

# High-Speed VLSI Architectures for the AES Algorithm

Xinmiao Zhang, *Student Member, IEEE*, and Keshab K. Parhi, *Fellow, IEEE*

**Abstract**—This paper presents novel high-speed architectures for the hardware implementation of the Advanced Encryption Standard (AES) algorithm. Unlike previous works which rely on look-up tables to implement the SubBytes and InvSubBytes transformations of the AES algorithm, the proposed design employs combinational logic only. As a direct consequence, the unbreakable delay incurred by look-up tables in the conventional approaches is eliminated, and the advantage of subpipelining can be further explored. Furthermore, composite field arithmetic is employed to reduce the area requirements, and different implementations for the inversion in subfield  $GF(2^4)$  are compared. In addition, an efficient key expansion architecture suitable for the subpipelined round units is also presented. Using the proposed architecture, a fully subpipelined encryptor with 7 substages in each round unit can achieve a throughput of 21.56 Gbps on a Xilinx XCV1000 e-8 bg560 device in non-feedback modes, which is faster and is 79% more efficient in terms of equivalent throughput/slice than the fastest previous FPGA implementation known to date.

**Index Terms**—Advanced Encryption Standard, composite field arithmetic, key expansion, look-up table, Rijndael, subpipelining, substructure sharing.

## I. INTRODUCTION

CRYPTOGRAPHY plays an important role in the security of data transmission. In January 1997, the National Institute of Standards and Technology (NIST) invited proposals for new algorithms for the Advanced Encryption Standard (AES) to replace the old Data Encryption Standard (DES). After two rounds of evaluation on the 15 candidate algorithms, NIST selected the Rijndael as the AES algorithm [1] in October 2000.

The AES algorithm has broad applications, including smart cards and cellular phones, WWW servers and automated teller machines (ATMs), and digital video recorders. Compared to software implementations, hardware implementations of the AES algorithm provide more physical security as well as higher speed. Three architectural optimization approaches can be employed to speed up the hardware implementations: pipelining, subpipelining, and loop-unrolling. Among these approaches, the subpipelined architecture can achieve maximum speedup and optimum speed–area ratio in non-feedback modes. In order to explore the advantage of subpipelining further, each round unit needs to be divided into more substages with equal delay. However, the SubBytes and the InvSubBytes in the AES algorithm are traditionally implemented by look-up tables (LUT)

[2]–[6]. In LUT-based approaches, it can be observed that the unbreakable delay of LUTs is longer than the total delay of the rest of the transformations in each round unit. This feature prohibits each round unit from being divided into more than two substages to achieve any further speedup.

Non-LUT-based approaches, which employ combinational logic only, can be used to avoid the unbreakable delay of LUTs. However, these approaches involve inversions in *Galois Field*  $GF(2^8)$ , which may have high hardware complexities. Composite field arithmetic can be employed, such that the field elements of  $GF(2^8)$  are mapped to elements in some isomorphic composite fields, in which the field operations can be implemented by lower cost subfield operations. Composite field implementations are exploited in [7], [8]. However, it is not efficient to implement all the transformations in the AES algorithm in composite fields. Meanwhile, the composite field arithmetic may not be the optimum approach when the order of the field involved is small. In addition, none of the prior composite field arithmetic approaches has addressed applying subpipelining to speed up the AES algorithm in non-feedback modes.

The main contributions of this paper can be summarized as follows. This paper avoids use of LUTs and proposes use of composite field data path for the SubBytes and InvSubBytes transformations. The use of such data paths is the key for the design of high-speed subpipelined AES architectures. Composite field arithmetic has been employed in [7] to design efficient data paths. However, the design in [7] decomposes the inversion in  $GF(2^8)$  in the SubBytes and InvSubBytes to  $GF(((2^2)^2)^2)$ . Instead, the proposed architecture in this paper decomposes the inversion to  $GF((2^4)^2)$ , and the inversion in  $GF(2^4)$  is implemented by a novel approach, which leads to a more efficient architecture with shorter critical path and smaller area. Composite field arithmetic was also used in [8]. Nevertheless, it is not efficient to use composite field arithmetic in all the transformations of the AES algorithm as done in [8]. Another main contribution of this paper is the key expansion architecture. This paper, for the first time, presents a key expansion architecture which is well suited for subpipelined designs. In addition, this architecture can operate in an on-the-fly manner. Using the proposed data path and key expansion architecture, post-placement timing report shows a fully subpipelined encryptor of 128-bit key with 7 substages in each round unit can operate at a throughput of 21.56 Gbps on a Xilinx XCV 1000 e-8bg560 device in non-feedback modes. Architectures utilizing multiple substage subpipelining have been published recently [9]–[11]. However, their designs are less efficient. The architecture presented in this paper can achieve higher speed than the prior fastest FPGA implementation [10], and is 79% more efficient in terms of equivalent throughput slice.

Manuscript received November 19, 2003; revised March 28, 2003. This work was supported by the Army Research Office under Grant DA/DAAD19-01-1-0705.

The authors are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: jennizh@ece.umn.edu; parhi@ece.umn.edu).

Digital Object Identifier 10.1109/TVLSI.2004.832943

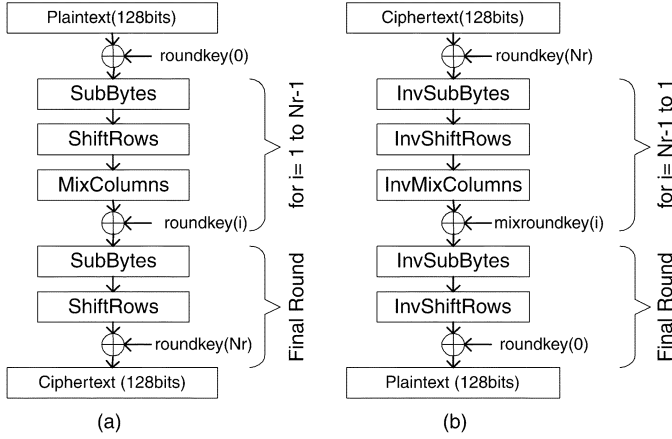


Fig. 1. The AES algorithm. (a) Encryption structure. (b) Equivalent decryption structure.

The structure of this paper is as follows. In Section II, we briefly describe the subpipelined architecture of the AES algorithm and composite field arithmetic. The detailed architecture for each transformation and the key expansion are presented in Section III. Section IV outlines the performance of the FPGA implementations and presents comparisons with prior efforts. Section V provides conclusions.

## II. THE SUBPIPELINED ARCHITECTURE OF THE AES AND COMPOSITE FIELD ARITHMETIC

### A. The AES Algorithm

The AES algorithm is a symmetric-key cipher, in which both the sender and the receiver use a single key for encryption and decryption. The data block length is fixed to be 128 bits, while the key length can be 128, 192, or 256 bits, respectively. In addition, the AES algorithm is an iterative algorithm. Each iteration can be called a round, and the total number of rounds,  $Nr$ , is 10, 12, or 14, when the key length is 128, 192, or 256 bits, respectively. The 128-bit data block is divided into 16 bytes. These bytes are mapped to a  $4 \times 4$  array called the State, and all the internal operations of the AES algorithm are performed on the State. Each byte in the State is denoted by  $S_{i,j}$  ( $0 \leq i, j < 4$ ), and is considered as an element of  $GF(2^8)$ . Although different irreducible polynomials can be used to construct  $GF(2^8)$ , the irreducible polynomial used in the AES algorithm is  $p(x) = x^8 + x^4 + x^3 + x + 1$ . Fig. 1 shows the block diagram of the AES encryption and the equivalent decryption structures.

In the encryption of the AES algorithm, each round except the final round consists of four transformations: the SubBytes, the ShiftRows, the MixColumns, and the AddRoundKey, while the final round does not have the MixColumns transformation. The SubBytes is a nonlinear transformation, which computes the multiplicative inverse of each byte of the State in  $GF(2^8)$  followed by an affine transformation. The SubBytes can be described by (1)

$$S'_{i,j} = MS_{i,j}^{-1} + C \quad (1)$$

where  $M$  is an  $8 \times 8$  binary matrix, and  $C$  is an 8-bit binary vector with only 4 nonzero bits. ShiftRows is a simple shifting transformation. The first row of the State does not change, while

the second, third and fourth rows cyclically shift one byte, two bytes and three bytes to the left, respectively. The MixColumns transformation considers the four bytes in each column of the State as the coefficients of a polynomial over  $GF(2^8)$ , and multiplies by  $a(x)$  modulo  $x^4 + 1$  where

$$a(x) = \{03\}_{16}x^3 + \{01\}_{16}x^2 + \{01\}_{16}x + \{02\}_{16}. \quad (2)$$

$\{y\}_{16}$  denotes the number “ $y$ ” in hexadecimal form, while the notation  $\{y\}_2$  used later in this paper stands for “ $y$ ” in binary form. In matrix form, the MixColumns transformation can be expressed as

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{02\}_{16} & \{03\}_{16} & \{01\}_{16} & \{01\}_{16} \\ \{01\}_{16} & \{02\}_{16} & \{03\}_{16} & \{01\}_{16} \\ \{01\}_{16} & \{01\}_{16} & \{02\}_{16} & \{03\}_{16} \\ \{03\}_{16} & \{01\}_{16} & \{01\}_{16} & \{02\}_{16} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}, \quad 0 \leq c < 4. \quad (3)$$

Finally, the AddRoundKey involves only bit-wise XOR operations.

The transformations in the decryption process perform the inverse of the corresponding transformations in the encryption process. Specifically, the InvSubBytes performs the following operation on each byte of the State

$$S'_{i,j} = (M^{-1}(S_{i,j} + C))^{-1}. \quad (4)$$

In the InvShiftRows, the first row of the State does not change, while the rest of the rows are cyclically shifted to the right by the same offset as that in the ShiftRows. The InvMixColumns multiplies the polynomial formed by each column of the State with  $a^{-1}(x)$  modulo  $x^4 + 1$ , where

$$a^{-1}(x) = \{0b\}_{16}x^3 + \{0d\}_{16}x^2 + \{09\}_{16}x + \{0e\}_{16}. \quad (5)$$

In matrix form, the InvMixColumns transformation can be expressed by

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{0e\}_{16} & \{0b\}_{16} & \{0d\}_{16} & \{09\}_{16} \\ \{09\}_{16} & \{0e\}_{16} & \{0b\}_{16} & \{0d\}_{16} \\ \{0d\}_{16} & \{09\}_{16} & \{0e\}_{16} & \{0b\}_{16} \\ \{0b\}_{16} & \{0d\}_{16} & \{09\}_{16} & \{0e\}_{16} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}, \quad 0 \leq c < 4. \quad (6)$$

The decryption structure can be derived by inverting the encryption structure directly. However, the sequence of the transformations will be different from that in encryption. This feature prohibits resource sharing between encryptors and decryptors. As can be observed from the operations involved in the decryption transformations, the InvShiftRows and the InvSubBytes can be exchanged without affecting the decryption process. Meanwhile, the InvMixColumns can be moved before the AddRoundKey, provided that the InvMixColumns are applied to the roundkeys before they are added. Taking these into consideration, an equivalent decryption structure as that in Fig. 1(b) can be used [1]. In this figure, the mixroundkeys are the modified roundkeys resulted from applying InvMixColumns to the roundkeys. The equivalent decryption structure has the

same sequence of transformations as that in the encryption structure, and thus, resource sharing between encryptors and decryptors are enabled.

In the AES algorithm, the key expansion process generates a total of  $4(Nr + 1)$  4-byte words ( $w_0, w_1, \dots, w_{4(Nr+1)-1}$ ). The initial key, which can be divided into  $Nk$  words ( $key_0, key_1, \dots, key_{Nk-1}$ ), is used as the initial  $Nk$  words, and the rest of the words are generated from the initial key iteratively. It can be computed that  $Nk$  is 4, 6, or 8, when the key length is 128, 192 or 256-bit, respectively. Each roundkey has 128 bits, and is formed by concatenating four words:  $roundkey(i) = (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3})$ . The key expansion process can be described by the pseudocode listed below [1].

### Key Expansion

for  $i = 0$  to  $Nk - 1$

$w_i = key_i$

end

for  $i = Nk$  to  $4(Nr + 1) - 1$

temp =  $w_{i-1}$

if  $(i \bmod Nk = 0)$

temp = SubWord(RotWord( $w_{i-1}$ )) XOR Rcon( $i/Nk$ )

else if  $(Nk > 6$  and  $i \bmod Nk = 4)$

temp = SubWord( $w_{i-1}$ )

end if

$w_i = w_{i-Nk}$  XOR temp

end

In the key expansion, the SubWord applies SubBytes transformation to each of the four bytes in a word, while the RotWord cyclically shifts each byte in a word one byte to the left. The Rcon is a constant word array, and only the leftmost byte in each word is nonzero.

### B. The Subpipelined Architecture

Three architectural optimization approaches can be used to speed up the AES algorithm in non-feedback modes by duplicating hardware for implementing each round, which is also called round unit. These architectures are based on pipelining, subpipelining and loop-unrolling. The pipelined architecture is realized by inserting rows of registers between each round unit. Similar to the pipelining, subpipelining also inserts rows of registers among combinational logic, but registers are inserted both between and inside each round unit. In pipelining and subpipelining, multiple blocks of data are processed simultaneously. Comparatively, loop unrolled or unfolded architectures can process only one block of data at a time, but multiple rounds are processed in each clock cycle. Among these architectural optimization approaches, subpipelining can achieve maximum speedup and optimum speed/area ratio in non-feedback modes.

Fig. 2 illustrates the architecture of subpipelining. The number of round units in each loop,  $k$ , is usually chosen as a divisor of  $Nr$ . When  $k = Nr$ , it is called a fully subpipelined architecture. If each round unit can be divided into  $r$  substages with equal delay, a  $k$ -round subpipelined architecture can achieve approximately  $r(1 + \tau)/(1 + r\tau)$  times the speed of a  $k$ -round pipelined architecture, where

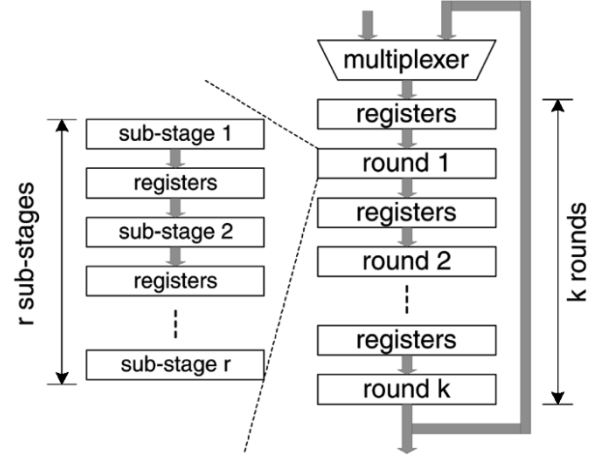


Fig. 2. The architecture of subpipelining.

$\tau = (t_{\text{setup}} + t_{\text{prop}} + t_{\text{mux}})/t_{\text{round}} \cdot t_{\text{setup}}$  and  $t_{\text{prop}}$  stand for the setup time and the propagation delay of a register, while  $t_{\text{mux}}$  denotes the delay of a multiplexer and  $t_{\text{round}}$  is the delay of the combinational logic in each round unit. It can be observed that the more substages with equal delay each round unit can be divided into, the larger speedup the subpipelining can achieve. Usually  $\tau$  is small, so there is almost  $r$  times speedup over pipelining at the cost of  $k(r - 1)$  additional rows of registers and slightly increased area caused by larger control logics.

However, dividing each round unit into arbitrary number of substages does not always bring speedup. Since the minimum clock period is determined by the indivisible component with the longest delay, dividing the rest of the round unit into more substages with shorter delay does not reduce the minimum clock period. Although more blocks of data are being processed simultaneously, the average number of clock cycles to process one block of data does not change. Therefore, the overall speed does not improve despite increased area caused by the additional registers. In a LUT-based implementation, it can be observed that nearly half the delay of a round unit is attributed to the LUTs, and thus, each round unit can be divided into only two substages to achieve some speedup without wasting any area. On the contrary, the longest unbreakable delay in the non-LUT-based approaches is the delay of individual logic gates. Accordingly, each round unit can be divided into multiple substages with approximately equal delay.

### C. Composite Field Arithmetic

The non-LUT-based implementations of the AES algorithm are able to exploit the advantage of subpipelining further. Nevertheless, these approaches may have high hardware complexities. Although two Galois Fields of the same order are isomorphic, the complexity of the field operations may heavily depend on the representations of the field elements. Composite field arithmetic can be employed to reduce the hardware complexity. 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 [12] if

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

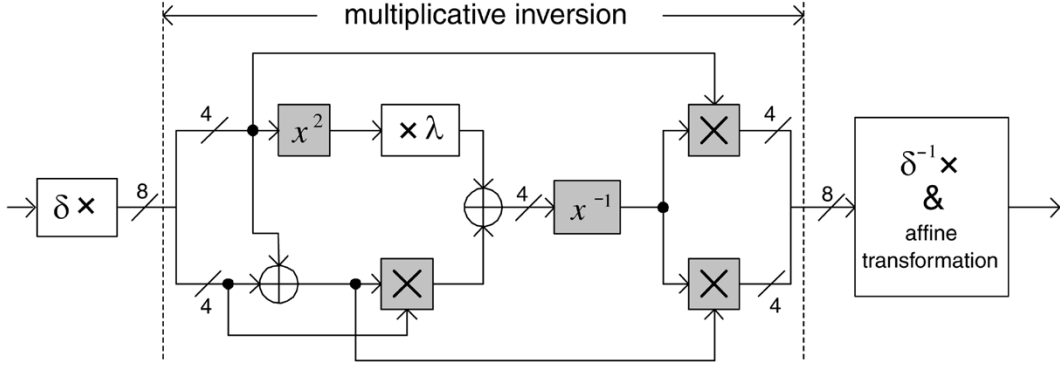


Fig. 3. Implementation of the SubBytes Transformation.

Composite fields will be denoted by  $GF((2^n)^m)$ , and a composite field  $GF((2^n)^m)$  is isomorphic to the field  $GF(2^k)$  for  $k = nm$ . Additionally, composite fields can be built iteratively from lower order fields. For example, the composite field of  $GF(2^8)$  can be built iteratively from  $GF(2)$  using the following irreducible polynomials [7]:

$$\begin{cases} GF(2) \Rightarrow GF(2^2) : & P_0(x) = x^2 + x + 1 \\ GF(2^2) \Rightarrow GF((2^2)^2) : & P_1(x) = x^2 + x + \phi \\ GF((2^2)^2) \Rightarrow GF(((2^2)^2)^2) : & P_2(x) = x^2 + x + \lambda \end{cases} \quad (7)$$

where  $\phi = \{10\}_2$  and  $\lambda = \{1100\}_2$ . Meanwhile, an isomorphic mapping function  $f(x) = \delta \times x$  and its inverse need to be applied to map the representation of an element in  $GF(2^8)$  to its composite field and *vice versa*. The  $8 \times 8$  binary matrix  $\delta$  are decided by the field polynomials of  $GF(2^8)$  and its composite fields. Such a matrix can be found by the exhaustive-search-based algorithm in [12]. The  $\delta$  matrix corresponding to  $p(x) = x^8 + x^4 + x^3 + x + 1$  and the field polynomials in (7) can be found as below:

$$\delta = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}. \quad (8)$$

Taking the isomorphic mapping into consideration, not all the transformations in the AES algorithm are suitable to be implemented in the composite field. In order to facilitate substructure sharing, the constant multiplications in the MixColumns/InvMixColumns transformation are implemented by first computing  $\{02\}_{16}S_{i,j}$ ,  $\{04\}_{16}S_{i,j}$  and  $\{08\}_{16}S_{i,j}$ , then adding those terms corresponding to the nonzero bits in the constants. For example, the constant multiplication of  $\{0b\}_{16} = \{00001011\}_2$  can be computed by adding  $S_{i,j}$ ,  $\{02\}_{16}S_{i,j}$  and  $\{08\}_{16}S_{i,j}$ . In this approach, the  $\{02\}_{16}S_{i,j}$ ,  $\{04\}_{16}S_{i,j}$  and  $\{08\}_{16}S_{i,j}$  can be computed once and shared by all the constant multiplications. Meanwhile, the number of terms, which need to be added is determined by the number of nonzero bits in the constants. Using the  $\delta$  matrix defined in (8), the constant multiplications of  $\{02\}_{16}$

and  $\{03\}_{16}$  in  $GF(2^8)$  in the MixColumns are mapped to constant multiplications of  $\{5f\}_{16}$  and  $\{5e\}_{16}$  in the composite field, respectively. Although the hardware overhead of the mapping of constants can be eliminated by computing the mapping beforehand, the composite field representations of  $\{02\}_{16}$  and  $\{03\}_{16}$  have more nonzero bits, which makes the constant multiplications more expensive. The same argument also holds for the constant multiplications used in the InvMixColumns transformation, where  $\{09\}_{16}$ ,  $\{0b\}_{16}$ , and  $\{0e\}_{16}$  are mapped to  $\{75\}_{16}$ ,  $\{2a\}_{16}$  and  $\{57\}_{16}$  in the composite field, respectively. The only exception is that the composite field representation of  $\{0d\}_{16}$ , which is  $\{09\}_{16}$ , has one less nonzero bit, but this is offset by the larger number of nonzero bits in the composite field representations of the other three constants. Furthermore,  $\{10\}_{16}S_{i,j}$ ,  $\{20\}_{16}S_{i,j}$  and  $\{40\}_{16}S_{i,j}$  also need to be computed as a result of the higher-weight nonzero bits in  $\{75\}_{16}$ ,  $\{2a\}_{16}$  and  $\{57\}_{16}$ , which adds more complexity to the hardware implementations. Therefore, it is more efficient to implement the MixColumns/InvMixColumns in the original field  $GF(2^8)$ . The ShiftRows/InvShiftRows is a trivial transformation, only cyclical shifting is involved, and thus its implementation does not depend on the representation of *Galois Field* elements. Meanwhile, the field addition, which is simply XOR operation, has the same complexity in the composite field and the original field. Additionally, the affine/inverse affine transformation can be combined with the inverse isomorphic/isomorphic mapping. Based on the above observations, it is more efficient to carry out only the multiplicative inversion in the SubBytes/InvSubBytes in the composite field, while keep the rest of the transformations in the original field  $GF(2^8)$ .

### III. DETAILED HARDWARE IMPLEMENTATION ARCHITECTURES

In this section, we present detailed architectures for each of the nontrivial transformations in the AES algorithm. The implementation of each transformation is optimized to reduce area and increase speed. Meanwhile, an efficient key expansion architecture suitable for subpipelined round units is proposed. Based on the analysis on the gate counts in the critical path of the round units and the key expansion, optimized subpipelining architectures of the AES algorithm are presented.

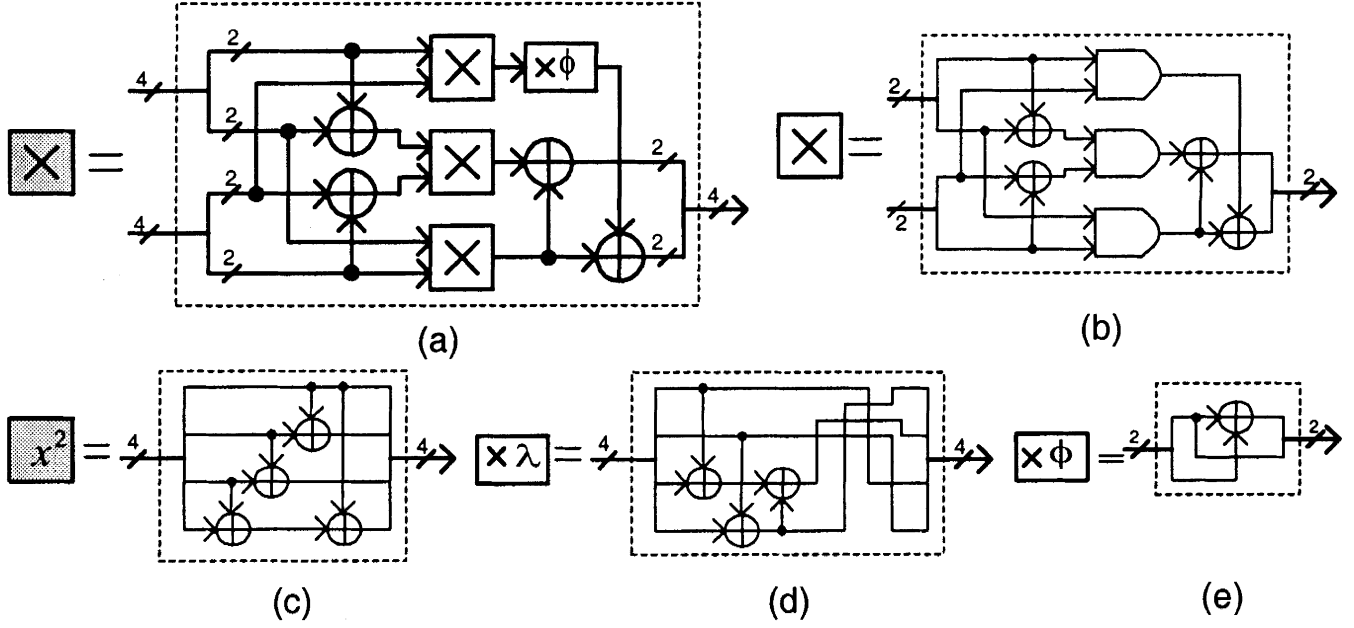


Fig. 4. Implementations of individual blocks: (a) multiplier in  $GF(2^4)$ ; (b) multiplier in  $GF(2^2)$ ; (c) squarer in  $GF(2^4)$ ; (d) constant multiplier ( $\times\lambda$ ); and (e) constant multiplier ( $\times\phi$ ).

#### A. Implementations of the SubBytes/InvSubBytes Transformation

The multiplicative inversion in  $GF(2^8)$  involved in the SubBytes/InvSubBytes is a hardware demanding operation, it takes at least 620 gates to implement by repeat multiplications in  $GF(2^8)$  [13]. However, the gate count can be reduced greatly by using composite field arithmetic. In the SubBytes transformation, using substructure sharing, the isomorphic mapping function can be implemented by 12 XOR gates with 4 XOR gates in the critical path. Meanwhile, the combined inverse isomorphic mapping and the affine transformation can be implemented by 19 XOR gates, and the critical path consists of 4 XOR gates also. In the composite field  $GF((2^4)^2)$ , an element can be expressed as  $s_h x + s_l$ , where  $s_h, s_l \in GF(2^4)$  and  $x$  is a root of  $P_2(x)$ . Using Extended Euclidean algorithm, the multiplicative inverse of  $s_h x + s_l$  modulo  $P_2(x)$  can be computed as in (9)

$$(s_h x + s_l)^{-1} = s_h \Theta x + (s_h + s_l) \Theta \quad (9)$$

where  $\Theta = (s_h^2 \lambda + s_h s_l + s_l^2)^{-1}$ . The proof of this equation is given in the Appendix.

According to (9), the multiplicative inversion in  $GF(2^8)$  can be carried out in  $GF((2^4)^2)$  by the architecture illustrated in Fig. 3. The multipliers in  $GF(2^4)$  can be further decomposed into multipliers in  $GF(2^2)$  and then to  $GF(2)$ , in which a multiplication is simply an AND operation. Fig. 4 illustrates this decomposition, together with the other blocks used in Fig. 3 except the inversion in  $GF(2^4)$  block. As can be observed from Fig. 4, a multiplier in  $GF(2^4)$  can be implemented by 21 XOR gates and 9 AND gates, with 4 XOR gates and 1 AND gate in the critical path. Table I summarizes the gate count and critical path of each block in the SubBytes except the block of inversion in  $GF(2^4)$  in Fig. 3.

TABLE I  
GATE COUNTS AND CRITICAL PATHS OF FUNCTIONAL BLOCKS IN THE SUBBYTES TRANSFORMATION

block	total # of gates	critical path
$\times\phi$	1 XOR	1 XOR
$\times\lambda$	3 XOR	2 XOR
$x^2$	4 XOR	2 XOR
multiplier in $GF(2^2)$	4 XOR + 3 AND	2 XOR + 1 AND
multiplier in $GF(2^4)$	21 XOR + 9 AND	4 XOR + 1 AND
$\times\delta$	12 XOR	4 XOR
$\times\delta^{-1}$ and affine	19 XOR	4 XOR

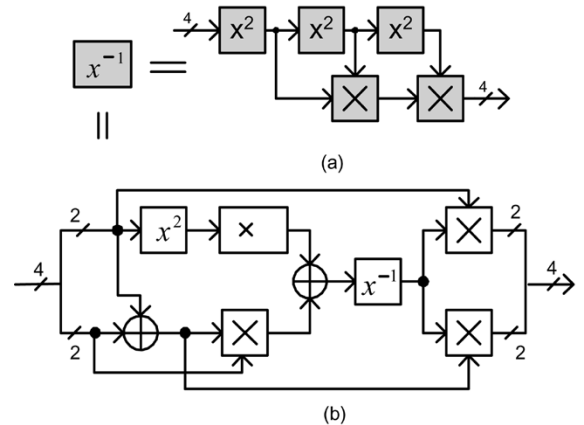


Fig. 5. Implementations of inversion in  $GF(2^4)$ . (a) Square-multiply approach. (b) Multiple decomposition approach.

The inversion in  $GF(2^4)$  can be implemented by different approaches.

- 1) Taking  $x \in GF(2^4)$ ,  $x^{-1} = x^{14} = ((x^2)^2)^2 \cdot (x^2)^2 \cdot x^2$ . Hence, the inversion can be implemented by repeat squaring and multiplying. This approach is illustrated in Fig. 5(a).

- 2) The inversion in  $GF(2^4)$  can be further decomposed by applying formulas similar to (9) iteratively. Fig. 5(b) shows the decomposition of inversion in  $GF(2^4)$  to  $GF((2^2)^2)$ . The combination of the squarer in  $GF(2^2)$  and the constant multiplier ( $\times\phi$ ) can be implemented by only exchanging the position of the two input bits so that no logic gate is involved. Meanwhile, it can be derived that the inversion of  $x' = \{x'_1, x'_0\} \in GF(2^2)$  is  $\{x'_1, x'_1 + x'_0\}$ , which can be implemented by one XOR gate.
- 3) Taking the four bits of  $x \in GF(2^4)$  as  $\{x_3, x_2, x_1, x_0\}$ , it can be derived that each bit in  $x^{-1} = \{x_3^{-1}, x_2^{-1}, x_1^{-1}, x_0^{-1}\}$  can be computed by the following equations:

$$\begin{cases} x_3^{-1} = x_3 + x_3x_2x_1 + x_3x_0 + x_2 \\ x_2^{-1} = x_3x_2x_1 + x_3x_2x_0 + x_3x_0 + x_2 + x_2x_1 \\ x_1^{-1} = x_3 + x_3x_2x_1 + x_3x_1x_0 + x_2 + x_2x_0 + x_1 \\ x_0^{-1} = x_3x_2x_1 + x_3x_2x_0 + x_3x_1 + x_3x_1x_0 + x_3x_0 \\ \quad + x_2 + x_2x_1 + x_2x_1x_0 + x_1 + x_0. \end{cases} \quad (10)$$

Using substructure sharing, the gate counts and delays of each of the three approaches are summarized in Table II. Composite field decomposition can reduce the hardware complexity significantly when the order of the field involved is large. However, for small fields, such as  $GF(2^4)$ , further decomposition may not be the optimum approach. As can be observed from Table II, direct implementation of the derived equation, (10), has the smallest gate count and the shortest critical path.

Similarly, from (4), the InvSubBytes transformation can be implemented according to the block diagram illustrated in Fig. 6. The combined inverse affine and the isomorphic mapping can be implemented by 17 XOR gates using substructure sharing, and the critical path consists of 4 XOR gates. Meanwhile, the inverse isomorphic mapping can be computed by 14 XOR gates with 3 XOR gates in the critical path. The same inversion architecture as that in Fig. 3 can be used in the InvSubBytes. Additionally, in a system where both the encryptor and decryptor need to be implemented on a small area, the SubBytes and InvSubBytes can share one multiplicative inversion unit.

### B. Implementations of the MixColumns/InvMixColumns Transformation

Various architectures have been proposed for the implementation of the MixColumns/InvMixColumns transformation [3], [5], [7], [14], [15]. Applying substructure sharing both to the computation of a byte and between the computation of the four bytes in a column of the State, an efficient MixColumns implementation architecture can be derived. Particularly, (3) can be rewritten as

$$\begin{cases} S'_{0,c} = \{02\}_{16}(S_{0,c} + S_{1,c}) + (S_{2,c} + S_{3,c}) + S_{1,c} \\ S'_{1,c} = \{02\}_{16}(S_{1,c} + S_{2,c}) + (S_{3,c} + S_{0,c}) + S_{2,c} \\ S'_{2,c} = \{02\}_{16}(S_{2,c} + S_{3,c}) + (S_{0,c} + S_{1,c}) + S_{3,c} \\ S'_{3,c} = \{02\}_{16}(S_{3,c} + S_{0,c}) + (S_{1,c} + S_{2,c}) + S_{0,c}. \end{cases} \quad (11)$$

According to (11), the MixColumns transformation can be implemented by the architecture shown in Fig. 7. The function of the block “XTime” is to compute constant multiplication by

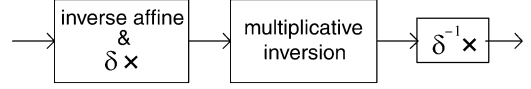


Fig. 6. Block diagram of the InvSubBytes transformation.

$\{02\}_{16}$ . An element of  $GF(2^8)$  can be expressed in polynomial form as  $S = s_7x^7 + s_6x^6 + s_5x^5 + s_4x^4 + s_3x^3 + s_2x^2 + s_1x + s_0$ , where  $s_0, s_1, \dots, s_7 \in GF(2)$ , and  $x$  is a root of the field polynomial  $p(x)$ . Then

$$\begin{aligned} \{02\}_{16}S &= xS = s_7x^8 + s_6x^7 + s_5x^6 + s_4x^5 \\ &\quad + s_3x^4 + s_2x^3 + s_1x^2 + s_0x \pmod{p(x)} \\ &= s_6x^7 + s_5x^6 + s_4x^5 + (s_3 + s_7)x^4 \\ &\quad + (s_2 + s_7)x^3 + s_1x^2 + (s_0 + s_7)s + s_7. \end{aligned}$$

Therefore, the “XTime” block can be implemented by 3 XOR gates with only one XOR gate in the critical path. As illustrated in Fig. 7, the total number of XOR gates for computing one column of the State is 108, and the critical path is 3 XOR gates.

Similarly, in the InvMixColumns transformation, (6) can be rewritten as

$$\begin{cases} S'_{0,c} = (\{02\}_{16}(S_{0,c} + S_{1,c}) + (S_{2,c} + S_{3,c}) + S_{1,c}) \\ \quad + (\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\ \quad + \{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{0,c} + S_{2,c})) \\ S'_{1,c} = (\{02\}_{16}(S_{1,c} + S_{2,c}) + (S_{3,c} + S_{0,c}) + S_{2,c}) \\ \quad + (\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\ \quad + \{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{1,c} + S_{3,c})) \\ S'_{2,c} = (\{02\}_{16}(S_{2,c} + S_{3,c}) + (S_{0,c} + S_{1,c}) + S_{3,c}) \\ \quad + (\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\ \quad + \{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{0,c} + S_{2,c})) \\ S'_{3,c} = (\{02\}_{16}(S_{3,c} + S_{0,c}) + (S_{1,c} + S_{2,c}) + S_{0,c}) \\ \quad + (\{02\}_{16}(\{04\}_{16}(S_{0,c} + S_{2,c}) \\ \quad + \{04\}_{16}(S_{1,c} + S_{3,c})) + \{04\}_{16}(S_{1,c} + S_{3,c})). \end{cases} \quad (12)$$

Using substructure sharing, (12) can be implemented by the architecture illustrated in Fig. 8. The “X4Time” block, which computes the constant multiplication of  $\{04\}_{16}$ , can be implemented by two serially concatenated “XTime” block. Alternatively, it can also be implemented according to the equation derived below

$$\begin{aligned} \{04\}_{16}S &= x^2S = s_7x^9 + s_6x^8 + s_5x^7 + s_4x^6 + s_3s^5 \\ &\quad + s_2x^4 + s_1x^3 + s_0x^2 \pmod{p(x)} \\ &= s_5x^7 + s_4x^6 + (s_3 + s_7)x^5 + (s_2 + (s_6 + s_7))x^4 \\ &\quad + (s_1 + s_6)x^3 + (s_0 + s_7)x^2 + (s_6 + s_7)x + s_6. \end{aligned}$$

Sharing  $s_6 + s_7$ , the “X4Time” block can be implemented by 5 XOR gates with 2 XOR gates in the critical path. It follows that the architecture in Fig. 8 can be implemented by 193 XOR gates with 7 XOR gates in the critical path. Meanwhile, the upper half in Fig. 8 is exactly the same as the architecture for the implementation of the MixColumns in Fig. 7. Therefore in a joint encryptor/decryptor implementation, only the architecture in Fig. 8 needs to be implemented for both the MixColumns and the InvMixColumns transformations.

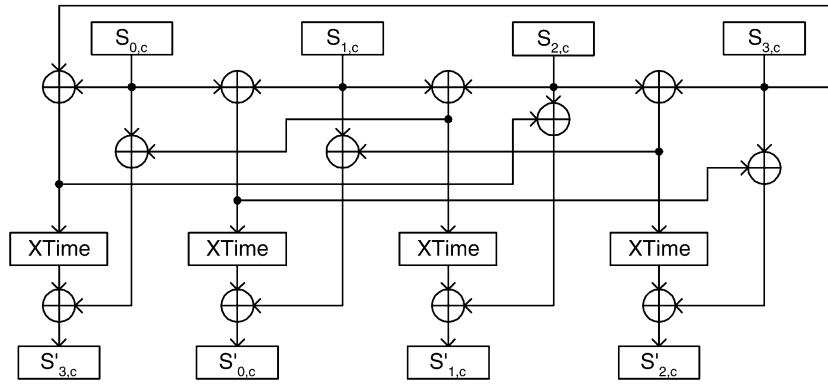


Fig. 7. Efficient implementation of the MixColumns transformation.

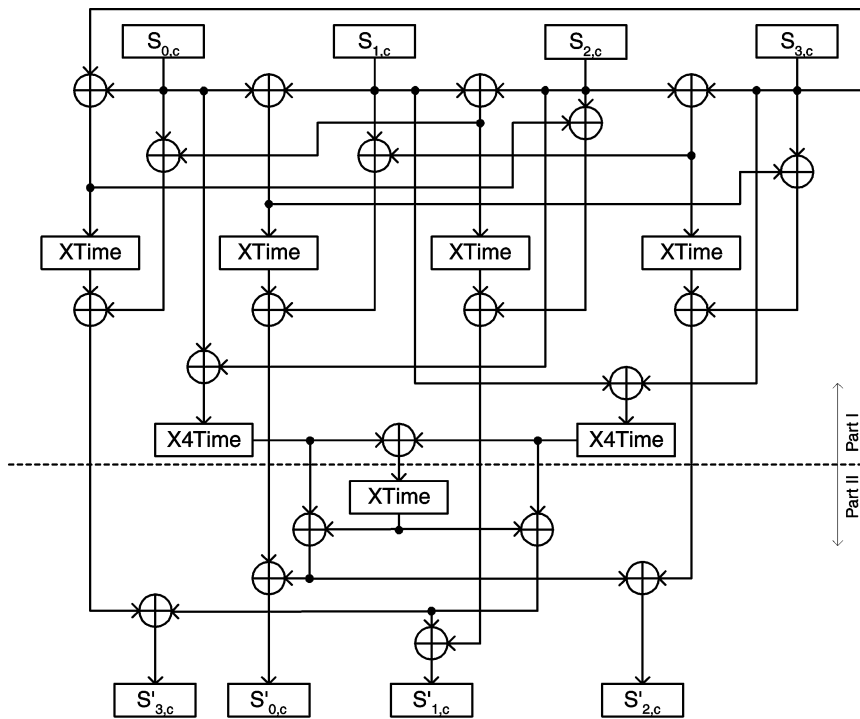


Fig. 8. Efficient implementation of the InvMixColumns transformation.

TABLE II  
GATE COUNTS AND CRITICAL PATHS FOR THE THREE IMPLEMENTATION APPROACHES OF INVERSION IN  $GF(2^4)$

approach	total number of gates	critical path
1	54XOR+18AND	12XOR+2AND
2	17XOR+9AND	7XOR+2AND
3	14XOR+9AND	3XOR+2AND

C. Implementations of Key Expansion

Roundkeys can be either generated beforehand and stored in memory or generated on the fly. In the former approach, roundkeys can be read out from memory using appropriate addresses, and there is no extra delay for decryption. However, this approach is not suitable for the applications where the key changes constantly. Meanwhile, the delay of memory access is unbreakable, which may offset the speedup achieved by subpipelining the round units. Therefore it is more advantageous

to generate roundkeys on the fly in a subpipelined architecture. Fig. 9 shows a key expansion architecture suitable for  $r$ -substage subpipelined AES algorithm with 128-bit key. This key expansion architecture is capable of generating all the  $Nr + 1$  roundkeys after  $r \times Nr$  clock cycles. For the purpose of clarity, the “clock” input is omitted for all the registers except the ones in the top row, which use “load(i)” ( $0 \leq i \leq Nr$ ) as the “clock” input. In Fig. 9, at the “start” signal, the initial key is loaded into the registers in the first column and the key expansion process begins. At clock cycle  $r \times i$ , the output of the registers in the first column is the corresponding “roundkey(i).” The Controller in Fig. 9 is designed so that the signal “load(i)” is initially “0,” goes to “1” in clock cycle  $r \times i$ , and stays at “1” afterwards. Such a controller can be easily implemented by two serially concatenated Johnson counters. The rising-edge active registers in the top row load “roundkey(i)” at clock cycle  $r \times i$ , respectively. After  $r \times Nr$  clock cycles, all the

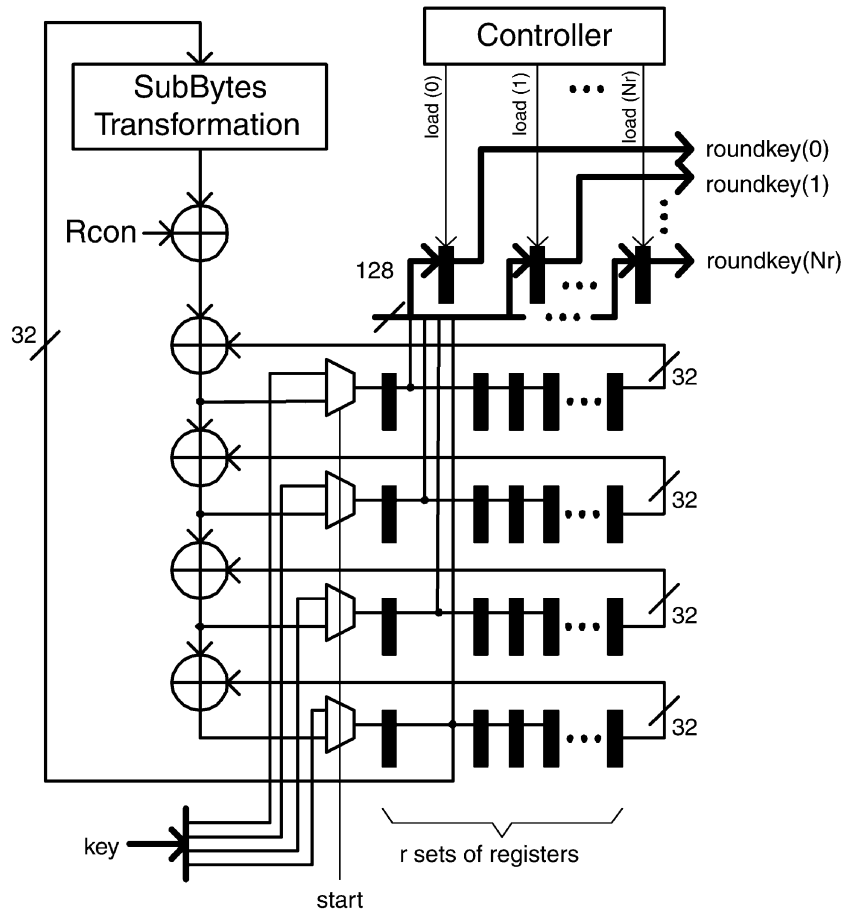


Fig. 9. The key expansion architecture for r-sub-stage subpipelined AES algorithm with 128-bit key.

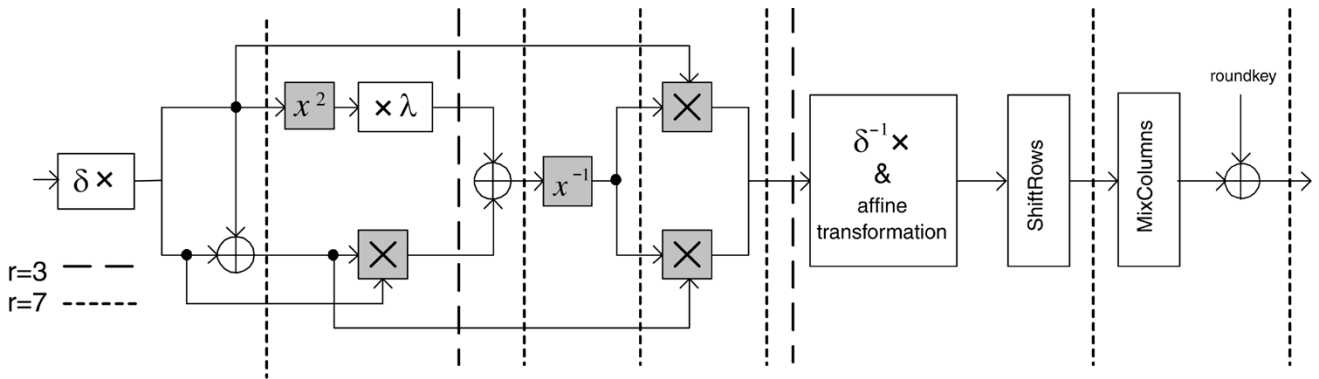


Fig. 10. Subpipelining cutsets for the round unit in encryptor.

$Nr + 1$  roundkeys are available at the output of the top row registers and are held there for the entire encryption/decryption process. As can be observed from Fig. 9, the critical path of the key expansion unit has 5 XOR gates and 1 2-to-1 Mux besides the SubBytes transformation block.

Using the key expansion architecture in Fig. 9, the data encryption and the key expansion can start simultaneously. However, since the roundkeys are used in reverse order, the decryption process can only start after the last roundkey is generated. Meanwhile, the InvMixColumns transformation needs to be performed on the roundkeys to get the mixroundkeys.

#### D. Subpipelined Round Unit and Key Expansion Unit

The subpipelined architecture can achieve maximum speedup if each round unit can be divided into substages with equal delay. Based on the analysis of the gate count in the critical path of each component, cutsets as illustrated in Fig. 10 can be added to divide the encryption round unit into  $r = 3$  and 7 substages with approximately equal delay. Since the roundkeys are generated on the fly, we need to divide the key expansion unit into the same number of substages with the same maximum delay as in the round unit to avoid extra buffers and delay. Assuming the same subpipelined SubBytes transformation is used in the key



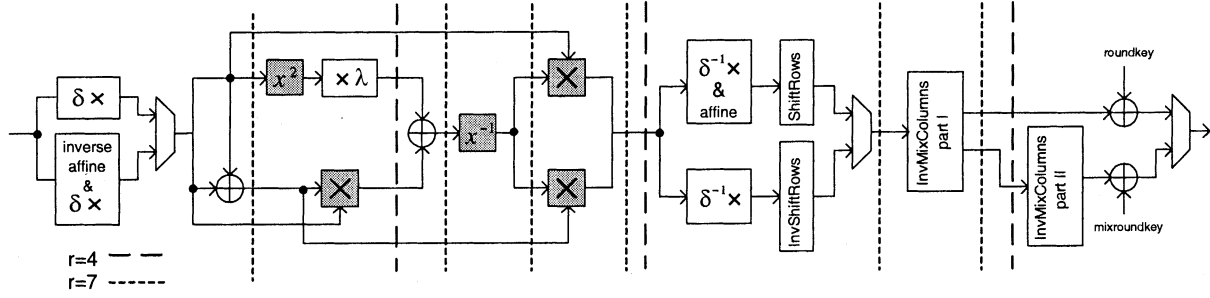


Fig. 11. Subpipelining cutsets for the round unit in joint encryptor/decryptor.

 TABLE III  
 COMPARISONS OF FPGA IMPLEMENTATIONS OF THE AES ALGORITHM

Design	Device	Frequency (Mhz)	Throughput (Mbps)	Slices	BRAMs	Mbps /Slice
Elbirt <i>et al</i> [2]	XCV1000-4	31.8	1938	10992	0	0.176
Mcloone <i>et al</i> [6] (pre-placement timing)	XCV812e-8	93.9	12020	2000	244	0.362
Jarvinen <i>et al</i> [9]	XCV1000e-8	129.2	16500	11719	0	1.408
Saggese <i>et al</i> [10]	XCV2000e-8	158	20300	5810	100	1.091
Standaert <i>et al</i> [11]	XCV3200e-8	145	18560	15112	0	1.228
This work (r=3)	XCV800-6	71.8	9184	9406	0	0.976
This work (r=3)	XCV812e-8	93.5	11965	9406	0	1.272
This work (r=7)	XCV1000-6	125.3	16032	11014	0	1.456
This work (r=7)	XCV1000e-8	168.4	21556	11022	0	1.956

expansion unit, the delay of the remaining path, which has 5 XOR gates and 1 MUX, does not exceed the maximum delay of the substages in the encryption round unit when  $r = 3$ . Thus this path can be left intact in this case. However, when  $r = 7$ , it needs to be divided into 2 substages. Since feedforward cutsets do not exist to break the path, retiming needs to be carefully explored. Meanwhile, a dummy substage is added after the AddRoundKey transformation in the round unit to keep the same number of substages as that in the key expansion.

Similarly, cutsets can also be added to the decryption round unit to achieve optimized subpipelining. However, since the critical path in the InvMixColumns exceeds that in the MixColumns, the decryption round unit may need to be divided into more substages to achieve the same speed as the encryption round unit. In a small-area joint encryptor/decryptor implementation, the subpipelining cutsets for the round units can be added as illustrated in Fig. 11. The InvMixColumns architecture is divided into two parts according to the dashed line in Fig. 8. Part I can implement the MixColumns transformation, and is shared for the implementation of both the MixColumns and the InvMixColumns. To achieve the same speed as a  $r = 3$  subpipelined encryptor, one extra cutset needs to be added to the round unit of a joint encryptor/decryptor. However, when  $r = 7$ , this can be done by moving the dummy cutset to break the InvMixColumns.

#### IV. IMPLEMENTATION RESULTS AND COMPARISON

As an example of our proposed architecture, fully subpipelined AES encryptors of 128-bit key, which has 10 copies

of the round unit and a key expansion unit, are implemented on FPGA devices. Xilinx ISE 5.1 i is used to synthesize the design and provide post-placement timing results. In the fully subpipelined encryptor implementations, data blocks are accepted every clock cycle. After an initial delay of  $r \times Nr + 1$  clock cycles, the corresponding ciphertext blocks appear at the output every clock cycle. The  $r = 3$  fully subpipelined architecture is implemented on Xilinx XCV800-6bg560 devices. This implementation can achieve a throughput of 9.19 Gbps. Since the extra cost of more subpipelining stages are registers and slightly larger counters in the Controller block, and the register usage on a XCV800-6bg560 device for  $r = 3$  design is only 35%, the  $r = 7$  design should be able to fit in the same device theoretically. However, due to the unique structure of Xilinx FPGA chips, chopping the combinational logic into smaller pieces makes the resource in each Configurable Logic Block (CLB) under-utilized. As a result, the  $r = 7$  design is unable to fit into a XCV800-6bg560 device. Implemented on a Xilinx XCV1000-6bg560 device, the  $r = 7$  fully subpipelined architecture can achieve a throughput of 16.03 Gbps. Table III shows the comparison with selected existing FPGA implementations. Using devices of the same speed as in [6], [9]–[11], a fully subpipelined architecture with 7 substages in each round unit can achieve 21.56 Gbps on a Xilinx XCV1000e-8bg560 device. As can be observed from Table III, our architecture can achieve higher speed than all prior FPGA implementations known to the authors, and is 79% more efficient than the previous fastest design [10] in terms of equivalent throughput/slice. In the computation of throughput/slice, one BlockRAM (BRAM) is equivalent to 128 slices [10].

Further speedup can be achieved by dividing each round unit into more substages with equal delay. In this aspect, it has advantages over the designs utilizing BRAMs on Xilinx FPGAs to implement SubBytes/InvSubBytes. Since the minimum clock period is decided by the unbreakable delay of BRAMs, a fully subpipelined implementation using BRAMs can not achieve higher speed even if larger FPGA devices are available.

## V. CONCLUSION

In this paper, efficient subpipelined architectures of the AES algorithm are presented. In order to explore the advantage of subpipelining further, the SubBytes/InvSubBytes is implemented by combinational logic to avoid the unbreakable delay of LUTs in the traditional designs. Additionally, composite field arithmetic is used to reduce the hardware complexity and different approaches for the implementation of inversion in subfield  $GF(2^4)$  are compared. As an example of our proposed architecture, fully subpipelined encryptors using 128-bit key are implemented on FPGA devices. Decryptors can be easily incorporated by using the subpipelined joint encryptor/decryptor round unit architecture presented in this paper, and we expect the throughput will be slightly lower than the encryptor-only implementations. Meanwhile, fully subpipelined encryptors/decryptors using other key lengths can be implemented by adding more copies of round units and modifying the key expansion unit slightly. Furthermore, the number of round units in a loop can be reduced to meet the requirements of small area applications. Future work will address the subpipelined variable key length implementations using composite field arithmetic.

## APPENDIX

*Proof:* The problem of finding the inverse of  $S(x) = s_h x + s_l$  modulo  $P_2(x) = x^2 + x + \lambda$  is equivalent to finding polynomials  $A(x)$  and  $B(x)$  satisfying the following equation:

$$A(x)P_2(x) + B(x)S(x) = 1. \quad (13)$$

Then  $B(x)$  in (13) is the inverse of  $S(x)$  modulo  $P_2(x)$ . Such  $A(x)$  and  $B(x)$  can be found by using the Extended Euclidean Algorithm for one iteration. First, we need to rewrite  $P_2(x)$  in the form of

$$P_2(x) = Q(x)S(x) + R(x) \quad (14)$$

where  $Q(x)$  and  $R(x)$  are the quotient and remainder polynomials of dividing  $P_2(x)$  by  $S(x)$ , respectively. By long division, it can be derived that

$$Q(x) = s_h^{-1}x + (1 + s_h^{-1}s_l) s_h^{-1}, \quad (15)$$

$$R(x) = \lambda + (1 + s_h^{-1}s_l) s_h^{-1}s_l. \quad (16)$$

Substituting (15) and (16) into (14) and multiplying  $s_h^2$  to both sides of the equation, it follows that

$$s_h^2 P_2(x) = (s_h x + (s_h + s_l))S(x) + (s_h^2 \lambda + s_h s_l + s_l^2). \quad (17)$$

Multiplying  $\Theta = (s_h^2 \lambda + s_h s_l + s_l^2)^{-1}$  to both side of (17), we get

$$\Theta s_h^2 P_2(x) = \Theta (s_h x + (s_h + s_l))S(x) + 1. \quad (18)$$

Since addition and subtraction are the same in the extended field of  $GF(2)$ , the first term on the right side of (18) can be moved to the left side. Comparing (13) and (18), it can be observed that

$$S^{-1}(X) = s_h \Theta x + (s_h + s_l) \Theta.$$

## REFERENCES

- [1] *Advanced Encryption Standard (AES)*, Nov. 26, 2001.
- [2] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalist. presented at *Proc. 3rd AES Conf. (AES3)*. [Online]. Available: <http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html>
- [3] V. Fischer and M. Drutarovsky, "Two methods of Rijndael implementation in reconfigurable hardware," in *Proc. CHES 2001*, Paris, France, May 2001, pp. 77–92.
- [4] K. Gaj and P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. presented at *Proc. 3rd AES Conf. (AES3)*. [Online]. Available: <http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html>
- [5] H. Kuo and I. Verbauwhede, "Architectural optimization for a 1.82 Gbits/sec VLSI implementation of the AES Rijndael algorithm," in *Proc. CHES 2001*, Paris, France, May 2001, pp. 51–64.
- [6] M. McLoone and J. V. McCanny, "Rijndael FPGA implementation utilizing look-up tables," in *IEEE Workshop on Signal Processing Systems*, Sept. 2001, pp. 349–360.
- [7] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact Rijndael hardware architecture with S-Box optimization," in *Proc. ASIACRYPT 2001*, Gold Coast, Australia, Dec. 2000, pp. 239–254.
- [8] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, "Efficient implementation of Rijndael encryption with composite field arithmetic," in *Proc. CHES 2001*, Paris, France, May 2001, pp. 171–184.
- [9] K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta, "A fully pipelined memoryless 17.8 Gbps AES-128 encryptor," in *Proc. Int. Symp. Field-Programmable Gate Arrays (FPGA 2003)*, Monterey, CA, Feb. 2003, pp. 207–215.
- [10] G. P. Saggese, A. Mazzeo, N. Mazocca, and A. G. M. Strollo, "An FPGA based performance analysis of the unrolling, tiling and pipelining of the AES algorithm," in *Proc. FPL 2003*, Portugal, Sept. 2003.
- [11] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat, "Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements & design tradeoffs," in *Proc. CHES 2003*, Cologne, Germany, Sept. 2003.
- [12] C. Paar, "Efficient VLSI architecture for bit-parallel computations in Galois field," Ph.D. dissertation, Institute for Experimental Mathematics, University of Essen, Essen, Germany, 1994.
- [13] M. H. Jing, Y. H. Chen, Y. T. Chang, and C. H. Hsu, "The design of a fast inverse module in AES," in *Proc. Int. Conf. Info-Tech and Info-Net*, vol. 3, Beijing, China, Nov. 2001, pp. 298–303.
- [14] C. C. Lu and S. Y. Tseng, "Integrated design of AES (advanced encryption standard) encrypter and decrypter," in *Proc. IEEE Int. Conf. Application Specific Systems, Architectures Processors*, 2002, pp. 277–285.
- [15] X. Zhang and K. K. Parhi, "Implementation approaches for the advanced encryption standard algorithm," *IEEE Circuits Syst. Mag.*, vol. 2, no. 4, pp. 24–46, 2002.



**Xinniao Zhang** received the B.S. and M.S. degrees in electrical engineering from Tianjin University, Tianjin, China, in 1997 and 2000, respectively. She is currently working toward the Ph.D. degree in the Department of Electrical and Computer Engineering, University of Minnesota-Twin Cities, MN.

Her research interests include efficient VLSI architecture design for error-correcting codes, digital signal processing systems, and cryptosystems.

Ms. Zhang is the recipient of the Best Paper Award at ACM Great Lake Symposium on VLSI 2004.



**Keshab K. Parhi** (S'85–M'88–SM'91–F'96) received the B.Tech., M.S.E.E., and Ph.D. degrees from the Indian Institute of Technology, Kharagpur, the University of Pennsylvania, Philadelphia, and the University of California at Berkeley, in 1982, 1984, and 1988, respectively.

He has been with the University of Minnesota, MN, since 1988, where he is currently the Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering. His research addresses VLSI architecture design

and implementation of physical layer aspects of broadband communications systems. He is currently working on error control coders and cryptography architectures, high-speed transceivers, ultra wideband systems, quantum error control coders and quantum cryptography. He has published over 350 papers, has authored the text book *VLSI Digital Signal Processing Systems* (New York: Wiley, 1999) and coedited the reference book *Digital Signal Processing for Multimedia Systems* (New York: Marcel Dekker, 1999).

Dr. Parhi is the recipient of numerous awards including the 2004 F.E. Terman award by the American Society of Engineering Education, the 2003 IEEE Kiyo Tomiyasu Technical Field Award, the 2001 IEEE W.R.G. Baker prize paper award, and a Golden Jubilee award from the IEEE Circuits and Systems Society in 1999. He has served on the editorial boards of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: ANALOG AND DIGITAL SIGNAL PROCESSING, TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, *Signal Processing*, and *Signal Processing Letters*. He currently serves on the editorial board of the *IEEE Signal Processing Magazine*, and is currently the Editor-in-Chief of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS (2004–2005 term). He served as Technical Program Cochair of the 1995 IEEE VLSI Signal Processing workshop and the 1996 ASAP Conference, and as the General Chair of the 2002 IEEE Workshop on Signal Processing Systems. He was a distinguished lecturer for the IEEE Circuits and Systems Society from 1996 to 1998.