**Sugier Jarosław**
*Wrocław University of Technology, Wrocław, Poland*

# Efficiency of FPGA architectures in implementations of AES, Salsa20 and Keccak cryptographic algorithms

## Keywords

block cipher, hash function, hardware implementation, loop unrolling, pipelining, FPGA

## Abstract

The aim of this paper is to test efficiency of automatic implementation of selected cryptographic algorithms in two families of popular-grade FPGA devices from Xilinx: Spartan-3 and Spartan-6. The set of algorithms include the Advanced Encryption Standard (AES) used worldwide as a symmetric cipher along with two hash algorithms: Salsa20 (developed with ECRYPT Stream Cipher Project) and Keccak permutation function (core of the new SHA-3 standard). The ciphers were expressed in 5 architectures: the basic iterative one (one instance of a round in hardware) and its four derivatives created by loop unrolling and pipelining. With each of the architectures implemented in both Spartan devices this gave the total of 30 test cases, which, upon automatic implementation, created a comprehensive and consistent base for comparison of the ciphers, applied architectures and FPGA devices used for implementation.

## 1. Introduction

Safe and reliable operation of contemporary complex technical systems very often rests on secure and dependable data acquisition, transmission or storage, hence cryptography methods have recently become an indispensable element in design and analysis of system safety and stability. Intensive research activity in this area led to development of a number of methods which are used in symmetric or asymmetric cryptography, digital fingerprinting or message authentication. If very high processing speed is required these algorithms need to be implemented in dedicated, often configurable hardware.

The aim of this paper is to test efficiency of automatic implementation of selected cryptographic algorithms in two families of popular-grade FPGA devices from Xilinx: more mature and established Spartan-3 ([15]) and newer, more advanced Spartan-6 ([16]). The set of cryptographic algorithms include the Advanced Encryption Standard (AES) – a symmetric block cipher along with two hash algorithms: Salsa20 and Keccak-*f*[400].

In the literature there are many proposals for efficient hardware implementations of these particular algorithms ([1], [4]-[8], [10]-[14], [17]). This paper does not aim at supplementing these kind of efforts. Instead, the goal of this work is to explore essential properties of the ciphers when they are implemented in an FPGA device in the basic iterative organization – one cipher round instantiated in hardware – along with other variants created by loop unrolling, optionally with pipelining.

Each of the three algorithms was realized in five architecture variants and all of them were implemented on two hardware platforms, so in total 30 test cases were investigated. This created a comprehensive and consistent base for comparison of the ciphers, architectures and FPGA devices.

The text is organized as follows. In the next chapter operation of the three algorithms is briefly discussed in the light of their further implementation in hardware. Then, in chapter 3 we will present the selected set of five organizations and comment on their specific realization for each of the algorithms. Finally, in chapter 4 parameters of the 30 test cases obtained after implementation will be presented which will provide a base for discussion of size, performance and efficiency characteristics of the algorithms in different hardware configurations.

## 2. Presentation of the algorithms

In the following discussion we will use the '⊕' and '<<' symbols for elementary operations executed

over bit vectors: bitwise exclusive or (xor) and left rotation by a given number of bits. Furthermore, $n_r$ will denote number of rounds repeated iteratively in the complete loop of the cipher.

## 2.1. Advanced Encryption Standard (AES)

The algorithm, which is formally defined in [9], belongs to a class of symmetric block ciphers, i.e. it uses the same *secret key* to both encryption and decryption of a fixed-size block of data – so called *state*. In this work we investigate the most widely used AES-128 version where both the data and the key are 128b long. From functional point of view organization of the cipher is a substitution-permutation network which processes the state in a series of 10 almost identical rounds. Each round uses its own key which is generated from the user-supplied external key by a separate *key expansion.* Data encoding and key expansion share very similar set of elementary transformations and constitute two 128b-wide processing paths which needs to be executed in parallel one along another.

The state is interpreted as 4x4 array of bytes and the round apply four elementary transformations upon it in the following order (see the right part of *Figure 1*):

- substitution *SBox*() where each byte of the state is replaced by another one according to a specific invertible static transcoding function;
- row shifting *SR*() where each the *k*-th row ($k$ = 0…3) of the state array is rotated by $k$ columns to the left in encryption or to the right in decryption;
- column mixing *MC*() operating on whole state columns rather than on individual bytes and calculating its result through an involved series of shift and xor operations (which models polynomial multiplication modulo $x^4 + 1$ over GF($2^8$) );
- key mixing where the round key is simply xor'ed bit-by-bit over the state vector.

The complete encryption path consist of one introductory round which is followed by 10 regular ones (of which the last one is slightly modified). Let:

$P$ – a 128b plaintext (the input),
$B_i$ – a state block that enters the $i$-th regular round $R_i$ ($i$ = 0 denotes the introductory round),
$K$ – external user key,
$K_i$ – the key generated in the expansion for round $i$,
$C$ – encoded ciphertext (the output).

The whole encoding can be expressed in the following way:

$$B_1 = P \oplus K$$
$$B_{i+1} = MC(\, SR(\, SBox(\, B_i \,)\,)\,) \oplus K_i, \quad i = 1 \ldots 9$$
$$C = SR(\, SBox\,(\, B_{10}\,)\,) \oplus K_{10}$$

Additionally to this, the keys $K_i$ need to be generated from the main key $K$ by another computations which, in turn, operate on 32b words $w_i$, $i$ = 0..43. Initially, the first four words are filled with bits from the user key:

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

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

$$w_{4i} = SBox(\, w4_{i-1} << 8\,) \oplus \mathrm{Rcon}[\, i\,] \oplus w_{4i-4}$$
$$w_{4i+1} = w_{4i} \oplus w_{4i-3}$$
$$w_{4i+2} = w_{4i+1} \oplus w_{4i-2}$$
$$w_{4i+3} = w_{4i+2} \oplus w_{4i-1}$$
$$K_i = \{w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}\}$$

Here 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. Data flow is visualized in the left part of *Figure 1*.

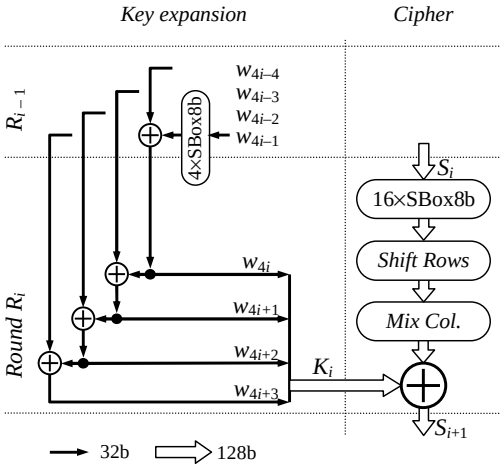


*Figure 1.* Data flow of one AES round: key expansion and cipher paths

Compared to other algorithms discussed in this paper the AES has the largest diversity of internal processing with additional hardware required for key expansion introducing extra complications. Although the state size is 128b, together with key expansion words the entire data path is 256b wide.

## 2.2. Salsa20 hash function

At its core the Salsa20 ([2]) is essentially a 512b hash function, i.e. Salsa20( **x** ) is a 512b hash value computed for the input **x** of the same size. Internally the computations are executed over the 512b state **q** which is divided into 16 x 32b words: **q** = ($q_0$, $q_1$, … $q_{15}$). The state is transformed in 20 rounds with different permutations of the state words passed as

the input to even- and odd-numbered rounds but otherwise the in-round processing is identical.
In Salsa20 there is no key expansion path; instead, the external key is embedded into the input **x** producing directly half of the 16 state words and then the 512b vector is processed in its entirety. Moreover, in the entire algorithm only the following three basic transformations are used, all operating on 32b words:
- bitwise exclusive or (xor) of the two words;
- arithmetic sum of the two words taken mod $2^{32}$, denoted as +;
- rotation of a word to the left by some given (constant) number of positions.

The basic organizational unit of the cipher is *a quarterround* function which transforms a group of four state words: *quarterrorund*($w_0$, $w_1$, $w_2$, $w_3$) = ($w_0$', $w_1$', $w_2$', $w_3$') in the following way:

$$w_1' = w_1 \oplus ( (w_0 + w_3) << 7 )$$
$$w_2' = w_2 \oplus ( (w_1' + w_0) << 9 )$$
$$w_3' = w_3 \oplus ( (w_2' + w_1') << 13 )$$
$$w_0' = w_0 \oplus ( (w_3' + w_2') << 18 )$$

The flow of data which results from the above equations is graphically visualized in *Figure 2*.

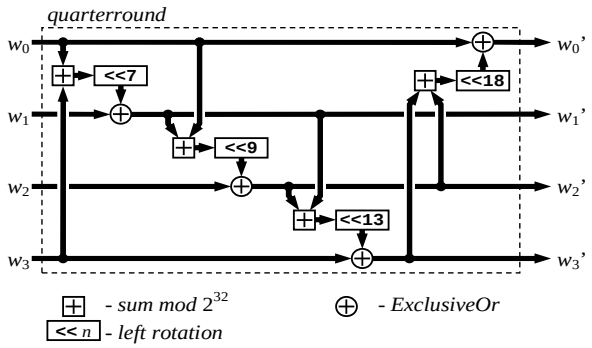


*Figure 2.* Transformations of the state words in the Salsa20 quarterround block

Four quarterrounds operating in parallel transform the entire state **q** and constitute a single round of the cipher. Depending on its input, a round can be one of the two kinds: a *row round* or a *column round*. A row round function is defined as *rowround*( **q** ) = **q**' such that

$$(q_0', q_1', q_2', q_3') = quarterround(q_0, q_1, q_2, q_3)$$
$$(q_5', q_6', q_7', q_4') = quarterround(q_5, q_6, q_7, q_4)$$
$$(q_{10}', q_{11}', q_8', q_9') = quarterround(q_{10}, q_{11}, q_8, q_9)$$
$$(q_{15}', q_{12}', q_{13}', q_{14}') = quarterround(q_{15}, q_{12}, q_{13}, q_{14})$$

whereas a column round function is defined as *columnround* ( **q** ) = **q**' such that

$$( q_0', q_4', q_8', q_{12}' ) = quarterround( q_0, q_4, q_8, q_{12} )$$
$$( q_5', q_9', q_{13}', q_1' ) = quarterround( q_5, q_9, q_{13}, q_1 )$$
$$( q_{10}', q_{14}', q_2', q_6' ) = quarterround( q_{10}, q_{14}, q_2, q_6 )$$
$$( q_{15}', q_3', q_7', q_{11}' ) = quarterround( q_{15}, q_3, q_7, q_{11} )$$

A column round followed by a row round make up so called *double round*:

$$doubleround( \mathbf{q} ) = rowround( columnround( \mathbf{q} ) )$$

and the entire hash is computed by applying ten times the double round to the input **x** and then by adding the result:

$$Salsa20( \mathbf{x} ) = doubleround^{10}( \mathbf{x} ) + \mathbf{x}$$

Compared to the AES Salsa20 has much simpler elementary operations: all of them operate bit-wise on 32b words only and, in particular, they do not involve 8b substitution boxes. Also uniformity of the rounds is very good with no need to expand the key. Nevertheless, with 512b wide data path and 20 rounds the entire cipher is significantly bigger in size.

## 3. Keccak-*f*[400] permutation function

The Keccak algorithm – or, more precisely, the family of 7 different in size Keccak algorithms – are built around Keccak-*f*[*b*] *permutation functions*: for parameter $l$ = 0, 1, … 6 each function operates on a state $A$ consisting of $b = 25 \times 2^l$ bits ($b$ = 25, 50, 100, 200, 400 800, or 1600) where a single word of $w = 2^l$ bits length is called *a lane*. Every function computes its result processing the state in a series of $n_r = 12 + 2l$ rounds ($n_r$ = 12, 14, 16, 18, 20, 22, or 24). In this work we include Keccak-*f*[400] in the comparison – with 20 rounds, 16b lanes and 400b state.
The reference specification in [3] describes one round of Keccak-*f*[b] as a sequence of 5 transformations:

$$\mathtt{Round} = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

Computing the permutation is equivalent just to applying the round function $n_r$ times:

```
Keccak-f[b]( A )
{
  for i = 0 to n_r – 1
    A = Round[b]( A, RC[ i ] );
  Return A;
}
```

where RC[ *i* ] are *w*-bit constants that are generated by specific binary linear feedback shift register (LSFR) defined in the specification. Operation of a single round is specified in a simple pseudo-code as follows:

```
Round[b]( A, RC )
{
  -- θ step
  for x = 0 to 4
    C[x] = A[x,0] xor A[x,1] xor A[x,2]
                  xor A[x,3] xor A[x,4];
  for x = 0 to 4
    D[x] = C[x-1] xor (C[x+1] << 1);
  for y = 0 to 4
    for x = 0 to 4
      A[x, y] = A[x, y] xor D[x];
  -- ρ and π steps
  for y = 0 to 4
    for x = 0 to 4
      B[y, 2*x + 3*y] = A[x, y]<<r[x, y];
  -- χ step
  for y = 0 to 4
    for x = 0 to 4
      A[x, y] = B[x, y] xor
          ((not B[x+1, y]) and B[x+2, y]);
  -- ι step
  A[0,0] = A[0,0] xor RC;
  return A;
}
```

In the above procedure the state *A* is represented as a 5 x 5 array of lanes and three additional auxiliary arrays of lanes: C[0…4], D[0…4] and B[0…4, 0…4] are needed to store intermediate values. Matrix r[*x*, *y*] used in the $\rho$ step provides 25 constant rotation offsets explicitly given in the specification. Furthermore, all index arithmetic is taken modulo 5 and rotating by a positive offset moves each bit in direction of the increasing index.

Although the above pseudo code is relatively simple and makes use only of bit negations, exclusive or operands and rotations, visualisation of the resultant data flow in a diagram similar to that of *Figures 1* or *2* is practically impossible. The 3-dimensional array *A*[*x*][*y*][*z*] is divided in some transformations into so called *planes* (*y* = const), by another ones into *slices* (*z* = const) and by yet another ones into *sheets* (*x* = const) – thus the state processing cannot be decomposed into paths operating on constantly separated words like it was in AES and Salsa20. The only paths that can be traced must be defined down to the level of individual bits – and this makes Keccak flow extremely elaborate.

## 3. Implementing the ciphers in hardware

All the algorithms investigated in this work have strict round-based organization, i.e. they apply their processing by repeatedly executing (almost identical) block of operations (*round*) over some chunk of data (*state*). In hardware implementations this fact can lead to many potential schemes that blend different aspects of iterative, pipelined and combinational architectures. Because the aim of this study was to verify scalability of the algorithms with respect to the number of rounds implemented in hardware, the following 5 organizations was selected for the test suite:

- **x1** – the basic iterative architecture with one round implemented in hardware and the state being passed though it repeatedly in $n_r$ clock cycles (i.e. each complete round is computed in one clock tick);
- **x2** – modification of the above with a combinational cascade of two rounds implemented in hardware with total computation done in $n_r/2$ clock cycles (in each clock tick the state is propagated through two rounds);
- **x5** – as the previous case but with 5 rounds in hardware and $n_r/5$ clock cycles required for complete computation;
- **PPL2** – the modified x2 organization with pipeline registers added after each round: two chunks of data are processed in parallel (twice the throughput) but the completion needs again $n_r$ clock cycles (in one clock tick the state is transformed by one round);
- **PPL5** – the pipelined x5 organization with 5 chunks of data processed in parallel and consequently higher throughput.

The x1 architecture is the one which takes the least amount of hardware resources and will be used as a point of reference in evaluation of the remaining cases. Their size (e.g. number of logic cells used in the FPGA array) should increase proportionally to the number of rounds implemented in hardware:

$$\begin{aligned} Size_{xk} &\approx Size_{x1} \cdot k \\ Size_{PPLk} &\approx Size_{x1} \cdot k \end{aligned} \qquad (1)$$

Additional registers which are added in the pipelined organizations usually do not introduce any extra burden in the FPGA arrays and therefore the above estimations are identical for both x*k* and PPL*k* cases.

Maximum frequency of operation – or the minimum clock period – depends on the other hand on the number of rounds the state must go through in one clock cycle:

$$Tclk_{xk} \approx Tclk_{x1} \cdot k \qquad (2)$$

$Tclk_{\mathrm{PPL}k} \approx Tclk_{\mathrm{x1}}$

### 3.1. AES

Uniformity of iterative processing in this cipher is questioned by the two factors: a) the initial round ($i = 0$) is significantly different form the following ones; b) the last round is slightly modified with one elementary transformation omitted.
Due to the first factor, all the AES architectures have an extra introductory round implemented before the regular loop although it is a very simple one: it consists of just 128b xor logic but it still makes estimation of eq. (1) too restrictive – actual sizes of x*k* and PPL*k* architectures should be somewhat smaller than "$\cdot k$". What is also important, its execution needs a separate clock cycle (which is needed anyway for preparation of the first round key, $K_1$) so the total computation time is $11 \cdot T_{clk}$ for x1 and PPL*k* architectures and 6 or 3 (1 + 10/2 or 1 + 10/5) $T_{clk}$ for x2 and x5 cases.
The second factor – different processing in the last round – requires special multiplexers for bypassing column mixing inside the round hardware, which again weakens the estimations (1) and (2).

### 3.2. Salsa20

Round repetitions are much more uniform in Salsa20 with only exception: the actual fragment of the cipher which is repeated iteratively is a double round (executed 10 times) rather than a single round, because this can be of a column or a row type. In this situation, implementation of a strict iterative scheme "20 repetitions of a single round" would lead to a 512b wide multiplexer which would switch between column and row round inputs, impairing both size and speed of the hardware.
In [13] we have shown that a better alternative is to consider a double round as an elementary unit of the iteration and such an organization – "10 repetitions of a double round" – was adopted in this work to be the basic "x1" architecture with $n_r = 10$. Therefore, the "x2" organization computes the result in 5, while "x5" – in 2 clock cycles. This is on par with latencies of the AES variants but (nearly) doubles the sizes.

### 3.3. Keccak

Compared to the AES and Salsa20, Keccak has the most uniform iteration with the only difference between the rounds in using 20 different 16b constants needed for the $\iota$ step. These constants could be computed on-the-fly by LFSR registers independently for each round instance but it was simpler to tabularize them in distributed ROM modules which, being relatively small, do not add noticeably to the total size but (compared to the LFSR operation) conveniently simplify timing of data distribution. This solution was optimal in both x*k* and PPL*k* architectures.
It should be noted that Keccak has the highest number of rounds in our comparison (20); with $n_r = 10$ AES and Salsa20 require half the iterations.

## 4. Results of the implementation

All 5 architectures of the 3 ciphers were described in the VHDL language at register transfer level (RTL) as closely as possible to the standard specification, using consistent coding style in all the cases. Then, the code was automatically synthesized and implemented in Xilinx ISE software ver. 14.7 with XST synthesis tool, and targeted for two FPGA devices – Spartan-3 (XC3S2000-5, package FGG676) and Spartan-6 (XC6SLX150-3, same package). This gave a total of 30 implementations under the tests.
Devices XC3S2000 and XC6S150 were selected to be sufficiently large to accommodate the most sized x5 or PPL5 architectures. In terms of occupied LUT generators (which is equivalent to the number of logic cells) they took from 15 (Keccak) to 50 (AES) percent of the resources in Spartan-3 chip and from 4 (Keccak) to 13 (Salsa20) percent in Spartan-6 device.
The smallest x1 design, on the other hand, needed just 4 ÷ 21% of Spartan-3 and merely 1.5 ÷ 3.7% of Spartan-6. This shows that size of the FPGA array did not limit the implementations and did not affect the results.

### 4.1. Implementations of the basic iterative architecture

Parameters after implementation of the basic iterative architectures in the two chips are given in *Table 1*. Design sizes are indicated by the number of used Look-Up Tables (LUT); number of registers (flip-flop elements) depend in a very little degree on efficiency of physical implementation and will not be considered in this analysis. The second column additionally lists the number of logic elements found in the longest (lengthiest) path in the design which also determined minimum clock period (given in the third column). Performance parameters were calculated from the value of minimum $T_{clk}$ which was estimated by the implementation tools in static timing analysis of the final, fully routed design.
The figures allows for comparison of the three algorithms and efficiency of their implementations on the two different platforms. As it was already discussed in [10] and [11], the AES in the older Spartan-3 array needs very large amount of LUT

elements for implementation of 8b substitution boxes hence the size of this particular design is exceptionally large, but in Spartan-6 its size is reduced and remains comparable with Keccak.
What is the most significant observation for Salsa20, on the other hand, is that its elementary operations are worst suited for aggregation in LUT elements: processing of one double round needs 102 (Spartan-3) and 50 (Spartan-6) levels of logic versus 3 ÷ 6 levels in AES or Keccak. This also affects performance (by far the lowest operating frequency) and explains why the LUT usage in Salsa is much higher than in Keccak on both platforms.

*Table 1*. Implementation parameters of the basic x1 architecture for the three ciphers and two platforms

| | Number of LUTs | Max. levels of logic | min $T_{clk}$ [ns] | $f_{max}$ [MHz] | Latency [ns] | Throughput [Mbps] |
|---|---|---|---|---|---|---|
| Spartan-3 | | | | | | |
| AES | 8 755 | 6 | 13.1 | 76.3 | 144 | 888 |
| Salsa20 | 3 535 | 102 | 51.9 | 19.3 | 519 | 987 |
| Keccak | 1 777 | 4 | 8.9 | 112 | 178 | 2 242 |
| Spartan-6 | | | | | | |
| AES | 1 400 | 3 | 6.3 | 160 | 69 | 1 860 |
| Salsa20 | 3 367 | 50 | 22.7 | 44.1 | 227 | 2 256 |
| Keccak | 1 339 | 3 | 4.9 | 204 | 98 | 4 090 |

The Keccak algorithm turns out to be the fastest one within this comparison: limited number of logic levels led to the highest frequency of operation which, thanks also to large amount of data processed in the state, gave the best throughput result.
Generally the newer, more powerful and faster Spartan-6 family shows its advantages over the predecessor reducing on average by half the minimum clock cycle: just by moving the same design to the new platform its throughput is doubled.

## 4.2. Scaling effects with increasing number of implemented rounds

In order to evaluate scaling of the algorithms with increasing number of implemented rounds (two in x2 and PPL2 designs or 5 in x5 and PPL5), parameters obtained for those architectures were compared to the estimates from equations (1) and (2). *Table 2* presents quotients of actual parameters and those estimates: 1.0 denotes ideal match, numbers lower than 1.0 – situation when the actual parameter is lower than its estimate, etc. Additionally, the table lists percentage of the longest delay which is generated by logic elements (with the remaining part attributed to routing resources) – this serves as an indication what the extra cost is induced by routing resources in each implementation.

*Table 2.* Speed ($T_{clk}$) and size (number of LUT) of the derived architectures as fractions of the values estimated from the x1 case; plus logic part of the lengthiest path

| | Spartan-3 | | | Spartan-6 | | |
|---|---|---|---|---|---|---|
| | $T_{clk}$ vs Est. | LUT vs. Est. | Longest path - logic [%] | $T_{clk}$ vs Est. | LUT vs. Est. | Longest path - logic [%] |
| AES | | | | | | |
| x2 | 0.80 | 0.61 | 30.5 | 0.78 | 0.84 | 21.8 |
| x5 | 0.67 | 0.36 | 27.9 | 0.76 | 0.71 | 18.6 |
| PPL2 | 1.03 | 0.66 | 26.6 | 0.91 | 0.82 | 25.5 |
| PPL5 | 1.04 | 0.47 | 26.9 | 0.89 | 0.70 | 25.0 |
| Salsa20 | | | | | | |
| x2 | 0.96 | 0.79 | 51.5 | 1.21 | 0.80 | 25.6 |
| x5 | 1.05 | 0.66 | 44.1 | 1.22 | 0.68 | 24.4 |
| PPL2 | 0.96 | 0.79 | 54.6 | 1.27 | 0.82 | 27.5 |
| PPL5 | 1.09 | 0.66 | 47.6 | 1.22 | 0.72 | 31.8 |
| Keccak | | | | | | |
| x2 | 0.89 | 0.79 | 26.0 | 1.10 | 0.81 | 15.6 |
| x5 | 0.85 | 0.66 | 30.4 | 1.62 | 0.54 | 10.2 |
| PPL2 | 0.97 | 0.84 | 30.7 | 1.72 | 0.79 | 14.3 |
| PPL5 | 0.98 | 0.69 | 38.8 | 1.44 | 0.51 | 13.9 |

With respect to these figures AES algorithm behaves in the most predictable way and offers results which are always close to or better than the expectations. In particular we can see that long combinational paths which are present in x2 and x5 organizations made possible efficient optimizations in partitioning of the logic into LUT generators, especially in Spartan-3 arrays. Such an optimization significantly reduced their use: the record is 36% of the expected LUT elements actually used in the x5 case in Spartan-3 (in Spartan-6 optimization is not as spectacular: at most down to 71%). Savings in $T_{clk}$ are unquestionable in x2 and x5 designs but are absent in the pipelined cases implemented in Spartan-3.
The reductions in $T_{clk}$ are not so evident for Salsa20 and Keccak. While in Spartan-3 Salsa20 designs actually do not offer any noticeable improvement

over the estimations (ratios 0.96 ÷ 1.09), for Keccak the x2 and x5 design can reduce clock period to 85 ÷ 89%.

Probably the most striking observation from *Table 2* is that, in contrast to AES, in the newer (and potentially much faster) Spartan-6 family reductions in clock period are negative for both Salsa20 and Keccak. For Salsa20 the actual clock periods are 21 ÷ 27% longer than expected even though at the same time the optimization in LUT usage remains quite good (down to 68 ÷ 82% vs. estimations). This negative tendency becomes dramatic in Keccak: clock periods are by 62% longer than expected in the largest x5 organization and, notably, pipelining added in the PPL5 case is only a partial solution (still an increase by 44%, not seen in any implementation of the two other ciphers).

The problems of Keccak in Spartan-6 should be attributed to routing congestion which is confirmed by comparing presented logic vs. routing ratio in the longest path. With the values of 10.2 ÷ 15.9%, Keccak designs in Spartan-6 have by far the lowest logic parts amongst all the tested cases. Such small values – and, consequently, high values for routing – indicate that configurable connection schemas used in the new Spartan-6 family do not fit particular requirements of propagation rules of Keccak individual bits which were noticed at the end of chapter 2.3. Neither AES nor Salsa20 presented such problems.

### 4.3. Evaluation of the two FPGA platforms

last point of analysis will be devoted to comparison of size and speed metrics between the two hardware platforms for all the 15 designs. *Table 3* presents ratios of numbers of LUT and values of $T_{clk}$ on the two platforms, i.e. the parameter for Spartan-3 was divided by the value for Spartan-6 and the quotient is included in the table.

What becomes evident when looking at the size comparison (the upper half of the table) is that AES is the only cipher that benefits remarkably from moving to the newer Spartan-6 platform: the size is reduced from 6.3 to 3.2 times. In Keccak the reductions are still noticeable although only by factors 1.3 ÷ 1.8. In Salsa20, on the other hand, number of LUT elements remains virtually unchanged with PP5 case being the only one when this number actually increases – and this despite the fact that 6-input LUT generators in Spartan-6 are *much* more powerful than their 4-input counterparts in Spartan-3. This again confirms that this potential of the new platform remains useless in implementation of atomic operations defined for this cipher.

*Table 3*. Ratios of size (number of LUT) and speed ($T_{clk}$) in Spartan-3 vs. Spartan-6 implementations

| | x1 | x2 | x5 | PPL2 | PPL5 |
|---|---|---|---|---|---|
| LUT ( S3 : S6 ) | | | | | |
| AES | 6.25 | 4.58 | 3.17 | 5.07 | 4.23 |
| Salsa20 | 1.05 | 1.03 | 1.02 | 1.00 | 0.97 |
| Keccak | 1.33 | 1.29 | 1.64 | 1.42 | 1.79 |
| $T_{clk}$ ( S3 : S6 ) | | | | | |
| AES | 2.09 | 2.14 | 1.83 | 2.36 | 2.43 |
| Salsa20 | 2.28 | 1.82 | 1.97 | 1.73 | 2.05 |
| Keccak | 1.82 | 1.48 | 0.96 | 1.03 | 1.25 |

Speed comparison adds another evidence of the same problems that plagued Keccak implemented in Spartan-6 and were visible in the previous point. While both AES and Salsa20 organizations reduce their clock periods by 2.43 ÷ 1.73 on the new platform, Keccak demonstrate significant problems with scaling when its size increases. For the x1 and x2 designs the $T_{clk}$ reduction is by 1.82 and 1.48, but in x5 the ratio is smaller than 1 i.e. clock period in Spartan-3 is actually shorter that in Spartan-6. It is a surprising and unusual situation that this design is slower in the new FPGA device than it was in its predecessor.

## 5. Conclusions

In this work we have analyzed efficiency of hardware implementations of three well-known cryptographic algorithms on two FPGA platforms, illustrating potential strengths and weaknesses when basic iterative architecture of any cipher is unrolled, with and without pipelining.

The results show that AES, the oldest of the ciphers, is the one which can be implemented in both Spartan-3 and Spartan-6 devices with the most predictable results. Its realization particularly in Spartan-6 is advantageous because in the older family 8b substitution boxes generate very large amount of resources but still even in such outsized designs in Spartan-3 unrolling and pipelining can be applied with positive and predictable effects.

Slasa20 turned out to be the algorithm with elementary operations which are the most difficult for implementation with LUT generators available in the FPGA array: data path of one double round in this cipher needed 102 (Spartan-3) or 50 (Spartan-6) levels of logic while in the other algorithms – at most 6. This led to large designs (large number of LUT

generators which were utilized to a little degree) and slow timing.

The problem with Keccak, on the other hand, is with routing congestion which start to appear in Spartan-6 devices in bigger (more unrolled) architectures but does not affect Spartan-3 array. As an extreme example, although AES and Salsa implementations are on average twice faster in Spartan-6, Keccak's x5 design runs 4% slower and the pipelined variant – not as much faster.

## References

[1] *ATHENa Database of FPGA Results*, available at http://cryptography.gmu.edu/athenadb/fpga_hash, access date: March 2015.

[2] Bernstein, D. J. (2008). The Salsa20 family of stream ciphers. *New Stream Cipher Designs*. Springer, 84-97.

[3] Bertoni, G., Daemen, J., Peeters, M. & Van Assche, G. (2011). *The Keccak reference*. http://keccak.noekeon.org/, access date: March 2015.

[4] Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R. & Sharif, M. U. (2012). Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs. *The Third SHA-3 Candidate Conference*, Washington, DC, USA.

[5] Gaj, K., Kaps J. P., Amirineni, V., Rogawski, M., Homsirikamol, E. & Brewster, B.Y. (2010). ATHENa – Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs. *20th International Conference on Field Programmable Logic and Applications*, Milano, Italy.

[6] Gaj, K., Southern, G., & Bachimanchi, R. (2007). Comparison of hardware performance of selected Phase II eSTREAM candidates. *Proc. State of the Art of Stream Ciphers Workshop*, eSTREAM, ECRYPT Stream Cipher Project, Report, Vol. 26, p. 2007.

[7] Junkg, B. & Apfelbeck, J. (2011). Area-efficient FPGA implementations of the SHA-3 finalists. *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, IEEE, 235-241.

[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] 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 March 2015).

[10] Sugier, J. (2012). Implementation of symmetric block ciphers in popular-grade FPGA devices. *Journal of Polish Safety and Reliability Association* 3, 2, 179-187.

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

[12] Sugier, J. (2013). Implementing Salsa20 vs. AES and Serpent in Popular-Grade FPGA Devices. *Advances in Intelligent and Soft computing: New Results in Dependability and Complex Systems*. Proc. 8th Int. Conf. Dependability and Complex Systems DepCoS-RELCOMEX, Springer, 224, 431-438.

[13] Sugier, J. (2013). Low-cost hardware implementations of Salsa20 stream cipher in programmable devices. *Journal of Polish Safety and Reliability Association* 4, 1, 121-128.

[14] Sugier, J. (2014). Low cost FPGA devices in high speed implementations of Keccak-f hash algorithm. *Advances in Intelligent and Soft computing: New Results in Dependability and Complex Systems*. Proc. 9th Int. Conf. Dependability and Complex Systems DepCoS-RELCOMEX, Springer, 286, 433-442.

[15] Xilinx, Inc. (2009). *Spartan-3 Family Data Sheet*. www.xilinx.com (ds099.pdf); retrieved March 2015.

[16] Xilinx, Inc. (2011). *Spartan-6 Family Overview*. www.xilinx.com (ds160.pdf); retrieved March 2015.

[17] Yan, J., & Heys, H. M. (2007). Hardware implementation of the Salsa20 and Phelix stream ciphers. *Proc. Canadian Conference on Electrical and Computer Engineering CCECE 2007*. IEEE, 1125-1128.