9

Architectural Designs For the Advanced Encryption Standard

In this chapter we present some of the most common architectural alternatives to implement Advanced Encryption Standard (AES) in reconfigurable hardware. The first factor to be considered on implementing AES is the application. There are high speed applications like High Definition TV (HDTV) and video conferencing where high performance is required. The target throughput, expressed in gigabits per second (Gbps), must be specified, and to achieve such a high performance we can replicate several functional units to increase parallelism. That would however imply higher power and hardware area requirements.

On the other hand, high speed designs are not always desired solutions. In some applications, such as mobile computing and wireless communications, smaller throughput is demanded. Then a good balance between hardware area and design performance should be achieved. In addition, since there are trends to incorporate secure electronic data exchange into low-end consumer products, inexpensive AES implementations are needed for PDAs (personal digital assistant), wireless devices and many other embedded applications. Furthermore, it has been suggested that applications in the domain of radio frequency identifiers (RFID), low-power AES chip may be needed, thus demanding extremely compact AES implementations.

9.1 Introduction

Two main factors impact an AES implementation for a given application: hardware area and timing performance. Quite frequently, both factors have an opposite effect: Although compact designs tend to occupy a small amount of hardware resources, they generally show low performances. On the contrary, achieving high speed gains requires that many modules should work simultaneously, thus demanding greater hardware area requirements.

Another important feature to be considered when choosing an architectural alternative for AES is related to its mode of use. Many applications use AES in the Electronic Code Book (ECB) mode in which a complete block is ciphered independently of all other blocks. Then, several blocks can be processed in parallel or pipeline strategies can be applied to increase performance. Nevertheless, it is noticed that ciphering is only a part of a secure application and that there exist applications for which ciphering is accomplished with authentication [214]. For those scenarios, a feedback mode is required. For example, in Cipher Block Chaining (CBC), a previous ciphered block is used to encrypt the present block. That however, prevents us from using pipeline architectures. Therefore, an iterative architecture with some authentication logic could be a solution.

From its evaluation process to post selection period, the Advanced Encryption Standard (AES) has been implemented on all kind of hardware and software platforms. Gladman [109] and Bertoni et al. [21], reported software implementations in which AES specification is manipulated to increase performance. AES software implementations have a throughput that ranges from 300 to 800 Mbps depending on the specific architecture and platform selected by the developers. Some efficient AES encryptor/decryptor core VLSI implementations have been also reported in [143, 376, 215, 303]. Performance of VLSI implementations ranges from 2 to 7.5 Gbps.

Similarly, various AES FPGA implementations have been reported in [102, 63, 83, 223]. Those are one round (*iterative*) or n rounds (*pipeline*) FPGA implementations optimized for encryption or encryption/decryption processes. Since published works have utilized an ample variety of FPGA devices, reported performance results are broadly variable ranging from 300 Mbps to up to 25 Gbps.

Clearly, modern FPGA technology has a great impact in implementation performances. Nonetheless, there are algorithmic and architectural strategies for different target applications that also influence the final performance. The asymmetric characteristics of AES encryption and decryption processes limit the implementation of high-performance AES cores. Each step for AES encryption has its inverse counterpart for decryption. Designing separated architectures for encryption and decryption processes would imply the allocation of a large amount of FPGA resources and the area requirements of such design might be difficult or even impossible to meet in several FPGA families of devices.

Published work about AES FPGA implementation covers a wide spectrum. Some designs [102, 63, 83] have considered only the encryption part of AES. For example, in [102, 63] an iterative design implementing one round is reported. In [63] key scheduling is also considered, however, in [102] key scheduling was ignored. The design in [83] implements all AES rounds with a pipeline organization but without key scheduling, whereas the design in [223] reported an FPGA implementation of a fully pipeline AES encryptor/decryptor core.

In this Chapter, various FPGA architectures of AES are presented. Those implementations cover all three basic processes: key scheduling, encryption and decryption. All are single-chip FPGA implementation. Different design architectures are considered by implementing AES encryptor, decryptor and encryptor/decryptor cores separately. Both iterative and pipeline techniques are applied showing diverse time-area tradeoffs. All AES implementations were optimized for low cost, high efficiency and/or high portability.

The rest of this Chapter is organized as follows. An introduction to AES algorithm is presented in Section 9.2. The basic transformations of the algorithm and their effects on the algorithm cryptographic strength are also explained in this Section. Section 9.3 gives a brief explaination of the AES modes of use. Section 9.4 describes various algorithmic optimization for implementing AES basic transformations on FPGAs. Those techniques help to improve overall algorithm performance by modifying the most costly operations of the algorithm. Section 9.5 deals with general architectures for AES implementation on FPGAs. Then, the algorithmic optimizations are mixed with architectural alternatives to obtain several different AES designs. Section 9.6 presents performance results for each design and compare them with published works. Finally, in Section 9.6.1 some recent trends on AES cores are reviewed providing a classification of several relevant designs. Concluding remarks are drawn in Section 9.7.

9.2 The Rijndael Algorithm

On October 2000, Rijndael was selected as a new Advanced Encryption Standard (AES) by NIST [253] replacing Data Encryption Standard (DES). The name 'Rijndael' is a rearrangement of the names of its two inventors Rijmen and Daemen [60].

Rijndael is a symmetric block cipher which takes two inputs, namely, the plaintext block to be encrypted and the secret key. It applies an iterative procedure at the end of which an output ciphertext block is produced. During a single iteration, a set of transformations, called a *round*, are applied to the state data block. For each round, a *round key* is generated through a process called key scheduling.

In this Section we give a short explanation of the algorithm behavior. We start explaining the difference between AES and Rijndael. Then, we describe AES basic structure and building blocks. Thereafter, the round transformation of the algorithm is specified. Finally, the process of key generation is described.

9.2.1 Difference Between AES and Rijndael

AES fixes the block sizes and key lengths from the range supported by Rijndael. Rijndael can process variable block and key lengths of 128, 192, and 256 bits. Moreover, Rijndael supports all possible combinations of those sizes for

block and key lengths. The number of rounds depends upon the combination of the selected block and key lengths as shown in Table 9.1. It can be seen that the number of rounds ranges from 10 to 14.

key length *(bits)*	Block length *(bits)*		
	128	192	256
128	10	12	14
192	12	12	14
256	14	14	14

Table 9.1. Selection of Rijndael Rounds

On the other hand, AES fixes the block length to 128 bits and supports key lengths of 128, 192 or 256 bits only. The most frequent AES case of use is with block and key lengths of 128 bits. In the rest of this chapter whenever we use the word AES, it means block and key lengths of 128 bits and therefore with the number of rounds equal to 10. Moreover, In the rest of this Chapter the names AES and Rijndael are used indistinctly.

9.2.2 Structure of the AES Algorithm

The basic structure of AES algorithm is shown in Figure 9.1.

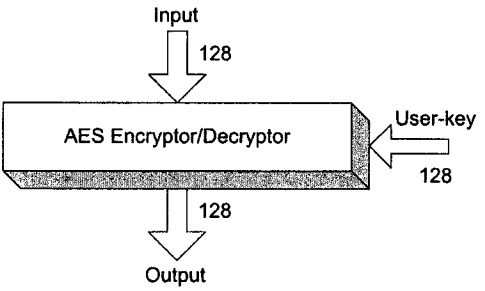

Fig. 9.1. Basic Structure of Rijndael Algorithm

For encryption, the input is a plaintext block and a key, and the output is a ciphertext block. For decryption, the input is a ciphertext block and a key (the same key used for encryption), and the output is the original plaintext. The basic algorithm flow for encrypting a single block of data is shown in Figure 9.2.

The AES cipher treats the input 128 bit block as a group of 16 bytes organized in a 4×4 matrix called *State* matrix. The algorithm consists of an initial

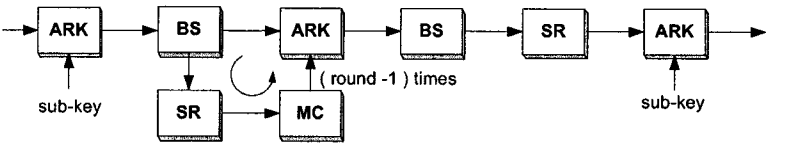

Fig. 9.2. Basic Algorithm Flow

transformation, followed by a main loop where nine iterations, called *rounds*, are executed. Each *round transformation* is composed of a sequence of four transformations: ByteSubstitution (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ARK). For each round of the main loop, a round key is derived from the original key through a process called *Key Scheduling*. At the last round MC step is skipped and consequently just three transformations, namely, BS, SR and ARK, are executed.

AES decryption can be performed by using same algorithm flow. However all four steps in the round transformation are replaced with their own inverses and the round keys for encryptions are used in the reverse order.

9.2.3 The Round Transformation

The round transformation is a sequence of four transformations BS, SR, MC and ARK. All four transformations contribute in AES strength by inducing *confusion* and *diffusion*, which are arguably the two most important properties that a strong symmetric cipher must have. Confusion makes the output dependent on the key. Ideally, every key bit influences every output bit. Diffusion makes the output dependent on previous input (plain/ciphertext). Ideally, each output bit is influenced by every (previous) input bit. Roughly speaking, those characteristics correspond to cipher's substitution and permutation.

Symmetric ciphers need to be complex, so they could not be analyzed easily. Also, their transformations need to be simple enough to be implemented efficiently in hardware or software. For AES, the general criteria for round transformation was inverse function and simplicity besides the step-specific criteria.

9.2.4 ByteSubstitution (BS)

It is a non-linear transformation where each input byte of the State matrix is independently replaced by another byte. BS can be seen as a highly non-linear function. There are a great finite number of possible BS functions, however some of them are more appropriate than others. In [60] some important properties about designing a BS function are discussed. *Non-linearity* and *algebraic complexity* being the most important of them.

The BS transformation of an input byte (8-bit vector) a is defined by two substeps:

1. **Inverse:** Let $x = a^{-1}$, the multiplicative inverse in GF(2^8) (except if $a = 0$ then $x = 0$).
2. **Affine Transformation:** Then the output is $y = M \times x \oplus b$, with the constant bit matrix M and byte b shown below:

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{9.1}$$

All bit operations are performed modulo 2.

BS is decomposed into two transformations. First each input byte is replaced with its multiplicative inverse (MI) in GF(2^8) with the element {00} being mapped to itself and then the affine transformation is applied as shown in Equation 9.1.

From the implementation point of view, BS can be considered as a look-up table, called *S-Box*, in which the input byte is considered as the address of the table where its substitution is found. Then an S-Box can be seen as a 256×8 look up table as shown in Figure 9.3. This is the easiest way to implement BS and for many applications it is enough to consider this way of implementing it[1].

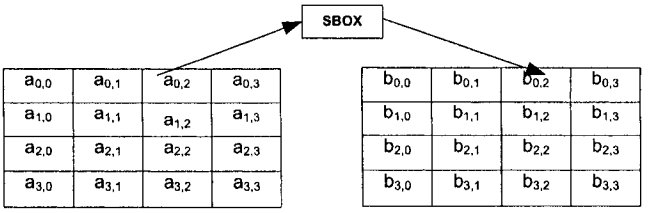

Fig. 9.3. BS Operates at Each Individual Byte of the State Matrix

If we look for a very compact or a high efficient design, we need to look for the calculation of BS. Multiplicative inverse can be found using the extended Euclidean algorithm [228][2]. Let x be the input byte and let us assume that we

[1] It has been proposed that also the multiplications associated to the MixColumn transformation can be implemented using the Look-up Table methodology [81].

[2] Formal definition of field multiplicative inverse and the extended Euclidean algorithm can be found in §4.1.2. Efficient computations of the multiplicative inverse were discussed in §6.3.

look for the inverse of the polynomial $a(x)$. The extended Euclidean algorithm can be used to find two polynomials $b(x)$ and $c(x)$ such that:

$$a(x) \times b(x) + m(x) \times c(x) = \gcd(a(x), m(x)) \tag{9.2}$$

where $\gcd(a(x), m(x))$ represents the *greatest common divisor* of the polynomials $a(x)$ and $m(x)$. If $m(x)$ is irreducible then we know for sure that $\gcd(a(x), m(x)) = 1$. Applying modular reduction to Equation 9.2 we get,

$$a(x) \times b(x) = 1 \bmod m(x) \tag{9.3}$$

which means that $b(x)$ is the inverse element of $a(x)$. The non-linearity of the AES S-box is introduced by applying the multiplicative inverse in $GF(2^8)$. The affine transformation has no impact on the non-linearity but it contributes in increasing the algebraic complexity.

Inverse Operation (IBS)

The inverse BS is obtained by applying inverse affine transformations followed by the multiplicative inverse in $GF(2^8)$. Therefore, the inverse of the affine transformation in Eqn. 9.1 is defined as follows.

$$x = M^{-1} \times y \oplus d$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{9.4}$$

For both affine and inverse affine transformations, multiplicative inverse is taken in $GF(2^8)$ with irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

9.2.5 ShiftRows (SR)

It is a cyclic shift operation where each row is rotated cyclically to the left using 0,1,2 and 3-byte offset for encryption as shown in Figure 9.4. *Diffusion optimality* is the design criteria for selecting the offsets which requires the four offsets to be different.

Inverse Operation (ISR)

The inverse operation of ShiftRows is called Inverse ShiftRows (ISR). It is a cyclic shift operation used for decryption where each row is rotated cyclically to the right using 0,1,2 and 3-byte offset.

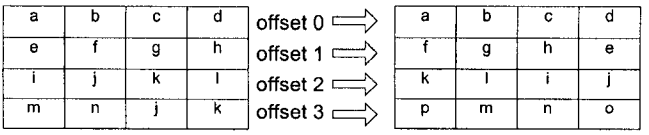

Fig. 9.4. ShiftRows Operates at Rows of the State Matrix

9.2.6 MixColumns (MC)

In this transformation, each column of the State matrix is considered a polynomial over GF(2^8) and is multiplied by a fixed polynomial $c(x)$ modulo $x^4 + 1$. The polynomial $c(x)$ is given by:

$$c(x) = 03.x^3 + 01.x^2 + 01.x + 02 \tag{9.5}$$

Let $b(x) = c(x) \cdot a(x) \bmod x^4 + 1$, then the modular multiplication with a fixed polynomial can be written as shown in Equation 9.6.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{9.6}$$

MixColumns operates on the columns of the state matrix as shown in Figure 9.5.

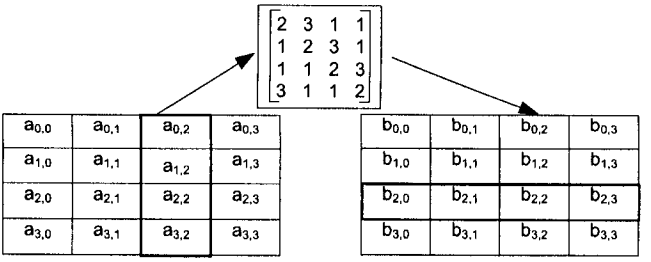

Fig. 9.5. MixColumns Operates at Columns of the State Matrix

The design criteria for MixColumns step includes *dimensions*, *linearity*, *diffusion* and *performance* on 8-bit processor platforme. The *Dimension* criterion it is achieved in the transformation operation on 4-byte columns.

Inverse Operation IMC

The inverse of MixColumns is called (IMC). The constant polynomial $c(x)$ given in Eqn. 9.5 is co-prime to $x^4 + 1$ and therefore invertible. Let $d(x)$ be the inverse of $c(x)$ and written as follows.

$$(03.x^3 + 01.x^2 + 01.x + 02).d(x) \equiv 01 \ (mod\ x^4 + 1) \tag{9.7}$$

From Eqn. 9.7, it can be seen that $d(x)$ is given by:

$$d(x) = 0B.x^3 + 0D.x^2 + 09.x + 0E \tag{9.8}$$

Similarly to MC, in IMC each column of the state matrix is transformed by multiplying with constant polynomial $d(x)$ written as a matrix multiplication as shown in Equation 9.9.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{9.9}$$

9.2.7 AddRoundKey (ARK)

In the last step, the output of MC is XOR-ed with the corresponding round key. This step is denoted as ARK. Figure 9.6 illustrates the effect of key addition on the state matrix.

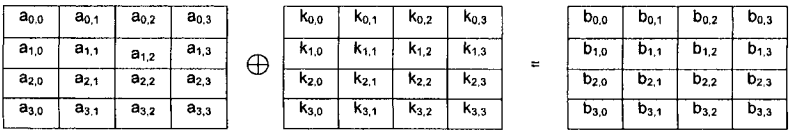

Fig. 9.6. ARK Operates at Bits of the State Matrix

Inverse Operation IARK

Inverse of ARK, called IARK, is essentially the same for encryption and decryption[3]. The only important thing to remember is that keys are applied for decryption in reverse order as in encryption.

[3] However, as is explained in §9.5.2, efficient implementations of AES encryptor/decryptor cores, require to append the IMC step to the generation of round keys for decryption.

9.2.8 Key Schedule

Both, encryption and decryption require the generation of round keys. Round keys are obtained through the expansion of secret user key by attaching each $j-th$ round a 4-byte word $k_j = (k_{0,j}, k_{1,j}, k_{2,j}, k_{3,j})$ to the user key. The original user key, consisting of 128 bits, is arranged as a 4×4 matrix of bytes.

Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key. Then, these four columns are recursively expanded to obtain 40 more columns. Let us assume we have computed columns up to $w[i-1]$. Then, we can compute the $i-th$ column, $W[i]$, as follows,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \tag{9.10}$$

where $T(w[i-1])$ is a non-linear transformation of $w[i-1]$ calculated as follows:

Let w, x, y, and z be the elements of column $w[i-1]$ then,

1. Shift cyclically the elements to obtain z, w, x, and y.
2. Replace each of the byte with the byte from BS $S(z)$, $S(w)$, $S(x)$ and $S(y)$.
3. Compute the round constant $r(i) = 02^{(i-4)/4}$ in GF(2^8).

Then, T($w[i-1]$) is the column vector, $(S(z) \oplus r(i), S(w), S(x), S(y))$. In this way, columns from w[4] to w[43] are generated from the first four columns.

The 16-byte round key for the $j-th$ round consists of the columns

$$(w[4j], w[4j+1], w[4j+2], w[4j+3])$$

Sometimes it results convenient to pre-compute the round keys once and for all and then store them. A similar process is utilized for generating round keys for the decryption process, although they should be used in the reverse order.

After the explanation of all four AES transformations and key schedule, we can write the sequence of those transformations when performing encryption and decryption as follows,

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: IARK→ IMC→ ISR→ IAF→ MI

9.3 AES in Different Modes

Most of the published work on AES implementation considers AES in Electronic Book Mode (ECB). In ECB mode, an individual plaintext block is converted to ciphertext block. Thus by collecting several plaintext and their ciphertext blocks, one can produce some pattern information which could

be helpful in recovering the original plaintext. ECB mode in some cases, is therefore not considered secure. The Cipher Block Chaining mode (CBC), the Cipher Feedback mode (CFB), and the Output Feedback mode (OFB) offer better security than ECB, but encryption of the block depends on the feedback of its previous block encipherment [253]. This property prevents using pipelining in which many different blocks are encrypted simultaneously. The encryption speed in CBC, CFB, and OFB modes is much slower as in ECB. Fortunately, there exists another mode, called Counter mode (CTR) which increases the security of ECB and has not dependencies among different blocks, thus allowing all operations to be fully pipelined to achieve high performance.

9.3.1 CTR Mode

In [100] a CTR mode implementation of AES is reported. In CTR mode, a plaintext is processed by encrypting a counter value with key 'K' and then by XORing the output with the plaintext to get the ciphertext. Figure 9.7 presents the counter mode. Decryption procedure takes the same process to recover the plaintext from the ciphertext. The counter value has no dependencies with previous output, thus pipelining can be fully used. Counter mode has no padding overhead which is required for ECB, CBC, and CFB modes when the data is not a multiple of block length. Counter mode does not propagates error and restrict the error to the specific block as compared to CBC and CFB modes which pass the error to the subsequent blocks.

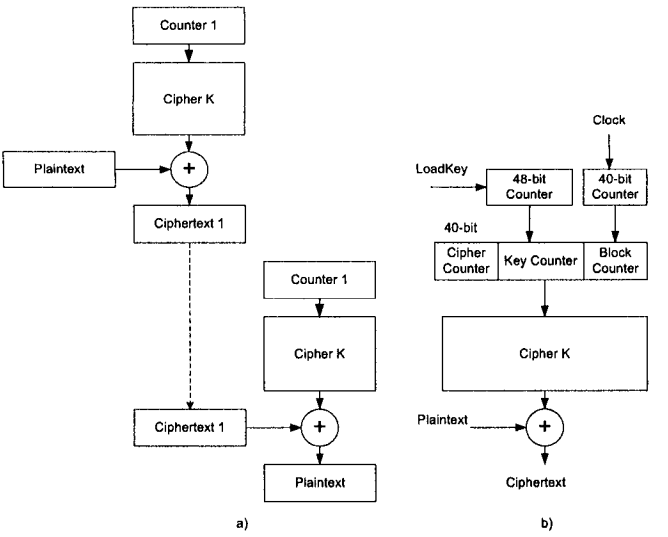

Fig. 9.7. Counter Mode Operations

Figure 9.7b, presents different counter blocks for obtaining cipher key 'K'. A three stage counter, 40-bit cipher identification, 48-bit key counter and 40-bit block counter, are used for each plaintext block. For each cipher artifact, there is a pre-assigned cipher ID. The key counter increases whenever a new key has been updated. Block counter increases for each block. The search space for each part is, although finite, large enough. If the block counter is exhausted, the key counter will be increased to avoid the use of the same key with the same counter value. Then, we guarantee that produced keys are all distinct. The counter value pairs can be used more than once.

The special requirement for CTR mode is that the same counter value and key should not be used to encrypt more than one block of data. If this happens, the plaintext would be recovered by XORing the two ciphertext, which in fact, equals to XORing the two plaintext. Especially when one of the plaintext is already known, the other one can be easily recovered by XORing the known plaintext with the output ciphertext after XOR.

9.3.2 CCM Mode

For applications in which more robustness is required, there is no choice and a feedback mode is mandatory. For example, the Wired Equivalent Privacy (WEP) protocol has been the most widely security tool used for protecting information in wireless environments. However, this protocol was broken in 2001 by Fluhrer et al. [1]. Based on that attack, nowadays there exist a variety of programs that can be downloaded from Internet to break the WEP Protocol in few seconds and with almost no effort. This situation has led to a search for new security mechanisms for guaranteeing reliable ways of protecting information in wireless mobile environments.

AES in CCM (Counter with CBC-MAC) proposed by Whiting et. al. in [378], has become one of the most promising solutions for achieving security in wireless networks. This mode simultaneously offers two key security services, namely, data Authentication and Encryption [214]. CCM means that two different modes are combined into one, namely, the CTR mode and the CBC-MAC. CCM is a generic authenticate-and-encrypt block cipher scheme that has been specifically designed for being use in combination with a 128-bit block cipher, such as AES. Currently, CCM mode has become part of the new 802.11i IEEE standard.

CCM Primitives

Before sending a message, a sender must provide the following information [378]:

1. A suitable encryption key K for the block cipher to be used.
2. A nonce N of $15 - L$ bytes. Nonce value must be unique, meaning that the set of nonce values used with any given key shall not contain duplicate values.

3. The message m, consisting of a string of $l(m)$ bytes where $0 \leq l(m) < 2^{8L}$.
4. Additional authenticated data a, consisting of a string of $l(a)$ bytes where $0 \leq l(a) < 2^{64}$. This additional data is authenticated but not encrypted, and is not included in the output of this mode.

Figure 9.8 shows CCM authentication and verification processes dataflow. Notice that because of the CBC feedback nature of the CCM mode a pipeline approach for implementing AES is not possible, therefore there is no option but to implement AES encryption core in an iterative fashion.

CCM Authentication consists on defining a sequence of blocks $B_0, B_1, \cdots, B_n$ and thereafter CBC-MAC is applied to those blocks so that the authentication field T can be obtained. Blocks B_is are defined as explained below.

First, the authentication data a is formatted by concatenating the string that encodes $l(a)$ with a itself, followed by organizing the resulting string in chunks of 16-byte blocks. The blocks constructed in this way are appended to the first configuration block B_0 [375]. Then, message blocks are added right after the (optional) authentication blocks a. Message blocks are formatted by splitting the message m into 16-byte blocks which will be the main part of the sequence of blocks

$$B_0, B_1, ..., B_n$$

needed by the authentication mode. Finally, the CBC-MAC is computed as,

$$\begin{aligned} X_1 &:= AES_E(K, B_0) \\ X_{i+1} &:= AES_E(K, X_i \oplus B_i) \text{ for } i = 1, ..., n \\ T &:= firstMbytes(X_{n+1}) \end{aligned} \tag{9.11}$$

Where AES_E is the AES block cipher selected for encryption, and T is the MAC value defined as above. If it is needed, the ciphertext would be truncated in order to obtain T.

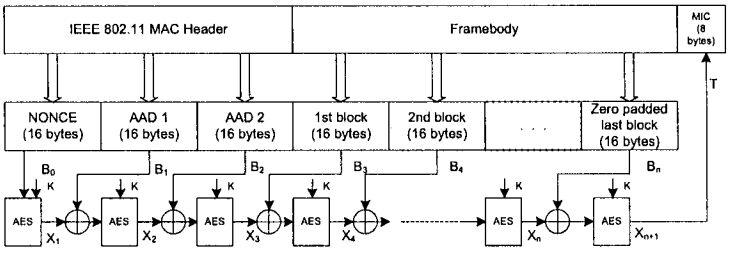

Fig. 9.8. Authentication and Verification Process for the CCM Mode

Figure 9.9 shows the CCM encryption/decryption process dataflow. CCM encryption is achieved by means of Counter (CTR) mode as,

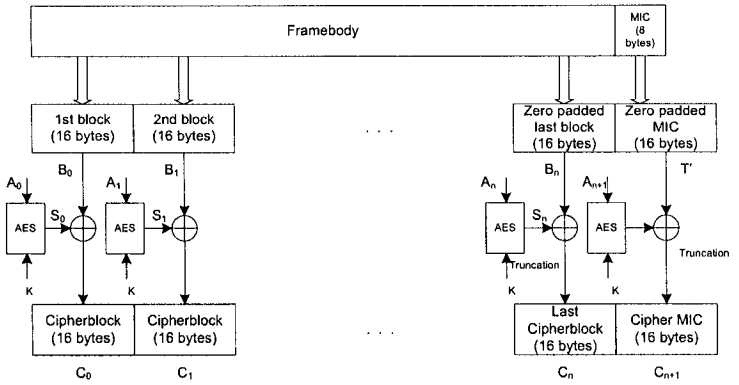

Fig. 9.9. Encryption and Decryption Processes for the CCM Mode

$$S_i := AES_E(K, A_i) \text{ for } i = 0, 1, 2, ..., \tag{9.12}$$
$$C_i := S_i \oplus B_i$$

where A_i stands for counters. See [378, 100] for more technical details about how to build the counters.

Plaintext m is encrypted by XORing each of its bytes with the first $l(m)$ bytes of the sequence resulting from concatenating the cipher blocks $S_1, S_2, S_3, ...$, produced by Eq. 9.12. The authentication value is computed by encrypting T with the key stream block S_0 truncated to the desired length as,

$$U := T \oplus firstMbytes(S_0) \tag{9.13}$$

The final result c consists of the encrypted message m, followed by the encrypted authentication value U.

At the receiver side, the decryption process starts by recomputing the key stream to recover the message m and the MAC value T. Figure 9.9 shows how the decryption process is accomplished in CCM Mode.

Message and additional authentication data is then used to recompute the CBC-MAC value and check T. If the T value is not correct, the receiver should not reveal the decrypted message, the value T, or any other information. Figure 9.8 describes how the verification process is accomplished.

It is important to notice that the AES encryption process is used in encryption as well as in decryption. Therefore, AES decryption functionality is not necessary in CCM-mode, which leads to save valuable hardware resources.

9.4 Implementing AES Round Basic Transformations on FPGAs

Strategies for efficient hardware implementation of AES on FPGA devices can be classified into two types: algorithmic and architectural optimizations. Algorithmic optimizations try to obtain some mathematical expressions to take advantage of FPGA structure. Architectural optimizations exploit design techniques such as iterative, pipelining and sub-pipelining. In addition, AES hardware implementation poses a challenge since encryption and decryption processes are not completely symmetrical which forces to have some additional observations while implementing a single encryptor/decryptor core.

In Subsection 9.2.3 it was described the basic round transformations, BS, SR, MC, and ARK, and their corresponding inverse transformations IBS, ISR, IMC, and IARK. That Subsection also describes the key schedule process to generate the necessary subkeys during an encryption or decryption process.

But before start discussing how to implement a full encryption or decryption core, let us analyze, from the algorithmic optimization point of view, some important implementation properties shown by the basic round transformations.

The most important operations for the basic transformations include polynomial multiplication in $GF(2^8)$ for BS/IBS, fixed-rotation for SR/ISR, constant polynomial multiplication in $GF(2^8)$ for MC/IMC, and simple addition (XOR) for ARK/IARK. Fixed-rotation is hardwired and does not consume FPGA's logic resources. The addition used in ARK/IARK is a simple XOR operation. Hence, BS/IBS and MC/IMC are the two key functional units in AES implementations. It has been estimated that BS/IBS and MC/IMC take more than 65% of the total area in the entire AES encryptor/decryptor implementation.

Perhaps, the most costly operation for BS/IBS is polynomial multiplication in $GF(2^8)$. We also need to perform a polynomial multiplication in $GF(2^8)$ for MC/IMC but we can take advantage from the fact that is a constant multiplication. Even though the latter transformation is relatively less costly than the former still it occupies considerable FPGA's resources. Therefore, both BS/IBS and MC/IMC are good candidates for improving overall performance of the round transformation.

In the rest of this Section, we present various approaches for implementing BS/IBS and MC/IMC.

Regarding BS/IBS two alternatives are considered. In the first approach pre-computed values are simply stored on the FPGA's built-in memory modules. This might be seen as an expensive solution but it helps to save valuable computational time. The second approach provides an alternative for constrained memory requirements and it is based on an on-fly computation strategy.

Similarly, two approaches for MC/IMC implementations are presented. First approach, that we have called *standard approach*, deals with the struc-

tural organization of MC/IMC transformations. The second approach called *modified approach* introduces a small modification before MC to perform IMC step. Finally, some structural changes are proposed in key schedule algorithm which can improve hardware performance by cutting path delays.

9.4.1 S-Box/Inverse S-Box Implementations on FPGAs

The straightforward approach for implementing BS is by using a look-up table in which pre-computed values are stored in memories. That requires memory modules with fast access. In FPGAs, there are two ways to organize memory: by using flip-flops and CLBs (i.e., FPGA fabrics), or by using FPGAs built-in memory modules called BRAMs (BlockRAMs).

Implementing BS/IBS by look-up tables is simple, fast and in many cases desirable. A single BS/IBS table would require 8-bit wide 256 entries. We can make some few observations about implementing BS/IBS using look-up tables.

Firstly, for the implementation of both encryption and decryption on a single chip two different separated look-up tables are required, thus duplicating memory requirements.

Secondly, if we want to increase performance, BS/IBS can be performed in parallel for the sixteen bytes of the state matrix. The fully parallelization of BS/IBS would therefore require 16 copies of the same look-up table, one per state matrix element. Finally, if high performance is required, unfolding the 10 rounds of AES to construct a pipeline architecture, would require 160 copies of the same look-up table.

In the following, we discuss some other alternatives to implement BS/IBS in FPGAs.

I. S-Box and Inverse S-Box Implementation

To avoid utilization of a considerable amount of FPGA resources, BS/IBS can be implemented using a look-up table. The look up table would be used for MI by implementation affine (AF) and inverse affine (IAF) transformations using some logic gates for BS and IBS respectively. The combination MI + AF implements BS for encryption and the combination IAF + MI gives IBS for decryption. For constructing an encryptor/decryptor core, two separated designs for encryption and decryption would result in high area requirements. From Section 9.2.4, we know that only one MI transformation in addition to AF and IAF transformations is required for both encryption and decryption. Therefore, a multiplexer can be used to switch the data path for either encryption or decryption as shown in Figure 9.10

II. S-Box and Inverse S-Box Based on Composite Field Techniques

BS/IBS implementations can be made using composite field techniques e.g. BS can be manipulated in $GF((2^4)^2)$ and even $GF(((2^2)^2)^2)$ instead of $GF(2^8)$.

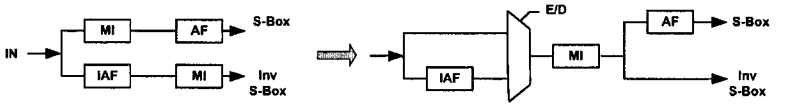

Fig. 9.10. S-Box and Inv. S-Box Using Same Look-Up Table

That would reduce memory requirements to 16×4 bits in $\mathrm{GF}(2^4)$ as compared to 256×8 bits in $\mathrm{GF}(2^8)$ for a single LUT. More hardware resources would be however used to implement the required logic in $\mathrm{GF}(2^4)$. Several authors [267, 242, 303] have designed AES S-Box based on the composite field techniques reported first in [267]. Those techniques use a three-stage strategy:

1. Map the element $A \in \mathrm{GF}(2^8)$ to a smaller composite field F by using an isomorphism function δ.
2. Compute the multiplicative inverse over the field F.
3. Finally, map the computations back to the original field.

In [242], an efficient method to compute the inverse multiplicative based on Fermat's little theorem was outlined. That method is useful because it allows us to compute the multiplicative inverse over a composite filed $\mathrm{GF}(2^m)^n$ as a combination of operations over the ground field $\mathrm{GF}(2^m)$. It is based on the following theorem:

Theorem 1 *[267, 121] The multiplicative inverse of an element A of the composite field $GF(2^m)^n$, $A \neq 0$, can be computed by,*

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma - 1} \bmod P(x) \tag{9.14}$$

$$\texttt{where} \qquad A^{\gamma} \in \mathrm{GF}(2^n) \;\&\; \gamma = \frac{2^{nm} - 1}{2^m - 1}$$

An important observation of the above theorem is that the element A^{γ} belongs to the ground field $\mathrm{GF}(2^m)$. This remarkable characteristic can be exploited to obtain an efficient implementation of the inverse multiplicative over the composite field. By selecting $m = 4$ and $n = 2$ in the above theorem, we obtain $\gamma = 17$ and,

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma - 1} = (A^{17})^{-1} A^{16} \tag{9.15}$$

In case of AES, it is possible to construct a suitable composite field F, by using two degree-two extensions based on the following irreducible polynomials.

$$\begin{array}{ll} F_1 = \mathrm{GF}(2^2) & P_0(x) = x^2 + x + 1 \\ F_2 = \mathrm{GF}((2^2)^2 & P_1(y) = y^2 + y + \phi \\ F_3 = \mathrm{GF}(((2^2)^2)^2 & P_2(z) = z^2 + z + \lambda \end{array} \tag{9.16}$$

$$\text{where } \phi = \{10\}_2,\ \lambda = \{1100\}_2$$

The inverse multiplicative over the composite field F_2 defined in the Equation 9.15, can be found as follows.

Let $A \in F_2 = \text{GF}(2^2)^2$ be defined in polynomial basis as $A = A_H y + A_L$, and let the Galois Fields F_1, F_2, and F_3 be defined as shown in Equation 9.16, then it can be shown that,

$$\begin{aligned} A^{16} &= A_H y + (A_H + A_L) \\ A^{17} &= A^{16} \cdot A = 0.y + (\lambda(A_H)^{16} A_H + (A_L)^{16} A_L) \\ &= \lambda(A_H)^2 + (A_L)^{16} A_L \end{aligned} \tag{9.17}$$

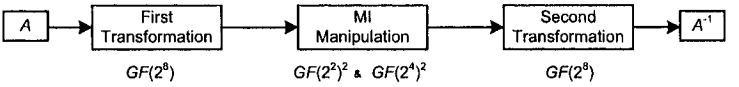

Fig. 9.11. Block Diagram for 3-Stage MI Manipulation

Figures 9.11 and 9.12 depict block diagram to three-stage inverse multiplier represented by Equations 9.15 and 9.17.

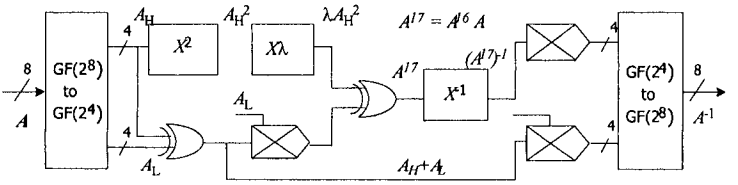

Fig. 9.12. Three-Stage Approach to Compute Multiplicative Inverse in Composite Fields

As it was explained before, in order to obtain the multiplicative inverse of the element $A \in F$ =GF(2^8), we first map A to its equivalent representation (A_H, A_L) in the isomorphic field $F_2 = \text{GF}\left((2^2)^2\right)$ using the isomorphism δ (and its corresponding inverse δ^{-1}). In order to map a given element A from the finite field F to its isomorphic composite field F_2 and vice versa, we only need to compute the matrix multiplication of A, by the isomorphic functions shown in Equation 9.18 given by [242]:

$$\delta = \begin{bmatrix} 1\,0\,1\,0\,0\,0\,0\,0 \\ 1\,1\,0\,1\,1\,1\,1\,0 \\ 1\,0\,1\,0\,1\,1\,0\,0 \\ 1\,0\,1\,0\,1\,1\,1\,0 \\ 1\,1\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,1\,1\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,0\,0\,1\,1 \end{bmatrix} \quad \delta^{-1} = \begin{bmatrix} 1\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,0\,0\,0\,1\,0\,0 \\ 0\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,1\,1\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,1\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1\,1 \\ 0\,1\,1\,1\,0\,1\,0\,1 \end{bmatrix} \tag{9.18}$$

The isomorphism function δ and δ^{-1} can be constructed as follows:

Let α and β be roots of a same primitive irreducible polynomial ($m(x) = x^8 + x^4 + x^3 + x^2 + 1$ can be used). First search for primitive element α in the field A and then search for β in the field B. Once δ and δ^{-1} are founded, the matrix representation can be obtained, where α^k is mapped to β^k or vice versa. Note that there could be more than one eligible isomorphism.

Also by taking advantage of the fact that A^{17} is an element of F_2, the final operation $(A^{17})^{-1}A^{16}$ of Equation 9.15 can be easily computed with further gate reduction. Last stage of algorithm consists of mapping computed value in the composite field, back to the field GF(2^8).

To further increase the depth of a pipeline architecture, MI can be calculated by a composite field approach dealing MI manipulation in GF(2^2) and GF(2^4) instead of GF(2^8).

In [113], BS has been computed rather than using a look-up table. The main goal of using this formulation is to get a high-performance AES encryptor core without depending on look-up tables.

Using the composite field technique, BS arithmetic in GF(2^8) is performed via several arithmetic blocks in GF(2^4). This effectively reduces an 8-bit calculation to a 4-bit one, resulting on several stages of computation with lower delays. That allows obtaining a sort of sub-pipelining architecture in which, instead of having 11 unfolded stages (each stage corresponding to a single round), each single round is further unfolded into several stages. Thus, BS is (sub)divided into four pipeline stages where the first round takes only one stage, each middle round takes seven stages, and the final round, in which MC is not required, takes six stages.

In order to keep all stages balanced, i.e., propagating similar delays, a pipeline architecture with a depth of 70 stages was proposed in [113]. After 70 clock cycles when the pipeline is full, each clock cycle will deliver a ciphered block. This technique achieves a throughput of 25.107 Gbps, the fastest one reported up to date of this book publication.

The idea of dividing computations in sub fields is further exploited to its extreme in [42], where 4-bit calculations are broken into several 2-bit ones. Authors in [42] explored as many as 432 different isomorphisms. Polynomial as well as normal basis were considered and using an exhaustive tree- search algorithm [153], those isomorphisms requiring the minimum number of gates were selected. Logic optimizations both at the hierarchical level of the Galois

Field arithmetic and at the low level of individual logic gates were performed. The authors also reused common expressions to save space and noticing that NAND gates take less space than other ones, they rewrite all expressions in terms of such gates. Authors reported results exploring a family of 432 implementations depending on the selected basis ranking from 138 to 195 gates. Such compact $S-$box implementations can be used in security for low-end customer products, such as PDAs, wireless devices and other embedded applications.

9.4.2 MC/IMC Implementations on FPGA

The MC/IMC transformations are essentially the inner-product operations on GF(2^8) expressed in equations 9.6 and 9.9. They can be realized using byte-level or bit-level substructure sharing methods [140].

For an encryptor/decryptor core MC/IMC steps are implemented separately and they can be realized in a small series of instructions. In case of FPGAs, these instructions can be realized by keeping in mind the basic CLB structure (4 input/1 output) in order to limit path delays and to save space. Let us call this approach the MC/IMC standard approach. Fortunately, there exists another approach for which the implementation of IMC is made by introducing small modification before MC. The first approach is efficient but needs separate implementation for MC and IMC. The MC/IMC modified approach reuses some modules which eliminates the need for separated implementation of MC/IMC.

MC and IMC Transformation: Standard Approach

Observing that constant terms in equations 9.6 and 9.9 are the same, it is possible to consider only the inner product that generates one output byte, Z in MC and Z_{inv} in IMC, for an input column $[ABCD]^T$:

$$Z = \{01\}A \oplus \{01\}B \oplus \{02\}D \oplus \{03\}E \tag{9.19}$$

Using the property of $\{02\}D = \{02\}D \oplus 0 = \{02\}D \oplus D \oplus D$, we can rewrite equation 9.19 in the following manner:

$$Z = (A \oplus B \oplus D \oplus E) \oplus \{02\}(D \oplus E) \oplus D) \tag{9.20}$$

We can use an efficient implementation of constant multiplication by 02 in GF(2^8) calculated by the functional block $xtime(v)$ and extracting the common factor in all columns $t = (A \oplus B \oplus D \oplus E)$, then equation 9.19 can be rewritten as:

$$Z = t \oplus xtime(D \oplus E) \oplus D) \tag{9.21}$$

Therefore, full MC transformation can be efficiently computed by using only 3 steps [21, 60]: an addition step, a doubling step and a final addition step.

Let us consider a complete output row of MC transformation. Consider now the element of *State* matrix's column one $a[0]$, $a[1]$, $a[2]$, and $a[3]$, then the transformed MC column $a'[0]$, $a'[1]$, $a'[2]$, and $a'[3]$ can be efficiently obtained as shown in Equation 9.22.

$$\begin{array}{c} t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; \\ v = a[0] \oplus a[1];\ v = xtime(v);\ a'[0] = a[0] \oplus v \oplus t; \\ v = a[1] \oplus a[2];\ v = xtime(v);\ a'[1] = a[1] \oplus v \oplus t; \\ v = a[2] \oplus a[3];\ v = xtime(v);\ a'[2] = a[2] \oplus v \oplus t; \\ v = a[3] \oplus a[0];\ v = xtime(v);\ a'[3] = a[3] \oplus v \oplus t; \end{array} \tag{9.22}$$

Observe that t is a common expression for the four outputs and it needs to be calculated just once. Next four rows are calculated in parallel and the circuit is the same except for some input data. Finally, the sum of three terms requires only eight CLBs, one per bit. Given that CLBs can compute 4-input/1-output functions, it is possible to embed the ARK transformation, which is just a sum, to the final expression. This does not require additional CLBs and improves performance since MC and ARK are computed at the same stage. This is expressed in the following manner:

$$\begin{array}{ll} \multicolumn{1}{c}{Step1} & \multicolumn{1}{c}{Step2} \\ v = a[1] \oplus a[2] \oplus a[3]; & xt_0 = xtime(a[0]); \\ v = a[0] \oplus a[2] \oplus a[3]; & xt_1 = xtime(a[1]); \\ v = a[0] \oplus a[1] \oplus a[3]; & xt_2 = xtime(a[2]); \\ v = a[0] \oplus a[1] \oplus a[2]; & xt_3 = xtime(a[3]); \end{array}$$

$$\begin{array}{c} Step3 \\ a'[0] = k[0] \oplus v \oplus xt_0 \oplus xt_1; \\ a'[1] = k[1] \oplus v \oplus xt_1 \oplus xt_2; \\ a'[2] = k[2] \oplus v \oplus xt_2 \oplus xt_3; \\ a'[3] = k[3] \oplus v \oplus xt_3 \oplus xt_0; \end{array} \tag{9.23}$$

The same strategy applied above for MC can be used to compute IMC. Considering again an input column $[ABCD]^T$, we can expressed Z_{inv} as:

$$Z_{inv} = \{0d\}A \oplus \{09\}B \oplus \{0e\}D \oplus \{0b\}E \tag{9.24}$$

Using the same property for constant multiplication by $\{02\}$, we can rewrite Equation 9.24 in the following manner:

$$Z_{inv} = D \oplus N \oplus xtime(M \oplus N) \oplus xtime(D \oplus E) \tag{9.25}$$

where:

$$
\begin{aligned}
T_0 &= A \oplus B \oplus D \oplus E \\
T_1 &= T_0 \oplus xtime(xtime(T_0)) \\
N &= T_1 \oplus xtime(xtime(B \oplus E)) \\
M &= T_1 \oplus xtime(xtime(A \oplus D))
\end{aligned}
$$

Full IMC transformation can be computed by using seven steps: four sum steps and three doubling steps. The difference is due to the fact that coefficients in Equation 9.9 have a higher Hamming weight than the ones in Equation 9.6. To overcome this drawback, we use the strategy depicted in Equation 9.25 where IMC manipulation is restructured and seven steps are cut to five steps. Moreover, as explained above, IARK is embedded into IMC resulting in six total steps. For final round (Round 10), MC/IMC steps are not executed; therefore a separated implementation of ARK can be made. Let us consider now a complete output row of IMC transformation embedded with and IARK transformation, where a, and a' stand as before.

Step 1	*Step 2*	*Step 3*
$t = a[0] \oplus a[1] \oplus a[3]$		$u = s'_0 \oplus s'_1 \oplus s'_2 \oplus s'_3;$
$s_0 = xtime(a[0]);$	$s'_0 = xtime(s_0);$	$v = s_0 \oplus s_1 \oplus s'_0 \oplus s'_2;$
$s_1 = xtime(a[1]);$	$s'_1 = xtime(s_1);$	$v = s_1 \oplus s_2 \oplus s'_1 \oplus s'_3;$
$s_2 = xtime(a[2]);$	$s'_2 = xtime(s_2);$	$v = s_2 \oplus s_3 \oplus s'_0 \oplus s'_2;$
$s_3 = xtime(a[3]);$	$s'_3 = xtime(s_3);$	$v = s_3 \oplus s_0 \oplus s'_1 \oplus s'_3;$

Step 4	*Step 5*	*Step 6*
$u = xtime(u);$	$t' = t \oplus u;$	$a'[0] = a[0] \oplus t' \oplus v \oplus k[0];$
		$a'[1] = a[1] \oplus t' \oplus v \oplus k[1];$
		$a'[2] = a[2] \oplus t' \oplus v \oplus k[2];$
		$a'[3] = a[3] \oplus t' \oplus v \oplus k[3];$

(9.26)

MC and IMC Transformation: Modified Approach

The strategy utilized above for MC and IMC yields up to three and six computational steps for encryption and decryption respectively. In order to minimize difference in number of steps, the following strategy can be used.

Observe that it should exist a 4×4 byte matrix $D(x)$ in $\mathrm{GF}(2^8)$ such that the constant MC matrix of Equation 9.6 can be related to the constant matrix of Equation 9.9 as,

$$
\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} D(x) \tag{9.27}
$$

Using the fact that both constant matrices in Equation 9.27 are the inverse of each other in the finite field $F = \mathrm{GF}(2^8)$, equation 9.27 can be solved using the AES irreducible pentanomial $m(x) = x^8 + x^4 + x^3 + x + 1$ [60] for the first column of $D(x)$ as shown in Equation 9.28.

$$\begin{bmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 0E \\ 09 \\ 0D \\ 0B \end{bmatrix} \tag{9.28}$$

where $d_{i,0}$, $i = 0, 1, 2, 3$ represent the four coefficients of the first column of $D(x)$. It follows that Equation 9.28 has a unique solution in the finite field F as given in Equation 9.29,

$$d_{0,0} = 5 \quad d_{1,0} = 0 \quad d_{2,0} = 4 \quad d_{3,0} = 0 \tag{9.29}$$

Hence, Equation 9.27 can be re-written as shown in Eq. 9.30.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix} \tag{9.30}$$

Equation 9.30 suggests an efficient way to compute IMC by re-using the MC transformation to obtain IMC constant matrix. This is useful since constant elements of second matrix in the right side of Equation 9.30 have a less Hamming weight as compared to the constants of the original matrix for IMC.

9.4.3 Key Schedule Optimization

Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key arranged into 4 x 4 matrix of bytes. Then, those four columns are recursively expanded to obtain 40 more columns as follows. Let the columns up to $w[i-1]$ have been determined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \tag{9.31}$$

Where $T(w[i-1])$ a is non-linear transformation based on the application of the S-Box to the four bytes of the column. It involves also an additional cyclic rotation of the bytes within the column and the addition of a round constant ($rcon$) for symmetric elimination [60]. Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be represented as:

$$w[0] = \begin{bmatrix} k_0 \\ k_4 \\ k_8 \\ k_{12} \end{bmatrix} \quad w[1] = \begin{bmatrix} k_1 \\ k_5 \\ k_9 \\ k_{13} \end{bmatrix}$$
$$w[2] = \begin{bmatrix} k_2 \\ k_6 \\ k_{10} \\ k_{14} \end{bmatrix} \quad w[3] = \begin{bmatrix} k_3 \\ k_7 \\ k_{11} \\ k_{15} \end{bmatrix} \tag{9.32}$$

Then according to the above expressions, the new columns $w'[0]$, $w'[1]$, $w'[2]$, and $w'[3]$ of the next round key can be calculated as shown in Equation 9.33.

$$\begin{array}{ll} \textit{Step 1} & \textit{Step 2} \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_4 = k_4 \oplus k'_0; \\ k'_1 = k_0 \oplus SBox(k_{14}); & k'_5 = k_5 \oplus k'_1; \\ k'_2 = k_0 \oplus SBox(k_{15}); & k'_6 = k_6 \oplus k'_2; \\ k'_3 = k_0 \oplus SBox(k_{12}); & k'_7 = k_7 \oplus k'_3; \end{array}$$

$$\begin{array}{ll} \textit{Step 3} & \textit{Step 4} \\ k'_8 = k_8 \oplus k'_4; & k'_{12} = k_{12} \oplus k'_8; \\ k'_9 = k_9 \oplus k'_5; & k'_{13} = k_{13} \oplus k'_9; \\ k'_{10} = k_{10} \oplus k'_6; & k'_{14} = k_{14} \oplus k'_{10}; \\ k'_{11} = k_{11} \oplus k'_7; & k'_{15} = k_{15} \oplus k'_{11}; \end{array} \tag{9.33}$$

But it was mentioned before that in a typical FPGA device, a 4 input look-up table can be configured indistinctly to handle 2, 3, or 4 input logic gates. Hence, we can save some time by parallelizing the above computation using only two steps. By applying redundant computations, Equation 9.33 can be rewritten as it is shown in Equation 9.34 for the first row. Parallel computations are applied to obtain k'_4, k'_8, and k'_{12}.

$$\begin{array}{ll} \textit{Step1} & \textit{Step2} \\ & k'_4 = k_4 \oplus k'_0; \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_8 = k_4 \oplus k_8 \oplus k'_0; \\ & k'_{12} = k_4 \oplus k_8 \oplus k_{12} \oplus k'_0; \end{array} \tag{9.34}$$

9.5 AES Implementations on FPGAs

The basic organization of the hardware implementation of the AES algorithm is shown in Figure 9.13 which represents three blocks: encryptor/decryptor

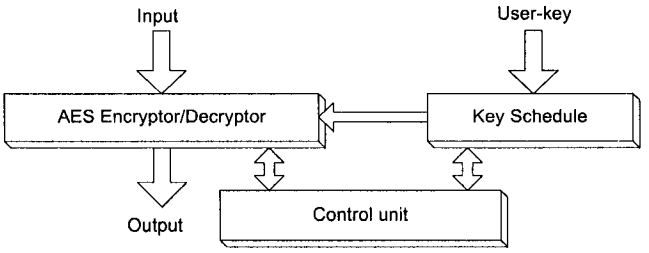

Fig. 9.13. Basic Organization of a Block Cipher

unit, key scheduling unit, and a control unit for synchronizing the flow of data between them.

Three main processes participate in AES:

- Key Schedule
- Encryption
- Decryption

The above three processes can be implemented using different design strategies showing distinct time-area tradeoffs. Depending on the application specification, the AES implementation can be carried out for just encryption, encryption/decryption on the same chip, separate encryption and decryption cores, or simply decryption. A separate implementation of AES encryptor or decryptor core would be less complex and efficient. Implementing AES encryptor/decryptor core on a single chip FPGA by mixing their common blocks, will give out an area efficient solution but one of them, either encryption or decryption could be performed at a time. To develop a full duplex operation having a capability to perform both encryption and decryption simultaneously would require relatively high hardware resources and consequently would become a bit slow.

For AES, key schedule implementations are different for an encryptor, decryptor or encryptor/decryptor cores. The usage of internal memory resources of an FPGA for storing pre-computed round-keys would be a simple approach. For encryption/decryption processes however it is recommendable not to use the same key for long time. A key schedule implementation will therefore provide a user the added flexibility of selecting encryption/decryption key of his own choice at any given time.

9.5.1 Architectural Alternatives for Implementing AES

Several approaches can be followed to implement AES on hardware achieving variable performance results [218].

Iterative architectures implement a reduced number of rounds (typically one) in an independent fashion. This kind of architectures occupy small area

of circuits but at the expense of low throughput. Unrolled architectures have a large number of rounds that are independently implemented in hardware. Pipelining allows to process multiples blocks of data at the same time at different stages to have higher throughput. Pipelining is achieved by putting rows of registers among different stages. Sub-pipelining inserts registers inside the round transformation to create sub-stages.

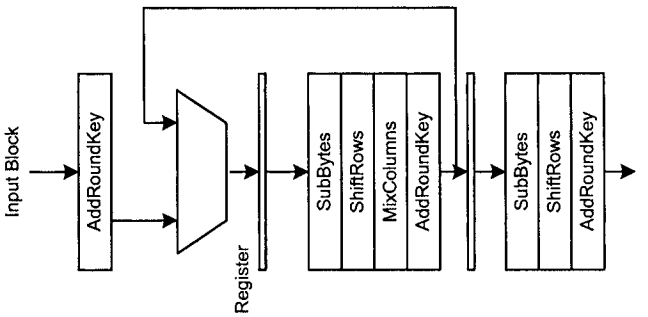

Fig. 9.14. Iterative Design Strategy

Block ciphers are of iterative nature, that is, n iterations of the same algorithm are made for a single encryption/decryption. An iterative design strategy would be a straightforward approach to implement the algorithm which executes n iterations of it by consuming n clock cycles for a single encryption/decryption as shown in Figure 9.14. The first round only considers ARK, the next nine rounds implement the four basic transformation, BS, SR, MC and ARK. The last round implements all but MC transformation. Clearly, it is an economical approach with respect to the hardware area and the cost has to be paid in terms of design speed which gets reduced with a factor of n. Such architectures would be useful for applications where hardware area is limited and speed is not more critical.

If reconfigurable platform is the choice for the implementation of a block cipher, a high speed architecture would result by implementing n rounds of the algorithm as modern FPGAs have enough logic density to accommodate massive circuits. The simplest way to improve performance is to use loop unrolling that expand the iterative structure by replicating rounds and conecting the output to the input of two consecutive rounds. This architecture is shown in Figure 9.15. By eliminating switches (multiplexers) and registers the accumulated delay can be reduced, but the duplication of multiple rounds incurs in large critical paths, which implies lower clock frequencies.

By putting registers between two consecutive rounds, which operate at the same clock cycle, we can achieve a pipeline architecture as shown in Figure 9.16.

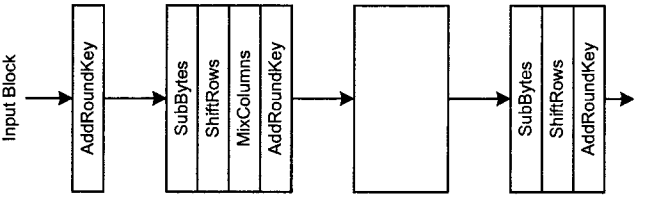

Fig. 9.15. Loop Unrolling Design Strategy

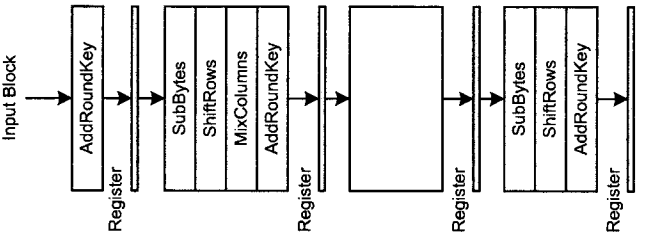

Fig. 9.16. Pipeline Design Strategy

Each round forms a pipeline stage of the data flow. The critical path is cut into stages although it is not diminished. The main advantage is that several different blocks can be processed at the same time but in different rounds of the encryption/decryption process. Once the pipeline is filled, the output blocks appear at each successive clock cycle. This allows to increase performance multiplied by the number of rounds or stages in the pipeline (typically eleven). This architecture increases throughput but it becomes costly in terms of hardware area.

FPGAs provide large number of flip-flops, which can be used to put several registers inside the different steps of a single round for a pipeline design strategy. This improves the performance of a pipeline architecture as those registers shift the internal results of a round while the final results are being transferred to the next round. It has been observed that careful use of those registers inside a round causes a significant increase in design performance. Figure 9.17 represents a sub-pipeline design strategy. This approach increases the depth of the pipeline up to 40 stages.

Although one can think that the increase in performance is folded as many times as the number of stages this is not completely exact. The problem is that all stages must have similar delays which is not true for AES. According to the formulation of BS, it is clear that its implementation takes longer delays than other basic transformations.

To keep balanced stages and at the same time to increase the depth of pipeline, we can break BS calculation by a four-stage composite field approach as it was explained in Section 9.4.1 and it is shown in Figure 9.18. Each middle

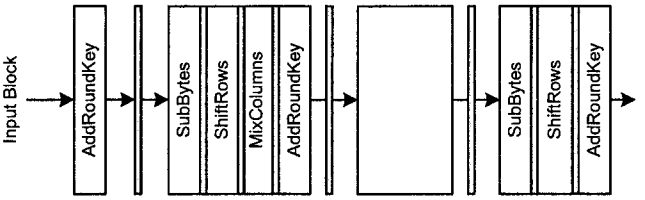

Fig. 9.17. Sub-pipeline Design Strategy

round is decomposed into seven stages, four from BS and one for SR, MC and ARK, each. That gives a 70 stages pipeline approach which reports high performance at the expense of great area requirements.

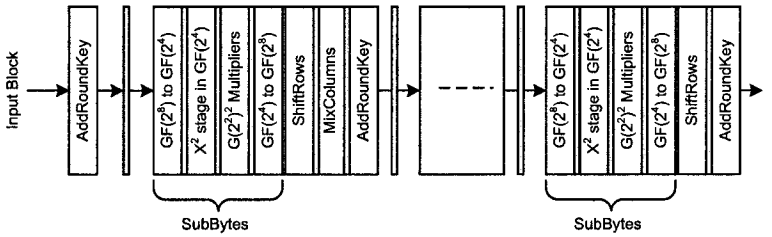

Fig. 9.18. Sub-pipeline Design Strategy with Balanced Stages

Pipelining and sub-pipelining are useful only when the cipher block is used in the ECB mode (electronic code book). As it was mentioned in Section 9.3, in the Output Feedback Mode (OFB) and in the CCM mode (Counter with CBC-MAC), pipelining looses its potential since a cipherblock is used to encrypt the next block. The only acceptable architecture for feed back modes is the iterative one, also called loop architecture.

In the rest of this section we disccuss some alternatives for implementing AES. All of them are intended to be implemented on a single-chip FPGA. There exists multi-chip implementations but as FPGA density is increasing, those implementations would be less meaningful in the future.

Varieties for AES implementation include encryptor, decryptor, and encryptor/decryptor cores using iterative or pipeline approaches. Each AES implementation targets specific criteria composed of factors like efficiency, cost, effectiveness and portability. Table 9.2 provides a roadmap to all implemented AES designs. It consideres four parameters: design (Sec.9.5), based on Section (Sec. 9.4), E/D/K module (encryption/decryption/key schedule) and architecture (encryptor, decryptor or encryptor/decryptor core). Key schedule implementations for encryptor, decryptor and encryptor/decryptor cores are also presented.

Table 9.2. A Roadmap to Implemented AES Designs

Design	Based on the Section	E/D/K Module	Architecture
Sec. 9.5.2	Sec. 9.4.3	(Key schedule)	For iterative & pipeline encryptor cores only
Sec. 9.5.2	Sec. 9.4.3	(Key schedule)	For Pipeline encryptor/decryptor cores
Sec. 9.5.3	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Iterative)
Sec. 9.5.3	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. 9.5.4	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor/decryptor core (Pipeline)
Sec. 9.5.4	Sec. 9.4.1 Sec. 9.4.2	S-box Composite field MC classic	Encryptor/decryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table Modified MC/IMC	Encryptor/decryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table Modified IMC	Decryptor core (Pipeline)

All designs presented in this section were completely synthesized and succesfully implement using Xilinx Foundation Tool F4.1i. All designs are either coded in VHDL or by using libraries of the target devices. CoreGenerator is another tool used for design entry.

9.5.2 Key Schedule Algorithm Implementations

Let the user key consisting of 16 bytes be arranged as:

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix} \tag{9.35}$$

The process of generating next round key is optimized as discussed in Section 9.4.3 and is shown in Figure 9.19. The KGEN block consists of four similar units where each unit contains an S-Box and four XORs. The first block is slightly different as a constant predefined value (*rcon*) is XOR-ed in each round. As shown in Figure 9.19, last four bytes $k_{12}, k_{13}, k_{14}, k_{15}$, of each round key are substituted with the bytes from S-Box and then various XOR operations are performed to get the next round key.

The KGEN block is the basic building block used to generate round Keys for all AES implementations. However, the key management for producing

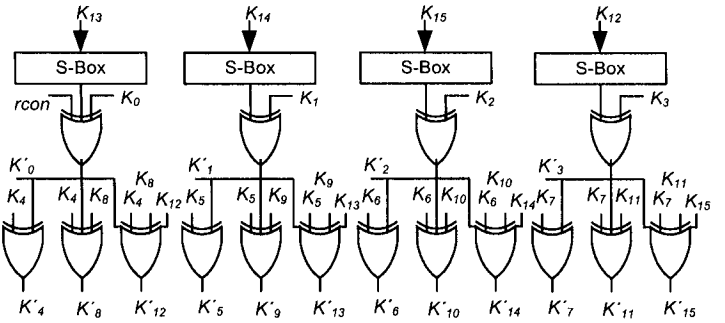

Fig. 9.19. KGEN Architecture

round keys differs depending on the particular implementation's strategy being used. For an encryptor core in iterative mode, round keys are also generated in iterative mode. For fully pipeline encryptor core, all round keys must be available before the encryption process starts. In a fully pipeline encryptor/decryptor core, the round keys for decryption are stored in reverse order as that of encryption.

Key Schedule for Iterative and Pipeline Encryptor Cores

For an encryptor core in iterative mode, a single round key is generated. The round key is fed to perform ARK step and also latched to feed back to KGEN block in order to get prepared for processing the next round key as shown in Figure 9.20. A multiplexer is used to switch the user-key first time and then for all rounds, each round key is used to generate the next round key.

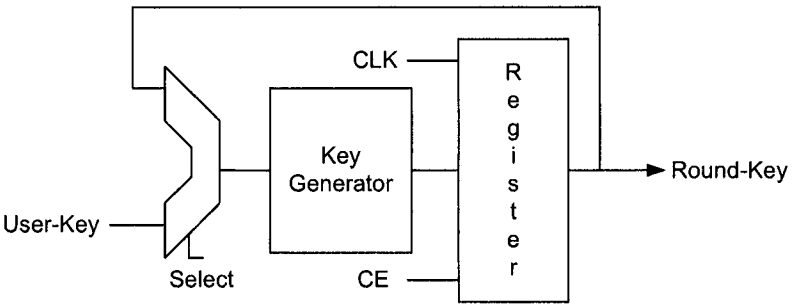

Fig. 9.20. Key Schedule for an Encryptor Core in Iterative Mode

For a fully pipelined encryptor core, the round keys must be available for each round permanently. The key generation process for a fully pipeline encryptor core is shown in Figure 9.21. The internal structure of each block is the same as shown in Figure 9.20, however, same block is replicated n (number of rounds) times. Once the round keys are generated, there is no need to repeat this process again and again. The same round keys serve for the whole session. For a fully pipeline encryptor core, the encryption process can be started in a parallel way, and there is no need to wait for the completion of all round keys.

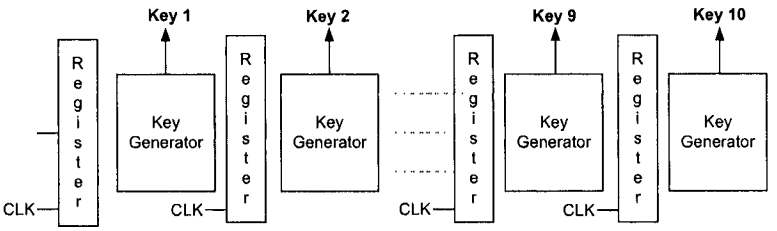

Fig. 9.21. Key Schedule for a Fully Pipeline Encryptor Core

Key Schedule for Encryptor/Decryptor Cores

For an encryptor/decryptor core on a single-chip FPGA, all the round keys must be generated and latched before the encryption/decryption processes start. The reason why round keys cannot be generated in a parallel way is because they are required in reverse order for decryption. The process of key generation is the same as explained above, however, round keys are stored in the registers for encryption and decryption in ascending or descending order respectively as shown in Figure 9.22. Besides this difference, the same blocks can be used for encryption and decryption processes.

As shown in Figure 9.22, round keys are generated by KGEN block as it was explained above by introducing two modifications. The first one deals with the generation of select signals (s_i) through an up/down counter. The main purpose of having those select signals is to choose the correct order for round keys either for the encryption or for the decryption process.

The second modification is the addition of IMC step which is required for generating round keys for decryption. It is applied through a multiplexer that allows passing round keys directly for encryption and switches the other line for applying IMC operation for the decryption round keys. IMC operation is performed before all the round keys are latched in their registers. Obeying algorithm description of the AES decryption process, this modification is not applied to first and last round keys.

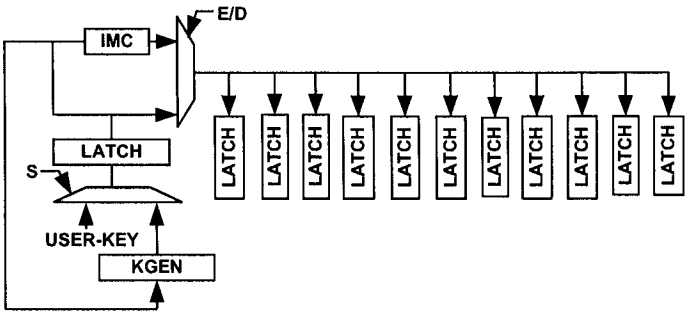

Fig. 9.22. Key Schedule for a Fully Pipeline Encryptor/Decryptor Core

IMC modifications discussed in Section 9.4.2 are applied in the IMC step for key scheduling as shown in Figure 9.23. This module is part of the second AES encryptor/decryptor core to be explained in the next Section.

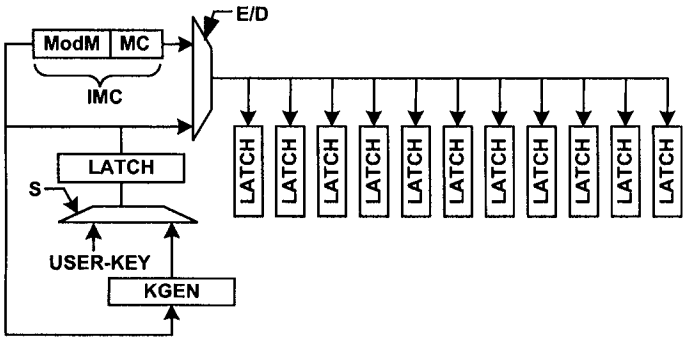

Fig. 9.23. Key Schedule for a Fully Pipeline Encryptor/Decryptor Core with Modified IMC

9.5.3 AES Encryptor Cores - Iterative and Pipeline Approaches

FPGAs implementations of AES encryptor cores are carried out using two strategies: iterative and pipeline.

AES Encryptor Core Using an Iterative Approach

For an iterative approach, instead of implementing n iterations of the algorithm, one iteration is implemented and n clock cycles are consumed to achieve final output. An AES iterative approach is shown in Figure 9.24.

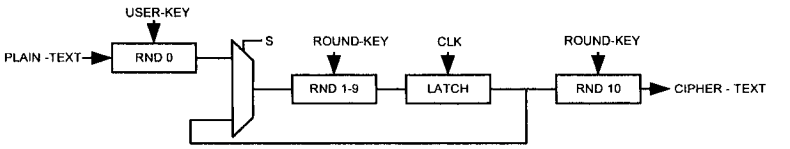

Fig. 9.24. Iterative Approach for AES Encryptor Core

The encryption process is presented in Figure 9.24, where RND0 is a simple ARK step: the user-key and plain-text are added. The RND1-9 block includes the four AES steps, namely, BS,SR,MC,ARK. Round keys are generated for all iterations of the algorithm. A multiplexer selects RND0 output at the first cycle and then selects the latch output for RND1-9 during the next nine cycles. RND10 is implemented separately without including the MC step.

The latch output is connected to the RND10 block and it is also fed-back to the multiplexer. All latch outputs passes through RND10 block but only during the tenth cycle its output is collected giving the final result. No clock cycle is therefore consumed to perform RND10.

Sixteen ROMs (256×8) are configured by using CLB in memory mode for performing the BS step of RND1-9. Since RND10 also includes the BS step, sixteen more ROMs are required for this step. The key scheduling algorithm also includes the BS step for the last four bytes of each round key (See Section 9.5.2) as shown in Figure 9.19, occupying four extra ROM blocks. A total of 36 ROM blocks are used for encryption part only. The SR step is combined with BS step. The MC and ARK steps are combined to reduce area requirements as discussed in Section 9.4.2.

The design was implemented on Xilinx VirtexE FPGA devices (XCV812BEG). It utilizes 36 ROMs, 385 I/O Blocks (95%) and 2744 slices (28%) to achieve a throughput of 258.5 Mbits/sec at 20.192 MHz. An encryption is completed in 10 clock cycles. That design does not make use of FPGA dedicated resources (BRAMs, etc.), hence it has a high portability and can be implemented virtually in every commercial FPGA device.

Fully Pipeline AES Encryptor Core

For a pipeline architecture, all AES rounds are unrolled. That is achieved by repeating one AES round 11 times as shown in Figure 9.25.

Similar to the iterative architecture, RND0 is just ARK step. The RND1-9 block includes all four steps BS, SR, MC, and ARK. The RND10 includes three steps BS, SR, ARK excluding MC step. 160 ROMs are required for 10 AES rounds instead of 16 ROMs occupied by the iterative architecture to perform BS step. Typically, the critical data path in pipeline architecture is longer, which implies that the design can run at lower speeds. However, by using dedicated memory modules BRAMs, as explained in the introduction Section, it is possible to reduce critical path delays.

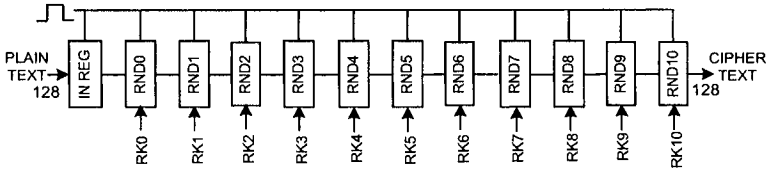

Fig. 9.25. Fully Pipeline AES Encryptor Core

The Virtex and VirtexE FPGA devices [397, 396] contain more than 280 BRAMs each of 4K. Each dual port BRAM can be configured as two single port BRAMs which reduces half of the memory requirements. A total of 80 BRAMs are therefore used to perform BS step. The same approach is used for key schedule implementation by occupying 20 BRAMs instead of 40 ROMs.

The design is targeted to Xilinx VirtexE FPGA devices (XCV812BEG) and occupies 2136 CLB slices (22%), 385 I/O Blocks (95%) and 100 BRAMs (35%). It uses a system clock of 22.41 MHz and data is processed at a rate of 2868 Mbits/sec. For a fully pipeline encryptor core, encryption starts from first clock cycle without initial delay. The round keys are generated in parallel. It takes 11 clock cycles to fill the pipeline first and then encrypted blocks start appearing at each consecutive clock cycle.

At first look, a comparison of the iterative and pipeline architectures suggests that the number of CLB slices occupied by the pipeline architecture seems to be less as compared to an iterative architecture. But this is accomplished at the price of occupying extra memory (100 BRAMs) needed to achieve desired fully pipeline architecture. The usage of dedicated memory resources (BRAMs) makes the pipeline design importable as it can only be targeted to those FPGA devices equipped with embedded memory functionality.

9.5.4 AES Encryptor/Decryptor Cores- Using Look-Up Table and Composite Field Approaches for S-Box

For an encryptor/decryptor core, each encryption step (BS, SR, MC, ARK) has its own inverse (IBS, ISR, IMC, IARK) which has to be implemented separately. The implementation of BS and IBS on a single chip is the most costly operation for AES implementation on FPGAs. In this design, two architectures are proposed for the BS/IBS implementation on FPGAs. First architecture proposes high performance implementations of BS/IBS step and second architecture is based on on-fly architecture scheme which tries to reduce memory requirements. The implementation of the remaining three steps SR, MC, and ARK is the same as the one described in Section 9.5.3. In the following, BS/IBS implementation strategies are discussed.

For encryption, BS implementation can be made by computing the Multiplicative Inverse (MI) of the input byte in $GF(2^8)$ followed by the affine

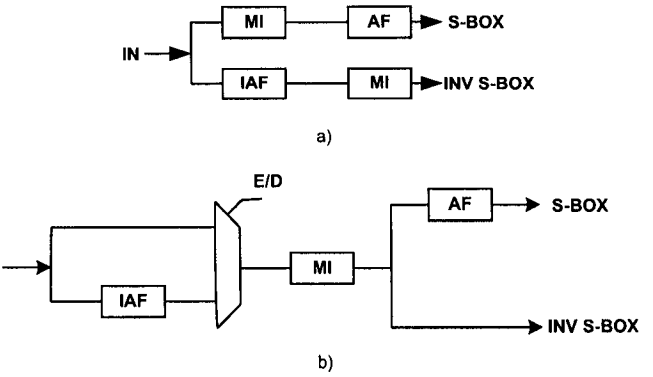

Fig. 9.26. S-Box and Inv S-Box Using (a) Different MI (b) Same MI

transformation (AF). For decryption, inverse affine transformation (IAF) is applied first followed by MI step. Implementing MI as look-up table requires memory modules, therefore, a separated implementation of BS/IBS causes the allocation of high memory requirements especially for a fully pipelined architecture. We can reduce such requirements by developing a single data path which uses one MI block for encryption and decryption. Figure 9.26 shows the BS/IBS implementation using single block for MI.

There are two design approaches for implementing MI: look-up table method and composite field calculation.

MI Using Look-Up Table Method

MI can be implemented using memory modules (BRAMs) of FPGAs by storing pre-computed values of MI. By configuring a dual port BRAM into two single port BRAMs, 8 BRAMs are required for one stage of a pipeline architecture, hence a total of 80 BRAMs are used for 10 stages. A separated implementation of AF and IAF is made. Data path selection for encryption and decryption is performed by using two multiplexers which are switched depending on the E/D signal. A complete description of this approach is shown in Figure 9.27

The data path for both encryption and decryption is, therefore, as follows:

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: ISR→ IAF→ MI→ IMC→IARK

The design targets Xilinx VirtexE FPGA devices (XCV2600) and occupies 80 BRAMs (43%), 386 I/O blocks (48%), and 5677 CLB slices (22.3%). It runs at 30 MHz and data is processed at 3840 Mbits/s.

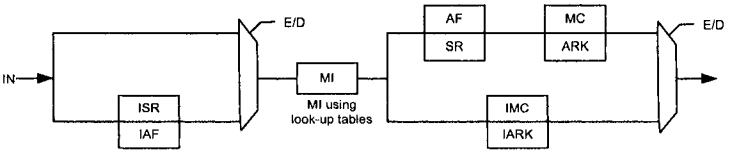

Fig. 9.27. Data Path for Encryption/Decryption

The data blocks are accepted at each clock cycle and then after 11 cycles, output encrypted/decrypted blocks appear at the output at consecutive clock cycles. It is an efficient fully pipeline encryptor/decryptor core for those cryptographic applications where time factor really matters.

MI with Composite Field Calculation

This is composite field approach that deals with MI manipulation in $GF(2^2)$ and $GF(2^4)$ instead of $GF(2^8)$ as it was explained in Section 9.4.1. It is a 3-stage strategy as shown in Figure 9.28.

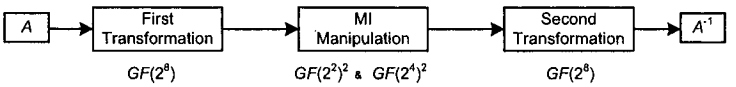

Fig. 9.28. Block Diagram for 3-Stage MI Manipulation

First and last stages transform data from $GF(2^8)$ to $GF(2^4)$ and vice versa. The middle stage manipulates inverse MI in $GF(2^4)$. The implementation of the middle stage with two initial and final transformations is represented in Figure 9.29 which depicts a block diagram of the three-stage inverse multiplier represented by Equations 9.15 and 9.17. It is noted that the Data path for encryption/decryption for this approach remains the same as the change in this approach is introduced in the MI manipulation.

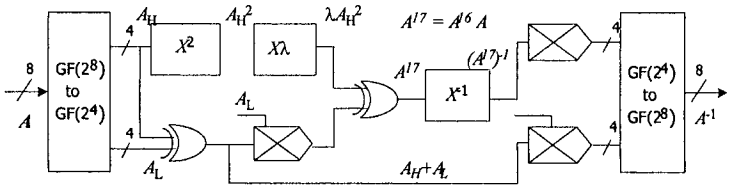

Fig. 9.29. Three-stage to Compute Multiplicative Inverse in Composite Fields

The circuit shown in Figure 9.30 and Figure 9.31 present a gate level implementation of the aforementioned strategy.

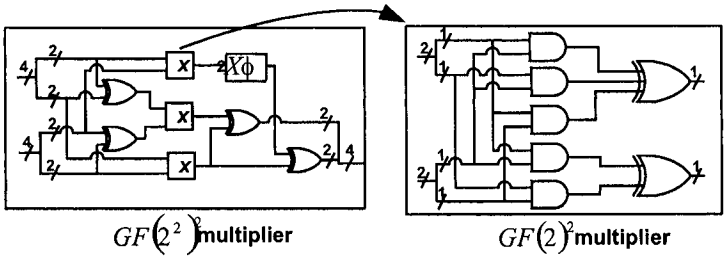

Fig. 9.30. $GF(2^2)^2$ and $GF(2^2)$ Multipliers

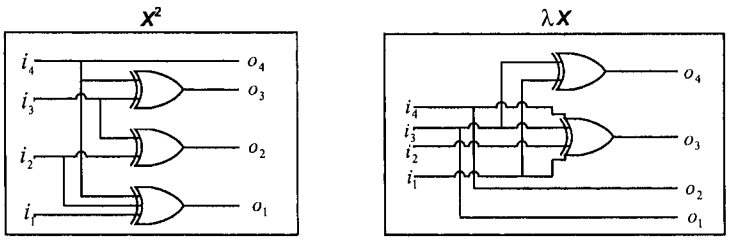

Fig. 9.31. Gate Level Implementation for x^2 and λx

The architecture is implemented on Xilinx VirtexE FPGA devices (XCV2600BEG) and occupies 12,270 CLB slices (48%), 386 I/O blocks (48%). It runs at 24.5 MHz and throughput achieved is 3136 Mbits/s. The increment on CLB slices utilized for this design is due to the manipulation for MI instead of using BRAMs. The increased design complexity causes the throughput to decrease when compared against the first design.

9.5.5 AES Encryptor/Decryptor, Encryptor, and Decryptor Cores Based on Modified MC/IMC

Three AES cores are presented in this Section. First design is an encryptor/decryptor core based on the ideas discussed in Section 9.4.2 for MC/IMC implementations. The second and third designs implement encryption and decryption paths separately for that design. There are two main reasons for the

separate implementation of encryption and decryption paths. First, to realize the effects of the modifications introduced in MC/IMC transformations. Second, most of reported AES implementations are either encryptor cores or encryptor/decryptor cores and few attention has been put to decryptor only cores.

Encryptor/Decryptor Core

This architecture reduces the large difference between the encryption/decryption time by exploiting the ideas explained in Section 9.4.2 for MC/IMC transformations. For this design, BS/IBS implementations are made by storing pre-computed MI values in FPGA's memory modules (BRAMs) with separate implementation of AF/IAF as explained in Section 9.5.4. The MC and ARK are combined together for encryption and a small modification ModM is applied before MC+ARK to get IMC operation as shown in Figure 9.32. Two multiplexers are used to switch the data path for encryption and decryption.

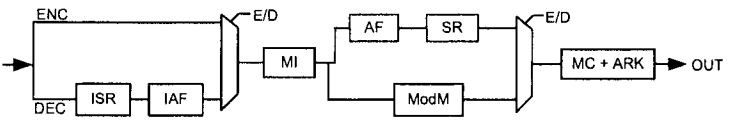

Fig. 9.32. AES Algorithm Encryptor/Decryptor Implementation

The data path for both encryption and decryption is, therefore, as follows:

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: ISR→ IAF→ MI→ ModM→ MC→ ARK

This AES encryptor/decryptor core occupies 80 BRAMs (43%), 386 I/O Blocks (48%) and 5677 slices (22.3%) by implementing on Xilinx VirtexE FPGA devices (XCV812BEG). It uses a system clock of 34.2 MHz and the data is processed at the rate of 4121 Mbits/sec. This is a fully pipeline architecture optimized for both time and space that performs at high speed and consumes less space.

Encryptor Core

It is a fully pipeline AES encryptor core. As it was already mentioned, the encryptor core implements the encryption path for AES encryptor/decryptor core explained in the last Section. The critical path for one encryption round is shown in Figure 9.33.

For BS step, pre-computed values of the S-Box are directly stored in the memories (BRAMs), therefore, AF transformation is embedded into BS. For

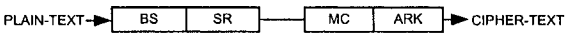

Fig. 9.33. The Data Path for Encryptor Core Implementation

the sake of symmetry, BS and SR steps are combined together. Similarly MC and ARK steps are merged to use 4-input/1-output CLB configuration which helps to decrement circuit time delays. The encryption process starts from the first clock cycle as the round-keys are generated in parallel as described in Section 9.5.2. Encrypted blocks appear at the output 11 clock cycles after, when the pipeline got filled. Once the pipeline is filled, the output is available at each consecutive clock cycle.

The encryptor core structure occupies 2136 CLB slices(22%), 100 BRAMs (35%) and 386 I/O blocks (95%) on targeting Xilinx VirtexE FPGA devices (XCV812BEG). It achieves a throughput of 5.2 Gbits/s at the rate of 40.575 MHz. A separated realization of this encryptor core provide a measure of timings for encryption process only. The results shows huge boost in throughput by implementing the encryptor core separately.

Decryptor Core

It is a fully pipeline decryptor core which implements the separate critical path for the AES encryptor/decryptor core explained before. The critical path for this decryptor core is taken from Figure 9.32 and then modified for IBS implementations. The resulting structure is shown in Figure 9.34.

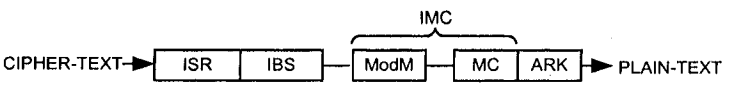

Fig. 9.34. The Data Path for Decryptor Core Implementation

The computations for IBS step are made by using look-up tables and precomputed values of inverse S-Box are directly stored into the memories (BRAMs). The IAF step is embedded into IBS step for symmetric reasons which is obtained by merely rewiring the register contains. The IMC step implementation is a major change in this design, which is implemented by performing a small modification ModM before MC step as discussed in Section 9.4.2. The MC and ARK steps are once again merged into a single module.

The decryption process requires 11 cycles to generate the entire round keys, then 11 cycles are consumed to fill up the pipeline. Once the pipeline is filled, decrypted plaintexts appear at the output after each consecutive clock cycle. This decryptor core achieves a throughput of 4.95 Gbits/s at the rate of 38.67 MHz by consuming 3216 CLB slices(34%), 100 BRAMs (35%) and 385

I/Os (95%). The implementation of decryptor core is made on Xilinx VirtexE FPGA devices (XCV812BEG).

A comparison between the encryptor and decryptor cores reveals that there is no big difference in the number of CLB slices occupied by these two designs. Moreover, the throughput achieved for both designs is quite similar. The decryptor core seems to be profited from the modified IMC transformation which resulted in a reduced data path. On the other hand, there is a significant performance difference between separated implementations of encryptor and decryptor cores against the combination of a single encryptor/decryptor implementation.

We conclude that separated cores for encryption and decryption provide another option to the end-user. He/she can either select a large FPGA device for combined implementation or prefer to use two small FPGA chips for separated implementations of encryptor and decryptor cores, which can accomplish higher gains in throughput.

Table 9.3. Specifications of AES FPGA implementations

	Core	Type	Device (XCV)	BRAMs	CLB(S) Slices	Throughput Mbits/s (T)	T/S
Sec. 9.5.4 [308]	E/D	P	2600E	80	6676	3840	0.58
Sec. 9.5.4 [308]	E/D	P	2600E		13416	3136	0.24
Sec. 9.5.5 [297]	E/D	P	2600E	100	5677	4121	1.73
Sec. 9.5.3 [311]	E	IL	812E		2744	258.5	0.09
Sec. 9.5.3 [311]	E	P	812E	100	2136	5193	2.43
Sec. 9.5.5 [307]	E	P	812E	100	2136	5193	2.43
Sec. 9.5.5 [306]	D	P	812E	100	3216	4949	1.54

9.5.6 Review of This Chapter Designs

The performance results obtained from the designs presented throughout this chapter are summarized in Table 9.3.

In Section 9.5.4 we presented two encryptor/decryptor cores. The first one utilized a Look-Up Table approach for performing the BS/IBS transformations. On the contrary, the second encryptor/decrpytor core computed the BS/IBS transformations based on an on-fly architecture scheme in $GF(2^4)$ and $GF(2^2)^2$ and does not occupy BRAMs. The penalty paid was on an increment in CLB slices.

The encryptor/decryptor core discussed in Section 9.5.5 exhibits a good performance which is obtained by reducing delay in the data paths for MC/IMC transformations, by using highly efficient memories BRAMs for BS/IBS computations, and by optimizing the circuit for long delays.

The encryptor core design of Section 9.5.3 was optimized for both area/time parameters and includes a complete set-up for encryption process. The user-

key is accepted and round-keys are subsequently generated. The results of each round are latched for next rounds and a final output appears at the output after 10 rounds. This increases the design complexity which causes a decrement in the throughput attained. However this design occupies 2744 CLB slices, which is acceptable for many applications.

Due to the optimization work for reducing design area, the fully pipeline architecture presented in Sections 9.5.3 and 9.5.5 consumes only 2136 CLB slices plus 100 BRAMs. The throughput obtained was of 5.2 Gbits/s. Finally, the decryptor core of (Sec. 9.5.5) achieves a throughput of 4.9 Gbits/s at the cost of 3216 CLB slices.

9.6 Performance

Since the selection of new advanced encryption standard was finalized on October, 2000, the literature is replete with reports of AES implementations on FPGAs. Three main features can be observed in most AES implementations on FPGAs.

1. **Algorithm's selection:** Not all reported AES architectures implement the whole process, i.e., encryption, decryption and key schedule algorithms. Most of them implement the encryption part only. The key schedule algorithm is often ignored as it is assumed that keys are stored in the internal memory of FPGAs or that they can be provided through an external interface. The FPGA's implementations at [102, 83, 63] are encryptor cores and the key schedule algorithm is only implemented in [63]. On the other hand the AES cores at [223, 366, 357] implement both encryption and decryption with key schedule algorithm.
2. **Design's strategy:** This is an important factor that is usually taken based on area/time tradeoffs. Several reported AES cores adopted various implementation's strategies. Some of them are iterative looping (IL) [102], sub-pipeline (SP) [83], one-round implementation [63]. Some fully pipeline (PP) architectures have been also reported in [223, 366, 357].
3. **Selection of FPGA:** The selection of FPGAs is another factor that influences the performance of AES cores. High performance FPGAs can be efficiently used to achieve high gains in throughput. Most of the reported AES cores utilized Virtex series devices (XCV812, XCV1000, XCV3200). Those are single chip FPGA implementations. Some AES cores achieved extremely high throughput but at the cost of multi-chip FPGA architectures [366, 357].

9.6.1 Other Designs

Comparing FPGA's implementations is not a simple task. It would be a fair comparison if all designs were tested under the same environment for all implementations. Ideally, performances of different encryptor cores should be

compared using the same FPGA, same design's strategies and same design specifications.

In this Section a summary of the most representative designs for AES in FPGAs is presented. We have grouped them into four categories: speed, compactness, efficiency, and other designs.

Table 9.4. AES Comparison: High Performance Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Good et al. [113]	E/D	P	XC3S2000-5	ECB	17425(0)	25107	1.44
Good et al. [113]	E/D	P	XCV2000e-8	ECB	16693(0)	23654	1.41
Zambreno et al. [400]	E	P	XC2V4000	ECB	16938(0)	23570	1.39
Saggese et al. [305]	E	P	XCVE2000-8	ECB	5819(100)	20,300	1.09
Standaert et al. [346]	E	P	VIRTEX3200E	ECB	15112(0)	18560	1.22
Jarvinen et al. [157]	E	P	XCV1000e-8	ECB	11719(0)	16500	1.40

*Throughput

In the first group, shown in Table 9.4, we present the fastest cores reported up to date. Throughput for those designs goes from 16.5 Gbps to 25.1 Gbits/s. To achieve such performances designers are forced to utilize pipelined architectures and, clearly, they need large amounts of hardware resources.

Up to this book's publication date, the fastest reported design achieved a throughput of 25.1 Gbits/s. It was reported in [113] and it applies a subpipelining strategy. The design divides BS transformation in four steps by using composite field computation. BS is expressed in computational form rather than as a look-up table. By expressing BS with composite field arithmetic, logic functions required to perform GF(2^8) arithmetic are expressed in several blocks of GF(2^4) arithmetic. That allows obtaining a sort of subpipelining architecture in which each single round is further unfolded into several stages with lower delays. This way, BS is divided into four subpipeline stages. As a result, there is a single stage in the first round, each middle round is composed of seven stages, while the final round, in which MC is not required, takes six stages. To keep balanced stages with similar delays, a pipeline architecture with a depth of 70 stages was developed. After 70 clock cycles once that the pipeline is full, each clock cycle delivers a ciphered block.

In the second group shown in Table 9.5 compact designs are shown. The bigger one in [297] takes 2744 slices without using BRAMs. The most compact design reported in [113] needs only 264 slices plus 2 BRAMS and it has a 2.2 Mbps throughput. In order to have a compact design it is necessary to have an iterative (loop) design. Since the main goal of these designs is to reduce hardware area, throughputs tend to be low. Thus, we can see that in general, the more compact a design is the lower its throughput.

Table 9.5. AES Comparison: Compact Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Good et al. [113]	E	IL	XCS2S15-6	ECB	264(2)	2.2	.008
Amphion CS5220 [7]	E	IL	XVE-8	ECB	421(4)	290	0.69
Weaver et al. [375]	E	IL	XVE600-8	ECB	460(10)	690	1.5
Chodowick et al. [52]	E	IL	XC2530-6	ECB	522(3)	166	0.74
Chodowick et al. [52]	E	IL	XC2530-5	ECB	522(3)	139	0.62
Rouvry et al. [302]	E	IL	XC3S50-4	ECB	1231(2)	87	0.07
Saqib [297]	E	IL	XCV812E	ECB	2744	258.5	0.09

*Throughput

Since BS is the most expensive transformation in terms of area, the idea of dividing computations in composite fields is further exploited in [113] to break 4-bit calculations into several 2-bit calculations. It is therefore a three stage strategy: mapping the elements to subfields, manipulation of the substituted value in the subfield and mapping of the elements back to the original field. Authors in [113] explored as many as 432 choices of representation both, in polynomial as well as normal basis representation of the field elements.

In the third group, a list of several designs is presented. We sorted the designs included according to the throughput over area ratio as is shown in Table 9.6[4]. That ratio provides a measure of efficiency of how much hardware area is occupied to achieve speed gains. In this group we can find iterative as well as pipelined designs. Among all designs considered, the design in [297] only included the encryption phase and the most efficient design in [223] reporting a throughput of 6.9 Gbps by occupying some 2222 CLB slices plus 100 BRAMs for BS transformation. We stress that we have ignored the usage of BRAMs in our estimations. If BRAMs are taken into consideration, then the design in [346] is clearly more efficient than the one in [223].

The designs in the first three categories implement ECB mode only. The fourth one, which is the shortest, reports designs with CTR and CBC feedback modes as shown in Table 9.7. Let us recall that a feedback mode requires an iterative architecture. The design reported in [214] has a good throughput/area tradeoff, since it takes only 731 slices plus 53 BRAMs, achieving a throughput of 1.06 Gbps.

As we have seen, most authors have focused on encryptor cores, implementing ECB mode only. There are few encryptor/decryptor designs reported. However, from the first three categories considered, we classified AES cores according to three different design criteria: a high throughput design, a compact design or an efficient design.

[4] In this figure of merit, we did not take into account the usage of specialized FPGA functionality, such as BRAMs.

Table 9.6. AES Comparison: Efficient Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
McLoone et al. [223]	E	P	XCV812E	ECB	2222(100)	6956	3.10
Standaert et al. [346]	E	P	VIRTEX2300E	ECB	542(10)	1450	2.60
Saqib et al. [307]	E	P	XCV812E	ECB	2136(100)	5193	2.43
Saggese et al. [305]	E	IL	XCVE2000-8	ECB	446(10)	1000	2.30
Amphion CS5230 [7]	E	P	XVE-8	ECB	573(10)	1060	1.90
Rodríguez et al. [297]	E/D	P	XCV2600E	ECB	5677(100)	4121	1.73
López et al [214]	E	IL	Spartan 3 3s4000	ECB	633(53)	1067	1.68
Segredo et al. [325]	E	IL	XCV600E-8	ECB	496(10)	743	1.49
Segredo et al. [325]	E	IL	XCV-100-4	ECB	496(10)	417	0.84
Calder et al. [41]	E	IL	Altera EPF10K	ECB	1584	637.24	0.40
Labbé et al. [193]	E	IL	XCV1000-4	ECB	2151(4)	390	0.18
Gaj et al. [102]	E	IL	XCV1000	ECB	2902	331.5	0.11

*Throughput

Table 9.7. AES Comparison: Designs with Other Modes of Operation

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Fu et al [100]	E	IL	XCV2V1000	CTR	2415 (NA)	1490	0.68
Charot et al. [49]	E	IL	Altera APEX	CTR	N/A	512	N/A
López et al [214]	E	IL	Spartan 3 3s4000	CBC	1031(53)	1067	1.03
López et al [214]	E	IL	Spartan 3 3s4000	CTR	731(53)	1067	1.45
Bae et al [15]	E	IL	Altera Stratix	CCM	5605(LC)	285	NA

*Throughput

After having analyzed the designs included in this Section, we conclude that there is still room for further improvements in designing AES cores for the feedback modes.

9.7 Conclusions

A variety of different encryptor, decryptor and encryptor/decryptor AES cores were presented in this Chapter. The encryptor cores were implemented both in iterative and pipeline modes. Some useful techniques were presented for the implementations of encryptor/decryptor cores, including: composite field approach for BS/IBS, look-up table method for BS/IBS, and modified MC/IMC approach.

All the architectures described produce optimized AES designs with different time and area tradeoffs. Three main factors were taking into account for implementing diverse AES cores.

- High performance: High performances can be obtained through the efficient usage of fast FPGA's resources. Similarly, efficient algorithmic techniques enhance design performance.
- Low cost solution: It refers to iterative architectures which occupy less hardware area at the cost of speed. Such architectures accommodate in smaller areas and consequently in cheaper FPGA devices.
- Portable architecture: A portable architecture can be migrated to most FPGA devices by introducing minor modifications in the design. It provides an option to the end-user to choose FPGA of his own choice. Portability can be achieved when a design is implemented by using the standard resources available in FPGA devices, i.e., the FPGA CLB fabric. A general methodology for achieving a portable architecture, in some cases, implies lesser performance in time.

For AES encryptor cores, both iterative and fully pipeline architectures were implemented. The AES encryptor/decryptor cores accomplished the BS/IBS implementation using two techniques: look-up table method and; composite fields. The latter is a portable and low cost solution.

The AES encryptor/decryptor core based on the modified MC/IMC is a good example of how to achieve high performance by using both efficient design and algorithmic techniques. It is a single-chip FPGA implementation that exhibits high performance with relatively low area consumption.

In short, time/area tradeoffs are always present, however by using efficient techniques at both, design and algorithm level, the always present compromise between area and time can be significantly optimized.