Sugier Jarosław
Wrocław University of Technology, Wrocław, Poland

Implementation of symmetric block ciphers in popular-grade FPGA devices

Keywords

cryptographic processor, AES, Serpent cipher, hardware implementation, pipelining, iterative architecture.

Abstract

In this paper we discuss hardware implementations of the two best ciphers in the AES contest – the winner Rijndael and the Serpent – in low-cost, popular Field-Programmable Gate Arrays (FPGA). After presenting the elementary operations of the ciphers and organization of their processing flows we concentrate on specific issues of their implementations in two selected families of popular-grade FPGA devices from Xilinx: currently the most common Spartan-6 and its direct predecessor Spartan-3. The discussion concentrates on differences in resources offered by these two families and on efficient implementation of the elementary transformations of the two ciphers. For case studies we propose a selection of different architectures (combinational, pipelined and iterative) for the encoding units and, after their implementation, we compare size requirements and performance parameters of the two ciphers across different architectures and on different FPGA platforms.

1. Introduction

Symmetric block ciphers, since proliferation of the DES algorithm in the 70. of the previous century, are the standard method of data protection that ensures safe and secure operation of contemporary IT systems. Today, after DES expiration due to insufficient cryptographic strength when compared to available processing power, its descendant – the AES algorithm – is the typical solution applied in this area. In this paper we discuss hardware implementations of the two best ciphers in the AES contest – the winner Rijndael and the Serpent – in low-cost, popular field-programmable gate arrays (FPGA). The text is organised as follows. The rest of this introductory chapter presents motivation of our specific implementation approach as well as the origin of the two algorithms which we have selected as the representatives of symmetric block ciphers applied in practice. Internal organisation of the two methods is briefly reminded in chapter 2 and then, in chapter 3, specific problems of their implementations in FPGA devices are described. Finally, chapter 4 discusses hardware verification of the proposed solutions.

1.1. Motivation of this work

The AES standard is more than 10 years old now and, obviously, there is a vast knowledge of possible soft- and hardware implementations of the block ciphers proposed at the time of its development. Because realisation of the cipher transformations directly in hardware was one of the important options taken into account from the very beginning in the contest, there are numerous solutions described in the literature that implement the ciphers in both mask- (ASIC) and field-programmable gate arrays (FPGA). The essential initial evaluation was included in [5] while other examples can be found, for example, in [6]-[9], [12] and [14]-[17]. Most of the typical solutions are highly customized for specific device architectures and / or operating environments. Being created first of all for topmost performance, their excellent operational parameters were possible thanks to elaborate optimizations which often involved manual fine-tuning of mapping, layout or routing phases during FPGA implementation. Also hardware platforms for these projects demanded the fastest, largest and often most expensive chip families in the FPGA world.

In this work we deliberately look from different point of view at the hardware implementation of the cipher unit. While the “top-notch specialization” approach

was natural in the early years of AES conception, today the situation has changed thanks to ever-growing capacity of programmable devices and another course of action becomes more and more common. First of all, the cipher module often becomes just one of the elements in the system-on-the chip implemented in a popular, often low-cost, hardware device. In designs of this kind it is not desirable, or even not possible, to make the cipher optimization the dominant aspect of the whole project. The module must share both resources and optimization effort appropriately with the rest of the system. Secondly, encoding / decoding throughput parameters do not need to reach multi-Gbps values in common equipment designed for personal use; numbers in the range of single Gbps are sufficient for popular transmission channels like High-Speed USB or consumer-grade mass-storage devices. In this situation not the performance of the unit (generally understood almost always as maximum data throughput) but its flexibility and fast, fully automatic implementation become highly valued features that facilitates installation of the cipher module in the whole design and, consequently, reduces time-to-market in device development.
Hence the aim of this work is to investigate low-cost FPGA implementations of the two best ciphers that emerged as a result of the AES contest. Based on the original results that are presented here one can compare the potential of the two methods and evaluate expenses at which their superior efficiency can be achieved. In particular, the terms "popular-grade" or "low-cost" that we refer to in the title and in the text are understood as follows: 1) the programmable device used for implementation is chosen from inexpensive, popular and commonly used line of FPGA chips, widely available on the market; 2) the design is described in hardware description language on the relatively high level of abstraction (no less than at Register Transfer Level, RTL) and then synthesized and implemented fully automatically by standard software provided by the chip manufacturer, without any special "handmade" optimization, neither in layout nor routing.

1.2. Origin of Rijndael (AES) and serpent

The Data Encryption Standard (DES), developed by IBM and standardized by US National Institute of Standards and Technology (NIST) in 1977, had been internationally used as the best encryption method until the mid-1990s when its strength was seriously questioned by successful attacks. Due to relatively short length of the DES key (56 bits) it became possible to complete brute-force exhaustive search of the entire key space in more and more acceptable times using specialized hardware platforms and / or distributed computing. Facing an imminent demise of DES, in January 1997 NIST issued a first call for a successor algorithm, to be called an Advanced Encryption Standard, or AES.
The request called for unclassified, publicly disclosed encryption algorithm, available royalty-free, worldwide. In response total of 15 new ciphers were submitted from several countries. After two conferences organized by NIST to promote public examination of the proposals (AES1, August 1998 and AES2, March 1999) the five finalists were announced in August 1999. Their AES2 votes were as follows:
- Rijndael: 86 positive, 10 negative
- Serpent: 59 positive, 7 negative
- Twofish: 31 positive, 21 negative
- RC6: 23 positive, 37 negative
- MARS: 13 positive, 83 negative

During the last AES3 conference in April 2000 the authors had the last chance to present their proposals and then in October 2000 NIST announced the final decision which was consistent with the AES2 voting: the winner was Rijndael cipher. Under the new name of AES it was announced the U.S. Federal Information Processing Standard 197 (FIPS 197) in November 2001 ([10]).

2. The algorithms

Both algorithms – the AES and the Serpent – are symmetric block ciphers that are examples of substitution-permutation networks (SPN). Their processing consists in a sequence of *rounds*, with every round being a specific set of *elementary operations* executed repeatedly over a given *block of data*. Independently from cipher (data) path there is a separate processing whose task is to provide every round with its individual *round key*, generated from user-supplied secret *external key*.
In the following presentation of the two ciphers we will use unified symbols which in some cases will be different from official nomenclature used in specifications submitted by the authors.

2.1. The AES (Rijnadael)

The AES cipher was initially developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and submitted to the AES contest under the name "Rijndael". The approved final standard is publicly available as a FIPS publication in [10] and, strictly speaking, is a subset of Rijndael with fixed block size of 128b(it) and allowed key sizes of 128, 192 or 256b (the original method accepts block and key sizes equal to any multiply of 32b from 128 to 256). In discussion of this paper we consider

exclusively the AES-128 version, i.e. we assume size of the key to be 128b.
In AES the 128b data block is considered to be a 4×4B(yte) array, termed *the State*. For 128b key, the whole encryption procedure is divided into exactly 10 rounds after one auxiliary round executed at the beginning of the process. During encryption every round consists of four elementary state transformations executed in specific order: *byte substitution* (*SBox*), *row shifting* (*SR*), *column mixing* (*MC*) and *addition* of a round key (bitwise XOR operation, denoted with the '⊕' symbol). The two exceptions are the initial round (numbered as 0) that consists only of addition of the external user key and the last one (number 10) that does not perform column mixing.
Let P be a 128b plaintext (input), B_i – a state block that enters the i-th round R_i, K – external user key, K_i – the key generated for round i, and C – encoded ciphertext (output). The complete data path of the AES can be expressed with the following equations:

$$B_1 := P \oplus K$$

$$B_{i+1} := MC(\, SR(\, SBox(\, B_i \,)\,)\,) \oplus K_i, \quad i = 1 \ldots 9 \quad (1)$$

$$C := SR(\, SBox\,(\, B_{10} \,)\,) \oplus K_{10}$$

As it was mentioned before, rounds 1-10 use extended keys that needs to be generated from the main key by a separate so called *key expansion* routine. These computations, in turn, operate on 32b words w_i, i = 0..43, which, upon elaboration, are directly copied to the round keys K_i. Initially, the first four words are filled with bits from the user key:

$$\{w_0, w_1, w_2, w_3\} := K \quad (2)$$

and then, for i = 1..10, every group of four words that creates round key K_i is computed as follows:

$$\begin{aligned} w_{4i} &:= SBox(\, w4_{i-1} <<< 8\,) \oplus \mathtt{Rcon}[\, i \,] \oplus w_{4i-4} \\ w_{4i+1} &:= w_{4i} \oplus w_{4i-3} \\ w_{4i+2} &:= w_{4i+1} \oplus w_{4i-2} \\ w_{4i+3} &:= w_{4i+2} \oplus w_{4i-1} \\ K_i &:= \{w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}\} \end{aligned} \quad (3)$$

where <<< denotes left rotation (always by 8 bits, in this case), the *SBox* transformation uses exactly the same substitution boxes as the cipher path, and the Rcon is a vector of ten 32b constants statically defined in the standard.

2.2. The serpent

Serpent ([1]-[3]) was developed by Ross Anderson (University of Cambridge Computer Laboratory), Eli Biham (Technion Israeli Institute of Technology), and Lars Knudsen (University of Bergen, Norway). In the version that was submitted for AES contest the method operates on 128b data blocks and requires 256 bit external key. The transformation flow is divided into 32 rounds (numbered 0-31) repeated over the data block with each round consisting of (nearly identical) sequence of elementary operations. As in the AES, each of the first 31 rounds requires separate 128-bit round key while the last round needs two keys; therefore, total of 33 round keys must be generated by a processing path called in this case *key schedule*.
Before the plaintext block enters the procedure a special bit reordering – so called *initial permutation IP* – is performed. The plaintext P after permutation gives block B_0, which is the input to the first round. At the end of the round chain, the output of the last round, B_{32}, undergoes the *final permutation FP* (which is an inverse of *IP*) giving the ciphertext *C*.
As the first transformation in each round the block B_i is XOR-ed with the round key K_i and then the resulting vector is passed through substitution boxes. The algorithm defines 8 different S-Boxes numbered 0 … 7 with each round R_i using S-Box number i mod 8. Subsequently, the vector created by S-Boxes undergoes *linear transformation LT* giving block B_{i+1} that is the input to the next round. In the last round R_{31} the linear transformation is replaced with XOR operation with K_{32} and therefore two keys are required in this round. The complete data path can be formally described as:

$$\begin{aligned} B_0 &:= IP(\, P \,) \\ B_{i+1} &:= LT(\, SBox_{i \bmod 8}(\, B_i \oplus K_i \,)\,), \quad i = 0 \ldots 30 \\ B_{32} &:= SBox_7(\, B_{31} \oplus K_{31} \,) \oplus K_{32} \\ C &:= FP(\, B_{32} \,) \end{aligned} \quad (4)$$

Operation of the key schedule is no less involved. First, a set of 32-bit *prekeys* w_i is created: the external key K is copied to the first ones numbered from –1 to –8

$$\{w_{-1}, w_{-2}, \ldots w_{-8}\} := K \quad (5)$$

and then another 132 prekeys $w_0 \ldots w_{131}$ are generated by the following affine recurrence:

$$w_i := (w_{i-1} \oplus w_{i-3} \oplus w_{i-5} \oplus \quad (6)$$
$$w_{i-8} \oplus \phi \oplus i) <<< 11$$

where ϕ symbol stands for the fractional part of the golden ratio value $(\sqrt{5}+1)/2$ (32-bit vector 0x9E3779B9 in hexadecimal notation). Having the prekeys, the round keys are calculated using the same set of 8 substitution boxes that are used in the cipher path. The rule is that key K_i is computed from a group of four prekeys w_{4i}, w_{4i+1}, w_{4i+2} and w_{4i+3} using S-boxes number $(3 - i) \bmod 8$:

$$K_0 := IP(SBox_3(w_0, w_1, w_2, w_3))$$
$$K_1 := IP(SBox_2(w_4, w_5, w_6, w_7))$$
$$K_2 := IP(SBox_1(w_8, w_9, w_{10}, w_{11})) \quad (7)$$
$$\ldots$$
$$K_{32} := IP(SBox_3(w_{128}, w_{129}, w_{130}, w_{131}))$$

3. Specifics of FPGA implementation

For both algorithms, relative simplicity of elementary operations as well as regular sequential structure of round series lead to effective implementation in both software and hardware, but some specific aspects of FPGA architecture do differentiate particular results. In brief discussion that follows we will see these problems in the context of popular-grade device families from Xilinx that were chosen for case studies of this work: Spartan-3 ([18]) and Spartan-6 ([19]). We will concentrate on the three aspects: programmable resources available in the selected devices (section 3.1), their use for realisation of the cipher transformations (section 3.2) and overall organization of cipher processing – i.e., the architecture of the unit (section 3.3).

3.1. Programmable resources

For simplified structure of the logic cell in the two Xilinx families please see *Figure 1*. In all FPGA devices from this producer, so called *Look-Up Table* (LUT) is the element located in every logic cell which is provided for generation of any combinational function. A single LUT is a ROM table filled with zeroes and ones during configuration according to the function which should be computed at its output. In case of Spartan-3 devices, the LUTs are 4-input tables holding 16b each thus they can generate any function of maximum 4 variables. A function of fever variables still must occupy one LUT while any wider function will use more of them (5-input function = 32b or 2 LUTs, 6-input function = 64b or 4 LUTs, etc.). In Spartan-6 architecture, in turn, every LUT table has the total capacity of 64b being sufficient for generation of a 6-input Boolean function but, alternatively, can be configured for generation of two different 5-input functions of the same variables. These configuration nuances will have significant impact on implementation of the cipher transformations.

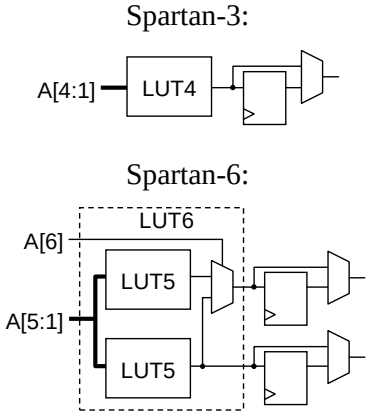

Figure 1. Simplified structure of the logic cell in Spartan-3 and Spartan-6 architectures

The signal which goes out of the LUT can be optionally stored in the flip-flop so virtually every signal generated in the array can be easily synchronously registered: introducing some amount of flip-flops into the FPGA project, like it is in pipelined designs, usually can be accomplished at very little additional cost.

3.2. Implementing elementary operations

Looking at equations (1) and (4) it can be seen that both algorithms – AES and Serpent – share very similar set of elementary transformations: (A) key addition, (B) a linear transformation (which is called "column mixing" in the AES terminology), and (C) bit substitution serving as a non-linear transformation. Generation of the round keys defined in equations (3) and (6)-(7) does not add any new kind of operations to this list.

From our discussion we will deliberately exclude any bitwise shift transformations, i.e. the elementary operation *SR* of the AES in equation (1) and vector rotations found in (3) and (6). Although in software implementation these would require due amount of processor cycles for byte transfers, in hardware, since no data is modified, they are just static bit permutations that are accomplished completely in routing and do not absorb any logic resources.

(A) *Key addition*. The first operation on the list is the simplest one conceptually: 128 bits of the data block are bitwise XOR-ed with suitable round key. From hardware point of view, this creates a set of 128 2-input functions which, without optimization, would require 128 LUT elements in Saprtan-3 or, thanks to possible generation of two functions in a single LUT, 64 elements in Spartan-6. In any case, since the functions are of only two variables, these would not be used to their full potential and only the optimization procedures of the implementation tool can improve this by fitting in the LUT the XOR function merged with some another transformation adjacent in the processing chain. Such merging was not explicitly defined in the VHDL code in the projects discussed later in this work, though.

(B) *Linear transformation*. Compared to the previous operation, this transformation is much more complex for realisation. Leaving behind its cryptographic merit, at the binary level it can be expressed as a matrix multiplication where each output byte is computed through series of sum mod 2 and shift operations that, down to the strictly hardware point of view, make up a combinational circuit. In hardware, every output bit of the *MC* / *LT* module is computed as some particular function that reads 3 to 7 bits from the input and passes them through a net of XOR gates. The resulting gate network in hardware is highly irregular and its implementation with LUTs, as well as its later optimization, is a major challenge for implementation tool. Along with the aim of this work, this task was left for fully automatic operation of the implementation software and no handmade optimization was applied in the VHDL code.

(C) *Non-linear transformation*. Operation of this kind is present in both ciphers and, putting aside its mathematical background again, in both cases it takes the form of statically defined transcoding table: 8b→8b in AES and 4b→4b in Serpent. The substitution operates independently on 8 / 4b parts of the data block and if this is to be executed in parallel on the whole vector (a principal option to choose when striving for high throughput) then 16 (AES) / 32 (Serpent) transcoding modules of appropriate type need to be placed one by one. In hardware, such transcoding is an 8- / 4-input combinational function that can be implemented either as a network of logic gates or as a 256×8 / 16×4 ROM lookup table. The latter method usually leads to more effective implementation, especially in FPGA arrays, where 8-input Boolean functions are considered wide and resources for their implementation are limited.

A ROM block in FPGA architecture, in turn, can be implemented either as a distributed memory module composed of elementary LUTs or its contents can be stored in one of Block RAM modules (BRAM) available on-chip next to the FPGA array as an additional resource.

In case of the AES, in Spartan-3 device a distributed ROM storing one 8b-wide *SBox* would take 256 × 8 / 16 = 128 LUT elements + additional logic for address decoding, or 256 × 8 / 64 = 16 LUTs in Spartan-6 array. Multiplying these numbers by 16 (all data block transcoded in parallel) we reach total of 2048 (Spartan-3) or 256 (Spartan-6) LUTs for representation of *SBox* operation in a single round. For Spartan-3 devices this can be a significant amount especially when such module is replicated for each round (×10) as it is in some architectures.

On the other hand, BRAMs are 16kb in size in both families and can be configured as 2k × 8. This amount of memory is more than enough for two independent S-boxes that can operate in one block using its dual-port feature (even then only 25% of total capacity is used). Hence minimum of 8 BRAMs per round is needed for AES *SBox* operation if all *State* bytes are to be transcoded in parallel.

In Serpent the substitution is defined for data chunks two times smaller and parallel processing of the whole data block needs 128b / 4b = 32 modules. The observations made above for AES regarding different implementation options are still valid but the numbers are different: every Serpent *SBox* implemented as a 16×4 distributed ROM would need just 4 LUTs in Spartan-3 or 2 LUTs in Spartan-6, giving total of 128 or 64 LUTs per round. The difference as compared to the AES is especially outstanding for the Spartan-3 family: implementing the AES on this platform with *Sboxes* stored in distributed memory is very costly and if all rounds are to be repeated in hardware, as they are, for example, in pipelined architectures of the whole cipher unit, then only the biggest devices are large enough to accommodate the required number of elements for this transformation alone. On the Spartan-6 platform this disparity is much smaller and the lower number of AES rounds (10 vs. 32) can compensate for it completely.

Using the block RAM for Serpent *SBoxes* is not a good option: in its case substitution needs 32 boxes per round so 16 block RAMs would be needed but with utilization of only 1/512 of their capacity. Moreover, all 32 rounds repeated in hardware would take 16 × 32 = 512 blocks – an unreasonably high occupancy compared to its effective results.

In case of the AES, apart from resource utilization the important difference between distributed vs. block RAM lies in operation mode: reading the distributed ROM in LUTs is purely asynchronous (memory contents is present at the data outputs only

after small combinatorial delay after the new read address has been established) while read operation of the BRAM must be synchronous (to read the memory contents after address is set one clock edge is required). Therefore when strictly combinatorial (asynchronous) operation of *SBox* transformation is required the LUT-based distributed memory has no alternatives in Xilinx devices.

3.3. Architectures of the cipher units

Both AES and Serpent implementations can be based on various hardware processing schemes thanks to regular structure of their data processing where a series of (nearly) identical rounds repeatedly transforms the same block of data. In a set of case studies investigated in this work all standard kinds of processing – combinational, pipelined and iterative – were applied in the VHDL language and then implemented in two specific FPGA devices. Their exhaustive discussion was not possible in this paper due to its size limit and can be found in papers [14] and [16]. In total, four kinds of typical architectures were devised and they were applied for the two ciphers as closely as possible.

(A) *Combinational architecture*. In this organization structure of the hardware simply reflects flow of the data that is being encoded. All rounds of the cipher (11 for AES and 32 for Serpent) are implemented as separate hardware modules that create a continuous combinational path from the input to the output registers. In-between, the unit operates as a combinational function that maps 256 input bits (data + key) into 128 output bits (ciphertext). The two designs were specified by expressing as closely as possible the original cipher specification in the VHDL language using strict RTL style. Substitution boxes, both 8b (AES) and 4b (Serpent), were defined according to general Xilinx templates recommended for ROM specification. The cascade of the modules that implement individual cipher rounds was easily constructed with a single `for...generate` statement which greatly improved conciseness and clarity of description.

(B) *Cipher-only (half) pipelined architecture*. The general idea of pipelining is to introduce evenly spaced registers along the combinational path so that in its synchronized operation multiple blocks of data are processed simultaneously during every clock cycle. Taking the combinational architectures of both ciphers as the starting designs for pipelining, the natural points of placing the pipeline registers are the signals B_i that cross boundaries of cipher rounds; this transforms each round into one pipeline stage (so called complete outer loop unrolling) and yields 11 pipeline stages for AES and 32 for Serpent. In this architecture the key generation path remains combinational and this fact slows down changes of the external key during operation of the unit: loading a new key input invalidates the pipeline contents for 11 or 32 clock ticks until new data fill all the cipher stages. This drawback may exclude this architecture from environments with frequent key changes but if it can be assumed that the key remains constant most of the time this is the optimal organization in terms of both speed and size.

(C) *Fully pipelined architecture*. In this organization the key generation path was pipelined in an equivalent way as the cipher one so that the key generator provides the cipher stage with relevant key together with data (i.e. the key must be computed one clock cycle *before* the data). There was no problem with such organisation in the AES unit: since in the first pipeline stage R_0 uses external (user) key, its special preparation is not required. Instead, during the first clock cycle when block B_1 is computed, the K_1 key is prepared simultaneously so that it is ready for the round R_1 in the next cycle. The total number of pipeline stages did not change.
In Serpent, three issues complicated an equivalent solution. Firstly, computation of K_i depends on prekeys w_i from *two* previous stages so additional registers are required for storing previous values of w_i and feeding them two stages down the pipeline must be implemented in the routing. Secondly, the last cipher round needs two keys, so it must be split into two stages: the first one contains key mixing with bit substitution and the second one performs final key mixing only. Such splitting increased the total latency of the unit to 33 clock cycles but, compared to the only alternative solution with 32 stages but with computation of K_{31} and K_{32} in one clock cycle, the shorter clock period compensates this increase more than adequately. Finally, the first Serpent round does not use unmodified external key; instead, K_0 must be computed in a regular way as any other key and during that the data in cipher path must wait going through a dummy (empty) stage introduced right at the beginning of the pipeline. This adds extra 128 flip-flops (negligible compared to the total resource consumption) but also, which is more undesirable, extends the pipeline length to 34. For more detailed discussion about inconveniencies of pipelining in the Serpent algorithm together with evaluation of possible intermediate solutions please refer to [14].

(D) *Iterative architecture*. The iterative architectures investigated for both ciphers were based on one round taken from the fully pipelined organizations. Such a round was supplemented with necessary multiplexing logic (loading the data in – looping

Table 1. The proposed architectures implemented in Spartan-6 (upper values) and Spartan-3 (lower values)

Spartan-6 Spartan-3	Available	*AES*						*Serpent*			
		Combinational	Half-pipelined	Half pipelined with BRAM	Fully pipelined	Fully pipelined with BRAM	Iterative	Combinational	Half pipelined	Fully pipelined	Iterative
Registers	93296	256	1536	256	2944	1664	817	256	4224	16768	806
	40960	271	5061	2771	3913	3913	781	256	4224	16768	783
LUTs	46648	8997	9087	3946	8884	3376	1367	16888	15523	22029	1566
	40960	34566	30426	25328	29976	24583	7986	18939	22708	26876	3995
RAMB8s	344			80		86					
RAMB16	40			20		20					
F_{max}[MHz]		24.4	195	154	215	168	160	7.95	196	169	180
		11.8	83.5	77.0	106	101	77.0	6.35	143	125	96.2
Latency [T_{CLK}]		1	11	11	11	11	11	1	32	34	34
		1	11	11	11	11	11	1	32	34	34
Latency [ns]		41.0	56.4	71.2	51.2	65.6	68.9	126	163	202	189
		84.8	132	143	104	109	143	158	224	272	353
Throughput [Gbps]		3.05	24.4	19.3	26.8	20.9	1.81	0.99	24.5	21.1	0.66
		1.47	10.4	9.62	13.2	12.6	0.88	0.79	17.9	15.6	0.35

back – loading the data out) and a simple controller responsible for counting the repetitions of the loop (round number) and supervising the multiplexers. The controller, in its minimal form, comprised a single "idle/busy" register and a round counter. In both architectures number of clock cycles required for encoding one block of data was identical to the number of pipeline stages in fully pipelined implementations. Again, some additional complications arose in Serpent module. While in the AES just one SBox transformation is used in all rounds, the Serpent defines 8 different SBoxes. This meant that some single "universal" *SBox* had to be created with the contents of all 8 *SBox*'es and an extra 3b input for selection signal, making its implementation with FPGA resources notably more complicated. For this reason one-round iterative architecture usually is not recommended for Serpent; instead, typically 8 rounds are implemented in hardware with the data block looped back 4 times (iteration scheme 8×4 instead of 32×1). Nevertheless, such organization was not chosen in this study for consistency of the results.

4. Implementations

All the 4 above architectures were implemented in Spartan-6 and, for comparison, in the previous family of Spartan-3 devices from Xilinx. From Spartan-6 family a middle-sized chip XC6SLX75 was selected as a representative test platform and it served this role very well. The initial plan was to use Spartan-3E sub family as a comparable alternative, but because it soon turned out that even the largest chip – XC3S1600E – was too small for combinational and pipelined AES designs, it was decided to revert to, nowadays somewhat obsolete, initial Spartan-3 family, and to select the XC3S2000 device.

Initially there were 8 designs (4 for each cipher) and their code was implemented in Xilinx ISE Design Suite version 13.4, twice for the two different target devices. It turned out that the software can optionally implement the two pipelined architectures of the AES with or without utilization of block RAM resources, so this led to the final total of 10 different cases. For other architectures, enabling the use of block RAM did not change the implementation results since the software did not decide to use it even though the VHDL code did include templates of ROM definitions (for *SBox* specification) and they were properly detected in reports of the synthesis tool.

Parameters of the 10 implementations related to their size and performance are included in *Table 1* and they confirm particular strengths of specific organizations: the combinational architectures leads to the shortest possible latency, pipelining is the best way to maximize raw throughput, and the iterative units are optimal if smallest possible resource utilization, at the cost of low performance, is needed. Evaluating utilization of block RAM for the two pipelined architectures of the AES, in Spartan-6 it resulted in remarkable savings in other resources (slices, registers and LUTs) which utilization

dropped roughly by half, but the performance was also affected although not so evidently (approx. 20% drop in the throughput). On Spartan-3 platform, on the other hand, the difference was not so apparent.

What can be also analysed is the difference in effectiveness of cipher implementations with the two tested device families. The general observation is that, putting aside size parameters which are difficult to compare between different architectures, it was the performance of the AES which benefited more from moving to the new family: on average the throughput increased by 100% while for Serpent the increase was around 20%. As it was discussed in section 3, the new capabilities of LUT elements in Spartan-6 are beneficial for implementation of the AES transformations, while in case of the Serpent they offer very little improvement over Spartan-3.

5. Conclusions

It is often said that the AES is faster but the Serpent, having more rounds, is more secure. This paper demonstrates this rule in the context of implementations which use popular-grade FPGA devices. With the previous generation of Spartan chips this principle was affected by the problems with AES implementation: its wide, 8 bit substitution boxes led to very high resource occupation and the Serpent attained additional advantage. The situation has changed with the new generation of Spartan devices form Xilinx: extended capabilities of LUT elements fit very well the needs of AES transformations whereas they bring little progress for the Serpent. The advantage of the latter cipher again remains mainly in better cryptographic strength.

References

[1] Anderson, R., Biham, E. & Knudsen, L. (1998). Serpent: A Proposal for the Advanced Encryption Standard. *Proc. First Advanced Encryption Standard (AES) Candidate Conf.* Ventura, California, http://www.cl.cam.ac.uk/~rja14/serpent.html (accessed April 2012).

[2] Anderson, R., Biham, E. & Knudsen, L. (2000). Serpent and Smartcards. Smart Card Research and Applications. *Proc. 3rd International Conf. CARDIS '98. Lecture Notes in Computer Science*, 1820.

[3] Anderson, R., Biham, E. & Knudsen, L. (2000). The Case for Serpent. *Proc. Third AES Candidate Conf*. New York, http://csrc.nist.gov/archive/aes/index.html (accessed April 2012).

[4] Chu, P.P. (2006). *RTL Hardware Design Using VHDL*. John Wiley & Sons, New Jersey.

[5] Gaj, K. & Chodowiec, P. (2000). Comparison of the hardware performance of the AES candidates using reconfigurable hardware. *Proc. Third AES Candidate Conf*. New York, http://csrc.nist.gov/archive/aes/index.html (accessed April 2012).

[6] Krukowski, Ł. & Sugier, J. (2010). Designing AES cryptographic unit for automatic implementation in low-cost FPGA devices. *Int. J. Critical Computer Based Systems*, 1, 104–116.

[7] Lázaro, J., Astarloa, A., Arias, J., Bidarte, U. & Cuadrado, C. (2004). High Throughput Serpent Encryption Implementation. *Field Programmable Logic and Application, Lecture Notes in Computer Science*, 3203.

[8] Liberatori, M., Otero, F., Bonadero, J.C. & Castineira, J. (2007). AES-128 Cipher. High Speed, Low Cost FPGA Implementation. *Proc. Third Southern Conf. on Programmable Logic*. Mar del Plata, Argentina, IEEE Comp. Soc. Press.

[9] Mroczkowski, P. (2000). *Implementation of the block cipher Rijndael using Altera FPGA*. Military University of Technology, Warsaw.

[10] National Institute of Standards and Technology (2001). Specification for the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication 197. http://csrc.nist.gov/publications/PubsFIPS.html (accessed April 2012).

[11] Osvik, D.A. (2000). Speeding up Serpent. *Proc. Third AES Candidate Conf*. New York, http://csrc.nist.gov/archive/aes/index.html (accessed April 2012).

[12] Piwko, K. (2010). *Hardware implementation of cryptographic algorithms in programmable logic devices*. Dissertation for M.Sc. degree, Wrocław University of Technology, Faculty of Electronics.

[13] RSA Laboratories (1997-99). DES Challenges. http:// www.rsa.com.

[14] Sugier, J. (2010). *Low-cost hardware implementation of Serpent cipher in programmable devices.* Monographs of System Dependability Vol. 3: Technical Approach to Dependability. Publishing House of Wrocław University of Technology, 159-172.

[15] Sugier, J. (2011). Implementing Serpent cipher in field programmable gate arrays. *5th International Conf. on Information Technology ICIT 2011* Amman, Jordan, 91-96.

[16] Sugier, J. (2012). Implementing AES and Serpent ciphers in new generation of low-cost FPGA devices. *Advances in Intelligent and Soft computing Vol. 170: Complex Systems and Dependability*. Springer, 273-288.

[17] Wójcik, M. (2007). *Effective implementation of Serpent algorithm*. Dissertation for M.Sc. degree, Faculty of Electronics and Information Technology, Warsaw University of Technology.

[18] Xilinx, Inc. (2009). Spartan-3 Family Data Sheet. DS099.PDF, www.xilinx.com (accessed April 2012).
[19] Xilinx, Inc. (2011). Spartan-6 Family Overview. DS160.PDF, www.xilinx.com (accessed April 2012).