Journal of Polish Safety and Reliability Association
Summer Safety and Reliability Seminars, Volume 8, Number 1, 2017

Sugier Jarosław
Wrocław University of Science and Technology, Faculty of Electronics, Poland

Memory resources in hardware implementations of BLAKE and BLAKE2 hash algorithms

Keywords

BLAKE hash algorithm, implementation efficiency, memory, loop unrolling, resource utilization

Abstract

In contemporary computer systems security issues are very important for both safety and reliability reasons thus application of appropriate cryptographic methods is a necessity in system design and maintenance. This paper deals with one such method – BLAKE hash function – and investigates its implementation in hardware. The algorithm was a candidate proposed for the SHA-3 contest and, although it was not selected in the final round as the winner, it was very well received for its cryptographic strength and performance, being still used as a hash method of choice in contemporary IT systems. In this paper we discuss a specific modification in hardware realizations of the function which eliminates need for involved data paths distributing message bits among the round units by using auxiliary memory modules for repetitive storage of the message inside each round instance. The idea was implemented in realizations of both BLAKE and BLAKE2 versions of the algorithm in four different organizations: the standard iterative one and three high-speed loop-unrolled architectures with 2, 4 and 5 rounds instantiated in hardware. Together with standard (without RAM) implementations this produced a total of 16 test cases: after implementation in a popular Spartan-3 device from Xilinx their parameters allowed for exhaustive evaluation of the proposed modification. The results reveal that the modification outstandingly enhances size of all the tested architectures: on average, occupation of the FPGA array is reduced at least by half while the improvements in speed, although not so spectacular, are also visible. Additional analyses indicate that the method can also increase overall efficiency of routing, helps in implementation of the loop-unrolled architectures and strengthens optimizations introduced by the BLAKE2 version of the algorithm.

1. Introduction

Although BLAKE eventually lost to Keccak in the SHA-3 competition the cipher is still often selected as a hash function of choice in contemporary data processing systems due to its excellent cryptographic strength and high efficiency in software. Potential of its hardware implementations, like of any other SHA-3 candidate, was extensively studied e.g. in [6]-[8]. In this paper we focus on one particular aspect of BLAKE hardware realizations, dealing with challenges caused by its specific peculiarity: the need of involved distribution of message bits among cipher rounds. Implementation of this distribution is much more cumbersome in hardware than in software and its elimination can significantly reduce FPGA utilization and improve overall performance. The proposed idea consists in replacing the distribution with repetitive storage of the message in RAM modules located within every round instance. In the paper we test this solution in 4 different architectures of the cipher: the standard iterative one and three loop-unrolled organizations with 2, 4 and 5 rounds instantiated in hardware, for both BLAKE and BLAKE2 variants. The results found after their implementation in popular Spartan-3 devices from Xilinx are compared to parameters of analogous architectures implemented without memory so that the savings in array utilization can be measured against supplementary cost of occupied block RAM modules.

The first results investigating potential of the proposed idea were presented in [12]. In this work we extend them by adding the BLAKE2 variant of the algorithm and by introducing exact evaluation of the required memory capacity which is crucial in ASIC (non-FPGA) implementations.

The contents of the paper is organized as follows. After introducing the family of BLAKE hash functions in the next section, in chapter 3 we present standard iterative and loop unrolled architectures used for its implementation, describe the proposed method of RAM application and evaluate required capacity of memory resources needed for its realization. Then in chapter 4 we discuss the results obtained after implementation of the modified architectures and evaluate them against known parameters of the analogous organizations without the modification. The analyses include direct comparison of the architectures before and after the modification as well as its influence on the efficiency of the loop unrolling mechanism and on optimizations introduced in the BLAKE2 version of the algorithm.

2. The family of BLAKE algorithms

In this study we are considering those size variants of the BLAKE algorithms which generate 256b hash output, internally handling 32b words and 512b state: BLAKE-256 and BLAKE2s.

2.1 BLAKE

In BLAKE-256 ([1]) the plaintext message m of length $l < 2^{64}$ bits is first padded with string “10…01$|l|_{64}$” in such a way that its total bit length is a multiple of 512 (where $|l|_{64}$ denotes 64-bit unsigned big-endian representation of the length l). Then the padded message is split into 512b blocks $m^0 \ldots m^{N-1}$ and the hash output $h(m)$ is iteratively computed according to the HAIFA iteration scheme [5]:

```
$h^0$ := IV
for $i = 0 \ldots N-1$
  $h^{i+1}$ := $compress(h^i, s, t^i, m^i)$
return $h^N$
```

The upper indices ^{i} will symbolize ordinal number of the message block and the lower ones – indices of the internal words inside the compression function; moreover, `IV` is a constant pattern initializing the hash chain value h^0 (adopted from the SHA-2 standard), s represents *a salt* (a 128b auxiliary free parameter provided for randomized hashing required e.g. in digital signature schemes), and t^i – a 64b counter giving a number of message bits hashed so far.

Like in other hash algorithms based on Merkle-Damgård construction, processing of a free-length message stream consists in repetitive application of *a compression function compress*() on one message block m^i. Its implementation is the actual challenge in realization of the algorithm. For definition of the function the specification introduces 16 constant words $c_0 \ldots c_{15}$ and ten 16-element permutations $\sigma_0 \ldots \sigma_9$ used for reordering message and constant words in the computations.

Internal processing of the compression is organized around *a state* - a 4x4 matrix of words $v_0 \ldots v_{15}$, and is executed as follows. Initially the state is filled with the current chain hash value h^i, the salt and the counter (but not with the m^i bits!), partially xor’ed with the $c_0 \ldots c_7$ constants:

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} = \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{bmatrix} \quad (1)$$

Then it goes through $n_r = 14$ rounds with each round modifying twice all the words by applying *a G function*:

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

and then

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

Each G_i() function call transforms a set of four v words given as explicit parameters; as an additional side input the message words are also loaded although they do not appear on the argument list. The ordinal function number $i = 0 \div 7$ determines which permutation, message and constant words are used within each specific G_i() instance so that all 16 message words and constants take part in each round. The first set of the functions in eq. (2) operate on words from column of the matrix, while the second in eq. (3) – on words from diagonals, and this corresponds to column and row rounds in ChaCha algorithm where the G function itself is called a quarterround ([3]-[4]).

The function $G_i(a, b, c, d)$ of a round number r $(0 \div 13)$ is defined as a sequence of the following operations:

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

where $r' = r \bmod 10$ (number of the σ permutations is limited to 10) and the operators denote the following transformations of the words:

$\oplus$ - bitwise xor of two bit vectors,
$+$ - addition mod 2^{32} of two bit vectors (i.e. regular 32b addition with carry out ignored),
$>>$ - right rotation by a constant number of positions.
After 14 iterations, the state produced by the last round is xor'ed with the input h^i and the salt s to give the return value of h^{i+1}:

$$h^{i+1} := h_i \oplus s_{i \bmod 4} \oplus v_i \oplus v_{i+8}, \quad i = 0 \dots 7. \qquad (5)$$

2.2 BLAKE2

Extensive tests and cryptanalyses during the SHA-3 contest proved that the original BLAKE proposal offered a very large security margin. In 2013 the authors, based upon experience gathered after the public evaluation, proposed an improved version of the method – called BLAKE2 – with modifications aimed mainly towards its simplification and optimization ([2]). In a brief summary the following changes were introduced (the 256-bit version of the algorithm is considered):
• number of rounds was reduced from 14 to 10;
• message padding was simplified and its functionality was partially replaced with *finalization flags* f_0 and f_1 which signal the last message block and the last node in tree hashing;
• initialization of the h^0 chain value was extended with *parameter block* (which includes, among others, the salt and the finalization flags);
• the state initialization (eq. 1) was changed into:

$$\begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{bmatrix} = \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ \mathrm{IV}_0 & \mathrm{IV}_1 & \mathrm{IV}_2 & \mathrm{IV}_3 \\ t_0 \oplus \mathrm{IV}_4 & t_1 \oplus \mathrm{IV}_5 & f_0 \oplus \mathrm{IV}_6 & f_1 \oplus \mathrm{IV}_7 \end{bmatrix} \qquad (6)$$

• the salt was removed from argument list of the compression function and was kept exclusively for the h^0 initialization;
• definition of the G function was simplified as:

$$\begin{aligned} a &:= a + b + m_{\sigma r(2i)} \\ d &:= (d \oplus a) >> 16 \\ c &:= c + d \\ b &:= (b \oplus c) >> 12 \\ a &:= a + b + m_{\sigma r(2i+1)} \\ d &:= (d \oplus a) >> 8 \\ c &:= c + d \\ b &:= (b \oplus c) >> 7, \end{aligned} \qquad (7)$$

The constants c were dropped altogether from the specification and they are not used neither in (6) nor in (7). Also, because the number of rounds has been cut to 10, there is no need to introduce r' in (7) as the round number modulo 10.
For efficiency of hardware implementations investigated in this paper most of the above changes are of little significance: resources needed for both the h^0 and state initializations are negligible when compared to the hardware representing the actual compression executed in the eight instances of the G functions. Also the reduction in the number of rounds, while obviously cutting the number of clock cycles needed for completion of the computation, can be done with trivial adjustments in the control unit. Nevertheless, removing the c constants indeed to some extent simplifies hardware realization of the G function: keeping a total of 16 words, each 32b wide, in multiple ROM modules and multiplexing them on the two inputs of each G instance is the primary difference between the hardware of BLAKE and BLAKE2 realizations – although it is a relatively minor one looking on complexity of the remaining parts in equations (7).

3. Implementing the algorithms in hardware

3.1 The loop unrolling mechanism

Processing scheme of the BLAKE algorithms is typical to any round-based cipher and it can be efficiently implemented in software in a CPU-based system in an iterative manner: operations of a single round are expressed in the code once and then applied to the state variables v_i repeatedly in a loop n_r times. When transferring the algorithm to hardware (either ASIC or FPGA) 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, 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 (that is, in n_r clock cycles) in a manner resembling software iterations. 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 them four times. In this paper, after universal taxonomy presented e.g. by Gaj et al. ([6]), an architecture with k unrolled rounds will be denoted as xk while the basic iterative one – as x1.
In this study we focus on high speed organizations and the test suite consisted of the following 4 organizations:
• x1: the basic iterative architecture with one round implemented in hardware and the state being passed though it repeatedly in 14 (BLAKE) or 10 (BLAKE2)

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 or 5 clock cycles (with each clock tick the state is propagated through two rounds);
• x4: the cascade is built from 4 rounds and 4 or 3 clock cycles are required for complete computation (the final result is taken from the second round in the cascade to get $n_r = 14 = 3 \times 4 + 2$ or $10 = 2 \times 4 + 2$);
• x5: as in the previous case but with 5 rounds unrolled in hardware; in BLAKE 3 clock cycles are needed for complete computation (the final result is taken from the fourth round in the cascade to get $14 = 2 \times 5 + 4$) while in BLAKE2 the computation takes 2 cycles and the result is taken from the last round.

3.2 Peculiarities of message distribution

Although BLAKE followed the rules of in-round processing of ChaCha cipher ([3]), it introduced significantly different distribution of the message bits among the rounds. In ChaCha and in majority of other hash functions (including SHA-3 winner KECCAK) the message bits which enter the compression are routed only to the input to the first cipher round in parallel with other data like salt, counter or nonce, forming the initial value of the state. That is, the message bits enter only beginning of the round cascade and are not propagated to each round separately: after creating the initial state the message bits are not utilized afterwards. BLAKE uses a different approach: instead of being loaded at the input of the round cascade, the message words are sent to each of the G_i functions (two words per function) as the equations from the set (4) or (7) illustrate.
The authors consider this change as a relatively minor extension of the ChaCha processing scheme. 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 do not alter the overall arrangement of data handling and just adds another operations to the sequence of already running ones. In hardware, though, this means that the message bits must be provided separately to each G_i instance since they take part in the computations throughout all the iterations and not only in their initialization phase. This leads to creation of a completely new, 512b wide data path which has not been needed neither in ChaCha, Salsa20 nor in Keccak as we have analysed in our previous works ([9]-[10]). In effect this doubles total width of the data path running along the round cascade from 512b (the state) to 1024b (the state plus the message bits). This is illustrated in *Figure 1* taking the x4 case of BLAKE as an example; in BLAKE2 the data paths remain identical and only the input parameters of the compression function are different (the salt is replaced with the initialization flags) but this does not affect the considered problem.

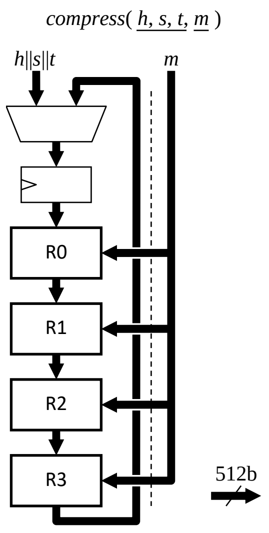

Figure 1. Creating the initial value of the state and distributing the message bits to all the rounds in the BLAKE compression function.

Handling the message words is additionally complicated by permutations $\sigma_0 \ldots \sigma_9$: in each round a different permutation of m_i is used so switching between them requires supplementary multiplexers controlled by the round counter. This aspect is important and will be analysed later in this chapter.

3.3 The proposed application of memory

In order to address the above mentioned problem we propose taking an advantage of block RAM modules available in the FPGA chip which constitute the implementation platform. The main idea presented initially in [12] is to assign one such module for each instance of the G_i function implemented in hardware and keep the m_i words directly within, so that their involved dissemination and multiplexing is avoided. *Figure 2* compares realization of the G_i function in a standard way without RAM (left) and with the proposed extension (right).

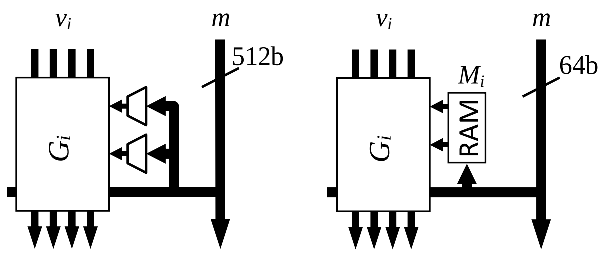

Figure 2. Distributing and multiplexing the message bits for a G_i module in the standard implementations (left) versus storing them in a dual-port RAM unit assigned to the module (right).

Xilinx Spartan FPGA devices which are used in this study offer as additional resources complementing the programmable array so called block RAM modules and these were used for the purpose of message storage in the implementations. Each module can store 16kb of data and can be configured with different depth vs. width ratios – in organizations from 16k x 1b to 512 x 32b. For our application the last case – 512 x 32b – is a suitable one with each m_i word occupying exactly one memory location. Additionally, the modules have full dual port functionality, i.e. their contents can be simultaneously accessible (both for reading as well as for writing) through two equivalent ports. This dual port feature is ideally suited to the needs of the BLAKE distribution: one module can concurrently read two different m_i words in one clock cycle as they are required for computation in one G_i function, and the total number of modules can be reduced by half compared to application of single-port memories. Still, the number of utilized modules is relatively high: every complete round in hardware needs 8 RAM units so their final number in the investigated architectures range from 8 (the case of the x1 organization) to 40 (x5). These figures should be compared to the total of 104 block RAM units offered by the particular Spartan-3 chip selected in this work for implementation tests. Also, utilization of the RAM capacity is quite low: of the total 512 cells in each just 16 (1/32) are actually taken by the complete message.

All the RAM modules must be loaded with the message words before the actual computation begins. This loading introduces obligatory initialization phase which adds extra delay and in some cases can remarkably slow down the total execution time. Nevertheless, thanks to the dual port interface two message words can be loaded in parallel so the loading operation needs 8 clock cycles and these cycles can be much shorter than the ones required during actual computation (which starts afterwards).

Further issues arise regarding memory synchronization. The block RAM is a fully synchronous module also in read operation, i.e. when the read address is established the read data appears on the outputs only after the clock edge. This means that without appropriate compensation in clock cycles when computing some particular round number j the RAM outputs would present message words for the previous round $j-1$. In order to solve this problem one void clock cycle is needed to "precharge" RAM outputs. In BLAKE, where m words are mixed with c constants, the counters used for reading σ permutations of m words need to be one cycle ahead of those used for addressing the c constants so the two sets of counters are needed. In BLAKE2, as there is no need to address the c constants, the counters do not need to be doubled. All in all, together with the 8 clock cycles needed to load message data to RAM modules the preliminary phase adds in total 9 clock cycles before the actual computation of the compression can start.

3.4 Evaluating memory size

In the following analysis let's first concentrate on the BLAKE variant of the algorithm.

Distribution of the message words which are needed in particular G_i functions depends on the σ permutations and these are presented in *Table 1* ([1]). Any given hardware round instance Rj selects $\sigma_{r \bmod 10}$ for computation of round number r and this specific permutation determines which m_i words will be loaded by its G functions according to the table. Looking at the columns assigned to each G_i instance and considering which round numbers will be computed by the Rj (depending on the architecture not all round numbers are computed in all the Rj instances) we can find how many message words will be needed in each G_i module – hence what capacity of the associated memory module is required.

Table 1. The σ permutations and their elements assigned to the instances of the G function.

	G_0		G_1		G_2		G_3		G_4		G_5		G_6		G_7	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Let's μ_i denote number of message words appearing in equations (4) of specific G_i function and M_i – the associated memory module. In the basic x1 architecture all the rounds are computed in the one (and only) *R0* module so all the permutations affect the G instances contained within. Because not all word indices appear in the first two columns of the table which are assigned to the G_0 instance (the missing ones are 3 and 4) so the associated M_0 module needs to store $\mu_0 = 14$ words, without m_3 and m_4. Actually, most of the other modules in this configuration must store 14 words, with the exceptions of M_4 (in those columns only index 3 is omitted and $\mu_4 = 15$) and M_7 ($\mu_7 = 13$, without m_6, m_7 and m_{12}).

In the unrolled architectures the round instances compute specific (and not all) rounds so the reductions can be greater. In the extreme x5 case each Rj instance uses just two permutations and there are some G

modules where they generate $\mu_i = 3$ (e.g. the *R2* applies σ_2 and σ_7 and the G_0 instance inside this round receives m_{11}, m_8 in the first permutation and m_{13}, m_{11} in the second one).
The results of this kind of tedious evaluation (which is not included here in completeness to preserve space) of the μ_i values in all the architectures are presented in *Table* 2. For each round instance *Rj* in every organization the second column lists the permutations applied within (if the sequence returns modulo 10 to beginning of the list the repeated permutations are insignificant and are not listed), the third one gives min-max range of the μ_i values and the fourth – total capacity of the M_i units. The last two columns summarize memory for the whole design (in all rounds): the fifth expresses the total storage in a number of 32b words and the last one in kilobits.
As it turns out, with increasing loop unrolling factor *k* reductions in μ_i can almost compensate the rise in the total number of M_i modules: the x5 organization needs only 40% more net storage than the x1 one despite the fact that the number of the M_i modules has increased from 8 to 40. This compensation is not so effective in the x4 case because of the irregularities in reductions of the permutations for this particular unrolling factor (4 is not a divisor of 14).
In the BLAKE2 algorithm the reduction of n_r from 14 to 10 changes these results only in the x4 case. In x1, x2 and x5 the 4 extra rounds of the original BLAKE return modulo to the beginning of the permutation sequence so they do not add any new message words to those already identified in the first 10 rounds . Only in the x4 case the BLAKE2 permutation sequences are actually shorter and the total storage is reduced by a remarkable 27% as compared to x4 in the BLAKE variant – and this is the only new case added in the table.
As the final remark we should add that the above evaluation estimates the RAM volumes as they can be implemented in ASIC technologies. In the FPGA implementations which are investigated in the rest of this paper the storage had to be implemented with the block RAM units of a constant (not configurable) size, as it was already noted in the previous subchapter, and one unit per *G* instance must be used regardless of its actual occupancy.

Table 2. Memory required for storing the message words in all four architectures.

	Permu-tations	$\mu_{0\div7}$	In round $\sum_i \mu_i$	Total $\sum_R \sum_i \mu_i$	Total [kb]
			BLAKE		
x1 *R0*	$\sigma_0\sigma_1\ldots\sigma_9$	13 ÷ 15	112	112	3.50
x2 *R0*	$\sigma_0\sigma_2\ldots\sigma_8$	7 ÷ 10	69	142	4.44
R1	$\sigma_1\sigma_3\ldots\sigma_9$	8 ÷ 10	73		
x4 *R0*	$\sigma_0\sigma_4\sigma_8\sigma_2$	5 ÷ 8	56	207	6.47
R1	$\sigma_1\sigma_5\sigma_9\sigma_3$	6 ÷ 8	59		
R2	$\sigma_2\sigma_6\sigma_0$	4 ÷ 6	45		
R3	$\sigma_3\sigma_7\sigma_1$	5 ÷ 6	47		
x5 *R0*	$\sigma_0\sigma_5$	4	32	156	4.88
R1	$\sigma_1\sigma_6$	4	32		
R2	$\sigma_2\sigma_7$	3 ÷ 4	31		
R3	$\sigma_3\sigma_8$	3 ÷ 4	30		
R4	$\sigma_4\sigma_9$	3 ÷ 4	31		
			BLAKE2		
x4 *R0*	$\sigma_0\sigma_4\sigma_8$	5 ÷ 6	45	150	4.69
R1	$\sigma_1\sigma_5\sigma_9$	4 ÷ 6	44		
R2	$\sigma_2\sigma_6$	2 ÷ 4	30		
R3	$\sigma_3\sigma_7$	3 ÷ 4	31		

4. Results

4.1 Implementing the designs

Both versions of the cipher were implemented in all four architectures in configurations where the main hardware module computing the compression function was equipped with some basic input / output registers providing means for iterative hashing of the message in 512b chunks. Then the eight designs were automatically synthesized and implemented by Xilinx ISE software with XST synthesis tool for the Spartan-3 XC3S5000-5 device ([13]). The chip was selected because it was sufficiently large to accommodate even the most sized x5 organization. The same approach was applied in our previous works on BLAKE ([10]-[11]) so an already existing test platform was uniformly extended to accommodate BLAKE2 version, keeping the ability to produce comparable results.
The results obtained after implementation of the two ciphers without and with RAM, in all 4 organizations – a total of 16 test cases – are presented in *Table 3*. Speed aspect is represented in the first column by the value of the minimum clock period as it was estimated after static timing analysis of the final, fully routed design. The two next columns provide parameters which illustrate effectiveness (or difficulties) of the implementation process, i.e. how the complex logical transformations of the algorithms were realized with programmable resources of the array: for the longest

combinational path in the design the second column gives number of logic elements it contains 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% indicates problems with routing of connections between logic elements of the array. Size characteristics are reported in the last two columns which give the total numbers of utilized LUT generators and slices.

Table 3. Parameters of the BLAKE implementations.

	min. T_{clk} [ns]	Levels of logic	Routing delay [%]	Size: LUTs	Size: Slices
BLAKE RAM					
x1	40.3	62	44.4	4961	2860
x2	83.6	100	57.1	8684	4894
x4	157.7	180	53.0	16142	8638
x5	197.3	229	53.3	20448	10913
BLAKE Std.					
x1	45.7	66	50.6	9155	5415
x2	88.9	118	51.2	16928	10039
x4	189.7	203	58.7	32933	19000
x5	244.1	258	61.7	41923	23232

Table 4. Parameters of the BLAKE2 implementations.

	min. T_{clk} [ns]	Levels of logic	Routing delay [%]	Size: LUTs	Size: Slices
BLAKE2 RAM					
x1	40.0	66	41.3	3935	2353
x2	78.0	119	46.2	6863	3809
x4	151.9	219	47.6	12396	6576
x5	194.9	267	50.2	15426	8254
BLAKE2 Std.					
x1	43.4	29	59.7	8370	4995
x2	86.7	85	58.7	15058	8620
x4	195.7	209	59.9	27906	16549
x5	246.7	250	61.1	34492	20065

The results of the standard BLAKE implementations for comparison are cited from [11].

4.2 Size and speed effects of the proposed RAM application

Based on the data from *Tables 3* and *4* we can compare size and speed of implementations modified in a way proposed in this paper against the results of the standard approach without RAM utilization. Such a comparison is the purpose of *Figure 3*: the minimum clock period T_{clk} (speed) and the number of LUT generators (size) for implementations with RAM are expressed as percentages of corresponding values of standard (no RAM) realizations – for all four organizations and for both versions of the hash. As one can see, in all cases the percentages are below the 100% level, i.e. the RAM implementations were faster (shorter T_{clk}) and smaller than their traditional counterparts.

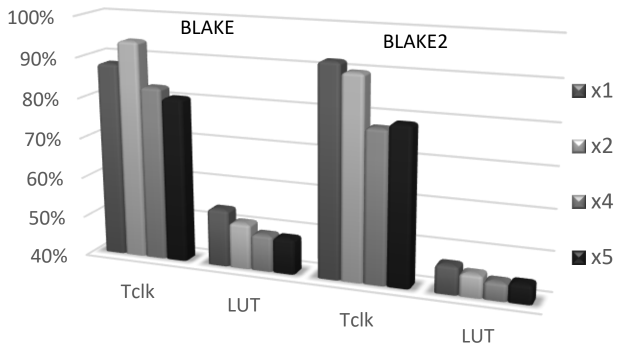

Figure 3. Speed and size of the proposed RAM-based implementations vs. standard results.

While it was obviously expected that moving part of the logic from the FPGA array to the block RAM should reduce LUT utilization, the actual scale of this reduction is outstanding: LUT numbers were cut approximately by half, in the BLAKE implementations on average to 51%, in BLAKE – even to 45%. The scale of the improvement indicates what burden was placed on the implementation tools when the message words, 32b each, are to be delivered and selected in multiplexers twice in each G_i module: this task alone took approximately half of the designs with only rest of the resources busy with actual hashing (which – as in any other cryptographic algorithm – is a very complex job on its own).

The improvement, although not so stable across all configurations, is seen also in performance characteristic: while reading the message words from the block RAMs does introduce some delay, passing them through the distributed logic in the standard architectures turned out to be even slower so the overall clock period is reduced by 6 - 22%. The average reduction is slightly better for the BLAKE2 variant. Specific individuality is observed in the case of the x2 design of BLAKE with the RAM modification: in this particular configuration the optimization procedures especially efficiently reduced levels of logic but this was accompanied with a disproportional increase in the routing part of the longest path (the only case when an increase in any parameter is observed) so the reduction in the overall clock period is actually the smallest across all the cipher/ architecture combinations.

In order to explain noticeably higher T_{clk} percentages of the x1 and x2 architectures in the BLAKE2 variant we must consider other parameters from *Tables 3* and *4.* As they show, these two version of the algorithm were treated differently by the optimization procedures with regard to the formation of the longest path. *Figure 4* compares number of logic levels and percentage of routing delay in the RAM vs. standard implementations. While in the original BLAKE the level of logic with application of RAM was uniformly reduced approx. by 11% and so the routing part of the delay (with the abovementioned exception of the x2 case), in BLAKE2 the optimization took a different path: in the x1 and x2 cases it was better to increase the number of logic levels in order to gain superior reductions in routing delay. Nevertheless the overall minimum clock period was shortened, albeit not so efficiently as in the corresponding BLAKE configurations. The x4 and x5 configurations were optimized like in the BLAKE cases (comparable reductions both in levels of logic and in routing delay) and this led to better reductions in clock period.

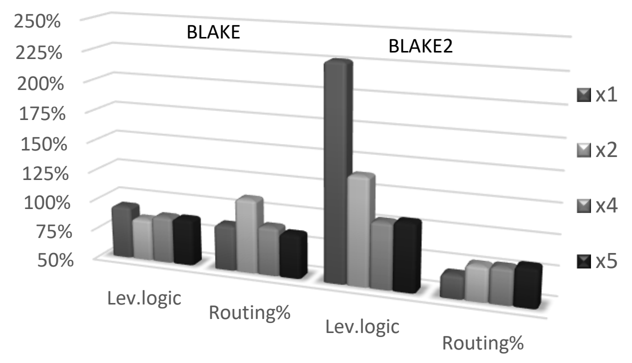

Figure 4. Parameters of the longest path in the RAM-based implementations vs. standard results.

4.3 Scaling efficiency

Scalability in implementation of the loop unrolled architectures is the ability to keep size and speed efficiency in proportion to the number of rounds instantiated in hardware. As the previous studies have shown e.g. in [9] some contemporary cryptographic algorithms may exhibit significant weaknesses in this aspect, mainly due to their very involved and irregular internal organisation which is difficult to map to the FPGA array in larger (highly unrolled) organizations. In order to consistently evaluate scalability of the unrolling mechanism among diversity of BLAKE variants and organizations, the analysis presented in this point was based on comparative relations rather than on evaluation of absolute values of the parameters, in a manner similar to the one applied already in [10].
In every cipher / organization combination the x1 architecture was taken as a point of reference and its characteristics were used for estimation of size and speed of the derived architectures in the following way. The size of each unrolled architecture x*k* should increase proportionally to the number of rounds implemented in hardware (the unrolling factor *k*) and we estimate

$$Size_{xk} \approx Size_{x1} \cdot k \tag{8}$$

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, i.e.:

$$Tclk_{xk} \approx Tclk_{x1} \cdot k \tag{9}$$

These two equations and the parameters of the x1 architectures in both algorithms were used for calculating the estimated clock periods and numbers of LUTs for the x2 - x5 cases and *Figure 5* presents the results as the ratios *actual_value* / *estimation*. The lower the ratio, the faster (shorter T_{clk}) or the smaller (lower number of LUT) was the actual design in comparison to what could be expected from its x1 case. The value of 100% is the threshold separating “better than” (<100%) from “worse than” (>100%) the expected.
Considering the speed aspect (T_{clk} value) first, it should be noted that only RAM-based architectures behave close to the expectations for the both cipher variants and their deviations from the estimated values are within ±5% margin, while the standard implementations in most of the cases are noticeably worse than estimations. What’s more, deeper problems are exhibited in the BLAKE2 version despite its much reduced size: the x4 and x5 cases are by 13-14% slower than expected (in BLAKE – by 4-7%) and these are the worst results across all the cases. This suggests that advantages brought by RAM application are more important for implementation efficiency than the BLAKE2 simplifications.
When looking at changes in speed scaling which were brought by the proposed RAM application one can see that they are almost always positive i.e. the ratios are smaller in the RAM versions than in their standard counterparts, with an only exception of the x2 case in the BLAKE version. This is also the only situation when the standard implementation behaves better than expected and the RAM-based worse – again an indication of singularity of this particular case as it was already noticed in the first analysis of this chapter.

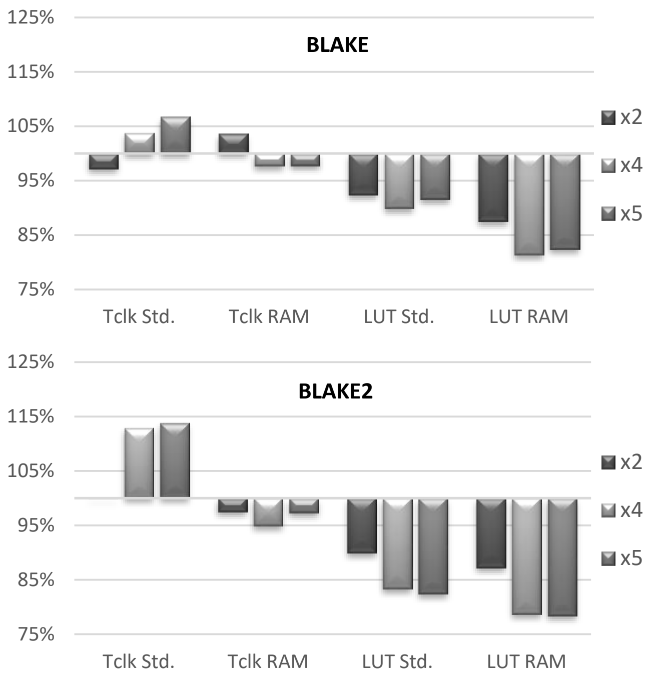

Figure 5. Parameters of the unrolled architectures as percentages of the estimations based on the x1 case.

Effects of the scaling in size are more consistent and always positive: in all the 12 unrolled cases the architectures are always smaller than estimations and the decreases are evidently bigger than those in the T_{clk} evaluation. The best reductions by up to 13-23% are noted for RAM-based BLAKE2 cases and the smallest – for BLAKE implemented in the standard way (up to 8-10%) but variety across all the combinations is not as wide as it was in the speed parameter. It is worth noting that again the reductions are better in the proposed RAM versions than in their standard counterparts, this time without a single exception.

4.4 BLAKE2 versus BLAKE

In the last analysis we will compare parameters of the two versions of the algorithm, again looking at the the ratios of respective parameters between the two versions of the cipher, implemented in a standard way and with memory (*Figure 6*).
As it was already remarked in chapter 2.2, BLAKE2 slightly simplifies processing of the *G* function by removing xor operations which use sixteen 32-bit constant words. It is reasonable that the resultant proportional reductions in LUT numbers are bigger in RAM implementations (down to 75-79%) than in the standard ones (to 82-91%) because in those cases the removed hardware reduces smaller designs. Still it is worth noting that this relatively minor amendment in the equations can have remarkable influence on size, reducing LUT utilization in the x5 RAM case by ¼. In the standard implementations, on the other hand, this effect scales with the unrolling factor more intensively than in the RAM ones.

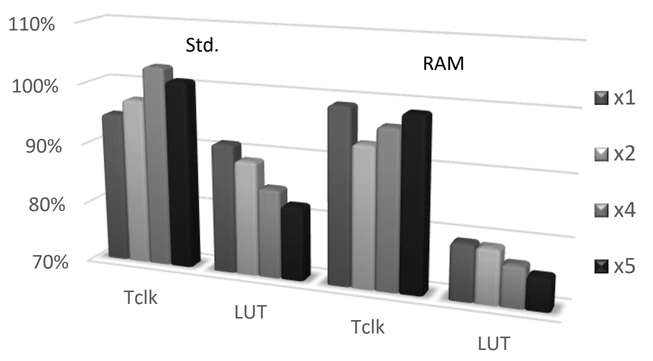

Figure 6. Parameters of BLAKE2 vs. BLAKE implementations.

The situation is not so consistent in the T_{clk} parameter. First of all, the two standard cases x4 and x5 are slower in the BLAKE2 version than in the original BLAKE one. Although their clock periods increased by a small margin (+3 and +1%), it is against the general trend that smaller designs (with shorter paths) are faster and this indicates some problems in automatic routing. This situation was improved by applying RAM: all the cases were faster in the BLAKE2 version but the progress (reductions down to 93-99%) are far from the ones observed in the numbers of LUT elements. With the only exception of the x1 case, the reductions in the T_{clk} period were better in RAM versions than in the equivalent standard ones.

5. Conclusions

In this paper we have discussed a modification in hardware implementations of the BLAKE and BLAKE2 hash algorithms which, by using block memory units provided as auxiliary FPGA resources, eliminated the need for involved data paths distributing message bits among the round modules. These path are specific peculiarity of this algorithm and are not needed neither in KECCAK (the new SHA-3 standard) nor in ChaCha (upon which the BLAKE processing was constructed). The idea was implemented in realizations of both BLAKE and BLAKE2 versions of the cipher in 4 different organizations: the standard iterative one and three high-speed loop-unrolled architectures with 2, 4 and 5 rounds instantiated in hardware. Together with standard (without RAM) implementations used for comparison this produced a total of 16 test cases considered in this paper: after implementation in a popular Spartan-3 device from Xilinx their parameters allowed for exhaustive evaluation of the proposed modification.
The results reveal that the modification outstandingly enhanced size of all the tested architectures: on average, occupation of the FPGA array was reduced at

least by half. The improvements in speed, although not so spectacular, are also visible, in case of some unrolled architectures by up to 1/5. Additional analyses indicated that the proposed modification can improve overall efficiency of routing, helps in generation of the loop-unrolled architectures and strengthens optimizations introduced in the BLAKE2 version of the algorithm.

Whether these improvements compensate the extra cost of memory blocks introduced to the design remains an open issue. In ASIC designs the amount of extra memory needed for repetitive storage of the message inside each round instance ranges from 3.5kb in the standard iterative architecture to 5 – 6.5kb in the unrolled organizations for the cases when 4 or 5 rounds are realized in hardware. In the FPGA engineering practice, though, it is common that the design fitted in the device does not use all RAM resources and we have shown that in such situations, if there remain some free memory blocks in the chip, using such "leftovers" for improvements in BLAKE / BLAKE2 implementation is definitely an option worth consideration.

References

[1] Aumasson, J.-P., Henzen, L., Meier, W. & Phan, R.C.-W. (2010). SHA-3 proposal BLAKE, version 1.3. https://www.131002.net/blake/blake.pdf; accessed: March 2017.

[2] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., & Winnerlein, C. (2013). BLAKE2: simpler, smaller, fast as MD5. Jacobson M., Locasto M., Mohassel P., Safavi-Naini R. (eds) *Applied Cryptography and Network Security ACNS 2013*. Springer LNCS, **7954**, 119-135.

[3] Bernstein, D.J. (2008). ChaCha, a variant of Salsa20 http://cr.yp.to/chacha/chacha-20080128 .pdf; accessed: March 2017.

[4] Bernstein, D.J. (2008). The Salsa20 Family of Stream Ciphers. Robshaw M., Billet O. (eds) *New Stream Cipher Designs*. Springer LNCS 4986.

[5] Dunkelman, O., & Biham, E. (2006). A framework for iterative hash functions: Haifa. *2nd NIST Cryptographich Hash Workshop*, **22**.

[6] 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.

[7] 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**, p. 2007.

[8] 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.

[9] Sugier, J. (2015). Popular FPGA Device Families in Implementation of Cryptographic Algorithms. Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) *Theory and Engineering of Complex Systems and Dependability. Proc. 11th Int. Conf. Dependability and Complex Systems DepCoS-RELCOMEX*. Springer AISC, **365**, 485-495.

[10] Sugier, J. (2016). Implementation Efficiency of BLAKE and Other Contemporary Hash Algorithms in Popular FPGA Devices. Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) *Proc. 11th Int. Conf. Dependability and Complex Systems DepCoS-RELCOMEX*. Springer AISC, **470**, 457-467.

[11] Sugier, J. (2016). Implementing SHA-3 candidate BLAKE algorithm in Field Programmable Gate Arrays. *J. Polish Safety and Reliability Association*, **7**(1), 193-200.

[12] Sugier, J. (2017). Simplifying FPGA Implementations of BLAKE Hash Algorithm with Block Memory Resources. *Procedia Engineering*, **178**, 33-41.

[13] Xilinx, Inc. (2009). Spartan-3 Family Data Sheet. www.xilinx.com (ds099.pdf); accessed: March 2017.