Realization of the Round 2 AES Candidates using Altera FPGA

Viktor Fischer
MICRONIC s. r. o., Dunajská 12, Košice, Slovakia
www.micronic.sk

Abstract
This paper presents an evaluation of five Round 2 Advanced Encryption Standard (AES) candidates from the viewpoint of their realization in a FPGA. After the analysis of the general characteristics of the algorithms a general cipher structure is defined. Using this structure, the suitability of available FPGA families is evaluated. Finally, three algorithms – RIJNDAEL [5], SERPENT [6] and TWOFISH [7] - are realized in VHDL and implemented in the selected FPGA family.

1. Introduction

One of restrictions given by the NIST on the AES candidates was the possibility of their hardware realization. Two conferences have been organized for AES candidates presentation and evaluation. But few assessments of hardware implementation of the proposed algorithms have been published up to now [2]. The final report of the first round [8] has explicitly asked designers for hardware implementation evaluation of all algorithms.

The aim of this paper is to evaluate AES candidates from the viewpoint of their hardware realization, using Field Programmable Gate Arrays (FPGA). In the first paragraph we shall analyze the algorithms regarding their limits in implementation in FPGAs. In the next chapter we shall define a basic structure of a generalized cipher and we shall specify common parameters of the cipher so that different algorithms could be easily compared. In the following paragraph we shall describe the structure of the algorithms selected for the implementation (RIJNDAEL and TWOFISH) and we shall briefly describe the implementation of the blocks and discuss different solutions from the viewpoint of the surface occupation and the speed. Finally, we shall present the results of VHDL implementation of these algorithms in Altera FPGA.

2. Analysis of the Round 2 AES candidates

In this chapter we shall analyze AES candidate algorithms regarding their suitability for implementation in FPGA. Special attention will be paid on:

a) the evaluation of the operations used for encryption and decryption,
b) the difference between encryption and decryption,
c) the possibility of on-the-fly key calculation and evaluation of the RAM capacity for storing the key in the FPGA,
d) the estimation of necessary resources like Logic Elements, RAM and ROM,
e) the estimation of the speed for different logic configurations.

Since number of rounds for some ciphers depends on the length of the key, in the next analysis we will suppose that all ciphers use 128-bit input block and 128-bit user key. This will also simplify the comparison of the algorithms. While some algorithms support on-the-fly key computing, others don't. To give the same starting conditions for all of them we presume, that round keys are pre-calculated and stored in the local memory.
In the next analysis we denote operations as follows:

$a + b$ integer addition modulo 2^{32}
$a - b$ integer subtraction modulo 2^{32}

a $\oplus$ *b*	bit-wise exclusive or
a $\times$ *b*	integer multiplication modulo 2^{32}
a <<< *b*	rotation of *a* by *b* position to the left
a >>> *b*	rotation of *a* by *b* position to the right
a << *b*	shift of *a* by *b* position to the left
a >> *b*	shift of *a* by *b* position to the right

2.1 MARS

MARS is a shared-key (symmetric) encryption algorithm [3] supporting 128-bit blocks and variable key size ranging from 128 to 1248 bits. Algorithm can be described as follows (for further details see [3]):

Encryption

Inputs/outputs: Data stored in four 32-bit I/O registers *D*[3], *D*[2], *D*[1], *D*[0]
32-bit round keys *K*[0], ... , *K*[39]

Algorithm: 1. *D*[0] = *D*[0] + *K*[0], ..., *D*[3] = *D*[3] + *K*[3];
2. Forward mixing – 8 rounds (substitutions, 32-bit additions, 32-bit XOR, rotations by 24 bits, word permutations);
3. FOR *i* = 0 TO 15 DO:
    3. (*out*1, *out*2, *out*3) = *E*(*D*[0], *K*[2i + 4], *K*[2i + 5]);
    4. *D*[0] = *D*[0] <<< 13; *D*[2] = *D*[2] + *out2*;
    5. IF (*i* <8) THEN (*D*[1] = *D*[1] + *out*1) ELSE (*D*[1] = *D*[1] $\oplus$ *out*3);
    6. IF (*i* <8) THEN (*D*[3] = *D*[3] $\oplus$ *out*3) ELSE (*D*[3] = *D*[3] + *out*1);
    7. *D*[3], *D*[2], *D*[1], *D*[0] $\leftarrow$ *D*[0], *D*[3], *D*[2], *D*[1];
6. Backward mixing – 8 rounds (substitutions, 32-bit subtractions, rotations by 24 bits, word permutations);
7. *D*[0] = *D*[0] + *K*[36], ..., *D*[3] = *D*[3] + *K*[39].

Where *E*(*in, key1, key2*) represents *E*-function realizing several operations: 32-bit addition of input data and *key1*, 32-bit multiplication modulo 2^{32} of rotated input data and *key2*, two data-dependent rotations, substitution using two 256-element S-boxes with 32-bit output, two 32-bit XOR operations and three fixed rotations. *out*1, *out*2 and *out*3 are 32-bit outputs of the *E*-function.

Decryption

Decryption process is the inverse of the encryption process. The code for decryption is similar, but not identical to the code for decryption (e. g. rotation direction is inverted, additions are replaced by subtractions, etc.).

Algorithm evaluation

a) From the analysis of the algorithm it follows that MARS cipher has a relatively complicated structure motivated by the robustness. It uses operations (multiplication modulo 2^{32} and data-dependent rotations) which are not easy to implement in an FPGA. Realization of S-boxes needs 16384 ROM bits. These parameters seem to limit the implementation of MARS cipher in FPGA because of extensive usage of resources.

b) Reversed order of subkeys during decryption can be classified as a very slight difference between encryption and decryption. Since decryption replaces in some cases the addition with the subtraction, some additional resources are needed to implement both encryption and decryption in the same circuit. Encryption and decryption use the same subkeys – no additional RAM space is needed.

c) Algorithm does not directly support on-the-fly subkey computation. It needs 40 32-bit subkeys, thus 1280-bit RAM organized preferably in 40 x 32 bits.

d) Although the cipher core uses a small amount of RAM, it requires relatively high capacity ROM to implement S-boxes. Implementation of addition, subtraction, fixed bit-wise rotation and block rotation (exchange) should not cause any problems. On the other hand, the use of 32-bit multiplication and also data-dependent rotations will necessitate the employment of huge logic blocks and the addition of clock cycles in all rounds. It seems, that realization of MARS cipher in FPGA having reasonable parameters will be very difficult, if possible.

2.2 RC6™

RC6™ is a block cipher using 128-bit input/output blocks, divided into four 32-bit words [4]. Although number of rounds can vary, we have chosen the version with 20 rounds, where [(20 x 2) + 4] round keys are added to 32-bit blocks and other operations are executed in the following scheme:

Encryption

Inputs/outputs: Plaintext and ciphertext stored in four 32-bit I/O registers A, B, C, D
32-bit round keys $S[0]$, ... , $S[43]$

Pseudo-code:
1. $B = B + S[0]$; $D = D + S[1]$;
2. FOR i = 1 TO 20 DO:
    3. $t = (B \times (2B + 1) <<< 5$; $u = (D \times (2D + 1) <<< 5$;
    4. $A = ((A \oplus t) <<< u) + S[2i]$; $C = ((C \oplus u) <<< t) + S[2i + 1]$;
    5. $(A, B, C, D) = (B, C, D, A)$;
6. $A = A + S[42]$; $C = C + S[43]$.

Decryption

Inputs/outputs: Plaintext and ciphertext stored in four 32-bit I/O registers A, B, C, D
32-bit round keys $S[0]$, ... , $S[43]$

Pseudo-code:
1. $C = C - S[43]$; $A = A - S[42]$;
2. FOR i = 20 DOWNTO 1 DO:
    3. $(A, B, C, D) = (D, A, B, C)$;
    4. $u = (D \times (2D + 1) <<< 5$; $t = (B \times (2B + 1) <<< 5$;
    5. $C = ((C - S[2i + 1]) >>> t) \oplus u$; $A = ((A - S[2i] >>> u) \oplus t$;
6. $D = D - S[1]$; $B = B - S[0]$.

Algorithm evaluation

a) From the analysis of the pseudo-code it follows that this algorithm uses some operations (fixed rotation, XOR) which are easy to implement. Since internal structure of most FPGAs is optimized for fast 32-bit addition and subtraction realization, these operations can be realized very efficiently. But FPGA structure is not well suited for 32-bit multiplication used in lines 4 and 5 for both encryption and decryption. Variable rotations used in lines 5 could cause another problem for cipher implementation. On the contrary, RC6™ does not use any S-boxes.

b) Reversed order of subkeys during decryption can be classified as a very slight difference between encryption and decryption. Since decryption uses subtraction rather than addition (lines 1 and 5), some additional resources are needed to implement both encryption and decryption in the same circuit. Encryption and decryption use the same subkeys – no additional RAM space is needed.

c) Algorithm does not directly support on-the-fly subkey computation. For 20 rounds it needs 44 32-bit subkeys resulting in 1408-bit ROM capacity.

d) From the above analysis it follows, that the cipher core uses relatively small amount of RAM and it does not need ROM for S-boxes. Addition, subtraction, bit-wise rotation and block exchange can be implemented in a very simple way. But the use of 32-bit multiplication and also data-dependent rotation

will necessitate the addition of important multilevel logic blocks. This will probably lead to poorer performance and worth surface usage of the cipher.
e) Since all operations of the round can't be realized in parallel in one clock period, the algorithm will be executed in multiple of 22 clock periods. The final number of clock periods depends on implementation of 32-bit multiplication and data-dependent rotation.

2.3 RIJNDAEL

RIJNDAEL is a block cipher using 128, 192 and 256-bit input/output blocks and keys [5]. The size of both can be chosen independently. As it is explained in the beginning of chapter 2, in the next analysis we use 128 bits for both I/O block and user key. Therefore the cipher in this configuration will operate in 10 rounds.

Encryption

Inputs/outputs: Plaintext and ciphertext stored in one 128-bit I/O register R
128-bit round keys K_0, ... , K_{10}

Pseudo-code:
1. $R = R \oplus K_0$;
2. FOR i = 1 TO 10 DO:
3. {
4. $R = S_8(R)$;
5. $R = P_8(R)$;
6. IF (i < 10) THEN $R = MC(R)$;
7. $R = R \oplus K_i$;
8. }

Decryption

Inputs/outputs: Ciphertext and plaintext stored in one 128-bit I/O register R
128-bit round keys K_0, ... , K_{10}

Pseudo-code:
1. $R = R \oplus K_{10}$;
2. FOR i = 9 TO 0 DO:
3. {
4. IF (i >0) THEN $R = IMC(R)$;
5. $R = IP_8(R)$;
6. $R = IS_8(R)$;
7. $R = R \oplus K_i$;
8. }.

Here $MC(\ )$ and $IMC(\ )$ denotes MixColumn function and its inverse, both realizing matrix multiplication on 32-bit blocks in GF(2^8), $P_8(R)$ and $IP_8(R)$ represents byte permutation and its inverse and $S_8(R)$ and $IS_8(R)$ denotes byte substitution and its inverse applied byte-wise on the whole 128-bit word.

Algorithm evaluation

a) RIJNDAEL has a relatively simple structure, while most of operations can be easily implemented in FPGA. Since matrix multiplication could cause problems for implementation of the algorithms on 8-bit processor, authors have proposed a *XTime*( ) function. This 8-bit function is applied byte-wise on 32-bit blocks. It can be easily implemented in FPGA. Implementation of $MC(\ )$, $IMC(\ )$, $P_8(R)$, $IP_8(R)$, $S_8(R)$ and $IS_8(R)$ is discussed more in details in section 3.3). Algorithm uses 2 types of fixed 8-bit S-boxes: one for encryption and another one for decryption.

b) There is a quite important difference between encryption and decryption: the order of the operations, but also their definition is changed: *MC*( ) is replaced by *IMC*( ), $P_8(R)$ by $IP_8(R)$ and $S_8(R)$ by $IS_8(R)$. Differences between these functions will be discussed in section 3.3. During decryption the subkeys are used in reverse order and furthermore the *IMC*( ) function has to be applied on keys K_1, ... , K_9.
c) The round keys can be calculated easily from the user key using operations as XOR and rotation on 32-bit data. So the key schedule computation is very fast. Decryption applies subkeys in reverse order and the *IMC*( ) function has to be applied on keys K_1, ... , K_9. Therefore decryption could be slower than encryption.
Encryption and decryption use 11 128-bit keys, so the RAM capacity should be al least 1408 bits (if we suppose, that *IMC*( ) function is calculated on-the-fly during decryption).
d) We can conclude that the cipher core should use relatively small amount of logic resources to realize rotations and XOR operations. *XTime* function will simplify necessary matrix multiplication. The cipher uses two 8-bit S-boxes that should be stored in ROM. The size of ROM memory depends on the number of bytes that should be substituted in one clock period. If the whole 128-bit word should be processed in one period, 16 identical S-boxes have to be used for encryption and 16 S-boxes for decryption. This requires the total ROM capacity of 65536 bits.
e) Since all operations of the round can be realized in parallel in one clock period, the algorithm could be executed in 11 clock periods. But this fast version of the cipher would hardly be realizable, due to a size limitation of ROM blocks in FPGA. For example, the algorithm using 8 S-boxes (half number of S-boxes) will be executed in half speed (22 clock periods).

2.4 SERPENT

SERPENT is a 32-round SP-network operating on four 32-bit words [6], thus giving a block size of 128 bits. It uses 33 128-bit subkeys obtained from a 256-bit user key. The user key can be shorter, but in that case it has to be padded with one "1" followed by a necessary amount of "0" to get a 256-bit key. 32 rounds of the cipher use 8 different S-Boxes, each of which maps four input bits to four output bits. Each S-box type is used in four rounds. The same type is used 32 times in parallel in one round.
The bit slice version of the algorithm can be described as follows:

Encryption

Input: Plaintext stored in four 32-bit input registers X_0, X_1, X_2, X_3
128-bit round keys K_0, ... , K_{32}

Output: Ciphertext stored in X_0, X_1, X_2, X_3

Pseudo-code:
1. $B_0 = X_0, X_1, X_2, X_3;$
2. FOR i = 0 TO 30 DO:
3. {
4. $X_0, X_1, X_2, X_3 = S_j (B_i \oplus K_i)$;
5. $X_0 = X_0 <<< 13$; $X_2 = X_2 <<< 3$;
6. $X_1 = X_1 \oplus X_0 \oplus X_2$; $X_3 = X_3 \oplus X_2 \oplus (X_0 << 3)$;
7. $X_1 = X_1 <<< 1$; $X_3 = X_3 <<< 7$;
8. $X_0 = X_0 \oplus X_1 \oplus X_3$; $X_2 = X_2 \oplus X_3 \oplus (X_1 << 7)$;
9. $X_0 = X_0 <<< 5$; $X_2 = X_2 <<< 22$;
10. $B_{i+1} = X_0, X_1, X_2, X_3$;
11. }
12. $X = S_7 (X \oplus K_{31}) \oplus K_{32}$;

Where
j – index of S-boxes, $j = i$ mod 8,
B_i - intermediate data,
$S_i (B_i \oplus K_i)$ - substitution of $(B_i \oplus K_i)$ using S-boxes.

Decryption
Decryption is the reverse order encryption using the inverse of the S-boxes, as well as the inverse linear transformation.

Algorithm evaluation
a) Looking at the pseudo-code we can find that this algorithm uses operations as exclusive or, rotations, shifts and substitutions. All of them can be implemented in FPGA very easily (see section 2.6). Algorithm uses 8 types of 4-bit S-boxes.
b) Reversed order of subkeys and S-boxes during decryption can be classified as a very slight difference between encryption and decryption. Inverse S-boxes and inverse linear transformation will need some more resources in a combinatorial part of the cells. Encryption and decryption use the same subkeys – no additional RAM space is needed.
c) Subkeys can be calculated using exclusive or, rotation and substitution operations. They can be calculated on the fly but only in one direction - decryption necessitates unwinding of the subkeys. Cipher requires 132 32-bit subkeys so the minimum RAM capacity is 4224 bits.
d) We can expect that the cipher will use a small amount of logic resources for realization of internal 128-bit register, logical operations and S-boxes. A minimum of 8 4-bit S-boxes is needed for low-cost implementation. High-speed design will use 8 groups of 4 identical S-boxes to substitute 128 bits in parallel. The realization of S-boxes will have dominant impact on the efficiency of the cipher.
e) Since all operations of the round can be realized in parallel in one clock period, the algorithm can theoretically be executed in 32 clock periods.

2.5 TWOFISH

TWOFISH is a 128-bit block cipher [7]. It can work with several key lengths – 128, 192, or 256 bits. It consists of 16 rounds based on modified Feistel structure. This modification concerns XOR operation and rotation by one bit applied on two output blocks of the round.

Encryption

Input: Plaintext split into four 32-bit words X_0, X_1, X_2, X_3
32-bit subkeys K_0, ... , K_{39}, S_0, S_1

Output: Ciphertext stored in X_0, X_1, X_2, X_3

Pseudo-code:
1. $X_0 = X_0 \oplus K_0$; $X_1 = X_1 \oplus K_1$; $X_2 = X_2 \oplus K_2$; $X_3 = X_3 \oplus K_3$;
2. FOR r = 0 TO 15 DO:
3. {
4. $G_0 = g(X_0)$; $G_1 = g(X_1 <<< 8)$;
5. $P_0 = G_0 + G_1$; $P_1 = P_0 + G_1$;
6. $F_0 = P_0 + K_{2r+8}$; $F_1 = P_1 + K_{2r+9}$;
7. $X_2 = ((F_0 \oplus X_2) >>> 1)$; $X_3 = (F_1 \oplus (X_3 <<< 1))$;
8. $X_0 \leftrightarrow X_2$; $X_1 \leftrightarrow X_3$;
11. }
12. $X_0 = X_2 \oplus K_4$; $X_1 = X_3 \oplus K_5$; $X_2 = X_0 \oplus K_6$; $X_3 = X_1 \oplus K_7$;

where $g(\ )$ represents a function using 4-bit S-boxes, subkeys S_0 and S_1 and a Maximum Distance Separable (MDS) matrix. It realizes key-dependent permutations and MDS matrix multiplication on 32-bit input values. Function $g(\ )$ is explained more in details in section 4.2.

Decryption

Decryption is very similar to encryption: it uses the same structure, but the subkeys are applied in reverse order. Also, line 7 should be replaced by the following code:

7. $X_2 = (F_0 \oplus (X_2 <<< 1)); X_3 = ((F_1 \oplus X_3) >>> 1);$

Algorithm evaluation

a) TWOFISH has a rather complicated structure, but it uses mostly operations that can be easily implemented in a FPGA. The biggest problem to face seems to be the realization of MDS matrix multiplication. Algorithm uses 8 types of fixed 4-bit S-boxes. They can be completed as a combinatorial function or as a lookup table.

b) Strong feature of this algorithm is that there is a very slight difference between encryption and decryption. During decryption the keys are applied in reverse order. Encryption and decryption use the same subkeys – no additional RAM space is needed.

c) There are two different sets of subkeys: *S* and *K*. Subkeys *S* are obtained as a result of multiplying a part of a user key with RS matrix. Subkeys *K* can be computed in a structure very similar to that used for encryption (*h*( ) function followed by PHT transform). Both sets of the keys can be calculated on the fly in random order. Cipher requires 42 32-bit subkeys, so the RAM capacity should be at least 1344 bits.

d) From the above analysis, it follows that the cipher should need a relatively small amount of logic resources for additions, rotations, XOR operations and key-dependent permutations. Potential problem lies in the realization of MDS matrix multiplication. To reduce the surface usage, the same block can be used to calculate *h*( ) function for two blocks (see line 4 of the algorithm) in two steps. Relatively small resources will be needed to realize 8 4-bit S-boxes.

e) Since all operations of the round can be realized in parallel in one clock period, the algorithm can theoretically be executed in 16 clock periods.

2.6 Classification of basic operations used by ciphers

In this section the ability to implement basic cipher operations in Altera FPGA will be discussed. The analysis has shown that all ciphers use mainly next operations:

- *Bit-wise addition modulo 2 (XOR)* – this operation is easily realizable in FPGA using input lookup table of Logic Element (LE). XOR operation with two to four inputs can be realized in each LE.
- *Fixed rotations and shifts* – also these operations can be easily implemented but in this case routing resources will be used: cell interconnections can be reordered in a very simple way to realize rotations or shifts in both directions.
- *S-boxes* – they can be implemented as a lookup table using internal memory or as a combinatorial function. 4-bit S-boxes can be implemented by the use of both methods depending on the in-circuit memory availability. 8-bit S-boxes should preferably be realized using LUT, because combinatorial function would occupy many resources.
- *Additions and subtractions modulo 2^{32} on 32-bit data* – although these operations are not so elementary as XOR, they still can be easily realized in FPGA. Fast carry chain interconnection signals are dedicated in Altera FPGA to easy this task in an efficient manner.
- *Matrix multiplication in GF(2^8)* – these operations used in both RIJNDAEL and TWOFISH ciphers constitute the main obstacle in realization of these ciphers in programmable devices. Authors of RIJNDAEL propose a method using so called *XTime* function [5] to solve this problem. This 8-bit function can be easily implemented in FPGA and the matrix multiplication represents XOR operations applied on the outputs of this function. Since square matrices in both ciphers contain constant elements (polynomials in $GF(2^8)$), it can be shown, that multiplication can be replaced by several XOR operations.
- *Data-dependent rotations* – they can't be realized by a simple interconnection of register cells as it is in the case of the fixed rotations. A state machine or a counter will be necessary to control number of cycles - several clock cycles will be needed to rotate data to the final position. This bit manipulation will decrease overall speed of the cipher and it will increase the possibility of timing attacks.

- *32-bit multiplication modulo 2^{32}* – these operations used in both MARS and RC6 ciphers represent the main restriction of realization of these ciphers in FPGA. Altera propose macrofunctions for 8 x 8-bit multiplication implementation. To realize 32 x 32-bit multiplication a multilevel multiplier has to be designed. Such a structure will occupy excessive resources. Multiple clock periods needed to obtain final result will probably slow down the cipher in a great extent.

2.7 Selection of the algorithms to be implemented

In this section we will give an overview of critical problems concerning implementation of all ciphers.

MARS – 32-bit multiplication and data-dependent rotation will occupy an important part of the design. These operations will also degrade the cipher performance. Large S-boxes will take up considerable resources, too.

RC6 – as in the case of MARS cipher, 32-bit multiplication and data-dependent rotation will engage relatively great part of the device. Also, the cipher performance will be degraded by these operations.

RIJNDAEL – while 8-bit S-boxes can't be realized as a combinatorial function, they have to be built as a lookup table. To speed up the cipher more S-boxes have to be employed. The total ROM memory capacity could limit the implementation of a fast algorithm in FPGA. Another limiting factor could be the difference between encryption and decryption: although algorithm is almost the same for both cipher modes, *MixComumn* function, S-boxes, subkeys and also byte-wise rotation direction are changed.

SERPENT – relatively high RAM capacity is necessary to store the subkeys. To speed up the cipher, great amount of S-boxes should be used. The way that these S-boxes will be realized will determine the performance of the circuit.

TWOFISH – its relatively complicated key schedule could cause some difficulties. But the encryption and decryption algorithm has no significant weakness from the point of view of the implementation in FPGA.

Overview of basic parameters of the Round 2 AES candidates is presented in Table 1.

Table 1 – Basic parameters of the Round 2 AES candidates

Parameter	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
Number of rounds	2+16+2	20	10	32	16
Operations used	Complex	Complex	Simple	Very simple	Simple
Number of subkeys	40	44	11	33	42
Size of subkeys	32 bits	32 bits	128 bits	128 bits	32 bits
Total RAM bits	1280	1408	1408	4224	1344
Number of S-Boxes	2	none	1	8	8
Size of S-Boxes	8192 bits (256 x 32)	-	2048 bits (256 x 8)	64 bits (16 x 4)	64 bits (16 x 4)
Total ROM bits	16384	-	2048	512*	512*

* If S-boxes are realized using lookup table (embedded memory).

Using previous analysis we have selected as the most suitable for hardware implementation RIJNDAEL, SERPENT and TWOFISH ciphers.

While MARS and RC6 seem to give acceptable results in software realization, their hardware implementation in FPGA could be less competitive because the use of 32-bit multiplication and data-

dependent rotations, causing additional needs of hardware resources. For these reasons and also for the lack of time we have decided to exclude these ciphers from our development effort up to now.

3. Implementation of selected ciphers

3.1 Implementation strategy

To obtain comparable results for different ciphers we have unified their configuration and we have defined implementation limits in the next manner:

- The size of the input/output block will be limited to 128-bits**.**
- User key is supposed to have 128-bits.
- Round keys are pre-calculated and stored in local memory (EAB). Selected FPGA family (Altera Flex 10KxxxE) contains memory blocks of 4096 bits, which will be in our case organized in 16 x 256 bits. Using two memory blocks we can save 256 32-bit subkeys. Note that SERPENT needs 132 32-bit words to store the subkeys.
- Three kinds of strategies will be applied on the designs: in the first one we will search a "fair" cipher configuration, where all ciphers will have the same configuration parameters, especially the width of processed data. In the second one we will look for the fastest configuration for each cipher and finally, in the last strategy we will search the minimal (the most economic) configuration.
- We do not evaluate the possibilities of employment of pipelining structures in the ciphers.
- Each cipher is interfaced with the host system via the same interface (see section 3.2).
- The use of S-boxes (the way of their implementation and the number of S-boxes) was motivated by the design strategies (fair configuration, fast configuration and minimal configuration).

3.2 Implementation of the external interface

The ciphers are interfaced with the host system by the way of the 32-bit interface. Two 128-bit registers used to store plaintext and ciphertext (input and output registers) are accessible via 32-bit data bus in a sequential manner. The encryption (decryption) starts automatically after reception of the forth 32-bit data. The cipher is managed using control register containing encryption/decryption flag, run flag and a reset bit. Data and control registers are accessible using /CS_DATA and /CS_CTRL signals. Read/write action is realized at the rising edge of signal /WR or /RD.

Key memory is organized in 256 32-bit words. The pre-calculated subkeys can be entered to the cipher via separated 32-bit local interface that can be connected for example to the local memory. New subkey is written to the internal memory at the rising edge of a KEY_STRB signal, when /WR_KEY is low.

The interface contains control unit and 128-bit input and output registers, but in does not include cipher control state machine. It occupies about 380 Logic Elements in Altera Flex 10K family.

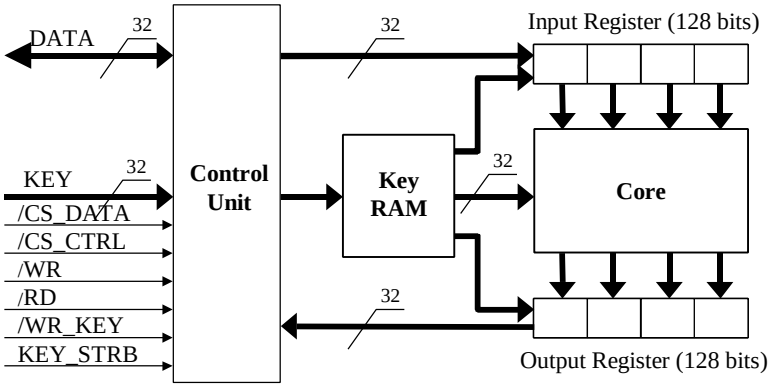

Figure 1 - Block diagram of the cipher

3.3 Implementation of the RIJNDAEL cipher

The encryption and decryption algorithm of the RIJNDAEL cipher is shown in Figure 2.

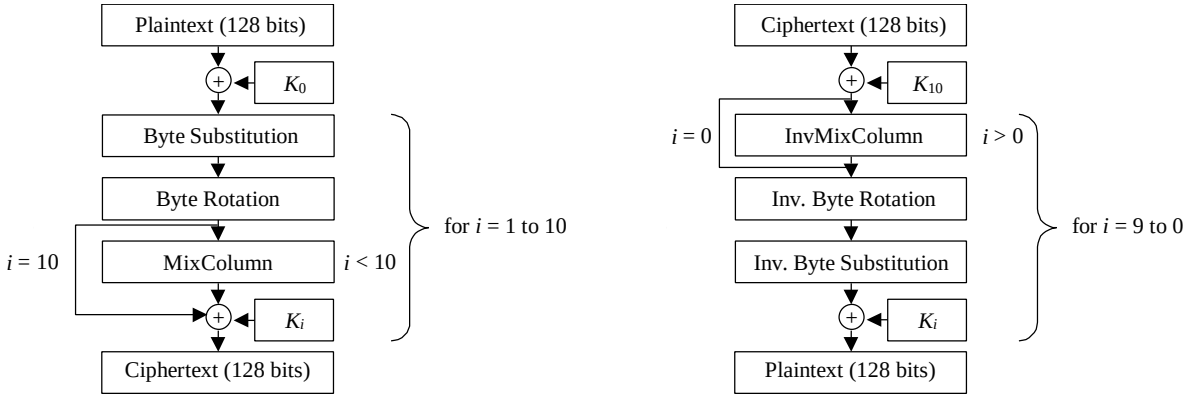

Figure 2 - Encryption (a) and decryption (b) algorithm

RIJNDAEL cipher is composed of four blocks: subkey addition modulo 2 (XOR), byte substitution (using two types of S-boxes: one for encryption and another one for decryption), *MixColumn* (*InvMixColumn* for decryption) function and byte rotation (exchange).
Byte substitution needs two types of S-boxes organized in 8 x 256 bits. Since Altera Flex 10KAE family contains RAM blocks with 4096 bits, we have chosen a configuration of 8 x 512 bits per block. That way in one block both encryption and decryption S-boxes can be saved.
MixColumn and *InvMixColumn* functions are applied on 32-bit words and they represent following matrix multiplication (encryption matrix, left and decryption matrix, right):

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \bullet \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} \qquad \begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \bullet \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

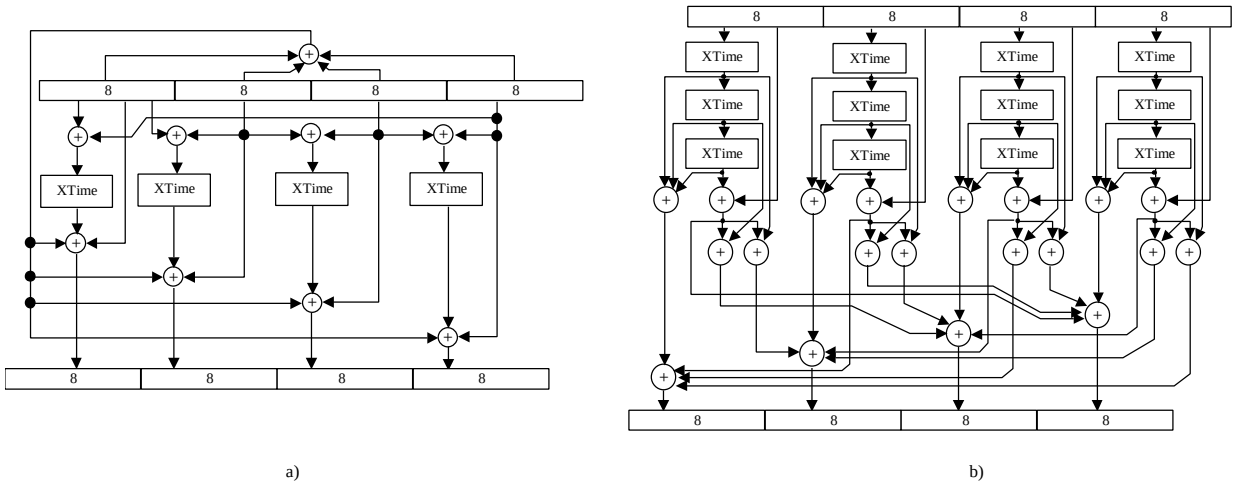

Figure 3 - Function *MixColumn( )* (a) and *InvMixColumn( )* (b)

There are two possibilities to implement matrix multiplication: the first method is based on the algorithm developed by authors of the cipher [5]. It was elaborated for software implementation of the matrix multiplication on 8-bit processors and it uses the *XTime*( ) function representing multiplication by two of one byte in GF(2^8). Final structures based on this algorithm representing matrix multiplication are presented in Figure 3 for both *MixColumn* (a) and *InvMixColumn* (b) functions. The *XTime*( ) function is presented in Figure 4.

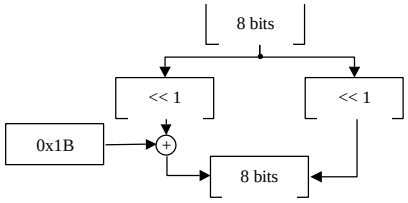

Figure 4. Function *XTime( )*

The second method consists in realization of matrix multiplication using the fact that the matrix is composed only of constants. Since one operand of the multiplication is constant, multiplication in GF(2^8) can be replaced by few additions modulo 2 that are simple to realize. For example, operation:

$$Y = X \bullet 0x03,$$

where X and Y are input and output 8-bit values and the symbol $\bullet$ represents the multiplication in GF(2^8) using the primitive polynomial $x^8 + x^4 + x^3 + x + 1$, can be implemented using following bit-wise additions modulo 2:

$y_7 = x_7 \oplus x_6$ $\quad$ $y_5 = x_5 \oplus x_4$ $\quad$ $y_3 = x_7 \oplus x_3 \oplus x_2$ $\quad$ $y_1 = x_7 \oplus x_1 \oplus x_0$

$y_6 = x_6 \oplus x_5$ $\quad$ $y_4 = x_7 \oplus x_4 \oplus x_3$ $\quad$ $y_2 = x_2 \oplus x_1$ $\quad$ $y_0 = x_7 \oplus x_0$

In that way matrix multiplication can be replaced by several additions modulo 2. We have implemented and compared both methods. Although both of them seem to be different, they describe in a different way the same combinatorial function. Therefore after the synthesis we have obtained almost the same results. The slight difference is probably caused by different minimization of the logic in two cases by the compiler.

Byte rotation (exchange) is specified in Table 2. It is very easy to implement (byte indexing in VHDL) and it uses only routing resources.

Table 2 - Byte rotation for encryption (*BR*) and decryption (*IBR*)

Orig.	***BR***	***IBR***	**Orig.**	***BR***	***IBR***	**Orig.**	***BR***	***IBR***	**Orig.**	***BR***	***IBR***
B_0	B_0	B_0	B_4	B_4	B_4	B_8	B_8	B_8	B_{12}	B_{12}	B_{12}
B_1	B_{13}	B_5	B_5	B_1	B_9	B_9	B_5	B_{13}	B_{13}	B_9	B_1
B_2	B_{10}	B_{10}	B_6	B_{14}	B_{14}	B_{10}	B_2	B_2	B_{14}	B_6	B_6
B_3	B_7	B_{15}	B_7	B_{11}	B_3	B_{11}	B_{15}	B_7	B_{15}	B_3	B_{11}

Fast configuration

The fast configuration of RIJNDAEL should use as much S-boxes as possible. To substitute 128 bits at once, 16 S-boxes organized in 8 x 512 bits will be needed. Since two S-boxes, one for encryption and one for decryption, occupy together one memory block (EAB), for the fast configuration we shall need 16 EAB to implement S-boxes. Subkeys are also stored in EAB, but memory blocks with subkeys are organized in 16 x 256 bits. To cover the whole 128-bit data word, 8 memory blocks are needed. So the total memory use for the fast configuration will be 24 EAB. Thanks to this encryption and decryption will be done in 10 clock periods.

Fair configuration

In the fair configuration the RIJNDAEL cipher should process the same amount of data in one round as for example TWOFISH. Since TWOFISH processes in one round two 32-bit data, in the fair configuration the RIJNDAEL cipher should deal with 64-bit data word, too. Therefore 8 memory blocks will be needed to implement S-boxes and 4 blocks to implement the subkey memory. Thus, 12 EAB will be used and 10 rounds will be executed in 20 clock periods in the fair configuration.

Minimum configuration

In the minimum configuration the RIJNDAEL cipher should use as few memory blocks as possible. Since the key memory block is always organized in 16 x 256 bits, the minimum reasonable data width will be 16 bits. In that case the cipher will need two EAB for S-boxes and one for subkeys, giving a total of 3 EAB. Because in the minimum configuration the data width is 16 bits, 8 clock periods will be necessary to execute one round and so the complete encryption/decryption process will take 80 clock periods.

The results of all configurations are presented in Table 4, 5 and 6.

3.4 Implementation of the SERPENT cipher

Algorithm of the SERPENT cipher is described in section 2.4. All the operations it uses are very simple to implement in FPGA. The key-mixing phase (addition modulo 2) in the beginning of each of 32 rounds is followed by the fixed rotations and two additions modulo 2 of three 32-bit blocks. Rotations and additions are repeated two times on different blocks. All these operations need a minimum amount of resources and they can be executed in one clock period. The only exception is the key mixing operation in the last round, where the second clock period will be needed for the key addition. So the final period count will be 33.

The algorithm uses 8 types of 4-bit S-boxes. Since it works with up to128-bit data, the way of how these S-boxes will be implemented will have a general influence on the cipher performance. As it was mentioned in section 2.6, 4-bit S-boxes can be realized as a lookup table using embedded memory blocks or they can be implemented as a combinatorial function. Four S-boxes can be realized in one EAB: two for encryption and two for decryption. For fast configuration, where 128 bits (32 nibbles) are substituted in parallel, 32 x 8 = 256 4-bit S-boxes will be needed. This number is doubled, if encryption and decryption have to be implemented in one circuit. So the fast configuration would need 128 memory block to implement S-boxes. Even for fair and minimum configuration the number of EAB to realize S-boxes would be too big: 64 and 16, respectively. For this reason we have decided to implement S-boxes as the combinatorial function.

Fast configuration

The fast configuration of SERPENT should use as much S-boxes as possible. As it was explained in the previous paragraph, S-boxes are realized as combinatorial functions. Each function needs a minimum of four logic elements. To implement 512 S-boxes at least 2048 logical elements will be needed. Subkeys are stored in EAB organized in 16 x 256 bits. To cover the whole 128-bit data word 8 EAB are used. Encryption and decryption will be made in 33 clock periods.

Fair configuration

In the fair configuration the SERPENT cipher should process 64-bit data word in one round. Therefore at least 1024 LE will be needed to implement S-boxes and 4 memory blocks to implement the subkey memory. Thus, 33 rounds will be executed in 66 clock periods in the fair configuration.

Minimum configuration

As it is explained for the RIJNDAEL cipher, in the minimum configuration the cipher data width is 16 bits. From this point of view SERPENT could use only one EAB for subkeys, but the memory block capacity is not high enough to store 4224 bits of round keys (see Table 1). Therefore in minimum configuration we use 2 EAB for subkeys and. 64 S-boxes necessary for this configuration will need at least 256 logic elements. Because in the minimum configuration the data width is 16 bits, 8 clock periods will be necessary to execute one round. The complete encryption/decryption process will be finished in 264 clock periods.

The results of all configurations of the SERPENT cipher are presented in Table 4, 5 and 6.

3.5 Implementation of the TWOFISH cipher

The algorithm of the TWOFISH cipher is described in section 2.5. A round function (see Figure 4.a) is realized using two $g(\ )$ functions and the Pseudo-Hadamard transform (PHT). The $g(\ )$ function involves two bit-wise additions modulo 2 with keys S_0 and S_1 to obtain key-dependent substitution. It also includes q_0 and q_1 permutation functions (see Figure 4.B) and a MDS function. Round function contains operations like fixed rotations, additions modulo 2 and additions modulo 2^{32} that are easy to implement. We shall now discuss the design of MDS matrix, q_0 and q_1 permutations and S-boxes.
MDS function represents following MDS matrix multiplication:

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix} \bullet \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

It seems to be difficult to implement, but it is shown in [9] that since MDS matrix contain only three types of constants, only few operations of multiplication in GF(2^8) have to be executed. While one operand of the multiplication is always constant, multiplication procedure can be replaced by several additions modulo 2, which are easy to implement. For example, operation:

$$Y = X \bullet 0x5B,$$

where X and Y are input and output 8-bit values and the symbol $\bullet$ represents the multiplication in GF(2^8) with the primitive polynomial $x^8 + x^6 + x^5 + x^3 + x + 1$, can be implemented using following bit-wise operations:

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

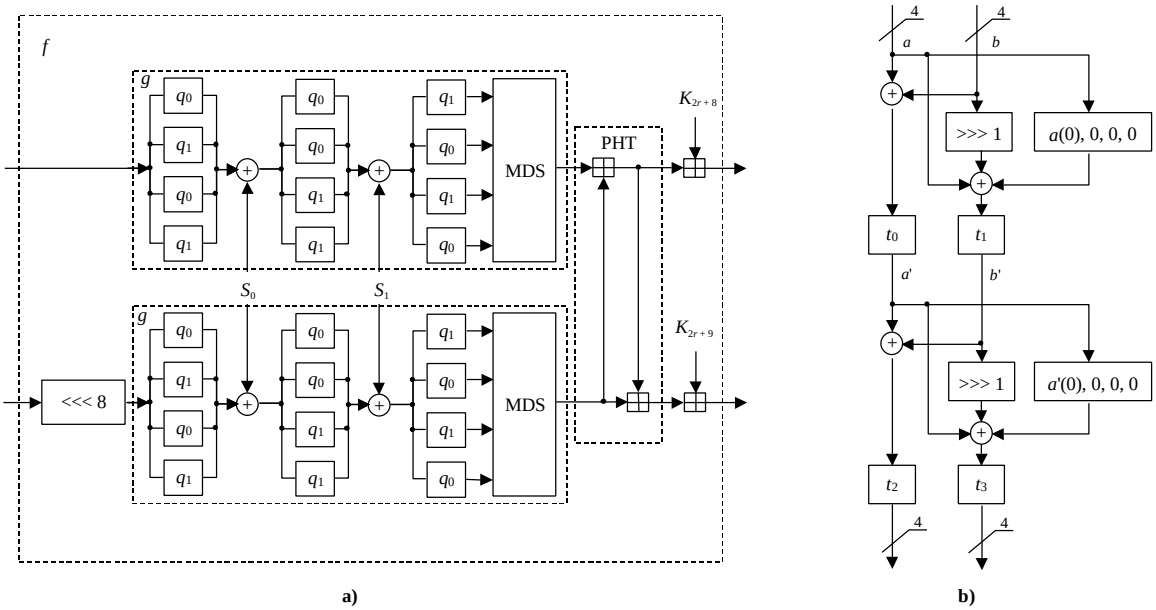

Figure 4. Single round *f*( ) function (a) and *q*( ) function

The multiplication Y = X • 0xEF can be replaced by following additions modulo 2:

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

Each permutation (q_0 and q_1) represents a fixed function that can be described by the structure shown in Figure 4.B. It is evident that permutation q is easy to implement.
S-boxes t_0 , t_1 , t_2 and t_3 map 4-bit input to 4-bit output and they are different for q_0 and q_1. We have realized each S-box as a combinatorial function. One S-box implementation needs 4 Logic Elements. To execute $g(\ )$ function in one clock period 4 x 4 x 3 = 48 S-boxes (192 Logic Elements) will be needed.

Fast configuration
Four 32-bit subkeys (two *K* keys and two *S* keys) are required in one round, 8 EAB should be used in the fast configuration. To speed up the design, we have used two $g(\ )$ functions as it is presented in Figure 4.a. Each of them uses 48 4-bit S-boxes. So the total number of S-boxes to be implemented is 96 (at least 768 logic elements). Encryption and decryption will be realized in 17 clock periods.

Fair configuration
The fair configuration of TWOFISH differs from the fast configuration in number of EAB – only 4 memory blocks are used, so the keys are red in two clock periods.

Minimum configuration
The minimum configuration includes only one $g(\ )$ function and data from upper and lower path are multiplexed to this block. We could also reduce the number of q blocks and so the number of S-boxes, but we think that this wouldn't save many resources (several additional multiplexers would be needed) and the control logic would be more complex.

The results of all configurations are presented in Table 4, 5 and 6.

4. Results of implementation using Altera FPGA

We have selected VHDL (Very High Speed Integrated Circuit Hardware Description Language) to synthesize the ciphers. The choice of VHDL should insure portability of the code to the devices of other vendors. Nevertheless, although most of the code written in VHDL is portable, up to now the description of embedded memories is vendor specific.
The devices have been synthesized using Altera MaxPlus2, version 9.3 development system. We have chosen ALTERA FLEX10KE family to realize the ciphers, because it contains large embedded memory blocks. To obtain comparable results, we have used the same circuit for all ciphers (FLEX10K130EQC240-1). The parameters obtained are presented in Tables 4, 5 and 6.

Table 4 - Fast configurations

Algorithm	EAB usage									Usage of Logic Elements			Speed (Mbits/s)		
	For subkeys			For S-boxes			Total								
	E	D	B	E	D	B	E	D	B	E	D	B	E	D	B
RIJNDAEL	8	8	8	16	16	16	24	24	24	1585	2145	3348	232.7	211.5	179.0
SERPENT	8	8	8	-	-	-	8	8	8	3678	3780	5816	125.5	119.0	111.4
TWOFISH	8	8	8	-	-	-	8	8	8	1950	1935	2104	81.5	81.5	80.3

Table 5 - Fair configurations (E = Encryption, D = Decryption, B = Both encryption and decryption)

Algorithm	EAB usage									Usage of Logic Elements			Speed (Mbits/s)		
	For subkeys			For S-boxes			Total								
	E	D	B	E	D	B	E	D	B	E	D	B	E	D	B
RIJNDAEL	4	4	4	8	8	8	12	12	12	1604	2098	3320	121.9	110.8	93.8
SERPENT	4	4	4	-	-	-	4	4	4	2238	2309	3270	52.5	52.5	52.5
TWOFISH	4	4	4	-	-	-	4	4	4	1870	1798	1915	73.7	73.7	72.6

Table 6 - Minimum configurations

Algorithm	EAB usage									Usage of Logic Elements			Speed (Mbits/s)		
	For subkeys			For S-boxes			Total								
	E	D	B	E	D	B	E	D	B	E	D	B	E	D	B
RIJNDAEL	1	1	1	2	2	2	3	3	3	1673	2033	3324	31.6	28.7	24.3
SERPENT	2	2	2	-	-	-	2	2	2	1385	1402	1579	13.4	13.4	13.4
TWOFISH	1	1	1	-	-	-	1	1	1	1318	1302	1409	26.7	26.7	25.3

5. Conclusions

In this paper we have evaluated AES candidates from the point of view of their hardware realization. After a brief analysis we have chosen three candidates for hardware implementation in the FPGA. While it is really difficult to compare cipher designs for efficiency, we have tried to realize similar structure for all selected algorithms in order to obtain comparable results. It is clear, that the results given in the previous paragraph are relative and that they depend significantly on the used technology. Nevertheless, designs presented in this paper represent hardware implementations of different AES candidates on the same platform. Subjective estimations on performance tradeoffs and on chip size of each author could be so evaluated in a more objective manner.

Even though the speed of ciphers implemented in FPGA is comparable with that attained with software implementation, the use of hardware for encryption and decryption can free up the CPU from a time-consuming task and increase overall system security. Additional logic can be put into the circuit to enlarge system performance.

References

[1] B. Schneier, *Applied Cryptography Second Edition*, John Wiley & Sons, 1996.

[2] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall and N. Ferguson, "Performance Comparison of the AES Submissions", *2nd AES conference*, Rome, Italy, March 1999.

[3] C. Burwick et al., "MARS - a candidate cipher for AES", *1st AES conference*, Ventura, CA, August 1998.

[4] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin "The RC6™ Block Cipher", *1st AES conference*, Ventura, CA, August 1998.

[5] J. Daemen, V. Rijmen, "AES Proposal: Rijndael", *1st AES conference*, Ventura, CA, August 1998.

[6] E. Biham, R. Anderson, L. Knudsen, "SERPENT, A Proposal for the Advanced Encryption Standard", *1st AES conference*, Ventura, CA, August 1998.

[7] B. Schneier et al., "TWOFISH: A 128-Bit Block Cipher", *1st AES conference*, Ventura, CA, August 1998.

[8] James Nechvatal et al. "Status Report on the First Round of the Development of the Advanced Encryption Standard", *NIST report*, 1999.

[9] P. Chodowiec, K. Gaj, "Implementation of the Twofish Cipher Using FPGA Devices", *Technical Report, George Mason University*, July 1999.