subtraction are the same in the extended field of $GF(2)$, comparing (3.6) and (3.11), it can be observed that

$$\Theta s_h^2 q_1(x) + \Theta(s_h x + (s_h + s_l))S(x) = 1$$

$$S^{-1}(x) = B(x) = s_h\Theta x + (s_h + s_l)\Theta \tag{3.12}$$

According to (3.12), the multiplicative inversion in $GF(2^8)$ can be implemented in $GF((2^4)^2)$ by the architecture illustrated in Fig. 3.3.

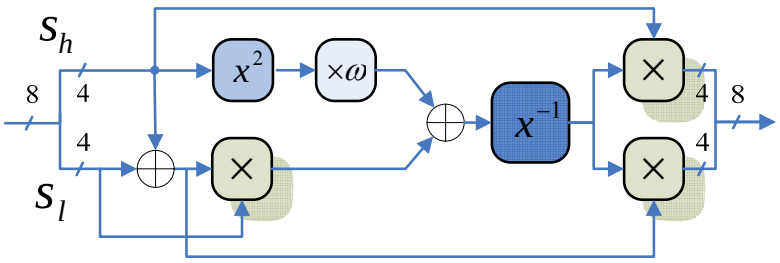


Fig. 3.3 The multiplicative inversion based on composite field $GF((2^4)^2)$.

For introducing the sub-operations in Fig. 3.3, let the elements in $GF(2^4)$ is represented as polynomial of degree 4, i.e., $A(x)=\sum_{i=0}^{3} a_i x^i, B(x)=\sum_{i=0}^{3} b_i x^i$ where $a_i, b_i \in GF(2)$. Therefore, the hardware optimization of these sub-operations can be obtained by using the following equations.

$$A^2(x)=a_3x^3+(a_3\oplus a_1)x^2+a_2x+(a_2\oplus a_0) \tag{3.13}$$

$$A(x)\times\omega=a_0x^3+a_3x^2+a_2x+(a_1\oplus a_0) \tag{3.14}$$

In particular, the combination of the squarer (x^2) and the constant multiplier ($\times\omega$) can be cost-effective as (3.15).

$$A^2(x)\times\omega=(a_0\oplus a_0)x^3+a_3x^2+(a_3\oplus a_1)x+a_0 \tag{3.15}$$

And the multiplication of these two field elements can be expressed as (3.16). By extracting the common factors in the bit-level expressions, we can apply the combination and integration of sub-factors for further area reduction.

$$\begin{aligned} A(x)\times B(x)=&\{a_3b_0\oplus a_2b_1\oplus a_1b_2\oplus(a_0\oplus a_3)b_3\}x^3+\\ &\{a_2b_0\oplus a_1b_1\oplus(a_0\oplus a_3)b_2\oplus(a_2\oplus a_3)b_3\}x^2+\\ &\{a_1b_0\oplus(a_0\oplus a_3)b_1\oplus(a_2\oplus a_3)b_2\oplus(a_1\oplus a_2)b_3\}x+\\ &\{a_0b_0\oplus a_3b_1\oplus a_2b_2\oplus a_1b_3\} \end{aligned} \tag{3.16}$$

The most complicated operation in Fig. 3.3 is the inversion in $GF(2^4)$. As the definition of the field element in $GF(2^4)$, the inversion of $A(x)$ is equivalent to $A^{14}(x)$. Thus, our approach simplifies the equation $A^{-1}(x)=A^{14}(x)$ directly based on the logic optimization techniques as illustrated in (3.17)

$$
\begin{aligned}
A^{-1}(x) &= A^{14}(x) \\
&= \{a_3(a_1 \oplus a_2 \oplus a_3) \oplus (a_1 \oplus a_2 \oplus a_3) \oplus a_1a_2a_3\}x^3 + \\
&\quad \{a_0(a_1 \oplus a_2) \oplus a_0a_3(a_2 \oplus a_3) \oplus (a_2 \oplus a_3)\}x^2 + \\
&\quad \{a_0(a_1 \oplus a_2) \oplus a_1(a_2 \oplus a_3) \oplus a_3 \oplus a_0a_1a_3)\}x + \\
&\quad \{a_2(a_0 \oplus a_1) \oplus a_0a_1(a_2 \oplus a_3) \oplus (a_0 \oplus a_1) \oplus (a_2 \oplus a_3)\}
\end{aligned} \tag{3.17}
$$

3.2.3 The Comparison of Multiplicative Inversion

Observing other approaches, the multiplicative inversion can be implemented by different irreducible polynomials to analyze the area cost and path delay. In [19], the authors use the (3.2) as their irreducible polynomials, and the implementation of the inversion in $GF((2^2)^2)$ is described in Fig. 3.4.

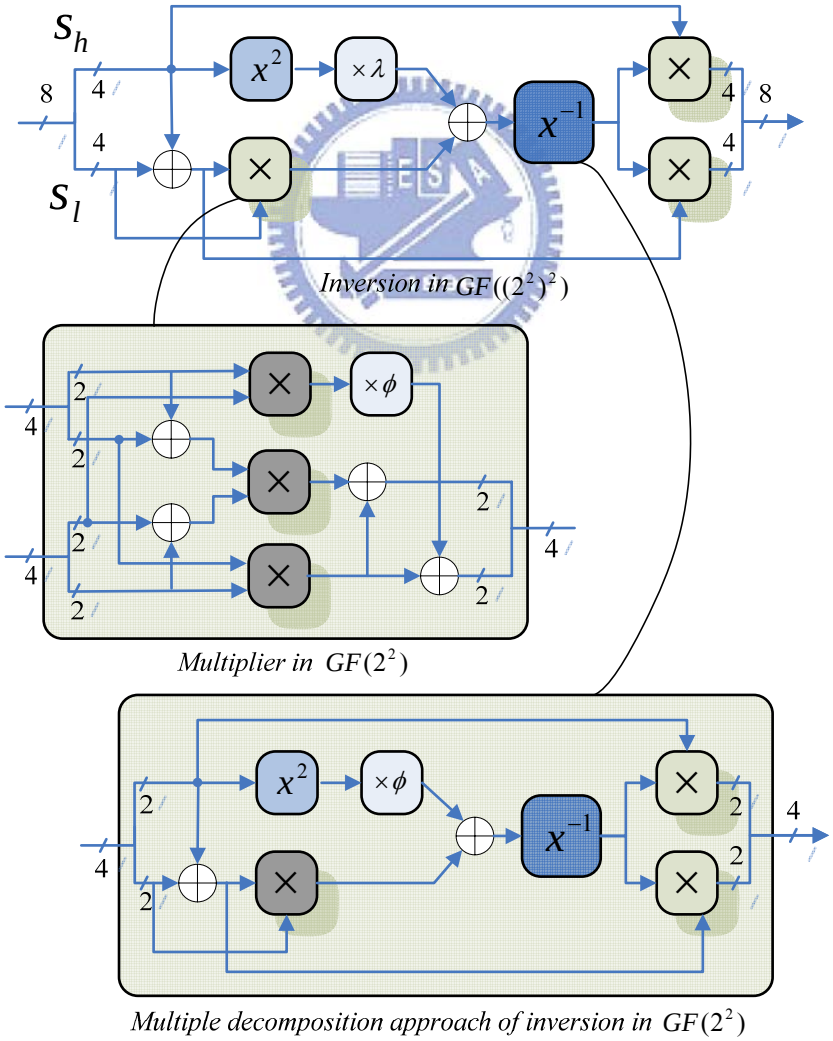


Fig. 3.4 The implementation of the inversion in $GF(((2^2)^2)^2)$

However in [22], the inversion in $GF((2^2)^2)$ is directly implemented by (3.18) using sub-expression sharing, not the multiple decomposition as described in Fig. 3.4. Moreover, it has the smallest gate count and the shortest critical path.

$$\begin{aligned} B^{-1}(x) = &\{a_3 \oplus a_3a_2a_1 \oplus a_3a_0 \oplus a_2\}x^3 + \\ &\{a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_0 \oplus a_2 \oplus a_2a_1\}x^2 + \\ &\{a_3 \oplus a_3a_2a_1 \oplus a_3a_1a_0 \oplus a_2 \oplus a_2a_0 \oplus a_1\}x + \\ &\{a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_1 \oplus a_3a_1a_0 \oplus a_3a_0 \oplus \\ &a_2 \oplus a_2a_1 \oplus a_2a_1a_0 \oplus a_1 \oplus a_0\} \end{aligned} \tag{3.18}$$

The comparison results of the individual composing modules are listed in Table 3.1. Observing the results in [19][22], composite field decomposition can reduce the hardware complexity significantly when the order of the field involved is large. However, for a small field, such as $GF(2^4)$, further decomposition may not be the optimum approach. For this reason, we select the approach that implement the derived equation by the common sub-expression sharing techniques in $GF((2^4)^2)$.

Table 3.1 Performance analysis of the inversion in section 3.2.3.

Modules	[19] 's		[22] 's		[34] 's		Ours	
	Area	Delay	Area	Delay	Area	Delay	Area	Delay
$x^2 \times \lambda$ $x^2 \times \omega$	7XOR	4XOR	7XOR	4XOR	4XOR	2XOR	2XOR	1XOR
Multiplier in $GF(2^4)$ $GF((2^2)^2)$	21XOR+ 9AND	4XOR+ 1AND	21XOR+ 9AND	4XOR+ 1AND	17XOR+ 16AND	4XOR+ 1AND	15XOR+ 16AND	4XOR+ 1AND
Inverison in $GF(2^4)$ $GF((2^2)^2)$	17XOR+ 9AND	7XOR+ 2AND	14XOR+ 9AND	3XOR+ 2AND	13XOR+ 8AND	3XOR+ 2AND	14XOR+ 11AND	4XOR+ 1AND
Inverison in $GF((2^4)^2)$ $GF(((2^2)^2)^2)$	95XOR+ 36AND	17XOR+ 4AND	92XOR+ 36AND	13XOR+ 4AND	76XOR+ 56AND	13XOR+ 4AND	69XOR+ 59AND	14XOR+ 3AND

In Table 3.1, the comparison between ours and the similar approach in [34] is also illustrated. In [34], a different irreducible polynomial (3.19) is be used to extend the composite field $GF((2^4)^2)$. The multiplicative inversion also can be found by using the extended Euclidean algorithm, and the authors illustrate a new algorithm of common sub-expression elimination (CSE) to optimize the hardware cost of all the bit-level equations. Although another irreducible polynomial is applied, the difference in the hardware cost is limited.

$$\begin{cases} GF(2^4): \quad q_0(x) = x^4 + x + 1 \\ GF((2^4)^2): q_1(x) = x^2 + \gamma x + \omega \qquad (r = 0001), (\omega = \{1001\}) \end{cases} \tag{3.19}$$

3.3 *MixColumns*() Optimization

In general, the multiplication of two elements of $GF(2^8)$ is required in *MixColumns*(), and it is achieved by repeating *xtime*(). Since the implementation of *xtime*() function is based on the value of irreducible polynomial $m(x)$, the changeable $m(x)$ and *MixColumns* matrix will increase the complexity of multiplication significantly in *MixColumns*().

Therefore, in our proposed approach, after the irreducible polynomial $m(x)$ and the *MixColumns* matrix C are given, the value of $xtime^n(c_i)$ will be calculated in advance and be stored in 4×8 8-bits registers, where c_i is the entry of *MixColumns* matrix, and $i \in \{0,1,2,3\}$. In the following, we take the operation of one column (3.20) in *MixColumns*() as an example to describe our approach.

$$t(x) = C \cdot s(x) = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & c_0 & c_1 & c_2 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & c_2 & c_3 & c_0 \end{bmatrix} \cdot \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \tag{3.20}$$

Suppose $s_0 = 0 \times CA$ in (3.20), the calculation of $c_0 \cdot s_0$ is achieved by

$$
\begin{aligned}
c_0 \cdot (0 \times CA) = {} & 1 \cdot xtime^7(c_0) + 1 \cdot xtime^6(c_0) + \\
& 0 \cdot xtime^5(c_0) + 0 \cdot xtime^4(c_0) + \\
& 1 \cdot xtime^3(c_0) + 0 \cdot xtime^2(c_0) + \\
& 1 \cdot xtime\ (c_0) + 0 \cdot (c_0)
\end{aligned}
\tag{3.21}
$$

In other words, let the elements s_0 is represented as polynomial of degree 8, i.e., $s_0 = s_{07}x^7 + s_{06}x^6 + s_{05}x^5 + s_{04}x^4 + s_{03}x^3 + s_{02}x^2 + s_{01}x + s_{00}$, and (3.21) can rewrite as follow. Thus, the multiplication in $GF(2^8)$ can be implemented by one 8-bit matrix multiplier.

$$
c_0 \cdot s_0 = C_{c0} \cdot s_0 = \begin{bmatrix} xtime^0(c_0) & xtime^1(c_0) & xtime^2(c_0) & \cdots & xtime^7(c_0) \end{bmatrix} \begin{bmatrix} s_{00} \\ s_{01} \\ s_{02} \\ s_{03} \\ s_{04} \\ s_{05} \\ s_{06} \\ s_{07} \end{bmatrix}
\tag{3.22}
$$

In summary, the configurability of *MixColumns*()/*InvMixColumns*() is provided by pre-computed and stored the C_{c0}, C_{c1}, C_{c2} and C_{c3} in 4×8 8-bits registers. In addition, *MixColumns*() and *InvMixColumns*() transformations can also easily share the same hardware by changing the coefficient according to the processing mode.

3.4 The Hardware Architecture

In this work, a half-duplex parameterized cipher engine is proposed. The encryption and decryption data paths are efficient combined based on the modified order in Fig. 3.5.

3.4.1 The Direct Architecture

Its direct architecture is depicted in Fig. 3.6. The solid line is the encryption path, and the dash line is the decryption path. The data procedure is a 128-bit architecture, i.e., 16 bytes are processed simultaneously. Based on the approach of composite field arithmetic, the finite field inverter and the matrix multipliers for field conversion are implemented, and the matrix multipliers are exploited to realize the *MixColumns( )* / *InvMixColumns*() transformation.

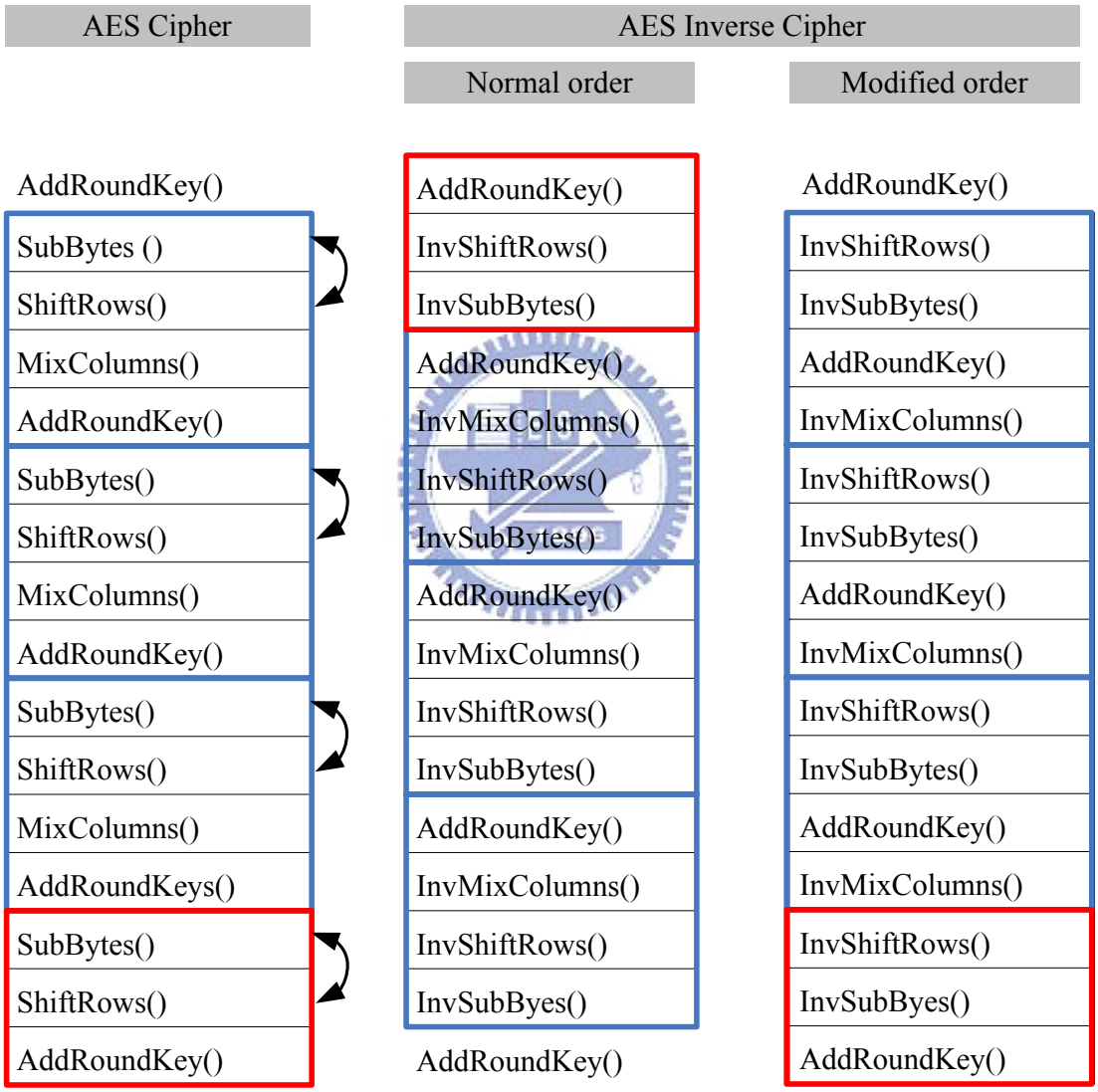


Fig. 3.5 The operation order of encryption and decryption

Although these approaches have the benefit of on-line configurability, they will induce longer critical path than the traditional approaches[19][34]. Since the modular multiplications in original AES implementations are with constants, they can reduce

the area and shorten the path delay efficiently. Therefore, the approach that combines the matrix multipliers in the S-Box and *MixColumns*() transformation is proposed to reduce the computation path in our configurable Rijndeal design.

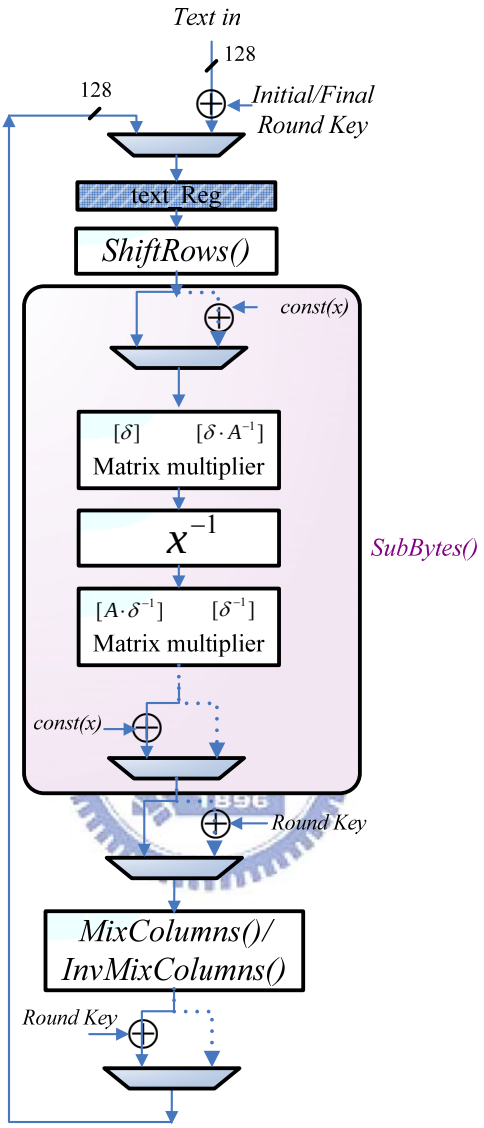


Fig. 3.6 The direct architecture of parameterized cipher engine in this work.

3.4.2 The Combination of *SubBytes*() and *MixColumns*()

In this section, the combination of matrix multipliers in *SubBytes*() and *MixColumns*() is introduced. Several representations are used to explain our approach easily. In the following, *Key*(*i*) represents the key of *i* th round, and *Key*(*0*) is the initial key. According to the composite field arithmetic, *SubBytes*() transformation is rewritten as $A \cdot \delta' \cdot Inv(\delta'^{-1} \cdot x) \oplus c_A$, where A is the affine matrix, δ', δ'^{-1} is the

new isomorphism functions described in section 3.2.2, c_A is *const*(*x*), and *Inv*() is multiplicative inversion in $GF(2^4)$, and $R_e(i)$ represents the intermediate values produced after round function *i* times. Thus, the series transformations of encryption can be rewrote as follow:

$$\begin{cases} R_e(0) = x \oplus Key(0) & (3.23) \\ R_e(i) = MixColumns(A \cdot \delta'^{-1} \cdot Inv(\delta' \cdot ShiftRows(R_e(i-1))) + c_A) + Key(i) \\ \qquad\qquad 1 \le i \le Nr-1 & (3.24) \\ R_e(Nr) = A \cdot \delta'^{-1} \cdot Inv(\delta' \cdot ShiftRows(R_e(Nr-1))) + c_A + Key(Nr) & (3.25) \end{cases}$$

where *Nr* represents number of rounds, which is defined in Sec. 2.1.2.

Since our goal is to separate the affine transformation and the isomorphism function from S-Box and merge them with *MixColumns*(). In other words, (3.24) is modified as (3.26), and the input of next round, $R_e(i)$, will be redefined as $R'_e(i) = \delta' \cdot R_e(i)$, shown in (3.27).

$$R_e(i) = MixColumns(A \cdot \delta'^{-1} \cdot (Inv(ShiftRows(\delta' \cdot R_e(i-1))) \oplus \delta' \cdot A^{-1} \cdot c_A)) \oplus Key(i) \quad (3.26)$$

$$\begin{aligned} R'_e(i) &= \delta' \cdot R_e(i) \\ &= \delta' \cdot MixColumns(A \cdot \delta'^{-1} \cdot (Inv(ShiftRows(R'_e(i-1))) \oplus \delta' \cdot A^{-1} \cdot c_A)) \oplus \delta' \cdot Key(i) \\ &= MixColumns'(Inv(ShiftRows(R'_e(i-1))) \oplus c'_A) \oplus \delta' \cdot Key(i) \end{aligned} \quad (3.27)$$

$$c'_A = \delta' \cdot A^{-1} \cdot c_A \quad (3.28)$$

$$MixColumns'(x) = \delta' \cdot MixColumns(A \cdot \delta'^{-1} \cdot x) \quad (3.29)$$

Moreover, these parameters in (3.27) can be calculated beforehand to reduce the computation path delay. In particular, the new *MixColumns*() can be depicted as (3.29) by the change of C'_{c0}, C'_{c1}, C'_{c2}, and C'_{c3}, because the matrix multiplication of *MixColumns*() transformation (3.22) can be rewrite as (3.30).

$$\begin{aligned} c'_0 \cdot s_0 &= C'_{c0} \cdot s_0 \\ &= (\delta' \cdot C_{c0} \cdot A \cdot \delta'^{-1}) \cdot s_0 \end{aligned} \quad (3.30)$$

Although the initial round key addition (3.31) and the final round function (3.32) are also differing slightly from the traditional functions, the critical path is still dominated by the data path that computes one AES round function. Thus, comparing (3.27) and (3.24), the computation path of two 8-bit matrix multiplication is removed form the critical path after the optimized combination. The approach to optimize for speed requirement is achieved.

$$R'_e(0) = \delta' \cdot (x + Key(0)) \tag{3.31}$$

$$R'_e(Nr) = A \cdot \delta'^{-1} \cdot (Inv(ShiftRows(\delta' \cdot R'_e(Nr-1))) \oplus c'_A) \oplus Key(Nr) \tag{3.32}$$

Using the same approach, the operation order of decryption in Fig. 3.5 can also be represented as $R_d(i)$, showed in (3.33)(3.34)(3.35), and the proof of the modified intermediate value, $R'_d(i)$, is given in (3.37)(3.39)(3.40).

$$\begin{cases} R_d(0) = x + Key(Nr) & (3.33) \\ R_d(i) = InvMixColumns(\delta'^{-1} \cdot Inv(\delta' \cdot A^{-1} \cdot (InvShiftRows(R_d(i-1)) \oplus c_A)) \\ \qquad \oplus Key(Nr-i)) \qquad 1 \le i \le Nr-1 & (3.34) \\ R_d(Nr) = \delta'^{-1} \cdot Inv(\delta' \cdot A^{-1} \cdot (InvShiftRows(R_d(Nr-1)) \oplus c_A)) \oplus Key(0) & (3.35) \end{cases}$$

$$R_d(i) = InvMixColumns(\delta'^{-1} \cdot Inv(InvShiftRows(\delta' \cdot A^{-1} \cdot R_d(i-1)) \oplus \delta' \cdot A^{-1} \cdot c_A) \oplus Key(Nr-i)) \tag{3.36}$$

$$\begin{aligned} R'_d(i) &= \delta' \cdot A^{-1} \cdot R_d(i)) \\ &= \delta' \cdot A^{-1} \cdot InvMixColumns(\delta'^{-1} \cdot Inv(InvShiftRows(\delta' \cdot A^{-1} \cdot R_d(i-1)) \oplus \delta' \cdot A^{-1} \cdot c_A) \\ &\quad \oplus Key(Nr-i)) \\ &= \delta' \cdot A^{-1} \cdot InvMixColumns(\delta'^{-1} \cdot \{Inv(InvShiftRows(\delta' \cdot A^{-1} \cdot R_d(i-1)) \oplus \delta' \cdot A^{-1} \cdot c_A) \\ &\quad \oplus \delta' \cdot Key(Nr-i)\}) \\ &= InvMixColumns'(Inv(InvShiftRows(R'_d(i-1))) \oplus c'_A)) \oplus \delta' \cdot Key(Nr-i) \end{aligned} \tag{3.37}$$

$$InvMixColumns'(x) = \delta' \cdot A^{-1} \cdot InvMixColumns(\delta'^{-1} \cdot x). \tag{3.38}$$

$$R'_d(0) = \delta \cdot A^{-1}(x + Key(Nr)) \tag{3.39}$$

$$R'_d(Nr) = \delta'^{-1} \cdot \{Inv((InvShiftRows(R'_d(Nr-1)) \oplus c'_A)) \oplus \delta' \cdot Key(0)\} \tag{3.40}$$

Taking the hardware resource shared between the encryption and the decryption into consideration, the circuit in Fig. 3.7 is an implementation according to the equation of $R'_e(i)$ and $R'_d(i)$. Note that the matrix multipliers which located at the both ends of the multiplicative inversion are separated from the computation path of one AES round function.

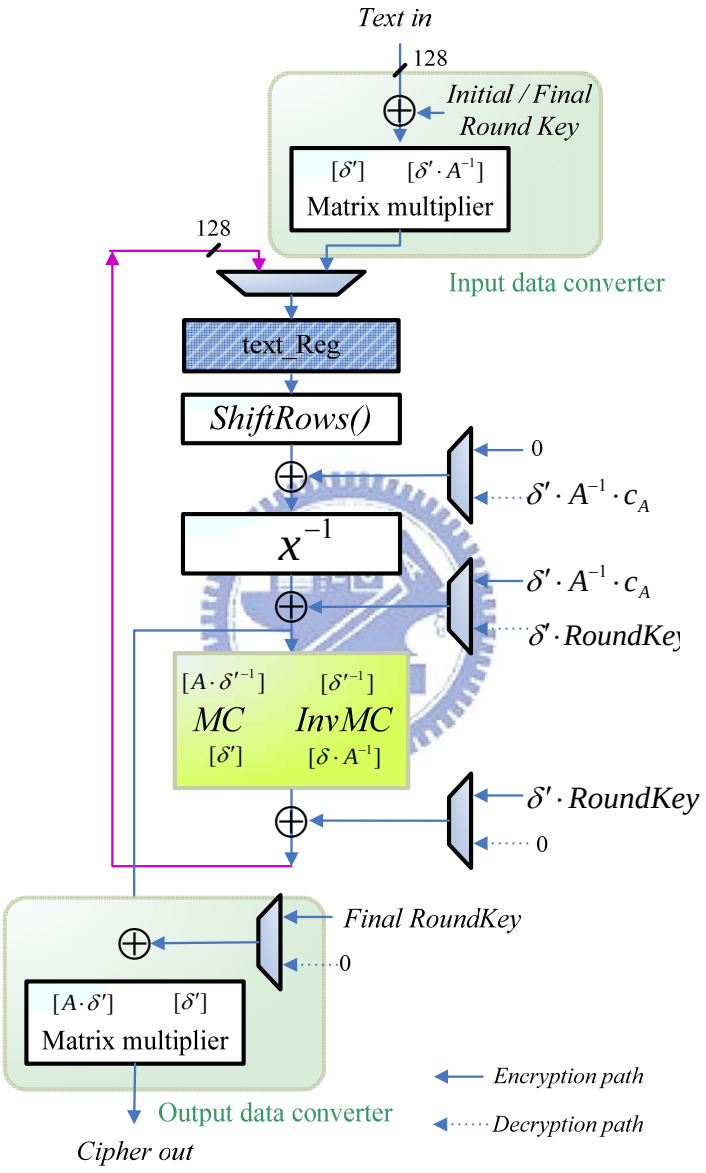


Fig. 3.7 The combined architecture of the parameterized cipher engine.

Our design is synthesized using the Synopsys Design Version. The critical path is detailed in Table 3.2. The multiplicative inversion in $GF(2^4)$ occupies about 38% of the delay time. The second major component is neither *MixColumns*() nor *AddRoundKey*, but the selectors. The requirement to use selectors is not obvious from

the original Rijndael algorithm specification, where they appear as conditional branches and data selections. Because of the wide data width, the optimization of the data selection is considered carefully.

Table 3.2 The critical path of the cipher engine

Component	**Critical path delay (ns)**
Register output and setup	0.17
Selector	0.28
ShiftRows()	0.09
XOR	0.11
Inversion in $GF(2^4)$	1.68
Selector and XOR	0.31
MixColumns()/*InvMixColumns*()	0.55
Selector	0.14
Total	**3.33**

(0.18μm CMOS standard cell)

Chapter 4
3-in-1 Key Expansion Design

The key generator that generates the forward and reverse round keys for the encryption and the decryption is another issue needs to be considered. The on-the-fly key expansion is an approach that generates each round key in the operation time of each round function. Therefore, different from the pre-computation approach, it is unnecessary to use additional memory to store the sub-keys, and can support a better trade-off between cost and performance than others. In this approach, the key generator for 128-bit key size only is illustrated in [21][25] , and another one for three different key size is proposed in [35] .

In this chapter, the 3-in-1 key generator to cooperate with the cipher engine is proposed. Our design will produce one 128-bit round key per clock cycle for three different types of key length: 128-bit, 192-bit, and 256-bit. The basic architecture is made reference to [35], and an efficient architecture is proposed and the shorter critical path and lower area overhead is obtained by optimizing the order of data selection.

4.1 The Data Flow Graph of Key Expansion

According to AES algorithm specification and the representations in Fig. 4.1 [35], the data flow graphs for three different types of key length are derived in Fig. 4.2, Fig. 4.3, and Fig. 4.6. The details are described in the following sections.

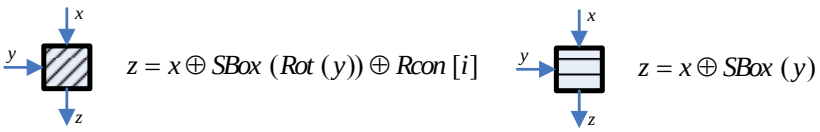


Fig. 4.1 The representations of operation in key expansion.

4.1.1 128-bit Key Expansion

The initial cipher key is denoted by the array of 4-byte words, [w0, w1, w2, w3], and a single round function of key expansion is illustrated in Fig. 4.2. Since the number of rounds (*Nr)* is 10 when the key length is 128-bit, the final round key will be produced as [w40, w41, w42, w43], and this will be the initial input of key expansion in decryption procedure. Because the length of cipher key is equal to the length of the *State* array, it is quite straight forward to generate the round key for each clock cycle.

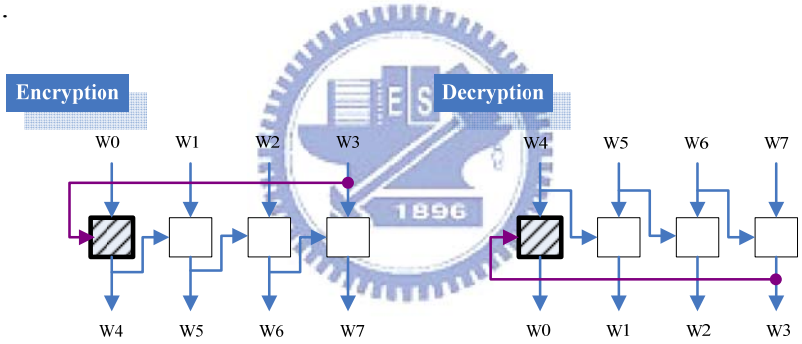


Fig. 4.2 The 128-bit key expansion for the encryption/decryption.

4.1.2 192-bit Key Expansion

The data flow is similar to the one described above, but the initial cipher key becomes the array of 6-byte words [w0, w1, w2, w3, w4, w5]. Moreover, the 192-bit key is concurrently computed for each cycle shown in Fig. 4.3. However, the length of round key required by the cipher engine is still 128-bit, not 192-bit. This different bit length will cause the incompatible timing diagram. In order to solve this problem, the key expansion routine is rearranged such that only one 128-bit round key are produced for each time frame. The results for the encryption and the decryption are demonstrated in Fig. 4.4 and Fig. 4.5. For the rearranged data flow graph, the new

round functions are represented as $f_{R0}(w)$, $f_{R1}(w)$, $f_{R2}(w)$, and $f_{R3}(w)$.

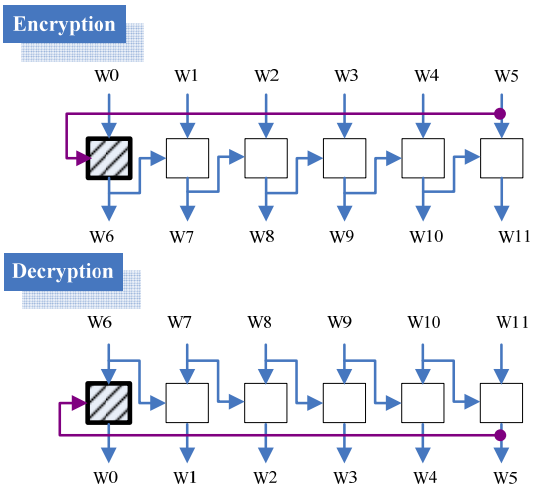


Fig. 4.3 The 192-bit key expansion for the encryption/decryption.

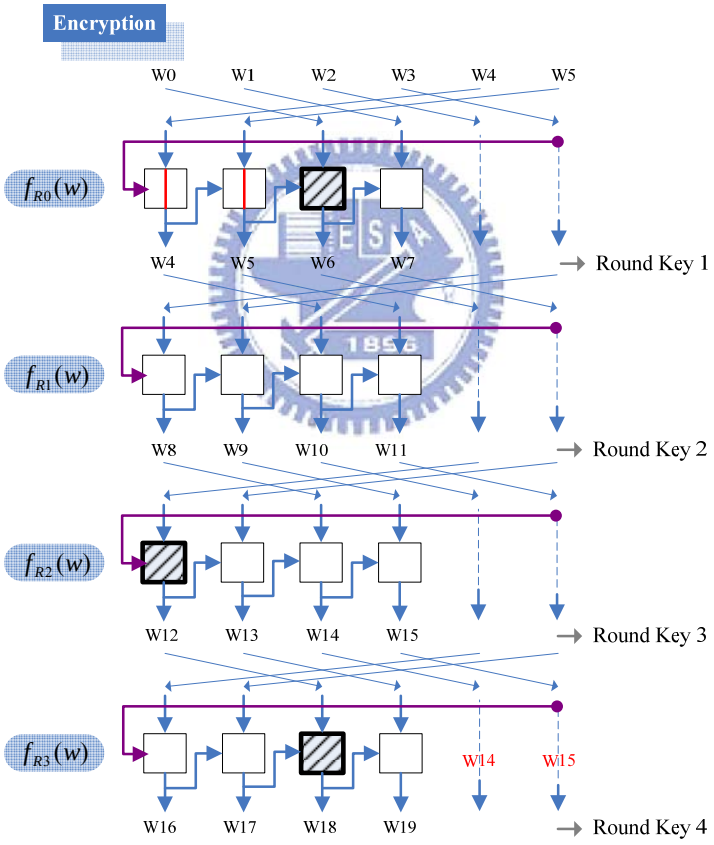


Fig. 4.4 The rearrangement of the 192-bit key expansion for the encryption.

At the start of the key expansion for the encryption, which is shown in Fig. 4.4, the initial cipher key applies the round function $f_{R0}(w)$ to produce next round key, and go on. For the 192-bit cipher key, the number of rounds is 12. Thus, the final round key will be represented as [w48, w49, w50, w51], and output from the round

function $f_{R2}(w)$. For this reason, note that the first round function for the decryption will be $f_{R2}(w)$, not $f_{R0}(w)$, and the following data flow can be easily found by reversing the computing order.

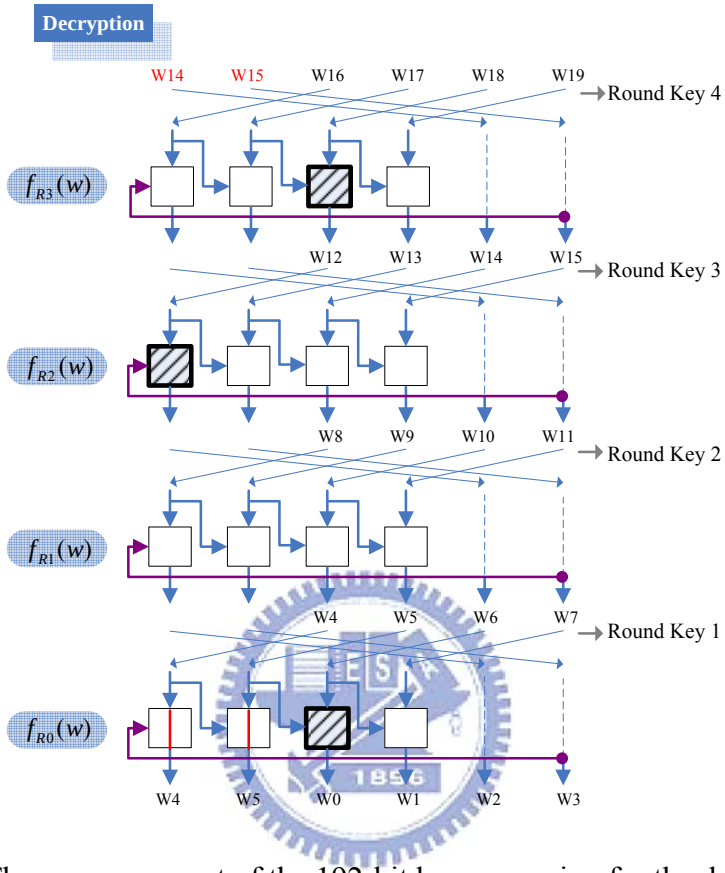


Fig. 4.5 The rearrangement of the 192-bit key expansion for the decryption.

4.1.2 256-bit Key Expansion

As described above, Fig. 4.6 shows the original data flow graph.

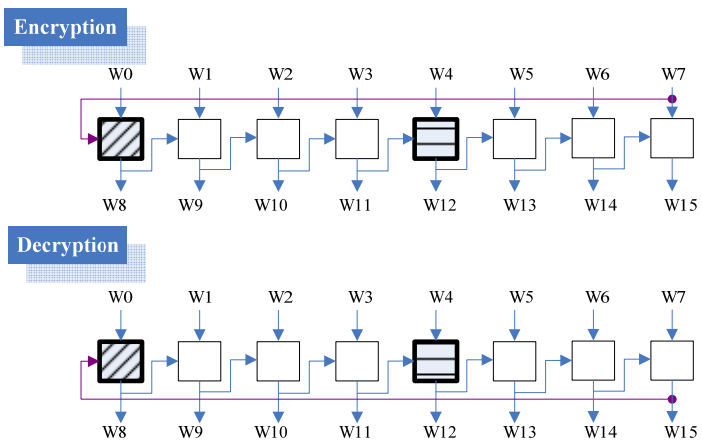


Fig. 4.6 The 256-bit key expansion for the encryption/decryption.

Observing the results of rearrangement shown in Fig. 4.7 and Fig. 4.8, it is more similar with 128-bit key expansion. The data flow of the encryption and the decryption are almost the same, since the first round function for decryption is still $f_{R0}(w)$.

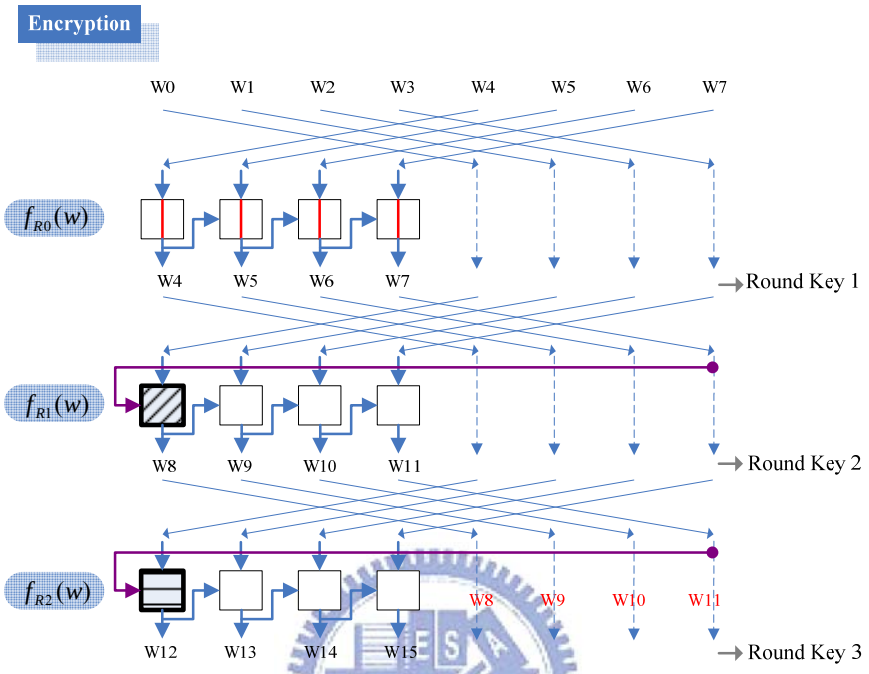


Fig. 4.7 The rearrangement of 256-bit key expansion for the encryption.

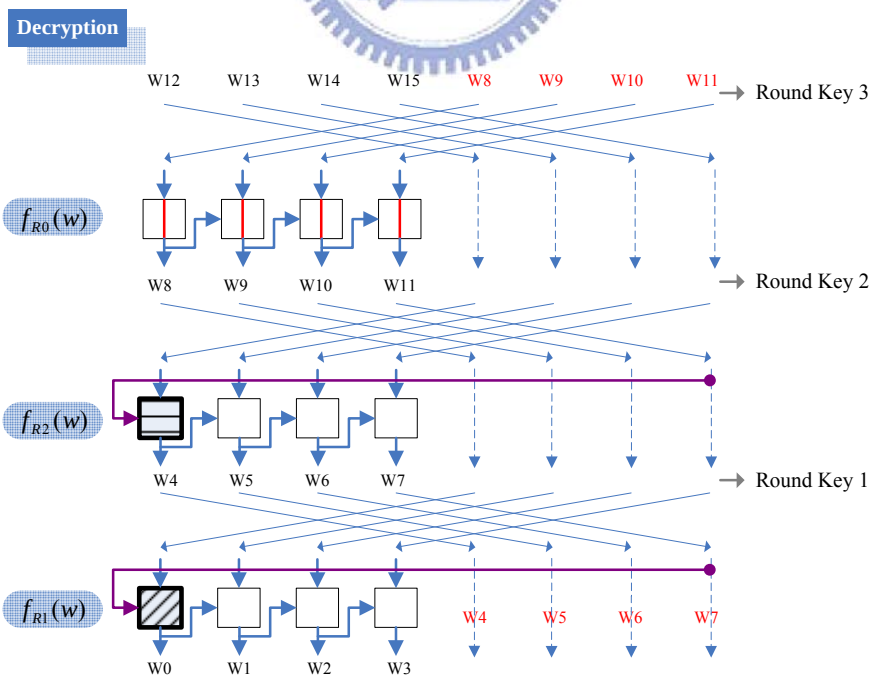


Fig. 4.8 The rearrangement of 256-bit key expansion for the decryption.

In summary, by properly shuffling the input key for each round function, only 4 computing elements are used to realize the key expansion for different key length.

4.2 The Hardware Architecture of 3-in-1 Key Generator

Fig. 4.9 shows the hardware architecture of the 3-in-1 key generator based on the rearranged data flow graph. LR0, LR1, …, LR7 are 32-bit registers for storing the intermediate round key. Each component is illustrated as follow.

(1) Controller for 3-in-1 key generator:

Fig. 4.10 shows the state diagram of the controller for 3-in-1 key generator. Since the timing diagram of 128-bit key expansion is pure and easy to control, the state and transition, which indicate that the 128-bit key length is selected in the finite state machine (FSM), are ignored. Based on this FSM, the controller can generate the proper control signals for the data flow control.

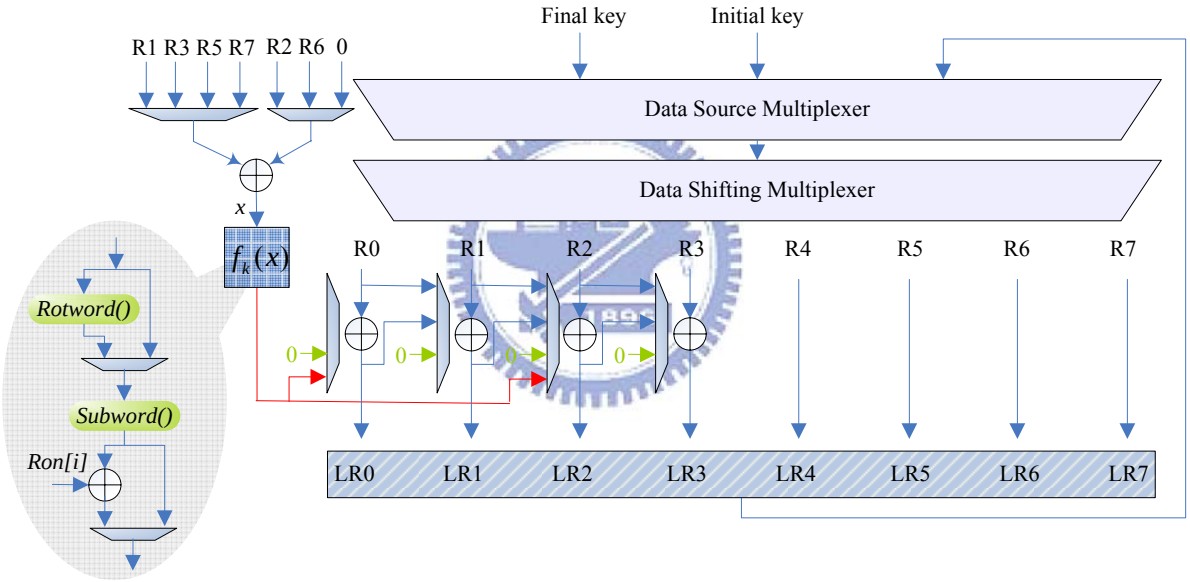


Fig. 4.9 The architecture of 3-in-1 key generator module.

Specifically, the initial input for key generator to execute the decryption procedure is the final round key. Thus, while the reset of coprocessor or the change of initial cipher key is launched, the key generator will execute encryption procedure once to obtain the final round key and stored it in the register beforehand. The data flow control of this function will be managed by the main controller shown in Fig. 5.1.

(2) Data Source Multiplexer :

Once the key generator is reloaded, the initial key for the encryption or final key for decryption will be selected to take a fresh start.

(3) Data Shifting Multiplexer :

It is used to shift the input key of each round function. If the input as the array is denoted as [w0, w1, w2, w3, w4, w5, w6, w7], and the function can be demonstrated in Table 4.1.

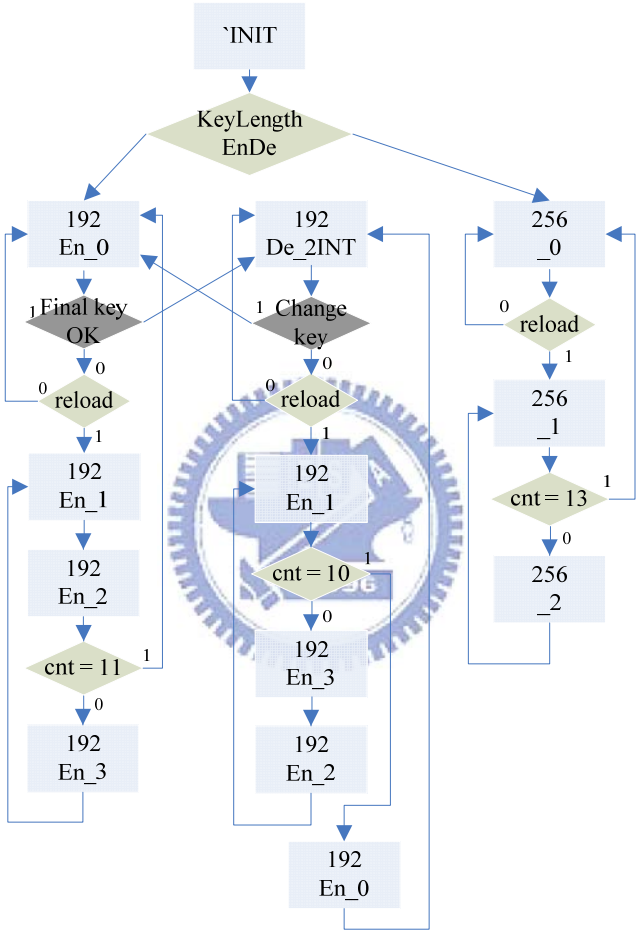


Fig. 4.10 State diagram of the controller for 3-in-1 key generator.

Table 4.1 The function of data shifting multiplexer in the key expansion

Key length	En/De	*Data order*
128-bit	**En**	W0 W1 W2 W3
	De	W0 W1 W2 W3
192-bit	**En**	W4 W5 W0 W1 W2 W3
	De	W2 W3 W4 W5 W2 W3
256-bit	**En**	W4 W5 W6 W7 W0 W1 W2 W3
	De	W4 W5 W6 W7 W0 W1 W2 W3

(4) $f_k(x)$:

In our proposed design, the S-Box function $f_k(x)$ is divided from the data path of round function, and additional XOR gates and multiplexers are used. Taking 128-bit key expansion as an example, if the encryption/decryption data path is implemented as Fig. 4.11, one combinational loop is introduced. In order to eliminate the combination loop, 32-bit XOR gate and two multiplexers are used to select the input of $f_k(x)$ in different data path locations. It is shows in Table 4.2

Table 4.2 The input table for the S-Box in key expansion.

Key length	Round function	x
128-bit	***En***	R3
	De	R2 ⊕ R3
192-bit	***En*** $f_{R0}(x)$	R1
	En $f_{R1}(x)$	
	En $f_{R2}(x)$	R5
	En $f_{R3}(x)$	R5 ⊕ R6*
	De $f_{R0}(x)$	R5
	De $f_{R1}(x)$	

Key length	Round function	x
192-bit	***De*** $f_{R2}(x)$	R5
	De $f_{R3}(x)$	R1
256-bit	***En*** $f_{R0}(x)$	
	En $f_{R1}(x)$	R7
	En $f_{R2}(x)$	R7
	En $f_{R0}(x)$	
	De $f_{R1}(x)$	R7
	De $f_{R2}(x)$	R7

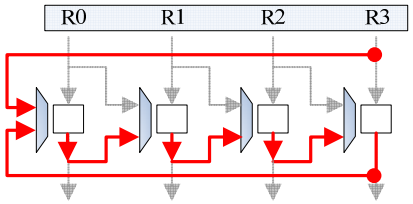


Fig. 4.11 The combination loop in 128-bit key expansion data path.

In particular, the round function $f_{R3}(x)$ in the 192-bit key expansion for the encryption is the special case. Observing the data path graph in Fig. 4.4, the input of S-Box function $f_k(x)$ is " *w17* ", which can be produced by (4.1). Since the implementation of (4.1) will lead to the combinational loop, the (4.2) is utilized, and the value of " *w5* " is computed and stored in " R6 " register while the round function $f_{R0}(x)$ is processed.

$$w17 = w14 \oplus w15 \tag{4.1}$$

$$= w10 \oplus w11 \oplus w15 = w5 \oplus w15 \tag{4.2}$$

In summary, the critical path in the key generator is illustrated in Table 4.3. The *SubWord*() transformation occupies about 58% of the delay time, and the second major component is the sequence of XOR operations. From our test results, the generation of sub-keys on the fly creates the longest critical path in our C-AES coprocessor. Thus, it is the bottleneck for increasing throughput in our design, and the comparison of 3-in-1 key generator is listed in Table 4.4.

Table 4.3 The critical path of the key generator.

Component	**Critical path delay (ns)**
Register output and setup	0.27
Selector and *RotWord*()	0.54
Matrix multiplier	0.48
Inversion in $GF(2^4)$	1.36
Matrix multiplier	0.40
Selector and XOR	0.79
Total	**3.84**

Table 4.4 Comparison of 3-in-1 key generator

	Verbauwhede [24]	**Su** [35]	**Ours**
Technology	0.18μm	0.25μm	0.18μm
Gate counts	60.1K	26.7K	21.7K
Critical Path	10ns	N/A	3.84ns

Chapter 5

The Implementation of C-AES coprocessor

In this chapter, the top-level architecture of our C-AES coprocessor is introduced. It provides the capacity for changing the parameters of each transformation, and the original AES algorithm is also included as well. In addition, it also supports all specified key lengths, such as 128, 192, and 256 bits, and both ECB and CBC operation modes. Moreover, the round keys for the encryption and the decryption are generated on the fly without ant internal memory.

5.1 Top-level View

The top-level view of C-AES coprocessor is shown in Fig. 5.1. It consists of an I/O interface module, three controllers, a key generator, and a cipher engine. The I/O interface serves as a data collector through a 32-bit data bus. These controllers generate control signals for data transportation, parameter initialization, key expansion, and encryption/decryption based on the processing mode. To perform an encryption/decryption process initially, the I/O interface first gathers the slice of all necessary data, such as parameters, initial cipher key, IV, and plain text/cipher text. During the data access operations are manipulated, the parameter initialization is processed simultaneously. If the processing mode is decryption, the final round key will be computed and stored beforehand by the key generator. Once the parameters and initial/final key are ready, the main controller will take over the control and execute the encryption or the decryption procedure whenever one 128-bit plaintext is ready at the I/O interface. Then, AES round function will be applied for 10, 12, or 14 times depending on the key length. Finally, the processed data will be retrieved through the 32-bit data bus. The encryption/decryption procedure will be executed

iteratively until no plain text/cipher text is fed or a new processing mode command is received.

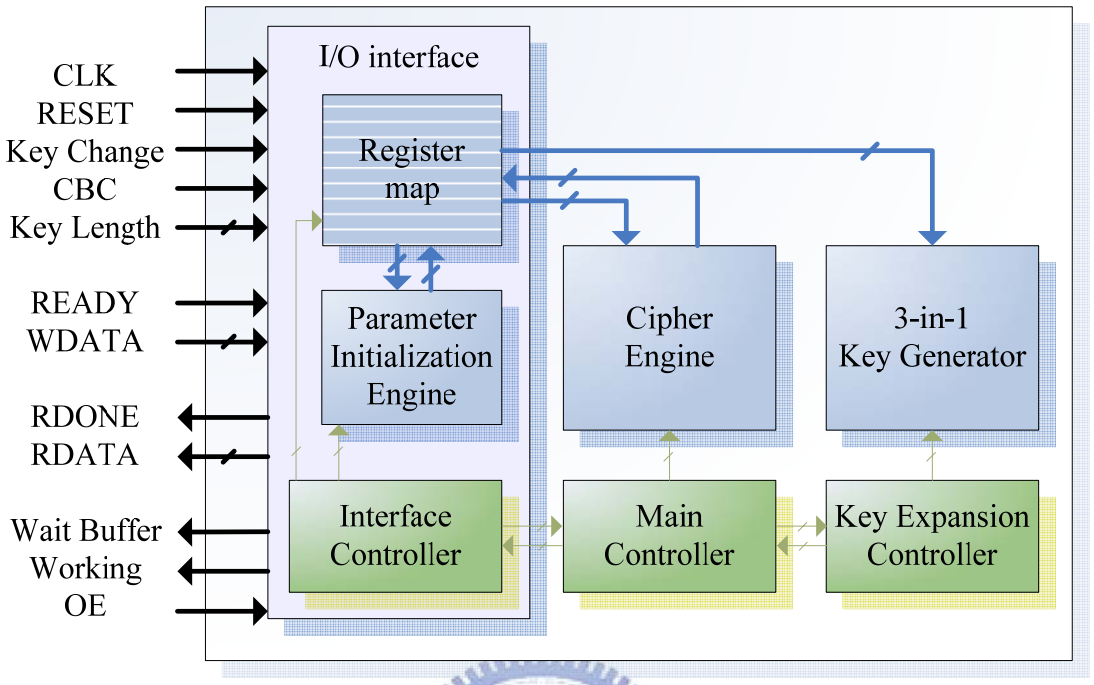


Fig. 5.1 Block diagram of the C-AES coprocessor.

The design of the cipher engine has been shown in section 3.4.2, and the architecture was depicted in Fig. 3.7. The *SubBytes*(), *ShiftRows*(), *MixColumns*(), and *AddRoundKey* transformations was rearranged and merged such that the data path appears in a more regular way for both encryption and decryption.

The round keys used during the encryption/decryption procedure are expanded on the fly by the key generator, and the architecture of key generator was described in Fig. 4.9. It was designed to support all specified key lengths and produces one 128-bit round key per clock cycle to cooperate with the cipher engine.

Table 5.1 Pin definition of C-AES coprocessor

Signal name	Direction	Width	Description
CLK	I	1	Clock signal.
RESET	I	1	Reset signal.
Key Change	I	1	Reload controller of initial cipher key.
CBC	I	1	1: CBC mode / 0: ECB mode.
Key Length	I	2	00: 128-bit / 01: 192-bit / 10: 256-bit.
READY	I	1	The valid signal of WDATA.

WDATA	I	32	The write data bus from the bus.
RDONE	O	1	The valid signal of RDATA.
RDATA	O	32	The read data bus to the bus.
Wait Buffer	O	1	Indicates if the I/O buffer is full.
Working	O	1	Indicates if the cipher engine is working.
OE	I	1	Indicates if the slave gets access to the bus.

The detailed design of other modules, such as the controllers, and I/O interface are discussed in the following paragraphs.

5.2 I/O Interface

The I/O interface is designed to be compatible with AMBA AHB slave protocol in order to make our C-AES coprocessor easily to integrate into a system. The 32-to-128-bit input buffer caches the 32-bit input data from the data bus to form a block of the necessary data, while the output buffer is used to cache the 128-bit output block from the cipher engine.

5.2.1 Input Interface

In our proposed architecture, besides the initial cipher key, IV, and text, the parameters of each transformation also need to be given. The bit number of each parameter is listed in Table 5.2. Thus, it requires 10 clock cycles to transmit these parameters via 32-bit data bus.

Table 5.2 The bit number of each changeable coefficient.

Parameters	Bit number (bits)
isomorphism matrix **(** δ' **)**	64
inverse isomorphism matrix **(** δ'^{-1} **)**	64
affine matrix **(** A **)**	64
inverse affine matrix **(** A^{-1} **)**	64
affine constant **(***const*(*x*)**)**	8
Irreducible polynomial **(***m*(*x*)**)**	8
row vector of C **(**$[c_0, c_1, c_2, c_3]$**)**	32

Based on the schedule list in Fig. 5.2, the order of data transfer is determined by

the processing mode command. Initially, the data is transferred in the order parameters, key, IV, text. Thus, the most critical latency, which requires 26 clock cycles, occurs in CBC mode, and then, the parameter initialization and encryption/decryption process will be performed. If the parameters and key are given at the beginning and not changed, the following data transfer of input is only required 4 clock cycles to transmit text.

Such a series of data movement and control in the input interface are achieved by enable a pointer to contain the address. It is denoted as WADDR in Fig. 5.2, and the destination transfer address is assigned by the interface controller. While the last transfer address is reached, it also means one 128-bit text block is ready at input interface. Once the cipher engine is not working and output buffer is not full, the main controller will take over the control and execute the encryption/decryption procedure. Otherwise, the signal of " Wait Buffer " will be pull HIGH to indicate that the new data can not be written to the input interface.

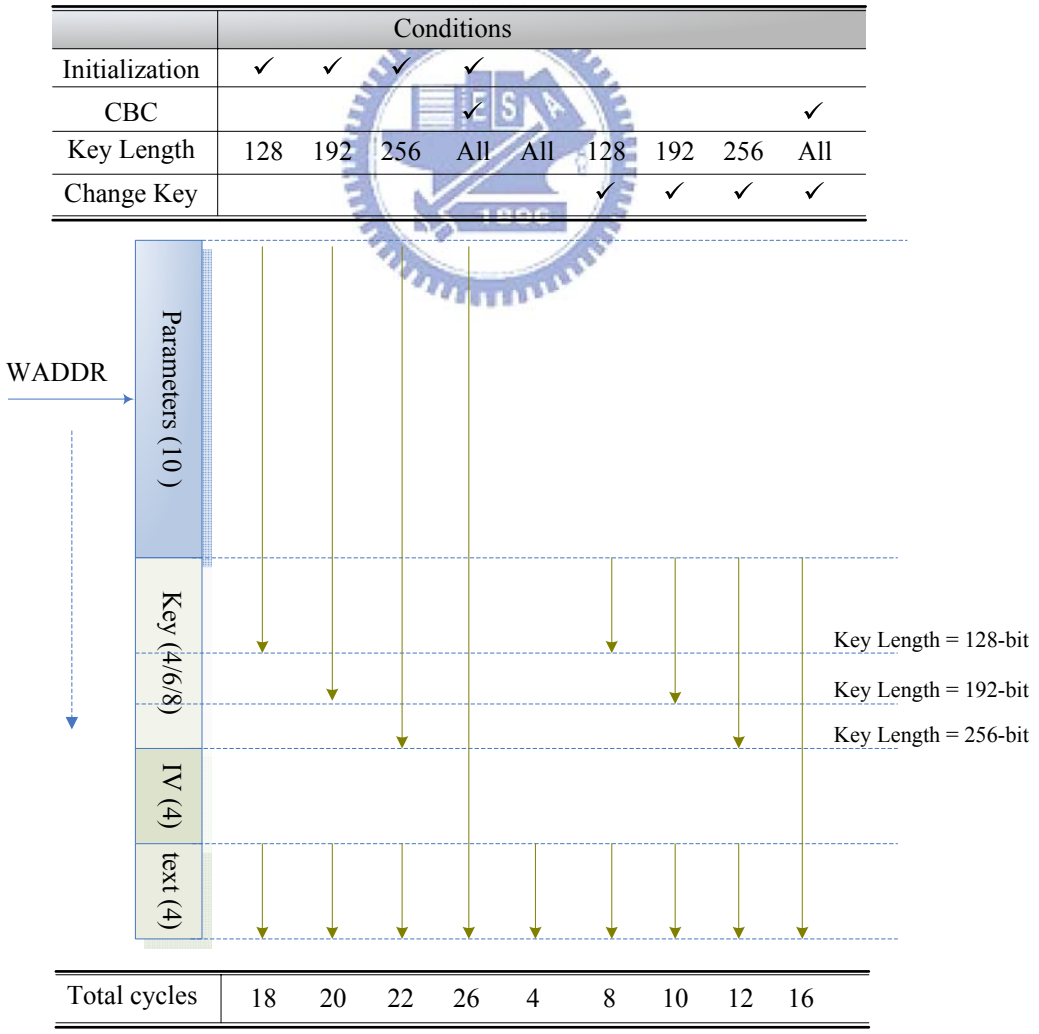


Fig. 5.2 Clock distribution in the different transfer modes.

5.2.2 Output Interface

The output interface is used in a transmission that contains two separate 128-bit buffers. While one buffer is prepared to receive the next 128-bit output from the cipher engine, the data in the other buffer is being sent to the data bus. Also as described in above section, only if one of the output buffers is empty, the cipher engine will write the encryption/decryption result to the output buffer, read the new data from the input buffer and continue the computation.

5.3 Parameter initialization Engine

The parameter initialization engine contains computation logics and several registers to generate and store the necessary coefficients for our cipher engine and key generator. Fig. 5.3 shows the block diagram of this module. There are 8 64–bit registers and 3 8-bit registers used to store all necessary parameters listed in Table 5.3 .

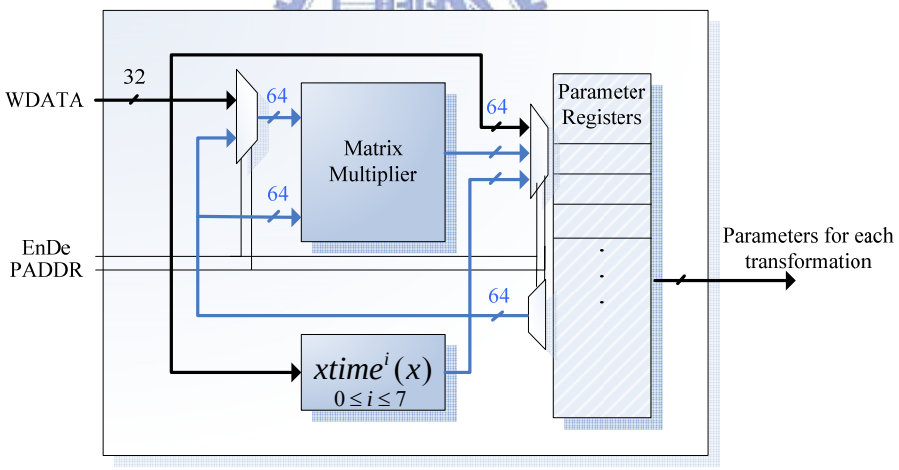


Fig. 5.3 Simple block diagram of parameter initialization engine.

Since the shortest latency shown in Fig. 5.2 is 18 clock cycles, the parameter initialization can spent 18 cycles to compute, and will not introduce more delay into latency. Thus, in order to reduce hardware cost, the compatible input order of parameters and the architecture, which uses only 8 8-bit matrix multipliers, is proposed and the configuration time just matches 18 cycles. The detailed input order and computation schedule is listed in Fig. 5.4. In addition, the calculation of C_{c0}, C_{c1}, C_{c2}, C_{c3}, are achieved by $xtime^i(c_j)$, $0 \le i \le 7$, $0 \le j \le 3$, which is described in

Section 3.3.

Table 5.3 The necessary parameters for the cipher engine and the key generator.

Parameters		Bit Number (bits)
Encryption	Decryption	
δ'		64
δ'^{-1}		64
$A \cdot \delta'^{-1}$		64
$\delta' \cdot A^{-1}$		64
$\delta' \cdot C_{c0} \cdot A \cdot \delta'^{-1}$	$\delta' \cdot A^{-1} \cdot C_{c0} \cdot \delta'^{-1}$	64
$\delta' \cdot C_{c1} \cdot A \cdot \delta'^{-1}$	$\delta' \cdot A^{-1} \cdot C_{c1} \cdot \delta'^{-1}$	64
$\delta' \cdot C_{c2} \cdot A \cdot \delta'^{-1}$	$\delta' \cdot A^{-1} \cdot C_{c2} \cdot \delta'^{-1}$	64
$\delta' \cdot C_{c3} \cdot A \cdot \delta'^{-1}$	$\delta' \cdot A^{-1} \cdot C_{c3} \cdot \delta'^{-1}$	64
$m(x)$		8
c_A		8
$\delta \cdot A^{-1} \cdot c_A$		8

Input Order		Computation	
		Encryption	Decryption
$c_A, m(x)$	1		
c_0, c_1, c_2, c_3	2	$\Rightarrow C_{c0}, C_{c1}, C_{c2}, C_{c3}$	
	3		
A^{-1}	4		
	5		
δ'	6	$\Rightarrow \delta' A^{-1}$	
	7	$\Rightarrow \delta' A^{-1} Ca$	
δ'^{-1}	8		
	9		
A	10	$\Rightarrow A\delta'^{-1}$	
	11	$\Rightarrow \delta' C_{c0}$	$\Rightarrow \delta' A^{-1} C_{c0}$
	12	$\Rightarrow \delta' C_{c1}$	$\Rightarrow \delta' A^{-1} C_{c1}$
	13	$\Rightarrow \delta' C_{c2}$	$\Rightarrow \delta' A^{-1} C_{c2}$
	14	$\Rightarrow \delta' C_{c3}$	$\Rightarrow \delta' A^{-1} C_{c3}$
	15	$\Rightarrow \delta' C_{c0} A\delta'^{-1}$	$\Rightarrow \delta' A^{-1} C_{c0} \delta^{-1}$
	16	$\Rightarrow \delta' C_{c1} A\delta'^{-1}$	$\Rightarrow \delta' A^{-1} C_{c1} \delta^{-1}$
	17	$\Rightarrow \delta' C_{c2} A\delta'^{-1}$	$\Rightarrow \delta' A^{-1} C_{c2} \delta^{-1}$
	18	$\Rightarrow \delta' C_{c3} A\delta'^{-1}$	$\Rightarrow \delta' A^{-1} C_{c3} \delta^{-1}$

Fig. 5.4 The computation schedule of parameter initialization.

Chapter 6
Verification and Result Comparison

In this chapter, the design methodology and verification based on Intellectual Property (IP) reuse are introduced. In section 6.1, the rules in IP Qualification (IPQ) Guidelines are described. We follow these rules to implement the synthesizable HDL code of our design in the front end. Moreover, the chip design flow and verification in each level are illustrated in section 6.2 and 6.3. Finally, the experimental results and comparison are given in section 6.3.

6.1 IP-Based Design

IP-based and platform-based designs are more and more important in SoC (System-on-Chip) era. The design time can be decreased to meet the increasing complexity on single chip by using the reusable IP, and let the verification more efficient by the platform-based design flow. Generally speaking, Silicon Intellectual Property (SIP) may be divided into three types described as follow. In our proposed design, the soft IP implementation is focused in the front end.

(1) Soft IP indicates that IP designed in the form of synthesizable HDL code.
(2) Firm IP indicates that IP delivered in the form of gate-level netlist after synthesis.
(3) Hard IP indicates that IP delivered generally in the form of GDSII format, which is fully placed, routed and optimized for power, size, or performance, and mapped to specific process technology.

6.1.1 IP Qualification Guideline Overview

The general rules proposed in the IP Qualification (IPQ) guidelines are a set of best practices for creating reusable designs for use in an SoC design methodology.

There practices are based on several reusable methodology literatures and experiences from Steering Committee members of IPQ Alliance in developing reusable designs. Reusable macros that have already been designed and verified help users aware of all need-to-know issues in advance. If the blocks do not conform to this standard for reusable methodology, the efforts for integrating pre-existing blocks into new SoC could become excessively high.

The quality criteria, which have to be taken into account, come from various sources: The reuse methodology manual (RMM) contains a set of rules and guidelines that help ensure that a design is reusable and technology-independent. IPQ describes that language subset of VHDL or Verilog that are synthesizable and verifiable with any compliant tool. Further efforts on quality are under way in the Virtual Socket Interface Alliance (VSIA).

6.1.2 Soft IP Design Flow

The standard soft IP design flow is illustrated in Fig. 6.1. IP creators must follow the rules in the IP Qualification guidelines, which are the basis for industry-wide solutions to develop reusable and higher quality IP. Here, the IPQ guidelines classify the reusable methodology into three categories:

(1) Design guidelines:

The design guidelines include coding rules and design issues. Soft IP that follows the rules can ensure that the HDL code is readable, portable and reusable. In addition, these rules also help achieve optimal synthesis and simulation results. The guidelines are categorized as follow:

- HDL (Verilog & VHDL) coding guidelines.
- Design style guidelines.
- Synthesis script guidelines.

(2) Verification guidelines:

In verification guidelines, a set of rules are provided which need to be followed by IP creators to improve the verification quality of the IP. The guidelines are categorized as follow:

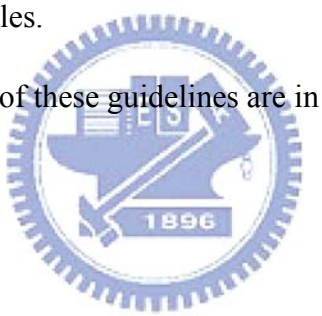
- Soft IP verification guidelines.
- Coding guidelines in writing testbench codes.
- IP prototyping.

(3) Deliverable guidelines:

In verification guidelines, the rules ensure that users can obtain all the necessary information about this IP. According to the documents and script files provided by IP creators, users can rebuild the whole design on their workstations or servers. The guidelines are categorized as follow:

- General deliverables.
- Documentation deliverables.
- Design files deliverables.
- Verification deliverables.
- Hardware related software deliverables.
- IP prototyping deliverables.

The detailed descriptions of these guidelines are in the IP Qualification v1.0.



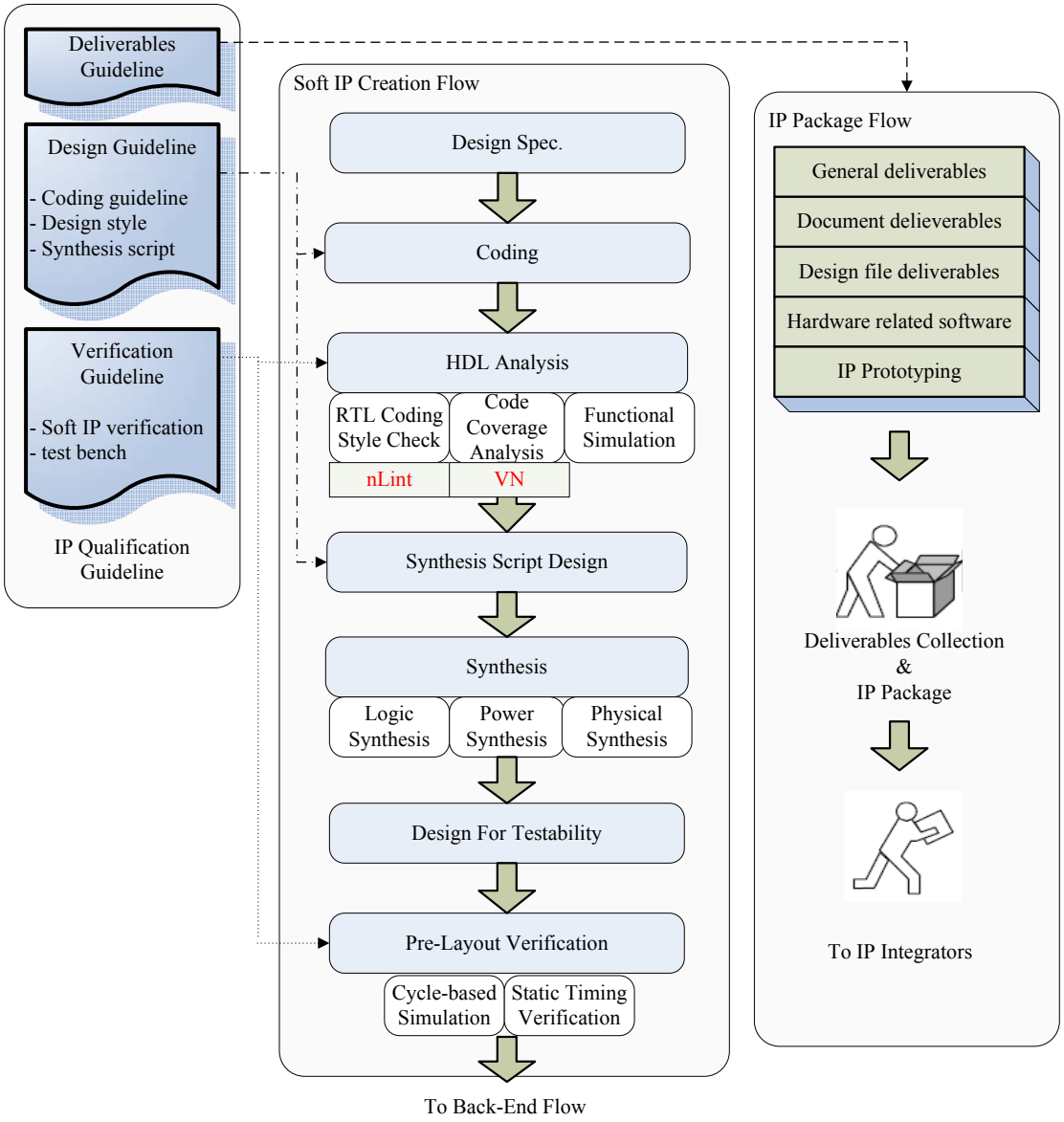


Fig. 6.1 Soft IP design flow.

6.2 Chip Design Flow

Our chip design flow is shown in Fig. 6.2. The RTL code is designed and simulated in Verilog-XL compiler, and Synopsys Synthesis tool is used to synthesis our design with one scan chain and create the gate level netlist.

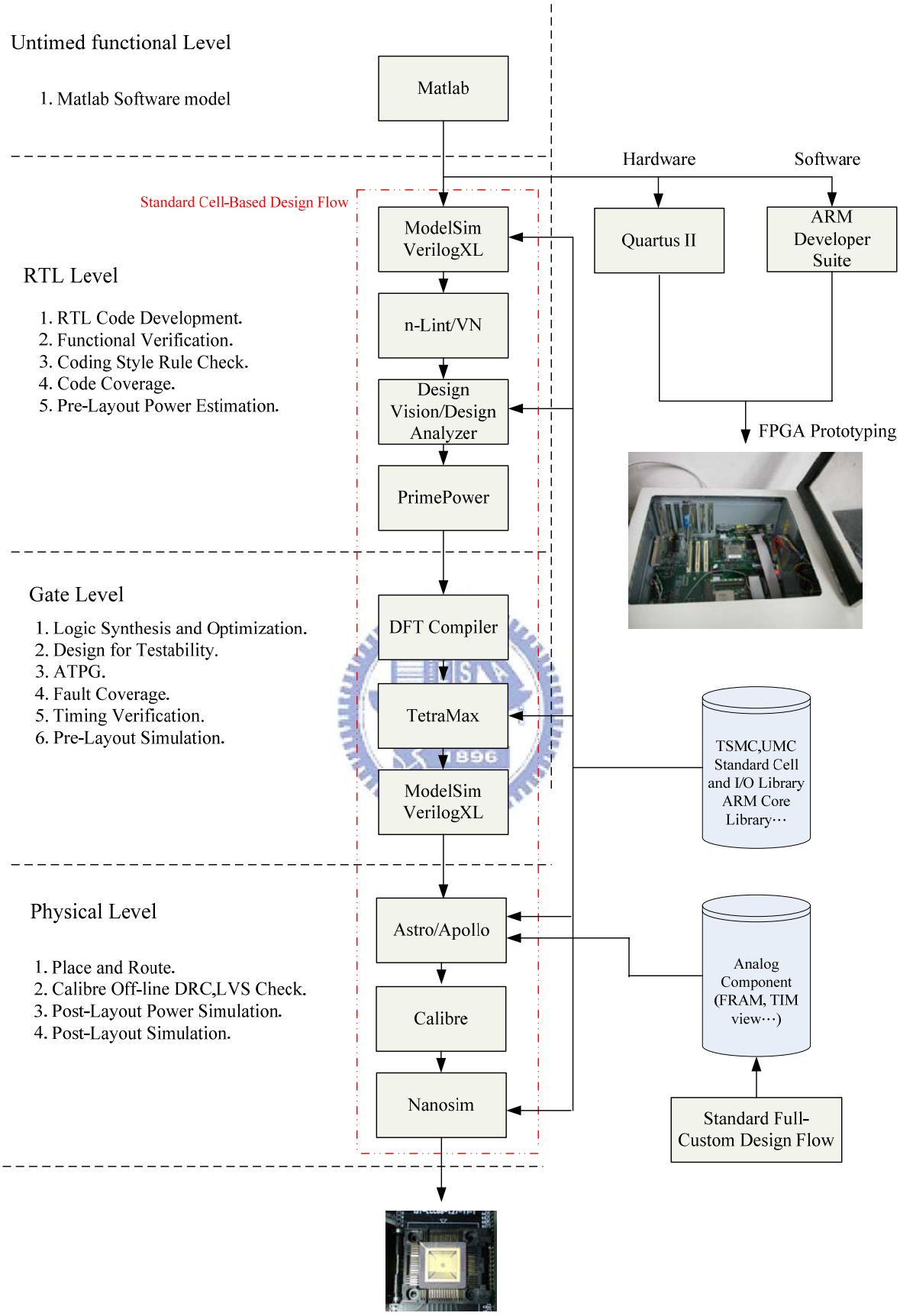


Fig. 6.2 Cell-based design flow

Then, the gate level netlist is applied to gate level simulation and compared the result with RTL code simulation to check out the correctness. We use Apollo to placement and routing, and Calibre to check DRC and LVS result. After post-layout level gate simulation is correct, NanoSim is exploited to take transistor level simulation.

6.3 Verification Strategy

Since a single verification strategy would not sufficiently handle the complexity in SoC problems, a multilevel verification approach is developed. It contains several functional models to verify a single IP, and will increase the verification speed and efficiency at the system level. In the following sections, the implemented functional models and verification are described.

6.3.1 Untimed functional model

The first complete model of our proposed design is presented in abstract form as an untimed functional model (UFM), in which all functionality is implemented with MATLAB to verify the correctness of the configurable AES algorithm. Besides, it can also produce the test patterns efficiently for following simulation models. The MATLAB software model is shown in Fig. 6.3.

6.3.2 Timing Accurate model

The timing accuracy of a model illustrates how similarly it behaves to the constraints of the final design with respect to time. In our proposed design flow, the synthesis tool generates the timing accurate gate-level netlist from the RTL code, and the gate propagation delays are analyzed by those constraints defined in the specification of UMC 0.18μm CMOS technology. After synthesis, the gate-level simulation at the highest estimated operation frequency is needed for verifying the correctness of the synthesis result.

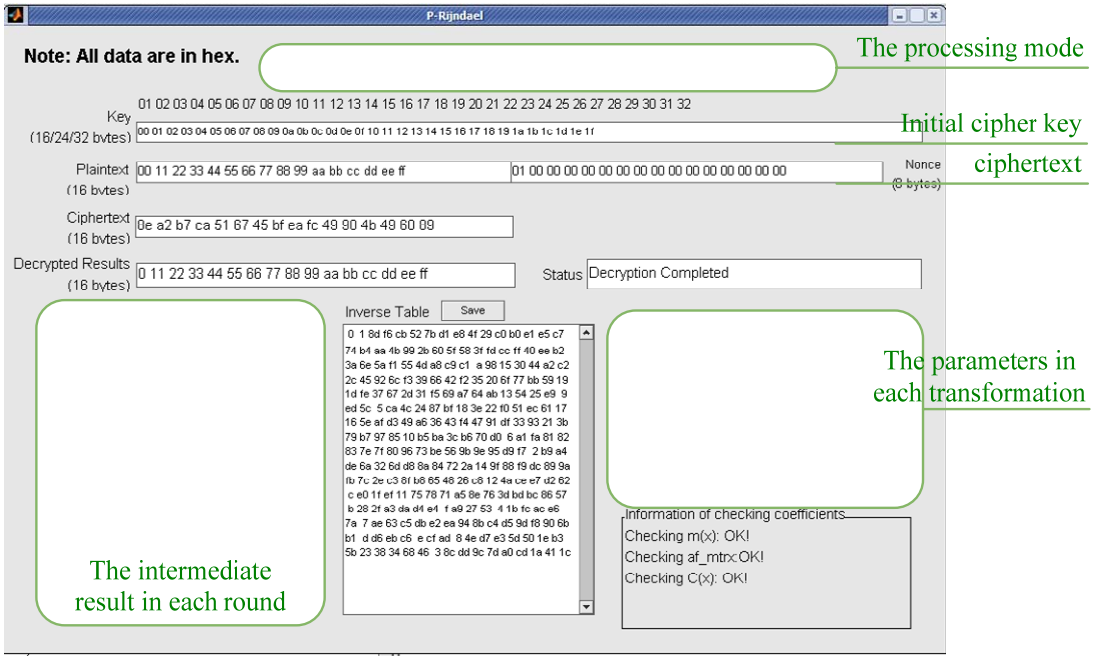


Fig. 6.3 MATLAB software model.

6.3.3 FPGA Prototyping

An FPGA prototyping is implemented on the ARM Integrator/Logic Module (LM), which provides a platform for developing digital IPs on the AMBA-based SoC design. The ARM Integrator contains ARM CPU, AMBA bus and FPGA. The further details about this platform are described in [37], and the system architecture of the C-AES coprocessor on the ARM Integrator is shown in Fig. 6.4. Within LM, the registers listed in Table 6.1 are mapped to our C-AES coprocessor. Thus, the ARM CPU on core module can manipulate our C-AES coprocessor easily by these registers. Fig. 6.5 shows that we debug in the ARM Developer Suite (ADS), and a test bench of the encryption/decryption loop is simulated.

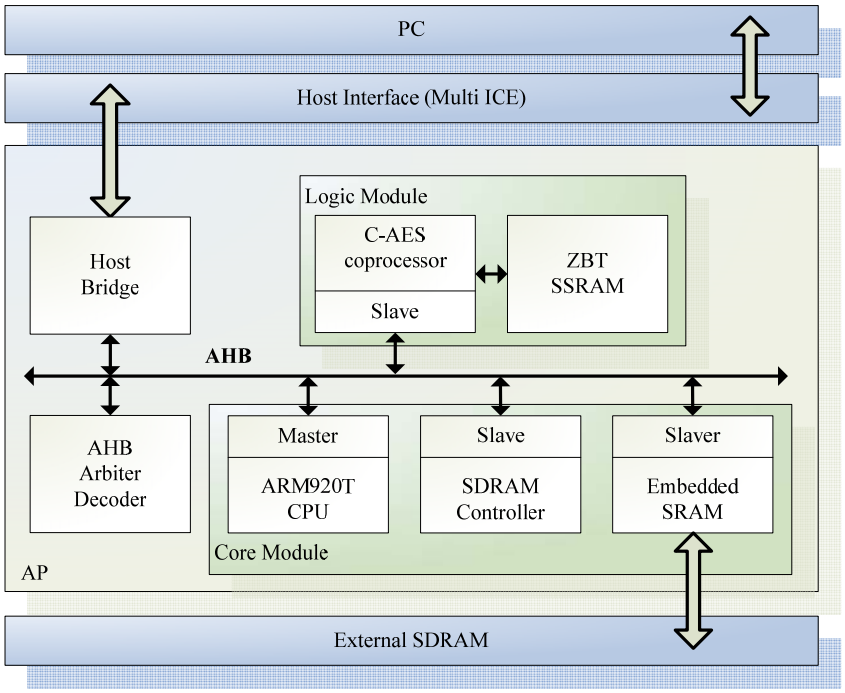


Fig. 6.4 The C-AES coprocessor on the ARM Integrator.

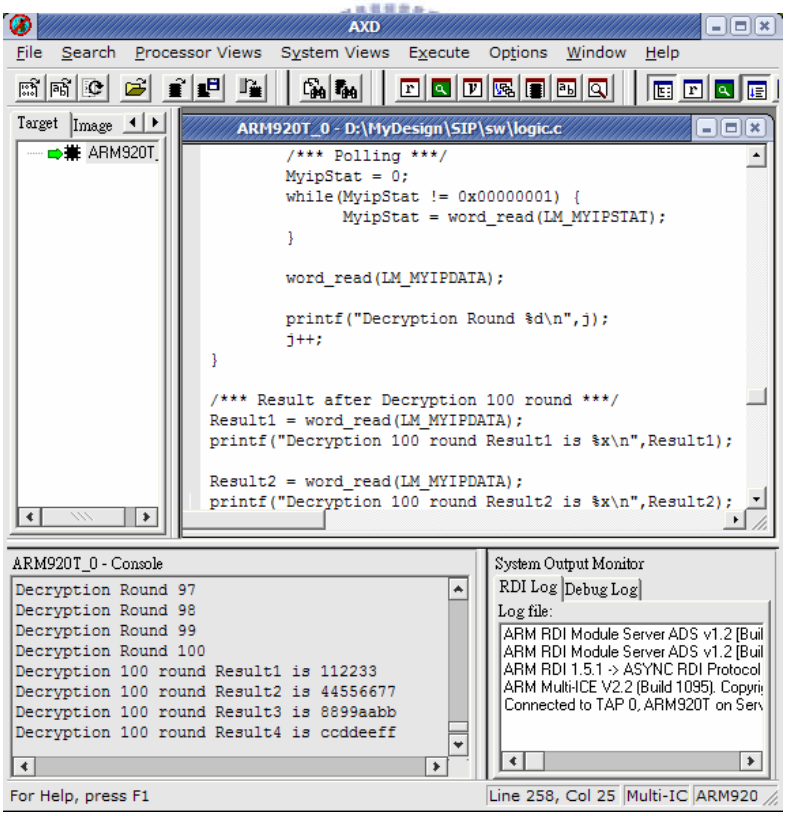


Fig. 6.5 The hardware driver running on the ARM ADS.

Table 6.1 Register map of the C-AES coprocessor.

address	Size	Function
0xCC00_0000	9	Each bit represents the control or response signal of the C-AES coprocessor separately, such as [RESET, Key Change, CBC, Key Length, RDONE, Wait Buffer, Working, OE]
0xcCC0_0004	32	Represents WDATA or RDATA according the direction of data transfer.

6.3.4 Coding Style Rule Check

A programmable rule checker has been integrated in the IP Qualification framework. The SpringSoft nLint is used for static lint checking. The lint tool can find errors and warnings in many aspects including naming, synthesis, simulation and DFT issues. Common syntax errors, such as typing errors, unmatched bus width, and undeclared objects, can be quickly located. Moreover, some logical errors like unreachable state can also be found. The lint tool indicates bad coding style that may load to poor readability and reusability. Our proposed design passes the lint tool checking with all rules defined by IPQ Alliance.

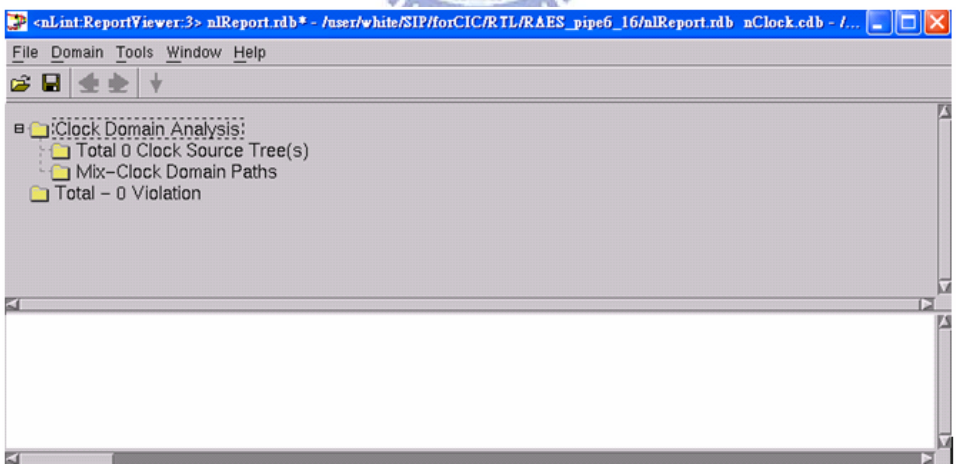


Fig. 6.6 The report of coding style rule check

6.3.5 Code Coverage

Generally speaking, a coverage-driven verification methodology makes the verification flow more complete and efficient, and coverage report gives us a sense of

the good and the bad of our HDL design and test bench. The coverage-driven verification can be performed using several coverage metrics. A simple example of these metrics is the code coverage. By investigating the code coverage helps the designer find untested or redundant code in early stage of development and the quality of the stimuli can be measured. Therefore, coverage gives the information that you need to know when you are ready for RTL sign-off. With a high coverage score, you can have more confidence that the code, in passing, works correctly, and we use Verification Navigator to measure the code coverage. The report is listed in Fig. 6.7.

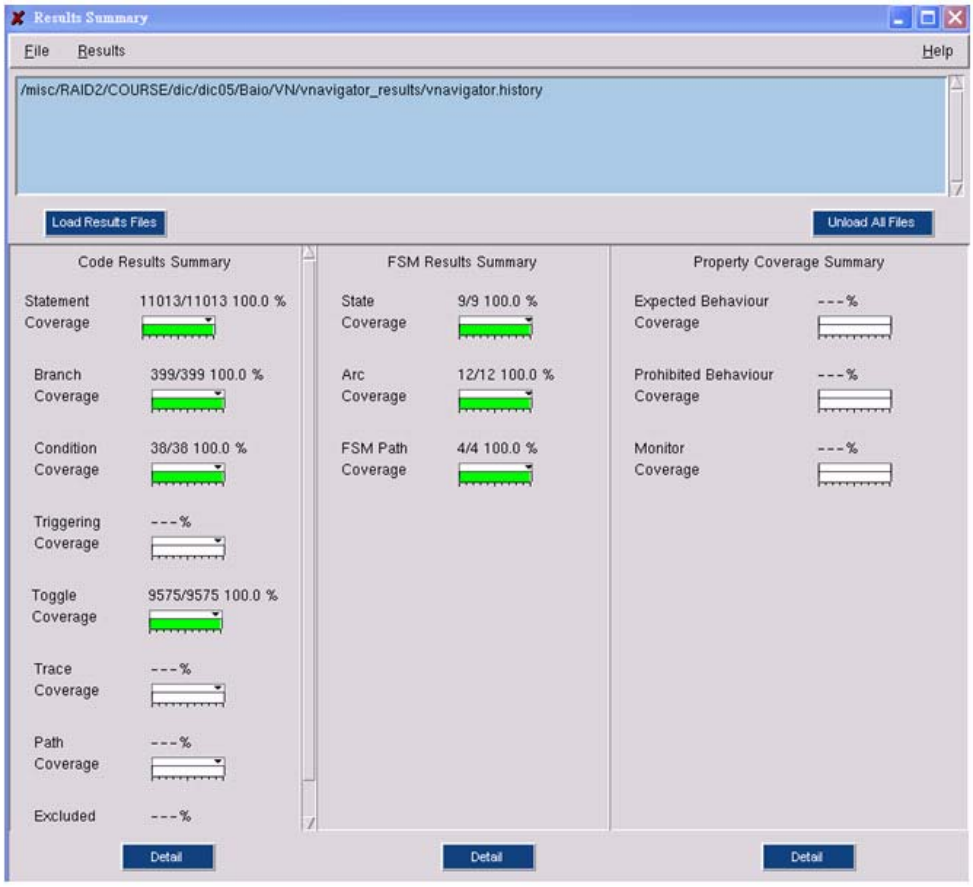
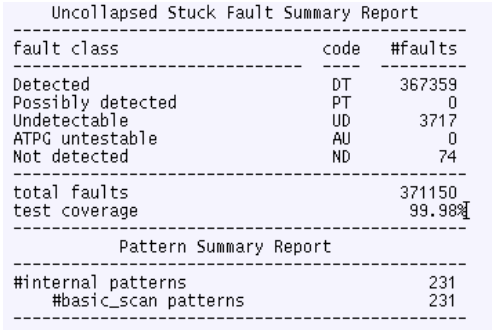


Fig. 6.7 The report of code coverage estimated by Verification Navigator.

6.3.6 Design for Testability

Considering the ASIC testing, the scan chain design is inserted. In our design flow, the Synopsys DFT compiler is used to conduct in-depth testability analysis at the Register Transfer Level (RTL), and to implement the effective test structures at the hierarchical block level. The report of fault coverage shown in Fig. 6.8 is calculated by TetraMax, and it is 99.98% with 231 test patterns.



```
    Uncollapsed Stuck Fault Summary Report
-----------------------------------------------
fault class                     code   #faults
------------------------------  ----  ---------
Detected                         DT     367359
Possibly detected                PT          0
Undetectable                     UD       3717
ATPG untestable                  AU          0
Not detected                     ND         74
-----------------------------------------------
total faults                             371150
test coverage                             99.98%
-----------------------------------------------
          Pattern Summary Report
-----------------------------------------------
#internal patterns                          231
    #basic_scan patterns                    231
-----------------------------------------------
```

Fig. 6.8 The report of fault coverage calculated by TetraMax.

6.3.7 Physical Verification

In physical verification, Automatic Placement and Routing (APR), on-line Design Rule Check (DRC) and Layout Versus Schematic (LVS) are done by Synopsys Astro, and off-line DRC and LVS are verified by Mentor Graphics Calibre. Finally, the post-layout simulation is passed using Verilog-XL.

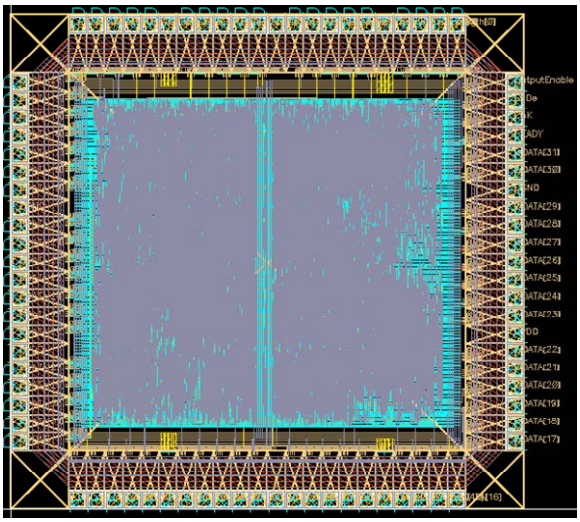
6.4 Results and Comparisons

The C-AES coprocessor design has been implemented using a UMC 0.18μm CMOS technology. It was synthesized using a standard-cell library. The critical path of only about 3.84ns shown in Table 6.2 is obtained.

Table 6.2 The comparison between cipher engine and key generator.

	Cipher Engine	**Key Generator**
Gate Counts (K)	38.55	21.68
Percentage of area size (%)	47.60	26.77
Critical path (ns)	3.33	3.84

Fig. 6.9 shows a chip layout of the C-AES coprocessor, and the whole chip has a size of around $1.72 \times 1.66 mm^2$, with a gate count of around 80,986 gates. The I/O interface takes 25.42% of the overall area, since the selectors with wide data width and the registers for storing IV, initial cipher key, text, and parameters described in Sec. 5.3 are required. The key generator module consumes about 26.77% of the area, and the main cipher engine module occupies 47.60% of the overall area. All these data are summarized in Table 6.3.



Technology	UMC 0.18μm CMOS technology
Core VDD	1.8V
Clock Rate	250MHz
Core Size	$1.721 \times 1.655 mm^2$
Gate Count	80.99K
Throughput	3.20Gbps (128-bit key) 2.67Gbps (192-bit key) 2.29Gbps (256-bit key)

Fig. 6.9 Chip layout and feature of C-AES coprocessor.

In Table 6.4, we compare our implementation with some other AISCs presented recently. In [12], the authors presented a test chip that provides the AES encryption and decryption with different block sizes (128, 192, and 256 bits) and key lengths (128, 192, and 256 bits). Here the best performance (with the block size of 128 bits) is shown for comparison. Due to its LUT-based implementation of the S-Box, the hardware cost is high.

Table 6.3 Area statistics of C-AES coprocessor.

Component	Gate counts (K)	Percentage (%)
Cipher Engine	**38.55**	**47.60**
- *ShiftRows*()	0.35	0.43
- 16 8-bit inverters in $GF(2^4)$	5.31	6.56
- 16 8-bit matrix multipliers	4.23	5.22
- new *MixColumns*()	18.32	22.62
Key Generator	**21.68**	**26.77**
Main controller	**0.17**	**0.21**
Input interface	**17.72**	**21.88**
- Registers	6.61	8.16
- $xtime^i(x)$	1.23	1.52
- 4 8-bit Matrix Multipliers	1.02	1.26
Output interface	**2.87**	**3.54**
- Registers	1.37	1.69
total	**80.99**	**100**

Another case is a pipelined design implemented by a 0.35μm CMOS technology

[23]. The S-Boxes were implemented based on the work reported in [38], instead of LUT-based design. However, theirs supports keys of 128, 192, and 256 bits by the same way described in [12], it requires an addition memory to store all the necessary round keys in advance. Compared with other designs which generate the round keys on the fly, it occupies extensive hardware resources. In addition, it should be noted that a pipelined design has the difficulty in maintaining the same throughput rate in a feedback cipher mode such as CBC. For example, the performance of their 4-stage pipeline design will be scaled down by four in the feedback cipher modes

In [19], the authors presented a compact architecture for the Irondale algorithm, where the hardware resources can be efficiently shared between data encryption, data decryption, and even key expansion. Table 6.4 only shows the result for their 128-bit data path—using one clock cycle for each round. Moreover, the S-Box is also optimized by introducing the composite field $GF(((2^2)^2)^2)$. Since the data paths of 192 and 256-bit key expansion are not suitable for developing the compact hardware, their key generator only supports 128-bit key length, and the round keys are generated on-the-fly. Under the logic optimization applied to the constant arithmetic components, it has a very small gate counts.

In [34], another AES-128 module was implemented, and it is very similar to above concept [19], but has a smaller area. This is mainly because a new common sub-expression elimination (CSE) is applied to reduce the area cost. In addition, they also focus on the merge functions of the affine transformation and *MixColumns*() to increasing throughput. In our design, a new combined architecture described in Sec. 3.4.2 provides a more effective method for high throughput.

In [35], it is also a configurable AES coprocessor that all the encryption and decryption procedures are the same as the original AES algorithm, but *m*(*x*), *Affine matrix*, *const*(*x*) and the row vector *c*(*x*), are all configurable. In their design, additional 16 256×16-bit and 16 64-bit ROMs are used to store all alternatives of these parameters. For this reason, their design can only spend 3 clock cycles for parameter initialization, while our approach, which is described in Sec. 5.3, requires at least 18 cycles to receive the input data and compute the parameters. In order to support the configurability, they extend the composite field arithmetic approach to implement the new data path of round function, which can be processed with a fixed

irreducible polynomial in the composite field $GF((2^4)^2)$. In other words, *MixColumns*() transformation is still executed by the multiplier in composite field $GF((2^4)^2)$. Although the difficulty in providing the configurability is solved, the overhand is quite considerable. It is because that additional 32 8-bit matrix multipliers for S-Boxes, and 64 8-bit field multipliers for *MixColumns*() are required in the data path of round function. Thus, our approach, which combines the matrix multipliers in S-Boxes and *MixColumns*(), provides a successful solution for high speed and low cost.

Table 6.4 Comparison of AES designs

	Verbauwhede [12]	**Su** [23]	**Satoh** [19]	**Hsiao** [34]	**Su** [35]	**Ours**
Technology	0.18μm	0.35μm	0.11μm	0.18μm	0.25μm	0.18μm
Clock Rate	125MHz	200MHz	224MHz	117MHz	66MHz	250MHz
Throughput	1.6	2.381			0.844	3.20
(Gbps)	1.33	2.008	2.61	1.49	0.704	2.67
	1.14	1.736			0.603	2.28
Gate Counts	173K	58.430K	21.337K	16.917K	200.5K	80.99K
Throughput/	9.25	41.49			4.21	39.51
Gate Count	7.68	34.98	122.23	88.08	3.51	32.97
(Kbps/gate)	6.59	30.24			3.01	28.15
On-the-fly Key Generation	No	No	Yes	Yes	Yes	Yes
configurability	No	No	No	No	Yes	Yes

Although the throughput to gate count ratio is about 3 times smaller to the best value reported in Table 6.4, our C-AES coprocessor can easily provide a new encryption algorithm by arbitrarily selecting a combination of the parameters, and all specified key lengths can be supported. Considering the pre-gate throughput rate and functionality, our design is quite competitive.

Chapter 7
Conclusions and Future Work

7.1 Conclusions

A Rijndael algorithm with changeable coefficients is designed in this work, and we use the on-the-fly key generator instead of memory-based key scheduler to reduce the hardware resources. In addition, ECB and CBC operating modes are supported in our design. The whole chip has a size of around $1.72 \times 1.66 mm^2$, with a gate count of around 80,986 gates. The throughput is about 3.2Gbps for 128-bit keys, 2.67Gbps for 192-bit keys, and 2.29Gbps for 256-bit keys, respectively. The goal of this design is providing customized security for virtual private network (VPN) application. In VPN, sessions do not need to compatible with standard traffics; hence, the enterprise can configure their own coefficients to protect their network. In addition, our designs provides throughput over gigabit per seconds, so they are suitable for Fast Ethernet or Giga Ethernet.

7.1 Future Work

The C-AES coprocessor is also designed to operate in AMBA system with AHB specification. It resides in the address map of an ARM compatible processor, and serves as a coprocessor to provide encryption computing. Someday, it may become the building block of network security processor. To manipulate the data transfer efficiently, the future work may lie in developing the AHB bus mastering capability to off-load data movement and encryption operations form the host processor. Thus similar to the behavior of DMA devices, the data transfer and processing are all done by our coprocessor, which leaves the host processor to execute more sophisticated flow control or exception handling.

Bibliography

[1] National Institute of Standards and Technology (NIST), *Advanced Encryption Standard* (*AES*), National Technical Information Service, Springfield, VA 22161, Nov. 2001.

[2] National Institute of Standards and Technology (NIST), *Data Encryption Standard* (*DES*), National Technical Information Service, Springfield, VA 22161, Oct. 1999.

[3] W. Stallings, *Cryptography and Network Security: Principles and Practice. 3rd ed.*, Prentice-Hall Inc., Upper Saddle River, N.J., 2003.

[4] E. Barkan, and E. Biham, "In How Many Ways Can You Write Rijndael?", *Proceedings of ASIACRYPT*, Dec. 1-5, 2002, pp. 160-175, Springer-Verlag, 2002.

[5] P. Fergguson and G. Huston, "What is a VPN?—Part I," *The Internet Protocol Journal*, vol. 1, pp. 2–19, June 1998. http://www.cisco.com/warp/public/759/.

[6] J. Daemen, and V. Rijmen, "AES Proposal: Rijndael,"*AES Algorithm Submission*, Sep. 3, 2000.

[7] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "A comparative study of performance of AES final candidates using FPGAs," *Cryptographic Hardware and Embedded Systems* (*CHES*) *2000*, vol. 1965 of *LNCS*, pp. 125–140, Springer-Verlag, Aug. 2000.

[8] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays," *Proc. RSA Security Conf.*, Cryptographer's Track, vol. 2020 of *LNCS*, pp. 84–99, Springer-Verlag, Apr. 2001.

[9] P. Chodowiec, K. Gaj, P. Bellows, and B. Schott, "Experimental testing of the Gigabit IPSec compliant implementations of Rijndael and triple DES using SLAAC-1V FPGA accelerator board," *Proc. Information Security Conf. (ISC),* vol. 2200 of *LNCS,* pp. 220–234, Springer-Verlag, Oct. 2001.

[10] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta, “A fully pipelined memoryless 17.8 Gbps AES-128 encryptor,” *Proc. Int. Symp. Field-Programmable Gate Arrays* (*FPGA*), (Monterey), pp. 207–215, ACM Press, 2003.

[11] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta, “A fully pipelined memoryless 17.8 Gbps AES-128 encryptor,” *Proc. Int. Symp. Field-Programmable Gate Arrays* (*FPGA*), (Monterey), pp. 207–215, ACM Press, 2003.

[12] I. Verbauwhede, P. Schaumont, and H. Kuo, “Design and performance testing of a 2.29-GB/s Rijndael processor,” *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 569–572, Mar. 2003.

[13] V. Fischer and M. Drutarovsky, “Two methods of Rijndael implementation in reconfigurable hardware,” *Cryptographic Hardware and Embedded Systems* (*CHES*) *2001*, vol. 2162 of *LNCS*, pp. 77–92, Springer-Verlag, May 2001.

[14] S. Morioka and A. Satoh, “A 10Gbps full-AES crypto design with a twisted-BDD S-Box architecture,” *Proc. IEEE Int. Conf. Computer Design* (*ICCD*), (Freiburg, Germany), pp. 98–103, Sept. 2002.

[15] K. Gaj and P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. *Proc. 3rd AES Conf.* (*AES3*). [Online].
Available: http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html

[16] M. McLoone and J. V. McCanny, “Rijndael FPGA implementation utilizing look-up tables,” *IEEE Workshop on Signal Processing Systems*, Sept. 2001, pp. 349–360.

[17] M. McLoone and J.V. McCanny, “Apparatus for Selectably Encrypting and Decrypting Data,” *UK Patent Application* No. 0107592.8, Filed 27, March 2001.

[18] V. Rijmen, “Efficient implementation of the Rijndael S-box.”

http://www.esat.kuleuven.ac.be/˜rijmen/rijndael/sbox.pdf.

[19] A. Satoh, S. Morioka, K. Takano, S. Munetoh, “A Compact Rijndael Hardware Architecture with S-box Optimization”, *ASIACRYPT 2001*, Lecture Notes in Computer Science 2248, Springer, 2001, pp. 239-254

[20] J. Wolkerstorfer, E. Oswald, and M. Lamberger, “An ASIC implementation of the AES SBoxes,” *CT-RSA 2002*, vol. 2271 of *LNCS*, pp. 67–78, Springer-Verlag, 2002.

[21] S. Mangard, M. Aigner, and S. Dominikus, “A highly regular and scalable AES hardware architecture,” *IEEE Trans. Computers*, vol. 52, pp. 483–491, Apr. 2003.

[22] Xinmiao Zhang; Parhi, K.K., **“**High-speed VLSI architectures for the AES algorithm”, *IEEE Trans. VLSI Systems*, Vol 12, Issue 9, pp. 957-967, Sept. 2004

[23] T.-F. Lin, C.-P. Su, C.-T. Huang, and C.-W. Wu, “A high-throughput low-cost AES cipher chip,” *Proc. 3rd IEEE Asia-Pacific Conf. ASIC*, (Taipei), pp. 85–88, Aug. 2002.

[24] H. Kuo and I. Verbauwhede, “Architectural optimization for a 1.82 Gbits/sec VLSI implementation of the AES Rijndael algorithm,” *Cryptographic Hardware and Embedded Systems* (*CHES*) *2001*, vol. 2162 of *LNCS*, Springer-Verlag, May 2001.

[25] J. H. Shim, D.W. Kim, Y. K. Kang, T.W. Kwon, and J. R. Choi, “A Rijndael cryptoprocessor using shared on-the-fly key scheduler,” *Proc. 3rd IEEE Asia-Pacific Conf. ASIC*, (Taipei), pp. 89–92, Aug. 2002.

[26] J. Guajardo and C. Paar. “Efficient Algorithms for Elliptic Curve Cryptosystems” *Advances in Cryptology—CRYPTO ’97*, *Lecture Notes in Computer Science*, vol. 1294 pp. 342–356. Springer-Verlag, August 1997.

[27] A. Rudra, P.K. Dubey, C.S. Jutla, V. Kumar, J.R. Rao, and P. Rohatgi. “Efficient Rijndael Encryption Implementation with Composite Field Arithmetic” *Workshop on Cryptographic Hardware and Embedded Systems* (*CHES2001*), pp. 175–188, May 2001.

[28] C Paar, “Efficient VLSI Architecture for Bit-Parallel Computations in Galois Fields” PhD Thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994

[29] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, “Unified hardware architecture for 128-bit block ciphers AES and Camellia”, *Cryptographic Hardware and Embedded Systems* (*CHES*) *2003*. Aug. 2003, Springer-Verlag.

[30] IEEE P1363. “IEEE Standard Specifications for Public-Key Cryptography” *IEEE Computer Society*, August 2000.

[31] L. Reyzin, B. Kaliski, “Storage-Efficient Basis Conversion Techniques” *Contribution to IEEE P1363a*, February 2000.

[32] J.L. Fan and C. Paar. “On Efficient Inversion in Tower Fields of Characteristic Two” *International Symposium on Information Theory*, page 20. IEEE, June 1997.

[33] M. H. Jing, Y. H. Chen, Y. T. Chang, and C. H. Hsu, “The design of a fast inverse module in AES,” *Proc. Int. Conf. Info-Tech and Info-Net*, vol. 3, Beijing, China, Nov. 2001, pp. 298–303.

[34] S. F. Hsiao, M. C. Chen, C. S. Tu, “Memory-Free Low-Cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Sunfunctions in Transformations” *IEEE Trans. Circuit and Systems*, VOL. 53, NO. 3, MARCH 2006

[35] C. P. Su, C. L. Horng, C. T. Huang, C. W Wu, “A configurable AES processor for enhanced security” *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific* Vol. 1 Page(s):361 - 366 Jan. 2005

[36] Chih-Hsu Yen, Tsung-Yao Pai, and Bing-Fei Wu, “The Implementations of the Reconfigurable Rijndael Algorithm with Throughput of 4.9Gbps” *Proceedings of 16th VLSI Design/CAD Symposium*, 2005.

[37] Integrator/LM-EP20K600E+ user Guide
http://www.arm.com/pdfs/DUI0146C_LM600.pdf

[38] J. Wolkerstorfer, E. Oswald, and M. Lamberger, “An ASIC implementation of the SBoxes,” *CT-RSA 2002*, vol. 2271 of *LNCS*, pp. 67–78, Springer-Verlag, 2002.