EURASIP Journal on Advances in Signal Processing

RESEARCH Open Access

A reconfigurable and compact subpipelined architecture for AES encryption and decryption

Ke Li, Hua Li* and Graeme Mund

*Correspondence: hua.li@uleth.ca

Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, AB, Canada

Abstract

AES has been used in many applications to provide the data confidentiality. A new 32-bit reconfigurable and compact architecture for AES encryption and decryption is presented and implemented in non-BRAM FPG in this paper. It can be reconfigured for the options of different key sizes which is very flexible for the users to apply AES for various application environments. The proposed design employs a single-round architecture and subpipeling to minimize the hardware cost. The fully composite field GF($(2^4)^2$)-based encryption/decryption and keyschedule lead to the lower hardware complexity and efficient subpipelining for 32-bit data path. In addition, a new subpipelined on-the-fly keyschedule over composite field GF($(2^4)^2$) is proposed for all standard key sizes (128-, 192-, 256-bit) which generates the roundkeys simultaneously and efficiently. This feature is very useful and efficient when the main key has been changed since AES is a symmetric-key cryptography and the session key usually changes frequently. The proposed reconfigurable and compact design has higher throughput and lower hardware cost. It achieves throughputs of 375Mbits/s with 128-bit key, 318Mbits/s with 192-bit key and 275Mbits/s with 256-bit key on VIRTEX XC4VSX25-12, and the total number of slices is 1766. The proposed reconfigurable and compact AES architecture can be efficiently applied in computing-restricted environments such as wireless and embedded devices.

Keywords: AES, FPGA, Pipelining, Fully composite field, Reconfigurable architecture, Computing-restricted environments

1 Introduction

Advanced Encryption Standard (AES) based on Rijndael encryption algorithm has been used to replace DES in security services [1–3]. Hardware AES implementations are attractive because it provides better throughput as well as higher physical security. Compared with Application-Specific Integrated Circuit (ASIC), field programmable gate array (FPGA) becomes more and more popular because of its scalability, re-programmability, and fast development.

Numerous FPGA [4–10] and ASIC [7, 11, 12] implementations of the AES have been presented and evaluated. Other AES implementations have also been proposed such as GPU-based [13], Multicore Processor-based [14], and Rapid Single-Flux-Quantum Circuits-based [15] implementations. Fully unrolled schemes [6, 8] can achieve high

throughput, but there are much more area and energy cost which only suitable for high-end applications. Another approach is only implementing a single-round unit and applies the same unit in different rounds.

In this paper, a compact and reconfigurable design of AES with low hardware cost and adequate throughput is proposed and implemented in a non-BRAM FPGA. This design applies a 32-bit single-round unit, which costs much less hardware area than the 128-bit fully unrolled schemes. In order to reduce the hardware complexity further, we convert the arithmetic operations of AES from field GF(2^8) to field GF((2^4)^2). Unlike the previous designs in [6, 8, 12, 16] where partial-composite field AES is applied, we conduct the entire AES operations in GF((2^4)^2) to minimize the overhead of isomorphic mapping functions. In our design, only two forward mapping functions and one backward mapping function are used. In addition, subpipelining is applied to improve the throughput/area ratio.

The standard announced by NIST [2] indicates that AES is a block cipher with 128-bit block size and 128-, 192-, 256- bit key sizes. These three key sizes are specified for various security levels. The capability to deal with all key sizes makes reconfigurability an important feature of AES implementations. The previous work of [6, 8, 12, 17–20] applied the on-the-fly key generator to support instant key changing. The design in [8] made a subpipelined keyschedule, but it only supported 128-bit key size. When subpipelining on-the-fly keyschedule is employed in an AES implementation, the stages in keyschedule must be synchronized with the stages in the cipher, because they share the same clock. In this design, we propose a subpipelined on-the-fly keyschedule over field GF((2^4)^2), which supports all three key sizes.

The issue of secure communication in computing-restricted environments, such as personal digital assistants (PDAs), wireless devices, and many other embedded devices, has become more important recently. In order to apply AES in these devices, the AES implementations must be cost efficient. The objective of this research work is to design a reconfigurable and compact AES architecture which can be applied to the computing-restricted devices. The proposed architecture can be reconfigured to three different AES key sizes which is very useful when the users change the main key and also change the key sizes for different security levels because AES is a symmetric-key cryptography and the session key usually changes frequently. We also propose a subpipelined on-the-fly keyschedule for three options of key sizes that make the proposed architecture be easily implemented on non-BRAM FPGA.

The remainder of the paper is organized as the following. In Sect. 2, AES algorithm is introduced. The proposed compact and reconfigurable AES architecture is presented in Sect. 3. Implementation and performance are included in Sect. 4. Sections 5 and 6 are the conclusion and future work.

2 AES algorithm

AES is a symmetric block cipher with block size of 128-bit and three key sizes of (128-, 192-, or 256-bit) [1–3]. The AES parameters depend on the key size (Table 1, the size of word is 32 bits): AES runs iteratively on four transformations (inv-/Subbytes, inv-/ShiftRows, inv-/MixColumns and addroundkey) with different sequences in encryption and decryption. Figure 1 illustrates the basic architecture of AES. In the initial round ($r=0$),

Table 1 AES parameters [2]

	Key length (Nk words)	Block size (Nb words)	Number of rounds (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

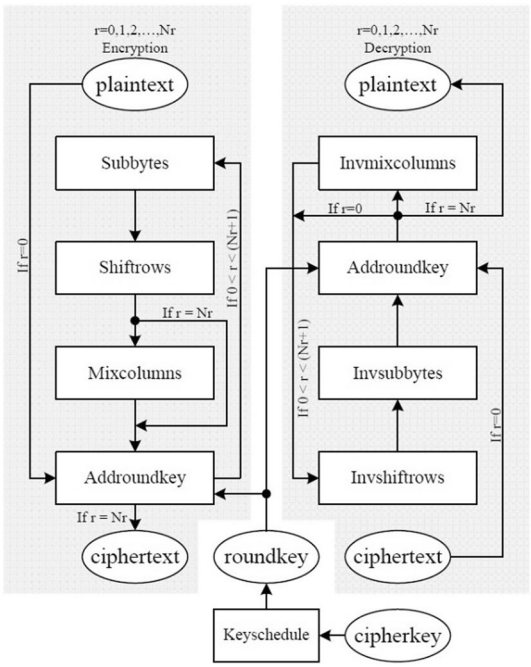

Fig. 1 AES architecture [1, 2]

only addroundkey is performed; in the final round ($r=Nr$), it skips inv-/MixColumns. The keyschedule module expands cipherkey to $(Nr+1)\times 4$ words of roundkeys. Each round applies a unique 128-bit roundkey in the addroundkey operation [1, 2].

2.1 Subbytes

Subbytes are the only non-linear transformation in AES which is also called S-Box. S-Box is a 16×16 matrix containing all possible 256 8-bit values, which is used to perform a non-linear byte-by-byte substitution of the state.

Considering a byte $\{x_7x_6x_5x_4x_3x_2x_1x_0\}$, Subbytes transformation has two steps [1]:

(i) $\{x'_7\, x'_6\, x'_5\, x'_4\, x'_3\, x'_2\, x'_1\, x'_0\}$ is its multiplicative inverse in GF(2^8) field, modulo the irreducible polynomial $m(x)=x_8+x_4+x_3+x+1$; {00000000}'s multiplicative inverse in GF(2^8) field is itself;

(ii) An affine transformation over GF(2) is conducted on the inverse of $\{x_7x_6x_5x_4x_3x_2x_1x_0\}$ (Eq. 1 [1]).

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

2.2 ShiftRows

This transformation circularly shifts each row of the state to the left on encryption. As in Fig. 2 [1], the top row of the state is noted as row(0), and the bottom row is noted as row(3). The ShiftRows perform $i-$byte circular left shift to row(i) ($i=0, 1, 2, 3$).

2.3 MixColumns

This transformation treats each column of the state as a four-term polynomial over GF(2^8) and transforms each column to a new one by multiplying it with a constant polynomial $a(x)=\{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$ modulo x^4+1. Equation 2 [1] is the matrix form of MixColumns.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{2}$$

2.4 Addroundkey

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

2.5 Keyschedule

Keyschedule derives roundkeys from the cipherkey. It consists of key expansion and roundkey selections. Figure 3 [2] shows the keyschedule algorithm which generates roundkeys for AES-128, AES-192 and AES-256. The functions used in keyschedule are the following [1, 2]:

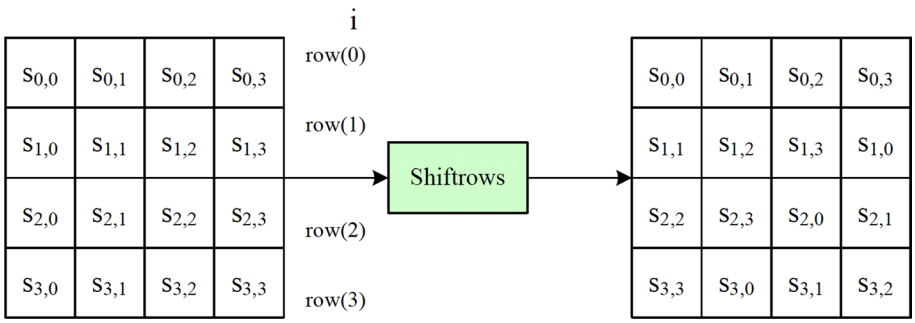

Fig. 2 AES ShiftRows [1]

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

Fig. 3 Pseudo-code for AES keyschedule [2]

- Rotword: One-byte circular left shift on a word.
- Subword: Using S-Box to perform a byte substitution on each byte.
- XOR with Rcon: XORing with a round constant Rcon[j], Rcon[j] = (RC[j], 0, 0, 0) with RC[1] = 1, RC[j] = 2 · RC[j − 1].

3 32-bit subpipelined reconfigurable and compact architecture for AES

In this section, the 32-bit reconfigurable and compact AES architecture is proposed. In our design, the data path is 32-bit. That is one operation, for example, s-box, will be applied four times to process the 128 bits of one block in plaintext. The subpipelined on-the-fly keyschedule for different key sizes is also presented to provide the roundkey simultaneously and efficiently. In addition, the equivalent cipher [1] is adopted to make the same data flow for encryption and decryption and to share the reusable units.

3.1 32-bit single-round unit

Roll unfolded architecture is widely used to achieve high throughput. It conducts multiple rounds on one block by implementing more than one round units on the hardware. The more round units the architecture includes, the higher the hardware cost. The alternative scheme, which is called the single-round unit architecture, can be applied to simplify the hardware complexity. Instead of unfolding all the round units in devices, it implements a single-round unit which costs approximately $1/N_r$ area of the unfolded scheme.

We propose a 32-bit single-round unit for a compact AES architecture. It needs iterating four times to perform a round on a block (128-bit), once every 32 bits.

3.2 Full composite field architecture with keyschedule

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

The key step of S-Box is calculating multiplicative inverse of each byte. Since the introduction of composite field GF((2^4)^2), the calculation of multiplicative inverse over GF((2^4)^2) has been investigated [6, 8, 12, 16]. The architectures in [5, 18, 21] applied the field GF((2^4)^2) to affine transformation in S-Box. By decomposing these operations from GF(2^8) to its subfield GF(2^4), the hardware complexity of S-Box can be decreased dramatically.

In Fig. 4a, in each round before S-Box, it needs an isomorphic mapping function (MAP) from GF(2^8) to GF((2^4)^2); and the inverse mapping (MAP^{-1}) afterwards. If key size is 128 bits, it applies 10 times S-Box to the plaintext and the cipherkey, which means that it needs 20MAPs and 20 MAP^{-1} s for the encryption of 128-bit data. In order to save the cost of MAP and MAP^{-1}, we propose a new 32-bit complete composite field approach (Fig. 4b). The GF((2^4)^2) field applies in all transformations in encryptor/decryptor and keyschedule. As illustrated in Fig. 4b, one MAP and one MAP^{-1} are applied in encryption, and one MAP is applied in keyschedule. This is a constant overhead which is not affected by the number of rounds.

We use the composite field defined by Wolkerstorfer et al. [21]. There are two irreducible polynomials (Eqs. 3 and 4) involved in multiplication and inversion in GF((2^4)^2).

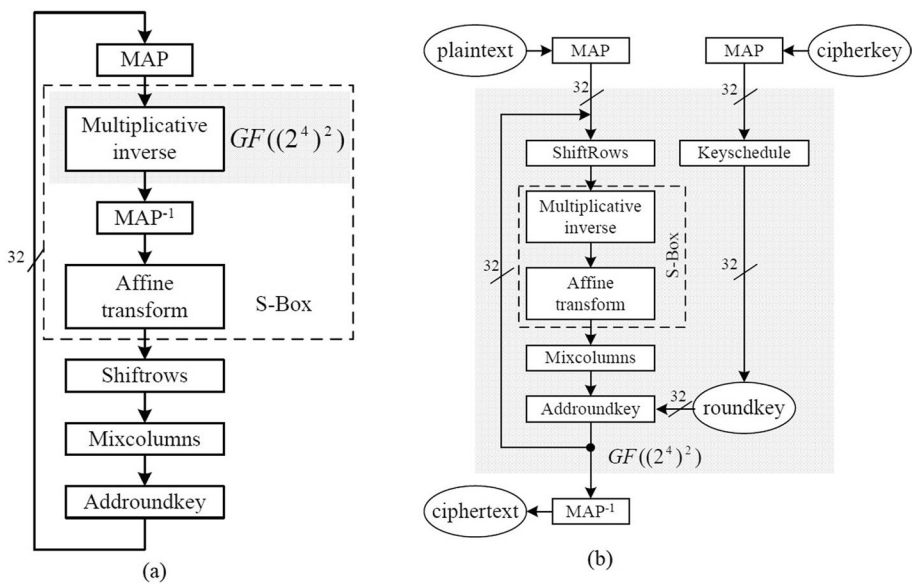

Fig. 4 **a** Partial in composite field **b** Complete in composite field

$$n(x) = x^2 + \{1\}x + \{e\}, \{e\} \text{ denotes } \{1110\} \tag{3}$$

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

The irreducible polynomial for the field GF(2^8) in AES is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{5}$$

The isomorphic mapping functions between field GF(2^8) and field GF((2^4)^2) are determined by the irreducible polynomials of field GF(2^8) (Eq. 5) and field GF((2^4)^2) (Eqs. 3 and 4). We use the following mapping formulas in [21] to convert the representations between GF(2^8) and GF((2^4)^2).

$$\begin{aligned}
&a_h x + a_l = \text{MAP}(a), a_h, a_l \in \text{GF}\left(2^4\right), a \in \text{GF}\left(2^8\right) \\
&a_A = a_1 \oplus a_7, a_B = a_5 \oplus a_7, a_C = a_4 \oplus a_6 \\
&a_{l0} = a_C \oplus a_0 \oplus a_5, a_{l1} = a_1 \oplus a_2, a_{l2} = a_A, a_{l3} = a_2 \oplus a_4 \\
&a_{h0} = a_C \oplus a_5, a_{h1} = a_A \oplus a_C, a_{h2} = a_B \oplus a_2 \oplus a_3, a_{h3} = a_B
\end{aligned} \tag{6}$$

In Eq. 6, a is an element in field GF(2^8). MAP(a) converts a to its isomorphic element in GF((2^4)^2), which is represented as $a_h x + a_l$.

$$\begin{aligned}
&a = \text{MAP}^{-1}(a_h x + a_l), a \in \text{GF}\left(2^8\right), a_h, a_l \in \text{GF}\left(2^4\right) \\
&a_A = a_{l1} \oplus a_{h3}, a_B = a_{h0} \oplus a_{h1} \\
&a_0 = a_{l0} \oplus a_{h0}, a_1 = a_B \oplus a_{h3}, a_2 = a_A \oplus a_B \\
&a_3 = a_B \oplus a_{l1} \oplus a_{h2}, a_4 = a_A \oplus a_B \oplus a_{l3}, a_5 = a_B \oplus a_{l2} \\
&a_6 = a_A \oplus a_{l2} \oplus a_{l3} \oplus a_{h0}, a_7 = a_B \oplus a_{l2} \oplus a_{h3}
\end{aligned} \tag{7}$$

In Eq. 7, $a_h x + a_l$ is an element in field GF((2^4)^2). MAP^{-1}($a_h x + a_l$) converts $a_h x + a_l$ to its isomorphic element in GF(2^8), which is represented as a.

3.3 Subpipelined encryptor/decryptor and keyschedule

Pipelining is applied in the designs to optimize speed/area ratio of AES. By inserting registers in combinational logic circuits, multiple blocks of hardware are running simultaneously. The frequency of the design is determined by the maximum delay between registers. We reduce the maximum delay and increase the frequency by optimizing the balance between stages. A 32-bit single-round subpipelined architecture in full composite field is proposed where one round unit is implemented and subpipelined into eight substages. To generate the roundkeys synchronously, we present an on-the-fly keyschedule. The encryption/decryption unit and the key expansion unit share the same clock which leads to the fact that the clock frequency is determined by the maximum delay in both units. This makes the balance of substage in keyschedule as important as in encryptor/decryptor. We propose a new subpipelined keyschedule on composite field for all standard key sizes. The most costly part of keyschedule is still S-Box. We divide it into the same substages as in encryptor/decryptor.

3.4 Double-block subpipelined architecture

The proposed architecture for encryptor is illustrated in Fig. 5. The decryption can be easily implemented by the equivalent cipher [1]. The eight 32-bit registers (four in ShiftRows, three in Subbytes and one between Subbytes and MixColumns) are used to cut one round unit into eight substages, which leads to an eight clock cycles initial delay to generate the first 32-bit ciphertext. *clk counter* is a clock register counter generated in keyschedule. It is used to synchronize encryptor/decryptor and keyschedule. We use a double-block (block A and B) data flow in our subpipelined architecture.

Figure 5a illustrates the subpipelining in ShiftRows operation, and Fig. 5b shows the subpipelining in Subbytes operation. We can see that the mappings from $GF(2^8)$ to $GF((2^4)^2)$ are only required once after the inputs of plaintext and cipherkey. The inverse mapping ($GF((2^4)^2)$ to $GF(2^8)$) is applied to the final output in order to get the cipher text.

The 3-to-1 multiplexer ("mul") is controlled by the clk counter:

- *Case a* In initial round, where $0 \leq$ clk counter < 8, 128-bit plaintext is MAPed into $GF((2^4)^2)$ and XORed with the according roundkey in four clock cycles, 32 bits at

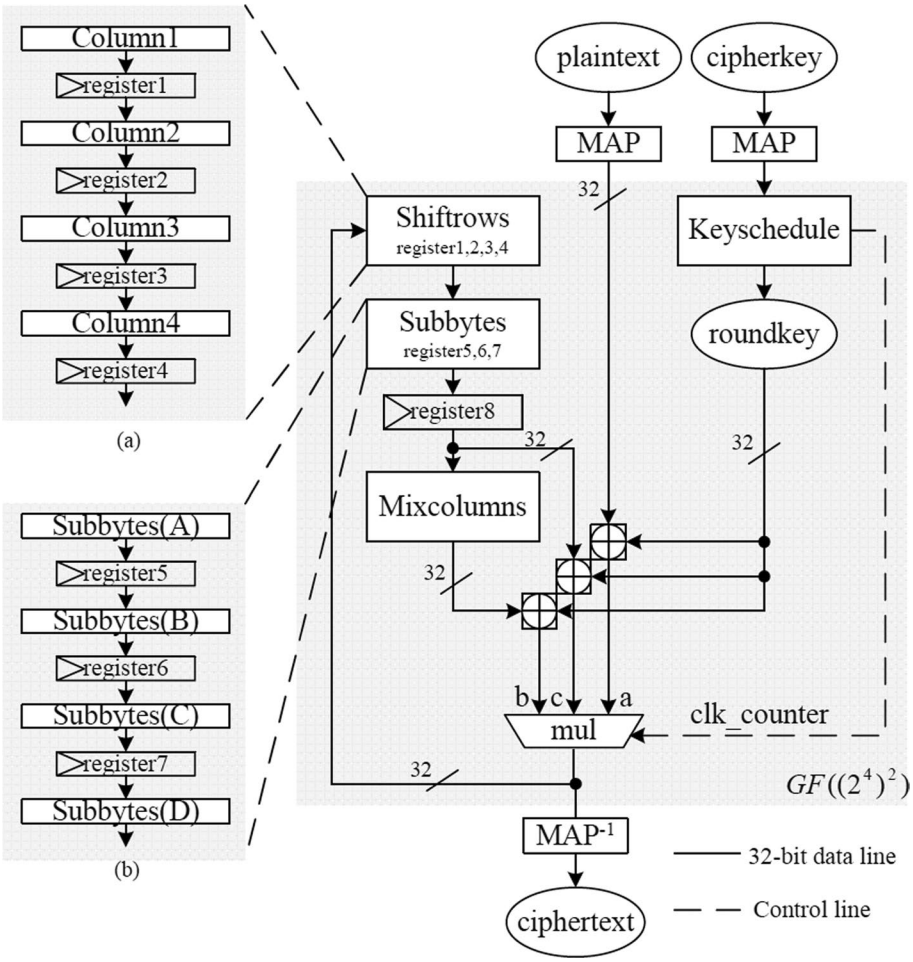

Fig. 5 AES encryption architecture

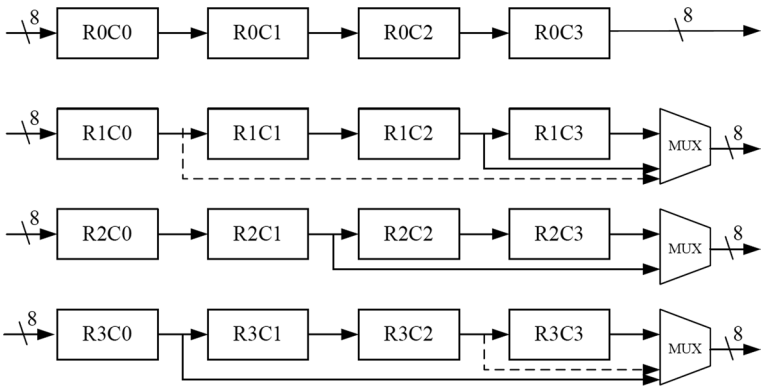

Fig. 6 ShiftRows operations in encryption and decryption

each clock. The result is the outcome of the initial round ($Nr = 0$) which is the input of the second round;

- *Case b* In normal rounds, where $8 \leq \text{clk counter} < Nr \times 8$, the output of MixColumns XORs with the corresponding roundkey.
- *Case c* In the last round, where $Nr \times 8 \leq \text{clk counter} < (Nr + 1) \times 8$. The output of Subbytes XORs with the corresponding roundkey. The ciphertext is obtained.

The detail operations of the ShiftRows, Subbytes and MixColumns are presented in the following.

3.4.1 ShiftRows

We use our proposed ShiftRows operation [22] in the design. It includes sixteen 8-bit registers and three 2-to-1 multiplexers. The block of data is shifted column by column. Two blocks of data are processed in the pipeline.

Our ShiftRows operation is designed in a column fashion (Fig. 6). In the architecture, the data (32-bits) in the columns are shifted in the order of column instead of rows. Each column is composed of four shift registers, and each register has 8 bits. By transforming the ShiftRows operation to a column fashion operation, we can make the design of Mixcolumns operation easier, since all the data in one column are required in the MixColumn operation.

The following are the ShiftRows procedure for encryption.

(1) *First row* No shift. We just let the data flow through.
(2) *Second row* Circular left shift operation. In this case, we connect the output of register R1C2 and the output of R1C3 to a multiplexer in order to select the output.
(3) *Third row* Switch data. Switch the data between first element and third element, second element and fourth element in the row. The outputs of R2C1 and R2C3 are connected to a Multiplexer.
(4) *Fourth row* Circular right shift operation. Similar to the case of second row, we connect the output of register R3C0 and the output of R3C3 to a Multiplexer.

Similarly, we can derive the procedures for Inverse ShiftRows (Inv-ShiftRows) operations:

(1) *First row* No shift.
(2) *Second row* Circular right shift operation. We connect the output of register R1C0 and the output of R1C3 to a Multiplexer.
(3) *Third row* Switch Data. Same as the operation in ShiftRows of encryption.
(4) *Fourth row* Circular left shift operation. We connect the output of register R3C2 and the output of R3C3 to a multiplexer in order to select the output.

The multiplexers are controlled by some clock counters and the encryption/decryption signals.

3.4.2 Subpipelined Subbytes

The key step of Subbytes is the calculation of the multiplicative inverse. Figure 7 illustrates the architecture of Subbytes proposed in [8]. It uses multiplication in $GF(2^4)^2$ three times. It also needs one inversion (x^{-1}), one constant multiplier with $\{e\}$ ($\times e$, $\{e\}$ is in hexadecimal notation, which is '1110' in binary notation), one squarer and two 4-bit XORs ($\oplus$).

We proposed a 32-bit subpipelined compact s-box architecture in composite field of $GF(2^4)^2$ with balanced substages and efficient performance [23]. Considering x, y, $z \in GF(2^4)$, x, y and z are represented in binary notation where $x = \{x_3x_2x_1x_0\}, y = \{y_3y_2y_1y_0\}, z = \{z_3z_2z_1z_0\}$. Let a, b, c, d, e and f be *1-bit* values, which equal to *0* or *1*. $\oplus$ stands for XOR-operation. x_0y_1 means $x_0 \wedge y_1$. Equations 8, 9, 10 and 11 [21] are used to calculate squaring, constant multiplication with $\{e\}$, multiplication and multiplicative inverse.

$$\boldsymbol{y} = \boldsymbol{x}^2 : y_0 = x_0 \oplus x_2, y_1 = x_2, y_2 = x_1 \oplus x_3, y_3 = x_3 \tag{8}$$

$$\boldsymbol{y} = \boldsymbol{x} \times \{\boldsymbol{e}\} : a = x_0 \oplus x_1, b = x_2 \oplus x_3, y_0 = x1 \oplus b, y_1 = a, y_2 = a \oplus x_2, y_3 = a \oplus b \tag{9}$$

$$\begin{aligned} &\boldsymbol{z} = \boldsymbol{x} \times \boldsymbol{y} : a = x_0 \oplus x_3, b = x_2 \oplus x_3, c = x_1 \oplus x_2 \\ &z_0 = (x_0y_0) \oplus (x_3y_1) \oplus (x_2y_2) \oplus (x_1y_3), z_1 = (x_1y_0) \oplus (ay_1) \oplus (by_2) \oplus (cy_3) \\ &z_2 = (x_2y_0) \oplus (x_1y_1) \oplus (ay_2) \oplus (by_3), z_3 = (x_3y_0) \oplus (x_2y_1) \oplus (x_1y_2) \oplus (ay_3) \end{aligned} \tag{10}$$

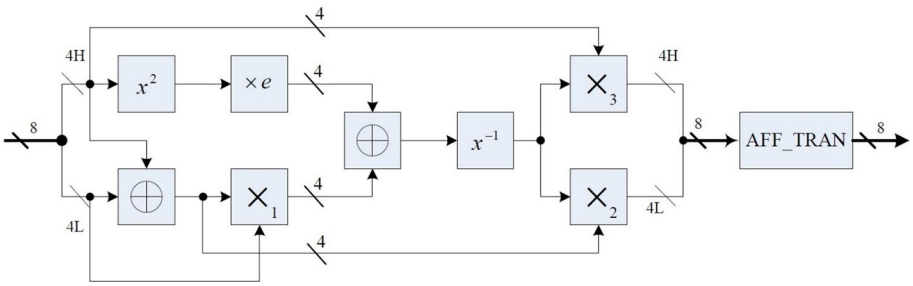

Fig. 7 Subbytes in composite field $GF((2^4)^2)$ [8]

$$
\begin{aligned}
&\boldsymbol{y} = \boldsymbol{x}^{-1} : a = x_1 \oplus x_2 \oplus x_3 \oplus (x_1x_2x_3), y_0 = a \oplus x_0 \oplus (x_0x_2) \oplus (x_1x_2) \oplus (x_0x_1x_2)\\
&y_1 = (x_0x_1) \oplus (x_0x_2) \oplus (x_1x_2) \oplus x_3 \oplus (x_1x_3) \oplus (x_0x_1x_3)\\
&y_2 = (x_0x_1) \oplus x_2 \oplus (x_0x_2) \oplus x_3 \oplus (x_0x_3) \oplus (x_0x_2x_3)\\
&y_3 = a \oplus (x_0x_3) \oplus (x_1x_3) \oplus x_2x_3
\end{aligned} \tag{11}
$$

In our design which is illustrated in Fig. 5, Subbytes should be cut into four substages. The key to an efficient subpipelining technology is to balance the delays of these substages.

We derive a new Eq. 12 from Eq. 11 to reduce the delay caused by x^{-1}.

Equation 12 is derived in three steps:

(1) In Eq. 11, replace "a" by its expression:

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

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

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

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

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

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

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

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

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

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

$$
\begin{aligned}
&y_0 = a \oplus b \oplus d \oplus ((x_0 \oplus x_3)e)\\
&y_1 = a \oplus f \oplus x_3(1 \oplus x_1 \oplus c)\\
&y_2 = f \oplus x_2 \oplus x_3(1 \oplus x_0 \oplus b)\\
&y_3 = d \oplus x_3(e \oplus x_0 \oplus d)
\end{aligned} \tag{12}
$$

According to Eq. 12, we design the logic circuit illustrated in Fig. 8 to perform x^{-1} over GF$(2^4)^2$. Besides multiplicative inversion, other operations in Fig. 7 are the three multiplications ($\times 1, \times 2$ and $\times 3$). In order to decrease the maximum delay caused by multiplication, we separate each multiplication into two steps and put each step in different substages. The registers between each substage store the result of the first step of multiplication and pass it to the second step. We decompose these three multipliers into two different manners (*AB-type* and *MN-type*) to achieve the best balance.

AB-type The *AB-type* multiplication is based on Eq. 13 which is derived from Eq. 10. *Step A* calculates the value of all the binomials; *Step B* conducts XOR of every four

values to generate z0, z1, z2 and z3. A register is inserted between Step A and Step B to store $p_0, p_1, \ldots, p_{15}$. The multiplication " $\times_1$" in Fig. 7 is separated as $\times_{1A}$ and $\times_{1B}$ in Fig. 8;

$$z = x \times y(AB - \text{type})$$

Step A:

$$a = x_0 \oplus x_3, b = x_2 \oplus x_3, c = x_1 \oplus x_2, p_0 = x_0y_0, p_1 = x_3y_1, p_2 = x_2y_2, p_3 = x_1y_3$$

$$p_4 = x_1y_0, p_5 = ay_1, p_6 = by_2, p_7 = cy_3, p_8 = x_2y_0, p_9 = x_1y_1, p_{10} = ay_2, p_{11} = by_3$$

$$p_{12} = x_3y_0, p_{13} = x_2y_1, p_{14} = x_1y_2, p_{15} = ay_3$$

Step B:

$$\begin{aligned} z_0 &= p_0 \oplus p_1 \oplus p_2 \oplus p_3, z_1 = p_4 \oplus p_5 \oplus p_6 \oplus p_7 \\ z_2 &= p_8 \oplus p_9 \oplus p_{10} \oplus p_{11}, z_3 = p_{12} \oplus p_{13} \oplus p_{14} \oplus p_{15} \end{aligned} \tag{13}$$

MN-type The *MN-type* multiplication is based on Eq. 14 which is also derived from Eq. 10. *Step M* creates the value of *a, b* and *c*; *Step N* implements the rest of Eq. 10. A register is inserted between *Step M* and *Step N* to store *a, b, c*. The multiplications of " $\times_2$" and " $\times_3$" in Fig. 7 are separated as $\times_{2M}$ and $\times_{2N}$, $\times_{3M}$ and $\times_{3N}$ in Fig. 8.

$$z = x \times y\big(MN - type\big)$$

Step M:

$$a = x_0 \oplus x_3, b = x_2 \oplus x_3, c = x_1 \oplus x_2$$

Step N:

$$\begin{aligned} z_0 &= x_0y_0 \oplus x_3y_1 \oplus x_2y_2 \oplus x_1y_3, z_1 = x_1y_0 \oplus ay_1 \oplus by_2 \oplus cy_3 \\ z_2 &= x_2y_0 \oplus x_1y_1 \oplus ay_2 \oplus by_3, z_3 = x_3y_0 \oplus x_2y_1 \oplus x_1y_2 \oplus ay_3 \end{aligned} \tag{14}$$

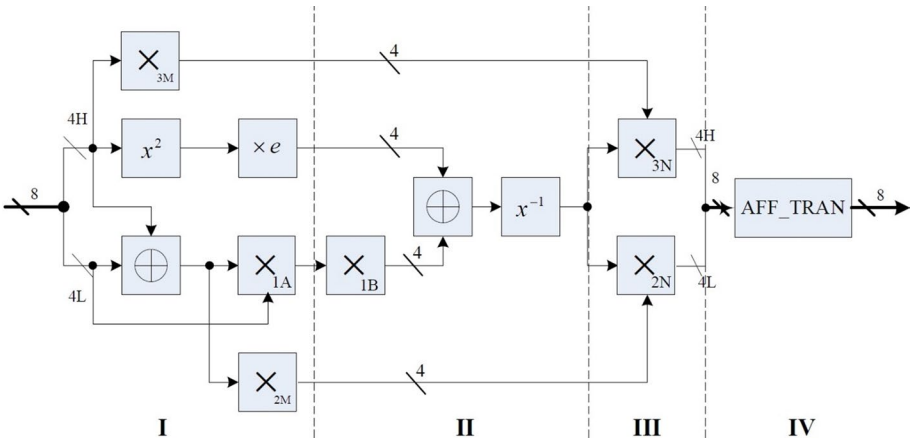

Fig. 8 Pipelined Subbytes in composite field GF($(2^4)^2$)

The last operation in Subbytes is the affine transformation. We derive *Eq. 21* to do the affine transformation in $GF(2^4)^2$ based on Eqs. 1,6 and 7.

Consider $p \in \mathrm{GF}(2^4)^2,\ q \in \mathrm{GF}(2^8) : p = \{p_7p_6p_5p_4p_3p_2p_1p_0\}, q = \{q_7q_6q_5q_4q_3q_2q_1q_0\}$

For Eq. 6:

(1) Replace a_A, a_B, a_C with their expressions:

$$a_{l0} = a_4 \oplus a_6 \oplus a_0 \oplus a_5, a_{l1} = a_1 \oplus a_2, a_{l2} = a_1 \oplus a_7, a_{l3} = a_2 \oplus a_4$$

$$a_{h0} = a_4 \oplus a_6 \oplus a_5, a_{h1} = a_1 \oplus a_7 \oplus a_4 \oplus a_6, a_{h2} = a_5 \oplus a_7 \oplus a_2 \oplus a_3, a_{h3} = a_5 \oplus a_7$$

(2) Let p replace $a_h x + a_l$, q replace a, we derive Eq. 15:

$$\begin{aligned} &\boldsymbol{p = MAP(q), p \in GF\left(2^4\right)^2, q \in GF\left(2^8\right)} \\ &p_0 = q_0 \oplus q_4 \oplus q_5 \oplus q_6, p_1 = q_1 \oplus q_2, p_2 = q_1 \oplus q_7, p_3 = q_2 \oplus q_4 \\ &p_4 = q_4 \oplus q_5 \oplus q_6, p_5 = q_1 \oplus q_4 \oplus q_6 \oplus q_7, p_6 = q_2 \oplus q_3 \oplus q_5 \oplus q_7, p_7 = q_5 \oplus q_7 \end{aligned} \tag{15}$$

Similar steps are applied in Eq. 7. Equation 16 is derived:

$$\begin{aligned} &\boldsymbol{q = MAP^{-1}(p), p \in GF\left(2^4\right)^2, q \in GF\left(2^8\right)} \\ &q_0 = p_0 \oplus p_4, q_1 = p_4 \oplus p_5 \oplus p_7, q_2 = p_1 \oplus p_4 \oplus p_5 \oplus p_7, q_3 = p_1 \oplus p_4 \oplus p_5 \oplus p_6 \\ &q_4 = p_1 \oplus p_3 \oplus p_4 \oplus p_5 \oplus p_7, q_5 = p_2 \oplus p_4 \oplus p_5, q_6 = p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_7, \\ &q_7 = p_2 \oplus p_4 \oplus p_5 \oplus p_7 \end{aligned} \tag{16}$$

In the following, we derive Eq. 21 based on Eqs. 1, 15 and 16.

Let x', y be the element in $\mathrm{GF}(2^8)$.: $x' = \{x'_7x'_6x'_5x'_4x'_3x'_2x'_1x'_0\}, y = \{y_7y_6y_5y_4y_3y_2y_1y_0\}$

.

According to Eq. 1:

$$\begin{aligned} &y_0 = x_0^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus x_7^{'} \oplus 1, y_1 = x_0^{'} \oplus x_1^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus x_7^{'} \oplus 1 \\ &y_2 = x_0^{'} \oplus x_1^{'} \oplus x_2^{'} \oplus x_6^{'} \oplus x_7^{'}, y_3 = x_0^{'} \oplus x_1^{'} \oplus x_2^{'} \oplus x_3^{'} \oplus x_7^{'} \\ &y_4 = x_0^{'} \oplus x_1^{'} \oplus x_2^{'} \oplus x_3^{'} \oplus x_4^{'}, y_5 = x_1^{'} \oplus x_2^{'} \oplus x_3^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus 1 \\ &y_6 = x_2^{'} \oplus x_3^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus 1, y_7 = x_3^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus x_7^{'} \end{aligned} \tag{17}$$

We convert y to $\mathrm{GF}(2^4)^2$ and also represent x' in $\mathrm{GF}(2^4)^2$ to derive the affine transformation in $\mathrm{GF}(2^4)^2$.

(1) Let w represent y in $GF(2^4)^2$. By *Eq. 15* (map from $GF(2^8)$ to $GF(2^4)^2$):

$$\begin{aligned} &w_0 = y_0 \oplus y_4 \oplus y_5 \oplus y_6, w_1 = y_1 \oplus y_2, w_2 = y_1 \oplus y_7, w_3 = y_2 \oplus y_4, w_4 = y_4 \oplus y_5 \oplus y_6 \\ &w_5 = y_1 \oplus y_4 \oplus y_6 \oplus y_7, w_6 = y_2 \oplus y_3 \oplus y_5 \oplus y_7, w_7 = y_5 \oplus y_7 \end{aligned} \tag{18}$$

(2) Let z be the $GF(2^4)^2$ format of x'. From Eq. 16:

$$x_0' = z_0 \oplus z_4, x_1' = z_4 \oplus z_5 \oplus z_7, x_2' = z_1 \oplus z_4 \oplus z_5 \oplus z_7, x_3' = z_1 \oplus z_4 \oplus z_5 \oplus z_6$$
$$x_4' = z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7, x_5' = z_2 \oplus z_4 \oplus z_5, x_6' = z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_7,$$
$$x_7' = z_2 \oplus z_4 \oplus z_5 \oplus z_7 \tag{19}$$

(3) Replace y in Eq. 18 with x' in Eq. 17 and replace x' with its $GF\left(2^4\right)^2$ format z:

$$\begin{aligned} w_0 &= y_0 \oplus y_4 \oplus y_5 \oplus y_6 \\ &= (x_0^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus x_7^{'} \oplus 1) \oplus (x_0^{'} \oplus x_1^{'} \oplus x_2^{'} \oplus x_3^{'} \oplus x_4^{'}) \\ &\quad \oplus \left(x_1^{'} \oplus x_2^{'} \oplus x_3^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus 1\right) \\ &\quad \oplus \left(x_2^{'} \oplus x_3^{'} \oplus x_4^{'} \oplus x_5^{'} \oplus x_6^{'} \oplus 1\right) \ \text{(by Equation 17)} \\ &= x_2^{'} \oplus x_3^{'} \oplus x_5^{'} \oplus x_7^{'} \oplus 1 = (z_1 \oplus z_4 \oplus z_5 \oplus z_7) \\ &\quad \oplus (z_1 \oplus z_4 \oplus z_5 \oplus z_6) \oplus (z_2 \oplus z_4 \oplus z_5) \\ &\quad \oplus (z_2 \oplus z_4 \oplus z_5 \oplus z_7) \oplus 1 \ \text{(by Equation 19)} \\ &= z_6 \oplus 1 = (z_6)' \end{aligned}$$

Similarly, we can get:

$$w_1 = (z_1 \oplus z_2 \oplus z_7)', w_2 = (z_0 \oplus z_5 \oplus z_6 \oplus z_3)', w_3 = z_1 \oplus z_5 \oplus z_6 \oplus z_7$$
$$w_4 = z_0 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7, w_5 = z_1 \oplus z_5 \oplus z_6, w_6 = (z_2 \oplus z_6 \oplus z_7)'$$
$$w_7 = (z_3 \oplus z_5)' \tag{20}$$

(4) For the consistency of the other equations in this paper, we replace w by y, z by x (x,$y \in GF\left(2^4\right)^2$) in Eq. 20 and let $a = x_5 \oplus x_6 \oplus x_7$, we derive

$$\mathbf{y} = \mathbf{AFF\_TRAN}(\mathbf{x}):$$
$$a = x_5 \oplus x_6 \oplus x_7$$
$$y_0 = (x_6)', y_1 = (x_1 \oplus x_2 \oplus x_7)', y_2 = (x_0 \oplus x_3 \oplus x_5 \oplus x_6)', y_3 = x_1 \oplus a$$
$$y_4 = x_0 \oplus x_2 \oplus x_4 \oplus a, y_5 = x_1 \oplus x_5 \oplus x_6, y_6 = (x_2 \oplus x_6 \oplus x_7)', y_7 = (x_3 \oplus x_5)' \tag{21}$$

Figure 8 describes the proposed subpipelined architecture of Subbytes in GF($(2^4)^2$). The dashed lines stand for the registers.

We cut an AES round unit into 8 substages with the maximum delay determined by part II (Fig. 8) in Subbytes. The inverse S-box can use the same multiplicative inverse in encryption except that the inverse affine transformation is applied before the multiplicative inverse. We also derive the following formula for the inverse affine transformation in GF$(2^4)^2$:

$$\boldsymbol{y = Inv\_AFF\_TRAN}(\boldsymbol{x})$$
$$a = x_1 \oplus x_5 \oplus x_6, b = x_0 \oplus x_2 \oplus x_7$$
$$y_0 = b', y_1 = (x_0 \oplus x_1 \oplus x_6)', y_2 = x_0 \oplus x_3 \oplus x_5 \oplus x_6, y_3 = (x_7 \oplus a)' \tag{22}$$
$$y_4 = x_1 \oplus x_4 \oplus x_5 \oplus b, y_5 = a, y_6 = x_0^{'}, y_7 = x_3 \oplus x_5$$

Figure 9 illustrates the design of S-box in encryption and decryption. It can process eight bits input in GF$(2^4)^2$. Four units are required to process the 32-bit data path.

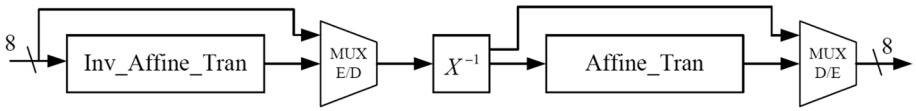

Fig. 9 S-Box operation in composite field GF($(2^4)^2$) for encryption and decryption

3.4.3 MixColumns on GF($(2^4)^2$)

MixColumns are another transformation which involves mathematical operations in GF($(2^4)^2$). We derive the following formulas to perform MixColumns in composite field.

Since GF($(2^4)^2$) is an isomorphic field to GF(2^8), and {02}, {03}, {01} in GF(2^8) are mapped to {26}, {27}, {01}, respectively, in GF($(2^4)^2$), the MixColumns operation described by Eq. 2 can be mapped directly to Eq. 23.

$$\begin{bmatrix} 26 & 27 & 01 & 01 \\ 01 & 26 & 27 & 01 \\ 01 & 01 & 26 & 27 \\ 27 & 01 & 01 & 26 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{23}$$

Observing that in GF($(2^4)^2$), $\{27\} = \{26\} \oplus \{01\}$, Eq. 23 is equal to Eq. 24, where $j = 0, 1, 2, 3$:

$$\begin{aligned} S'_{0,j} &= \{26\} \times (S_{0,j} + S_{1,j}) + S_{1,j} + S_{2,j} + S_{3,j} \\ S'_{1,j} &= \{26\} \times (S_{1,j} + S_{2,j}) + S_{0,j} + S_{2,j} + S_{3,j} \\ S'_{2,j} &= \{26\} \times (S_{2,j} + S_{3,j}) + S_{0,j} + S_{1,j} + S_{3,j} \\ S'_{3,j} &= \{26\} \times (S_{0,j} + S_{3,j}) + S_{0,j} + S_{1,j} + S_{2,j} \end{aligned} \tag{24}$$

Equation 24 presents the MixColumn transformation of one column of a state. The MixColumn transformation can be implemented by the parallel structure in Fig. 10.

In the following, we derive Eq. 28 to calculate $x \times 26$ in GF($(2^4)^2$). That is, we represent the results of x × {02} in GF($(2^4)^2$).

(1) Let $x, y \in GF(2^8)$, $y = x \times \{02\}$:

$$y_0 = x_7, y_1 = x_0 \oplus x_7, y_2 = x_1, y_3 = x_2 \oplus x_7, y_4 = x_3 \oplus x_7, y_5 = x_4, y_6 = x_5, y_7 = x_6 \tag{25}$$

(2) Convert y to the element in GF($(2^4)^2$). Let w represent y in GF($(2^4)^2$), that is Eq. 18.

(3) Let z be the $GF(2^4)^2$ format of x:

$$\begin{aligned} &x_0 = z_0 \oplus z_4, x_1 = z_4 \oplus z_5 \oplus z_7, x_2 = z_1 \oplus z_4 \oplus z_5 \oplus z_7, x_3 = z_1 \oplus z_4 \oplus z_5 \oplus z_6 \\ &x_4 = z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7, x_5 = z_2 \oplus z_4 \oplus z_5, x_6 = z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_7, \\ &x_7 = z_2 \oplus z_4 \oplus z_5 \oplus z_7 \end{aligned} \tag{26}$$

(4) Replace x and y with their corresponding GF($(2^4)^2$) format z and w:

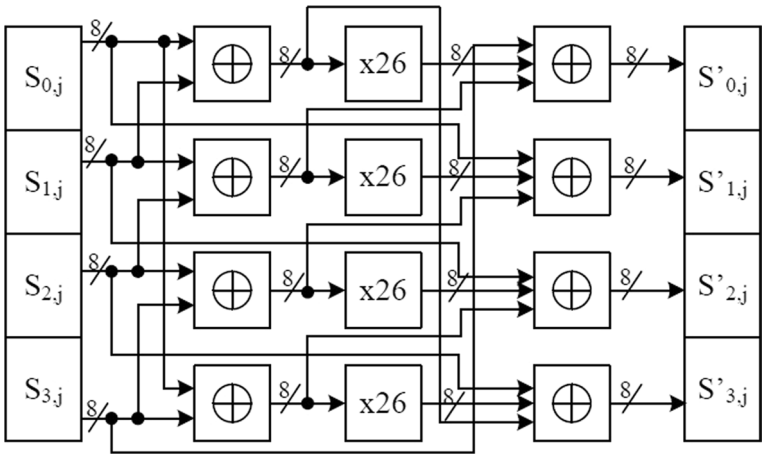

Fig. 10 MixColumns operation in composite field GF($(2^4)^2$)

$$
\begin{aligned}
w_0 &= y_0 \oplus y_4 \oplus y_5 \oplus y_6 \left(\text{by Equation 18}\right) \\
&= x_7 \oplus (x_3 \oplus x_7) \oplus x_4 \oplus x_5 \left(\text{by Equation 25}\right) \\
&= x_3 \oplus x_4 \oplus x_5 \\
&= (z_1 \oplus z_4 \oplus z_5 \oplus z_6) \oplus (z_1 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_7) \oplus (z_2 \oplus z_4 \oplus z_5) \left(\text{by Equation 26}\right) \\
&= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7
\end{aligned}
$$

Through the same procedures, we can derive:

$$
\begin{aligned}
&w_0 = z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7, w_1 = z_0 \oplus z_2 \oplus z_4, w_2 = z_0 \oplus z_1 \oplus z_3 \oplus z_4 \oplus z_5 \\
&w_3 = z_1 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_6, w_4 = z_3 \oplus z_6, w_5 = z_0 \oplus z_3 \oplus z_6 \oplus z_7 \\
&w_6 = z_1 \oplus z_4 \oplus z_7, w_7 = z_2 \oplus z_5
\end{aligned} \tag{27}
$$

(5) For consistency, replace z with x, and replace w with y ($x,y \in GF\left(2^4\right)^2$):

$$
\begin{aligned}
&\boldsymbol{y} = \boldsymbol{x} \times 26,\ \boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{GF}\left(2^4\right)^2 \\
&a = x_2 \oplus x_4, b = x_3 \oplus x_6 \oplus x_7, c = x_1 \oplus x_5 \\
&y_0 = a \oplus b \oplus x_5, y_1 = a \oplus x_0, y_2 = c \oplus x_0 \oplus x_3 \oplus x_4, y_3 = c \oplus a \oplus x_6 \\
&y_4 = x_3 \oplus x_6, y_5 = b \oplus x_0, y_6 = x_1 \oplus x_4 \oplus x_7, y_7 = x_2 \oplus x_5
\end{aligned} \tag{28}
$$

In this design for both encryption and decryption, we will modify the MixColumn and InvMixColumn architecture proposed by Fischer et al. [24]. We need to map the previous architecture from GF(2^8) to GF($(2^4)^2$). It can be seen that we only need to modify the "xtime" operation. That is, to calculate "xtimes" in GF($(2^4)^2$).

3.4.4 Subpipelined keyschedule

There are two approaches to implement keyschedule: (1) pre-calculated keyschedule and (2) on-the-fly keyschedule. In the pre-calculated keyschedule, the ($Nr+1$) 128-bit roundkeys are generated before the encryption or decryption begins and stored in the

memory. The addroundkey operation accesses the roundkeys by referring to the corresponding address in the memory. The advantage of this approach is that the keyschedule only needs to be performed once; however, the drawbacks include:

(i) The $(Nr+1)$ roundkeys cost $(Nr+1)\times 128$ bits memory space;
(ii) The cipherkey should not change frequently. Every time it changes, the roundkeys must be recalculated.

In this paper, we propose a new 32-bit pipelined on-the-fly keyschedule in fully composite field ($GF((2^4)^2)$) with 128-, 192-, 256-bit key sizes, where each 128-bit roundkey is generated at every four clock cycles (32-bit at each clock). The following shows the 32-bit roundkeys at each clock cycle (KA(i), and KB(i) represent the round keys for block A and block B, each is 32-bit, $0 \le i \le 4Nr+3$).

The roundkeys for block A:
roundkey[0]={KA(0), KA(1), KA(2), KA(3)}
roundkey[1]={KA(4), KA(5), KA(6), KA(7)}
……
roundkey[Nr]={KA(4Nr), KA(4Nr+1), KA(4Nr+2), KA(4Nr+3)}
The roundkeys for block B:
roundkey[0]={KB(0), KB(1), KB(2), KB(3)}
roundkey[1]={KB(4), KB(5), KB(6), KB(7)}
……
roundkey[Nr]={KB(4Nr), KB(4Nr+1), KB(4Nr+2), KB(4Nr+3)}

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

All mathematic operations in keyschedule are transformed into field $GF((2^4)^2)$. Subword shares the same structure as in Subbytes. Xorrcon is a simple XOR operation with a round constant, which is initially {01} and multiplied by {02} at each keyschedule round. Keyschedule round is defined as follows. It begins when *clk counter* = 0. If key size is 128 bit, keyschedule round cycle is four; if key size is 192 bit, keyschedule round cycle is six; if key size is 256 bit, keyschedule round cycle is eight. We know that in $GF((2^4)^2)$, {01} is still {01} and {02} is mapped to {26}. We can use *Eq. 28* to generate round constant for each keyschedule round.

The proposed keyschedule has three key size options: Key128, Key192 and Key256. The notation of *roundkey32* stands for 32-bit roundkey for each clock cycle, *roundkey* stands for 128-bit roundkey for a round of AES.

For decryption, the roundkey32 must be created in the reverse order. The last Nk roundkey32 from encryption is stored in a 256-bit register to be used as the initial decipherkey roundkey32 for decryption. For a given cipherkey, at least one encryption operation must be performed in order to store the final Nk roundkey32 for use during decryption. Multiplexers are then used to select between the cipherkey

and decipherkey, based on encryption or decryption mode, respectively. Since the decipherkey roundkey32 is already in GF($(2^4)^2$), they do not pass through the MAP operation.

Figure 11 illustrates the keyschedule architecture. The multiplexers *mul1* and *mul2* are used to reconfigure the pipeline for each of the three key sizes.

SA, SB, SC and SD are the four sections of subword operation with interspersed registers. RW is the outcome of rotword. RC generates the round constant for xorrcon in GF($(2^4)^2$). Multiplexor *mul3* is used to select the correct previous roundkey32 as input to the subword operation. Multiplexor *mul4* selects the appropriate calculated result to serve as the next roundkey32.

Table 2 summarizes the reconfigurable control of the multiplexers to generate three key sizes ($\bullet$ represents that the multiplexer is enabled for the corresponding key size, and the numbers represent the input selections of the multiplexer depending on the corresponding clock cycles).

When key size is 128 bits, the encryptor round number is ten. Two blocks A and B need 22 roundkeys. In our design, the first step is to map (MAP) cipherkey from GF(2^8) to GF($(2^4)^2$). After that, it performs its isomorphic functions in GF($(2^4)^2$). The output of keyschedule is roundkey32s represented in GF($(2^4)^2$). They are the exact format required in encryption where the message blocks are represented in GF($(2^4)^2$). No inverse MAP is required in keyschedule. SA, SB, SC and SD are the four sections of subword operation. We place three registers among the four substages in subword. RW is the outcome of rotword. RC generates the round constant for xorrcon in GF($(2^4)^2$).

4 Implementation performance and comparison

Many studies of hardware AES implementations have been published. Table 3 summarizes the functions provided by different FPGA implementations.

We do not use BRAM in our design in order to make the architecture suitable for wireless and embedded devices. Our proposed architecture has been simulated and synthesized with Xilinx Synthesis Technology (XST) ISE 10, and implemented on a Xilinx

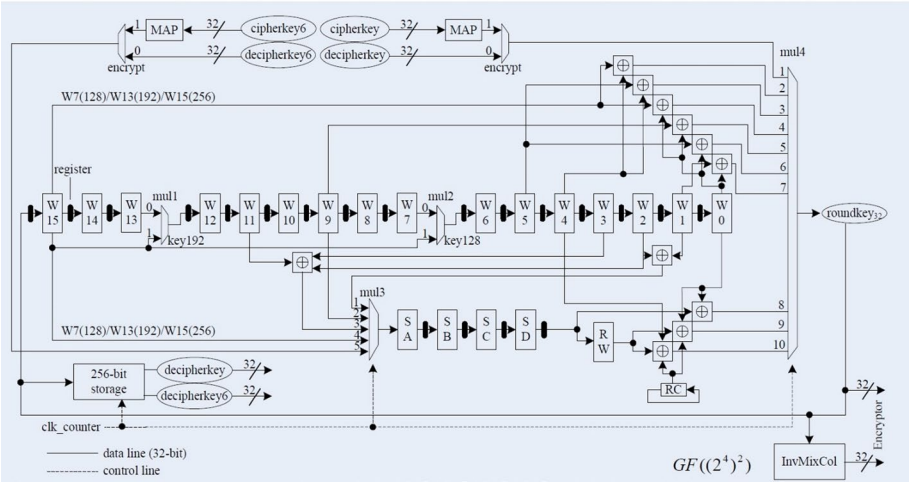

Fig. 11 Architecture of keyschedule-128/192/256

Virtex-4 device. From the synthesis result, we also optimize the delay time between different stages in our design to improve the performance. Table 4 illustrates the synthesis results with Virtex-4 XC4VSX25 and performance comparison.

Compared with the previous architectures, our design focuses on the low cost, non-BRAM implementations. Pramstaller et al. proposed a compact design costing 1125 slices in [5] with throughput of 215 Mbps for 128-bit, 180 Mbps for 192-bit, and 156 Mbps for 256-bit in the maximum frequency of 161 MHz. However, the round keys were pre-calculated by the key generator and RAM required to store those keys. We generate the round keys on-the-fly which is very useful and efficient when the key has been changed (AES is a symmetric-key cryptography, and the session key usually changes frequently.) In addition, our throughput increases greatly for each of the three key sizes. Furthermore, we propose a new subpipelined keyschedule which can support all three key sizes (128, 192, 256-bits). The time delays between the stages in encryption/decryption and keyschedule have been optimized in our architecture. We also present a new 32-bit complete composite field approach where the $GF((2^4)^2)$ field arithmetic applies in all transformations in encryptor/decryptor and keyschedule to save the cost of mapping between $GF(2^8)$ and $GF((2^4)^2)$ greatly. In addition, the 32-bit data path in our design can reduce the hardware cost greatly and can be efficiently applied in computing-resources restricted environments, such as wireless devices and embedded devices.

5 Conclusion

AES is an important and popular cryptographic algorithm to secure the information and data transmission. In this paper, we propose a compact reconfigurable FPGA architecture for the AES implementation. The 32-bit single-round unit design results in low area cost, which makes it suitable for low-end devices. The combinational logic approach of S-Box eliminates the need for BRAMs.

In our architecture, a fully $GF((2^4)^2)$ composite field arithmetic is applied in all transformations in encryption/decryption and keyschedule to save the cost of mapping greatly. That is, only one MAP and one MAP^{-1} are applied in encryption/decryption, and one MAP is applied in keyschedule. Full composite field-based design decreases hardware complexity of arithmetic operations in AES. In addition, we apply subpipelining technology in both encryptor/decryptor and keyschedule modules to optimize the speed/area ratio. The capability to deal with three key sizes makes our design an efficient reconfigurable architecture of AES. The performance comparison indicates that the proposed AES architecture achieves better performance than previous work.

Table 2 Summary of reconfigurable control for keyschedule-128/192/256

Key_Size options		mul1	mul2	mul3	mul4
Key_128	Encryption		• (1)	• (4)	• (1, 4, 9)
	Decryption		• (1)	• (1)	• (1,7, 9)
Key_192	Encryption	• (1)		• (3, 5)	• (1, 2, 4, 5, 9, 10)
	Decryption	• (1)		• (2)	• (1,3, 6, 7, 9, 10)
Key_256	Encryption			• (4)	• (1, 4, 8, 9)
	Decryption			• (2)	• (1,8, 9)

Table 3 Function comparisons of different AES architectures

Design	Encryption	Decryption	KeySchedule	Key Size	BRAM
Chodowiec et al. [4]	•	•	Pre-Calculate	128	•
Satoh et al. [12]	•	•	On-The-Fly	128	
Järvinen et al. [19]	•		On-The-Fly	128	
Good et al. [6]	•	•	On-The-Fly	128	
Zhang et al. [8]	•		On-The-Fly	128	
Chang et al. [20]	•	•	On-The-Fly	128	•
Pramstaller et al. [5]	•	•	Pre-Calculate	128/192/256	
McLoone et al. [17]	•	•	On-The-Fly	128/192/256	•
Bulens et al. [25]	•	•	Pre-Calculate	128	•
Our Design	•	•	On-The-Fly	128/192/256	

Table 4 Design synthesis results and performance comparison

Key Sizes	Frequency (MHz)	Area (Slices)	Clock cycles	Throughput (Mbps)	Throughput (Mbps) [5]
128 bit	129	1766	88	375	215
192 bit	129	1766	104	318	180
256 bit	129	1766	120	275	156

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

6 Future work

In the future, we will synthesize our FPGA prototype, optimize the design and implement it in VLSI. We believe the performance of the proposed architecture could be increased with current VLSI design tools and technology, and develop a new reconfigurable and efficient AES encryption/decryption chip which can be easily embedded into the wireless and computing-restricted devices to provide the security services.

Author contributions

KL and HL proposed the reconfigurable and compact AES architecture for encryption and decryption. GM implemented the design with Xilinx FPGA. All authors read and approved the final manuscript.

Declarations

Competing interests

There are no conflict/competing interests.

Received: 28 July 2022 Accepted: 14 December 2022
Published online: 09 January 2023

References

1. W. Stallings, *Cryptography and Network Security-Principles and Practices*, 4th edn. (Pearson Prentice hall, 2006)
2. NIST. Announcing the advanced encryption standard (AES). Available at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, 2001.

3. Daemen J, Rijmen V. AES proposal: Rijndael. Technical report, National Institute of Standards and Technology (NIST). Available at http://www.nic.funet.fi/pub/crypt/cryptography/symmetric/aes/nist/Rijndael.pdf, 2000.
4. P. Chodowiec, K. Gaj, Very compact FPGA implementation of the AES algorithm. Cryptogr Hardw Embed Syst CHES **2003**, 319–333 (2003)
5. N. Pramstaller, J. Wolkerstorfer, A universal and efficient AES co-processor for field programmable logic arrays, in *Field programmable logic and application*. ed. by J. Becker, M. Platzner, S. Vernalde (Springer Berlin Heidelberg, Berlin, Heidelberg, 2004), pp.565–574. https://doi.org/10.1007/978-3-540-30117-2_58
6. T. Good, M. Benaissa, AES on FPGA from the fastest to the smallest, in *Cryptographic hardware and embedded systems – CHES 2005*. ed. by J.R. Rao, B. Sunar (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005), pp.427–440. https://doi.org/10.1007/11545262_31
7. N. Pramstaller, S. Mangard, S. Dominikus, J. Wolkerstorfer, Efficient AES implementations on ASICs and FPGAs, in *Advanced encryption standard – AES*. ed. by H. Dobbertin, V. Rijmen, A. Sowa (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005), pp.98–112. https://doi.org/10.1007/11506447_9
8. X. Zhang, K.K. Parhi, High-speed VLSI architectures for the AES algorithm. IEEE Trans VLSI Syst **12**(9), 957–967 (2004)
9. Gaj K, Chodowiec P. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. In: AES candidate conference, pp. 40–54, 2000.
10. Liberatori M, Otero F, Bonadero JC, Castineira J. AES-128 Cipher. high speed, low cost FPGA implementation. In: 2007 3rd southern conference on programmable logic, pp. 195–198, 2007.
11. A. Rudra, P.K. Dubey, C.S. Jutla, V. Kumar, J.R. Rao, P. Rohatgi, Efficient rijndael encryption implementation with composite field arithmetic, in *Cryptographic hardware and embedded systems — CHES 2001*. ed. by Ç.K. Koç, D. Naccache, C. Paar (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001), pp.171–184. https://doi.org/10.1007/3-540-44709-1_16
12. A. Satoh, S. Morioka, K. Takano, S. Munetoh, A compact Rijndael hardware Architecture with S-Box Optimization, in *Advances in Cryptology*. ed. by C. Boyd (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001), pp.239–254. https://doi.org/10.1007/3-540-45682-1_15
13. W.-K. Lee, H.J. Seo, S.C. Seo, S.O. Hwang, Efficient implementation of AES-CTR and AES-ECB on GPUs with applications for high-speed FrodoKEM and exhaustive key search. IEEE Trans Circuits Syst II Express Briefs **69**(6), 2962–2966 (2022)
14. A.A. Pammu, W.-G. Ho, N.K.Z. Lwin, K.-S. Chong, B.-H. Gwee, A high throughput and secure authentication-encryption AES-CCM algorithm on asynchronous multicore processor. IEEE Trans Inf Forensics Secur **14**(4), 1023–1036 (2019)
15. Y. Zhou, G.-M. Tang, J.-H. Yang, P.-S. Yu, C. Peng, Logic design and simulation of a 128-b AES encryption accelerator based on rapid single-flux-quantum circuits. IEEE Trans Appl Supercond **31**(6), 1–11 (2021)
16. Hodjat A, Verbauwhede I. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In: 12th annual IEEE symposium on field-programmable custom computing machines, pp. 308–309, 2004.
17. McLoone M´, McCanny JV. High performance single-chip FPGA Rijndael algorithm implementations. In: Cryptographic hardware and embedded systems --- CHES 2001, pp. 65–76, 2001.
18. N. Yu, H.M. Heys, Investigation of compact hardware implementation of the advanced encryption standard. Can Conf Electr Comput Eng **2005**, 1069–1072 (2005)
19. Järvinen K, Tommiska M, Skyttä J. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In: ACM/SIGDA eleventh international symposium on field programmable gate arrays, pp. 207–215, 2003.
20. Chang C-J, Huang C-W, Tai H-Y, Lin M-Y. 8-bit AES implementation in FPGA by multiplexing 32-bit AES operation. In: The first international symposium on data, privacy, and E-commerce (ISDPE 2007), pp. 505–507, 2007.
21. J. Wolkerstorfer, E. Oswald, M. Lamberger, An ASIC implementation of the AES SBoxes, in *Topics in cryptology — CT-RSA 2002*. ed. by B. Preneel (Springer Berlin Heidelberg, Berlin, Heidelberg, 2002), pp.67–78. https://doi.org/10.1007/3-540-45760-7_6
22. H. Li, J. Li, A new compact architecture for AES with optimized ShiftRows operation. In: Proceedings of 2007 IEEE international symposium on circuits and systems, pp. 1851–1854, New Orleans, USA, May 27–30, 2007.
23. K. Li, H. Li, An efficient and compact subpipelined s-box architecture for AES. In: Proceedings of the ISCA 2nd international conference on advanced computing and communications, pp 45–49, Los Angeles, USA, 2012.
24. V. Fischer, M. Drutarovsky, P. Chodowiec, F. Gramain, InvMixColumn decomposition and multilevel resource sharing in AES implementations. IEEE Trans VLSI Syst **13**(8), 989–992 (2005). https://doi.org/10.1109/TVLSI.2005.853606
25. P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, G. Rouvroy, Implementation of the AES-128 on virtex-5 FPGAs, in *Progress in cryptology – AFRICACRYPT 2008*. ed. by S. Vaudenay (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008), pp.16–26. https://doi.org/10.1007/978-3-540-68164-9_2

Publisher's Note