A Universal and Efficient AES Co-processor for Field Programmable Logic Arrays*

Norbert Pramstaller and Johannes Wolkerstorfer

Institute for Applied Information Processing and Communciations (IAIK), TU Graz,
Inffeldgasse 16a, A–8010 Graz, Austria
{Norbert.Pramstaller, Johannes.Wolkerstorfer}@iaik.at

Abstract. In this article we present a compact and efficient co-processor that calculates the Advanced Encryption Standard (AES). It implements the whole functionality of the AES algorithm: all key lengths (128-bit, 192-bit, and 256-bit) are supported for both, encryption and decryption. Furthermore, it supports the Cipher Block Chaining mode. Due to an innovative AES State representation the complete AES co-processor is well suited for low-end FPGAs. The integrated AMBA interface facilitates the integration of the co-processor in System-on-Chip designs too. An implementation on a Xilinx Virtex-E FPGA device uses only 1,125 CLB slices and no block RAMs. Our FPGA implementation reaches a throughput of 215 Mbps at a clock frequency of 161.0 MHz.

1 Introduction

The National Institute of Standards and Technology (NIST) selected the Rijndael algorithm among several other algorithms as the Advanced Encryption Standard (AES) in October 2000. In winter 2001, the AES algorithm became the Federal Information Processing Standard FIPS-197 [1].

Due to the increasing importance of reconfigurable devices, numerous FPGA AES implementations have been published within the last years. These implementations mainly focus on high throughput rates [4,5]. By using techniques like loop unrolling and pipelining, they are able to report throughput rates up to 12,160 Mbps [4]. Applying such techniques leads to AES hardware implementations that require a huge amount of FPGA resources that are only available for expensive devices and can only be used for high-end applications. Considering low-end applications, high throughput rates are not always required (e.g. wireless communications) and high-end FPGAs are too expensive.

In this article we present a new AES architecture that is supported by most of the FPGA product-families and can be implemented using inexpensive low-end FPGAs. It is the first known AES FPGA implementation that does not require on-chip block RAMs. Besides supporting the complete AES standard, it features the Cipher Block Chaining mode (CBC). The design relies on an

* The work described in this paper has been supported [in part] by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

J. Becker, M. Platzner, S. Vernalde (Eds.): FPL 2004, LNCS 3203, pp. 565–574, 2004.

unconventional but effective hardware architecture that was conceived to map efficiently on reconfigurable hardware like FPGAs from Xilinx [12]. An innovative AES State representation and highly optimized VHDL code—without target specific extensions—helped to obtain a small circuit with a high maximum clock frequency.

The remainder of this article is structured as following: a short description of the AES algorithm is given in Section 2 and Section 3 presents the proposed architecture and discusses design considerations. Section 4 introduces the basic features of FPGAs we exploited for our implementation. Finally, we present results in Section 5 and conclusions are drawn in Section 6.

2 The AES Algorithm

The AES algorithm is a symmetric block cipher that encrypts 128-bit plaintext data with a 128-bit, 192-bit, or 256-bit cipher key [1]. As other symmetric ciphers, AES applies a so-called round function iteratively to the plaintext to compute the ciphertext. The number of iterations (Nr) depends on the key-length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. The round function consists of the transformations *SubBytes*,*ShiftRows*, *MixColumns*, and *AddRoundKey*. An extra round key for each round is used for the key-dependent transformation *AddRoundKey*. The round keys are derived from the cipher key with the key-expansion function. Figure 1 depicts the AES dataflow, the round-function transformations for encryption, and reveals that the 128-bit State is organized as a 4×4 matrix of bytes.

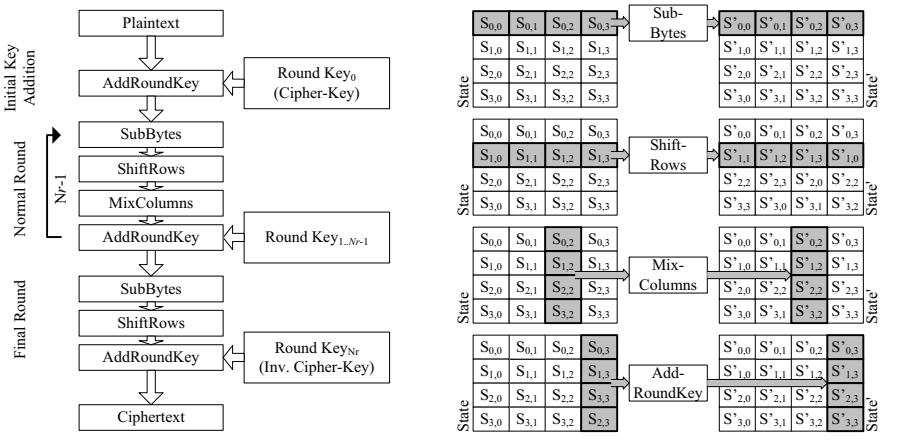

Fig. 1. AES dataflow (left) and round-function transformations (rigth) for encryption

The plaintext input to the AES algorithm becomes the initial State. During the initial key addition the plainttext is added with the cipher key. This is

followed by iteratively applying the round function to the State (normal round). The last step of the AES algorithm it the final round that differs slightly from the normal round by omitting the *MixColumns* transformation. After transforming the State during the final round, the State holds the according ciphertext that is the output of the algorithm.

All intermediate 128-bit results are called State too. Each transformation of the round function transforms the 128-bit State into a modified 128-bit State. *SubBytes*, the only non-linear operation of AES, is a multiplicative inversion in $GF(2^8)$ followed by an affine transformation. This function is applied to each byte o the State individually. *ShiftRows* rotates each row of the State by an offset equal to the row index, and *MixColumns* is a constant coefficient multiplication of each column with coefficients that are elements of $GF(2^8)$. Finally, the *AddRoundKey* transformation is the bit-by-bit addition of data and round key. This addition corresponds to an XOR-operation.

Each iteration of the round function requires a 128-bit round key for the *AddRoundKey* transformation. The round key is derived from the cipher key by applying the key-expansion function [1]. This function is based on the *SubBytes* transformation and simple XOR-operations. Obtaining the initial round key requires no transformations: the first 128 bits of the cipher key are used for the initial key addition. All subsequent round keys are iteratively derived from its predecessor.

Decryption is done by inverting the process of encryption: the round iterations are executed in the reverse order. This requires to generate round keys in reverse order too. Even the sequence of the round functions (*SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*) is reversed and their inverse functions are applied: *AddRoundKey*, *InvMixColumns*, *InvShiftRows*, and *InvSubBytes*. *AddRoundKey* requires no extra inverse function because the XOR-function is its own inverse.

Several modes of operation are defined for symmetric block ciphers [2]. Two common modes are the Electronic Codebook mode (ECB) and the Cipher Block Chaining mode (CBC). ECB is the simplest mode. It applies the cipher function to the plaintext blocks individually. This mode is not recommended to encrypt large quantities of data because repeated input blocks will produce the same output for a given key. The CBC mode alters each input block by combining the result of the previous cipher block with the current input block. This prevents repeated blocks to produce the same output.

3 Architecture of the AES Co-processor

This section describes the architecture of the AES co-processor. Starting with a swift overview, we will present details to highlight some innovative improvements that make it possible to come up with an efficient AES FPGA implementation.

3.1 Related Work

Gaj et al. [4] published the fastest known FPGA implementation. For encryption and decryption with 128-bit keys, a throughput of 12,160 Mbps on a Xilinx Virtex XCV1000BG560-6 device is reported. McLoone et al. [5] achieve a throughput of 6,956 Mbps for 128-bit keys only. They also presented encryption engines for 192-bit or 256-bit keys with accordingly lower throughput. Their combined encryption and decryption implementation can handle 128-bit keys and achieves a throughput of 3,239 Mbps on a Xilinx Virtex-E XCV3200E-8CG1156 device. The third implementation published by Dandalis et al. [7] also provides encryption and decryption for 128-bit keys. They achieve a throughput of 353 Mbps on a Xilinx Virtex XCV1000BG560-6 device. Fischer et al. [6] published a non-pipelined design supporting encryption and decryption for 128-bit keys. They report a throughput of 451 Mbps of their fast configuration and 115 Mbps for an economic configuration. A drawback of their design is the missing on-chip round-key generation. Chodowiec et al. [8] presented an implementation for low-end devices. Using only few resources they achieve a throughput from 139 Mbps up to 166 Mbps depending on the used FPGA device.

All implementations (except [6,8]) use a considerable amount of hardware resources. For instance, [5] requires 138 block RAMs for 256-bit keys. This demands the use of expensive million-gate FPGA devices.

As shown above, most published hardware implementations focus on high throughput rates and do not provide a non-parameterizable design to support the complete AES standard. Furthermore, the high throughput implementations [4,5] do not support the Cipher Block Chaining mode (CBC).

3.2 Architecture

Basic components of the AES co-processor, as shown in Fig. 2, are the AMBA APB interface [3], the data unit, the key unit, and the control unit. The key unit calculates the key-expansion function. All round keys are pre-calculated and stored in the key unit. Pre-calculated round keys allow fast encryption/decryption of different data blocks for the same cipher key because no additional key expansion is required. The data unit holds the State and performs all AES transformations: *AddRoundKey*, *(Inv)SubBytes*, *(Inv)ShiftRows* and *(Inv)MixColumns*. When encryption or decryption has completed, the ciphertext (plaintext in case of decryption, respect.) is stored in the data unit. The control unit receives commands from the AMBA interface and generates control signals for all other modules. In addition to control round-key calculation, encryption and decryption, it also sequences data loading and unloading.

The architecture is similar to the architecture presented in [9]. Differences are a modified State representation and a modified round-key calculation scheme. Due to a non-pipelined approach, the same performance for all modes of operations (ECB and CBC) is reached. Next, we describe the AES data unit, the AES State representation, and the key unit in detail.

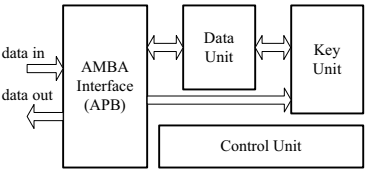

Fig. 2. Architecture of the AES co-processor

Data Unit. The data unit, schematically depicted in Fig. 3, stores the State, all intermediate results of the round function applied to the State and the output data when encryption or decryption has completed. The major difference to all other published AES implementations is the innovative State representation that consists of two States. One State contains the actual State values and the other State stores newly calculated values. Figure 3 depicts the two States, referred to as StateA and StateB. In each cycle, 32 bits (one row or one column) of either StateA or StateB are altered. Using a second State provides a lot of benefits without the need of additional recourses: *(Inv)ShiftRows* comes for free and no State transposition between column and row operations is required.

Storage elements in FPGAs can be efficiently implemented by using synchronous RAMs because the basic logic elements of FPGAs, called slices, can be configured as 16×1 bit synchronous RAM. Two slices provide 16×1 bit synchronous dual-port RAM functionality (see Section 4). Dual-port RAMs allow concurrent reading and writing to the RAM. Due to these technology features, the State-RAM as depicted in Fig. 3 is implemented as four slices of 8×8 bit synchronous dual-port RAMs to allow addressing the slices independently.

The data unit performs all transformations of the round function: *(Inv)ShiftRows*, *(Inv)SubBytes*, *(Inv)MixColumns* and *AddRoundKey*. *AddRoundKey* and *(Inv)MixColumns* are applied to the State column-by-column, whereas *(Inv)ShiftRows* and *(Inv)SubBytes* are applied to the State row-by-row. Due to the slice architecture of the RAM that holds the State, it is not possible to read/write from/to the RAM column-by-column. Hence, a transposition of the State is necessary if a row-oriented operation follows a column-oriented operation, or vice versa. Transposition would require a reorganization of the State before further operations can be performed. By using two States, transposition can be implemented by accordingly addressing the State-RAM. Furthermore, *(Inv)ShiftRows* can be combined with transposing the State. As a consequence of this, *(Inv)ShiftRows* and transposition come for free. In the sequel we describe the memory organization and State transposition for encryption. The same approach can easily be modified for decryption.

When a row-oriented operation follows a column-oriented operation (or vice versa), the State must be transposed. Combining row and column transformations minimizes the number of required transpositions: *ShiftRows* is combined with *SubBytes* and *AddRoundKey* is combined with *MixColumns* (see Fig. 3). This approach requires only one transposition per round. Encryption requires

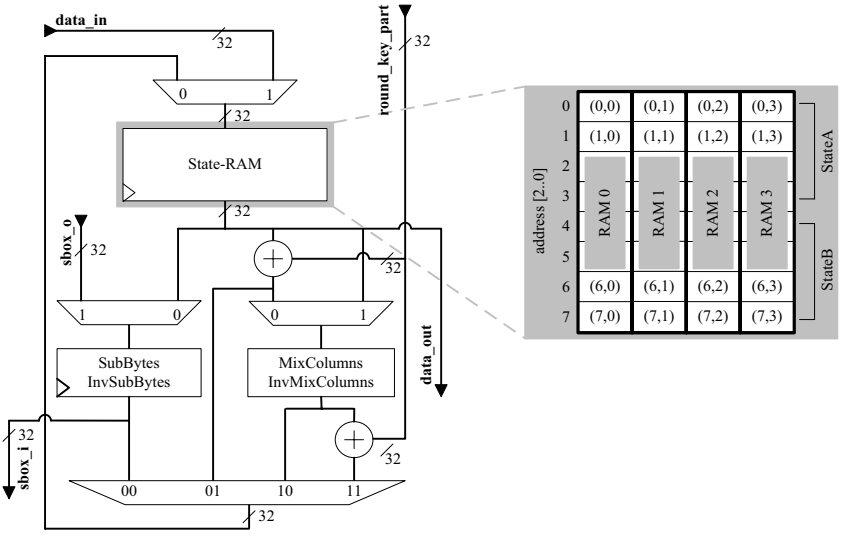

Fig. 3. Architecture of the data unit and State-RAM

SubBytes followed by *ShiftRows*. Since *ShiftRows* does not affect the byte values and *SubBytes* is applied to each byte of the State individually, the order of both operations does not matter. This fact eases the address generation for the State-RAM.

For explaining the State transposition we consider the State as 4×4 matrix: $\mathbf{S} = (s_{i,j})_{j=0..3}^{i=0..3}$. The *ShiftRows* transformation described in [1] can then be expressed as follows:

$$\mathbf{S'} = ShiftRows(\mathbf{S}) = (s_{i,j-i \bmod 4})_{j=0..3}^{i=0..3} \ . \tag{1}$$

If we replace the State by the transposed State, we obtain:

$$\mathbf{S'^T} = ShiftRows(\mathbf{S^T}) = (s_{i+j \bmod 4,j})_{j=0..3}^{i=0..3} \ . \tag{2}$$

With the result of (2) the addressing of the StateB-RAM can be determined: the indices (i, j) must be substituted with $(i + j \bmod 4, j)$. Due to the even number of AES rounds for all key lengths, *ShiftRows* is always applied to StateB only. Thus, the resulting index tuples can be directly mapped to the RAMs. The first part of the tuple index specifies the RAM slice and the second part specifies the RAM address. Since we operate on StateB, we must add an offset of 4 to the index value to get the correct address. Figure 4 shows the transposition of the State, including *ShiftRows* and *SubBytes* for encryption.

Implementation of *(Inv)SubBytes* and *(Inv)MixColumns*. The *(Inv)SubBytes* transformation is based on [10]. One difference is that the byte inversion in GF(2^8) is implemented by using a synchronous ROM.

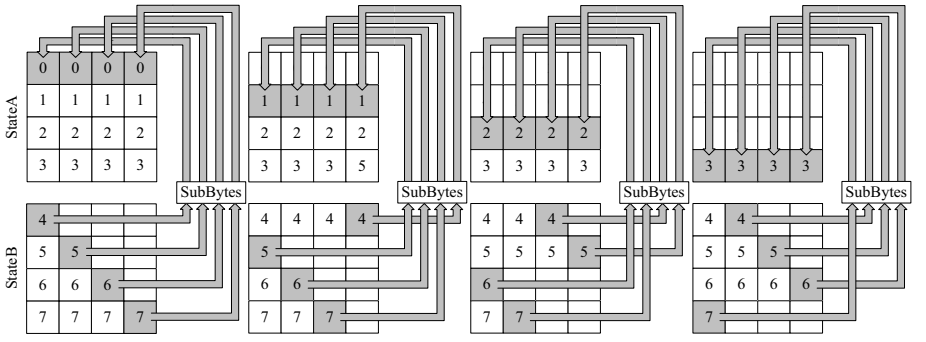

Fig. 4. ShiftRows and SubBytes for encryption

(Inv)MixColumns is similar to the architecture presented in [11]. For further details refer to [10,11].

Key Unit. The key unit holds the round keys and performs the key-expansion function. For each new cipher key, the round keys are pre-calculated to allow rapid encryption of subsequent data blocks for the same cipher key—no further key expansion has to be done. Because decryption uses the encryption round-keys in the reverse order, the key-expansion function must only be calculated once. Hence, the round keys stored in the key store are used for both, encryption and decryption.

The key-expansion function needs the *SubBytes* functionality. To keep the required hardware resources small, *SubBytes* is shared between key unit and data unit (multiplexor-input *sbox_o* in Fig. 3). This can be done easily because the four *SubBytes* units are not used by the data unit during the calculation of the round keys.

The memory of the key unit is separated from the memory of the data unit, because the access of a common memory would be a throughput bottleneck. The key store is implemented as a 64×32 bit synchronous single-port RAM.

An innovative aspect of our implementation is that the key unit can handle 128-bit, 192-bit and 256-bit keys with minimal additional hardware requirements. Supporting all key lengths increases the needed hardware resources for the key unit by only 7.8%. The size of the key memory for 256-bit keys is the same as for 128-bit keys. For 128-bit keys, the key-expansion function derives 44 32-bit round-key parts from the cipher key. This requires a 64×32 bit RAM. 256-bit keys produce 63 32-bit round-key parts fitting the 64×32 bit RAM.

4 Exploiting FPGA Features

The basic building blocks of Xilinx FPGAs are Configurable Logic Blocks (CLBs) [12]. CLBs are arranged in a rectangular matrix and are wired by programmable

interconnect. A CLB contains four logic cells (LUTs) that can be programmed to have different functionality: combinational logic (an arbitrary Boolean function of four inputs), logic and a register, or synchronous 16×1 bit RAM. Combining two logic blocks allows to implement a 16×1 bit dual-port RAM. Besides CLBs, Xilinx FPGAs offer block RAM that can store 4096 bits. Block RAM can be configured at ratios between 4096×1 and 256×16 and may have dual-port functionality. Block RAMs are also suitable for implementing synchronous ROMs.

When multiple, fast, and small RAMs are required, distributed (LUT-based) RAMs offer an ideal solution. The benefit is that the RAM cell is adjacent to the logic and thus, the wiring from the logic to the RAM is negligible. This improves the timing behavior. Multiple distributed RAMs can be merged to either enlarge the address space or the word width. Enlarging the word width is unproblematic (LUTs in parallel), but enlarging the address space can cause performance loss. For instance, a 32×1 bit RAM requires two 4-input LUTs whose outputs need to be multiplexed. This leads to a worse timing behavior and an increased amount of hardware resources. In such cases it makes sense to use block RAMs instead of using distributed RAMs.

When customizing hardware for FPGAs, it is in general more efficient to use RAM instead of registers for storage because the cost of RAM is relatively low in comparison to storing information in registers. For instance, the State-RAM (see Section 3.2) would require 31.2 times more hardware resources when implemented with registers. As stated in Section 3.2, the second State of the State-RAM comes for free. This is due to the fact that a RAM with a depth between 1 and 16 requires always one 4-input LUT. So, the second State causes no additional cost.

Using synchronous RAMs and ROMs provides more flexibilities for the implementation. Depending on the target technology and available on-chip resources, it can be chosen whether distributed RAM or block RAM should be used for implementing the storage elements.

ALTERA devices do not support distributed RAM but provide Embedded System Blocks (ESBs) that provide the same functionality as block RAMs in XILINX devices. Hence, our design is also suitable for ALTERA FPGAs.

5 Implementation Results and Comparisons

This section compares the proposed AES co-processor with the works referred to in Section 3.1. In order to provide comparable results, we implemented our co-processor on a Xilinx Virtex-E XCV1000EBG560-8 device.

The performance results given in Table 1 are for the ECB mode. Most of these implementations claiming high throughput rates will have similar performance figures when operating in CBC mode. The CBC mode is strictly recommended and commonly used for encrypting high-speed data streams (e.g. as it is used for encrypting data transfers over networks) and hence, the above-listed high throughput rates lose their significance.

Table 1. Hardware resources and throughput comparison

Work	**Device**	**#CLB-slices**	**#BRAM**	ECB mode **Throughput** [Mbps]
Gaj et al. [4]	Xilinx XCV1000	12,600	80	12,160
McLoone et al. [5] (I)	Xilinx XCV812E	2,222	100	6,956
McLoone et al. [5] (II)	Xilinx XCV3200E	2,577	112	5,800
McLoone et al. [5] (III)	Xilinx XCV3200E	2,995	138	5,000
McLoone et al. [5] (IV)	Xilinx XCV3200E	7,576	102	3,239
Dandalis et al. [7]	Xilinx XCV1000	5,673	?	353
Fischer et al. [6] (I)	FLEX 10KE200-1	2,530	24	451
Fischer et al. [6] (II)	ACEX 1K50-1	1,213	10	115
Chodowiec et al. [8]	Xilinx XC2S30-6	222	3	166
Our proposal	Xilinx XCV1000E	**1,125**	**0**	**215**

[5]: enc.: (I)AES-128, (II)AES-192, (III)AES-256, enc./dec.:(IV)AES-128
[6]: AES-128 enc./dec.: (I) fast configuration, (II) economic configuration

As shown in Table 1 our implementation is the only one that does not require any block RAMs and in contrast to most of the other implementations, it supports the complete AES standard. Furthermore, the presented AES co-processor supports the CBC mode and is equipped with a 32-bit AMBA APB interface that eases the integration with processors used in System-on-Chip designs [3]. If we do not consider the CBC mode and the AMBA bus interface, our approach is still comparable with the above-listed works but we would require less hardware resources (-26 %).

Our implementation utilizes 9.16% of the available logic cells on a Xilinx Virtex-E XCV1000EBG560-8 device. 90.8% of the logic resources and 100% of the on-chip BRAMs can be used by other circuits like a LEON2 or an ARM processor. For a stand-alone application a low-end FPGA (e.g. Xilinx SpartanII XC2S100-6) is sufficient for implementing the complete AES co-processor— the other approaches (except [8]) do not fit on a SpartanII device. The high throughput designs do not support this flexibility and require expensive million-gate FPGAs.

The maximum clock frequency on a XCV1000 FPGA is 161 MHz. At this frequency, a throughput of 215 Mbps for AES-128, 180 Mbps for AES-192, and 156 Mbps for AES-256 is achieved for both ECB mode and CBC mode.

6 Conclusion

In this article we presented a compact AES co-processor for low-end FPGA devices. It implements the whole functionality of AES. In addition to covering the complete AES standard it supports the Cipher Block Chaining mode (CBC). We have shown that due to an innovative State representation the co-processor is well suited for inexpensive low-end FPGAs—most of the competing approaches

require expensive multi-million gate FPGAs. An implementation on a Xilinx Virtex-E device uses only 1,125 CLB-slices and no block RAMs. Our FPGA implementation reaches a throughput of 215 Mbps at a clock frequency of 161 MHz for encryption and decryption. The AES co-processor has a convenient 32-bit interface that facilitates the integration in System-on-Chip designs too.

References

1. National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES)* Federal Information Processing Standards Publication 197 (FIPS PUB 197), Nov. 2001.
2. National Institute of Standards and Technology (NIST), *Recommendation for Block Cipher Modes of Operation – Methods and Techniques*, NIST Special Publication SP 800-38a, `http://csrc.nist.gov/publications/nistpubs/`, Dec. 2001.
3. ARM Limited, *AMBA 2.0 Specification*, `http://www.arm.com/armtech/`.
4. P. Chodowiec, P. Khuon, and K. Gaj, *Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining*, Proceedings of the Symposium on Field Programmable Gate Arrays – FPGA 2001, pp. 94–102, ACM Press, 2001.
5. M. McLoone and J. McCanny, *High Performance Single Chip FPGA Rijndael Algorithm Implementations*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2001, LNCS 2162, pp. 65–76, Springer Verlag, 2001.
6. V. Fischer and M. Drutarovský, *Two Methods of Rijndael Implementation in Reconfigurable Hardware*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2001, LNCS 2162, pp. 77–92, Springer Verlag, 2001.
7. A. Dandalis, V. Prasanna, J. Rolim, *A Comparative Study of Performance of AES Final Candidates Using FGPAs*, The Third Advanced Encryption Standard (AES) Candidate Conference, Available from `http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/aes3agenda.html`, 2000.
8. P. Chodowiec and K. Gaj, *Very Compact FPGA Implementation of the AES Algorithm*, Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2003, LNCS 2779, pp. 319–333, Springer Verlag, 2003.
9. S. Mangard, M. Aigner, and S. Dominikus, *A Highly Regular and Scalable AES Hardware Architecture*, IEEE Transactions on Computers, Vol. 52, No. 4, pp. 483–491, April 2003.
10. J. Wolkerstorfer, E. Oswald, and M. Lamberger, *An ASIC implementation of the AES SBoxes*, Proceedings of the Cryptographer's Track at the RSA Conference 2002, LNCS 2271, Springer Verlag, Feb. 2002.
11. J. Wolkerstorfer, *An ASIC implementation of the AES-MixColumn operation*, Proceedings of Austrochip 2001, pp. 129–132, Vienna, Austria, 12 October 2001.
12. Xilinx Incorporated, *Silicon Solutions — Virtex Series FPGAs*, `http://www.xilinx.com/products/`.