Sugier Jarosław
Wrocław University of Technology, Wrocław, Poland

Low-cost hardware implementations of Salsa20 stream cipher in programmable devices

Keywords

FPGA, stream cipher, hardware implementation, pipelining, iterative architecture

Abstract

Salsa20 is a 256-bit stream cipher that has been proposed to eSTREAM, ECRYPT Stream Cipher Project, and is considered to be one of the most secure and relatively fastest proposals. This paper describes hardware implementation of various architectures of this cipher in popular Field Programmable Gate Arrays (FPGA). The implemented architectures are based on the loop-unrolled data flow organization and after pipelining they can reach the throughput in the range of 20 – 30 Gbps even after fully automatic implementation in popular low-cost families of Spartan-3 and Spartan-6 from Xilinx. More resource-limited iterative architectures achieve speed of 1 – 2 Gbps. The results that are included in this work present potential of the algorithm when it is implemented in a specific FPGA environment and provide some information for evaluation of cipher effectiveness in contemporary popular programmable devices.

1. Introduction

A hardware implementation is an important option for every cipher and numerous realizations using both mask- (ASIC) and field- programmable gate arrays (FPGA) silicon devices have been developed for AES contest candidates or for algorithms that took part in the ECRYPT project. In this work we present results obtained after implementation of different architectures of Salsa20 stream cipher in popular-grade FPGA devices. While most of the solutions described in the literature are customized for specific device architectures and/or operating environments and they are highly optimized to reach maximum efficiency either in speed or in size, in this work we look from different point at the task of cipher implementation.

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 devices used for implementation are 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.

The text is organized as follows. In the next section we discuss organization of the Salsa20 algorithm as a hash function and as a stream cipher, then we introduce the three basic kinds of cipher implementation in hardware: the combinational, the pipelined and the iterative one, and finally we evaluate the results of automatic implementation of these architectures in the two selected families of FPGA devices.

2. The Salsa20 cipher

Salsa20 family of stream ciphers [2]-[3] has been developed by Daniel J. Bernstein from the University of Illinois at Chicago, USA, in 2005 and submitted to the eSTREAM project. After passing all phases of selection unmodified it has been included in the final portfolio of Profile 1 (software) ciphers along with 4 other proposals.

At its core the Salsa20 cipher is a hash function which operates in the counter mode as a stream cipher: the 64B[yte] input consisting of 32B of the *key* (or twice repeated 16B key) together with 8B *nonce* plus 8B *counter* and 16 constants bytes is

hashed into 64B result which is then XOR'ed with the plaintext. *State* of the cipher is also 64B wide and is represented as a series of 4B *state words*. During decryption the same hash result is XOR'ed with the ciphertext stream to produce plaintext. There is no feedback of the data stream to the hash stream.

2.1. The Salsa20 hash function

The Salsa20 hash function consists in application of 20 rounds which are executed over the state **q** = (q_0, q_0, … q_{15}), where each q_i represents a single 32b[it] state word. Different permutations of the state words are used as input to even- and odd-numbered rounds but otherwise the in-round processing is identical so the whole organization is very uniform. Moreover, in the entire algorithm only the following three basic transformations are used, all operating on the entire 32b words:

- bitwise Exclusive-Or (XOR) of the two words, denoted as ⊕;
- sum mod 2^{32} of the two words, denoted as + (since there is no other kind of addition used here there is no risk of confusion);
- rotation to the left by the given (constant) number of bits, denoted as <<.

In contrast to the contemporary symmetric block ciphers, in Salsa20 there is no key pre-processing path running in parallel with data (cipher) path which would compute a separate key for each round; the user supplied external key is embedded into the input 64 bytes producing directly half of the 16 state words and then the 512b vector is processed in its entirety.
The elementary 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') such that

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

The above equations are given in specific order in which they can be sequentially executed modifying w_i words *in place*: first, w_1 is replaced with the new value w_1' which is computed from the current w_0 and w_3 words, then w_2 is replaced with the new value computed from w_1' and w_0, then w_3 is replaced with the value computed from w_2' and w_1', and finally w_0 is replaced with the value computed from w_3' and w_2'. In this way the quarterround can be implemented in software as a chain of four transformations executed one after another without any temporary registers for intermediate storage of w_i' values. The flow of data which results from the above equations is graphically visualized in *Figure 1*.

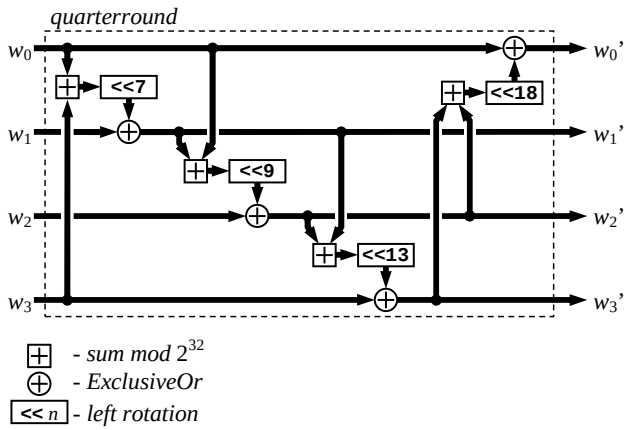

Figure 1. Transformations of the state words in the Salsa20 quarterround function

Four quarterrounds operating in parallel constitute a single round of the cipher and this 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

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

Justification for the round names becomes evident after the quarterround inputs are marked on the 16 words of the state **q** visualized as a 4×4 matrix, as it is done in *Figure 2*. In both cases the four quarterround functions are loaded with, respectively, rows (in a row round) and columns (in a column round) extracted from the matrix and then rotated: the *k*-th row or column (*k* = 0 … 3) is rotated by *k* positions to the left or upwards so that the diagonal elements are given always as the first quarterround argument w_0.
A column round followed by a row round make up a so called *double round*:

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

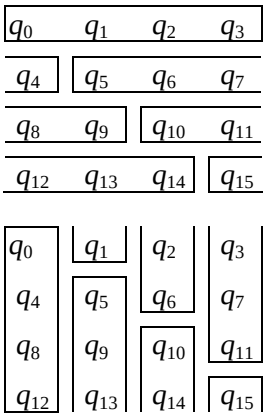

Figure 2. Arrangement of the state words at the inputs to the quarterrounds in a row round (above) and a column round (below)

Having introduced all the above elementary components the final hash function can be defined. In short, to generate the Salsa20 hash value for a 64B input **x**, first the double round is applied ten times to it and then the result is added:

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

Nevertheless, strictly speaking, since all the core definitions operate on a sequence of 4B words, the input **x** as well as the result needs to be transformed using little endian notation (btw note that the '+' sign in the above equation is the sum mod 2^{32} applied on a word-by-word basis). The complete and unambiguous specification of the computational flow for calculation of $Salsa20(\mathbf{x})$ begins with transformation of the 64 bytes into 16 words:

$$x_0 = \mathbf{x}(3\ldots0)$$
$$x_1 = \mathbf{x}(7\ldots4)$$
$$\ldots$$
$$x_{15} = \mathbf{x}(63\ldots60)$$

then states the application of ten double rounds:

$$(y_0, y_1, \ldots y_{15}) = doubleround^{10}(x_0, x_1, \ldots x_{15})$$

and ends with the inverse transformation of the result into 64 bytes which make up the final hash value:

$$Salsa20(\mathbf{x}) = (littleendian^{-1}(y_0 + x_0), littleendian^{-1}(y_1 + x_1), \ldots littleendian^{-1}(y_{15} + x_{15}))$$

2.2. The Salsa20 encryption function

As mentioned in the introduction, the Salsa20 encryption scheme is a hash function operating in a counter mode where the hash result is XOR'ed with plaintext to give ciphertext (during encryption) or with ciphertext to give plaintext (during decryption). This scheme is outlined in *Figure 3*. The simple XOR operation as the final and the only transformation applied to the plaintext makes encryption and decryption equally efficient. It also allows to use the same hash module in both operations what significantly simplifies either software or hardware implementations.

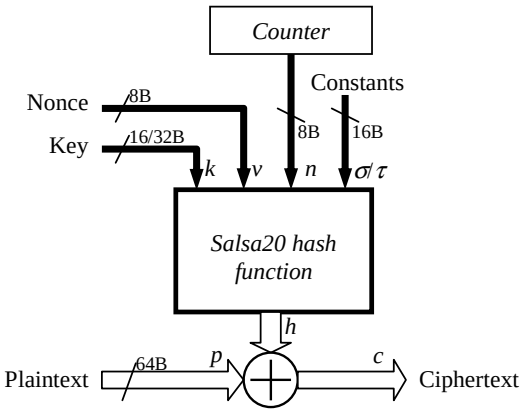

Figure 3. Salsa20 hash function operating as a stream cipher

To create 64 bytes of the input to the hash function, the 16 or 32B *key k*, the 8B *nonce* (*number once* – a unique message identifier) and the 8B *counter n* are expanded with 16 bytes of constants:

$$(\sigma_0, \sigma_1, \sigma_2, \sigma_3) = (0x61707865, 0x3120646E, 0x79622D36, 0x6B206574)$$
$$(\sigma_0, \sigma_1, \sigma_2, \sigma_3) = (0x61707865, 0x3320646E, 0x79622D32, 0x6B206574)$$

which are ASCII encoded strings "expand 16-byte key" and "expand 32-byte key". If the key is 16B the hash value *h* is computed using σ_i constants with the key repeated twice in the input:

$$h = Salsa20(\sigma_0, k, \sigma_1, v, n, \sigma_2, k, \sigma_3)$$

In a case when the key is 32B long it is split into halves $k = (k_L, k_H)$ and supplemented with σ_i constants:

$$h = Salsa20(\sigma_0, k_L, \sigma_1, v, n, \sigma_2, k_H, \sigma_3)$$

Figure 4 visualizes mapping of the encryption input onto the state matrix of the hash function for the case of 32B key. It can be seen that the constant words are placed on the diagonal of the matrix with the two nonce words located in its upper half and the two counter words in the lower; the key words are included in the both halves.

σ_0	k_0	k_1	k_2
k_3	σ_1	v_0	v_1
n_0	n_1	σ_2	k_4
k_5	k_6	k_7	σ_3

Figure 4. Cipher parameters as the 16 words loaded to the hash function (case of a 32B key shown)

Since the counter *n* (which is incremented for each block of the generated cipher) is 64b long, the maximum length of the encoded stream is limited to 2^{64} 64B blocks or 2^{70} bytes (≈1 billion TB). As mentioned before, the Salsa20 decryption function would have exactly the same structure as the one in *Figure 3* but with 64B ciphertext block entering the input and the same size plaintext leaving the output.

3. Implementing the cipher in configurable hardware

Any round-based cipher can be efficiently implemented in software using an iterative scheme where operations of the single round are coded once and then applied to the state variables repeatedly in a loop as many times as required. Parallel execution of multiple program threads which is viable in contemporary multi-core processors can be utilized to speed-up execution of the round provided that its processing can be separated into multiple independent tasks (in Salsa20 the quarterround transformations are ideally suited to such a parallelization).
When transferring the algorithm to hardware the designer is facing a larger diversity of feasible options. In general, the iterative loop can be completely unrolled with all the rounds replicated in hardware as a cascade of modules or it can be unrolled in part (e.g. one fourth of the rounds replicated in hardware with the state signals being passed through four times). Another possibilities comes up if the loop is not unrolled: just one hardware module can be implemented in hardware and its operation on the set of state signals is repeated as many times as there are rounds in the cipher, or the processing of the single round can be divided into multiple execution of a sub-module – again, in the case of Salsa20 these would be the quarterround function – and execution of one round is accomplished in multiple steps – i.e. in multiple clock cycles.
The choice of architecture type usually depends on specifically defined cost criteria and comes from the optimal balance between required speed vs. acceptable size of the hardware: organizations with completely unrolled loop can finish encoding in one clock cycle but for contemporary ciphers with many rounds of complicated processing they usually demand very large (or even huge) amounts of resources. On the other hand, the iterative organizations need to implement in silicon just one round or even a part of it (e.g. one fourth in case of a single Salsa20 quarterround) so the size of the design becomes reduced to a small fraction but the speed of the processing is almost equally decreased.
In this paper, analogously to our previous work [7]-[8], we will examine implementations of the Salsa20 encryption function for the three fundamental types of architectures that are introduced in sections 3.2 – 3.4. Before their presentation, in section 3.1 we will discuss some basic characteristics of the selected hardware platforms.

3.1. Specifics of FPGA implementation

Xilinx, Inc. is the inventor of Field Programmable Gate Array (FPGA) devices and still one of the most successful suppliers of these circuits worldwide. Like in [8], it was decided to choose two popular-grade device families from this manufacturer for case studies included in this work: an older, now more archetypal, Spartan-3 [9] and a newer Spartan-6 [10]. In this point we will concisely discuss the basic aspects of these two architectures that affect efficiency of cipher implementation, in both size and speed characteristics of the resultant hardware.

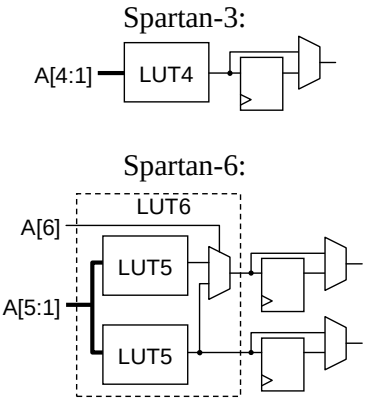

Figure 5. Configurable resources of a single logic cell of Xilinx Spartan-3 and Spartan-6 devices

For a simplified structure of the logic cell in the two Xilinx families please see *Figure 5*. In all FPGA devices from this manufacturer, so called *Look-Up Table* (LUT) is the element located in every 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 presented 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 any 6-input Boolean function but, alternatively, can be configured for generation of any two 5-input functions provided that they share the same set of input variables.

A 6-input / 2×5-input LUT generator found in Spartan-6 may present a significant advantage in implementation of *wide* Boolean functions (i.e. functions of many variables) when compared to 4-input LUTs of Spartan-3. As we have shown in [7]-[8], this can be especially beneficial e.g. in a case of the AES substitution boxes which create an important and substantial part in definition of the cipher. While a Serpent S-Box is a 4-input function which fits efficiently in Spartan-3 LUT elements, AES S-Boxes have 8 inputs and their mapping in Spartan-3 is problematic. In such case switching to Spartan-6 gives noticeable improvements that are not so much evident when moving Serpent to the new architecture.

Additionally, in both 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 registers into the FPGA project, like it is e.g. in pipelined designs, usually can be accomplished at very little additional cost.

In both Spartan families two logic cells represented in *Figure 5* make up *a slice* – an elementary unit of FPGA organization which includes two LUTs and 2 (Spartan-3) or 4 (Spartan-6) flip-flops. Size of any design after its implementation in the array is usually expressed as a number of occupied slices with numbers of utilized LUTs and registers communicated as the two supplementary measures. Moreover, it should be noted that in real designs all resources (all LUTs and all registers) are never actually utilized inside *every* occupied slice; in some of them the registers adjacent to occupied LUTs remain idle or – conversely – only a register is utilized and the related LUT remains unused.

In the following discussion we will present parameters of the designs after their implementation in XC3S2000 (Spartan-3) and XC6SLX75 (Spartan-6) devices. In all cases the VHDL specification was synthesized by the XST synthesis tool in Xilinx ISE 14.4 design suite and then implemented for the given two devices. The implementation was fully automatic, without any hand-made optimizations neither in placement nor in routing. For every design we will specify the basic speed and size parameters: the minimum clock period (T_{clk}) and the maximum operating frequency (f_{max}) as they were estimated by the post-place & route static timing analysis, the latency expressed in clock cycles and in nanoseconds, the overall throughput in Gbps calculated for the f_{max}, size of the design expressed in the number of occupied slices, LUTs and registers, and a synthetic performance measure which is commonly used for estimation of speed vs. size efficiency – Mbps of the throughput per one occupied slice.

3.2. The combinational architecture

In this organization hardware structure closely reflects flow of the data that is being encoded. All 20 rounds of the cipher are implemented as separate hardware modules that create a continuous combinational path from registers on the input of the first round of the Salsa20 hash function to registers on the outputs of the last round. In-between, the design operates as a combinational function that maps 512 input bits (key, nonce, counter and constants) into 512 output bits (hash) thus the reported maximum frequency of operation is actually estimated speed of operation of the entire hash function. In practical applications additional logic would be required for XOR'ing the plaintext with the hash value but this would require so minimal resources that it would not affect the estimated speed of operation and also the increase in resource consumption would be negligible.

The design was specified by porting the specification [2] to the VHDL language using strict RTL style: there were no instances of library elements, no sequential (procedural) descriptions were inserted and no explicit references to any specific hardware attributes were made so that the same code could be synthesized for entirely different device family, even form a different manufacturer. Definition of all internal signals was based on the IEEE standard std_logic_vector type and at the bottom of the design hierarchy there was only one kind of entity – the one with implementation of the quarterround function. Its internal structure followed closely data paths that can be identified in *Figure 1* and, due to very simple basic operations of this cipher, it did not require any sub-modules: both the XOR and the addition of the 32b vectors were done with the IEEE.std_logic_1164 standard operators whereas the rotations were described as a simple bit reordering in the signal vectors and expressed just as concurrent signal assignments that do not require any logic at all (in hardware implementations, as opposed to software realizations, rotations are done exclusively in routing and actually do not require any resources).

After constructing both kinds of the rounds by using 4 direct instances of the quarterround entities, the double round entity was defined and – finally – the 10-element cascade of such entities was constructed with a single for…generate statement in a concise and clear manner.
During implementation in the Xilinx ISE design suite it turned out that the tools are unable to automatically route this design on neither platform (Spartan-3 nor Spartan-6) due to size and complexity of signal connections. The synthesis could be accomplished successfully without any significant messages just like the translate, map and placement steps, but the router tool, after substantially longer running time, quit with a message about a very dense and congested design.

Table 1. Size of the combinational architecture vs. combinational implementations of other ciphers [8]

	Spartan-3		**Spartan-6**	
	Salsa20	**AES**	**Salsa20**	**Serpent**
Slices	14 838	17 428	5 079	5 243
LUTs	28 670	34 566	19 040	16 888

To analyse this situation, *Table 1* shows the size parameters for this architecture as they were reported after the placement step and compares them with equivalent combinational architectures of two other well-known ciphers that were successfully implemented with the same software tools and using the same design methodology in [8]: AES [6] in XC3S2000 and Serpent [1] in XC6SLX75. Although it can be seen that the size of the Salsa20 design expressed in terms of occupied slices and LUT elements is comparable, and actually somewhat lower, than size of successfully implemented combinational organizations of AES and Serpent, in this case complexity and size of the Salsa20 combinational network, undivided into a separate cipher and key paths, turned out to be a prohibitive factor for the router tool. Commenting on processing complexity of these three ciphers it should be noted that the AES consists of 10 rounds working on 128b of data in cipher path and 128b of data in key expansion path (256b in total) and Serpent has 32 rounds working also on 128b in cipher plus 128b in key paths, whereas Salsa20 transforms a single uniform 512b state vector in 20 rounds.
This fact is an interesting observation about complexity of the Salsa20 hash function. Nevertheless, although the combinational architecture could not be automatically implemented in hardware it is included in this discussion because it made a starting point for development of another successful architectures which are described in the following two sections.

3.3. The pipelined architectures

The general idea of pipelining is to introduce registers evenly spaced along the combinational path so that in synchronized operation multiple blocks of data are processed simultaneously one by another inside the pipeline stages at the same time during every clock cycle. In the above defined combinational architecture of the Salsa20 cipher the natural points of placing the pipeline registers are the signals that cross boundaries between the cipher rounds; this transforms each round into a separate pipeline stage. In technical terms such organization can be interpreted as a complete outer loop unrolling and leads to 20 pipeline stages – a valid hash value appears 20 clock cycles after loading the data at the inputs. Although such modification does not improve the latency (the time delay measured from loading the data to reading the result) which can be actually somewhat longer compared to the combinational propagation due to non-zero flip-flop switching time and non-ideal pipelining, the overall throughput of the circuit (amount of data processed in unit time) rises substantially thanks to the simultaneous processing of multiple data blocks in the pipeline stages.
Adding large amount of registers (512 × number of pipeline stages) may seem to be a considerable increase in resource usage but in case of FPGA architectures this increase is easily absorbed by the array. As discussed in section 3.1, in these devices a flip-flop is included in every logic cell right at the output of the combinational configurable element (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 improves their utilization.
In a case study of this paper we have tested two variants of this architecture: pipelining the data path with 10 stages, i.e. with stage boundaries at Salsa20 double rounds, and with 20 stages, i.e. with stage boundaries at every column and row round. The results for both families of FPGA devices can be found in *Table 2*.
First of all it should be noted that after transforming the combinational organization (which could not be implemented) into a pipelined one the implementation completed succesfuly so the table lists also performance parameters which were derived from minimal clock frequency estimated by the static timing analysis of the completely routed

design. At first it may seem to be a perplexing behaviour that enlarging an already huge design by adding a considerable amount of flip-flops made possible its implementation, but this actually confirms that the routing of the previous architecture was impossible due to very long combinational paths that run across all the cipher rounds. After splitting these paths into a number of shorter segments the task of the router was much simpler hence the tool was able to complete it successfully.

Table 2. Parameters of the two versions of the pipelined designs: with 10 and 20 pipeline stages

	Spartan-3		**Spartan-6**	
	P.x10	**P.x20**	**P.x10**	**P.x20**
min T_{clk} [ns]	58.0	24.0	22.7	15.2
f_{max}[MHz]	17.2	41.6	44.0	65.9
Latency [T_{clk}]	10	20	10	20
Latency [ns]	579.9	480.6	227.5	303.4
Throughput [Gbps]	**8.8**	**21.3**	**22.5**	**33.7**
Mbps / Slice	**0.77**	**1.74**	**4.7**	**6.36**
Occupied Slices	**11 539**	**12 254**	**4 765**	**5 307**
Slice LUTs	21 827	21 235	17 268	21 140
Slice Registers	5 888	11 008	5 888	11 008

Secondly, the listed size parameters can be compared to that reported in *Table 1* to see that adding the flip-flops, although in large quantity, did not produce any significant increase in the number of occupied slices and even in case of Spartan-3 device their number has actually decreased by nearly 20%. This observation proves that the regular, round-based structure of Salsa20 transformations, like it was in case of the AES and Serpent ciphers, is very well suited for the pipelining and the additional registers needed for this purpose can be located in the FPGA slices without incurring any increase in the total design size (or even, as in the considered case of Spartan-3, it can help to better utilize the slices).
Finally, the presented parameters allow evaluation of the two pipeline variants: with 10 and with 20 stages. It would be natural to expect that twice shorter pipeline stages in P.x20 organization should give roughly twice shorter clock period and this is the case for Spartan-3 architecture: the decrease is from 58 to 24ns, so it is even greater than expected (evidently the implementation of the P.x10 variant again posed some additional difficulties). Nevertheless the decrease is not so big on Spartan-6 platform: 22.7 vs. 15.2ns is the fall by one third only. Consequently, the absolute values of the latency parameter are in favour of P.x20 on Spartan-3 platform and of P.x10 on Spartan-6. There is no such discrepancy when looking at the throughput values since they will always be doubled for the P.x20 variant and therefore it will always prevail; the difference is further modified by the T_{clk} and f_{max} – widened in Spartan-3 and narrowed in Spartan-6. Since there is no significant differences in design sizes, the Mbps per slice value varies in the same way as the raw throughput.

3.4. The iterative architectures

The iterative architectures investigated in this work were based on one round taken from both versions of the pipelined organizations presented in the previous section, so we will examine two iterative variants: I.x10 and I.x20.
The I.x10 needed 10 clock cycles to complete the processing of a single block before the next one can be loaded but it had implemented in hardware the whole double round, i.e. a column round followed by the row round, each comprising four instances of quarterround entities. Such a module was supplemented with a necessary multiplexing logic (loading the data in – looping back – loading the data out) and a simple controller responsible for counting loop repetitions (round number) and supervising the multiplexers. The controller included just a single "idle/busy" register and a rudimentary round counter; no more complicated extra logic was necessary. Since all the ten repetitions of the double round are strictly identical, such a simple scheme was sufficient.
The I.x20 variant computed one block of data in 20 clock cycles and in general it contained the same simple control logic as the I.x10, but here the 20 iterations were not exactly the same: the even-numbered rounds (when numbered from 0 to 19) should apply the column round transformation while the odd-numbered ones – the row round one. In both cases the needed hardware consisted of just four quarterround modules and they were replicated in hardware only once but also two blocks of extra multiplexing of inputs and outputs were necessary to differentiate permutations of the state words in the even and odd iterations.
Parameters of the two variants obtained after their implementation in the selected devices are included in *Table 3*. One can see that the size difference is far from 1:2 ratio as one could expect: number of slices in I.x10 architecture is greater by only 5-10% despite the fact that it includes implementation of two rounds vs. just one in I.x20. This shows that the overhead introduced with the extra multiplexing on the input and output counterweights the expected savings. The performance comparison is again different on the two hardware platforms. In the older architecture of Spartan-3 the reduction in Ix.20 T_{clk} is almost to the expected 50% so the overall throughput and Mbps

per slice are practically the same. In Spartan-6, on the other hand, we see not so large decrease and therefore the performance is in favour of the Ix.10 variant.

Table 3. Two versions of the iterative architecture: with 10 and 20 repetitions

	Spartan-3		**Spartan-6**	
	I.x10	**I.x20**	**I.x10**	**I.x20**
min T_{clk} [ns]	51.7	24.7	20.8	12.4
f_{max}[MHz]	19.4	40.4	48.0	80.5
Latency [T_{clk}]	10	20	10	20
Latency [ns]	516.8	494.6	208.3	248.5
Throughput [Gbps]	**0.99**	**1.04**	**2.46**	**2.06**
Mbps / Slice	**0.49**	**0.56**	**3.00**	**2.60**
Occupied Slices	**2 036**	**1 858**	**818**	**791**
Slice LUTs	3 374	3 250	2 955	2 611
Slice Registers	1 286	1 294	1 317	1 296

4. Conclusions

Despite fully automatic implementation and very simple specification in the VHDL language, the proposed pipelined architectures of Salsa20 ciphers can reach a throughput levels over 20 Gbps in older Spartan-3 and over 30 Gbps in newer Spartan-6 devices. FPGA implementations described in [4]-[5] and [11] are not directly comparable but have significantly lower speed (e.g. 1.2 Gbps in Spartan-3 device in [4]) and also lower throughput to area ratios (0.74 Mbps/slice in [4] and 0.2 Mbps/slice in a highly iterative architecture proposed in [11]).
Comparing the two variants of pipelined and iterative architectures one can try to recommend the optimal organization with the good balance between speed and size. In case of the P.x10 and P.x20 architectures the latter seems to be a better option: although its latency is bigger on Spartan-6 platform, the significant increase in both throughput and Mbps/slice ratio is a more than an adequate compensation. In case of the iterative architectures, size reduction in I.x20 is not as large as expected and the I.x10 turns out to be a better overall performer, especially in the newer Spartan-6 device.

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] Bernstein, D.J. (2005). The Salsa20 Stream Cipher. *Proc. SKEW - Symmetric Key Encryption Workshop*, Aarhus, Danemark, 26-27 May 2005. Also available at http://cr.yp.to/snuffle.html (accessed April 2013).

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

[4] 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.

[5] Good, T., & Benaissa, M. (2007). Hardware results for selected stream cipher candidates. *Proc. State of the Art of Stream Ciphers Workshop*, 191-204.

[6] 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).

[7] 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.

[8] 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.

[9] Xilinx, Inc. (2009). *Spartan-3 Family Data Sheet*. DS099.PDF, www.xilinx.com (accessed April 2013).

[10] Xilinx, Inc. (2011). *Spartan-6 Family Overview*. DS160.PDF, www.xilinx.com (accessed April 2013).

[11] 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.