

AES as A Stream Cipher

Bin ZHOU, Kris Gaj, *Department of ECE, George Mason University*

Abstract—This paper presents implementation of advanced encryption standard (AES) as a stream cipher, using pure logic S-Box and compact architecture. Several different aspects are discussed, including on-the-fly key scheduling, compact AES round architecture, folded registers and so on. Three different Datapath widths are navigated and compared. The performances are tested with Xilinx Spartan 3 XC3S50 FPGA and also Xilinx Virtex 5 FPGA. The 32-bit compact design could fit within the smallest Xilinx Spartan 3 FPGA, including the on-the-fly 32-bit key scheduling. And the result shows that AES has a good potential to fit to stream cipher requirements.

Index Terms—AES, Stream Cipher, pure logic S-Box, compact AES round

I. INTRODUCTION

AES is an encryption standard by the U.S. government. It is well used world wide and has been intensively studied. It was announced by the National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) [3]. And it is also the most popular algorithm used in symmetric key cryptography. Usually, AES is considered as a block cipher, which operates on a fixed, larger number of bits to provide a bigger throughput and higher security. Meanwhile, stream cipher is another kind of symmetric key cipher, where cipher text is a function of the **current and all preceding blocks** of plaintext. Stream cipher is usually used in limited resource environment, such as cell phones, network stream media, wireless network and mobile devices and so on. It has a relatively smaller key size and could be implemented in a more efficient way. As a well-known alternative, by feeding back its key stream, block cipher could be adopted as a stream cipher. So in this paper, we use Counter Mode (CTR) AES to make it as a stream cipher.

As a main contribution to stream ciphers, in 2004, the ECRYPT stream cipher project (eSTREAM) [3] was launched to introduce new stream ciphers and introduce a wide range of systems. The candidates are: DECIM, Edon80, F-FCSR, Grain, MICHKEY, Moustique, Pomaranch and Trivium. In [2], they were compared with each other in terms of performance, throughput and area. Here we have a goal to implement the stream mode AES and compare with the above candidates, too.

And there is some work done on the AES as a stream cipher. In [1], the AES was implemented on a small FPGA using an application specific instruction processor; in [2] and [5], a compact architecture is introduced, using the data path widths

equal to 64-bit, 32-bit, and 8-bit.

II. AES STREAM CIPHER ARCHITECTURE

A. Standard Iterative AES Architecture

AES is a block cipher based on a round operation and combined with key scheduling functionality. Standard AES has a 128-bit block size and three variable key size (128-bit, 192-bit and 256 bit) with the round repeated 10, 12, 14 times, respectively. In this paper, we focus on 128-bit key size. The round is composed of four operations: ShiftRows, SubByte, MixColumn and AddRoundkey. The first round has only AddRoundkey and the last round has no MixColumn. Figure 1 shows the basic diagram of AES iterative operation.

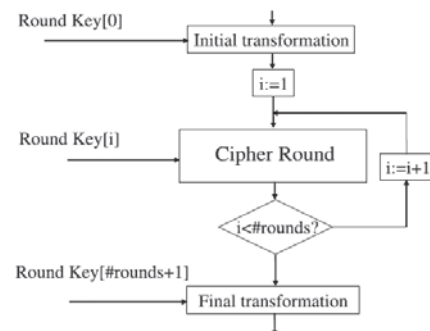


Fig. 1. Diagram of basic iterative AES [4]

And the cipher round of encryption is shown in Figure 2.

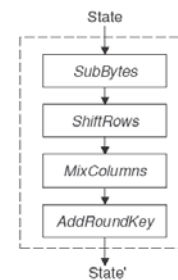


Fig. 2. Encryption round

B. Mode Selection

The AES Counter (CRT) mode, Output Feedback (OFB) mode and Cipher Feedback (CFB) mode are suitable for stream cipher. However, when using as a stream cipher, the resources are more restricted, which requires the working mode should be as small as possible and better pipelined to gain speedup.

First, we consider the same block for both encryption and

decryption. The electrical codebook mode requires separate encryption and decryption block (figure 3), which is not good. This will consume more resources for only one block. So here we eliminate ECB mode.

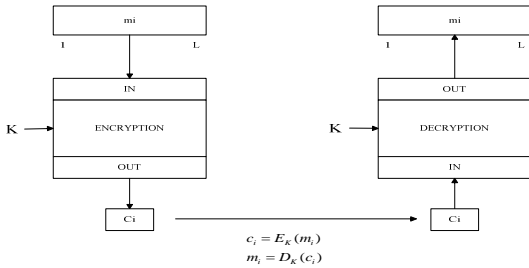


Fig 3 ECB mode, separate encryption and decryption

Then we consider the pipeline requirement for the stream cipher. It seems that intermediate stages could be added to CRT, OFB and CFB modes. The difference between CRT with the other two modes is that CRT has no feed back to the state part. As shown in Figure 4, it is easy to add pipeline stages to CRT mode. The next stage $IS_{i+1} = IS_i + 1$ is predictable and the pipeline could run without waiting for the intermediate states.

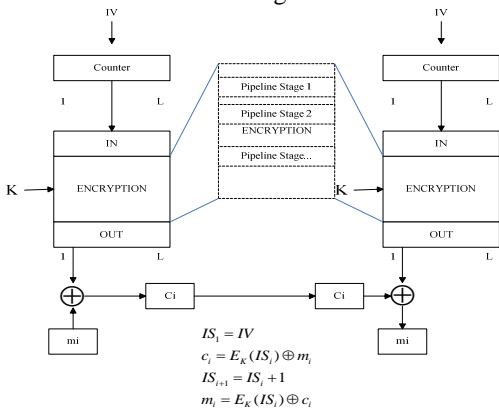


Fig 4 Adding Pipeline stages to CRT mode

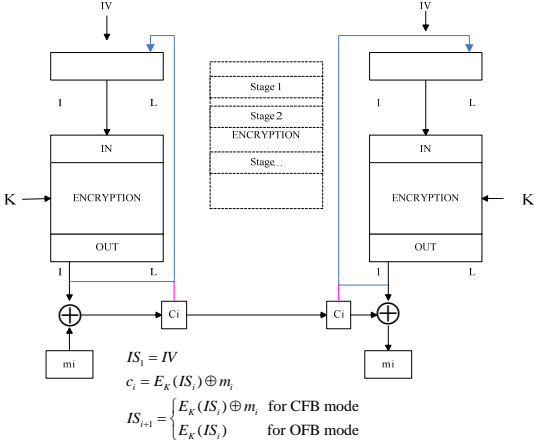


Fig 5 Adding intermediate stages to Feedback modes

However, things are different for OFB and CFB modes when adding stages. Figure 5 shows why this is not good for pipeline. When adding stages, the next state value IS_{i+1} will depend on the last stage value IS_i in both modes. So the next iteration

could not start until the last stage completes. This will slowdown the whole process rather than speed it up. So we choose CRT mode as our implementation subject

C. eSTREAM Cipher Specification

The interface for the whole module is shown in figure 5. The system first reset all the state and registers. Then when the key_iv_write is set, the outer circuit will input key and initial vector into the internal storage. After it's finished, the cipher will indicate this by inserting key_iv_ready. Then the user could first tell the encryption by setting data_in_write. The cipher will response data_in_ready when everything goes right. After that, the use could feed the data throught data_in. After the data is encrypted, the write signal will tell outer and d is the output ciphertext. For 128-bit block size AES, here d is 128-bit. And for other compact mode AES, the d is also 128-bit because AES require the whole block operation.

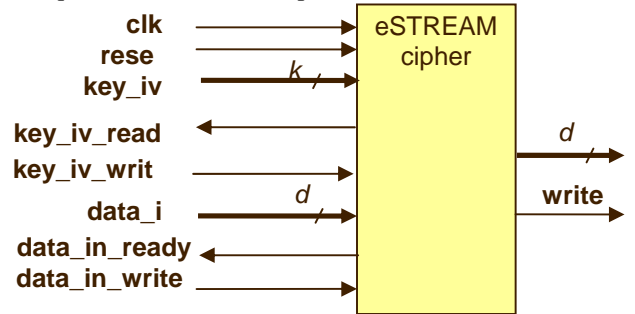


Fig 6 eSTREAM cipher interface

D. On-the-fly Key Scheduling

The on-the-fly key scheduling will generate the key with the round operation goes. Usually, the key is stored in a RAM and read out with the rounds. However, the on-the-fly key scheduling will save the resources. It also performs the same functionality with key scheduling, but will buffer at least 3 columns of key words to generate the next round keys. 32-bit on-the-fly key scheduling [4]

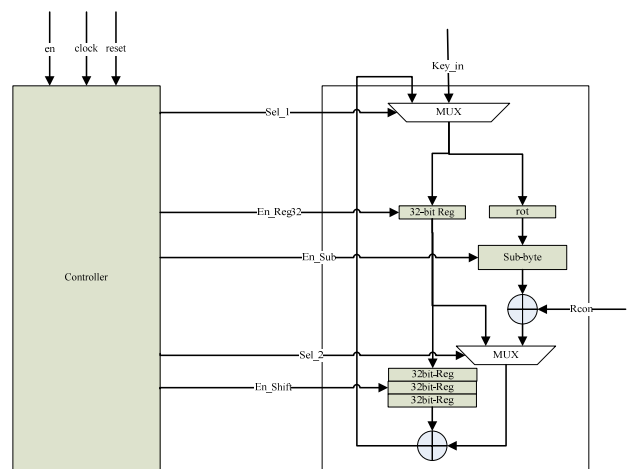


Fig 7 32-bit on-the-fly key scheduling

Figure 7 shows how the 32-bit key scheduling works. First, the initial keys are fed to a register and a shift register. Then when the fourth key word comes, it goes through the Rotation and SubByte operation, XOR with the first key word, generates the fifth key. After that, the following 3 key words are generated by XOR the other key words.

64-bit 3-in-1 on-the-fly key scheduling [4]

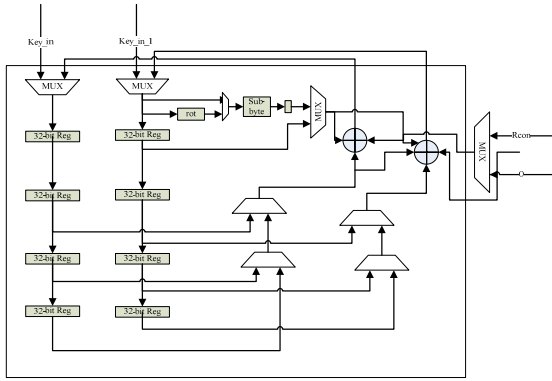


Fig 8 64-bit on-the-fly key scheduling

for $i \bmod Nk = 0$

$$k_i = k_{i-Nk} \oplus \text{Sub}(\text{Rot}(k_{i-1})) \oplus \text{Rcon}_{i/Nk}$$

$$k_{i+1} = k_{i+1-Nk} \oplus k_{i-Nk} \oplus \text{Sub}(\text{Rot}(k_{i-1})) \oplus \text{Rcon}_{i/Nk}$$

for $(Nk = 8)$ and $(i \bmod Nk) = 4$

$$k_i = k_{i-Nk} \oplus \text{Sub}(k_{i-1})$$

$$k_{i+1} = k_{i+1-Nk} \oplus k_{i-Nk} \oplus \text{Sub}(k_{i-1})$$

otherwise

$$k_i = k_{i-Nk} \oplus k_{i-1}$$

$$k_{i+1} = k_{i+1-Nk} \oplus k_{i-Nk} \oplus k_{i-1}$$

Fig 9 Operation of 3-in-1 64-bit key scheduling

Fig 8 shows a 3-in-1 64-bit on-the-fly key scheduling module. This could generate keys for 128, 192 and 265 bits key size. They use different numbers of registers for the buffering. Figure 9 shows the operations performed on different key position. Nk is the key round number. For 128-bit key, it's 4, and 6, 8 for 192-bit and 256-bit, respectively.

64-bit Simplified on-the-fly key scheduling

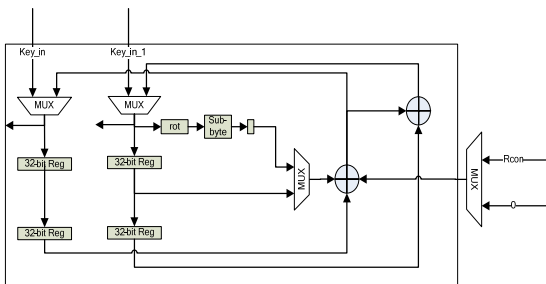


Fig 10 64-bit simplified on-the-fly key scheduling

However, the 128-bit key size on-the-fly scheduling could be simplified to Figure 10 by removing the unnecessary buffer registers and the MUX for the selection. Also this will only perform 128-bit key size scheduling, which is our focus in this paper.

E. Compact Architecture

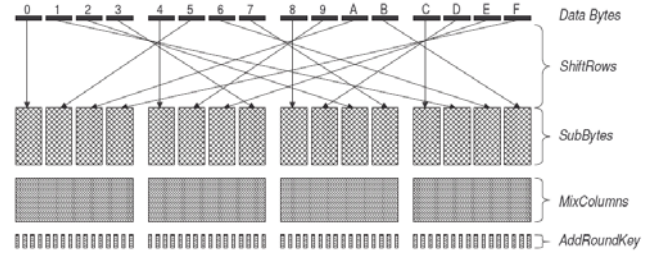


Fig 11 Basic AES operation round

From the basic round architecture, we can explore a great parallelism which shows the SubByte, MixColumn and AddRoundKey are separated within their own operands. Only the ShiftRows operation expands to all the bytes. The simple starter is to store all the intermediate results into a memory and then try to reuse the SubByte, MixColumns and AddRoundKey components.

32-bit Compact Architecture

The 32-bit compact architecture is a naturally folded one. Since the MixColumn will take 32-bit input and output, it's nice to feed it with a 32-bit word. The input and output memory should remain 128-bit to hold all the intermediate results. The ShiftRow operation is performed by reading the input from different input memory positions.

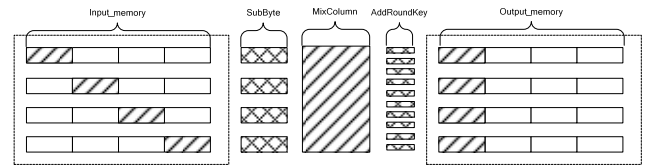


Fig 12 32-bit Compact round

The following steps will perform the whole round with 4 clock cycles [4].

1. Read input bytes: 0, 5, A, F; execute SubBytes, MixColumns and AddRoundKey; write results to the output at locations: 0, 1, 2, 3. This step is highlighted in the Fig. 10
2. Repeat above operations for input bytes: 4, 9, E, 3; write results at output locations: 4, 5, 6, 7.
3. Repeat above operations for bytes: 8, D, 2, 7; write results at locations: 8, 9, A, B.
4. Repeat above operations for bytes: C, 1, 6, B; write results at locations: C, D, E, F. Output now becomes input for the next step.

64-bit Compact Architecture

The 64-bit compact is quite similar to the 32-bit one. The input memory and output memory are revised to perform the correct addressing. And the round operation is shortened to 2 steps. Figure 13 shows how the first step operates on the input and output memory.

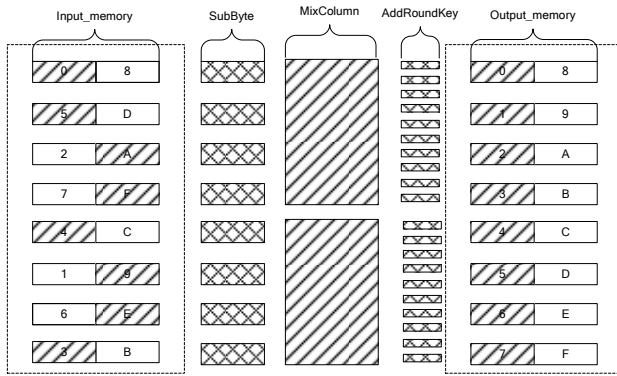


Fig 13 64-bit Compact Architecture

1 Read input bytes: 0, 5, A, F, 4, 9, E, 3; execute SubBytes, MixColumns and AddRoundKey on them; write results to the output at locations: 0, 1, 2, 3, 4, 5, 6, 7. This step is highlighted in the Fig. 12

2. Repeat above operations for bytes: 8, D, 2, 7, C, 1, 6, B; write results at locations: 8, 9, A, B, C, D, E, F. Output now becomes input for the next step.

8-bit Compact Architecture

The 8-bit compact architecture is much similar to the 32-bit. The MixColumn operation requires at least 32-bit word, so a buffer register is added before it. Since each clock cycle, only one byte operand is fed to the SubByte, there's only one SubByte module needed. As a result, one-byte adder could be shared within four bytes. accepts color graphics in the following formats: EPS, PS, TIFF, Word, PowerPoint, Excel, and PDF. The resolution of a RGB color TIFF file should be 400 dpi.

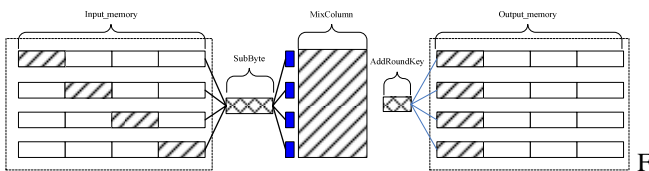


Fig 14 8-bit Compact Architecture

Shared Memory Compact Architecture

In above architectures, the input memory and output memory should be switched after one iteration, which will introduce more resource usage and more complexity in control logic. By observing that the memory position will be free after all the data are fed to the processing unit, we can reuse them. Figure 14 shows how to connect the output back to the input memory. By doing this, the output could share the input memory and save half of the total memory usage. This will add complexity to the control logic especially to the memory address part. The memory address transformation is shown in figure 15.

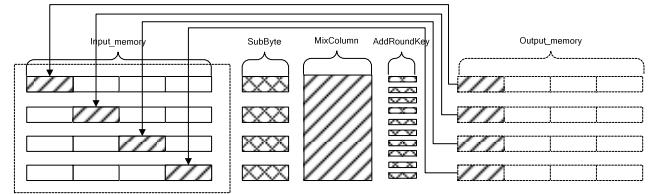


Fig 15 Shared Memory Compact Architecture

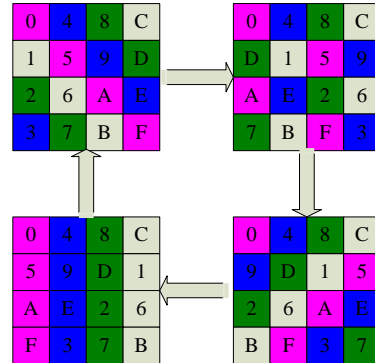


Fig 16 The Address transformation of shared memory

And this shows a very nice characteristic that the address could return to its original after 4 clock cycles, which is the time duration of one round. Although the address seems complex, it could also be resolved by split the memory into four small rows. Each row of the memory could use its own address control or even can be fit to a SRL to save resources for Xilinx FPGA. This memory schema is particularly fit to 8-bit compact architecture because only a byte was written each clock cycle and the address could be controlled more efficiently.

Resource Utilization Comparison of Compact Architectures

The above compact architectures have different resource usage. The smaller Datapath means smaller resources. However, smaller Datapath will require more clock cycles to process one round. The 64-bit could process one whole round in 2 clock cycles, while 8-bit will require 16 clock cycles. So it's a balance between resource cost and performance. In the next section, the performances will be tested and show how to choose from them.

TABLE I
RESOURCE UTILIZATION

Arch	MEMORY	SubByte	MixColumn	AddRoundKey
64-bit	256*8	8	2	64
32-bit	256*8	4	1	32
8-bit	256*8+32	1	1	8
32-bit (ShareMem)	128*8	4	1	32
64-bit (shareMem)	128*8	8	2	64
8-bit (shareMem)	128*8+32	1	1	8

F. Pure Logic S-box

S-box in AES is a diffusion transformation composed of a multiplicative inverse in GF(2⁸) and an affine transformation. It

has an equation as follows:

$$s' = MX \cdot (X^{-1} \cdot s)^{-1}$$

Here the MX is the affine transformation matrix and X^{-1} is the transformation matrix to $GF(2^8)$. All the multiplication is in $GF(2^8)$. Figure X shows the SubByte operation.

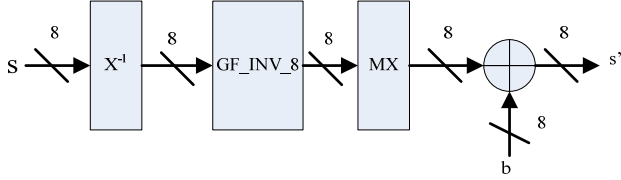


Fig 17 S-Box Operation for encryption

And the multiplication inverse in $GF(2^8)$ could be decomposed of operations into a sequence of operations in $GF(2^4)$ (including addition, multiplication, and inversion). And operations in $GF(2^4)$ can be expressed in terms of operations in $GF(2^2)$. In the end, operations in $GF(2^2)$ could easily be expressed by the operations in $GF(2)$, which only consists simple XOR gate (addition) and AND gate (multiplication). Inverse of 1 in $GF(2)$ is 1, and the inverse of 0 does not exist. Thus, the entire inversion in $GF(28)$ can be decomposed into a logic circuit composed of XOR and AND gates only. [4]

III. IMPLEMENTATION

The designs are following basic algorithm state machine (ASM) plus Datapath and controller scheme. First, a functional simulation model was build and the behavior is verified by ModelSim again the test vectors. Then, the design was implemented in a synthesizable VHDL and Xilinx XST was used to synthesis the whole design. The tools used are shown in table II.

TABLE II
IMPLEMENTATION TOOLS

Design Step	TOOL
VHDL Simulation	ModelSim SE 6.2b
FPGA Synthesis	Xilinx XST
FPGA Implementation	Xilinx ISE 9.1
Target FPGA	Xilinx Spartan 3
Alternative Target	Xilinx Virtex 5

IV. RESTULS

A. Results Matrix For Spartan 3

TABLE III
RESULTS FOR SPARTAN3

S-Box	Data path	Area (slices)	Max Clock	Data rate (bits/Clock cycle)	Max Throughput	Throughput /Area (10^3)
logic	8	594	75.622MHz	0.73	55.20M	92.93
logic	32	728	62.514MHz	2.91	181.2M	248.9
logic	64	958	62.655MHz	5.82	364.65M	380.63
table	8	689	139.590MHz	0.73	101.73M	147.6
table	32	1347	122.485MHz	2.91	356.4M	264.6
table	64	1350	117.595MHz	5.82	684.4M	506.7

Table III shows how these designs results. The smaller the Datapath, the smaller the throughput is, and also the area. But the results also show bigger Datapath have better efficiency because its throughput/area ratio is bigger. This is because the controller logic will consume a bigger resource for smaller data path, especially for the address generation part. This could be improved by design a smarter controller or even store all the control state to a ROM.

B. Results for Key Scheduling

TABLE IV
RESULTS FOR VIRTEX5

S-Box	Data path	Area (slices)	Max Clock
logic	32	187	79.695MHz
logic	64	243	61.333MHz

Table IV shows how the on-the-fly key scheduling will consume the resources. It shows a significant portion of the whole AES compact encryption resource usage.

C. Comparison with eSTREAM Candidates

TABLE V
COMPARISON WITH ESTREAM CANDIDATES

S-Box	Data path	Area (slices)	Max Clock	Data rate	Max Throughput	Throughput /Area (10^3)
logic	8	594	75.622MHz	0.73	55.20M	92.93
logic	32	728	62.514MHz	2.91	181.2M	248.9
logic	64	958	62.655MHz	5.82	364.65M	380.63
table	8	689	139.590MHz	0.73	101.73M	147.6
table	32	1347	122.485MHz	2.91	356.4M	264.6
table	64	1350	117.595MHz	5.82	684.4M	506.7
DECIM V2		80	185MHz	0.25	46.25	580
DECIM 128		89	174MHz	0.25	43.5	490
Grain 128		50	196MHz	1	196	3,920
Trivium		50	240MHz	1	240	4,800
Moustique		278	225MHz	1	225	810

D. Results Analysis

The results show that the AES could be implemented as a stream cipher in different ways. AES shows a good potential to be used as a stream cipher. However, the resources used by AES are bigger due to its complex design. This will need more efforts to simplify the design.

V. CONCLUSION

The design here achieves a nice result for AES as a stream cipher. By comparing with other eSTREAM cipher candidates, AES shows its potential to implement in a very compact way. However, There's still more work to simplify the design to achieve a better resource usage. And the SRL could be used to optimize the design when targeting Xilinx FPGA. Another more promising optimization is to store all the control state to a ROM to accomplish a very compact design. And if the key scheduling part is well integrated with the round function, this will show a good result, too. Future work includes adding pipeline stages to the round design; unfold the entire round to achieve a better throughput and so on.

REFERENCES

- [1] Tim Good and Mohammed Benaissa, AES as Stream Cipher on a Small FPGA,
- [2] David Huang, Mark Chaney, Shashi Karanam, Nich Tom and Kris Gaj, *Comparison of FPGA-Targeted Hardware Implementations of eSTREAM Stream Cipher Candidates*, CA: Wadsworth, 2008, pp. 123–135.
- [3] eSTREAM Project, could be found at <http://www.ecrypt.eu.org/stream/>.
- [4] Kris Gaj and Pawel Chodowiec, “FPGA and ASIC Implementations of AES”, book chapter,
- [5] NIST, “Announcing the ADVANCED ENCRYPTION STANDARD (AES)”, Federal Information Processing Standards Publication 197, November 26,2001
- [6] Network Working Group, R. Housley, “RFC 3686 - Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)”, Internet RFC/STD/FYI/BCP Archives, January 2004.
- [7] Pawe_l Chodowiec and Kris Gaj, “Very Compact FPGA Implementation of the AES Algorithm”, *Cryptographic Hardware and Embedded Systems -- CHES 2003*: 5th.
- [8] H. Lipmaa, P. Rogaway, and D. Wagner, “CTR-Mode Encryption, Comments to NIST concerning AES Modes of Operations”, *Symmetric Key Block Cipher Modes of Operation Workshop*, 2000.
- [9] Kris. Gaj, “AES implementations in software and hardware”, ECE746 Lecture, GMU.