Chapter 10
FPGA and ASIC Implementations of AES

Kris Gaj and Pawel Chodowiec

10.1 Introduction

In 1997, an effort was initiated to develop a new American encryption standard to be commonly used well into the next century. This new standard was given a name AES, *Advanced Encryption Standard*.

A new algorithm was selected through a contest organized by the National Institute of Standards and Technology (NIST). By June 1998, 15 candidate algorithms had been submitted to NIST by research groups from all over the world. After the first round of analysis was concluded in August 1999, the number of candidates was reduced to final five. In October 2000, NIST announced its selection of *Rijndael* [7] as a winner of the AES contest. The official standard was published in November 2001 as FIPS (Federal Information Processing Standard) number 197 [1].

The primary criteria used by NIST to evaluate AES candidates included security, efficiency in software and hardware, and flexibility. In the absence of any major breakthroughs in the cryptanalysis of the final five candidates, and because of the relatively inconclusive results of their software performance evaluations, hardware efficiency evaluations presented during the third AES conference provided a very substantial quantitative measure that clearly differentiated AES candidates among each other [9, 10, 12, 17, 21, 42]. The importance of this measure was reflected by a survey performed among the participants of the AES conference, in which the ranking of the candidate algorithms coincided very well with their relative speed in hardware [16, 18].

The AES evaluation process resulted in the first efficient hardware architectures for AES. The university groups contributed first implementations of AES based on FPGAs (field programmable gate arrays) [5, 9, 11, 18]. The National Security Agency group and industry groups provided the first implementations targeting ASICs (application-specific integrated circuits) [21, 42].

George Mason University
e-mail: {kgaj,pchodow1}@gmu.edu

Ç.K. Koç (ed.) *Cryptographic Engineering*, DOI 10.1007/978-0-387-71817-0_10,

A substantial progress in the development of the new architectures for AES has been made after the conclusion of the contest, as a result of focusing research efforts on a single secret-key encryption standard. This progress proceeded in several major directions.

One direction was the development of high-speed, highly pipelined architectures for non-feedback cipher modes. This direction led to the development of AES implementations operating with the speeds of tens of Gigabits per second [22, 23, 25, 26, 29, 32, 34–36, 41]. The second direction was the development of compact architectures for AES, optimized for the minimum area. This effort led to the emergence of architectures with 64-, 32-, and even 8-bit data paths [2, 6, 19, 20, 27, 33, 45].

The third direction was the optimization of basic operations of AES, including logic-only implementation of *SubBytes* [3, 4, 28–31, 33, 44] and optimizations and decompositions of the *MixColumns* and *InvMixColumns* transformations [6, 14, 15, 43]. Still, a different direction was the development of new architectures for the entire encryption/decryption unit [13].

In this chapter, we will review the AES algorithm from the point of view of knowledge required for efficient hardware implementations. We will then describe several alternative ways of implementing all basic operations and the entire cipher. We will conclude with our recommendations regarding the optimum choice of particular design options and the entire hardware architecture for AES depending on requirements of a particular application.

10.2 AES Cipher Description

10.2.1 Basic Features

AES is a symmetric-key block cipher. AES operates on 128-bit data blocks and accepts 128-, 192-, and 256-bit keys. It is an iterative cipher, which means that both encryption and decryption consist of multiple iterations of the same basic round function, as shown in Figure 10.1.

In each round, a different *round* (or *internal*) key is being used. In AES, the number of cipher rounds depends on the size of the key. It is equal to 10, 12, or 14 for 128-, 192-, or 256-bit keys, respectively.

Based on the internal structure of a round function, AES belongs to the group of SP-network block ciphers. This means that the main transformations employed in this cipher are substitutions and permutations applied to all bits of data block in every round. Data blocks are internally represented in a square form, called State, which is shown in Figure 10.2. In this diagram, each field represents one byte of data.

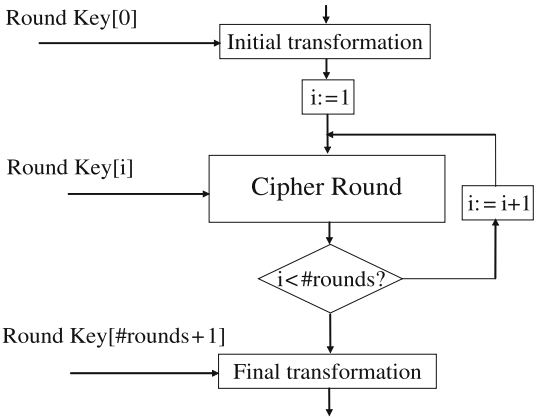

Fig. 10.1 Flowchart of a generic iterative cipher.

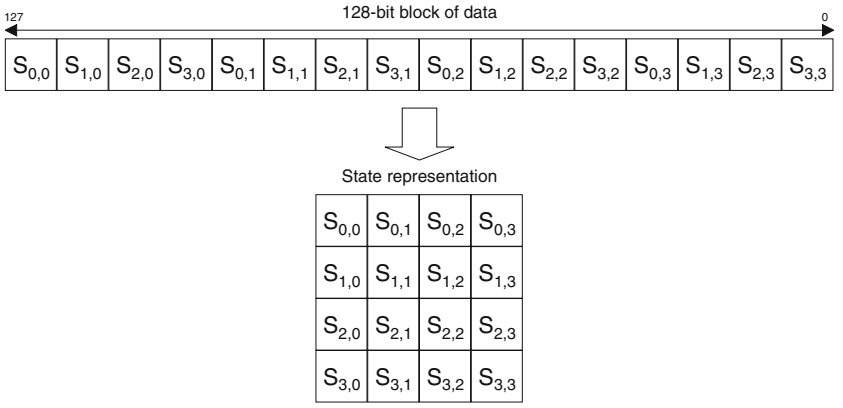

Fig. 10.2 State representation of 128-bit data blocks.

10.2.2 Round Operations

AES encryption round employs consecutively four main operations: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. Since Rijndael is an SP-network cipher, it requires an inversed version of all transformations for decryption. These inverse transformations are called *InvSubBytes*, *InvShiftRows*, *InvMixColumns*, and *InvAddRoundKey*. Please note that the last transformation of an encryption round, *AddRoundKey*, is equivalent to a bitwise XOR and therefore is an inverse of itself. The structure of encryption and decryption rounds is shown in Figure 10.3.

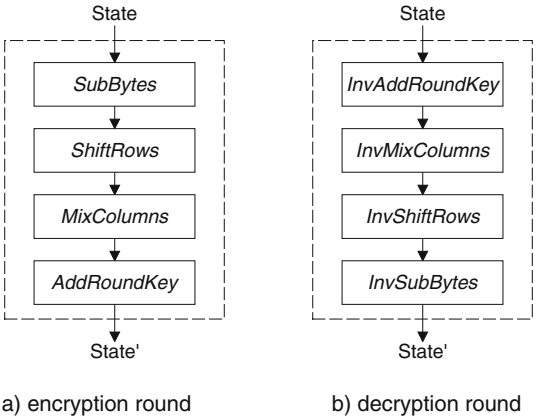

Fig. 10.3 Structure of AES encryption and decryption round.

10.2.2.1 Operations in the Galois Field $GF(2^8)$

Two of the AES round operations, *SubBytes* and *MixColumns*, rely on operations in the Galois field $GF(2^8)$. Each element of this field can be treated as either an 8-bit string (in the binary or hexadecimal representation) or as a polynomial of degree seven or less, with coefficients in $\{0,1\}$ (polynomial basis representation). The coefficients of a polynomial are equal to the respective bits of the binary representation. For example, $\{03\}$ in hexadecimal is equivalent to $\{0000\ 0011\}$ in binary, and to

$$c(x) = 0 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 1 \cdot 1 = x + 1 \qquad (10.1)$$

in the polynomial basis representation. The multiplication of elements of $GF(2^8)$ in AES is accomplished by multiplying the corresponding polynomials modulo a fixed irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

For example, multiplying a variable element $a = a_7a_6a_5a_4a_3a_2a_1a_0$ by a constant element $\{03\}$ is equivalent to computing

$$\begin{aligned} b(x) &= b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \\ &= (a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot (x+1) \\ &\quad \bmod (x^8 + x^4 + x^3 + x + 1) \end{aligned} \qquad (10.2)$$

After several simple transformations

$$\begin{aligned} b(x) \quad &= (a_7 + a_6) \cdot x^7 + (a_6 + a_5) \cdot x^6 + (a_5 + a_4) \cdot x^5 + (a_7 + a_4 + a_3) \cdot x^4 \\ &\quad + (a_7 + a_3 + a_2) \cdot x^3 + (a_2 + a_1) \cdot x^2 + (a_7 + a_1 + a_0) \cdot x + (a_7 + a_0) \end{aligned}$$

where "+" represents an addition modulo 2, i.e., an XOR operation. Each bit of a product b can be represented as an XOR function of at most three variable input bits, e.g., $b_7 = (a_7 + a_6)$, $b_4 = (a_4 + a_3 + a_7)$, etc.

10.2.2.2 *SubBytes* and *InvSubBytes*

The *SubBytes* operation transforms individual bytes of the internal state as shown in Figure 10.4. Internally, it is composed of two basic operations:

1. Multiplicative inversion in the Galois field $GF(2^8)$ with the reduction polynomial $m(x)$ specified by Equation (10.3). Element $\{00\}$ is mapped onto itself.

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{10.3}$$

2. Affine transformation over $GF(2)$:

$$b'_i = b_i + b_{(i+4)\ mod\ 8} + b_{(i+5)\ mod\ 8} + b_{(i+6)\ mod\ 8} + b_{(i+7)\ mod\ 8} + c_i \tag{10.4}$$

where byte c has value $\{63\}$ or $\{01100011\}$.
Equation (10.5) shows the affine transformations in the matrix form.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{10.5}$$

The inversed version of *SubBytes*, called *InvSubBytes*, employs identical multiplicative inversion in $GF(2^8)$ and an inversed affine transformation. Equation (10.6) shows the inverse affine transformations in the matrix form:

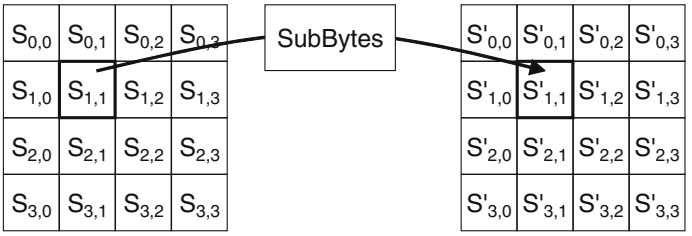

Fig. 10.4 Application of *SubBytes* to State.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{10.6}$$

The corresponding equation in $GF(2)$ is given in (10.7) as

$$b'_i = b_{(i+2)\ mod\ 8} + b_{(i+5)\ mod\ 8} + b_{(i+7)\ mod\ 8} + d_i \tag{10.7}$$

The internal structure of *SubBytes* and *InvSubBytes* is shown in Figure 10.5.

10.2.2.3 *ShiftRows* and *InvShiftRows*

The *ShiftRows* and *InvShiftRows* cyclically shift three bottom rows of the State by a different number of positions, one, two, and three, respectively, as shown in Figure 10.6. Without those operations all-round transformations would be limited only to the columns of the State.

10.2.2.4 *MixColumns* and *InvMixColumns*

MixColumns and *InvMixColumns* operations are defined over 4-byte words that represent a column of the State as shown in Figure 10.7. These 4-byte words are considered as polynomials (of degree of at most 3) with coefficients in $K = GF(2^8)$, defined in the ring of polynomials $K[X]$ modulo $M(X) = X^4 + 1$ and denoted as

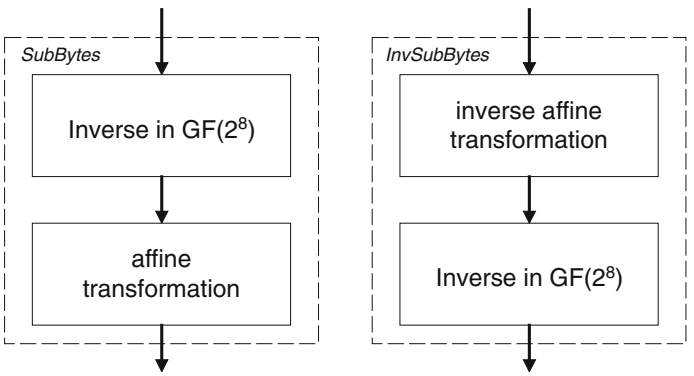

Fig. 10.5 Composition of *SubBytes* and *InvSubBytes*.

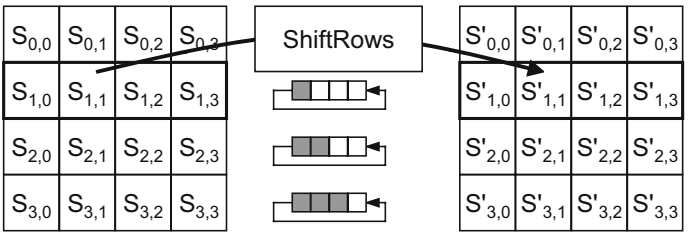

Fig. 10.6 *ShiftRows* transforming rows of a State.

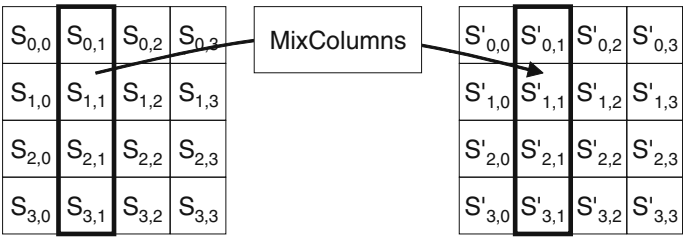

Fig. 10.7 *MixColumns* transforming a column of a State.

$R = K[X]/(X^4+1)$. Addition of these polynomials corresponds to bit-wise XOR of their coefficients. Their multiplication is reduced modulo $M(X) = X^4 + 1$.

Since $X^j \bmod (X^4+1) = X^{j \bmod 4}$, the operation consisting of multiplication of $a(X) = a_3X^3 + a_2X^2 + a_1X + a_0$ by a fixed polynomial $c(X) = c_3X^3 + c_2X^2 + c_1X + c_0$ gives a product

$$\begin{aligned} B(X) &= b_3X^3 + b_2X^2 + b_1X + b_0 \\ &= (c_3a_0 + c_2a_1 + c_1a_2 + c_0a_3)X^3 \\ &\quad + (c_2a_0 + c_1a_1 + c_0a_2 + c_3a_3)X^2 \\ &\quad + (c_1a_0 + c_0a_1 + c_3a_2 + c_2a_3)X \\ &\quad (c_0a_0 + c_3a_1 + c_2a_2 + c_1a_3) \end{aligned} \tag{10.8}$$

This operation can be written as a multiplication of a vector $[A]$ by a circular matrix $[C]$:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.9}$$

The polynomial $M(X)$ was selected such that it effectively shifts rows of the State. From the cryptographical point of view, this operation mixes bytes across the State and creates a strong dependence between all input bytes $a_0 \ldots a_3$ and an output byte b_i.

The *MixColumns* transformation multiplies each column of the State by a constant polynomial $c(X)$ in the ring R. The $c(X)$ is defined as follows:

$$c(X) = (x+1)X^3 + X^2 + X + x \tag{10.10}$$

The *InvMixColumns* transformation is the inverse of the *MixColumns* operation. *InvMixColumns* multiplies each column of the State by

$$d(X) = (x^3 + x + 1)X^3 + (x^3 + x^2 + 1)X^2 + (x^3 + 1)X + (x^3 + x^2 + x) \tag{10.11}$$

where $d(X) = c^{-1}(X)$ is the inverse of $c(X)$ in R. Polynomials $c(X)$ and $d(X)$ are often expressed with coefficients in the hexadecimal format:

$$c(X) = 03 \cdot X^3 + 01 \cdot X^2 + 01 \cdot X + 02 \tag{10.12}$$

$$d(X) = 0B \cdot X^3 + 0D \cdot X^2 + 09 \cdot X + 0E \tag{10.13}$$

Multiplication of one column of the State by $c(X)$ in R (part of the *MixColumns* operation) can be written in a matrix form:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.14}$$

The expression of the *InvMixColumns* operation in a matrix form is as follows:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E\ 0B\ 0D\ 09 \\ 09\ 0E\ 0B\ 0D \\ 0D\ 09\ 0E\ 0B \\ 0B\ 0D\ 09\ 0E \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.15}$$

Polynomials representing columns of a State have coefficients which are considered as polynomials (of degree of at most 7) with coefficients in Galois field $GF(2)$. A byte $a(x)$ (or a in simplified notation) is a sum $a(x) = \sum_{0 \le i \le 7} \alpha_i x^i$, where $\alpha_i \in \{0,1\}$. In other words, bytes a are elements of the Galois field $K = GF(2^8)$ constructed using the reduction polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

$$K = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1) \tag{10.16}$$

Addition of polynomials in K corresponds to simple bit-wise exclusive OR (XOR) of the polynomial coefficients. Multiplication of polynomials in the field K corresponds to their multiplication modulo irreducible polynomial $m(x)$ from Equation (10.3). The same polynomial is used in the *SubBytes* operation for calculation of a multiplicative inverse.

10.2.3 Iterative Structure

A flowchart describing AES encryption in terms of basic operations—*SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*—is shown in Figure 10.8. Please note

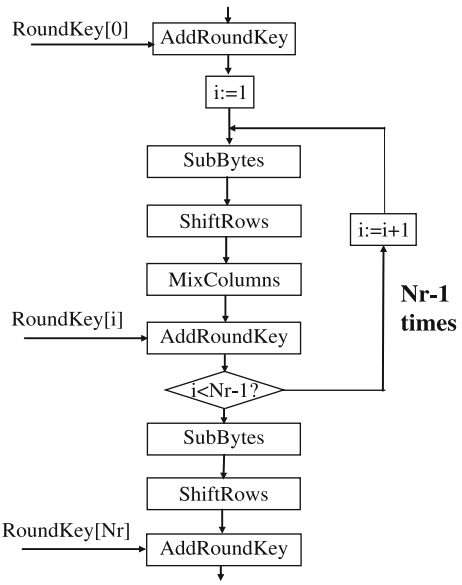

Fig. 10.8 AES encryption flowchart.

that the number of cipher rounds, Nr, depends on the size of an encryption key. The first round is preceded by an initial transformation *AddRoundKey*, in agreement with a generic structure of an iterative block cipher shown in Figure 10.1. The last round, number Nr, is slightly different from the remaining $Nr-1$ rounds, in that it does not contain the *MixColumns* operation.

By a straightforward inversion of the order of operations and by replacing all basic operations by their respective inverses, we obtain the AES decryption flowchart shown in Figure 10.9. Simple regrouping of basic operations leads to an equivalent decryption flowchart, shown in Figure 10.10, which has the same basic structure as an encryption flowchart. The differences amount to providing round keys in the reverse order, replacing all basic operations by their inverses, and swapping the order of operations one and two, and three and four within each round. The operations number one and two during each round of encryption, *SubBytes* and *ShiftRows*, can be performed in an arbitrary order, as shown in Figure 10.11a. Similarly, the operations number one and two during each round of decryption *InvShiftRows* and *InvSubBytes* can be swapped without affecting the result (see Figure 10.11b). By applying this last change, we obtain the decryption flowchart, shown in Figure 10.12, which is most often used as a basis of hardware implementation.

10.2.4 Key Scheduling

Key scheduling in AES is a process aimed at generating $(Nr+1)$ *round keys* based on a single external key. This process consists of two phases called `KeyExpansion`

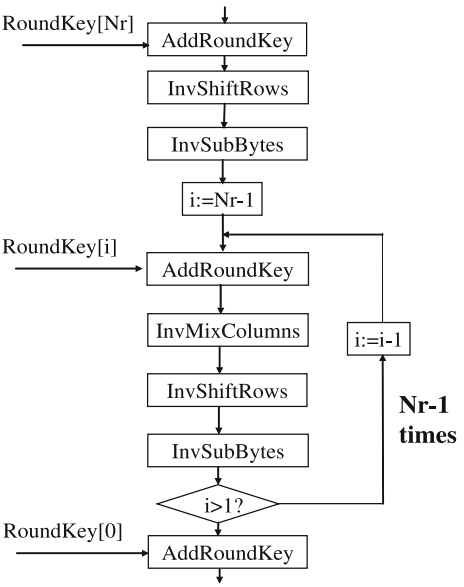

Fig. 10.9 AES decryption flowchart obtained by the straightforward inversion of the encryption flowchart.

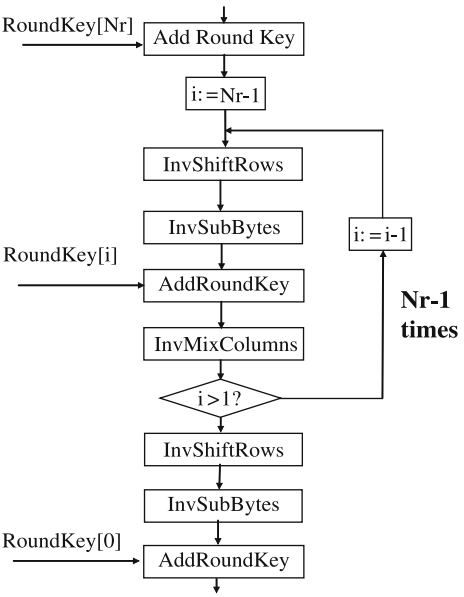

Fig. 10.10 AES decryption flowchart after regrouping of basic operations.

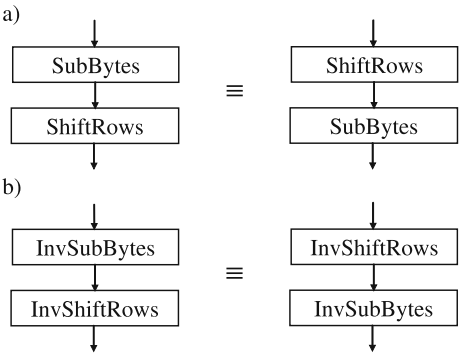

Fig. 10.11 Equivalence between two sequences of basic operations: (**a**) *SubBytes* followed by *ShiftRows*, (**b**) *InvSubBytes* followed by *InvShiftRows*.

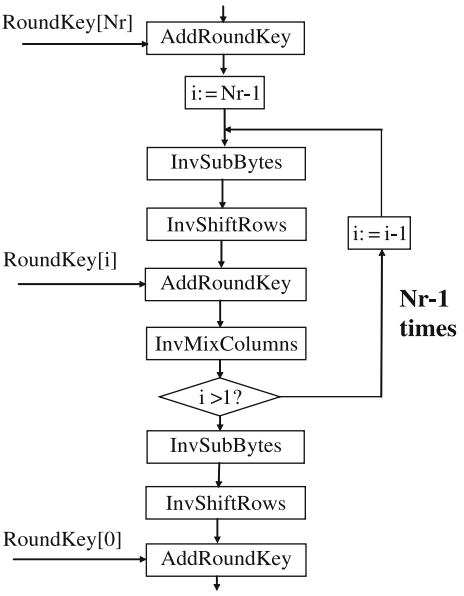

Fig. 10.12 AES decryption flowchart after regrouping of basic operations and swapping *InvSubBytes* with *InvShiftRows*.

and `RoundKeySelection`, as shown in Figure 10.13. Please note that all rectangular fields in this and two subsequent figures correspond to 32-bit words (and not single bytes).

The pseudocode of `KeyExpansion` is shown in Figure 10.16. The output array of words `k[i]` is first initialized with the `Nk` words of the external key, Key. For majority of subsequent values of `i`, `k[i]` is computed by simply XORing an immediately preceding word `k[i-1]` with a word `Nk` positions earlier `k[i-Nk]`, as shown in Figure 10.14. If an index `i` is a multiple of `Nk`, a different transfor-

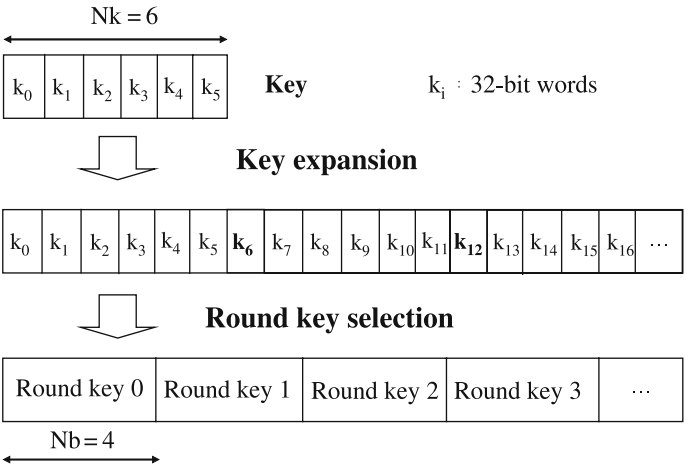

Fig. 10.13 Decomposition of key scheduling into *KeyExpansion* and *RoundKeySelection* for the case of Nk = 6 (192-bit key) and Nb = 4 (128-bit data block).

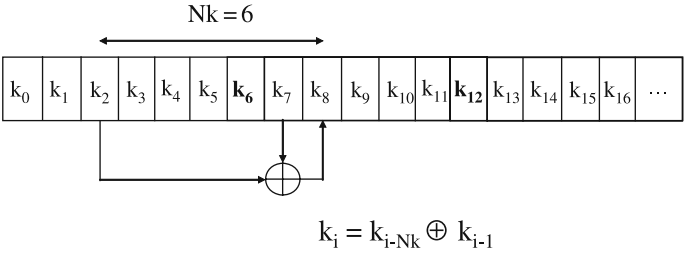

Fig. 10.14 Formula for *KeyExpansion* for i mod $Nk \neq 0$.

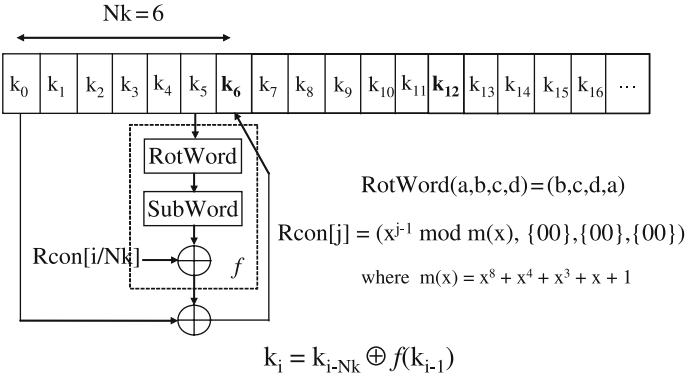

Fig. 10.15 Formula for *KeyExpansion* for i mod $Nk = 0$.

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

  i=0
  while (i < Nk)
    k[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 = k[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
    k[i] = k[i-Nk] xor temp
    i = i + 1
  end while

end
```

Fig. 10.16 Pseudocode for the *KeyExpansion* phase of *KeyScheduling*.

mation, shown in Figure 10.15, is used. In this transformation, *RotWord* is a cyclic rotation of bytes within a word, *SubWord* is a *SubBytes* transformation applied independently to each byte of an input word, and `Rcon[i]` is an array of four constants defined in $GF(2^8)$. If `(Nk > 6) and (i mod Nk) = 4`, a simplified version of the same transformation is applied.

10.3 FPGA and ASIC Technologies

Cryptographic transformations can be implemented in both software and hardware. Software implementations are designed and coded in programming languages, such as C, C++, Java, and assembly language, to be executed, among others, on general purpose microprocessors, digital signal processors, and smart cards. Hardware implementations are designed and coded in hardware description languages, such as VHDL and Verilog HDL, and are intended to be realized using two major implementation approaches: application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs).

Application-specific integrated circuits (ASICs) are designed all the way from the behavioral description to the physical layout and then sent for a fabrication in a semiconductor foundry. Field programmable gate array (FPGA) can be bought off the shelf and reconfigured by designers themselves. With each reconfiguration,

which takes only a fraction of a second, an integrated circuit can perform a completely different function.

FPGA consists of thousands of universal reconfigurable logic blocks, connected using reconfigurable interconnects and switches, as shown in Figure 10.17. Additionally, modern FPGAs contain embedded higher-level components, such as memory blocks, multipliers, multipliers–accumulators, and even microprocessor cores. Reconfigurable input/output blocks provide a flexible interface with the outside world. Reconfiguration, which typically lasts only a fraction of a second, can change a function of each building block and interconnects among them, leading to a functionally new digital circuit.

In Table 10.1, we collect and contrast features of implementations of cryptographic transformations based on ASICs and FPGAs (hardware) and microprocessors (software). The performance characteristics of ASICs and FPGAs are almost identical, as demonstrated by the first group of features, and substantially different from the performance characteristics of general purpose microprocessors. Both ASICs and FPGAs can make a full use of parallel processing and pipelining and operate on arbitrary size words. In general purpose microprocessors, parallel processing and pipelining are limited by the number and internal structure of the processor functional units and by the instruction level parallelism. Additionally, all functional units operate on the fixed-size arguments only.

The primary difference between ASICs and FPGAs in terms of the performance characteristics is a smaller speed of FPGAs caused by the delays introduced by the circuitry required for reconfiguration. As a result of this speed penalty, any digital circuit implemented in an FPGA is typically slower than the same circuit implemented in an ASIC, assuming that both integrated circuits are fabricated using the same semiconductor technology (in particular, using the same transistor size).

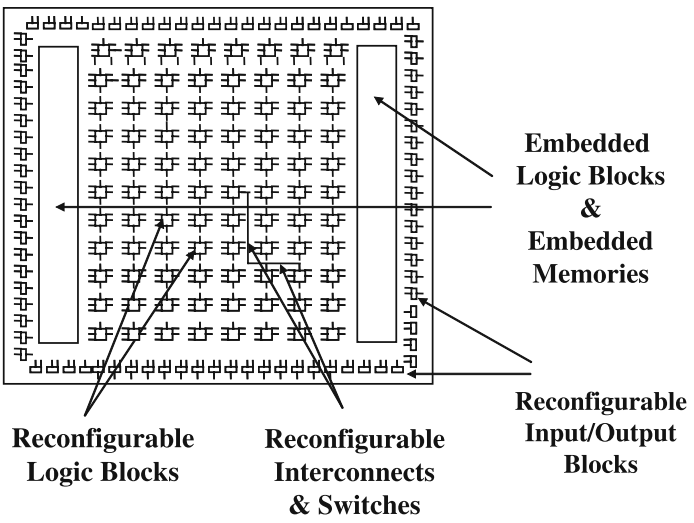

Fig. 10.17 General structure and main building blocks of an FPGA.

Table 10.1 Characteristic features of implementations of cryptographic transformations in ASICs, FPGAs, and microprocessors.

	ASICs	FPGAs	Microprocessors
Performance characteristics			
Parallel processing	Yes	Yes	Limited
Pipelining	Yes	Yes	Limited
Word size	Variable	Variable	Fixed
Speed	Very fast	Fast	Moderately fast
Functionality			
Algorithm agility	No	Yes	Yes
Tamper resistance	Strong	Limited	Weak
Access control to keys	Strong	Moderate	Weak
Development process			
Description languages	VHDL, Verilog HDL	VHDL, Verilog HDL	C, C++, Java, assembly language
Design cycle	Long	Moderately long	Short
Design tools	Very expensive	Moderately expensive	Inexpensive
Maintenance and upgrades	Expensive	Inexpensive	Inexpensive

The recent study performed at the University of Toronto [24] quantified the performance differences between the current generation of ASICs and FPGAs. A set of 23 benchmarks covering applications in the area of cryptography, digital signal processing, and communications were included in the study. The study concluded that for circuits containing only combinational logic and flip-flops, the ratio of silicon area required to implement them in FPGAs and ASICs is on average 40. For circuits that could take advantage of dedicated blocks present in modern FPGAs, such as multiplier/accumulators and block memories, these blocks reduced the average area gap significantly to as little as 21. The ratio of critical path delay, from FPGA to ASIC, was found to be roughly 3–4, with less influence from embedded memories and embedded logic blocks. The dynamic power consumption ratio was approximately 12 times and, with hard blocks, this gap generally became smaller.

The common features of FPGAs and microprocessors concern mostly functionality and do not affect performance. Both general purpose microprocessors and FPGAs can be easily reconfigured in real time to perform a different algorithm. The disadvantage of this feature is a limited tamper resistance; the contents of an FPGA can be, at least in theory, modified by an unauthorized user. In practice, the contents of an FPGA are typically downloaded during the initialization from the read-only memory, such as EPROM, which cannot be easily tampered with, at least remotely. The access control to cryptographic keys in FPGAs is also stronger than in software, but weaker than in ASICs.

The development process in both hardware implementation approaches is very similar. In both FPGAs and ASICs, the circuit is described using a hardware description language, verified using a digital circuit simulator, and then tested using a prototyping board. The primary difference between FPGAs and ASICs is that FPGAs do not require the physical design (layout), fabrication, and testing for

physical defects. As a result, the design cycle is significantly shorter and the design tools and testing much less expensive. The interesting similarity between FPGAs and software is a possibility of remote maintenance and upgrading, based on electronic patches.

10.4 Parameters of Hardware Implementations

Hardware implementations of secret-key ciphers can be characterized using several performance parameters. Below we provide our definitions of major parameters and derive formulas that demonstrate mutual dependencies among these parameters.

10.4.1 Throughput and Latency

Encryption (decryption) throughput is defined as the number of bits encrypted (decrypted) in a unit of time. Typically, the encryption and decryption throughputs are equal, and therefore only one parameter is reported. A typical unit of throughput is Mbit/s (megabit per second) or Gbit/s (gigabit per second). It is worth mentioning that 1 Mbit/s = 10^6 bit/s, and not 2^{20} bit/s, and 1 Gbit/s = 10^9 bit/s, and not 2^{30} bit/s.

Encryption (decryption) latency is defined as the time necessary to encrypt (decrypt) a single block of plaintext (ciphertext). The typical unit of latency in the current technology is ns (nanosecond).

The encryption (decryption) latency and throughput are related by

$$Throughput = \frac{block\_size \cdot number\_of\_blocks\_processed\_simultaneously}{latency} \quad (10.17)$$

In applications where large amounts of data are encrypted or decrypted, throughput determines the total encryption/decryption time and thus is the best measure of the cipher speed. In applications where a small number of plaintext (ciphertext) blocks is processed, the total encryption/decryption time depends on both throughput and latency.

10.4.2 Area

The area required for the cipher implementation is an important parameter for the following reasons:

- Cost
  The area of an integrated circuit is a primary factor determining its cost. It is traditionally assumed that the cost of an integrated circuit is directly proportional

to the circuit area. This dependence is not always accurate, especially taking into account the cost of a package, which is determined by the number of the circuit inputs and outputs.

- Limit on the maximum area
  In certain hardware environments, there exists a limit on the maximum area of a cryptographic unit. This limit may be imposed by the cost, available fabrication technology, power consumption, or any combination of these factors. For example, in smart cards and microcontrollers, both cost and power consumption limit the area of the embedded encryption units; in FPGAs, the area is limited by the available fabrication technology and the cost of a programmable device.

In ASIC implementations, the area required by the cryptographic unit is typically expressed in μm^2. Two related measures are the transistor count and the logic gate count. Values of all three measures are closely correlated, but not necessarily strictly proportional to each other. All three measures are reported by the tools used for the automated logic synthesis of ASICs. In the semi-custom design methodology, these values are a function of the standard cell library used during logic synthesis.

In FPGA implementations, the only circuit size measures reported by the CAD tools are the number of basic configurable logic blocks and the number of equivalent logic gates. It is commonly believed that out of these two measures, the number of basic configurable logic blocks approximates the circuit area more accurately. Measuring and comparing circuit area in FPGAs are additionally complicated by the existence of embedded logic blocks and embedded memories. The specifications of FPGA devices typically do not provide any information about the relative ratio of the areas used by embedded blocks and basic reconfigurable logic blocks.

10.5 Hardware Architectures of Symmetric Block Ciphers

10.5.1 Hardware Architectures vs. Block Cipher Modes of Operation

Symmetric-key block ciphers are used in several operating modes. From the point of view of hardware implementations, these modes can be divided into two major categories:

1. Non-feedback modes, such as electronic code book mode (ECB) and counter mode (CTR).
2. Feedback modes, such as cipher block chaining mode (CBC), cipher feedback mode (CFB), and output feedback mode (OFB).

In the non-feedback modes, encryption of each subsequent block of data can be performed independently from processing other blocks. In particular, all blocks can be encrypted in parallel. In the feedback modes, it is not possible to start encrypting the next block of data until encryption of the previous block is completed.

As a result, all blocks must be encrypted sequentially, with no capability for parallel processing. The limitation imposed by the feedback modes does not concern decryption, which can be performed on several blocks of ciphertext in parallel for both feedback and non-feedback operating modes.

In the old security standards, the encryption of data was performed primarily using feedback modes, such as CBC and CFB. Using these standards did not permit to fully utilize the performance advantage of the hardware implementations of secret-key ciphers, based on parallel processing of multiple blocks of data. The situation has been partially remedied by including a counter mode in the NIST recommendations on the AES modes of operation. Other non-feedback modes of operation are currently under investigation by the cryptographic community.

10.5.2 Basic Iterative Architecture

The basic hardware architecture used to implement an encryption/decryption unit of a typical secret-key cipher is shown in Figure 10.18. One round of the cipher is implemented as a combinational logic and supplemented with a single register and a multiplexer. In the first clock cycle, input block of data is fed to the circuit through the multiplexer and stored in the register. In each subsequent clock cycle, one round of the cipher is evaluated, the result is fed back to the circuit through the multiplexer, and stored in the register. The two characteristic features of this architecture are

- Only one block of data is encrypted at a time.
- The number of clock cycles necessary to encrypt a single block of data is equal to the number of cipher rounds, #rounds.

The throughput and latency of the basic iterative architecture, $Throughput_{iterative}$ and $Latency_{iterative}$, are given by

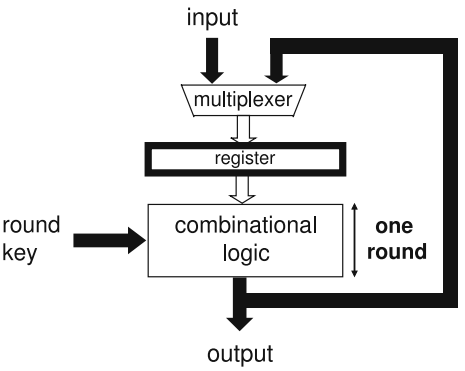

Fig. 10.18 Basic iterative architecture of a block cipher.

$$Throughput_{iterative} = \frac{block\_size}{\#rounds \cdot T_{CLK_{iterative}}} \tag{10.18}$$

$$Latency_{iterative} = \#rounds \cdot T_{CLK_{iterative}} \tag{10.19}$$

where $T_{CLK_{iterative}}$ is a clock period of the basic iterative architecture.

10.5.3 Loop Unrolling

An architecture *with partial loop unrolling* is shown in Figure 10.19b. The only difference compared to the basic iterative architecture is that the combinational part of the circuit implements K rounds of the cipher, instead of a single round. K must be a divisor of the total number of rounds, #*rounds*.

The number of clock cycles necessary to encrypt a single block of data decreases by a factor of K. At the same time the minimum clock period increases by a factor slightly smaller than K, leading to an overall relatively small increase in the encryption throughput, and decrease in the encryption latency, as shown in Figure 10.20. Because the combinational part of the circuit constitutes the majority of the circuit area, the total area of the encryption/decryption unit increases almost proportionally to the number of unrolled rounds, K. Additionally, the number of internal keys used in a single clock cycle increases by a factor of K, which in hardware implementations typically implies the almost proportional growth in the area used to store internal keys.

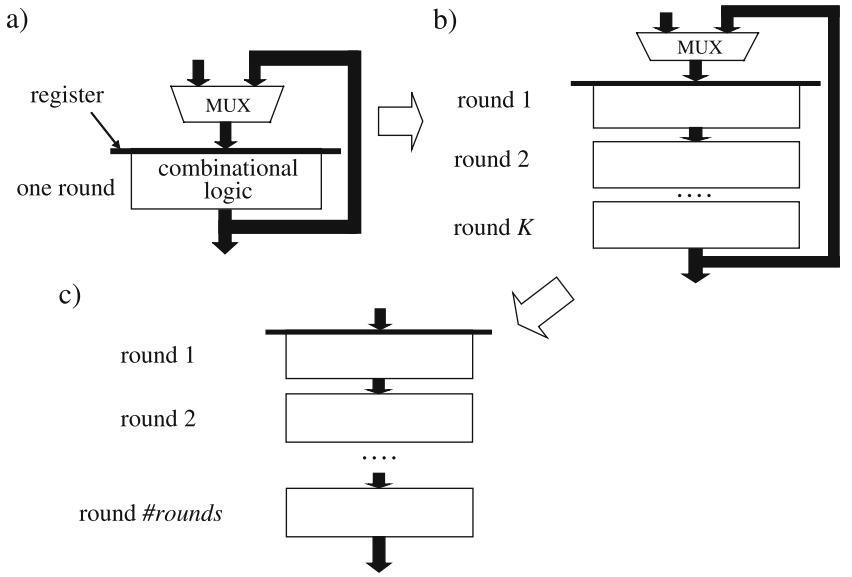

Fig. 10.19 Three hardware architectures suitable for feedback cipher modes: (**a**) basic iterative, (**b**) with partial loop unrolling, (**c**) with full loop unrolling.

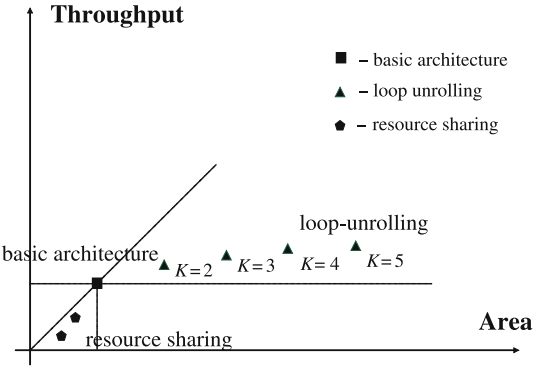

Fig. 10.20 Throughput vs. area characteristics of hardware architectures suitable for feedback cipher modes.

Architecture *with full loop unrolling* is shown in Figure 10.19c. The input multiplexer and the feedback loop are no longer necessary, leading to a small increase in the cipher speed and decrease in the circuit area compared to the partial loop unrolling with the same number of rounds unrolled.

In summary, loop unrolling enables increasing the circuit speed in both feedback and non-feedback operating modes. Nevertheless this increase is relatively small and incurs a large area penalty. As a result, choosing this architecture can be justified only for feedback cipher modes, where none other architecture offers speed greater than the basic iterative architecture, and only for implementations where large increase in the circuit area can be tolerated.

10.5.4 Pipelining

A traditional methodology for design of high-performance implementations of secret-key block ciphers operating in non-feedback cipher modes is shown in Figure 10.21. The basic iterative architecture, shown in Figure 10.21a, is implemented first and its speed and area determined. Based on these estimations, the number of rounds K that can be unrolled without exceeding the available circuit area is found. The number of unrolled rounds, K, must be a divisor of the total number of cipher rounds, #rounds. If the available circuit area is not large enough to fit all cipher rounds, architecture with partial outer-round pipelining, shown in Figure 10.21b, is applied. The difference between this architecture and the architecture with partial loop unrolling, shown in Figure 10.19b, is the presence of registers inside of the combinational logic on the boundaries between any two subsequent cipher rounds. As a result, K blocks of data can be processed by the circuit at the same time, with each of these blocks stored in a different register at the end of a clock cycle. This technique of parallel processing of multiple streams of data by the same circuit is called pipelining. The throughput and area of the circuit with partial outer-round

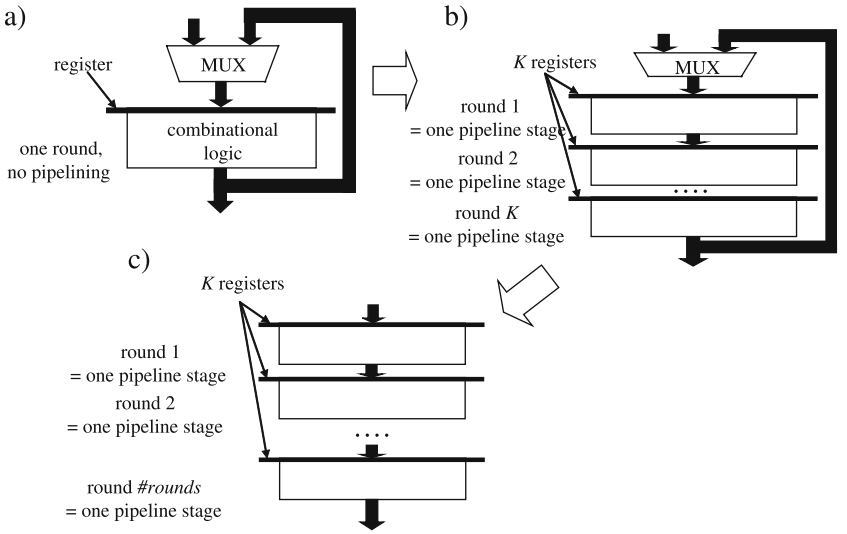

Fig. 10.21 Three hardware architectures used traditionally to implement non-feedback cipher modes: (**a**) basic iterative, (**b**) with partial outer-round pipelining, (**c**) with full outer-round pipelining.

pipelining increase proportionally to the value of K, as shown in Figure 10.23, the encryption/decryption latency remains the same as in the basic iterative architecture, as shown in Figure 10.24. If the available area is large enough to fit all cipher rounds, the feedback loop is no longer necessary and full outer-round pipelining, shown in Figure 10.21c, can be applied.

An optimized design methodology for implementing non-feedback cipher modes is shown in Figure 10.22. Before loop unrolling, the optimum number of pipeline registers is inserted inside of a cipher round, as shown in Figure 10.22b. The entire round, including internal pipeline registers is then repeated K times (see Figure 10.22c). The number of unrolled rounds K depends on the maximum available area or the maximum required throughput.

The primary advantage of the latter methodology is shown in Figure 10.23. Inserting registers inside of a cipher round significantly increases cipher throughput at the cost of only marginal increase in the circuit area. As a result, the throughput to area ratio increases until the number of internal pipeline stages reaches its optimum value k_{opt}. Inserting additional registers may still increase the circuit throughput, but the throughput to area ratio will deteriorate. The throughput to area ratio remains unchanged during the subsequent loop unrolling. The throughput of the circuit is given by

$$Throughput_{pipelined}(K,k) = \frac{K \cdot block\_size}{\#rounds \cdot T_{CLK_{inner\_round}}(k)} \tag{10.20}$$

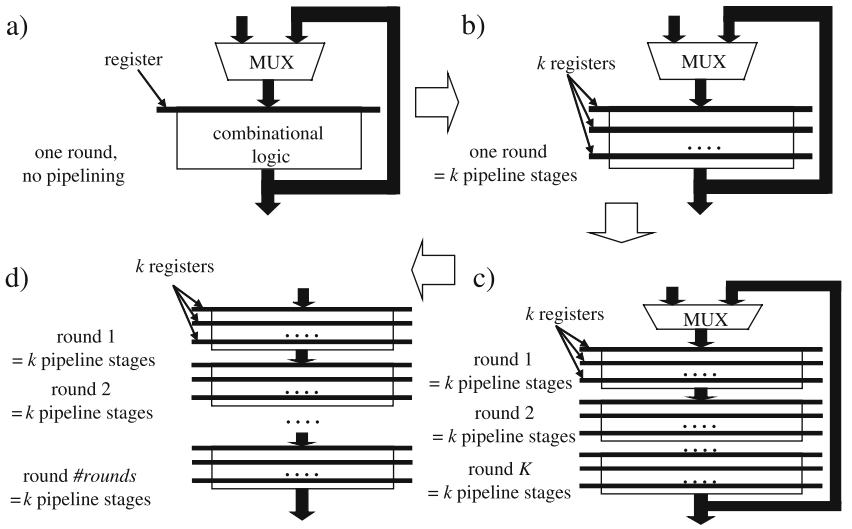

Fig. 10.22 Four hardware architectures suitable for non-feedback cipher modes: (**a**) basic iterative, (**b**) with inner-round pipelining, (**c**) with partial inner- and outer-round pipelining, (**d**) with full inner- and outer-round pipelining.

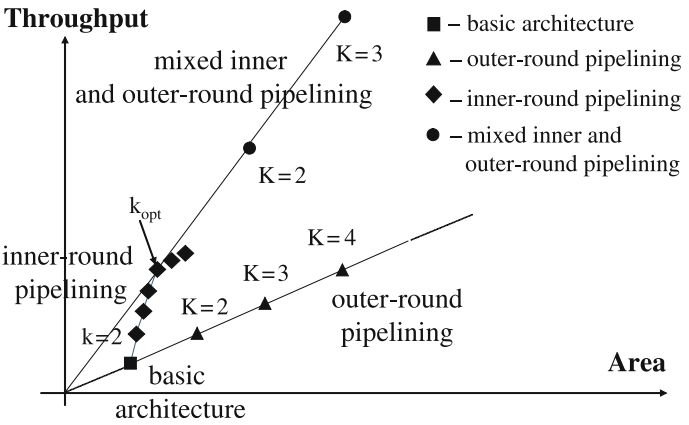

Fig. 10.23 Throughput vs. area characteristics of hardware architectures suitable for non-feedback cipher modes.

where k is the number of inner-round pipeline stages, K is the number of outer-round pipeline stages, and $T_{CLK_{inner\_round}}(k)$ is the clock period in the architecture with the k-stage inner-round pipelining. For a given limit in the circuit area, mixed inner- and outer-round pipelining shown in Figure 10.22c offers significantly higher throughput compared to the pure outer-round pipelining (see Figure 10.23).

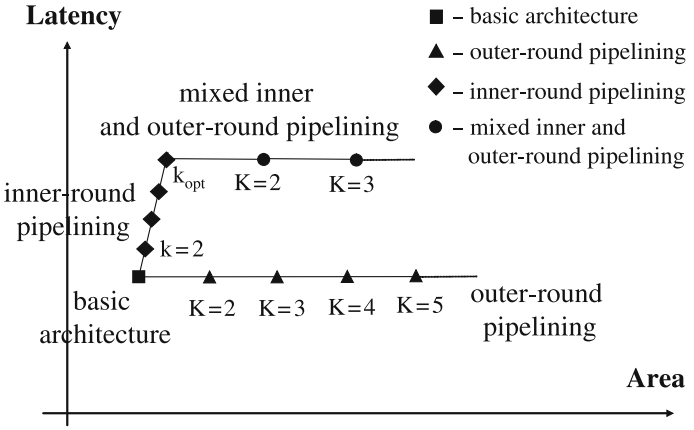

Fig. 10.24 Latency vs. area characteristics of hardware architectures suitable for non-feedback cipher modes.

When the limit on the circuit area is large enough, all rounds of the cipher can be unrolled, as shown in Figure 10.22d, leading to the throughput given by

$$Throughput_{fully\_pipelined}(K, k_{opt}) = \frac{block\_size}{T_{CLK_{inner\_round}}(k_{opt})} \tag{10.21}$$

where k_{opt} is the number of inner-round pipeline stages optimum from the point of view of the throughput to area ratio. The only side effect of our methodology is the increase in the encryption/decryption latency. This latency is given by

$$Latency_{fully\_pipelined}(K, k) = \#rounds \cdot k \cdot T_{CLK_{inner\_round}}(k) \tag{10.22}$$

This latency does not depend on the number of rounds unrolled, K. The increase in the encryption/decryption latency, typically in the range of single microseconds, usually does not have any major influence on the operation of the high-volume cryptographic system optimized for maximum throughput. This is particularly true for applications with a human operator present on at least one end of the secure communication channel.

The input/output timing characteristics of three basic secret-key cipher architectures are shown in Figure 10.25. In the basic iterative architecture, a new block of data must be fed into the system only once per *#rounds* clock cycles. In case of the inner-round pipelining, there are periods of time when the input must be fed into the cryptographic core every clock cycle, even though an average input/output throughput is much lower (Figure 10.25b). In the full mixed inner- and outer-round pipelining, input blocks are fed to the encryption unit every clock cycle (Figure 10.25c).

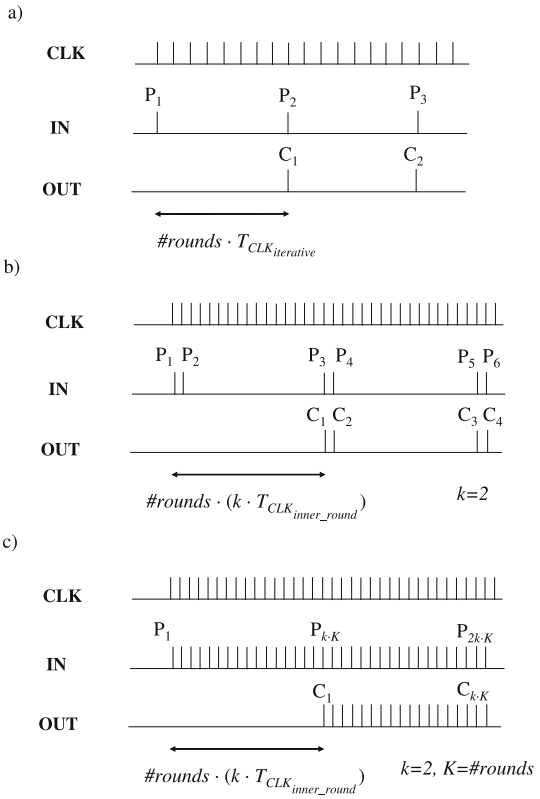

Fig. 10.25 Input/output timing characteristics of various architectures: (**a**) basic iterative architecture, (**b**) inner-round pipelining, (**c**) full inner- and outer-round pipelining.

10.5.5 Limits on the Maximum Clock Frequency of Pipelined Architectures

Throughput of the architecture with the mixed inner- and outer-round pipelining is directly proportional to the maximum clock frequency for the inner-round pipelining (see Equation (10.20)). The following factors may limit the maximum clock frequency,

$$f_{CLK_{inner\_round}}(k) = \frac{1}{T_{CLK_{inner\_round}}(k)} \tag{10.23}$$

in this architecture:

1. *delay of a single round divided by k*
   For small values of k, it is usually possible to divide the combinational portion of a single round into k stages with equal (or at least approximately equal) delays. The delay of a single stage, equal to the delay of a single round divided by k,

determines the minimum clock period of the circuit, $T_{CLK_{inner\_round}}(k)$, as shown in Figure 10.26a.

2. *delay of the largest indivisible operation*
   For some ciphers, when the number of internal pipeline stages k increases, it becomes more and more difficult to divide the combinational portion of a single round into stages with equal delays. At certain point, introducing additional internal registers to the circuit may require dividing an elementary operation of the cipher, such as an S-box or addition, into several stages. This division may be difficult to accomplish if the operation is performed using a standard library cell, look-up table, special carry propagate circuitry, or if the operation is so simple that it cannot be easily divided into less-complex atomic operations. This case is shown in Figure 10.26b.
3. *delay of the control unit*
   The control unit determines the data flow in the circuit. This unit is responsible for generating enable signals for all registers and memories in the circuit and address inputs for all memories and major multiplexers. The time necessary to generate and distribute these signals, counted from the rising edge of the clock, may be greater than the time necessary to propagate data between two adjacent registers in the pipeline, as shown in Figure 10.26c. This is especially true

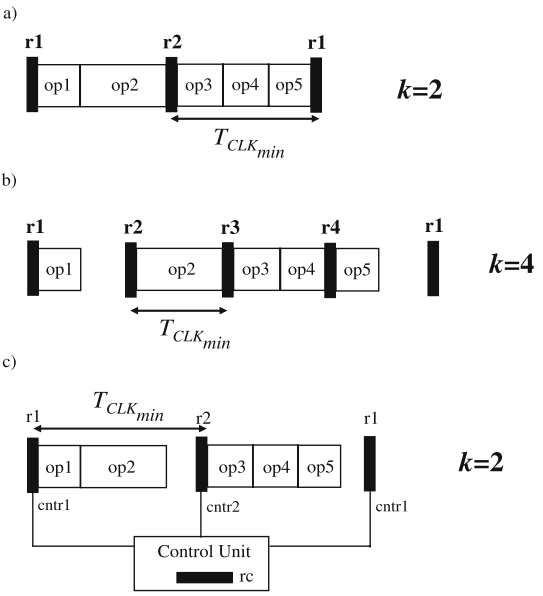

Fig. 10.26 Limits on the minimum clock period in the architecture with inner-round pipelining: (**a**) ideal situation, evenly divided round; (**b**) clock period limited by the largest indivisible operation; (**c**) clock period limited by the control unit, i.e., the time necessary to generate and distribute control signals.

for control signals with large fanouts distributed globally to every stage of the pipeline.

4. *limit on the maximum latency*
   Increasing the number of inner-round pipeline stages, k, increases the overall latency of the cipher, by a factor of approximately $(k-1)\cdot(t_P+t_{su})$, where t_P and t_{su} denote the propagation delay and the setup time of a register, respectively. This approximation does not take into account any changes in the routing (interconnect) delays. If the specification of the cryptographic system imposes a limit on the maximum latency, $Latency_{max}$, this limit may determine the maximum possible number of inner-round pipeline stages, k_{max}.

$$k_{max} \leq \frac{(Latency_{max} - Latency_{iterative})}{\#rounds \cdot (t_P + t_{su})} \tag{10.24}$$

5. *limit on the maximum input/output bandwidth*
   We define the input/output bandwidth as a frequency of an external clock used to control the transmission of data between the integrated circuit and an external environment. The input/output bandwidth necessary to sustain the throughput of the circuit working in the mixed inner- and outer-round pipelining is given by

$$Bandwidth = \frac{Throughput(K,k)}{bus\_width} = \frac{K}{\#rounds} \cdot \frac{block\_size}{bus\_width} \cdot f_{CLK_{inner\_round}}(k) \tag{10.25}$$

   where $f_{CLK_{innerround}}(k)$ is a frequency of the clock for a k-round inner-round pipelining. The circuit is assumed to have two independent ports of the width bus_width used for input and output, respectively. In case of using the same bus for both input and output, the bandwidth must be at least twice as high to sustain the same throughput. The maximum bandwidth may limit the maximum value of the product $K \cdot f_{CLK_{inner\_round}}(k)$ and thus the maximum number of inner- and outer-round pipeline stages.

10.5.6 Compact Architectures with Resource Sharing

For majority of ciphers, it is possible to significantly decrease the circuit area by time sharing of certain resources (e.g., S-boxes in AES). This is accomplished by using the same functional unit to process two (or more) parts of the data block in different clock cycles, as shown in Figure 10.27.

In Figure 10.27a, two parts of the data block, $D0$ and $D1$, are processed in parallel, using two independent functional units F. In Figure 10.27b, a single unit F is used to process two parts of the data block sequentially, during two subsequent

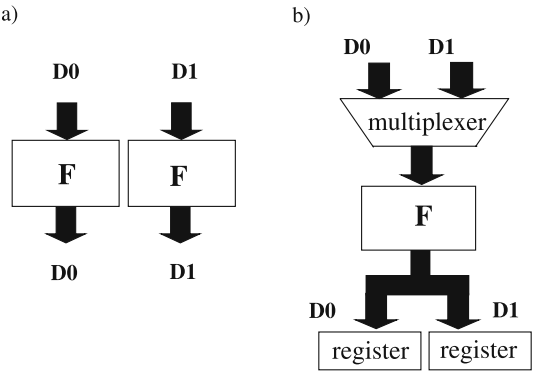

Fig. 10.27 Basic idea of resource sharing: (**a**) parallel execution of two instantiations of the functional unit F, no resource sharing; (**b**) resource sharing of the functional unit F.

clock cycles. This technique can be used to reduce the basic datapath in AES from 128 bits to 64 bits, 32 bits, and even 8 bits.

10.6 Implementation of Basic Operations of AES in Hardware

10.6.1 SubBytes and InvSubBytes

10.6.1.1 Look-Up Table

SubBytes is composed of 16 identical 8×8 S-boxes working in parallel. *InvSubBytes* is composed of the same number of 8×8-bit inverse S-boxes. Each of these S-boxes can be implemented independently using a 256×8-bit look-up table. A look-up table is implemented in digital systems using ROM (read-only memory). In this memory, input to an S-box is connected to the address lines, and the output is obtained at the data out bus.

Each of the AES *SubBytes* look-up tables is of the size of 256 bytes = 2048 bits = 2 kilobits. If encryption and decryption are implemented together within the same circuit, both uninverted and inverted 256 byte look-up tables can be placed within one 512 byte memory block. In this case, the most significant bit of an address is a control bit that distinguishes between encryption and decryption.

If a dual port ROM memory is available, which is often the case in FPGAs, the same memory can implement two S-boxes working in parallel.

In Xilinx FPGAs embedded memories are typically implemented as memories with synchronous output. This feature means that the output of the memory does not change until the next rising edge of the clock. This kind of synchronous ROM (read-only memory) is equivalent to a regular asynchronous ROM followed by a register. This feature of Xilinx Block RAMs determines uniquely the location of a register in the basic iterative architecture of AES.

10.6.1.2 Look-Up Table and Logic

The total size of the look-up tables necessary to implement both encryption and decryption can be reduced by a factor of two using knowledge of an internal structure of *SubBytes* and *InvSubBytes*, shown in Figure 10.5. In this case, only *inversion in* $GF(2^8)$ is implemented using look-up tables. These look-up tables are shared between the encryption and decryption units. The *affine transformation* and the *inverse affine transformation* can be implemented easily using an array of XOR gates. Up to 6-input (4-input) XOR gates are required in order to implement affine (inverse affine) transformation using one level of gates. If only 2-input XOR gates are available, up to three (two) layers of such gates might be necessary to implement the same transformations.

10.6.1.3 Logic Only

The amount of memory required to implement *SubBytes* and *InvSubBytes* can be reduced to zero by utilizing the internal logic structure of *inversion in* $GF(2^8)$. This approach makes particular sense for ASIC implementations, in which memory is typically costly in terms of the circuit area.

In FPGAs, memory blocks are always present independently whether they are used or not, but their replacement by logic may be still justified. For example, memory might be already used to implement some other functions, such as input/output buffers. Additionally, in case of deeply pipelined architectures (see Section 10.5.4), memory-based implementation can impose an artificial restriction on the minimum clock period (as described in Section 10.5.5), while the logic-based implementation can be further pipelined.

The basic idea of the logic-only implementation is to notice that inversion in $GF(2^8)$ can be decomposed into a sequence of operations in $GF(2^4)$ (including addition, multiplication, and inversion), as shown in Figure 10.29. Similarly, operations in $GF(2^4)$ can be expressed in terms of operations in $GF(2^2)$ (see Figures 10.30, 10.32, and 10.35) and operations in $GF(2^2)$ in terms of operations in $GF(2)$ (see Figures 10.33, 10.34, 10.36, and 10.37). The operations in $GF(2)$ can be implemented using simple XOR gate (addition) and AND gate (multiplication). An inverse of 1 in $GF(2)$ is 1, and the inverse of 0 does not exist. Thus, the entire *inversion in* $GF(2^8)$ can be decomposed into a logic circuit composed of XOR and AND gates only.

The complexity (number of equivalent logic gates) and critical path (delay) of this circuit depend on the choice of the specific representation for each field $GF(2^{2k})$ using components of the underlying field $GF(2^k)$, for $k = 4, 2$, and 1. The initial choice of the specific representations was provided by Satoh et al. [33]. This choice was later examined by Mentens et al. [28]. The authors compared 64 different polynomial basis representations, and found a representation giving a 5% improvement over [33].

Canright [3, 4] has extended this comparison to include normal basis representations of the components of the fields $GF(2^{2k})$. He investigated a total of 432 different representation choices and concluded that his best choice gives a 20% improvement in terms of the total gate count compared to [33].

The details of the optimum design are shown in the hierarchical form in Figures 10.28, 10.29, 10.30, 10.31, 10.32, 10.33, 10.34, 10.35, 10.36 and 10.37. The top level design of the *SubBytes*/*InvSubBytes* circuit is shown in Figure 10.28.

X is an 8×8 basis conversion matrix, which changes the Galois field representation from the optimum representation used for internal computations within the $GF(2^8)$ inverter to the standard AES polynomial representation used in the remaining calculations. X^{-1} is a matrix describing the conversion in the opposite direction. M is an 8×8 matrix and $b = \{63\}$ is an 8×1 bit vector, where $y' = M \cdot y + b$ is an equation describing the affine transformation of *SubBytes*.

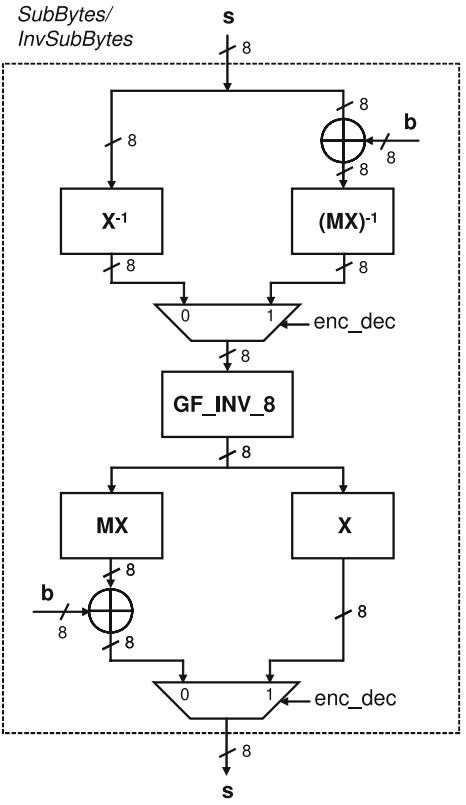

Fig. 10.28 Implementation of *SubBytes* and *InvSubBytes* using logic only, according to [3, 4]. The notation follows conventions introduced in [3]. *enc_dec* is a select signal equal to 0 for encryption and 1 for decryption. X is an 8×8 basis conversion matrix, M is an 8×8 matrix, and b is an 8×1 bit vector, where $y' = M \cdot y + b$, with $b = \{63\}$, is an equation describing the affine transformation of *SubBytes*.

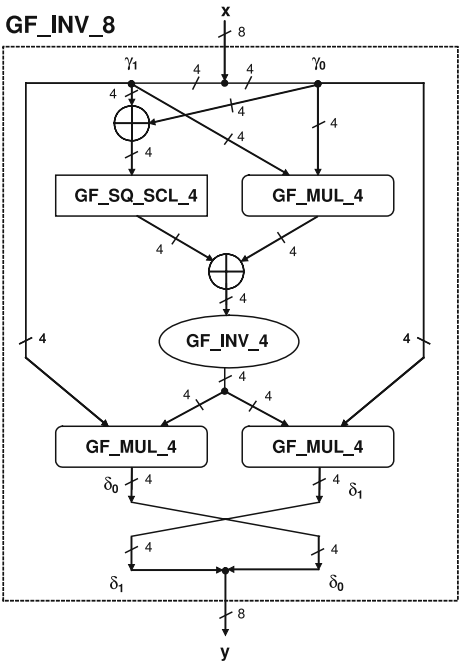

Fig. 10.29 $GF(2^8)$ inverter, GF_INV_8. Notation: GF_INV_4—$GF(2^4)$ inverter (see Figure 10.30), GF_MUL_4—$GF(2^4)$ multiplier (see Figure 10.32), and GF_SQ_SCL_4—$GF(2^4)$ squarer and scaler (see Figure 10.35).

The data path for encryption (see Figure 10.28) includes conversion from the standard polynomial representation to the internal representation, X^{-1}, inversion in $GF(2^8)$, conversion back to the standard representation, X, combined with the multiplication by matrix M of affine transformation, MX, followed by the addition of the vector b of the affine transformation. Thus, the entire *SubBytes* transformation can be described by the equation

$$s' = (MX) \cdot (X^{-1}s)^{-1} + b \tag{10.26}$$

The data path for decryption starts from adding the vector b, followed by the multiplication by M^{-1} and X^{-1}, combined into a product $X^{-1}M^{-1} = (MX)^{-1}$, followed by inversion in $GF(2^8)$, and multiplication by X. Thus, the entire *InvSubBytes* transformation can be described by the equation

$$s' = X \cdot ((MX)^{-1}(s+b))^{-1} \tag{10.27}$$

In both cases, the convention for the order of bits within vectors s' and s is as given below in Equation (10.28).

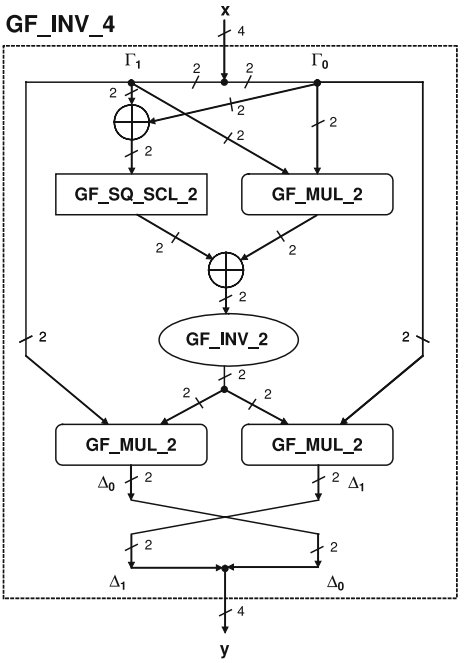

Fig. 10.30 $GF(2^4)$ inverter, GF_INV_4. Notation: GF_INV_2—$GF(2^2)$ inverter (see Figure 10.31), GF_MUL_2—$GF(2^2)$ multiplier (see Figure 10.33), and GF_SQ_SCL_2—$GF(2^2)$ squarer and scaler (see Figure 10.36).

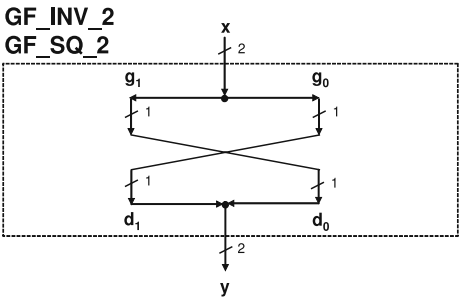

Fig. 10.31 $GF(2^2)$ inverter, GF_INV_2; equivalent to $GF(2^2)$ squarer, GF_SQ_2.

$$s' = \begin{bmatrix} s'_7 \\ s'_6 \\ s'_5 \\ s'_4 \\ s'_3 \\ s'_2 \\ s'_1 \\ s'_0 \end{bmatrix} \qquad s = \begin{bmatrix} s_7 \\ s_6 \\ s_5 \\ s_4 \\ s_3 \\ s_2 \\ s_1 \\ s_0 \end{bmatrix} \tag{10.28}$$

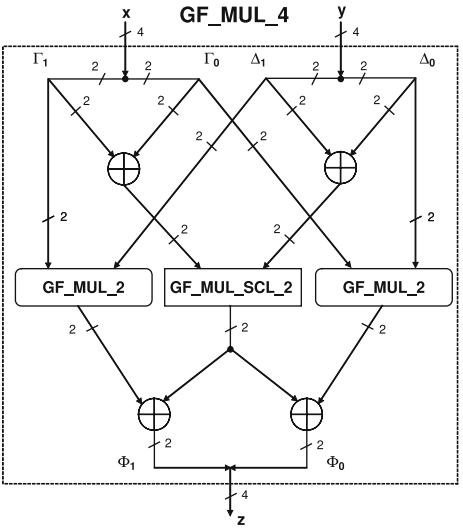

Fig. 10.32 $GF(2^4)$ multiplier, GF_MUL_4. Notation: GF_MUL_2—$GF(2^2)$ multiplier (see Figure 10.33) and GF_MUL_SCL_2—$GF(2^2)$ multiplier and scaler (see Figure 10.34).

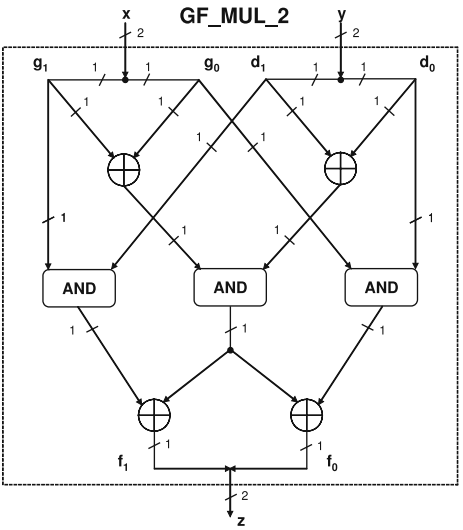

Fig. 10.33 $GF(2^2)$ multiplier, GF_MUL_2.

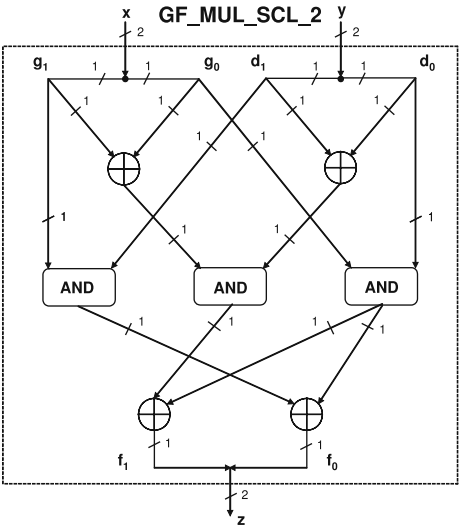

Fig. 10.34 $GF(2^2)$ multiplier and scaler, GF_MUL_SCL_2. It performs multiplication Nxy, where x, y, N $\in GF(2^2)$, x and y are input variables, and N is a constant.

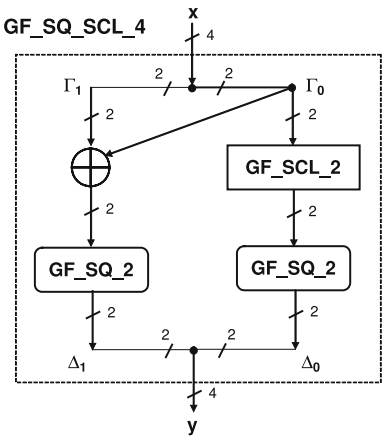

Fig. 10.35 $GF(2^4)$ squarer and scaler, GF_SQ_SCL_4. It performs operation νx^2, where x, $\nu \in GF(2^4)$, x is an input variable, and ν is a constant, $\nu = 0 \cdot z + N^2$. Notation: GF_SQ_2—$GF(2^2)$ squarer (see Figure 10.31) and GF_SCL_2—$GF(2^2)$ scaler (see Figure 10.37).

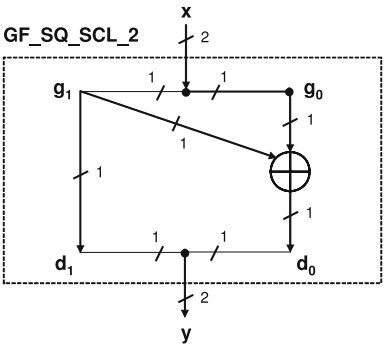

Fig. 10.36 $GF(2^2)$ squarer and scaler, GF_SQ_SCL_2. It performs operation Nx^2, where x, N $\in GF(2^2)$, x is an input variable, and N is a constant.

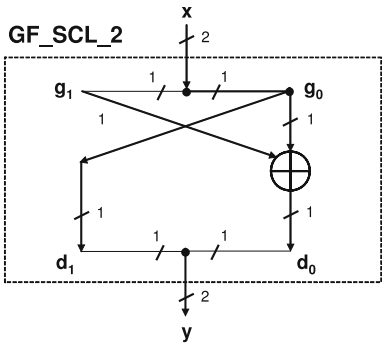

Fig. 10.37 $GF(2^2)$ scaler, GF_SCL_2. It performs operation Nx, where x, N $\in GF(2^2)$, x is an input variable, and N is a constant.

The exact forms of matrices X^{-1}, $(MX)^{-1}$, MX, and X are given by Equations 10.29–10.32.

$$X^{-1} = \begin{bmatrix} 1&1&1&0&0&1&1&1 \\ 0&1&1&1&0&0&0&1 \\ 0&1&1&0&0&0&1&1 \\ 1&1&1&0&0&0&0&1 \\ 1&0&0&1&1&0&1&1 \\ 0&0&0&0&0&0&0&1 \\ 0&1&1&0&0&0&0&1 \\ 0&1&0&0&1&1&1&1 \end{bmatrix} \tag{10.29}$$

$$(MX)^{-1} = \begin{bmatrix} 1\,0\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0\,1\,1 \\ 0\,1\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,1 \\ 1\,1\,0\,1\,0\,0\,0\,0 \\ 1\,0\,1\,0\,0\,1\,0\,0 \\ 0\,0\,0\,1\,1\,0\,0\,1 \\ 0\,1\,1\,1\,0\,0\,1\,1 \end{bmatrix} \tag{10.30}$$

$$MX = \begin{bmatrix} 0\,0\,1\,0\,1\,0\,0\,0 \\ 1\,0\,0\,0\,1\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,0\,1 \\ 1\,0\,1\,0\,1\,0\,0\,0 \\ 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,0\,1\,1\,0\,1 \\ 0\,0\,1\,1\,0\,0\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \end{bmatrix} \tag{10.31}$$

$$X = \begin{bmatrix} 0\,0\,0\,1\,0\,0\,1\,0 \\ 1\,1\,1\,0\,1\,0\,1\,1 \\ 1\,1\,1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,0\,0\,1\,0 \\ 0\,1\,1\,1\,1\,1\,1\,0 \\ 1\,0\,1\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,1\,0\,0 \end{bmatrix} \tag{10.32}$$

As shown in Figure 10.29, the $GF(2^8)$ inverter can be decomposed into one $GF(2^4)$ inverter, three $GF(2^4)$ multipliers, two $GF(2^4)$ adders (4-bit XOR gates), and one $GF(2^4)$ squarer and scaler. The $GF(2^4)$ inverter (see Figure 10.30) looks almost the same as $GF(2^8)$ inverter, with component operations in $GF(2^4)$ replaced by operations in $GF(2^2)$. The $GF(2^2)$ does not involve any logic, as it is equivalent to squaring and thus to a circular rotation by one bit position, as shown in Figure 10.31.

The $GF(2^4)$ multiplier can be decomposed into two $GF(2^2)$ multipliers, one $GF(2^2)$ multiplier/scaler, and four $GF(2^2)$ adders (2-bit XOR gates), as shown in Figure 10.32. The $GF(2^2)$ multiplier (see Figure 10.33) has the same structure, with the multipliers and the multiplier/scaler in $GF(2^2)$ replaced by AND gates (multipliers in $GF(2)$).

The $GF(2^4)$ squarer and scaler, shown in Figure 10.35, is an optimized version of the circuit that performs squaring in $GF(2^4)$ followed by multiplication by a specially chosen constant, ν (see [3] for more information about the optimum choice of ν). Similarly, the $GF(2^2)$ multiplier and scaler, shown in Figure 10.34, combines multiplication of two variables in $GF(2^2)$ followed by multiplication by a specifically chosen constant N. The $GF(2^2)$ squarer and scaler, shown in Figure 10.36,

combines squaring with multiplication by N. The multiplication by a constant N can be implemented using just one XOR gate, as shown in Figure 10.37.

Without any further optimizations, the $GF(2^8)$ inverter includes 88 two-input XOR gates and 36 two-input AND gates, and its critical path passes through 14 two-input XOR gates and 4 two-input AND gates. In an FPGA implementation based on 4-input look-up tables (LUTs), $GF(2^4)$ inverter, $GF(2^4)$ squarer and scaler, the $GF(2^2)$ multiplier, and the $GF(2^2)$ multiplier and scaler can all be implemented using look-up tables, so no lower level operations are required. The total number of look-up tables (LUTs) required to implement the $GF(2^8)$ inverter becomes 58, and its critical path delay is equal to the delay of eight logic levels (LUTs).

As explained in [3], multiple further optimizations based on resource sharing and optimum gate type choice can be used to further reduce circuit area. The derivation of the minimum-area circuit, together with a detailed justification of design choices can be found in [3, 4]. The appendices of [3] contain the corresponding C program, which can be used as a source of test vectors, and the manually optimized Verilog code, which can be used as a starting point for a hardware implementation.

10.6.2 MixColumns and InvMixColumns

10.6.2.1 Basic Implementation

The *MixColumns* transformation can be expressed as a matrix multiplication in the Galois field $GF(2^8)$ as shown in Equation (10.14). Each symbol in this equation (such as b_i, a_i, 03) represents an 8-bit element of the Galois field. Each byte of the result of a matrix multiplication (10.14) is an XOR of four bytes representing the Galois field product of a byte a_0, a_1, a_2, or a_3 by a respective constant. As a result, the entire *MixColumns* transformation can be performed using two layers of XOR gates, with up to 3-input gates in the first layer and 4-input gates in the second layer. In FPGAs, each of these XOR operations requires only one 4-bit look-up table.

The *InvMixColumns* transformation can be expressed as a matrix multiplication in the Galois field $GF(2^8)$ as shown in Equation (10.15).

The primary differences, compared to *MixColumns*, are the larger hexadecimal values of the matrix coefficients. Multiplication by these constant elements of the Galois field leads to the more complex dependence between the bits of a variable input and the bits of a respective product.

The entire *InvMixColumns* transformation can be performed using two layers of XOR gates, with up to 6-input gates in the first layer and 4-input gates in the second layer. Because of the use of gates with larger number of inputs, the *InvMixColumns* transformation has a longer critical path compared to the *MixColumns* transformation, and as a result, the decryption circuit imposes a limit on the minimum clock period of the entire encryption/decryption unit.

10.6.2.2 Implementations with Resource Sharing

Coefficients of $d(X)$ are more complex than coefficients of $c(X)$; therefore, decryption is always slower than encryption [8]. Moreover, hardware structures implementing *InvMixColumns* are always larger. To reduce hardware cost, the *InvMixColumns* matrix can be decomposed in such a way that some portion of the hardware will be re-used for *MixColumns* implementation. Since *MixColumns* and *InvMixColumns* functions are defined on 32-bit words, we will call this decomposition the word-level resource sharing. There are two possible approaches: parallel and serial *InvMixColumns* decomposition. In addition to word-level sharing, resources can be shared on a byte level and on a bit level [14].

Parallel InvMixColumns Decomposition

Parallel *InvMixColumns* decomposition was first proposed by J. Wolkerstorfer in [43]. It is based on the observation that $d(X)$ can be expressed using $c(X)$ in the following way:

$$d(X) = c(X) + e(X) \tag{10.33}$$

where $e(X)$ is an extension polynomial defined as

$$e(X) = x^3X^3 + (x^3 + x^2)X^2 + x^3X + (x^3 + x^2) \tag{10.34}$$

or in hexadecimal format

$$e(X) = \{08\}X^3 + \{0C\}X^2 + \{08\}X + \{0C\} \tag{10.35}$$

Equation (10.15) can thus be written in the form

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = ([C] + [E]) \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.36}$$

where

$$[C] = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \tag{10.37}$$

and

$$[E] = \begin{bmatrix} 0C\ 08\ 0C\ 08 \\ 08\ 0C\ 08\ 0C \\ 0C\ 08\ 0C\ 08 \\ 08\ 0C\ 08\ 0C \end{bmatrix} \tag{10.38}$$

When both *MixColumns* and *InvMixColumns* have to be implemented in the same piece of hardware, the matrix $[E]$ from Equation (10.36) is added to the matrix $[C]$ only during decryption, and thus the matrix $[C]$ is shared by both encryption and decryption processes. Implementation of such a structure in the hardware can be viewed as implementation of four identical blocks A (see Figure 10.38), each giving an output of one byte of $b(X)$. Inputs of these blocks are permuted in the same way as coefficients of the *MixColumns* (or *InvMixColumns*) matrix (see Figure 10.38b).

Serial InvMixColumns Decomposition

MixColumns and *InvMixColumns* are derived as mutual inverses, and therefore, they are related such that $c(X) \cdot d(X) = 1$. There exists another relationship between $c(X)$ and $d(X)$: the inverse $d(X)$ of the polynomial $c(X)$ in the ring R is given by the formula

$$d(X) = c^{-1}(X) = c^3(X) \tag{10.39}$$

Equation (10.39) suggests that the *InvMixColumns* operation can be realized by repeating *MixColumns* three times. For hardware implementations, Equation (10.39) can be expressed as

$$d(X) = c(X) \cdot c^2(X) \tag{10.40}$$

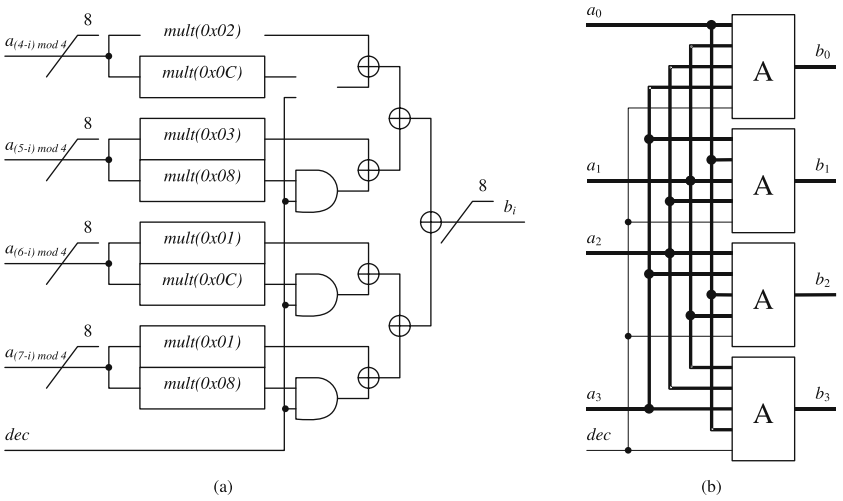

Fig. 10.38 Resource sharing based on parallel *InvMixColumns* decomposition: (**a**) line multiplication block A, (**b**) overall structure.

where

$$c^2(X) = x^2X^2 + x^2 + 1 \tag{10.41}$$

Therefore, the *InvMixColumns* function can be implemented using the *MixColumns* function and the $c^2(X)$ polynomial. The $c^2(X)$ polynomial can be expressed with coefficients in hexadecimal format:

$$c^2(X) = \{00\}X^3 + \{04\}X^2 + \{00\}X + \{05\} \tag{10.42}$$

Following Equation (10.40) the Equation (10.15) can be expressed as

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = ([C] \cdot [F]) \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10.43}$$

where $[C]$ has the same meaning as in Equation (10.37) and

$$[F] = \begin{bmatrix} 05\ 00\ 04\ 00 \\ 00\ 05\ 00\ 04 \\ 04\ 00\ 05\ 00 \\ 00\ 04\ 00\ 05 \end{bmatrix} \tag{10.44}$$

Comparing $c(X)$ and $c^2(X)$ we see that $c^2(X)$ is much simpler in implementation than $c(X)$ because two of its coefficients are equal to zero. Multiplication of matrices represents a serial arrangement of corresponding modules with common input (see Figure 10.39a) or common output (see Figure 10.39b). Using the same approach as for the parallel *InvMixColumns* decomposition, the hardware structure with a common input can be implemented using four instances of two types of blocks A and B (see Figure 10.40). This structure implements multiplication of the 4-byte input by one line of $c(X)$ and $c^2(X)$.

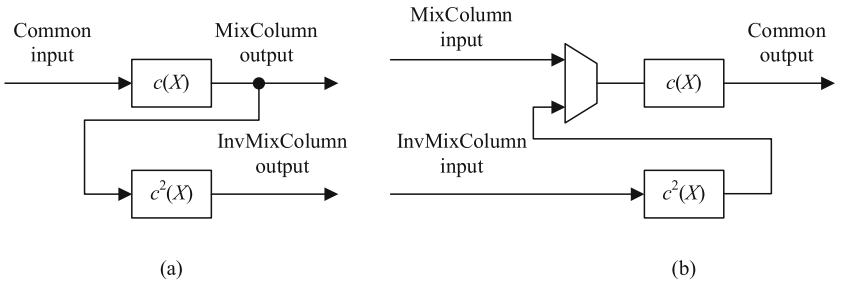

Fig. 10.39 Resource sharing based on serial *InvMixColumns* decomposition with (**a**) common input, (**b**) common output.

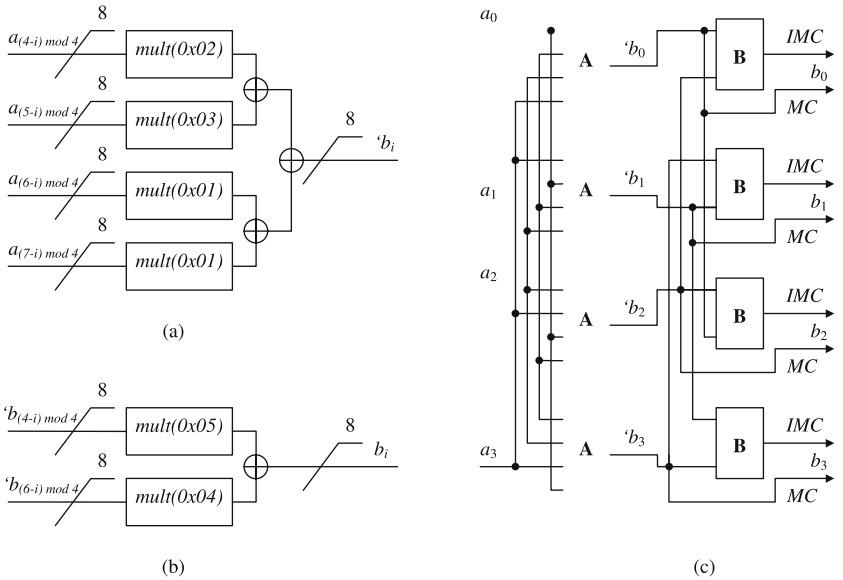

Fig. 10.40 Serial *InvMixColumns* decomposition with common input: (**a**) line multiplication block A, (**b**) line multiplication block B, (**c**) overall structure.

10.7 Hardware Architectures of a Single Round of AES

10.7.1 S-Box-Based Architecture

The block diagrams of the encryption/decryption unit based on S-boxes in the basic iterative architecture is shown in Figure 10.41. Only register R1 is present in the basic iterative architecture. The best placement for this register is either before or after the combined *SubBytes* and *InvSubBytes* transformation, where encryption and decryption data paths converge. The critical path is located in the decryption circuit and includes *InvShiftRows* (interconnects), *AddRoundKey* (XOR operation), *InvMixColumns*, a 3-to-1 multiplexer, and *InvSubBytes*. In this architecture, 11, 13, and 15 clock cycles are required in order to process one block of data for 128-, 192-, and 256-bit keys, respectively.

In Figure 10.42 several registers have been added in order to create an architecture with two stages of inner-round pipelining. The location of these registers has been chosen in such a way to divide the critical path into two approximately equal paths. This way the minimum clock period should be equal to approximately half of the clock period for the basic iterative architecture, allowing processing of data with approximately twice as high throughput in non-feedback cipher modes.

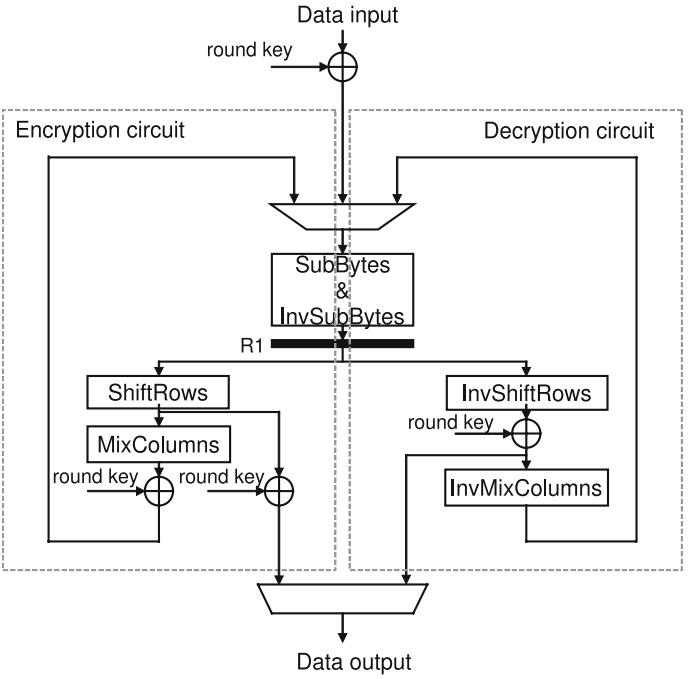

Fig. 10.41 Typical S-box-based AES basic iterative architecture.

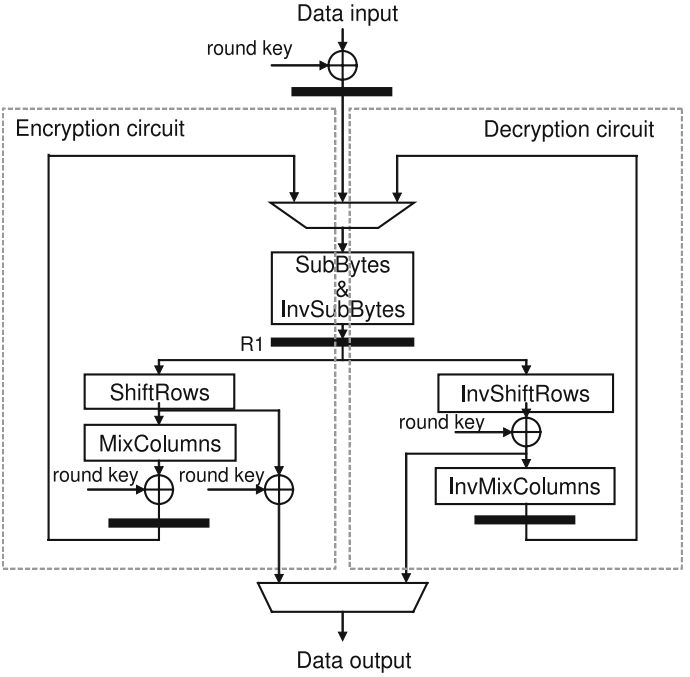

Fig. 10.42 S-box-based AES architecture with two stages of inner-round pipelining.

10.7.2 T-Box-Based Architecture

T-box-based algorithm for implementing AES was first proposed in [7] for a software implementation using 32-bit microprocessors. In [13], this approach was adapted for hardware implementations.

The T-box approach allows the computation of the entire round of AES using only table look-ups and XOR operations. In Figure 10.43, we show the mathematical description of AES round operations. This representation leads to the derivation of the T-box representation of an AES encryption round, as shown in Figure 10.44. The precomputed tables, called T-boxes, represent the combined application of the *SubBytes* and *MixColumns* transformations. They are defined as follows:

$$T_0[a] = \begin{bmatrix} 02 \cdot S[a] \\ S[a] \\ S[a] \\ 03 \cdot S[a] \end{bmatrix} \qquad T_1[a] = \begin{bmatrix} 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \\ S[a] \end{bmatrix} \tag{10.45}$$

$$T_2[a] = \begin{bmatrix} S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \\ S[a] \end{bmatrix} \qquad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ 03 \cdot S[a] \\ 02 \cdot S[a] \end{bmatrix} \tag{10.46}$$

SubBytes

$$\mathbf{b}_{i,j} = \mathbf{S}[\mathbf{a}_{i,j}]$$

ShiftRows

$$\begin{bmatrix} \mathbf{c}_{0,j} \\ \mathbf{c}_{1,j} \\ \mathbf{c}_{2,j} \\ \mathbf{c}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{0,j} \\ \mathbf{b}_{1,j+1} \\ \mathbf{b}_{2,j+2} \\ \mathbf{b}_{3,j+3} \end{bmatrix}$$

sums mod 4

MixColumns

$$\begin{bmatrix} \mathbf{d}_{0,j} \\ \mathbf{d}_{1,j} \\ \mathbf{d}_{2,j} \\ \mathbf{d}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{02} & \mathbf{03} & \mathbf{01} & \mathbf{01} \\ \mathbf{01} & \mathbf{02} & \mathbf{03} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{03} & \mathbf{01} & \mathbf{01} & \mathbf{02} \end{bmatrix} \begin{bmatrix} \mathbf{c}_{0,j} \\ \mathbf{c}_{1,j} \\ \mathbf{c}_{2,j} \\ \mathbf{c}_{3,j} \end{bmatrix}$$

AddRoundKey

$$\begin{bmatrix} \mathbf{e}_{0,j} \\ \mathbf{e}_{1,j} \\ \mathbf{e}_{2,j} \\ \mathbf{e}_{3,j} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_{0,j} \\ \mathbf{d}_{1,j} \\ \mathbf{d}_{2,j} \\ \mathbf{d}_{3,j} \end{bmatrix} \oplus \begin{bmatrix} \mathbf{k}_{0,j} \\ \mathbf{k}_{1,j} \\ \mathbf{k}_{2,j} \\ \mathbf{k}_{3,j} \end{bmatrix}$$

Fig. 10.43 Mathematical description of AES round operations.

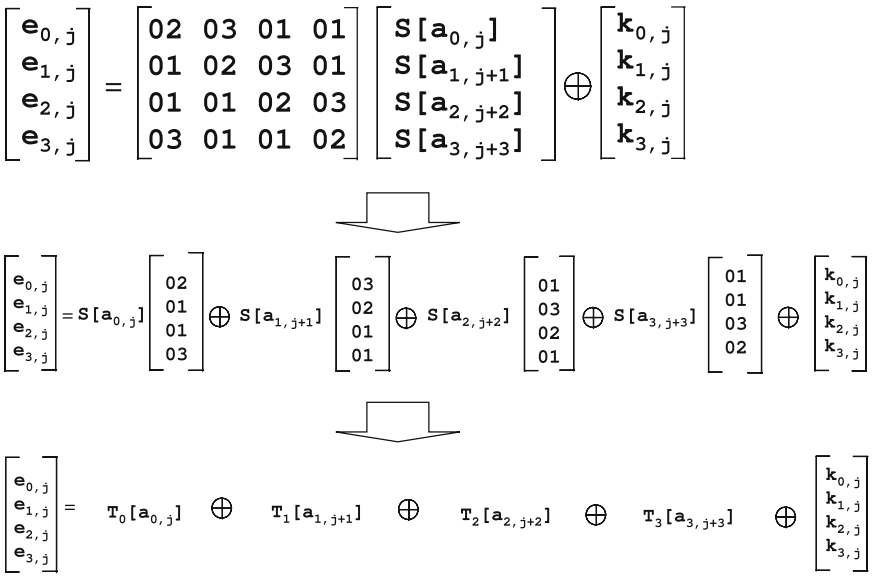

Fig. 10.44 Mathematical derivation of the T-box representation of an AES encryption round.

Compared to the S-box tables, which are of the size of 8×8 bits, the T-box tables are of the size of 8×32 bits. The entire 32-bit column of an output of a single round of AES, e_j, can be computed using the following formula:

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}] \oplus K_j \tag{10.47}$$

where T_0, T_1, T_2, T_3 are the precomputed 8×32-bit look-up tables, and K_j is a jth word of a round key K. All indices $j+1$, $j+2$, $j+3$ are computed modulo 4. For example,

$$e_2 = T_0[a_{0,2}] \oplus T_1[a_{1,3}] \oplus T_2[a_{2,0}] \oplus T_3[a_{3,1}] \oplus K_2 \tag{10.48}$$

In Figure 10.45, we show in a graphical form an example of computing the value of the column e_2 of the encryption round output using T-box approach.

Since *MixColumns* operation is not performed in the last round of encryption, the last round needs to be treated in a special way. In this round, S-boxes need to be used instead of T-boxes. Fortunately, no additional memory space is needed to implement S-boxes, as 1-byte outputs of the S-box transformation can be easily extracted from the 4-byte outputs of the T-box transformation corresponding to the same 1-byte input. For example,

$$S[a] = byte(1, T_0[a]) = byte(2, T_1[a]) = byte(3, T_2[a]) = byte(0, T_3[a]) \tag{10.49}$$

where $byte(n, X)$ represents the nth byte of a variable X.

In Figure 10.46, a block diagram of the encryption function based on the use of T-box look-up tables is shown. The encryption input consists of 16 bytes, arranged as follows:

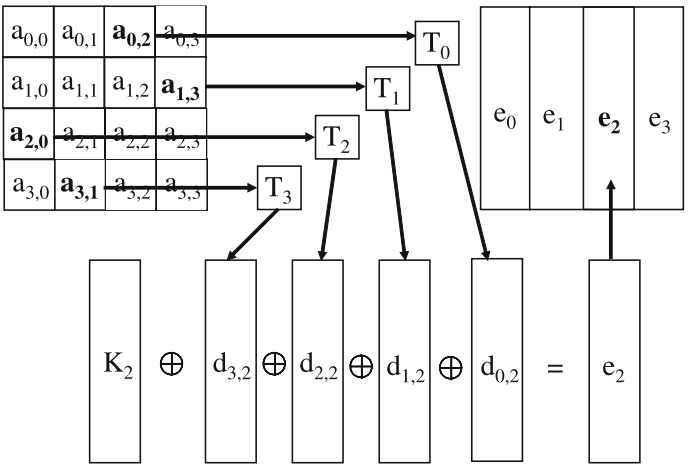

Fig. 10.45 An example of computing the value of the column e_2 of the encryption round output using T-box approach.

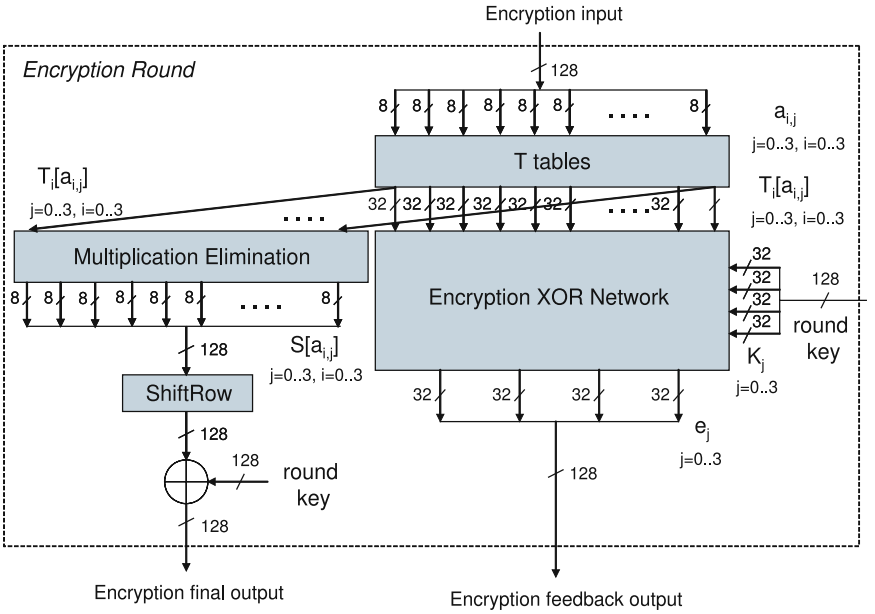

Fig. 10.46 Block diagram of the T-box-based AES encryption round.

$$a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, a_{2,1}, a_{3,1}, a_{0,2}, a_{1,2}, a_{2,2}, a_{3,2}, a_{0,3}, a_{1,3}, a_{2,3}, a_{3,3} \tag{10.50}$$

Each of these bytes is an input to the appropriate look-up table $T_i[a_{i,j}]$, with an index the same as a row index of the byte $a_{i,j}$. Sixteen such look-up tables working in parallel form a functional block called *T tables*. *Encryption XOR Network* is a block

that based on 16 T-box tables outputs $T_i[a_{i,j}]$ ($j = 0...3$, $i = 0...3$) and four words of the round key K, K_j ($j = 0...3$) computes four 32-bit columns of the encryption round output e_j. The exact dependence between inputs and outputs of this block is given by Equation 10.47. Finally, the *Multiplication Elimination* extracts values of the S-box outputs $S[a_{i,j}]$ based on the values of the T-box outputs $T_i[a_{i,j}]$. The operation of this block is given by Equation 10.49, and the block itself does not involve any logic, just routing.

A similar derivation can be performed in order to represent decryption using a separate set of inverse T-boxes, ${T_0}^{-1}$, ${T_1}^{-1}$, ${T_2}^{-1}$, ${T_3}^{-1}$. These inverse T-box tables are defined as follows:

$$T_0^{-1}[a] = \begin{bmatrix} 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \end{bmatrix} \qquad T_1^{-1}[a] = \begin{bmatrix} 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \\ 0D \cdot S[a] \end{bmatrix} \tag{10.51}$$

$$T_2^{-1}[a] = \begin{bmatrix} 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \\ 09 \cdot S[a] \end{bmatrix} \qquad T_3^{-1}[a] = \begin{bmatrix} 09 \cdot S[a] \\ 0D \cdot S[a] \\ 0B \cdot S[a] \\ 0E \cdot S[a] \end{bmatrix} \tag{10.52}$$

The equation describing an output of the decryption round is given below:

$$d_j = T_0^{-1}[a_{0,j}] \oplus T_1^{-1}[a_{1,j+3}] \oplus T_2^{-1}[a_{2,j+2}] \oplus T_3^{-1}[a_{3,j+1}] \oplus IMC(K_j) \tag{10.53}$$

where T_0^{-1}, T_1^{-1}, T_2^{-1}, T_3^{-1} are the precomputed 8×32-bit look-up tables, K_j is a jth word of a round key K, and *IMC* is the *InvMixColumns* transformation. All indices $j+3$, $j+2$, $j+1$ are computed modulo 4.

In the last round of decryption the *InvMixColumns* operation is not executed. As a result, the outputs of the *InvSubBytes* transformation, $S^{-1}[a]$, must be computed. In this case, however, the computation of $S^{-1}[a]$ as a function of $T_i^{-1}[a]$ requires some extra logic that implements multiplication by a constant in $GF(2^8)$ [13]. For example, given the value of $T_0^{-1}[a]$, $S^{-1}[a]$ can be computed as follows:

$$\begin{aligned} S^{-1}[a] &= 0E^{-1} \cdot byte(0, T_0^{-1}[a]) = 09^{-1} \cdot byte(1, T_0^{-1}[a]) \\ &= 0D^{-1} \cdot byte(2, T_0^{-1}[a]) = 0B^{-1} \cdot byte(3, T_0^{-1}[a]) \\ &= E5 \cdot byte(0, T_0[a]) = 4F \cdot byte(1, T_0[a]) \\ &= E1 \cdot byte(2, T_0[a]) = C0 \cdot byte(3, T_0[a]) \end{aligned} \tag{10.54}$$

where $byte(n, X)$ represents the nth byte of a variable X. Based on the above equations, each bit of $S^{-1}[a]$ can be computed using four different equations, each giving exactly the same value. As a result, for each bit, we can choose an equation with the smallest number of terms. If we do that, we can express the bits of $S^{-1}[a] = (s_7^{-1} s_6^{-1} s_5^{-1} s_4^{-1} s_3^{-1} s_2^{-1} s_1^{-1} s_0^{-1})$ as follows:

$$
\begin{aligned}
x &= byte(0, T_0^{-1}[a]) \\
y &= byte(1, T_0^{-1}[a]) \\
z &= byte(3, T_0^{-1}[a])
\end{aligned} \tag{10.55}
$$

$$
\begin{aligned}
s_7^{-1} &= y_7 \oplus y_4 \oplus y_1 \\
s_6^{-1} &= y_6 \oplus y_3 \oplus y_0 \\
s_5^{-1} &= y_5 \oplus y_2 \\
s_4^{-1} &= y_4 \oplus y_1 \\
s_3^{-1} &= z_3 \oplus z_2 \oplus z_1 \\
s_2^{-1} &= x_6 \oplus x_5 \oplus x_0 \\
s_1^{-1} &= x_7 \oplus x_5 \oplus x_4 \\
s_0^{-1} &= y_5 \oplus y_2 \oplus y_0
\end{aligned} \tag{10.56}
$$

For the computations using outputs from tables T_1^{-1}, T_2^{-1}, and T_3^{-1}, the input word must be rotated by one, two, and three byte positions, respectively, before applying the same transformation. This rotation is equivalent to defining variables x, y, and z as follows:

$$
\begin{aligned}
x &= byte(1, T_1^{-1}[a]) = byte(2, T_2^{-1}[a]) = byte(3, T_3^{-1}[a]) \\
y &= byte(2, T_1^{-1}[a]) = byte(3, T_2^{-1}[a]) = byte(0, T_3^{-1}[a]) \\
z &= byte(0, T_1^{-1}[a]) = byte(1, T_2^{-1}[a]) = byte(2, T_3^{-1}[a])
\end{aligned} \tag{10.57}
$$

and then applying transformation given by Equation 10.56. The entire *Decryption Round Circuit* is shown in Figure 10.47. The functional block T^{-1} *tables* consists of sixteen 8×32-bit look-up tables working in parallel. The operation of the *Decryption XOR Network* is given by Equation 10.53 and the operation of the *Inverse Multiplication Elimination* is given by Equation 10.55, 10.57, and 10.56.

In Figure 10.48, a circuit capable of performing both encryption and decryption using T-box approach is shown. The exact location of the register may depend on specific technology. For example, in Xilinx FPGAs, T tables are likely to be implemented using Block RAMs, which have synchronous outputs. Thus, the register would need to be placed at the output of the T *tables* and T^{-1} *tables* blocks, inside of the encryption and decryption rounds, shown in Figures 10.46 and 10.47, respectively.

Independently of the exact location of the register, the critical path is likely to be of the same length for the encryption and decryption circuits and includes two functional blocks: T/T^{-1} *tables* and *Encryption/Decryption XOR Network*.

Compared to the S-box-based implementation, the T-box-based implementation requires four times larger memory space, but fewer logic resources. In the T-box implementation the critical path is longer for encryption, but shorter for decryption,

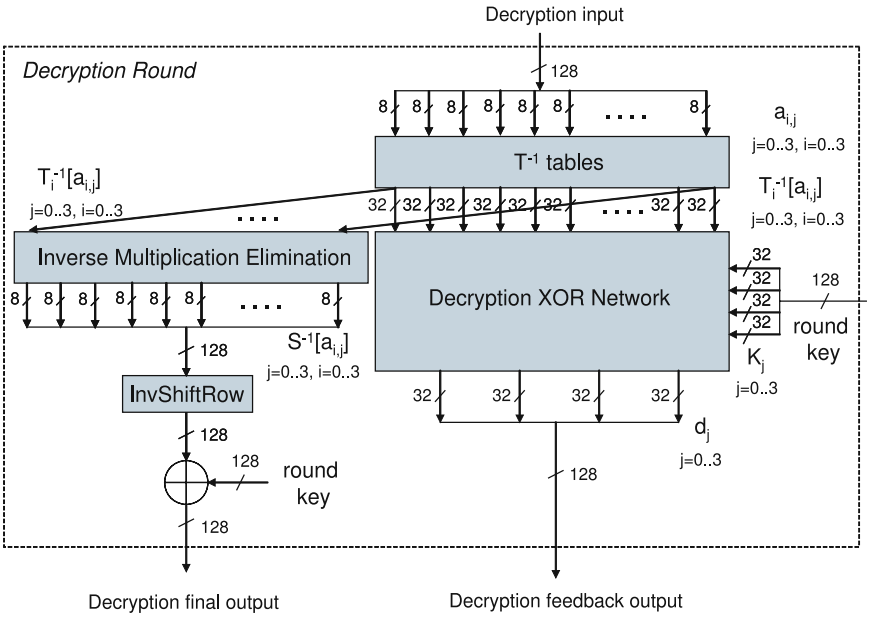

Fig. 10.47 Block diagram of the T-box-based AES decryption round.

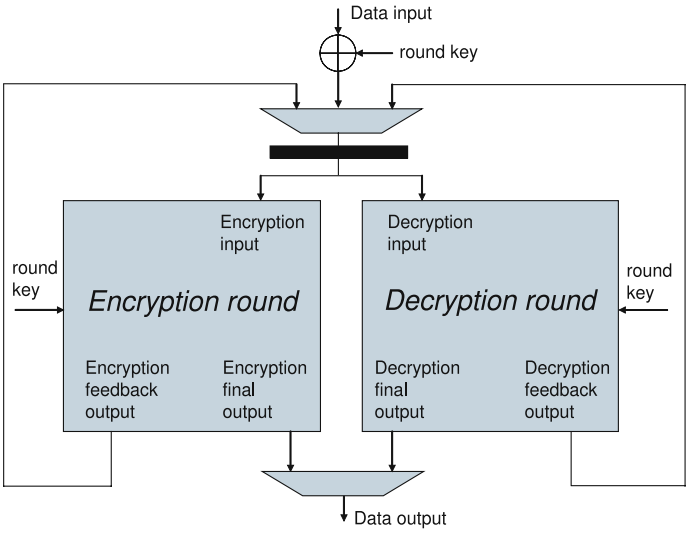

Fig. 10.48 Block diagram of the T-box-based AES encryption/decryption circuit.

compared to the S-box-based implementation. For example, in [13], the T-box-based approach was shown to produce a 8% speed-up for decryption, a 22% slowdown for encryption, and a 22% speed-up for encryption/decryption (a single-clock circuit capable of performing both operations) vs. equivalent implementations following the

S-box approach. All compared implementations targeted Altera FPGAs. The exact ratios of the costs and speeds for both approaches depend on the choice of a specific technology (FPGA vs. ASIC, specific FPGA family, specific ASIC standard-cell library, etc.) used for the implementation.

10.7.3 Compact Architectures

The AES round shown in Figure 10.49 reveals a great deal of parallelism. The data bytes are ordered from the most significant (byte 0) to the least significant (byte F) assuming big-endian representation. The round is composed of sixteen 8-bit S-boxes computing *SubBytes* and four 32-bit *MixColumns* operations, working independent of each other. The only operation that spans throughout the entire 128-bit block is *ShiftRows*.

It is possible to implement only four *SubBytes* and one *MixColumns* in order to compact the AES implementation. Ideally, the resources should be cut by four, while execution of one round should take four clock cycles. This approach would result in approximately four times lower performance than for the basic architecture.

Cutting the resources by 75% may not appear easy. The *folded round*, as we call the modified round, still must transform 128 bits, and storage for all 128 bits of the data block must exist. Another complication is related to the implementation of the *ShiftRows* operation. The data bytes processed in the AES round cannot return to the same positions in the block register because it would not execute the *ShiftRows* operation. On the other hand, those same bytes cannot be placed into locations indicated by *ShiftRows* because those locations are occupied by other bytes that have not yet been processed. Therefore, additional bits of intermediate results must be stored, and more logic resources are needed.

One of the possible architectures for a folded implementation is shown in Figure 10.50a. This architecture requires one 128-bit register, one 96-bit register, and one 32-bit-wide 4-to-1 multiplexer on top of the main cipher operations. The multiplexer becomes even bigger when both *ShiftRows* and *InvShiftRows* are

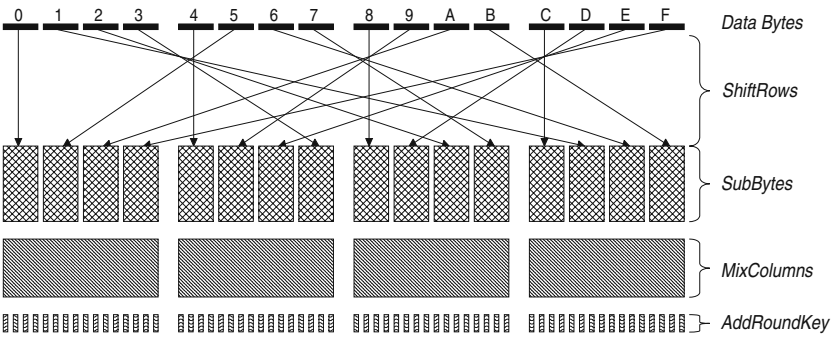

Fig. 10.49 Operations within AES encryption round.

implemented using same logic resources. The execution of one round takes four clock cycles. The authors believe that this, or very similar architecture, was implemented by A. Satoh et al. [33]. Their results show that the 4-cycle round takes 50% of the resources required by the 1-cycle round and yields four times lower throughput.

Another possible architecture is shown in Figure 10.50b. The 96-bit register is implemented as three 32-bit registers inserted into round operations creating a pipeline. In the case of FPGAs, those 32-bit registers will most likely be placed in the same Slices as logic operations yielding better resource utilization. The critical path is also shortened which permits the execution at a higher clock rate; however, the execution of the entire round requires seven, instead of four, clock cycles. The authors believe that this architecture was implemented by S. McMillan et al. [27], but no sufficient details are provided in this chapter. S. McMillan et al. reported only slight difference of 48 Slices (16%), and large difference of 24 Block RAMs (75%), between one-round unrolled and folded architecture.

10.7.3.1 Folded Register

The two folded architectures described above are very straightforward and resulted in small logic savings. In order to create a folded architecture with better parameters, we need to explore fine details of FPGA devices. Let us arrange data bytes into rows as shown in Figure 10.51. This data arrangement is consistent with a *state* introduced in [8]. The following exercise can now be executed in steps:

1. Read input bytes: 0, 5, A, F; execute *SubBytes*, *MixColumns*, and *AddRoundKey* on them; write results to the output at locations 0, 1, 2, 3. This step is highlighted in Figure 10.51.
2. Repeat above operations for input bytes 4, 9, E, 3; write results at output locations 4, 5, 6, 7.
3. Repeat above operations for bytes 8, D, 2, 7; write results at locations 8, 9, A, B.
4. Repeat above operations for bytes C, 1, 6, B; write results at locations C, D, E, F. Output now becomes input for the next step.

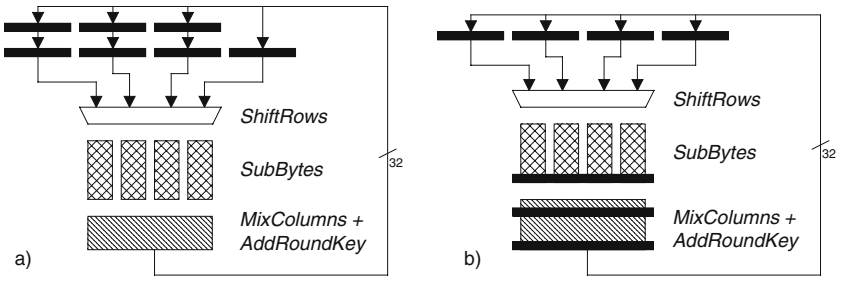

Fig. 10.50 Folded architectures (**a**) by A. Satoh et al. [33]; (**b**) by S. McMillan et al. [27].

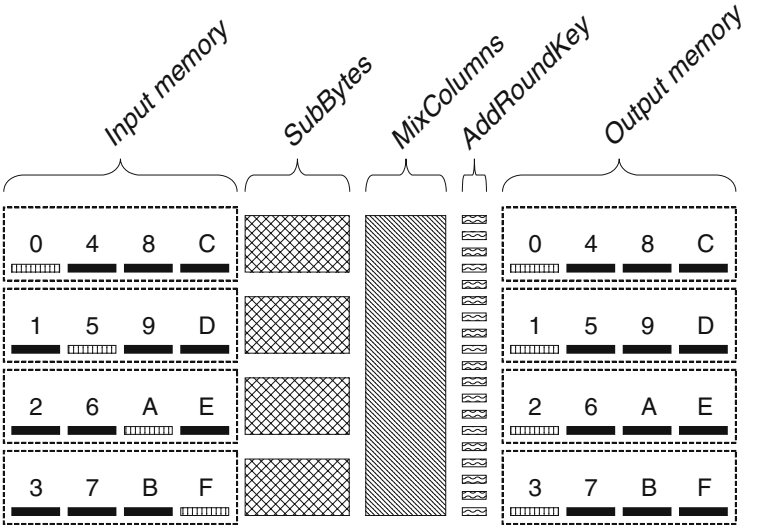

Fig. 10.51 Data arrangement in the folded architecture. Data bytes involved in the first step of calculation are highlighted.

In those four steps the entire AES round was executed including *ShiftRows* operation. At each step only one byte was read from each input row, and one byte was written to each output row. A similar exercise with identical conclusions can be executed for decryption transformation. Each row can be viewed as an addressable 8-bit-wide memory. The correct execution of *ShiftRows* and *InvShiftRows* is now resolved to the proper addressing of each of the memories at the consecutive clock cycles. At the fourth clock cycle output memories become input memories and vice versa.

10.7.3.2 FPGA Dual-Port RAM-Based Implementation

Each CLB Slice in Xilinx FPGAs contains two look-up tables (LUT), which are the primary resources for logic implementation. Typically LUTs are configured as small 16×1 ROM tables implementing logic functions of up to four inputs; however, other configurations are also possible. Two LUTs within the same Slice can implement a 16×1 dual-port RAM. An 8-bit-wide dual-port RAM can be implemented using eight CLB Slices. This memory can be divided into two banks, each addressed by a different port. One port is used for reading data from the memory, while the other one writes results back to the same memory. The switching between banks can be achieved by flipping one address bit in both ports every fourth clock cycle.

The dual-port RAM-based solution has major advantages over solutions presented in Figure 10.50:

- The logic resources required for storing intermediate results are far smaller.
- The multiplexer used before for *ShiftRows* and *InvShiftRows* is no longer needed.

- The complicated routing resulting from implementation of *ShiftRows* and *InvShiftRows* is avoided, yielding better performance.

10.7.3.3 FPGA Shift Register-Based Implementation

A better solution may result from the following observation: all bytes from the output of *AddRoundKey* are written into consecutive locations in the output memory in consecutive clock cycles. Therefore, we could use a simple shift register to shift computed data in without generating any addresses. Fortunately, LUTs can also be configured as 16-bit shift registers with variable taps, as shown in Figure 10.52. Four Slices can implement an 8-bit-wide, 16-bit-long shift register. The input of the shift register is used for shifting results in while the output, selected dynamically by changing tap address, is used for reading data out. This solution encompasses all of the advantages of the dual-port RAM-based solution and requires less than half of the logic resources than the dual-port RAM.

10.7.3.4 Encryption/Decryption Unit

The optimized circuit is capable of performing encryption and decryption. The AES encryption and decryption rounds substantially differ from the point of view of hardware implementations. One of the inconveniences arises from the fact that the *AddRoundKey* is executed after *MixColumns* in the case of encryption and before *InvMixColumns* in the case of decryption. Therefore, a switching logic is required to select appropriate data paths, which affect the performance, as shown in Figure 10.53.

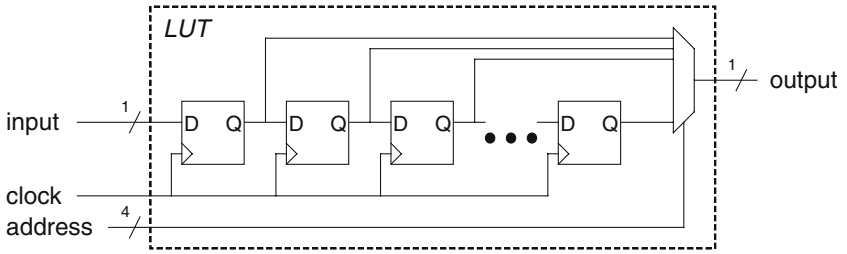

Fig. 10.52 Look-up table (LUT) configured as a shift register.

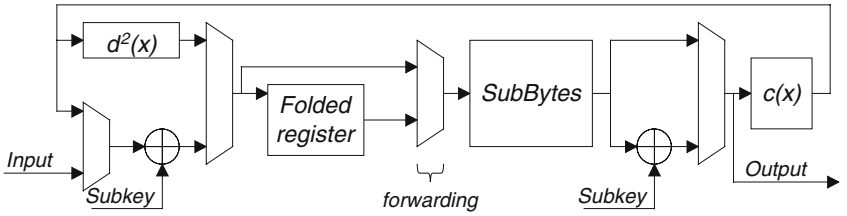

Fig. 10.53 Implementation of the encryption/decryption unit.

10.8 Implementation of Key Scheduling

An AES key scheduling unit can either generate round keys on the fly or it can store them in an internal key memory during the key setup phase and then read them from this memory whenever they are required by the encryption/decryption unit.

An AES key scheduling unit can support just one external key size, e.g., 128-bit key, or it can support all three key sizes described in the standard, i.e., 128-, 192-, and 256-bit keys. In the latter case, the unit is referred to as a 3-in-1 design.

Both kinds of units can be constructed in such a way that they produce 32 bits (one word = $\frac{1}{4}$th of a round key), 64 bits (two words = $\frac{1}{2}$ of a round key), or 128 bits (the entire round key) per clock cycle.

In the basic iterative architecture, only the last of these three designs allows the generation of round keys on the fly. The remaining designs require the key setup phase, during which the round keys are computed and stored in internal memory.

As an example, in Figure 10.54, we present a 3-in-1 key scheduling unit that produces 64-bits (a half of a round key) per clock cycle. The operation of the circuit is described by formulas given in Figure 10.54b that follow the pseudocode from Figure 10.16. The unit is capable of computing two 32-bit words of the key material (k_i and k_{i+1}) per one clock cycle, independently of the size of the main key.

Since each round key is 128-bit long (the size of the input block), two clock cycles are required to calculate each round key. Therefore, this key scheduling unit is not designed for computing subkeys on the fly. Instead, all round keys corresponding to the new main key are computed in advance and stored in the key memory. This computation can be performed in parallel with encrypting data using previous main key, therefore key scheduling does not impose any performance penalty.

The implementation of the key schedule suitable for a more compact encryption/decryption architecture supporting only 128-bit AES keys is shown in Figure 10.55. This architecture computes 32 bits of the key material per clock cycle, therefore, full key schedule execution for a 128-bit key takes 44 clock cycles. The computed round keys are stored in RAM.

The key schedule uses *SubBytes* operation that is identical to the one used in the encryption circuit. Since key schedule does not have to work simultaneously with the encryption unit, it is possible to time share S-boxes between both circuits.

10.9 Optimum Choice of a Hardware Architecture for AES

The choice of an optimum hardware architecture for AES depends on the following major factors:

1. Optimization criteria, such as minimum area, minimum power consumption, maximum throughput, maximum throughput to area ratio, etc.

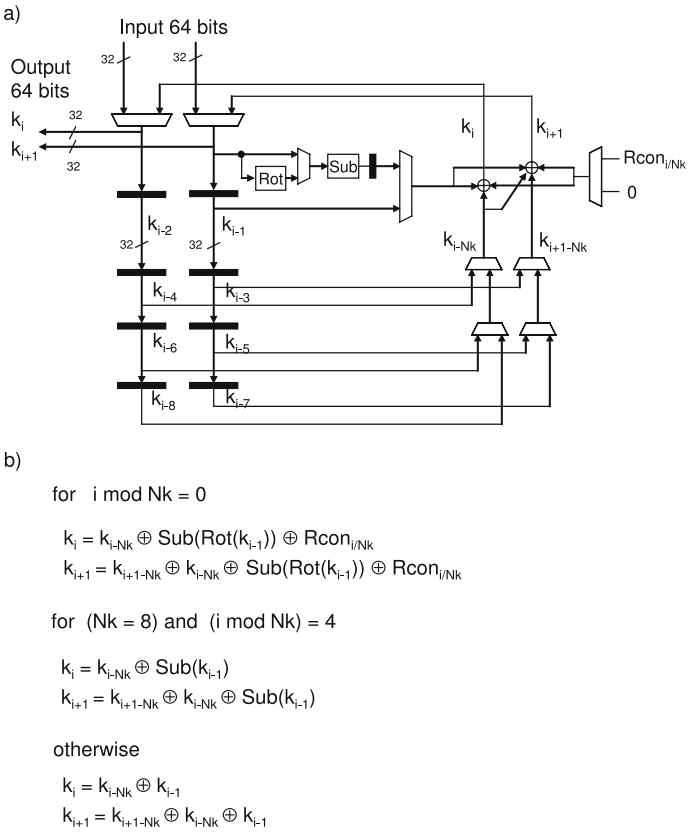

Fig. 10.54 The 3-in-1 key scheduling unit of AES: (**a**) main circuit, (**b**) formulas describing the operation of the circuit.

2. Support for feedback modes of operation, such as CBC and CFB, or non-feedback modes of operation, such as ECB and CTR modes.
3. Support for AES encryption only (e.g., in the block cipher modes of operation that require encryption only, such as CTR and CFB modes) or encryption and decryption (e.g., in the modes that require both operations, such as ECB and CBC).
4. Semiconductor technology of choice, such as ASIC or FPGA.
5. Resistance to side channel attacks, such as differential power analysis, timing analysis, etc.

In case area and/or power consumption are primary concerns, compact architectures described in Section 10.7.3 or the basic iterative architecture, described in Section 10.5.2, should be considered. This choice is independent of the requirements for feedback vs. non-feedback cipher modes and encryption only vs. encryption/decryption. S-box-based architectures will be preferred in this case, and

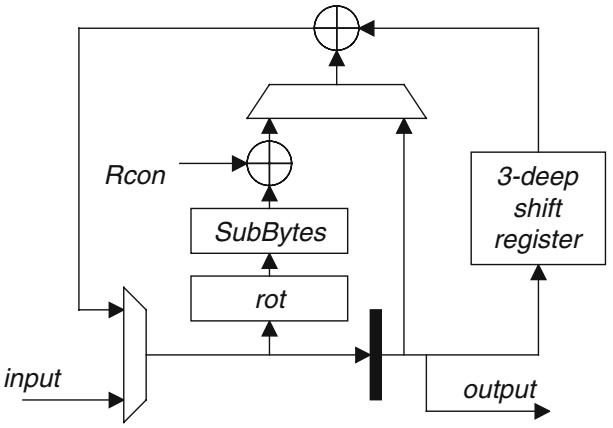

Fig. 10.55 Implementation of the key schedule.

S-boxes may be implemented using logic only (especially in ASIC implementations). In case both encryption and decryption are required, the resource sharing between *MixColumns* and *InvMixColumns*, based on the parallel or serial *InvMixColumns*, decomposition should be considered, as described in Section 10.6.2.

In case the maximum throughput is of primary concern, the choice of the hardware architecture depends on the operating modes that need to be supported.

As shown in Figure 10.20, the basic iterative architecture of AES assures the maximum throughput to area ratio for feedback operating modes such as CBC and CFB. It also guarantees near-optimum throughput and near-optimum area for these operating modes. Therefore, it is very likely to be commonly used in many practical implementations of AES targeting feedback cipher modes. In case only encryption needs to be supported (in the operating modes such as CFB), S-box-based architecture, described in Section 10.7.1, is preferred. In case both encryption and decryption need to be supported (e.g., in the CBC mode) the T-box-based architecture, described in Section 10.7.2, assures the maximum overall clock speed and data throughput.

In the non-feedback cipher modes of operation, such as counter mode, the architecture with the mixed inner- and outer-round pipelining, described in Section 10.5.4, offers the maximum circuit throughput. The S-box-based architecture with S-boxes implemented using logic only (see Sections 10.7.1 and 10.6.1) leads to the highest clock frequency. The throughput in the full mixed inner- and outer-round pipelining is given by

$$Throughput_{full\_mixed} = \frac{block\_size}{T_{CLK_{inner\_round}}(k_{opt})} \tag{10.58}$$

where $T_{CLK_{inner\_round}}(k_{opt})$ is the delay of a single pipeline stage for the optimum number of registers introduced inside of a single round. In FPGA implementations, this

delay is determined by the delay of a single CLB Slice and delays of interconnects between CLBs. As a result, the throughput does not depend on the complexity of a cipher round and tend to be similar for a large number of block ciphers, including AES. Therefore, the full mixed inner- and outer-round pipelining should be the architecture of choice for the implementations of AES targeting the highest possible throughput.

The choice of a hardware architecture depending on the resistance to side channel attacks is beyond the scope of this chapter. However, it should be noted that if the countermeasures against the side channel attacks are introduced at the circuit or logic levels, as proposed in multiple papers, such as [37–40], then all hardware architectures presented in this chapter might be equally secure.

10.10 Exercises

1. Using your knowledge about the internal structure of the *SubBytes* and *InvSubBytes* transformations, verify the correctness of the following entries of the AES S-box and AES Inverse S-box:

   a. S-box[89] = A7
   b. InverseS-box[89] = F2

2. Compute an output of the *MixColumns* transformation for the following sequence of input bytes "12 45 78 AB". Apply the *InvMixColumns* transformation to the obtained result to verify your calculations. Change the first byte of the input from "12" to "02", perform the *MixColumns* transformation again for the new input, and determine how many bits have changed in the output.
3. Compute the first two round keys of AES corresponding to the 128-bit key of all ones.
4. Draw a block diagram of the modified basic iterative architecture capable of encrypting messages in the counter mode using the minimum number of clock cycles. Compute the total time necessary to encrypt a message of the length of 1 MB using hardware implementation of Rijndael with a 128-bit input block and a 256-bit key, working in the counter mode with the size of a message block $k = 8$, assuming the modified basic iterative architecture operating with the clock frequency of 25 MHz.
5. Compute the total time necessary to encrypt a message of the length of 1 kB using hardware implementation of Rijndael with 128-bit input block and 192-bit key, working in the CFB mode with the size of a message block $k = 32$, assuming the basic iterative architecture operating with the clock frequency of 30 MHz.
6. Derive a formula for the contents of the look-up tables $T_0{}^{-1}$, $T_1{}^{-1}$, $T_2{}^{-1}$, $T_3{}^{-1}$ used for the decryption in the T-box-based implementation of AES. Compute the contents of the following components of these tables: $T_0{}^{-1}[1]$, $T_1{}^{-1}[2]$, $T_3{}^{-1}[254]$.

10.11 Projects

1. Develop and compare three different implementations of the *SubBytes InvSubBytes* operations using look-up tables, look-up tables and logic, and logic only.
   - compare the minimum clock period and use of logic resources (CLB Slices, Block RAMs, etc.) among the three designs
   - pipeline each design in order to obtain
     – minimum possible clock period and
     – maximum throughput to area ratio
     compare the three designs in terms of these parameters.
2. Develop and compare three different implementations of the *MixColumns InvMixColumns* operations using
   a. basic implementation
   b. compact implementation with parallel *InvMixColumns* decomposition
   c. compact implementation with serial *InvMixColumns* decomposition

   Compare the three designs in terms of the total resource usage, minimum latency for the *MixColumns* operation, and minimum latency for the *InvMixColumns* operation. Determine how suitable is each of the three designs for a hardware architecture with deep inner-round pipelining.
3. Develop and compare two different high-level implementations of AES in the basic iterative architecture
   a. based on S-boxes
   b. based on T-boxes

   Compare both implementations in terms of the maximum throughput, area, and throughput to area ratio.
4. Develop and compare three different implementations of the AES key scheduling unit with the number of output bits per clock cycle equal to
   a. 32 bits
   b. 64 bits
   c. 128 bits

   Compare all three implementations in terms of the minimum clock period and area.
5. Develop and compare three different implementations of the compact architecture of AES with the datapath width equal to
   a. 8 bits
   b. 32 bits
   c. 64 bits
   d. 128 bits (basic iterative architecture)

   Compare all four implementations in terms of the minimum clock period and area.

References

1. FIPS 197: Advanced Encryption Standard. National Institute of Standards and Technology, 2001, available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
2. Amphion. Documentation of cryptographic cores, available at http://www.amphion.com
3. D. Canright. A very compact Rijndael S-box. Technical Report NPS-MA-05-001, 2005.
4. D. Canright. A very compact S-box for AES. In J. R. Rao and B. Sunar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, LNCS, vol. 3659, pp. 441–455. Springer-Verlag, 2005.
5. P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. Master's thesis, George Mason University, Mar. 2002.
6. P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, LNCS vol. 2779, pp. 319–333. Springer-Verlag, 2003.
7. J. Daemen and V. Rijmen. AES proposal: Rijndael. Technical Report, 1999, available at http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf
8. J. Daemen and V. Rijmen. *The design of Rijndael: AES - The Advanced Encryption Standard*. Number ISBN 3-540-42580-2. Springer-Verlag, 2002.
9. A. Dandalis, V. K. Prasanna, and J. D. Rolim. A comparative study of performance of AES final candidates using FPGAs. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems Workshop (CHES'00)*, LNCS, vol. 1965 pp. 125–140. Springer-Verlag, 2000.
10. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 13–27. New York, USA, Apr. 13–14, 2000.
11. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI Systems*, 9(4):545–557, 2001.
12. V. Fischer. Realization of the round 2 candidates using Altera FPGA. *Comments for The Third Advanced Encryption Standard Candidate Conference (AES3)*, New York, USA Apr. 13–14, 2000.
13. V. Fischer and M. Drutarovský. Two methods of Rijndael implementation in reconfigurable hardware. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS vol. 2162, pp. 81–96. Springer-Verlag, 2001.
14. V. Fischer, M. Drutarovský, P. Chodowiec, and F. Gramain. InvMixColumn decomposition and multilevel resource sharing in Rijndael implementation. *IEEE Transactions on VLSI Systems*, 13(8):989–992, 2005.

15. V. Fischer and F. Gramain. Resource sharing in a Rijndael implementation based on a new MixColumn and InvMixColumn relation. unpublished.
16. K. Gaj and P. Chodowiec. Hardware performance of the AES finalists-survey and analysis results. Technical Report, George Mason University, 2000, available at http://ece.gmu.edu/crypto/AES_survey.pdf
17. K. Gaj and P. Chodowiec. Comparison of the hardware performance of the AES candidates using reconfigurable hardware. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 40–54. New York, USA, Apr. 13–14, 2000.
18. K. Gaj and P. Chodowiec. Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays. In *Proc. The Cryptographer's Track at the RSA Security Conference (CT-RSA'01)*, LNCS vol. 2020, pp. 84–99. Springer-Verlag, 2001.
19. T. Good and M. Benaissa. AES FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'05)*, LNCS, vol. 3659, pp. 427–440. Springer-Verlag, 2005.
20. Helion. Documentation of cryptographic cores. Available at http://www.heliontech.com
21. T. Ichikawa, T. Kasuya, and M. Matsui. Hardware evaluation of the AES finalists. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*, pp. 279–285. New York, USA, Apr. 13–14, 2000.
22. K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2003)*, pp. 207–215. Monterey, CA, Feb. 23–25, 2003.
23. H. Kuo and I. Verbauwhede. Architectural optimization for a 1.82Gbits/sec VLSI implementation of the AES Rijndael algorithm. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS, vol. 2162, pp. 51–64. Springer-Verlag, 2001.
24. I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Computer-Aided Design*, 62(2), Feb. 2007.
25. A. Lutz, J. Treichler, F. Gürkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fichtner. 2Gbit/s hardware realizations of RIJNDAEL and SERPENT: A comparative analysis. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, LNCS, vol. 2523, pp. 144–158. Springer-Verlag, 2002.
26. U. Mayer, C. Oelsner, and T. Köhler. Evaluation of different Rijndael implementations for high end servers. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, vol. 2, pp. 348–351. Scottsdale, Arizona, USA 2002.
27. S. McMillan and C. Patterson. JBits implementations of the Advanced Encryption Standard (Rijndael). In *Proc. Field-Programmable Logic and Applications (FPL'01)*, LNCS, vol. 2147, pp. 162–171. Springer-Verlag, 2001.

28. N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede. A systematic evaluation of compact hardware implementations for the Rijndael S-box. In J. R. Rao and B. Sunar, editors, *Proc. (CT-RSA'05)*, LNCS, vol. 3376, pp. 323–333. Springer-Verlag, 2005.
29. S. Morioka and A. Satoh. A 10 Gbps full-AES crypto design with a twisted-BDD S-Box architecture. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 98–103. Freiburg, Germany, 2002.
30. S. Morioka and A. Satoh. An optimized S-Box circuit architecture for low power AES design. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, LNCS, vol. 2523, pp. 172–186. Springer-Verlag, 2002.
31. A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. Rao, and P. Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In Ç. K. Koç and C. Paar, editors, *Proc. Cryptographic Hardware and Embedded Systems (CHES'01)*, LNCS, vol. 2162, pp. 171–184. Springer-Verlag, 2001.
32. G. Saggese, A. Mazzeo, N. Mazzocca, and A. Strollo. An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm. In *Proc. International Conference on Field-Programmable Logic and Applications (FPL'03)*, LNCS, vol. 2778, pp. 292–302. Springer-Verlag, 2003.
33. A. Satoh, S. Morioka, K. Takano, and S. Munetoh. A compact Rijndael hardware architecture with S-Box optimization. In *Proc. Theory and Application of Cryptology and Information Security (ASIACRYPT'01)*, LNCS, vol. 2248, pp. 239–254. Springer-Verlag, 2001.
34. P. R. Schaumont, H. Kuo, and I. M. Verbauwhede. Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. In *Proc. ACM Conference on Design Automation (DAC 2002)*, pages 634–639. New Orleans, Louisiana, USA, 2002.
35. N. Sklavos and O. Koufopavlou. Architectures and VLSI implementations of the AES-Proposal Rijndael. *IEEE Transactions on Computers*, 51(12): 1454–1459, 2002.
36. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In Ç. K. Koç and C. Paar, editors, *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, LNCS, vol. 2779, pp. 334–350. Springer-Verlag, 2003.
37. K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and differential routing - DPA resistance assessment. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, LNCS, vol. 3659, pp. 354–365. Springer-Verlag, 2005.
38. K. Tiri and I. Verbauwhede. Securing encryption algorithms against dpa at the logic level: next generation smart card technology. In Ç. K. Koç, C. Paar, and C. D. Walter, editors, *Cryptographic Hardware and Embedded*

Systems - CHES 2003, LNCS, vol. 2779, pp. 125–136, Cologne, Germany, 2003. Springer-Verlag.

39. K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proc. of Design Automation and Test in Europe (DATE 2004)*, pp. 246–251. Paris, France, 2004.
40. K. Tiri and I. Verbauwhede. A VLSI design flow for secure side-channel attack resistant ICs. In *Proc. of Design Automation and Test in Europe (DATE 2005)*, pp. 58–63, 2005.
41. I. Verbauwhede, P. Schaumont, and H. Kuo. Design and performance testing of a 2.29-GB/s Rijndael processor. *IEEE Journal of Solid-State Circuits*, 38(3):569–572, 2003.
42. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware performance simulations of Round 2 Advanced Encryption Standard algorithms. In *Proc. Third Advanced Encryption Standard Candidate Conference (AES3)*. New York, USA, Apr. 13–14, 2000.
43. J. Wolkerstorfer. An ASIC implementation of the AES MixColumn operation. In *Proc. Austrochip 2001*, pp. 129–132. Vienna, Austria, Oct. 12, 2001.
44. J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. In *Proc. The Cryptographer's Track at the RSA Security Conference (CT-RSA 2002)*, LNCS, vol. 2271, pp. 67–78. Springer-Verlag, 2002.
45. A. C. Zigiotto and R. d'Amore. A low-cost FPGA implementation of the Advanced Encryption Standard algorithm. In *Proc. Symposium on Integrated Circuits and Systems Design (SBCCI'02)*, pp. 191–196. Porto Alegre, Brazil, 2002.