Sugier Jarosław

University of Science and Technology, Wrocław, Poland

Implementing SHA-3 candidate BLAKE algorithm in Field Programmable Gate Arrays

Keywords

BLAKE algorithm, FPGA, hash function, implementation efficiency, loop unrolling, pipelining.

Abstract

BLAKE is a cryptographic hash function proposed as a candidate in SHA-3 contest where he successfully qualified to the final round with other 4 candidates. Although it eventually lost to Keccak it is still considered as a suitable solution with good cryptographic strength and great performance especially in software realizations. For these advantages BLAKE is commonly selected to be a hash function of choice in many contemporary IT systems in applications like digital signatures or message authentication. The purpose of this paper is to evaluate how the algorithm is suitable to be implemented in hardware using low-cost Field Programmable Gate Array (FPGA) devices, particularly to test how efficiently its complex internal transformations can be realized with FPGA resources when overall size of the implementation grows substantially with multiple rounds of the cipher running in parallel in hardware and capacity of the configurable array is used up to its limits. The study was made using the set of 7 different architectures with different loop unrolling factors and with optional application of pipelining, with each architecture being implemented in two popular families of FPGA devices from Xilinx. Investigation of the internal characteristic of the implementations generated by the tools helped in analysis how the fundamental mechanism of loop unrolling with or without pipelining works in case of this particular cipher.

1. Introduction

Contemporary Complex Information Systems (CIS) are involved and multifaceted amalgamates of technical, information, organization, software and human resources (users, administrators, technical support, etc.). Complexity and multiplicity of processes, their concurrency and their reliance on the in-system intelligence (human and artificial) significantly impedes construction of strict mathematical models and limits evaluation of adequate reliability measures.

In such context, ensuring appropriate security and confidentiality of information processing (at data acquisition, transmission, storage and retrieval levels) constitute one of the main and one of the most important challenges in design, implementation, maintenance and management of a CIS system. Malfunctions caused by security violations are so common and usual in system operation that in modern dependability analysis they are treated in the same way as the traditional reliability theory considered "classic" failures. To meet the challenge of their eradication it is necessary to apply appropriate cryptographic methods. In this paper we consider one class of such methods which is based on so called *hash functions*.

A hash function, formally, is a computationally efficient function which maps binary strings – called *messages* – of arbitrary length to binary strings of some fixed length, called *hash-values* or *message digests*. To be cryptographically efficient, a given hash function h should meet three essential conditions: a) it should be computationally infeasible to find two distinct inputs which evaluate to the same hash value, i.e. two colliding messages m_1 and m_2 such that $h(m_1) = h(m_1)$; b) it should be computationally infeasible to find a modification of some given message m which does not alters its digest $h(m)$; c) having a specific hash-value H, it should be infeasible to find an input (pre-image) m such that $h(m) = H$. Cryptographic applications of hash functions include message authentication methods (computing and comparing $h(m)$ with secured digest confirms that the message has not been modified), password protection (storing only

password digests instead of explicit passwords eliminates the risk of security breach if the storage is compromised, and is reliably sufficient to accept the password supplied by the user if its digest matches the stored one), or data identification (a relatively short digest can stand for a representation of a much larger data, enabling its faster identification).

In the literature there are many proposals for efficient hardware implementations of the new generation of hash algorithms – including BLAKE and other SHA-3 candidates [3]-[6], [9], [12] – and it was not the aim of this paper to supplement these kind of efforts. Instead, the goal of this work was to explore essential properties of BLAKE when it is implemented in an FPGA device in high-speed architectures. Starting with the basic iterative organization where one cipher round was realized in hardware and the data was repeatedly iterated through, other high-throughput architectures were created with loop unrolling (multiple instances of rounds operated in a cascade which reduced number of iterations) and with pipelining of a partially unrolled loop (registering data at the boundaries of the instantiated rounds allowed for parallel processing of multiple data sets, thus massively increasing the throughput).

Another dimension of the studies was introduced by testing two different devices as platforms for hardware implementation. All explored BLAKE architectures were implemented twice in chips from popular FPGA families manufactured by Xilinx, Inc.: the well-established Spartan-3 and the newer Spartan-6. This created a consistent base not only for evaluation of the concepts of different hardware organizations with various loop unrolling and pipelining mechanisms but also for comparison how the BLAKE's specifics are handled by the implementation tools when the older (Spartan-3) versus the newer, more advanced (Spartan-6) FPGA arrays are used.

Contents of the paper is organized as follows. In the next chapter we will introduce BLAKE hash function and briefly outline its background. Then, in chapter 3, we will present loop unrolled and pipelined architectures of its implementation and discuss specific features of this particular algorithm in FPGA environment. Finally, in chapter 4 we will discuss the results obtained after implementation of the tested architectures in the two FPGA chips and evaluate specific problems which were observed.

2. BLAKE hash algorithm

2.1. Origin: the SHA-3 contest

Hash functions applied in computer systems boast a long history of cryptographic development. The Message-Digest algorithms designed by Ronald

Rivest from MIT in the years 1989-92 were the first widely recognized and standardized methods for improving security of data processing in computer networks. Internet Society published them as official recommendations RFC 1319-21 (MD2, MD4 and MD5) and they soon established a base of reference in further research. In particular, core ideas of MD4 and MD5 methods were adopted in Secure Hash Algorithm (SHA-1) announced a U.S. Federal Information Processing Standard PUB 180-1 in 1995.

Constant increase in available computational power eventually made breaking the SHA-1 security more probable and in 2002 an amended PUB-2 standard was published. It introduced extended versions of the algorithm which became known under common name SHA-2. Still, foreseeing the need of an entirely new approach in hash design in order to compete with recent advances in cryptanalysis, in November 2007 the U.S. National Institute of Standard and Technology announced an open competition for development of a novel SHA-3 standard. Like it was in case of establishing the AES specification, the intention was to find the best possible solution as a result of a free public debate. 14 of the proposed submissions passed the initial verification and in July 2009 they were promoted to the second round where further detailed public examination proceeded. The best 5 algorithms – BLAKE, Grøstl, JH, KECCAK and Skein – were selected for the final round in December 2010 and from this group by decision of NIST KECCAK was selected as the winner.

Although BLAKE eventually lost in the SHA-3 contest the cipher was repeatedly acclaimed in all stages of competition for its good cryptographic strength and great performance especially in software realizations. For these advantages it is still often selected as a hash function of choice in many contemporary IT systems: for example, its variant was chosen as a checksum/validation method in recent extension to RAR file archive format, or it was applied in password-based key derivation function NeoScrypt which is intended to become an informational RFC recommendation.

2.2. Specification of the cipher

According to the official specification [1], construction of BLAKE was built on three well studied and widely accepted concepts: HAIFA round iteration scheme (an improved version of the standard Merkle-Damgård paradigm, providing resistance to long-message second preimage attacks), local wide-pipe internal organization (eliminating local collisions) and a compression function based on a modified Salsa20 stream cipher [2], [7], whose security has been intensively analyzed and proved to

be satisfactory despite relative simplicity of the processing.

In this work we will analyze BLAKE-256 – the variant of the cipher in which size of the words is 32b (leading to 512b state) and which produces 256b digest. To compute the hash, the message m of length $l < 2^{64}$ bits is first padded with bit string "10…01$[l]_{64}$" so that its total bit length is a multiple of 512 (where $[]_{64}$ means 64-bit unsigned big-endian representation). Then the padded message is split into 512b blocks m^0, …, m^{N-1} and the hash value h is computed iteratively in the following pseudo-code:

```
h⁰ = IV
for i = 0, …, N − 1
  hⁱ⁺¹ = compress( hⁱ, mⁱ, s, lⁱ )
return hᴺ
```

where:

IV – initial value of the hash defined as a 256b constant identical to the one in the SHA-2 standard,
s – so called *salt*, a unique 128b string parametrizing particular hash, supplied by the user,
l^i – number of message bits in m^i,
compress(h, m, s, t) – a *compression function* which completely processes one block of data.

2.3. BLAKE compression function

As the pseudo-code in the preceding point shows, processing of each message block consists in application of the compression function and its implementation is the main subject of this paper.

The function uses another 512b constant divided into 16 words $c_0…c_{15}$, and 10 permutations $\sigma_0…\sigma_9$ of the message words $m_0…m_{15}$ – both are statically defined in the specification[1]. The state is organized in a 4x4 matrix of words $v_0 … v_{15}$ which is initially filled with input data, in part xor'ed with constants $c_0…c_7$:

$$
\begin{bmatrix}
v_0 & v_1 & v_2 & v_3 \\
v_4 & v_5 & v_6 & v_7 \\
v_8 & v_9 & v_{10} & v_{11} \\
v_{12} & v_{13} & v_{14} & v_{15}
\end{bmatrix}
$$

$$
=
\begin{bmatrix}
h_0 & h_1 & h_2 & h_3 \\
h_4 & h_5 & h_6 & h_7 \\
s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\
t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7
\end{bmatrix}
\tag{1}
$$

---

[1] To avoid ambiguity superscripts x^i denote ordinal number of the 512b block of the message while subscripts x_i are used to number 32b words within the compression function in the course of processing of some block m^i.

Then the state goes through $n_r = 14$ rounds, with each round applying twice four G_i functions which modify all v_i words:

$$
\begin{array}{ll}
G_0(v_0, v_4, v_8, v_{12}); & G_1(v_1, v_5, v_9, v_{13}); \\
G_2(v_2, v_6, v_{10}, v_{14}); & G_3(v_3, v_7, v_{11}, v_{15});
\end{array}
\tag{2}
$$

and then

$$
\begin{array}{ll}
G_4(v_0, v_5, v_{10}, v_{15}); & G_5(v_1, v_6, v_{11}, v_{12}); \\
G_6(v_2, v_7, v_8, v_{13}); & G_7(v_3, v_4, v_9, v_{14}).
\end{array}
\tag{3}
$$

The first set of the G_i functions operates on words from each column of the matrix, while the second – on words taken from diagonals which corresponds to column and row rounds in Salsa20.

Finally each function $G_i(a, b, c, d)$ for a round number $r = 0…13$ is executed as a sequence of the following assignments:

$$
\begin{aligned}
a &= a + b + (m_{\sigma r'(2i)} \oplus c_{\sigma r'(2i+1)}) \\
d &= (d \oplus a) \gg 16 \\
c &= c + d \\
b &= (b \oplus c) \gg 12 \\
a &= a + b + (m_{\sigma r'(2i+1)} \oplus c_{\sigma r'(2i)}) \\
d &= (d \oplus a) \gg 8 \\
c &= c + d \\
b &= (b \oplus c) \gg 7
\end{aligned}
\tag{4}
$$

where $r' = r \bmod 10$ and the operators denote the following transformations of the state words:

$\oplus$ - bitwise xor of two bit vectors,
$+$ - addition mod 2^{32} of two vectors which are interpreted as positive numbers in natural binary code (i.e. regular 32b addition ignoring carry out),
$\gg$ - right rotation of the word by a constant number of positions.

After 14 iterations, the state produced by the last round is xor'ed with the input h and the salt s to give the return value of the compression h':

$$
h'_i = h_i \oplus s_{i \bmod 4} \oplus v_i \oplus v_{i+8}, \quad i = 0 … 7.
\tag{5}
$$

3. Implementing BLAKE in hardware

3.1. Organization of the processing

Any round-based cipher – including BLAKE – can be efficiently implemented in a CPU-based digital system in software using an iterative scheme: operations of a single round are expressed in the code once and then applied to the state variables repeatedly in an iterative loop n_r times. Parallel execution of multiple program threads which is viable in contemporary multi-core processors can be utilized to speed-up computation of one round

provided that its processing can be separated into multiple independent tasks. As it is shown in [3], BLAKE was the one of the 5 SHA-3 finalists which benefits to the greatest degree from this kind of processing, with four G_i functions inside the round naturally forming threads of parallel execution.

When transferring the algorithm to hardware the designer is facing a larger diversity of feasible implementation options. In general, there are two opposite extreme approaches: the iterative loop of the cipher can be completely unrolled with all the rounds replicated in hardware as a cascade of n_r modules (leading to a *very* large design), or the loop is not unrolled at all with just one round module implemented in hardware and its operation on state signals is repeated n_r times, i.e. in n_r clock cycles. Furthermore, as a mid-range solution the loop can be unrolled in part: one fourth, for example, of the rounds can be reproduced in hardware and the state signals are passed through four times. In this paper, after universal taxonomy proposed in [4], an architecture with k unrolled rounds will be denoted as xk while the basic iterative one – as x1.

Moreover, in the case of purely iterative organization with the loop not unrolled processing of the single round can be further divided into multiple execution of a sub-module – again, in the case of BLAKE these would be the G functions – and thus computations of one round could be accomplished in four steps, i.e. in four clock cycles, but with only ¼ of the round implemented in silicon. Because the aim of this study was to evaluate high speed variants of cipher organization and to verify scalability of BLAKE with respect to the number of rounds implemented in hardware, such low speed and size limited organizations were not investigated. Instead, the following 7 organizations were 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 14 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 7 clock cycles (in each clock tick the state is propagated through two rounds);

x4 – the cascade is built from 4 rounds and 4 clock cycles are required for complete computation (the final result is taken from the second round in the cascade in order to get $n_r = 3 \times 4 + 2$);

x5 – as the previous case but with 5 rounds in hardware and 3 clock cycles for complete computation (the final result is taken from the fourth round in the cascade and $n_r = 2 \times 5 + 4$);

PPL2 – the modified x2 organization with pipeline registers added after each round: two chunks of data are processed in parallel (increasing the throughput twice) but the completion needs again 14 clock cycles since in one clock tick the state is transformed by one round;

PPL4 – the pipelined x4 organization with 4 chunks of data processed in parallel and consequently higher throughput;

PPL5 – the pipelined x5 architecture analogous to the previous cases.

To implement in hardware one complete round of the compression function – which is the core of all 7 tested architectures – one must first instantiate eight blocks of G functions, each with different permutations of the state words loaded to its inputs. Realization of this function was the central aspect of the whole design.

3.2. Implementation of the *G* function

As it was in case of the quarterround entity in the Salsa20 cipher (which is equivalent to the BLAKE's G function), implementing the G function module followed closely equations (4). The design was specified by porting the equations 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 from a different manufacturer. Because elementary operations of eq. (4) are some standard binary transformations, no sub-modules were required: both the XOR and the addition of the 32b vectors were done with the `numeric_std` 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). As a result, the G function module was a plain 128b in / 128b out combinational circuit (although with quite involved internal structure at the bit level). The data paths of the 32b words which were produced by such approach are illustrated in *Figure 1*.

3.3. Implementing the compression function: specifics of BLAKE data distribution

Because BLAKE processing is very similar to that of Salsa20, implementation of the seven architectures (x1 … PPL5) was done in an analogous way as in [7]. Having constructed the round module with 8 direct instances of the G function entities, the basic

x1 architecture was created by adding just rudimentary control and multiplexing logic, while in all other derived architectures multiple instances of the round were simply instantiated repeatedly.
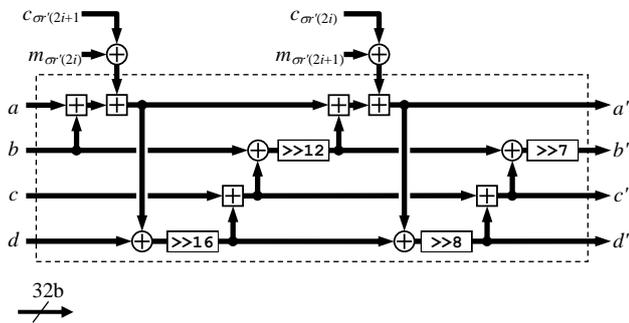

Figure 1. Data paths and elementary operations created by implementation of the *G* function

Unlike Salsa20, though, BLAKE requires entirely different distribution of the message bits among the rounds. In Salsa20 and in majority of other hash functions (including SHA-3 winner KECCAK) the message bits are only loaded as an input to the first cipher round in parallel with other data like salt, counter or nonce. That is, the message bits enter only beginning of the round cascade and are not routed to each round separately. BLAKE, although its authors utilized Slasa20 core transformations, introduced one significant modification: instead of being loaded at the input of the round cascade, the message words are sent to the *G* functions (two words per each function) as the following two equations from the set (4) indicate:

$$a = a + b + (m_{\sigma r'(2i)} \oplus c_{\sigma r'(2i + 1)})$$
$$(\dots)$$
$$a = a + b + (m_{\sigma r'(2i + 1)} \oplus c_{\sigma r'(2i)})$$
$$(\dots)$$

The authors introduced this new aspect as a relatively minor extension and indeed it may be so in software implementation: even if each *G* function operates in a separate thread of CPU execution, extra reads of RAM locations which store the message words did not alter the overall scheme of data handling and just added other operations to the sequence of already running ones. In hardware, though, this led to creation of a completely new, 512b wide data path which was not needed neither in Slasa20 nor in KECCAK organizations [7]-[8]. These paths are presented in *Figure 2* for the case when two rounds of the iteration are unrolled (architectures x2 and PPL2). This effectively doubled the total width of the data path running along the round cascade from 512b (the state) to 1024b (the state plus the message bits).

Distributing the message words was additionally complicated by permutations $\sigma_0 \dots \sigma_9$ used by the algorithm: in each round different permutation of m_i is used so switching between them required supplementary multiplexers controlled by the round counter. For example, in the x1 architecture each *G* function module reads 10 different pairs of m_i words thus in total 16 multiplexers 10:1 needed to be included, each switching 32b words. With loop unrolling the number of multiplexers is proportional the x*k* factor.

In the pipelined organizations further problem arose: since the pipeline works with multiple sets of state data, an analogous pipeline for the message bits had to be created (the right part of *Figure 2*), again doubling the number of registers required for data storage.

Because of involved multiplexing of the message words and their concurrent use it was not possible to use any sort of RAM resources which are available in the FPGA devices. Application of e.g. block RAM modules present in the Spartan families would greatly reduce register utilization but would be possible only in size-limited architectures, e.g. if each *G* function was computed sequentially in consecutive clock cycles (8 clock ticks per round computation) but this is not the direction tested in this work. As a result, all designs prepared in our test suite were fully autonomous, without any external memory requirements.
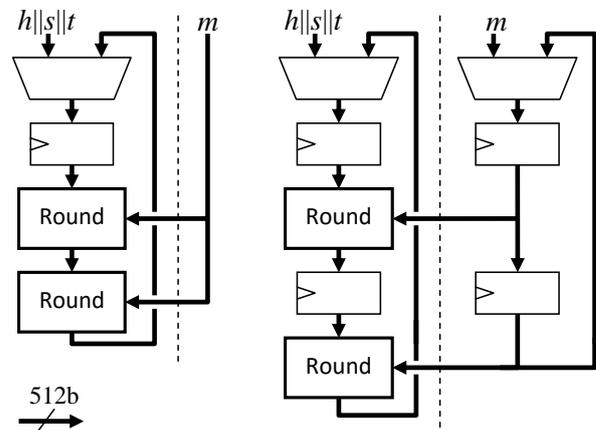

Figure 2. BLAKE specifics: distribution of message bits in, e.g., x2 (left) and PPL2 (right) architectures

4. Results

All seven architectures: the basic iterative x1 plus 3 unrolled ones and their pipelined variants were implemented and tested on two hardware platforms: Spartan-3 and Spartan-6 from Xilinx – the inventor of the FPGA devices and still their leading manufacturer. The source files describing the design were identical for both target platforms. In all cases

the hardware module computing the complete compression function was equipped with basic input / output registers providing means for sequential hashing of the message of arbitrary length in 512b chunks.

The code was automatically synthesized and implemented by Xilinx ISE software with XST synthesis tool, and targeted for two devices – Spartan-3 XC3S5000-5 [10] and Spartan-6 XC6SLX150-3 [11], both in FGG676 package. Devices XC3S5000 and XC6S150 were selected to be sufficiently large to accommodate the most sized x5 or PPL5 architectures. In terms of number of occupied slices (which are elementary configurable blocks in FPGA array, comprising a LUT function generator and a one-bit register) they took max. 70% of the resources in the Spartan-3 chip and max. 43% in the Spartan-6 device. The smallest x1 design, located on the other end of the spectrum, needed 16% of Spartan-3 and 11% of Spartan-6 arrays. This shows that the size of FPGA devices did not limit the implementations and did not affect the results.

4.1. Parameters of the implemented designs

Table 1 presents basic size and performance characteristics of the 7 architectures obtained after their implementation. Speed aspect is represented by the value of the minimum clock period as it was estimated after static timing analysis of the final, fully routed design. From this parameter maximum theoretical throughput of the message stream was derived taking into account number of clock cycles needed to compute the hash and, in case of pipelined variants, number of data streams processed simultaneously in pipeline stages.

Two middle columns provide parameters which illustrate effectiveness (or difficulties) of the implementation process, i.e. how the complex logical transformations of the algorithm were realized with programmable resources of the array: for the longest combinational path in the design the third column gives number of logic elements it contains (LUT generators) and the fourth – percentage of the propagation delay incurred by the routing resources (and not logic elements). Any significant rise in the latter parameter above 50-70% signifies problems with routing of propagation tracks between logic resources in the array: if the connections are too dense vs. distribution of logic elements, routing becomes congested and implementation in the FPGA array becomes problematic.

Size characteristics for each design are reported in the last two columns of the table where total numbers of utilized LUT generators and slices are given.

Table 1. Size and speed of the implemented designs

		T_{clk} [ns]	Throughput [Gbps]	Levels of logic	Routing delay [%]	LUTs	Slices
Spartan-3	x1	45.7	0.80	66	50.6	9155	5415
	x2	88.9	0.82	118	51.2	16928	10039
	x4	190	0.67	203	58.7	32933	19000
	x5	244	0.70	258	61.7	41923	23232
	PPL2	47.3	1.55	69	50.7	16817	10375
	PPL4	44.9	3.26	62	50.4	14591	9821
	PPL5	47.4	3.86	50	55.4	18066	11882
Spartan-6	x1	28.7	1.28	35	64.3	5621	2460
	x2	67.7	1.08	70	72.3	10150	4409
	x4	108	1.18	150	65.6	18994	8396
	x5	185	0.92	131	78.1	24368	8833
	PPL2	30	2.40	38	65.6	10918	4144
	PPL4	33.7	4.33	35	69.0	20185	7246
	PPL5	35.1	5.21	44	68.9	24236	9816

4.2. Performance evaluation

The data from *Table 1* allows to evaluate whether loop unrolling and pipelining have brought expected results in the derivative organizations with regard to their speed and size when the basic x1 design is used as a point of reference. The evaluation will be based on the following simple but rational estimation.

The minimum clock period – hence all performance parameters – in any unrolled or pipelined architecture should depend on the number of rounds the state must go through in one clock cycle. Assuming that in a perfectly regular implementation of xk or PPLk architecture propagation time through each instantiated round remains the same as in x1 case, the expected values of T_{clk} should be:

$$Tclk_{xk} \approx Tclk_{x1} \cdot k$$
$$Tclk_{PPLk} \approx Tclk_{x1} \qquad (6)$$

The same relations should hold also for the number of logic levels included in the longest combinational path of the design.

Using the estimations (6) one can compare them with actual parameters of x2 ÷ PPL5 designs and this is show in *Figure 3*: actual values of T_{clk} and number of logic levels in the longest path were divided by the estimations and the graph shows the quotient.

In Spartan-3 we can recognize that the actual parameters behave in some predictable relation to the estimates. The clock period is very close to expectations: in the unrolled architectures there is

slight increase from 97% (x2) to 107% (x5), so effectiveness of the implementation somewhat decreases with the number of rounds, but still keeping 107% of the expected clock period in a design with so long combinational paths as in x5 is a very good result. Moreover, we can see that increasing the length of combinational paths gives more opportunity for optimization procedures which can pack more logic into LUTs so the number of levels of logic decreases; since this is accompanied with an increase of routing part in the total delay (*Table 1*) these two trends compensate each other.
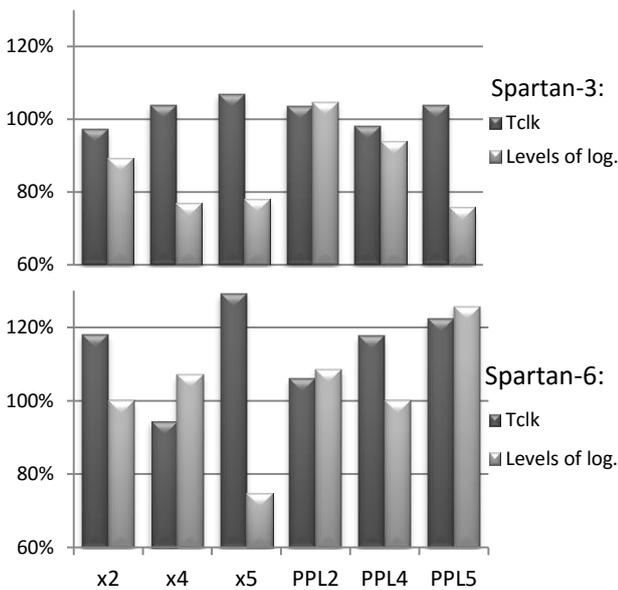

Figure 3. Actual vs. expected performance parameters of the derivative architectures

In the pipelined architectures there is no increase in the longest path which in all cases spans just one round so there is no such evident trend (with the exception in levels of logic which again is better packed in LUTs in larger designs) but the final T_{clk} values remain very close to the estimations.

The situation is not so clear and predictable when the same designs were implemented in the newer Spartan-6 device. The x4 design does not follow the overall trend and shows T_{clk} significantly shorter than expected (apparently this design fit particularly well structure of the array in the XC6SLX150 device) but all other designs reached clock period which are longer than expected, up to 129% in the x5 case. Also it is difficult to see any trend in the "Levels of logic" parameter. This could be related to the fact that routing was not so straightforward in this device: the routing part parameters in *Table 1* approach hazardously high levels of 70% (and in the exceptionally good x4 implementation this value is noticeably better than in x2 and x5) so the

implementation tools struggled to produce expected results.

4.3. Size evaluation

In a manner analogous to that from the previous point, one can estimate expected size parameters (i.e. the number of occupied LUTs or slices) of the derived architectures from the values of x1 case. In both expansion mechanisms – unrolling and pipelining – size of the whole design should increase proportionally to the number of rounds instantiated in hardware so:

$$Size_{xk} \approx Size_{x1} \cdot k$$
$$Size_{PPLk} \approx Size_{x1} \cdot k \qquad (7)$$

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 assumed identical for both xk and PPLk cases. Also the specific additional distribution of the message words to each round, as discussed in chapter 3.3, should scale according to these simple proportions.

On the other hand, the x1 size includes some input / output logic (e.g. the input multiplexers shown in *Figure 2*) which is not replicated with the rounds so the values computed from (7) can be somewhat overestimated.
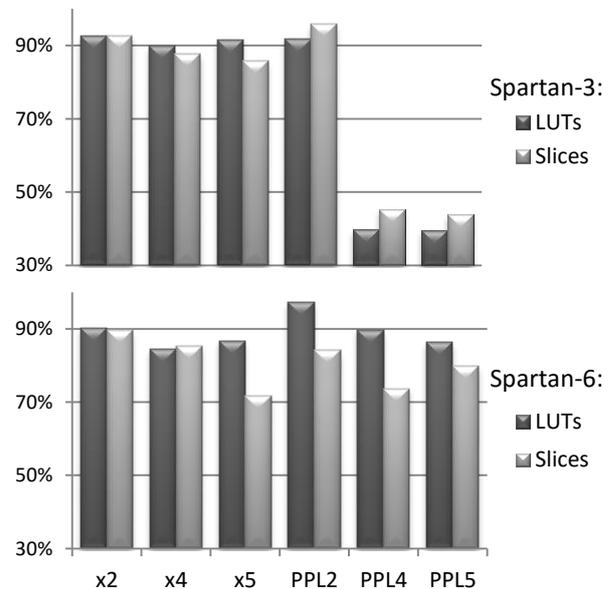

Figure 4. Actual vs. expected size parameters of the derivative architectures

This overestimation is truly confirmed by *Figure 4* where actual numbers of utilized LUTs and slices are compared against the estimates (7): almost all parameters at most oscillate close to 90% and, unlike

in *Figure 3*, none exceeds 97%. Again the graph for the older Spartan-3 family is more consistent: only two largest pipelined architectures were significantly optimized in size but all other ones use 90±6% of expected LUTs or slices. In Spartan-6, like it was already noted for *Figure 3*, routing close to congestion made implementation results less predictable but no bad cases were detected. On the contrary, unlike speed characteristics the size parameters at most are unexpectedly better than estimations with e.g. slice utilization in x5 and PPL4 cases down to 72-74% of the estimates.

5. Conclusions

In this paper we have analyzed high-speed implementations of the BLAKE hash functions based on concepts of loop unrolling and pipelining. The investigated basic iterative architecture can reach 0.8 (Spartan-3) or 1.3 (Spartan-6) Gbps throughput while the largest pipelined one – respectively, 3.9 and 5.2 Gbps. This is on par with other known results, but the aim of this work was not in delivering "yet-another-k%-better" solution. Instead, we have analyzed intrinsic internal characteristic of the implementations generated by the tools to identify how the fundamental mechanism of loop unrolling with or without pipelining works in case of this particular cipher.

Although BLAKE is often considered to be at its core a modification of Slasa20 hash function the introduced extensions are substantially more significant for hardware implementations than for software ones. The additional data paths provided for distribution of message bits to every round instance, which are not seen in Salsa20 or Keccak hashes, made BLAKE implementations more difficult for the tools.

Additionally, we have shown how the two FPGA platforms – Spartan-3 and Spartan-6 – are fit for implementation of this cipher. Although the newer and faster Spartan-6 devices do deliver more capable implementations in terms of raw throughput, routing problems that start to appear on this platform can make the results less predictable than they were in the older Spartn-3 arrays.

References

[1] Aumasson, J. P., Henzen, L., Meier, W. et al. (2010) *SHA-3 proposal BLAKE, version 1.3*, [available at: *https://www.131002.net/blake/blake.pdf*; retrieved March 2016].

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

[3] ECRYPT II Project (2012). *SHA-3 Hardware Implementations*, [available at: *http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations*; retrieved March 2016].

[4] Gaj, K., Homsirikamol, E., Rogawski, M. et al. (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., 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. 26, 2007.

[6] 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.

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

[8] 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.

[9] Tillich, S., Feldhofer, M., Issovits, W., et al. (2009). *Compact hardware implementations of the SHA-3 candidates ARIRANG, BLAKE, Grøstl, and Skein*. IACR Cryptology ePrint Archive, 349.

[10] Xilinx, Inc. (2009). *Spartan-3 Family Data Sheet*, [available at: *www.xilinx.com* (ds099.pdf); retrieved March 2016].

[11] Xilinx, Inc. (2011). *Spartan-6 Family Overview*. , [available at: *www.xilinx.com* (ds160.pdf); retrieved March 2016].

[12] 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.