Implementing AES and Serpent Ciphers in New Generation of Low-Cost FPGA Devices

Jarosław Sugier

Abstract. New generations of FPGA devices that are being continuously developed provide the designers with extended capabilities and create new options for implementation of contemporary ciphers. This work presents implementations of the two best algorithms of the AES contest – Rijndael and Serpent – in Spartan-6 devices from Xilinx and compares them with equivalent effects that were obtained in architectures of the previous generation. The included results allow for evaluation of implementation cost vs. efficiency in contemporary FPGA chips for these two cryptographic algorithms and also provide some conclusions about how the situation changes with development of new, more powerful programmable architectures.

1 Introduction

Dependable operation of numerous contemporary computer systems rely on data protection and this is assured with appropriate encryption methods. Among symmetric ciphers with secret key the AES algorithm is used as a standard solution in most of the applications with Serpent cipher being the main comparable alternative.

In this work we investigate various options for low-cost hardware implementations of the two ciphers and especially look at the changes that were caused in this area by new generation of Spartan-6 family of FPGA devices from Xilinx. The text is organized as follows: after presenting the two algorithms in chapter 2, the various hardware organizations of the cipher unit are introduced in chapter 3. Finally, chapter 4 discusses size and performance parameters that were obtained after implementation of all the variants in Spartan-6 and, for comparison, in Spartan-3 chips.

Jarosław Sugier
Wrocław University of Technology
Institute of Computer Engineering, Control and Robotics
ul. Janiszewskiego 11/17, 50-372 Wrocław, Poland
e-mail: jaroslaw.sugier@pwr.wroc.pl

W. Zamojski et al. (Eds.): Complex Systems and Dependability, AISC 170, pp. 273–287.
springerlink.com

2 The AES Contest: Rijndael vs. Serpent

The first widely used encryption algorithm, the Data Encryption Standard (DES), was developed by IBM and standardized by US National Institute of Standards and Technology (NIST) in 1977. In mid-90s its strength was seriously questioned by successful attacks ([13]) and in January 1997 NIST issued a first call for a successor algorithm, to be called an Advanced Encryption Standard or AES. In response 15 new cipher proposals were submitted from several countries. After two conferences organized to promote public examination of the methods (AES1, August 1998, and AES2, March 1999) the five finalists were announced in August 1999. Their scores in a voting which was organized during the AES2 conference were as follows:

– Rijndael: 86 positive votes, 10 negative;
– Serpent: 59 positive, 7 negative;
– Twofish: 31 positive, 21 negative;
– RC6: 23 positive, 37 negative;
– MARS: 13 positive, 83 negative.

After the last AES3 conference in April 2000, the final decision was announced which was consistent with the AES2 poll: the Rijndael was chosen as the winner. Under the new name of AES it was announced the U.S. Federal Information Processing Standard 197 (FIPS 197) in November 2001 ([10]).

Serpent and Rijndael belong to the same class of round-based cipher algorithms and bear significant resemblance. Both algorithms are symmetric block ciphers that are examples of substitution-permutation networks (SPN). Their processing consists in a set 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 path whose task is to provide every round with its individual *key*, generated form user-supplied secret *external key*.

To summarize the distinction between the two ciphers shortly, it is often said that Rijndael is faster (having fewer rounds) but Serpent is more secure. After the NIST final decision most of the attention concentrated on Rijndael for obvious reasons, but second-to-the-winner Serpent still deserves some consideration because of its advantages that won significant appreciation during the AES contest. It is worth noting that in the AES2 ballot it was the Serpent that received the least number of negative votes.

2.1 The AES (Rijndael) Algorithm

The Rijndael cipher ([10]) was initially developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and the finally approved AES standard, strictly speaking, is its subset with fixed block size of 128b[it] and allowed key sizes of 128, 192 or 256b. To focus the discussion in this paper we consider exclusively the AES-128 version, i.e. we assume size of the key to be 128b.

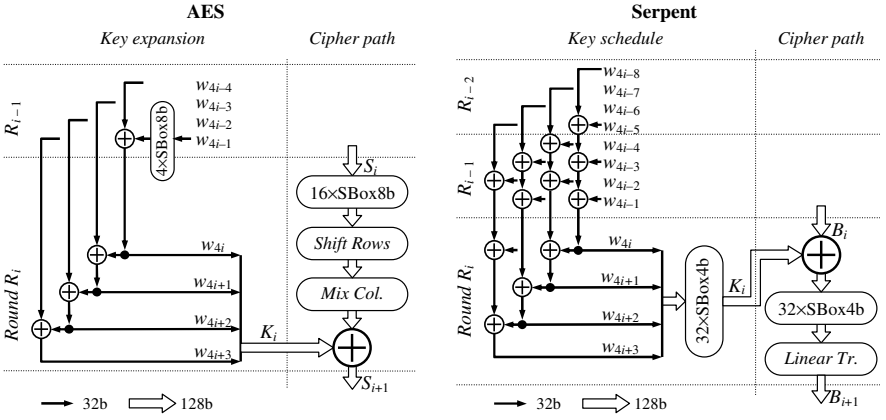

Fig. 1 Data flow in a single round of the AES (left) and Serpent (right) ciphers

Since AES allows only one block size of 128b, it always operates on 16B[yte] chunks of data that form a 4×4B array, termed *the State*. For 128b key, processing of the State during the encryption is divided into exactly 10 rounds plus one auxiliary executed at the beginning of the process.

Let P be a 128b plaintext, S_i – a state block that enters the i-th round R_i, K – external (user) key, K_i – round key, C – encoded ciphertext. The complete data path of the AES can be expressed with the following equations:

$$\begin{aligned} S_1 &:= P \oplus K \\ S_{i+1} &:= MC(\, SR(\, SBox\, (\, S_i\,)\,)\,) \oplus K_i \quad i = 1 \ldots 9 \\ C &:= SR(\, SBox\, (\, S_{10}\,)\,) \oplus K_{10} \end{aligned}$$

That is, the initial round (numbered as 0) consists only of addition of the external (user) key while every regular round number 1 to 9 contains four elementary state transformations executed in specific order: *byte substitution SBox*, *row shifting SR*, *column mixing MC* and *addition* (XOR) *of the round key*. The last round (number 10) does not include column mixing but the other three operations remain unchanged. Additionally, rounds 1÷10 use *extended keys* that need to be generated from the user key by a separate key *expansion* routine. Execution of a regular round (1÷9), along with generation of its key, is shown in the left part of Fig. 1.

The key expansion routine, in turn, operates on 32b words w_i, $i = 0..43$, which, after computation, are directly copied to the round keys K_i. The first four words are initialized with bits from the user key:

$$\{w_0, w_1, w_2, w_3\} := K$$

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

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

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 static vector of ten 32b constants defined in the standard.

2.2 The Serpent Algorithm

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 the contest the method operates on 128b blocks of data with 256b external key. If the user supplied key is shorter (call for the standard allowed also key lengths of 128 and 192b) simple expansion procedure is applied which ensures that the method always starts with the full 256b key. The transformation flow is divided into 32 almost identical rounds with every round using its own 128-bit round key generated by the *key schedule*; since the last round needs two keys, total of 33 different round keys are required.

In addition to the symbols defined above, now let the data block that enters the i-th round is denoted as B_i. Before the plaintext block enters the procedure a special bit reordering – so called Initial Permutation *IP* – is performed (this reordering has no cryptographic significance and was introduced only for bit-sliced implementations). The plaintext P after permutation gives block B_0, which is the input to the first round number 0. The output of the last round, R_{31}, after application of the Final Permutation *FP* (which is an inverse of IP) gives the ciphertext C.

The complete data path from the plaintext P to the ciphertext C can be formally represented by a sequence of the following equations:

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

Operation of a single round, together with generation of its key, is shown in the right part of Fig. 1. As the first transformation, the block B_i is XOR-ed with the round key K_i that is supplied by the key schedule, and then the resulting vector is passed through substitution boxes. The specification defines 8 different S-Boxes numbered 0 … 7 with each round R_i using S-Box number i mod 8. The vector created by S-Boxes finally 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 the extra last key K_{32} and therefore two keys are required in this round, to the total of 33 keys in the whole process.

The key generation in Serpent is no less involved. The schedule generates first a set of 32-bit *prekeys* w_i which are later used for computation of round keys. The starting 8 prekeys numbered from –1 to –8 are filled with bits of the external (user) key K (after its expansion to 256b, if necessary):

$$\{w_{-1}, w_{-2}, \ldots w_{-8}\} := K$$

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

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

where ϕ is the fractional part of the golden ratio $\left(\sqrt{5}+1\right)/2$ represented as 32-bit vector (0x9E3779B9 in hexadecimal notation).

The final round keys are calculated from the prekeys using the same set of 8 substitution boxes that are defined for the cipher path. The general rule is that the key K_i is computed from a group of four prekeys w_{4i}, w_{4i+1}, w_{4i+2} and w_{4i+3} that undergoes bit substitution and reordering:

$$\begin{aligned}
K_0 &:= IP(SBox_3(w_0, w_1, w_2, w_3)) \\
K_1 &:= IP(SBox_2(w_4, w_5, w_6, w_7)) \\
&\ldots \\
K_{31} &:= IP(SBox_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
K_{32} &:= IP(SBox_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}$$

To avoid repetitive use of the same substitution as later in the round, during computation of K_i the schedule uses S-boxes number $(3 - i)$ mod 8.

3 Implemented Architectures

Apart from relative simplicity of elementary operations at the binary level, ease of hardware implementation of the AES and Serpent algorithms comes from the fact that their processing flow is composed of (almost) identical rounds that are repeatedly executed over a given block of data. This leads to many potential processing schemes that blend different flavours of combinational, pipelined and iterative architectures ([4] – [9], [11] – [12], [14] – [16]).

In this study efficiency of hardware implementation of both ciphers will be tested using four essential types of processing: combinational, half (cipher-only) pipelined, fully (both cipher and key) pipelined, and iterative. For brevity, every implementation will be given a name starting with a letter A (for AES) or S (for Serpent) with indication of its type that follows: C (combinational), HP (half, i.e. cipher-only pipelined), FP (fully, i.e. both cipher & key, pipelined) and I (iterative). Since the AES pipelined architectures (AHP and AFP) can be optionally implemented with or without utilization of block-RAM resources in the FPGA chip, this leads to the total of 10 different implementations which will be investigated.

3.1 Combinational Dataflow

In this organization hardware structure closely follows 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 registers (plain text *P*) to the output registers (cipher text *C*). Inbetween, the module operates as a combinational function that maps 128 + 128 = 256 input bits (data + key) into 128 output bits (cipher). The only registers used in this design are located in the *P* and *C* ports. The *K* input is not registered thus only the $P \rightarrow C$ path is taken into account by the implementation tool during optimization of the propagation speed.

In both cases (AC and SC) the design was specified by porting the specification to the VHDL language using strict RTL style: there were no instances of library elements, no sequential (procedural) descriptions were inserted and the code was free from references to any specific hardware attributes. After definition of all internal signals as `std_logic_vector` type, particular elementary operations were defined as separate entities with exception of key mixing, which was implemented simply with built-in `xor` operator at the place of their occurrence. Substitution boxes, both 8b (AES) and 4b (Serpent), were defined according to general templates recommended for ROM specification. AES row shifting and rotations required in key expansion or key schedule were treated as simple bit reordering in `std_logic_vector` signals and expressed with concurrent signal assignments (in hardware implementations, as opposed to software realizations, these transformations are done exclusively in routing and actually do not require any logic). The other operations: column mixing *MC* and linear transformation *LT* at the binary level end up as pure XOR networks and were represented with due number of concurrent assignments. The cascade of the modules that implement individual cipher rounds was easily constructed with a single `for…generate` statement which improved greatly conciseness and clarity of description.

A diagram describing structure of these architectures would mostly reproduce Fig. 1 hardly introducing any new information and, for brevity of this work, it is not included.

3.2 Cipher-Only Pipelining

The general idea of pipelining is to introduce evenly spaced registers along the combinational path so that in its synchronized operation several blocks of data can be processed at the same time during every clock cycle. In the combinational architectures of both ciphers the natural points of placing the pipeline registers are the signals S_i / B_i that cross boundaries of cipher rounds; this transforms each round into one pipeline stage. In technical terms such organization can be interpreted as a complete outer loop pipelining ([5]) and yields 11 pipeline stages for AES vs. 32 for Serpent. This means that valid output appears 11 or 32 clock cycles after input and although it does not improves the latency (which is actually worse than in the case of combinational propagation due to non-zero flip-flop switching time and non-ideal pipelining) the throughput (amount of data processed

in unit time) rises enormously thanks to the parallel processing of multiple data in pipeline stages.

In this version of the 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 may exclude this architecture from environments with frequent key changes but if the key can remain constant most of the time it is the optimal organization in terms of both speed and size.

Adding large amount of registers (128b × number of pipeline stages) may seem to be a substantial increase in resource usage but in case of FPGA architectures this increase is easily absorbed by the array. In these devices a flip-flop is included in every logic cell right at the output of combinational configurable element (Look-Up Table, LUT) so the only actual difference is that now some of them are used for registering the LUT signal while in combinational organization they were left unused. This usually does not affect the total number of occupied logic cells but just increases their utilization.

3.3 Full Pipelinaing

The drawback of the half-pipelined architecture – incompatibility with applications that require frequent changes of the key – can be a significant weakness in many applications. In general it is not recommended to encode large amounts of data with the same key because the attacker could get some information about it without breaking the cipher, namely by statistical analysis of the encoded stream.

To prepare the encryption unit for loading a new key with every block of data the key generation path should be pipelined in an equivalent way as the cipher path. More precisely, the pipelined key generator should provide the cipher stage with relevant key together with data which leads to conclusion that the key must be computed one clock cycle *before* the data is processed.

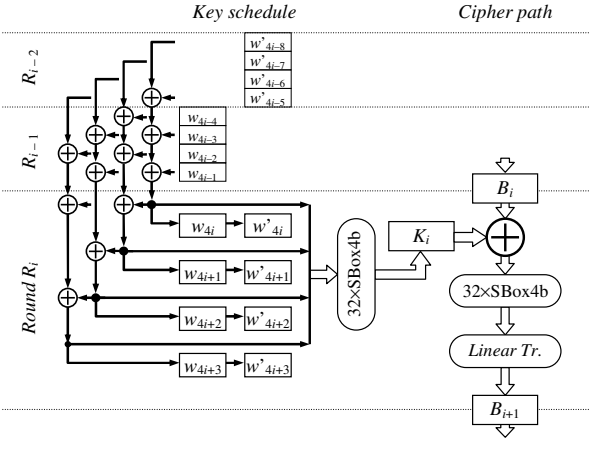

Fig. 2. Single round in fully pipelined implementation of the Serpent algorithm (SFP)

There is no problem with such organisation of the AES cipher: since in the first pipeline stage the round 0 uses external (user) key, its special preparation is not required. Instead, during the first clock cycle when the S_1 vector is computed, simultaneously the K_1 key can be prepared form K so that it is ready for round R_1 in the next cycle. The consecutive rounds work in the same way: R_i (i.e. S_{i+1}) is computed in parallel, simultaneously with preparation of the K_{i+1}.

Looking at the diagram in Fig. 1, the registers would be added right in the places of signals w_i and S_i. Thus, in case of the AES, the workflow of cipher and key paths was mapped in a natural way onto operation of the two pipelines in hardware.

In Serpent, in turn, situation at first looks similar: since computation of the round keys depends on prekeys w_i, these signals must be stored in pipeline registers. But the first problem is that, due to more complex key data dependency, computation of K_i in stage i depends on prekeys from not only stage $i - 1$ but also i – 2, so additional registers – denoted as w' – are required for storing previous values of w and feeding them *two* stages down the pipeline. This factor alone doubles the number of the key schedule registers. Moreover, w' registers are not located at LUT outputs – they are loaded with data from another registers – which is not an advantageous configuration for FPGA implementation.

Secondly, the last cipher round – R_{31} – 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 only final key mixing. An alternative solution – computation of two keys K_{31} and K_{32} in one clock cycle – is not a good option: the key schedule is relatively complex and a combinational path generating two keys would introduce unacceptable long delay holding back performance of the whole unit. Splitting the last round into two stages increases the total latency to 33 clock cycles but, compared to solution with 32 stages but with computation of K_{31} and K_{32} in one clock cycle, the shorter clock period compensates this more than adequately.

Another problem is that the first Serpent's 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 idle going through a dummy (empty) stage added right at the beginning of the pipeline. This adds extra 128 flip-flops (which is a negligible increase compared to the total resource consumption) but also extends pipeline length to 34.

Detailed descriptions of different options for pipelining Serpent unit can be found in [14] along with evaluation of their performance vs. size trade-offs. It was shown that the final optimum solution is reached after adding registers not only for w_i but also for K_i signals. The resulting architecture is shown in Fig. 2 where pipeline registers are marked as rectangles. In this organization new computed values of prekeys w are not only stored in the flip-flops, but in the same cycle they go through the SBoxes evaluating new K_i value which is latched in the extra

registers. As a result the longest combinational path (which decides about maximum frequency of operation of the whole unit) now runs from registers K_i to B_{i+1} and does not contain any elements belonging to key computation. Within the key schedule, on the other hand, there are two paths, both originating from *w*/*w*' flip-flops: the first one computes next values of *w* and the second one extends additionally through S-boxes to the registers K_i. Such a distribution of the elements in the combinational paths turned out to be the most balanced configuration for optimal (highest) performance.

The increase in speed that results from this amendment is accomplished at the cost of 33×128 = 4224 flip-flops but it was shown that this did not incurred any increase in total number of occupied logic cells – all the new registers were located at the outputs of the LUT elements used for implementation of the SBoxes and were absorbed in cells already occupied ([10]).

3.4 Iterative Loop

The two iterative architectures proposed in this study – AI and SI – are based on the structure of one round taken from the fully pipelined architectures (AFP and SFP). Such a single round was supplemented with necessary multiplexing logic (loading the data in – looping back – loading the data out) and a simple controller responsible for counting the repetitions of the loop (round numbers) and supervising the multiplexers. The controller, in its minimal form, comprises 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 AFP / SFP implementations. Every clock cycle completes processing which corresponds to one stage of the pipeline (usually equivalent to one cipher round, apart from the above discussed exceptions for the Serpent case).

One issue needs to be pointed out here, though. While in the AES there is just one SBox transformation used in all rounds in both data and key processing, the Serpent defines 8 different SBoxes, each one being applied in exactly four rounds in the cipher path and in another four rounds in the key path. In iterative organization where just one "universal" round is realized in hardware this means that the "universal" SBox must be created which includes the contents of all 8 regular substitution tables and additionally provides extra 3b input for selection signal. Such a solution is not elegant because, effectively, the SBox becomes a 7-input function (4b data + 3b selection) in place of a 4-input one, which makes its implementation with FPGA resources notably more complicated. For this reason one-round iterative implementation is usually not recommended for Serpent; instead it is proposed to implement 8 rounds in hardware with the data block looped back 4 times during the encoding (8 × 4 instead of 32 × 1). Nevertheless, such organization was not implemented in this study for consistency of examined solutions.

4 Implementation Results

All the 4 above architectures were implemented in Spartan-6 and, for comparison, in the previous family of Spartan-3 devices from Xilinx. There was 8 designs in total (4 for each cipher) and *the same* code was implemented twice in Xilinx ISE Design Suite version 13.4, for the two different target devices selected. Implementation was fully automatic, without any hand-made fine tuning neither in placement nor in layout. Since it turned out that AES pipelined architectures (AHP and AFP) can be optionally implemented with or without utilization of block RAM resources available in the FPGA chip, this gave the total of 10 different cases. The AHP and AFP architectures implemented with block RAM are marked with "_B' suffix.

From Spartan-6 family a middle-sized chip XC6SLX75 was selected as a representative test platform and it served this role very well but selection of Spartan-3 device was more difficult. The initial plan was to use Spartan-3E sub family intended for general, logic-optimized projects. As it soon turned out, even the largest 3E chip – XC3S1600E – was too small for combinational and pipelined AES designs. In other contemporary Spartan-3 families: I/O optimized Spartan-3A, flash-memory based Spartan-3AN and DSP oriented Spartan-3A DSP, only the largest Spartan-3A DSP chips were large enough but this family is optimized for different type of processing. Therefore it was decided to revert to, nowadays somewhat obsolete, initial Spartan-3 family, and to select the XC3S2000 device.

The results are presented in Tables 1 and 2. In general, different types of architectures behave as expected: the combinational organizations give the shortest latency, pipelining is the only way to maximize throughput, and the iterative units

Table 1 Implementation results for the Spartan-6 device (XC6SLX75-3)

	Available	AC	AHP	AHP_B	AFP	AFP_B	AI	SC	SHP	SFP	SI
Slice registers	93296	256	1536	256	2944	1664	817	256	4224	16768	806
Slice LUTs	46648	8997	9087	3946	8884	3376	1367	16888	15523	22029	1566
Slices	11662	2680	2529	1324	2352	1216	493	5243	4590	6629	536
RAMB8s	344			80		86					
F_{max}[MHz]		24.4	195	154	215	168	160	7.95	196	169	180
Latency [Tclk]		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
Throughput [Gbps]		3.05	24.4	19.3	26.8	20.9	1.81	0.99	24.5	21.1	0.66
Mbps / Slice		1.17	9.87	14.9	11.7	17.6	3.77	0.19	5.46	3.26	1.26
Max path: logic/routing [%]		21 79	28 72	38 62	30 70	38 62	21 79	15 85	25 75	18 82	32 68

Table 2 Implementation results for the Spartan-3 device (XC3S2000-5)

	Available	AC	AHP	AHP_B	AFP	AFP_B	AI	SC	SHP	SFP	SI
Slice registers	40960	271	5061	2771	3913	3913	781	256	4224	16768	783
Slice LUTs	40960	34566	30426	25328	29976	24583	7986	18939	22708	26876	3995
Slices	20480	17428	18799	14274	16103	13220	5948	9900	11793	18377	2145
RAMB16s	40			20		20					
F_{max}[MHz]		11.8	83.5	77.0	106	101	77.0	6.35	143	125	96.2
Latency [Tclk]		1	11	11	11	11	11	1	32	34	34
Latency [ns]		84.8	132	143	104	109	143	158	224	272	353
Throughput [Gbps]		1.47	10.4	9.62	13.2	12.6	0.88	0.79	17.9	15.6	0.35
Mbps / Slice		0.09	0.57	0.69	0.84	0.98	0.15	0.08	1.55	0.87	0.17
Max path: logic/routing [%]		27 73	24 76	30 70	19 81	34 66	28 72	30 70	33 67	28 73	31 69

are unsurpassed if smallest possible resource utilization, at the cost of low performance, is needed. It is worth noting, however, that from all the 4 architectures applied to the two ciphers only the two pipelined AES organizations were implemented with the use of block RAM resources in both Spartan-6 and Spartan-3 chips. In Spartan-6 this 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. In cases of other architecture / cipher combinations the implementation tool did not choose to use block RAM units, although the VHDL code did include templates of ROM definitions (for SBox specification) and they were properly detected in reports of the synthesis tool.

Looking at Figs. 3 and 4 we can better evaluate the results and see some remarkable relations. Comparing the effectiveness of AES and Serpent implementations in Spartan-6 it is seen that the AES is able to achieve notable better performance with significantly lower resource utilization: in combinational organization the AES reaches 307% of the Serpent's throughput with 51% of its slice size, while for the fully pipelined and iterative architectures these numbers are, respectively, 127% - 35%, and 275% - 92%. In Spartan-3 family, on the other hand, the relation is different: although generally the AES is able to reach higher levels of throughput (with one exception of the fully pipelined designs), its size is also bigger (again with exception of the xFP units). For combinational, fully pipelined and

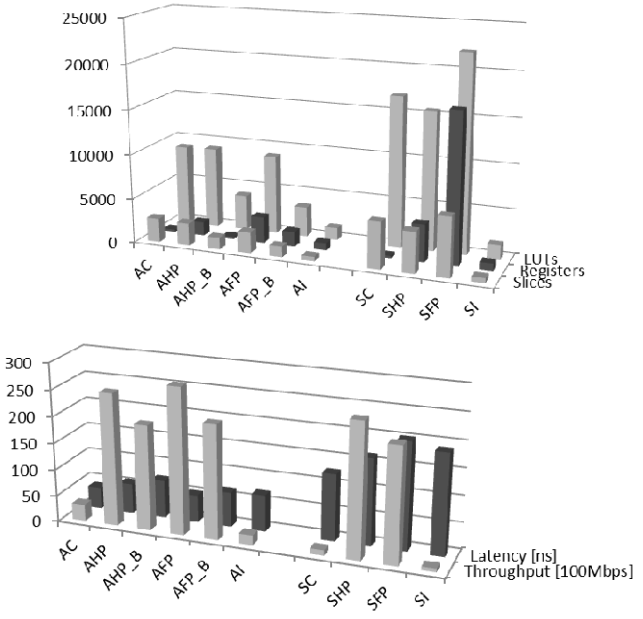

Fig. 3 Size and performance of AES and Serpent implementations in a Spartan-6 device

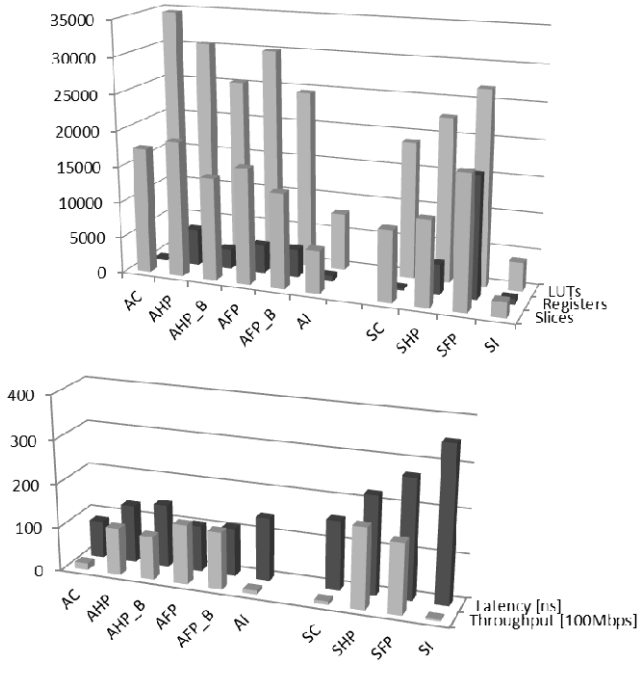

Fig. 4 Size and performance of AES and Serpent implementations in a Spartan-3 device

iterative architectures throughput and slice size ratios are, respectively, 186% - 176%, 84% - 88%, and 247% - 277%. This lead to conclusion that the new architecture of Spartan-6 family is better suited for implementation of the AES operations than the previous one.

This observation is confirmed by Fig. 5. In this graph we visualize size and performance not across different architectures implemented on the same platform but between the two platforms. What is instantly seen from the bars in the last row (slice ratio) is that while for AES the size ratios are in the range from 6.5 to 12 (meaning that the slice size is bigger by this factor in Spartan-3 than in Spartan-6), for Serpent these ratios are from 1.9 to 6 – so switching form Spartan-3 to Spartan-6 is much more beneficial for the AES implementations. Also the throughput ratio is in the range of 0.4 ÷ 0.6 for AES and 0.5 ÷ 08 for Serpent, indicating that the new family brings more progress for the Rijndael cipher.

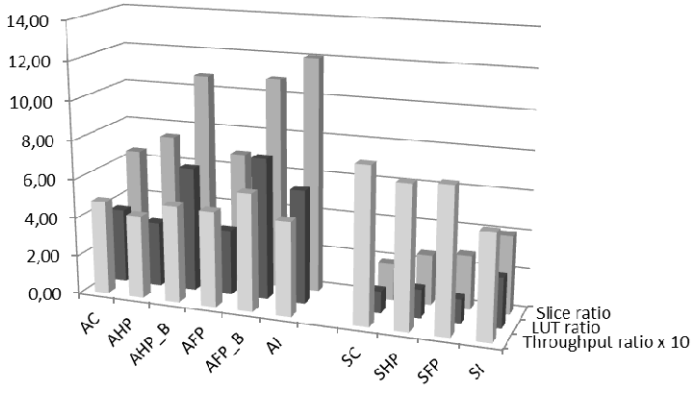

Fig. 5 Spartan-3 vs. Spartan-6 – ratio of performance and size parameters

This difference between the two methods can be explained looking at the most resource-hungry elementary operation in FPGA: the substitution function. In Spartan-3 every output bit of the AES SBox, being an 8-input function, requires 256 / 16 = 16 4-input LUTs for storing the substitution table plus some additional LUTs for multiplexing their outputs (in terms of ROM organization: for address decoding). Even not counting the extra multiplexing logic this needs 128 x 16 = 2048 LUTs in each round and 20480 LUTs in the entire 10-round cipher (vs., for example, total of 40960 LUTs in the whole mighty XC3S2000 chip). On the other hand, LUT elements in Spartan-6 are truly 6-input tables (but can also be configured as two 5-input LUTs what provide some amount of flexibility) so every SBox output is generated by 256 / 64 = 4 LUTs with much simpler multiplexing (which can be done in dedicated fast multiplexers and not in LUTs) and the whole ciphers needs 4 x 128b x 10 rounds = 5120 LUTs. In Serpent, in contrast, the SBoxes are 4-input functions so they perfectly fit already in Spartan-3 LUTs and moving to Spartan-6 does not bring any improvements in this aspect – in fact, Spartan-6 LUTs generating Serpent SBox functions are utilized not to their full potential.

5 Conclusions

We take for granted that new generations of FPGA chips bring larger sizes and faster operation but new architectural developments can sometimes change more than mere design performance and utilization parameters. In case of AES and Serpent ciphers new organization of array resources that was introduced with Spartan-6 family, especially larger Look-Up Tables used for generation of combinational functions in the design, substantially changed feasibility of various implementation options of the ciphers.

In the old Spartan-3 devices combinational and pipelined organizations of the AES units were unacceptable resource hungry and Serpent, despite much higher number of rounds, was a better option for these kinds of processing. The new Spartan-6 chips changed this situation and, effectively, advantage of the Serpent algorithm is again mainly in its better cryptographic strength.

References

[1] Anderson, R., Biham, E., Knudsen, L.: Serpent: A Proposal for the Advanced Encryption Standard. In: The First Advanced Encryption Standard (AES) Candidate Conference, Ventura, California, August 20–22 (1998), http://www.cl.cam.ac.uk/~rja14/serpent.html (accessed March 2012)

[2] Anderson, R., Biham, E., Knudsen, L.: Serpent and Smartcards. In: Quisquater, J.-J., Schneier, B. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 246–253. Springer, Heidelberg (2000)

[3] Anderson, R., Biham, E., Knudsen, L.: The Case for Serpent. In: Third AES Candidate Conference (AES3), New York, USA, April 13–14 (2000), http://csrc.nist.gov/archive/aes/index.html (accessed March 2012)

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

[5] Gaj, K., Chodowiec, P.: Comparison of the hardware performance of the AES candidates using reconfigurable hardware. In: Third AES Candidate Conference (AES3), New York, USA, April 13–14 (2000), http://csrc.nist.gov/archive/aes/index.html (accessed March 2012)

[6] Krukowski, Ł., Sugier, J.: Designing AES cryptographic unit for automatic implementation in low-cost FPGA devices. Int. J. Critical Computer Based Systems 3(1/2/3), 104–116 (2010)

[7] Lázaro, J., Astarloa, A., Arias, J.R., Bidarte, U., Cuadrado, C.: High Throughput Serpent Encryption Implementation. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 996–1000. Springer, Heidelberg (2004)

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

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

[10] National Institute of Standards and Technology, Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197 (2001), http://csrc.nist.gov/publications/PubsFIPS.html (accessed March 2012)
[11] Osvik, D.A.: Speeding up Serpent. In: Third AES Candidate Conference (AES3), New York, USA, April 13–14 (2000), http://csrc.nist.gov/archive/aes/index.html (accessed March 2012)
[12] Piwko, K.: Hardware implementation of cryptographic algorithms in programmable logic devices. Dissertation for M.Sc. degree, Wrocław University of Technology, Faculty of Electronics (2010)
[13] RSA Laboratories, DES Challenges (1997-1999), http://www.rsa.com
[14] Sugier, J.: Low-cost hardware implementation of Serpent cipher in programmable devices. In: Monographs of System Dependability Technical Approach to Dependability, vol. 3, pp. 159–172. Publishing House of Wrocław University of Technology (2010)
[15] Sugier, J.: Implementing Serpent cipher in field programmable gate arrays. In: The 5th International Conference on Information Technology, ICIT 2011, Amman, Jordan, May 11-13, pp. 91–96 (2011)
[16] Wójcik, M.: Effective implementation of Serpent algorithm. Dissertation for M.Sc. degree, Faculty of Electronics and Information Technology, Warsaw University of Technology (2007)