Implementation Approaches for the Advanced Encryption Standard Algorithm

by Xinmiao Zhang
and Keshab K. Parhi

1531-636X/12/$10.00©2002IEEE

Abstract—*This paper addresses various approaches for efficient hardware implementation of the Advanced Encryption Standard algorithm. The optimization methods can be divided into two classes: architectural optimization and algorithmic optimization. Architectural optimization exploits the strength of pipelining, loop unrolling and sub-pipelining. Speed is increased by processing multiple rounds simultaneously at the cost of increased area. Architectural optimization is not an effective solution in feedback mode. Loop unrolling is the only architecture that can achieve a slight speedup with significantly increased area. In non-feedback mode, sub-pipelining can achieve maximum speedup and the best speed/area ratio. Algorithmic optimization exploits algorithmic strength inside each round unit. Various methods to reduce the critical path and area of each round unit are presented. Resource sharing issues between encryptor and decryptor are also discussed. They become important issues when both encryptor and decryptor need to be implemented in a small area.*

Index Terms—Advanced Encryption Standard, Rijndael, *S*-box, key expansion, pipelining, loop unrolling, substructure sharing.

Xinmiao Zhang and Keshab K. Parhi are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, Minnesota, 55455. E-mails: {jennizh, parhi}@ece.umn.edu

This work has been supported by the Army Research Office under grant number DA/DAAD 19-01-1-0705.

This paper addresses efficient hardware implementation approaches for the AES algorithm. Compared to software implementations, hardware implementations provide more physical security as well as higher speed. Different applications of the AES algorithm may require different speed/area trade-offs. Some applications, such as smart cards and cellular phones, require small area. Other applications, such as WWW servers and ATMs, are speed critical. Some other applications, such as digital video recorders, require an optimization of speed/area ratio.

Introduction

Cryptography plays an important role in the security of data transmission. On January 2, 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). Among the 15 preliminary candidates, MARS, RC6, Rijndael [1], Serpent, and Twofish were announced as the finalist candidates on August 9, 1999 for further evaluation. After studying all available information and public comments on these finalist candidates, NIST announced in October 2000 that Rijndael was selected as the AES algorithm.

When selecting the AES algorithm, both efficient software and hardware implementations were taken into consideration. This paper addresses efficient hardware implementation approaches for the AES algorithm. Compared to software implementations, hardware implementations provide

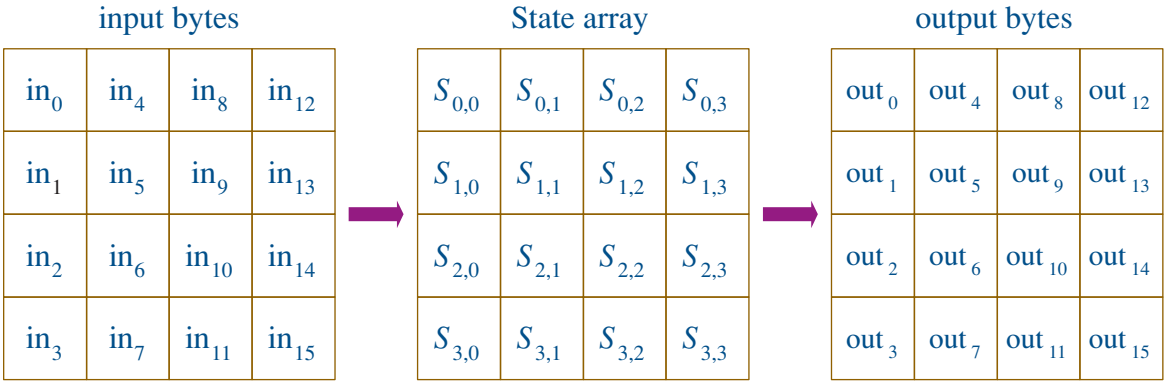


Figure 1. Mapping of input bytes, State array and output bytes.

more physical security as well as higher speed. Different applications of the AES algorithm may require different speed/area trade-offs. Some applications, such as smart cards and cellular phones, require small area. Other applications, such as WWW servers and ATMs, are speed critical. Some other applications, such as digital video recorders, require an optimization of speed/area ratio. Various optimizations for implementation are developed to suit the different demands of applications. Architectural optimizations make use of duplicating the round units, while algorithmic optimizations explore algorithm simplification inside each encryption/decryption round unit. In the next section, we briefly introduce the AES algorithm [2]. In the following section architectural optimization approaches are investigated. After that, algorithmic optimizations for each round unit in the AES algorithm are described. The last section explores resource sharing between encryptor and decryptor.

The AES Algorithm

The AES algorithm is a symmetric-key block cipher in which both the sender and receiver use a single key to encrypt and decrypt the information. Although in [1], the block length of Rijndael can be 128, 192, or 256 bits, the AES algorithm [2] only adopted the block length of 128 bits. Meanwhile, the key length can be 128, 192, or 256 bits. The AES algorithm's internal operations are performed on a two dimensional array of bytes called State, and each byte consists of 8 bits. The State consists of 4 rows of bytes and each row has *Nb* bytes. Each byte is denoted by $S_{i,j}$ ($0 \le i < 4$, $0 \le j < Nb$). Since the block length is 128 bits, each row of the State contains $Nb = 128 / (4 \times 8) = 4$ bytes. The four bytes in each column of the State array form a 32-bit word, with the row number as the index for the four bytes in each word. At the beginning of encryption or decryption, the array of input bytes is mapped to the State array as illustrated in Fig. 1, assuming a 128-bit block can be expressed as 16 bytes: in_0, in_1, in_2, … in_{15}. The encryption/decryption are performed on the State, at the end of which the final value is mapped to the output bytes array out_0, out_1, out_2, … out_{15}.

The key of the AES algorithm can be mapped to 4 rows of bytes in a similar way, except the number of bytes in each row denoted by *Nk* can be 4, 6, or 8 when the length of the key, *K*, is 128, 192, or 256 bits, respectively. The AES algorithm is an iterative algorithm. Each iteration can be called a round. The total number of rounds, *Nr*, is 10 when $Nk = 4$, $Nr = 12$ when $Nk = 6$, and $Nr = 14$ when $Nk = 8$.

Encryption

At the start of encryption, input is copied to the State array. After the initial roundkey addition, Nr rounds of encryption are performed. The first $Nr-1$ rounds are the same, with small difference in the final round. As illustrated in Fig. 2, each of the first $Nr-1$ rounds consists of 4 transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round excludes the MixColumns transformation.

SubBytes Transformation—SubBytes transformation is performed on each byte of the State using a substitution table (S-box). The S-box is constructed by first computing the multiplicative inverse of each element in GF(2^8) with irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$, the element {00} is mapped to itself. Then an affine transformation is applied which can be expressed in matrix form as

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \quad (1)$$

where b_i is the i^{th} bit of a byte. Here and elsewhere, a prime on a variable (e.g., b'_i) indicates that the variable is to be updated with the value on the right. For the convenience of expression, two hexadecimal characters are used to represent an 8-bit element of GF(2^8). For example, {6a} represents the element {01101010}.

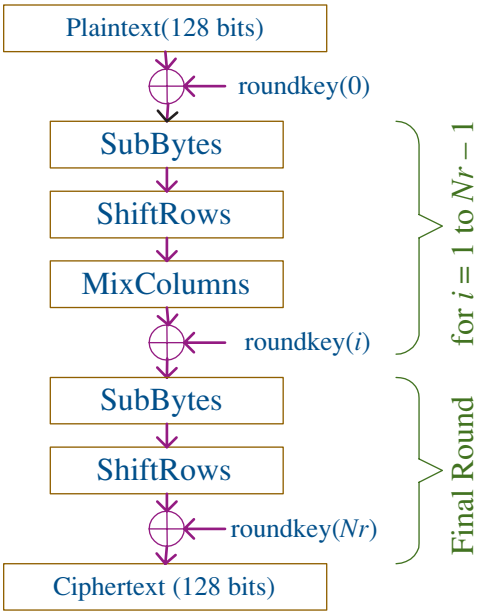


Figure 2. Encryption Structure of the AES algorithm.

ShiftRows Transformation—In this transformation, the bytes in the first row of the State do not change. The second, third, and fourth rows shift cyclically to the left one byte, two bytes, and three bytes, respectively, as illustrated in Fig. 3.

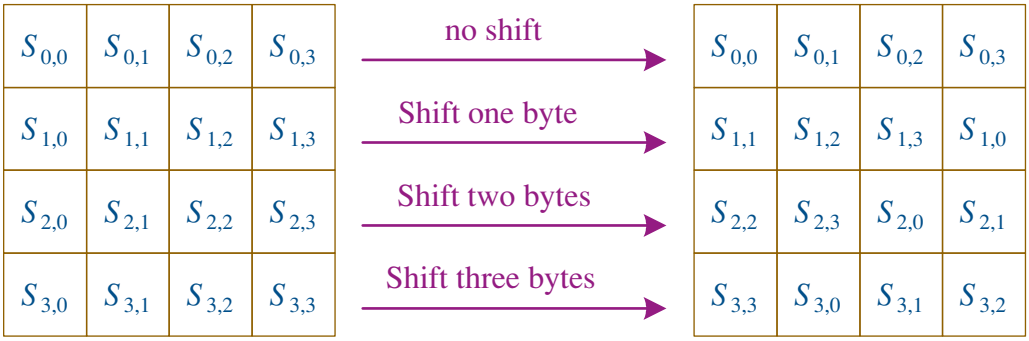


Figure 3. ShiftRows transformation.

```
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
```

Figure 4. Pseudo code for Key Expansion.

MixColumns Transformation—The MixColumns transformation is performed on the State column-by-column. Each column is considered as a four-term polynomial over GF(2^8) and multiplied by $a(x)$ modulo $x^4 + 1$, where

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

In matrix form, this transformation can be expressed as

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad 0 \le c < Nb \qquad (2)$$

AddRoundKey Transformation—In AddRoundKey transformation, a roundkey is added to the State by bitwise Exclusive-OR (XOR) operation. Each roundkey consists of Nb words generated from Key Expansion described below.

Key Expansion

In the AES algorithm, Key Expansion generates a total of $Nb(Nr + 1)$ words. The key, K, is used as the initial set of Nk words, and the rest of the words are generated from the key iteratively. The output of Key Expansion is an array of 4-byte words denoted by w_i, where $0 \le i < Nb(Nr + 1)$. Each roundkey in Fig. 2 is a concatenation of 4 words from the output of Key Expansion, roundkey(i) = (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}). The Key Expansion scheme can be expressed by the pseudo code in Fig. 4 [2].

SubWord in Fig. 4 applies S-box to each of the four bytes in a word. The function RotWord rotates each byte in a word one position to the left. For example, if the input is a word [a_0, a_1, a_2, a_3], RotWord returns the word [a_1, a_2 , a_3, a_0]. Rcon(i) is the round constant word array, whose value is [RC(i), {00}, {00}, {00}]. The values of RC(i) are listed as follows:

RC(1) = {01} RC(2) = {02}
RC(3) = {04} RC(4) = {08}
RC(5) = {10} RC(6) = {20}
RC(7) = {40} RC(8) = {80}
RC(9) = {1b} RC(10) = {36}.

Decryption

The encryption structure in Fig. 2 can be inverted to get a straightforward structure for decryption. Corresponding to the transformations in the encryption, InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey are the transformations used in the decryption. AddRoundKey is its own inverse, so the same name is used. Fig. 5(a) illustrates the straightforward structure of decryption. The roundkeys are the same as those in encryption generated by Key Expansion, but are used in reverse order.

InvSubBytes Transformation—InvSubBytes is the inverse transformation of SubBytes, in which the inverse S-box is applied to individual bytes in the State. The inverse S-box denoted by S^{-1}-box is constructed by first applying the inverse of the affine transformation in (1), then computing the multiplicative inverse in GF(2^8).

InvShiftRows Transformation—InvShiftRows is the inverse transformation of ShiftRows. In this transformation, the bytes in the first row of the State do not change; the second, third, and fourth row shift cyclically one byte, two bytes, and three bytes to the right, respectively.

InvMixColumns Transformation—InvMixColumns is the inverse transformation of MixColumns. Similar to MixColumns transformation, InvMixColumns transformation operates column-by-column on the State. Each column is considered as a polynomial over GF(2^8) and multiplied modulo $x^4 + 1$ by $a^{-1}(x)$ where

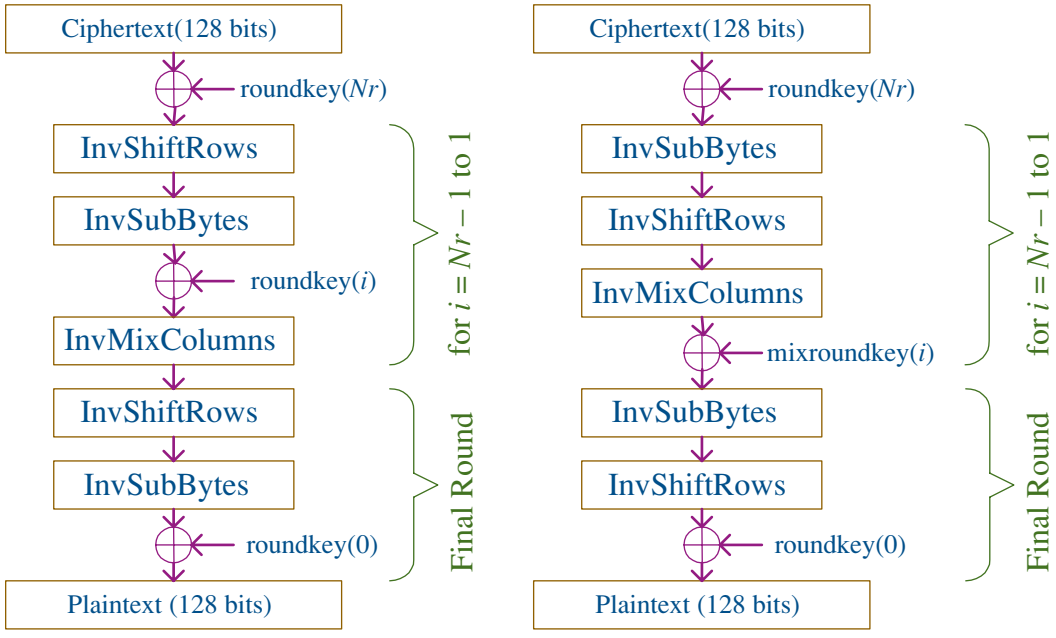
$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}.$$

In matrix form, this transformation can be expressed as

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

Equivalent Decryption Structure—As could be observed from Fig. 2 and Fig. 5(a), the sequences of transformations in the encryption structure and the straightforward decryption structure are totally different from each other. This difference can be an obstacle in resource sharing between the implementation of encryptor and decryptor. However, two properties of the AES algorithm allow for making the decryption structure have the same sequence of transformations as that in encryption with minor changes in the roundkeys [2].

- InvShiftRows transformation immediately followed by a InvSubBytes



(a) Straightforward Decryption Structure

(b) Equivalent Decryption Structure

Figure 5. Decryption structures of the AES algorithm.

transformation is equivalent to InvSubBytes transformation immediately followed by a InvShiftRows transformation.

- InvMixColumns transformation is linear, which means:

InvMixColumns(State XOR roundkey) =
InvMixColumns(State) XOR InvMixColumns(roundkey).

This allows for the exchange of InvMixColumns and AddRoundKey transformations if roundkeys are modified by InvMixColumns transformation before they are added up in the AddRoundKey transformation.

Taking these two properties into consideration, the structure of decryption can be modified to the equivalent structure in Fig. 5(b), in which mixroundkey(i) stands for the modified roundkey(i). The equivalent decryption structure has exactly the same sequence of transformations as that in the encryption structure; this feature enables more efficient implementations of joint encryptor/decryptor.

Architectural Optimization

A block cipher encrypts plain text in fixed-size n-bit blocks (n = 128 for

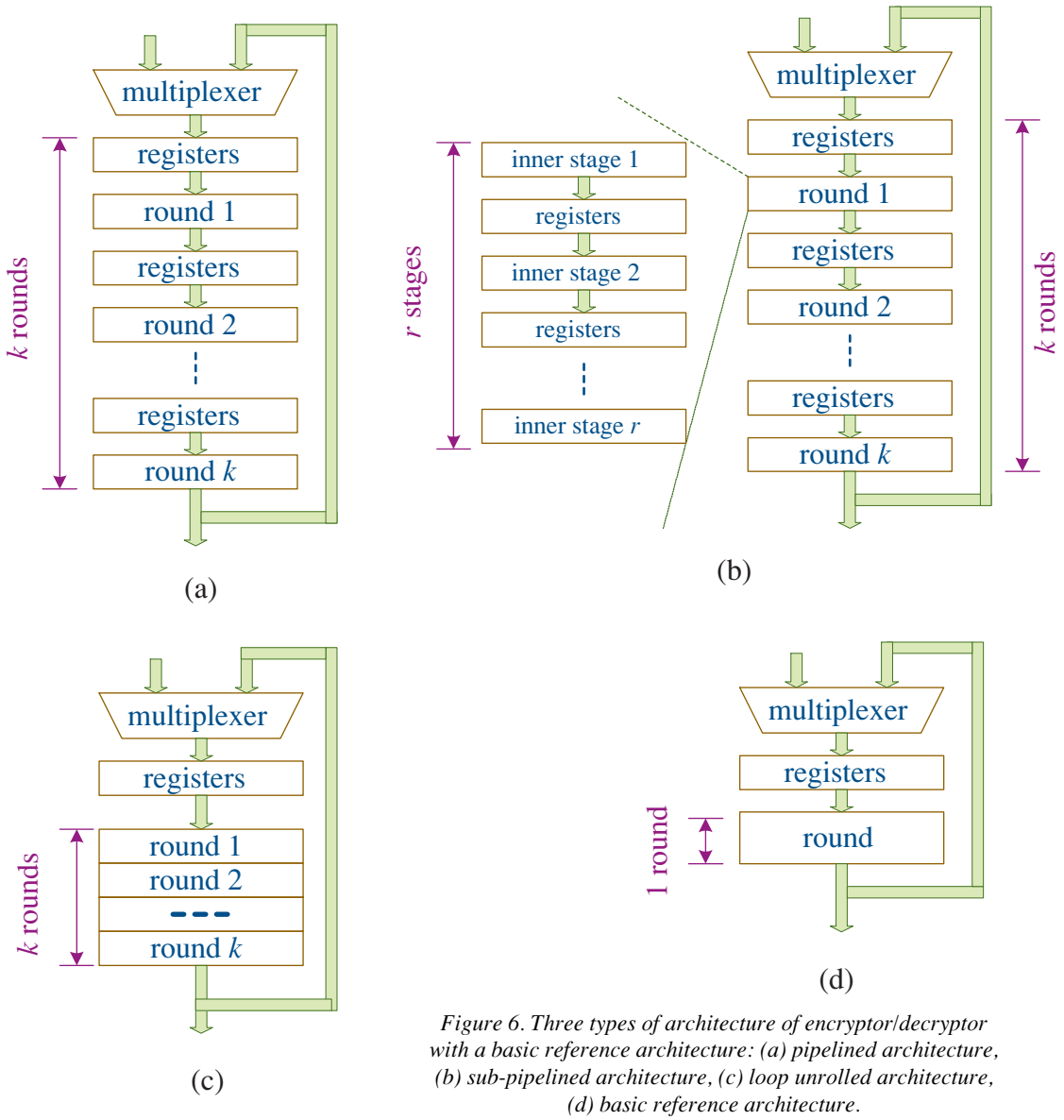


Figure 6. Three types of architecture of encryptor/decryptor with a basic reference architecture: (a) pipelined architecture, (b) sub-pipelined architecture, (c) loop unrolled architecture, (d) basic reference architecture.

Implementation Approaches for the AES Algorithm

AES). Messages longer than n bits are divided into n-bit blocks, and each block is encrypted separately. Basically, there are four modes of operation: electronic-codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB) and output feedback (OFB). Non-feedback (NFB) modes such as the ECB mode offer less security, but can achieve great speedup by processing multiple blocks simultaneously. The other three basic modes belong to feedback (FB) mode, which can offer a higher level of security but can hardly achieve any speedup by multi-block processing due to the existence of feedback—the processing of the next block cannot begin until the current block is finished.

Three types of architectures can be used to increase the speed of encryptor/decryptor by duplicating hardware for implementing each round, which is also called round unit in this paper. These architectures are based on pipelining, sub-pipelining, and loop unrolling. They are illustrated in Fig. 6 together with a basic reference architecture.

- **Pipelining**

  The pipelined architecture can increase the speed of encryption/decryption by processing multiple blocks of data simultaneously. It is realized by inserting rows of registers among combinational logic. Parts of logic between two consecutive registers form pipeline stages. Each pipeline stage is one round unit in this case. During each clock cycle, the partially processed data moves to the next stage and its place is taken by the subsequent data block. The number of round units in each loop, k, is usually chosen as a divisor of Nr and the maximum value of k is Nr, in which case it becomes a fully pipelined architecture. For a k-round pipelined architecture, when a partially processed block reaches the kth round, it will be fed back to the first round until all the Nr rounds are performed on this block. After the pipeline reaches its full depth, that is after the first block reaches the kth stage, k blocks of data are processed simultaneously in different stages and k blocks of data are processed every Nr cycles. The area of the pipelined architecture is proportional to k.

- **Sub-Pipelining**

  Similar to the pipelining, sub-pipelining also inserts rows of registers among combinational logic, but in this case, registers are inserted both between and inside each round unit. If each round unit can be divided into r stages with equal delay, a k-round sub-pipelined architecture can achieve approximately r times the speed of a k-round pipelined architecture with a slight increase of area caused by additional registers and control logic. However, dividing each round unit into an arbitrary number of stages does not always bring speedup. Since the minimum clock period is decided by the indivisible combinational element with the longest

Depending on different optimization criteria, different architectures can be employed. Optimization for maximum speed can be realized by a fully sub-pipelined architecture. In the application requiring minimum area, the basic architecture is desired. In the case of optimum speed/area ratio, sub-pipelining seems to be the best choice.

delay, dividing the rest of the round unit into more stages 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 is increased by the same proportion. Therefore the overall speed does not improve despite increased area caused by the additional registers.

- **Loop Unrolling**

  Loop unrolled or unfolded architectures can process only one block of data at a time, but multiple rounds are performed in each clock cycle. The unrolling or unfolding factor, k, is usually chosen as a divisor of Nr and the maximum value of k is Nr. The number of cycles to process one block of data is Nr / k in this case. Meanwhile, the clock period of a k-round loop unrolled architecture is increased to slightly smaller than k times the clock period of a pipelined architecture because of the setup time and propagation delay of registers. The area of this architecture is also proportional to the number of rounds in each loop.

Most of the proposed implementations can be classified into one of the above three architectures. Detailed studies of all these architectures were carried out in [3] and [4]. In this section, we separately address the speedup factor of these three architectures for FB and NFB modes compared to the basic reference architecture.

Architectural Optimization for Non-Feedback Modes

The speed of a system can be measured by throughput, which is given by

throughput = average number of bits processed/second.

In the case of the AES algorithm, it can also be expressed as

throughput = 128/(average number of clock cycles to process one block x clock period).(4)

Maximum achievable throughput for each architecture is compared in this section. In the basic architecture in Fig. 6(d), only one round is performed in each clock cycle, so Nr clock cycles are needed to finish processing one block of data. The minimum clock period t_{basic} can be expressed as

$$t_{basic} = t_{round} + t_{setup} + t_{prop} + t_{mux}. \quad (5)$$

In (5), t_{round} is the delay of the combinational logic in each round unit, t_{mux} denotes the delay of the multiplexer, whereas t_{setup} and t_{prop} stand for the setup time and propagation delay of the registers, respectively. From (4), the maximum achievable throughput of the basic architecture is given by

$$throughput_{basic} = 128 / (Nr \times t_{basic}).$$

In the pipelined architecture in Fig. 6(a), assuming k is a divisor of Nr, after the initial k clock cycles, k blocks of data are processed every Nr cycles. Meanwhile, the minimum clock period is the same as that of the basic architecture. The speedup of pipelined architecture over the basic architecture is

$$\text{throughput}_{\text{pipe}} / \text{throughput}_{\text{basic}} = k.$$

The area of this architecture is proportional to the number of pipeline stages, k. Trade-offs between area and speed can be easily made by changing k.

In the sub-pipelined architecture of the AES algorithm, each of the round units should be divided into no more than two stages according to the former discussion in this section. SubBytes/InvSubBytes is usually implemented by look-up tables. ShiftRows/InvShiftRows does not need any logic to implement, MixColumns/InvMixColumns can be implemented by XOR gates, and AddRoundKey is only one step of XOR operation. Hence each round unit is usually divided into $r = 2$ stages, one for SubBytes/InvSubBytes transformation and another for the rest of the transformations. Assuming the two stages in each round have equal delay, $2k$ blocks of data will be processed every $2Nr$ cycles after the pipeline reaches its full depth. Let $\tau = (t_{\text{setup}} + t_{\text{prop}} + t_{\text{mux}}) / t_{\text{round}}$. The minimum clock period of sub-pipelining is $(0.5 + \tau) / (1 + \tau)$ times that of the basic architecture. The speed up of a k-round sub-pipelined architecture with $r = 2$ is given by

$$\text{throughput}_{\text{sub-pipe}} / \text{throughput}_{\text{basic}} = k\,(1 + \tau) / (0.5 + \tau). \quad (6)$$

The area of sub-pipelined architecture is also proportional to the parameter k but does not change much with r. Increasing number of inner round stages only introduces more registers, whose area is small compared to the total area of implementation. The throughput of this architecture is $(1 + \tau) / (0.5 + \tau)$ times that of a pipelined architecture with the same k. Usually τ is small, so there is almost twice speedup over pipelining at the cost of $r - 1$ additional rows of registers.

In the loop unrolled architecture in Fig. 6(c), assuming k is a divisor of Nr, one block of data is processed every Nr / k cycles. However, the minimum clock period is increased to

$$t_{lu} = k \times t_{\text{round}} + t_{\text{setup}} + t_{\text{prop}} + t_{\text{mux}} \quad (7)$$

which is $(k + \tau) / (1 + \tau)$ times the minimum clock period of the basic architecture. Hence the speedup of a k-round loop unrolled architecture can be expressed as

$$\text{throughput}_{lu} / \text{throughput}_{\text{basic}} = (1 + \tau) / (1 + \tau / k). \quad (8)$$

The area of loop unrolled architecture is also proportional to the number of rounds per loop, k. Compared to the k-stage pipelined architecture, the speedup is much lower at roughly the same area.

Depending on different optimization criteria, different architectures can be employed. Optimization for maximum speed can be realized by a fully sub-pipelined architecture. In the application requiring minimum area, the basic architecture is desired. In the case of optimum speed/area ratio, sub-pipelining seems to be the best choice. Numerous implementations of these architectures on different technologies have been studied. The reported fastest FPGA implementation can reach 12 Gbit/sec on a Xilinx Virtex-E XCV812E-8BG560 device for a fully pipelined 128-bit key encryptor in NFB modes [5].

Xinmiao Zhang received the B.S. and the M.S. in electrical engineering from Tianjin University, Tianjin, China, in 1997 and 2000, respectively. She is currently a doctoral student with the Department of Electrical and Computer Engineering at the University of Minnesota-Twin Cities. Her research interests include efficient VLSI architectures for cryptography algorithms and soft-decision Reed-Solomon codes.

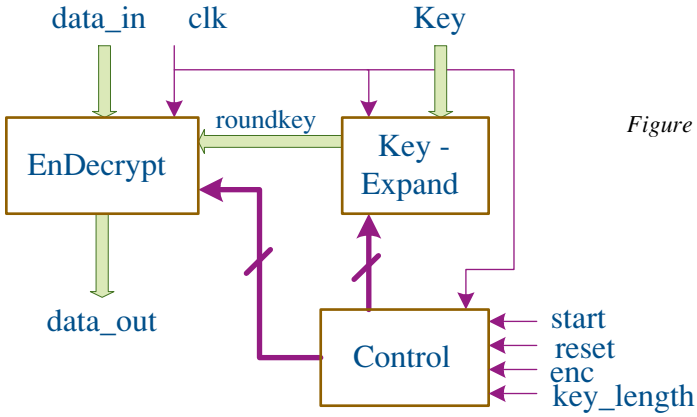


Figure 7. Block diagram of the AES system.

Architectural Optimization for Feedback Mode

In feedback modes, the encryption/decryption of the next block cannot start until the current block is finished. In this case, pipelining does not lead to any speedup, because only one stage is processing one block of data in each cycle, while the other stages are idle. Meanwhile, the area increases proportionally to *k*. Therefore pipelined architecture is not suitable for feedback applications. Loop-unrolled architecture, however, can bring some speedup at the cost of significantly increased area. The speedup which can be achieved is the same as that in non-FB modes given by (8). Sub-pipelining can even deteriorate the performance; $Nr \times r$ cycles are needed to encrypt/decrypt one block of data, but even in the optimum case when each inner stage has equal delay, the clock period is longer than t_{basic} / r because of the setup and propagation delay of the registers. The fastest implementation for FB modes reported so far employed a fully loop unrolled architecture, and achieved a throughput of 1950.03 Mbit/sec based on Mitsubishi Electric's 0.35 micron CMOS technology [6].

Algorithmic Optimization

A complete AES system can be divided into three major blocks: KeyExpand, Control, and EnDecrypt, as illustrated in Fig. 7. The Key-Expand block loads keys, performs Key Expansion transformation, and generates proper roundkeys under the control signals from the Control block. Control block takes 'start' signal, 'reset' signal, 'enc' signal, and 'key_length' signal from outside and generates all the control signals for the whole system. The 'enc' signal and the 'key_length' signal are optional. The 'enc' signal is the control signal for encryption/decryption; it is needed when the system can perform both encryption and decryption. The 'key_length' signal gives the key length information; it is needed when the system can perform multiple key length encryption/decryption. The EnDecrypt block gets roundkeys from the Key-Expand block and encrypts/decrypts 'data_in' according to the AES algorithm. Each of the architectures—pipelining, loop unrolling, and sub-pipelining—covered in the last section can be used in the EnDecrypt block. The speed and area trade-offs of the AES algorithm can not only be made by changing the overall architecture of the EnDecrypt block, but also by exploiting the implementation of each round unit. A variety of methods have been brought up to implement individual round unit [5, 7–11]. They are discussed in detail in this section.

Implementation of Separate Transformations

No optimization can be performed on ShiftRows/InvShiftRows and AddRoundKey transformations, since

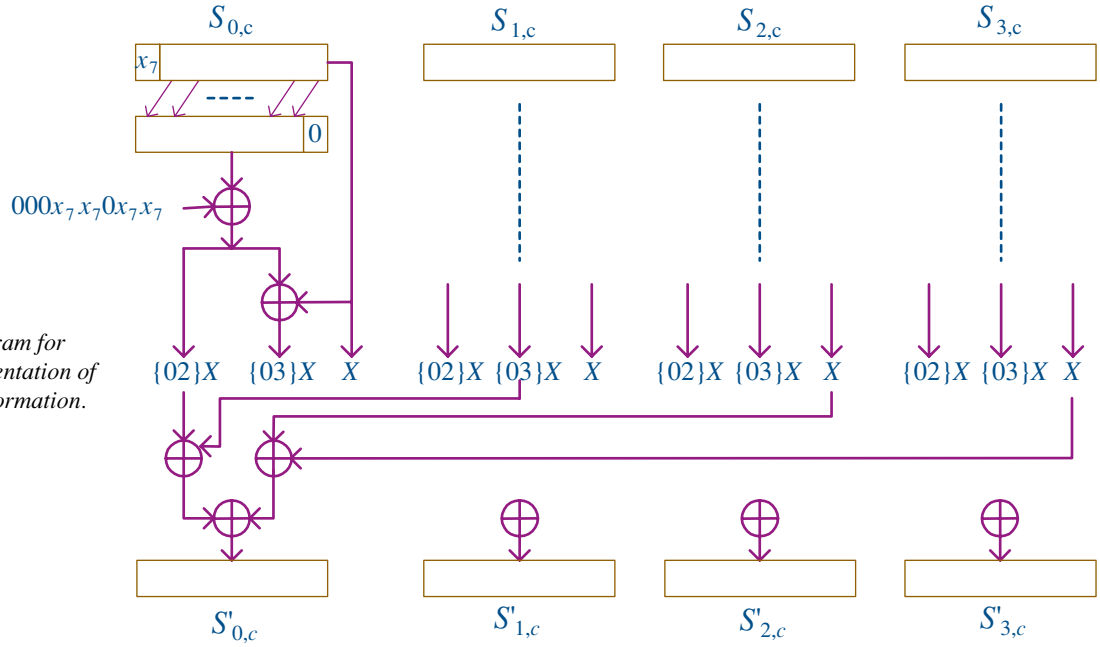


Figure 8. Block diagram for straightforward implementation of the MixColumns transformation.

no logic gates are needed for the former transformation and only one step of XOR operation is needed for the latter. However, different methods can be used to implement the SubBytes/InvSubBytes and MixColumns/InvMixColumns transformations.

Implementation of SubBytes/InvSubBytes—SubBytes/InvSubBytes is usually implemented by look-up tables. Each S-box/S^{-1}-box needs a look-up table of 256 x 8 = 2k-bits and each round needs 16 S-boxes/S^{-1}-boxes, so the area for look-up tables becomes huge when multiple round units are implemented. For area critical applications, a better choice is to map the arithmetic operations on GF(2^8) to isomorphic field GF((2^4)^2). This implementation requires smaller area for look-up tables, but has longer delay [11, 12].

Implementation of MixColumns/InvMixColumns—In the MixColumns transformation, we need to implement constant multiplication of {02} and {03} in GF(2^8). Assuming X is a byte in the State, $\{02\}X$ can be implemented by shifting and bit-wise XOR operations, and $\{03\}X$ can be computed by $(\{02\}X) \oplus X$, where $\oplus$ denotes Exclusive-OR. If X is expressed in binary form as $\{x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0\}$, $\{02\}X$ can be calculated by

$$\{02\}X = \{x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0\} \oplus \{0, 0, 0, x_7, x_7, 0, x_7, x_7\}. \quad (9)$$

Since $0 \oplus x_i = x_i$, (9) only needs 4 XOR gates to implement. The block diagram in Fig. 8 shows the straightforward way to calculate $S'_{0,c}$ $(0 \le c < 4)$ in the MixColumns transformation [8]. Since $\{01\}X = X$, X instead of $\{01\}X$ is used in this and the following figures. Calculation of $S'_{1,c}$, $S'_{2,c}$, and $S'_{3,c}$ can be done by connecting appropriate $\{02\}X$, $\{03\}X$, or X of $S'_{1,c}$, $S'_{2,c}$, and $S'_{3,c}$ to the last row of XOR gates in Fig. 8 according to (2). As shown in the figure, the critical path has 4 XOR gates and a total of (4 x 8 + 4) x 4 = 144 2-input XOR gates is needed to implement the MixColumns transformation for one column of the State.

The InvMixColumns transformation is more complicated. Constant multiplications used in the InvMixColumns transformation can be expressed as

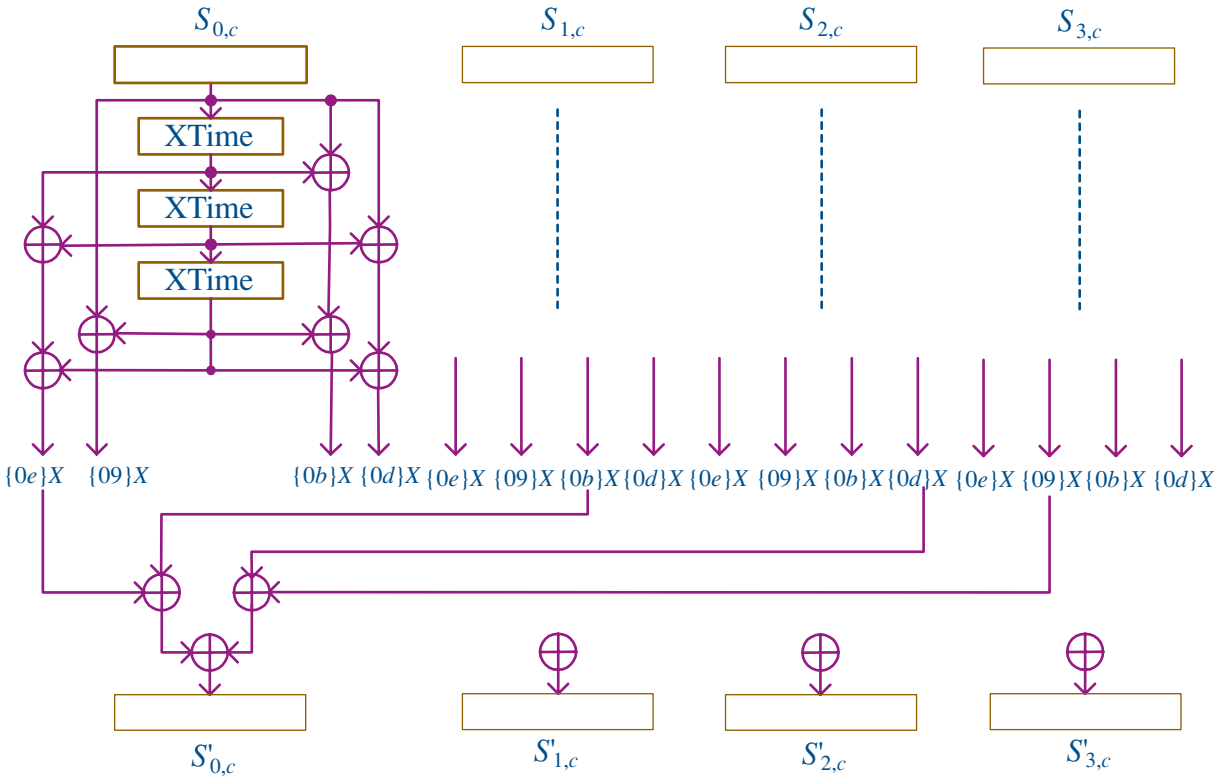


Figure 9. Block diagram for straight forward implementation of the InvMixColumns transformation.

$$\{0b\}X = \{08\}X \oplus \{02\}X \oplus X, \quad \{0d\}X = \{08\}X \oplus \{04\}X \oplus X,$$
$$\{09\}X = \{08\}X \oplus X, \quad \{0e\}X = \{08\ \}X\{ \oplus \{04\}X \oplus \{02\}X.$$

A straightforward way to calculate $S'_{0,c}$ $(0 \le c < 4)$ of the State in the InvMixColumns transformation is illustrated in Fig. 9. In order to simplify the diagram, an XTime block is introduced as shown in Fig. 10. XTime block implements the constant multiplication by {02} in GF(2^8) [1], each XTime block consists of 4 XOR gates and the critical path includes only one XOR gate. In the InvMixColumns transformation, the calculation for the other bytes can be carried out similarly according to (3). As shown in Fig. 9, the critical path is 6 XOR gates, and a total of (10 x 8 + 3 x 4) x 4 = 368 XOR gates is needed to implement the MixColumns transformation for one column of the State.

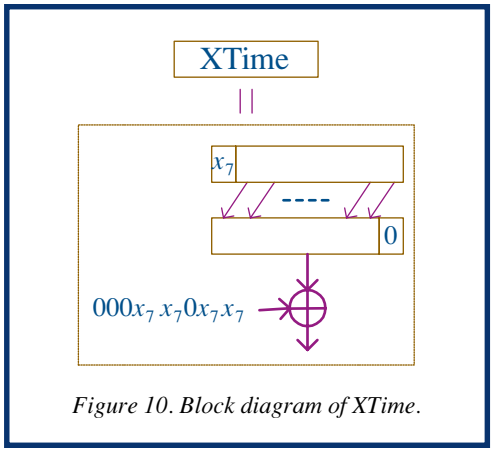


Figure 10. Block diagram of XTime.

Studies in [7, 9] and [10] have proposed alternative ways to implement the MixColumns/InvMixColumns transformation. Both the studies in [7] and [10] exploited the idea of substructure sharing. In [7]'s study, taking the bytes in the first row of the State for example,

$$S'_{0,c} = [\{02\}\ \{03\}\ \{01\}\ \{01\}] \times [S_{0,c}\ S_{1,c}\ S_{2,c}\ S_{3,c}]^T$$

can be rewritten as

$$S'_{0,c} = \{02\}(S_{0,c} + S_{1,c}) + S_{1,c} + (S_{2,c} + S_{3,c}), \quad (10)$$

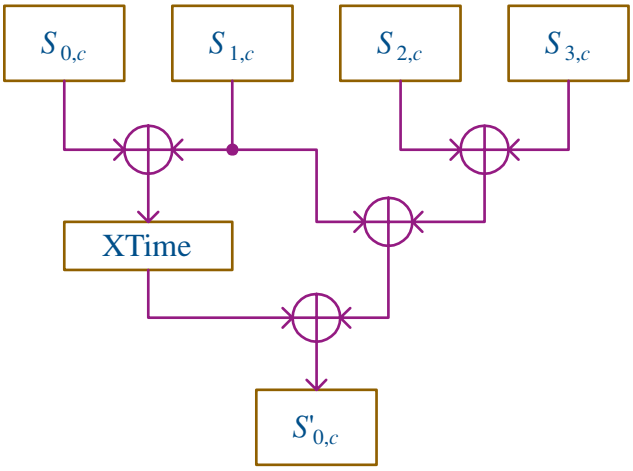


Figure 11. Block diagram for substructure sharing implementation of MixColumns transformation.

which can be realized as shown in Fig. 11. The computation of the other bytes in the State can be implemented by similar structure and the same number of XOR gates. Compared to Fig. 8, the total number of XOR gates for computing one column of the State remains 144, but the critical path has been reduced to 3 XOR gates.

The same substructure sharing idea can be used in the InvMixColumns transformation. For example, the bytes in row one of the State are calculated by

$$S'_{0,c} = [\{0e\}\ \{0b\}\ \{0d\}\ \{09\}] \times [S_{0,c}\ S_{1,c}\ S_{2,c}\ S_{3,c}]^T, \quad (11)$$

which can be rewritten as

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

(12) can be implemented as shown in Fig. 12. Compared to the straightforward implementation in Fig. 9, the number of XOR gates is reduced to (8 x 8 + 4 x 4) x 4 = 320 for computing one column of the State, but the critical path is increased to 7 XOR gates.

Substructure sharing can be used in another way as in [10]. Since

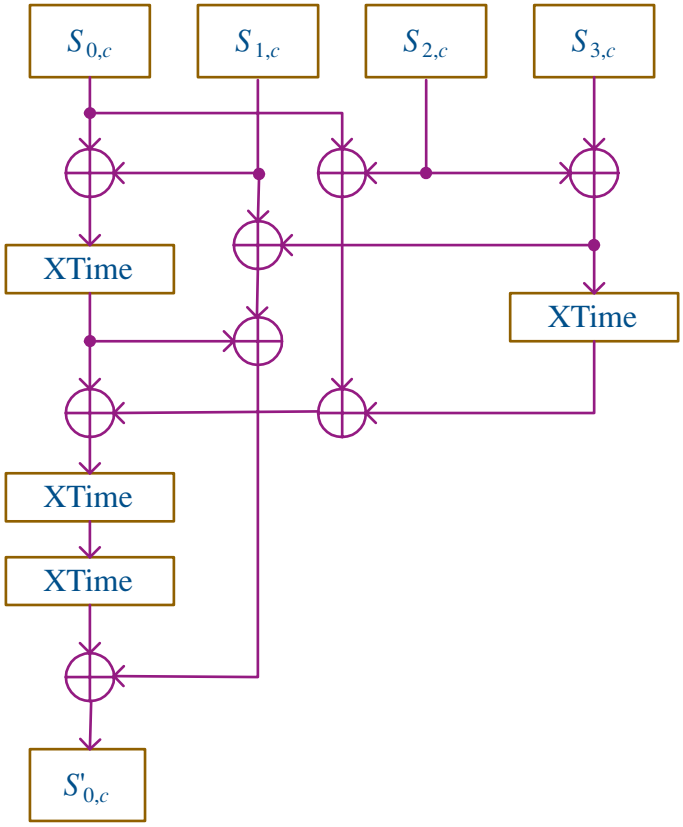


Figure 12. Block diagram for substructure sharing implementation of the InvMixColumns transformation.

$$\begin{aligned} \{0b\}X &= (\{08\}X \oplus X) \oplus \{02\}X, \\ \{0d\}X &= (\{08\}X \oplus X) \oplus \{04\}X, \\ \{09\}X &= \{08\}X \oplus X, \\ \{0e\}X &= \{08\}X \oplus \{04\}X \oplus \{02\}X. \end{aligned}$$

Three of the above equations have the common factor $\{08\}X \oplus X$. Therefore, hardware usage can be reduced by first calculating the common factor and then using it in the calculation of the other equations. This approach leads to the implementation shown on the following page in Fig. 13. In this figure, the critical path still has 7 XOR gates, but the total number of XOR gates has been reduced further to (8 x 8 + 3 x 4) 4 = 304 for calculation of one column of the State.

Another approach in [9, 10] makes use of the polynomial notation of elements in GF(2^8) with irreducible polynomial $m(\alpha) = \alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1$. For example, {03} can be expressed as

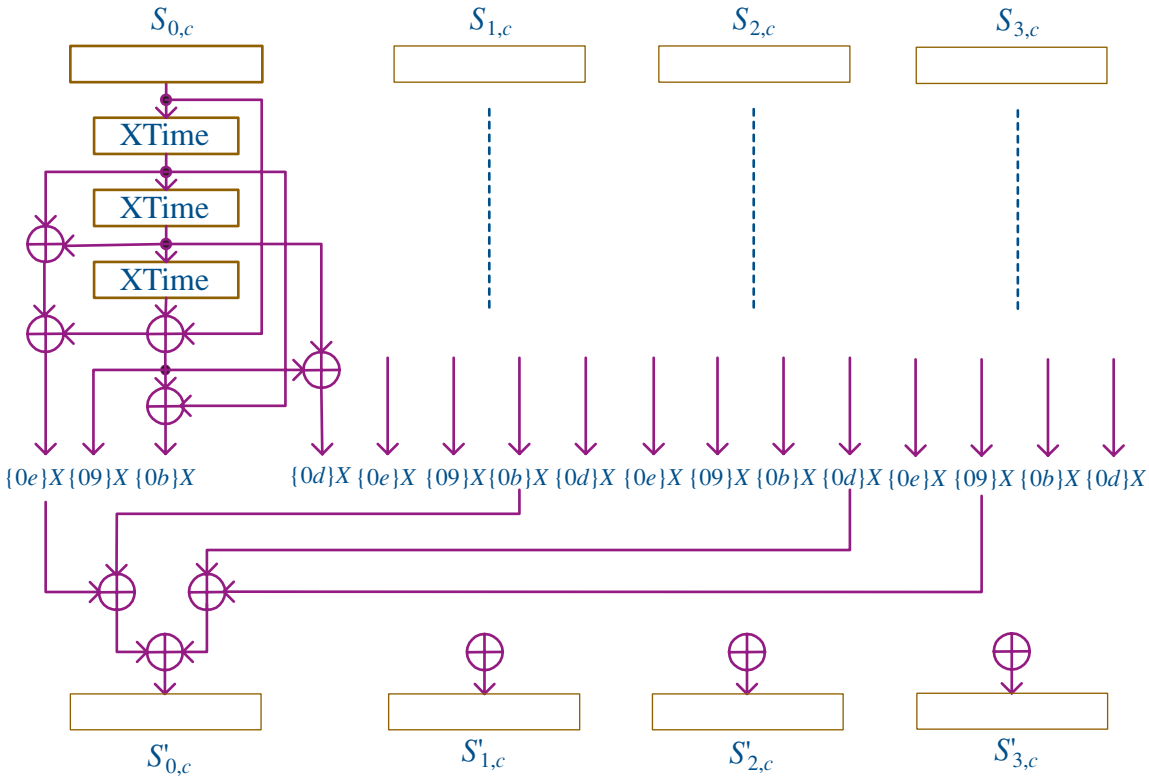


Figure 13. Block diagram for alternative substructure sharing implementation of the InvMixColumns transformation.

polynomial $\alpha+1$, $Y=\{03\}X=(\alpha+1)(x_7\alpha^7+x_6\alpha^6+x_5\alpha^5+x_4\alpha^4+x_3\alpha^3+x_2\alpha^2+x_1\alpha+x_0) \bmod m(\alpha)$, then each bit of Y can be expressed as

$$y_7=x_7\oplus x_6,\quad y_6=x_6\oplus x_5,\quad y_5=x_5\oplus x_4,\quad y_4=x_7\oplus x_4\oplus x_3,$$
$$y_3=x_7\oplus x_3\oplus x_2,\quad y_2=x_2\oplus x_1,\quad y_1=x_7\oplus x_1\oplus x_0,\quad y_0=x_7\oplus x_0.$$

Similarly, all the constant multiplications used in the MixColumns and InvMixColumns transformations can be calculated by the equations in Table I.

Studies in [9, 10] did not further investigate the individual bit calculation of constant multiplication. Direct implementation of the equations in Table I does not bring any area or critical path reduction. However, the idea

Table I. Individual Bit Expression for Constant Multiplications

	$\{02\}X$	$\{03\}X$	$\{09\}X$	$\{0b\}X$	$\{0d\}X$	$\{0e\}X$
y_7	x_6	$x_6\oplus x_7$	$x_4\oplus x_7$	$x_4\oplus x_6\oplus x_7$	$x_4\oplus x_5\oplus x_7$	$x_4\oplus x_5\oplus x_6$
y_6	x_5	$x_5\oplus x_6$	$x_3\oplus x_6\oplus x_7$	$x_3\oplus x_5\oplus x_6\oplus x_7$	$x_3\oplus x_4\oplus x_6\oplus x_7$	$x_3\oplus x_4\oplus x_5\oplus x_7$
y_5	x_4	$x_4\oplus x_5$	$x_2\oplus x_5\oplus x_6\oplus x_7$	$x_2\oplus x_4\oplus x_5\oplus x_6\oplus x_7$	$x_2\oplus x_3\oplus x_5\oplus x_6$	$x_2\oplus x_3\oplus x_4\oplus x_6$
y_4	$x_3\oplus x_7$	$x_3\oplus x_4\oplus x_7$	$x_1\oplus x_4\oplus x_5\oplus x_6$	$x_1\oplus x_3\oplus x_4\oplus x_5\oplus x_6\oplus x_7$	$x_1\oplus x_2\oplus x_4\oplus x_5\oplus x_7$	$x_1\oplus x_2\oplus x_3\oplus x_5$
y_3	$x_2\oplus x_7$	$x_2\oplus x_3\oplus x_7$	$x_0\oplus x_3\oplus x_5\oplus x_7$	$x_0\oplus x_2\oplus x_3\oplus x_5$	$x_0\oplus x_1\oplus x_3\oplus x_5\oplus x_6\oplus x_7$	$x_0\oplus x_1\oplus x_2\oplus x_5\oplus x_6$
y_2	x_1	$x_1\oplus x_2$	$x_2\oplus x_6\oplus x_7$	$x_1\oplus x_2\oplus x_6\oplus x_7$	$x_0\oplus x_2\oplus x_6$	$x_0\oplus x_1\oplus x_6$
y_1	$x_0\oplus x_7$	$x_0\oplus x_1\oplus x_7$	$x_1\oplus x_5\oplus x_6$	$x_0\oplus x_1\oplus x_5\oplus x_6\oplus x_7$	$x_1\oplus x_5\oplus x_7$	$x_0\oplus x_5$
y_0	x_7	$x_0\oplus x_7$	$x_0\oplus x_5$	$x_0\oplus x_5\oplus x_7$	$x_0\oplus x_5\oplus x_6$	$x_5\oplus x_6\oplus x_7$

Table II. Substructure Sharing in Individual Bit Calculation for the MixColumns Transformation after the First Round

	{02}A	{03}A
y_7	x_6	$x_6 \oplus x_7$
y_6	x_5	$x_5 \oplus x_6$
y_5	x_4	$x_4 \oplus x_5$
y_4	$x_3 \oplus x_7$	$x_3 \oplus x_4 \oplus x_7$
y_3	$x_2 \oplus x_7$	$x_2 \oplus x_3 \oplus x_7$
y_2	x_1	$x_1 \oplus x_2$
y_1	x_8	$x_1 \oplus x_8$
y_0	x_7	x_8

Table III. Substructure Sharing in Individual Bit Calculation for the MixColumns Transformation after the Second Round

	{02}A	{03}A
y_7	x_6	$x_6 \oplus x_7$
y_6	x_5	$x_5 \oplus x_6$
y_5	x_4	$x_4 \oplus x_5$
y_4	x_9	$x_4 \oplus x_9$
y_3	$x_2 \oplus x_7$	$x_2 \oplus x_9$
y_2	x_1	$x_1 \oplus x_2$
y_1	x_8	$x_1 \oplus x_8$
y_0	x_7	x_8

of substructure sharing can be also applied to the calculation of individual bits. After making modifications to the algorithm in [13], the following algorithm is derived to find the substructures that can be shared in the constant multiplications.

1. round = 0.
2. For $i, j = 0$ to 7 + round, count the number of times $x_i \oplus x_j$ appears in all the equations, denote the number by $N(i, j)$. Find the biggest number $N(m, n)$. If there is a tie, pick one at random.
3. if $N(m, n) > 1$, then replace $x_m \oplus x_n$ in all those equations with $x_{7 + \text{round}}$, otherwise STOP.
4. round = round + 1, go to step 2.

For example, in the MixColumns transformation, we need to calculate the 16 equations in the second and third columns in Table I for each byte in the State. The biggest number of times $x_i \oplus x_j$ appears is three when $(i = 0, j = 7)$ or $(i = 3, j = 7)$. We pick $x_0 \oplus x_7$ randomly and replace $x_0 \oplus x_7$ with x_8 in all equations, then the second and third columns in Table I become Table II. In the next round, the biggest number of $x_i \oplus x_j$ in common in all the equations in Table II is three, when $(i = 3, j = 7)$. Replacing $x_3 \oplus x_7$ with x_9 in Table II, Table III is derived. In the third round, the biggest $N(i, j)$ is 1, so the algorithm stops and Table III is the final table. MixColumns can be implemented by first computing x_8 and x_9, then using them in the computing of other equations according to Table III. Fig. 14 illustrates this implementation.

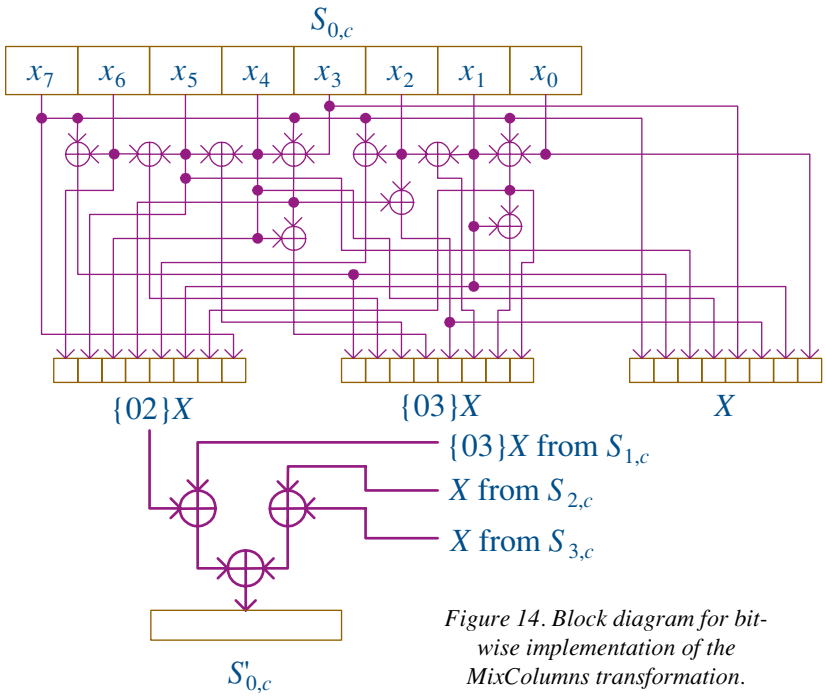


Figure 14. Block diagram for bit-wise implementation of the MixColumns transformation.

Table IV. Substructure Sharing in Individual Bit Calculation for the InvMixColumns Transformation

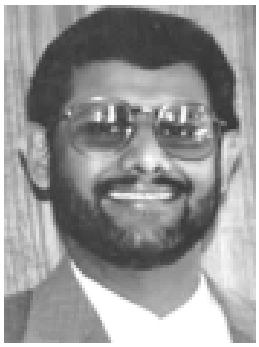
	$\{09\}X$	$\{0b\}X$	$\{0d\}X$	$\{0e\}X$
y_7	x_9	$x_6 \oplus x_9$	$x_5 \oplus x_9$	$x_4 \oplus x_8$
y_6	$x_6 \oplus x_{13}$	$x_8 \oplus x_{13}$	$x_6 \oplus x_{17}$	$x_5 \oplus x_{17}$
y_5	$x_7 \oplus x_{18}$	$x_9 \oplus x_{18}$	$x_8 \oplus x_{12}$	$x_4 \oplus x_6 \oplus x_{12}$
y_4	$x_4 \oplus x_{10}$	$x_{10} \oplus x_{17}$	$x_2 \oplus x_9 \oplus x_{15}$	$x_{12} \oplus x_{15}$
y_3	$x_{11} \oplus x_{13}$	$x_{11} \oplus x_{12}$	$x_{13} \oplus x_{14}$	$x_2 \oplus x_{14}$
y_2	x_{19}	$x_1 \oplus x_{19}$	$x_0 \oplus x_{16}$	$x_0 \oplus x_1 \oplus x_6$
y_1	x_{10}	$x_7 \oplus x_{14}$	$x_7 \oplus x_{15}$	x_{11}
y_0	x_{11}	$x_7 \oplus x_{11}$	$x_0 \oplus x_8$	$x_7 \oplus x_8$

where

$x_8 = x_5 \oplus x_6$	$x_9 = x_4 \oplus x_7$
$x_{10} = x_1 \oplus x_8$	$x_{11} = x_0 \oplus x_5$
$x_{12} = x_2 \oplus x_3$	$x_{13} = x_3 \oplus x_7$
$x_{14} = x_0 \oplus x_{10}$	$x_{15} = x_1 \oplus x_5$
$x_{16} = x_2 \oplus x_6$	$x_{17} = x_3 \oplus x_9$
$x_{18} = x_2 \oplus x_8$	$x_{19} = x_7 \oplus x_{16}$

The critical path remains 4 XOR gates as in Fig. 8, but the total number of XOR gates to calculate on a column of the State has been reduced to 4 x (10 + 3 x 8) = 136.

Applying the same algorithm to the equations in the last four columns in Table I, we get Table IV as the final table for substructure sharing in the InvMixColumns transformation. According to this table, the critical path of the InvMixColumns transformation can only have 6 XOR gates if tree adders are used. At the same time, the total number of XOR gates to calculate one column of the State has been reduced to (30 + 12 + 24) x 4 = 264.



Look-Up Table Implementation of the Whole Round Unit

Look-up tables not only can be used to implement the SubBytes/InvSubBytes transformation, they can also be used to incorporate MixColumns/InvMixColumns transformation [1, 5, 9, 14]. The *T*-box approach implements the combination of SubBytes, ShiftRows and MixColumns transformations by look-up tables. Beginning from the SubBytes transformation, the updated State after the MixColumns transformation can be expressed as, for $0 \le c < Nb$,

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{bmatrix} \begin{bmatrix} \text{SubBytes}(S_{0,c}) \\ \text{SubBytes}(S_{1,c+1}) \\ \text{SubBytes}(S_{2,c+2}) \\ \text{SubBytes}(S_{3,c+3}) \end{bmatrix}. \quad (13)$$

Instead of storing only the value of $\text{SubBytes}(S_{i,j})$ in the *S*-box approach, the *T*-box approach stores values of $\text{SubBytes}(S_{i,j})$, $\{02\}\text{SubBytes}(S_{i,j})$ and $\{03\}\text{SubBytes}(S_{i,j})$. Each *T*-box has three 8-bit outputs and can be expressed as

$$T(S_{i,j}) = \begin{bmatrix} T_1(S_{i,j}) \\ T_2(S_{i,j}) \\ T_3(S_{i,j}) \end{bmatrix} = \begin{bmatrix} \text{SubBytes}(S_{i,j}) \\ \{02\}\text{SubBytes}(S_{i,j}) \\ \{03\}\text{SubBytes}(S_{i,j}) \end{bmatrix},$$

for $0 \le i, j < 4$. Now (13) can be rewritten as

Keshab K. Parhi is Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering at the University of Minnesota. His research interests span the areas of VLSI architectures for digital signal and image processing, adaptive digital filters and equalizers, error control coders, cryptography architectures, high-level architecture transformations and synthesis, low-power digital systems, and computer arithmetic. He has over 350 papers in these areas, authored the text book "VLSI Digital Signal Processing Systems" (Wiley, 1999) and coedited the reference books "Digital Signal Processing for Multimedia Digital Signal Processing Systems" (Wiley, 1999) and "Digital Signal Processing for Multimedia Systems" (Marcel Dekker, 1999). His awards include the 2001 IEEE W.R.G. Baker paper prize, a 1999 Golden Jubilee medal from CASS, and the 2003 IEEE Kiyo Tomiyasu Technical Field award.

Implementation Approaches for the AES Algorithm

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} T_2(S_{0,c}) \oplus T_3(S_{1,c+1}) \oplus T_1(S_{2,c+2}) \oplus T_1(S_{3,c+3}) \\ T_1(S_{0,c}) \oplus T_2(S_{1,c+1}) \oplus T_3(S_{2,c+2}) \oplus T_1(S_{3,c+3}) \\ T_1(S_{0,c}) \oplus T_1(S_{1,c+1}) \oplus T_2(S_{2,c+2}) \oplus T_3(S_{3,c+3}) \\ T_3(S_{0,c}) \oplus T_1(S_{1,c+1}) \oplus T_1(S_{2,c+2}) \oplus T_2(S_{3,c+3}) \end{bmatrix} \quad 0 \le c < Nb. \qquad (14)$$

The combination of SubBytes, ShiftRows and MixColumns transformations can be implemented by XORing the outputs of T-boxes. In the final round of encryption, there is no MixColumns transformation, so S-box instead of T-box should be used. In the fully pipelined or fully loop unrolled architecture, this will not be a problem, since each round uses separate hardware. However, for other architectures in which one round unit is used to perform different rounds of encryption in different clock cycles, T-box cannot be simply replaced by S-box. Adding an additional S-box is a solution, but this will lead to extra area for look-up tables. Another solution is to extract S-box from T-box; S-box is exactly the T_1 output of a T-box [9]. The T-box implementation has shorter delay than the S-box approach. The delay of MixColumns is eliminated by adding a delay of 2 XOR gates if a tree adder is used to add up the four items in each row of the matrix on the right side of (14). Based on the same technology and assumptions, the T-box approach improves the speed of an encryptor from 7 Gbit/sec in [15] to 12 Gbit/sec in [5]. However, the price paid for shorter delay is the three-times-bigger look-up tables.

Correspondingly, T^{-1}-box can be used to implement the combination of InvSubBytes, InvShiftRows and InvMixColumns transformations. From the beginning of InvSubBytes transformation, the updated State after the InvMixColumns transformation can be expressed as

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{0e\} & \{0b\} & \{0d\} & \{09\} \\ \{09\} & \{0e\} & \{0b\} & \{0d\} \\ \{0d\} & \{09\} & \{0e\} & \{0b\} \\ \{0b\} & \{0d\} & \{09\} & \{0e\} \end{bmatrix} \begin{bmatrix} \text{InvSubBytes}(S_{0,c}) \\ \text{InvSubBytes}(S_{1,c+3}) \\ \text{InvSubBytes}(S_{2,c+2}) \\ \text{InvSubBytes}(S_{3,c+1}) \end{bmatrix} \qquad (15)$$

for $0 \le c < Nb$. Each T^{-1}-box stores four sets of values:

$\{09\}\text{InvSubBytes}(S_{i,j})$, $\{0b\}\text{InvSubBytes}(S_{i,j})$,
$\{0d\}\text{InvSubBytes}(S_{i,j})$, $\{0e\}\text{InvSubBytes}(S_{i,j})$.

Unlike T-box, each T^{-1}-box has four 8-bit outputs and is four times the size of an S-box, and can be expressed as

$$T^{-1}(S_{i,j}) = \begin{bmatrix} T_0^{-1}(S_{i,j}) \\ T_1^{-1}(S_{i,j}) \\ T_2^{-1}(S_{i,j}) \\ T_3^{-1}(S_{i,j}) \end{bmatrix} = \begin{bmatrix} \{09\}\text{InvSubBytes}(S_{i,j}) \\ \{0b\}\text{InvSubBytes}(S_{i,j}) \\ \{0d\}\text{InvSubBytes}(S_{i,j}) \\ \{0e\}\text{InvSubBytes}(S_{i,j}) \end{bmatrix} \quad 0 \le i, j < 4.$$

Now (15) can be rewritten as

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} T_3^{-1}(S_{0,c}) \oplus T_1^{-1}(S_{1,c+3}) \oplus T_2^{-1}(S_{2,c+2}) \oplus T_0^{-1}(S_{3,c+1}) \\ T_0^{-1}(S_{0,c}) \oplus T_3^{-1}(S_{1,c+3}) \oplus T_1^{-1}(S_{2,c+2}) \oplus T_2^{-1}(S_{3,c+1}) \\ T_2^{-1}(S_{0,c}) \oplus T_0^{-1}(S_{1,c+3}) \oplus T_3^{-1}(S_{2,c+2}) \oplus T_1^{-1}(S_{3,c+1}) \\ T_1^{-1}(S_{0,c}) \oplus T_2^{-1}(S_{1,c+3}) \oplus T_0^{-1}(S_{2,c+2}) \oplus T_3^{-1}(S_{3,c+1}) \end{bmatrix} \qquad (16)$$

for $0 \le c < Nb$.

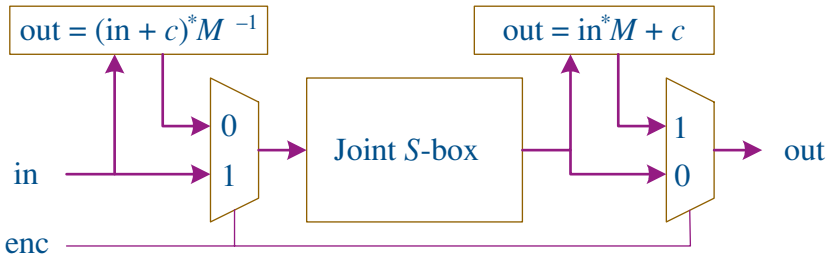


Figure 15. Joint implementation of the SubBytes and the InvSubBytes transformations.

Similar to the encryption case, S^{-1}-box needs to be used in the final round, but none of the outputs of T^{-1}-box is InvSubBytes($S_{i,j}$) this time. [9] brought up an efficient method to calculate InvSubBytes($S_{i,j}$) from the output of T^{-1}-box. For any specific $S_{i,j}$, each output byte of a T^{-1}-box can be expressed in binary form as $T_m^{-1}(S_{i,j}) = [t_{m7}, t_{m6}, t_{m5}, t_{m4}, t_{m3}, t_{m2}, t_{m1}, t_{m0}]$ ($m = 0, 1, 2, 3$), and InvSubBytes($S_{i,j}$) can be expressed in binary form as InvSubBytes($S_{i,j}$) = $[s_7^{-1}, s_6^{-1}, s_5^{-1}, s_4^{-1}, s_3^{-1}, s_2^{-1}, s_1^{-1}, s_0^{-1}]$. Since $\{09\}^{-1} = \{4f\}$, $\{0b\}^{-1} = \{c0\}$, $\{0d\}^{-1} = \{e1\}$, and $\{0e\}^{-1} = \{e5\}$,

$$\begin{aligned}\text{InvSubBytes}(S_{i,j}) &= \{4f\}T_0^{-1}(S_{i,j}) = \{c0\}T_1^{-1}(S_{i,j}) \\ &= \{e1\}T_2^{-1}(S_{i,j}) = \{e5\}T_3^{-1}(S_{i,j})). \quad (17)\end{aligned}$$

Each of the 8 bits of InvSubBytes($S_{i,j}$) can be computed as functions of individual bits in $T_m^{-1}(S_{i,j})$. Four sets of expressions of s_n^{-1} ($0 \le n < 8$) can be derived from (17). Expressions with the shortest delay are chosen for each s_n^{-1} from the four sets as shown in Table V. From Table V, at most two XOR gates are needed to compute each bit of InvSubBytes($S_{i,j}$) from T^{-1}-box. The critical path of the final round in T^{-1}-box approach consists of a look-up table, 2 XOR gates to extract the value of InvSubBytes($S_{i,j}$) and another XOR gate to add up the roundkey. The critical path of other round units includes a look-up table, 2 XOR gates to add up four outputs from different T^{-1}-boxes according to (15), using adder tree structure, and another XOR gate to add the roundkeys. We can observe that the delays of each round in a T^{-1}-box approach are the same, and equal the total delay of a look-up table and 3 XOR gates. Compared to the total delay of a look-up table and at least 6 XOR gates in the S^{-1}-box approaches, this approach has shorter delay but the price paid for that is the requirement of 4-times-bigger look-up tables of an S^{-1}-box approach, which makes pipelining or loop unrolling more expensive.

Implementation of Key Expansion

Roundkeys can either be generated beforehand and stored in memory or be generated on the fly. The former case is suitable for the applications which do not change keys constantly and can afford large area for memory. During encryption/decryption, roundkeys can be read out from memory by appropriate address, and there is no extra delay for decryption. In this case, reducing the critical path of Key Expansion can reduce the overhead, but will not speed up the whole system. While in the applications which need to change keys constantly, expanding keys on the fly is preferred. From Fig. 4, we can observe that the critical path of Key Expansion consists of one multiplexer, one S-box, and one XOR gate. Since the critical path of Key Expansion is shorter than that of a round unit, reducing the critical path of Key Expansion will not increase the speed

Table V. Extraction of S^{-1}-box from T^{-1}-box

$s_7^{-1} = t_{07} \oplus t_{04} \oplus t_{01}$	$s_6^{-1} = t_{06} \oplus t_{03} \oplus t_{00}$
$s_5^{-1} = t_{05} \oplus t_{02}$	$s_4^{-1} = t_{04} \oplus t_{01}$
$s_3^{-1} = t_{13} \oplus t_{12} \oplus t_{11}$	$s_2^{-1} = t_{36} \oplus t_{35} \oplus t_{30}$
$s_1^{-1} = t_{37} \oplus t_{35} \oplus t_{34}$	$s_0^{-1} = t_{15} \oplus t_{12} \oplus t_{10}$

of the whole system. Generating roundkeys on the fly eliminates the requirement for key storage, but brings overhead for decryption since decryption can only begin after the last roundkey is generated.

Joint Implementation Issues of Encryptor/Decryptor

Source sharing becomes important when only small area is available for implementing both encryptor and decryptor, as in smart cards and cellular phones. While the algorithmic strength in the last section can be exploited to reduce area, the design could be improved further by sharing the resources between encryptor and decryptor.

Joint Implementation of SubBytes and InvSubBytes

In [2], each S-box/S^{-1}-box requires a 2k-bit look-up table, and each round unit needs 32 such look-up tables to implement both encryption and decryption. However, studies in [1, 7] proposed that the SubBytes and InvSubBytes transformation can share a 2k-bit look-up table for each byte in the State. The SubBytes transformation can be expressed as

$$S'_{i,j} = M\, S_{i,j}^{-1} + c, \qquad (18)$$

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

and $c = [0\ 1\ 1\ 0\ 0\ 0\ 1\ 1]$.

The inverse of (18) is given by

$$S_{i,j} = (M^{-1}(S'_{i,j} + c))^{-1}. \qquad (19)$$

From (18) and (19), the SubBytes and InvSubBytes transformations can share look-up tables which only implement multiplicative-inverse in GF(2^8). Figure 15 illustrates the block diagram for a joint SubBytes and InvSubBytes transformation [7]. The Joint S-box block is a look-up table which stores the value of multiplicative inverse, while the two rectangular blocks in the top implement the corresponding matrix multiplication and addition. The signal 'enc' is '1' when it's in encryption mode, and is '0' otherwise. Since both M and M^{-1} are binary matrices, the matrix multiplication block can be implemented simply by XOR gates. Another approach is to store the value of S-box and S^{-1}-box in two separate ROMs, and read the initial values into RAMs at the beginning of encryption/decryption [5]. This approach eliminates the duplicated memory by 2 additional ROMs, but introduces an overhead of 256 clock cycles to read in the initial values.

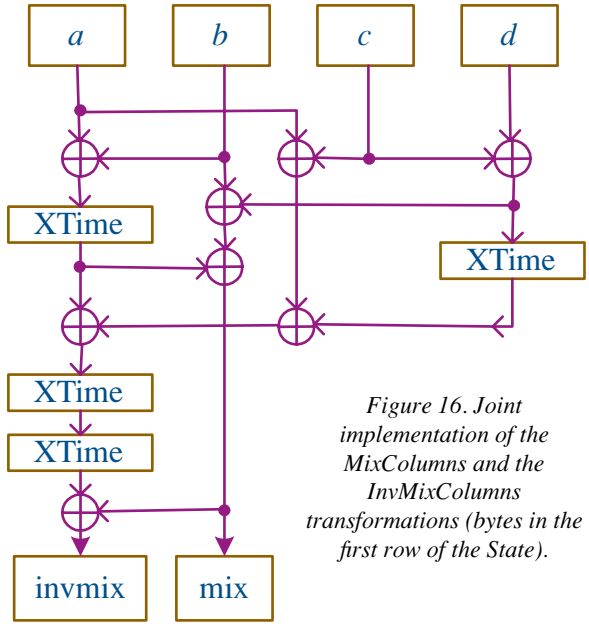


Figure 16. Joint implementation of the MixColumns and the InvMixColumns transformations (bytes in the first row of the State).

Resource Sharing in MixColumns and InvMixColumns

Although (12) leads to an InvMixColumns implementation with neither the shortest delay nor the smallest area, combined with (10), it leads to the hardware implementation with least area of joint MixColumns/InvMixColumns transformation. Figure 16 illustrates the diagram accord-

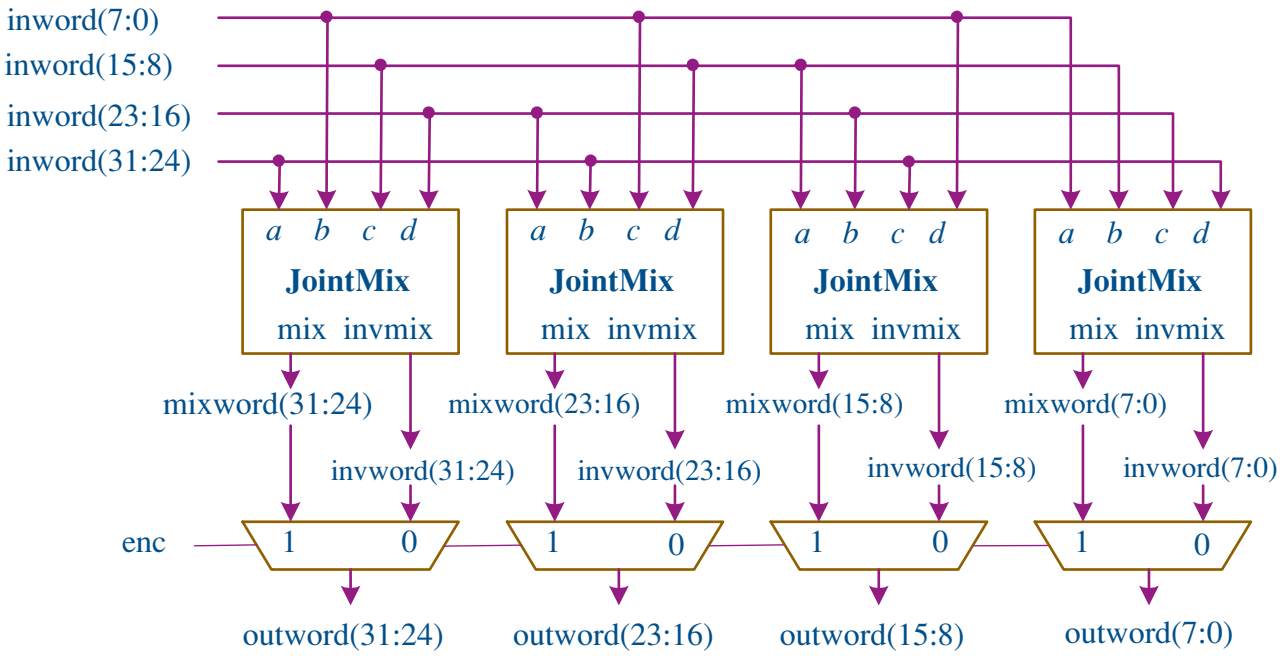


Figure 17. Joint implementation of the MixColumns and the InvMixColumns transformations (one column in the State).

ing to (10) and (12). The four inputs, '*a*', '*b*', '*c*', '*d*', and two outputs, 'mix' and 'invmix', all represent single bytes. '*a*', '*b*', '*c*', '*d*' are the four bytes in a column of the State with ascending row numbers. 'mix' and 'invmix' are the outcomes of applying MixColumns and InvMixColumns transformation to the inputs, respectively. The block diagram for applying MixColumns and InvMixColumns to the bytes in other rows can be obtained by exchanging the position of the input bytes according to (2) and (3). Figure 17 shows the diagram of applying MixColumns and InvMixColumns transformations to one column of the State. Each of the JointMix blocks consists of the diagram in Fig. 16. 'mixword' is the output of applying the MixColumns transformation to the 'inword', and 'invword' is the output of applying the InvMixColumns transformation to the 'inword'. The 'outword' gets the value of 'mixword' when 'enc' = '1', which indicates encryption mode, and gets the value of 'invword' otherwise.

Resource Sharing of Generating Roundkeys in Encryption and Decryption

In the applications with limited area, generating roundkeys on the fly is a better choice. [7] proposed an efficient architecture which can generate roundkey(i + 1) from roundkey(i) and *vice versa*. This architecture is illustrated in Fig. 18.

In Fig. 18, each of the 'roundkey' and the 'next-roundkey' consist of four words. Assuming the 'roundkey' is expressed by four words as (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}), the 4 sets of XOR gates from left to right get w_{4i}, w_{4i+1}, w_{4i+2}, and w_{4i+3} as one of the inputs from the 'roundkey' bus, respectively. The RotSub block in Fig. 18, which performs RotWord followed by SubBytes transformation, is made up of 4 *S*-boxes. Since all the lower bytes of the round constant Rcon are zeros, the step of adding round constant only needs to be performed on the most significant byte. Meanwhile, Rcon(i + 1) equals {02}Rcon(i) ($1 \le i < \mathrm{Nr} - 1$).

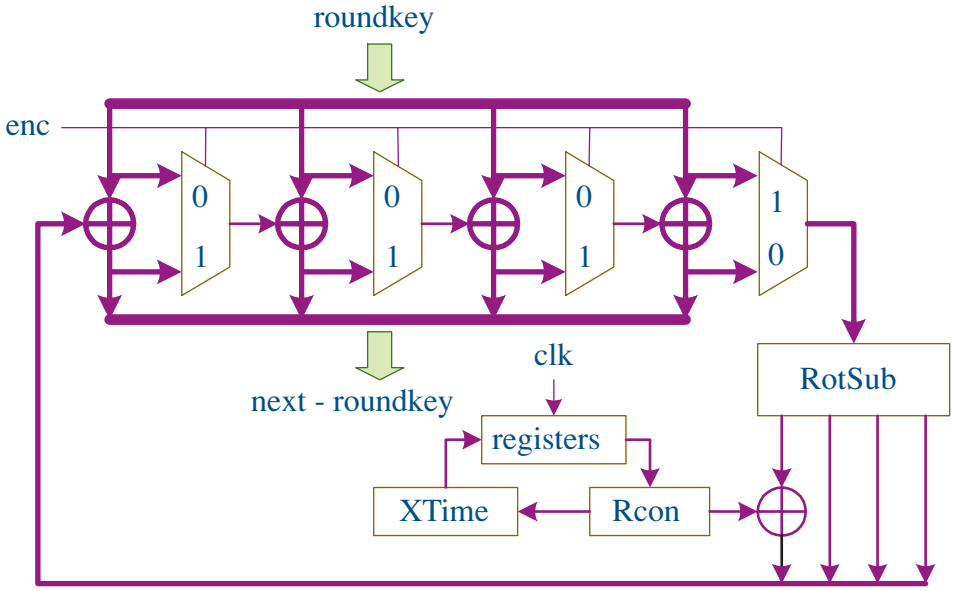


Figure 18. Joint implementation of Key Expansion in encryptor and decryptor.

Rcon(i + 1) can also be generated on the fly from the stored Rcon(1) = {01}. When 'enc' is '1', which stands for encryption mode, w_{4i+3} is loaded into the RotSub block, after the output of RotSub block was XORed with Rcon(i) and w_{4i}, $w_{4(i+1)}$ is generated at the output of the first XOR gate on the left. Consequently, $w_{4(i+1)+1}$, $w_{4(i+1)+2}$, and $w_{4(i+1)+3}$ are generated one by one as the updated data propagates through each multiplexer from left to right. When 'enc' is '0', $w_{4i+3} \oplus w_{4i+2} = w_{4(i-1)+3}$ $(0 < i \leq Nr)$ is loaded into the RotSub block, after the output of RotSub is XORed with Rcon(i), a temporary value temp = SubWord(RotWord($w_{4(i-1)+3}$)) $\oplus$ Rcon(i) is fed back to the leftmost XOR gate in Fig. 18. $w_{4(i-1)} = w_{4i} \oplus$ temp is generated after an XOR gate delay. As the updated data propagates through each of the multiplexers, $w_{4(i-1)+1}$, $w_{4(i-1)+2}$, and $w_{4(i-1)+3}$ are generated in a sequence.

Conclusion

Architectural and algorithmic optimization approaches for efficient hardware implementations of the AES algorithm have been addressed in this paper. In addition, the joint implementation issues of encryptor and decryptor are also discussed. None of the reported implementations have used all the applicable optimization methods discussed in this paper. Optimization approaches for the implementations supporting multiple key lengths and modes of operation require further study.

References

[1] J. Daemen and R.Rijmen, "AES Proposal: Rijndael", version 2, 1999. Available at http://www.esat.kuleuven.ac.be/~rijmen/rijndael.

[2] "Advanced Encryption Standard(AES)", Federal Information Processing Standards Publication 197, November 26, 2001.

[3] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalist", *The Third AES Conference (AES3)*, New York, April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[4] K. Gaj and P. Chodowiec, "Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware", *The Third AES Conference (AES3)*, New York, April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

Implementation Approaches for the AES Algorithm

[5] M. McLoone and J. V. McCanny, "Rijndael FPGA Implementation Utilizing Look-Up Tables", *IEEE Workshop on Signal Processing Systems*, pp. 349–360, September 2001.

[6] T. Ichikawa, T. Kasuya, and M. Matsui, "Hardware Evaluation of the AES Finalists", *The Third AES Conference (AES3)*, New York, April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[7] C. C. Lu and S. Y. Tseng, "Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter", *IEEE Transactions on Information Theory*, vol. 37, no. 5, pp. 1241–1260, September 1991.

[8] H. Kuo and I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm", *Proceedings CHES 2001*, pp. 51–64, Paris, France, May 2001.

[9] V. Fischer and M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware", *Proceedings CHES 2001*, pp. 77–92, Paris, France, May 2001.

[10] V. Fischer, "Realization of the Round 2 Candidates Using Altera FPGA", *The Third AES Conference (AES3)*, New York, Apr. 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[11] 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", *Proceedings CHES 2001*, pp. 171–184, Paris, France, May 2001.

[12] V. Rijmen, "Efficient Implementation of the Rijndael *S*-box", Available at http://www.esat.kuleuven.ac.be/~rijmen/rijndael.

[13] K. K. Parhi, *VLSI Digital Signal Processing Systems-Design and Application*, John Wiley & Sons, pp. 559–562, 1999.

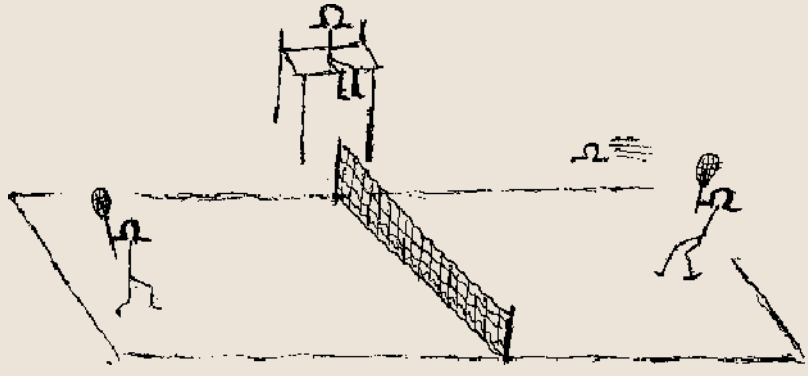
[14] J. Daemen and R.Rijmen, "Rijndael: The Advanced Encryption Standard", *Dr. Dobb's Journal*, pp. 137–139, March 2001.

[15] M. McLoone and J.V.McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm Implementation", *Proceedings CHES 2001*, pp. 65–76, Paris, France, May 2001.

[16] N. Weaver and J. Wawrzynek, "A Comparison of the AES Candidates Amenability to FPGA Implementation", *The Third AES Conference (AES3)*, New York, April 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.

[17] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography, CRC Press*, 1997.

THE ADVENTURES OF ...
...THE 'UMBLE OHM
...Shlomo Karni



Circuit Court Judge