A Very Compact FPGA Implementation of LED and PHOTON

N. Nalla Anandakumar[1,2], Thomas Peyrin[1] and Axel Poschmann[1,3]

[1] Division of Mathematical Sciences, School of Physical and Mathematical Science,
Nanyang Technological University, Singapore
[2] Hardware Security Research Group,
Society for Electronic Transactions and Security, India
[3] NXP Semiconductors, Germany
nallananth@gmail.com, thomas.peyrin@gmail.com, aposchmann@gmail.com

Abstract. LED and PHOTON are new ultra-lightweight cryptographic algorithms aiming at resource-constrained devices. In this article, we describe three different hardware architectures of the LED and PHOTON family optimized for Field-Programmable Gate Array (FPGA) devices. In the first architecture we propose a round-based implementation while the second is a fully serialized architecture performing operations on a single cell per clock cycle. Then, we propose a novel architecture that is designed with a focus on utilizing commonly available building blocks (SRL16). This new architecture, organized in a complex scheduling of the operations, seems very well suited for recent designs that use serial matrices. We implemented both the lightweight block cipher LED and the lightweight hash function PHOTON on the Xilinx FPGA series Spartan-3 (low-cost) and Artix-7 (high-end) devices and our new proposed architecture provides very competitive area-throughput trade-offs. In comparison with other recent lightweight block ciphers, the implementation results of LED show a significant improvement of hardware efficiency and we obtain the smallest known FPGA implementation (as of today) of any hash function.

Keywords: FPGA, lightweight cryptography, LED, PHOTON, SRL16.

1 Introduction

Lightweight devices such as RFID tags, wireless sensor nodes and smart cards are increasingly common in applications of our daily life. These smart lightweight devices might manipulate sensitive data and thus usually require some security. Classical cryptographic algorithms are not very suitable for this type of applications, especially for very constrained environments, and thus many lightweight cryptographic schemes have been recently proposed (block ciphers [20, 30, 16, 11, 39, 36, 5] or hash functions [2, 19, 6]). The main focus of lightweight cryptography research has been on the trade-offs between cost, security and performance in terms of speed, area and computational power. These primitives can be implemented either in software or in hardware platforms such as Field-Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC). Compared to ASICs, FPGAs offer additional advantages in terms of time-to-market, reconfigurability and cost.

Recently, Guo et al. proposed the lightweight block cipher LED [20] and the lightweight family of hash functions PHOTON [19], for which the hardware performance has only been investigated on ASICs. LED is based on AES-like design principles with a very simple key schedule. The internal unkeyed permutations of PHOTON can also be seen as an AES-like primitive. Up to now, no design space exploration of LED on FPGAs has been published. The proposed architecture is suited for the applications where low-cost FPGAs are deployed such as FPGA-based RFID tags [15] and low-power FPGAs [38] are deployed for battery powered applications such as FPGA-based wireless sensor nodes [14]. Hence, they represents popular platforms (FPGA-based RFID tags, FPGA-based wireless sensor nodes) for lightweight cryptographic applications.

Our contributions. In this study, we propose three architectures optimized for the implementation of the LED block cipher and the five different flavors of the PHOTON hash functions family on FPGAs. The first architecture computes one round per clock cycle, while the second is based on the architecture presented in LED [20] and PHOTON [19] for ASIC, and adapted in this paper to FPGA with slight modifications. Our most interesting contribution is the third architecture, also serial by nature, which performs the LED and PHOTON computations based on shift registers (SRL16), thanks to a non-trivial scheduling of the successive operations. This structure is actually strictly better than the second one since it achieves lower area and better throughput.

We emphasize that the goal of this paper is to cover a wide variety of new implementation trade-offs offered by crypto primitives using serialized or recursive MDS (Maximum Distance Separable) matrices (for which LED and PHOTON are the main representatives), on a wide variety of different Xilinx FPGA families, ranging from low-cost (Spartan-3) to high-end (Artix-7). Using our novel architecture, based on SRL16, one requires only 77 slices for LED-64 and 112 slices for PHOTON-80 on a Xilinx Spartan 3 (XC3S50) device, and 40 slices for LED-64 and 58 slices for PHOTON-80 on an Artix-7 (XC7A100T) device (while achieving reasonable throughput of 9.93 Mbps and 22.93 Mbps for LED-64, 6.57 Mbps and 18.33 Mbps for PHOTON-80). To the best of our knowledge, it represents the most compact hash function implementations on FPGAs.

The article is structured as follows. First we provide the description of LED and PHOTON in Section 2. Then, we provide in Section 3 and Section 4 our architectures and FPGA implementations of LED and PHOTON respectively. We finally draw conclusions in Section 5.

2 Algorithms descriptions

In this section, we describe the different versions of LED block cipher [20] and the PHOTON [19] family of hash functions.

2.1 LED

LED is a 64-bit block cipher based on a substitution-permutation network (SPN). It supports any key lengths from 64 to 128 bits. In this article, we will focus on a few main versions: 64-bit key LED (named LED-64) and 128-bit key LED (named LED-128). The number of rounds N depends on the key size, LED-64 has $N = 32$ rounds while LED-128 has $N = 48$ rounds.

One can view the 64-bit internal state as a 4 × 4 matrix of 4-bit nibbles and the round function as an AES-like permutation composed of the following four operations:

- *AddConstants:* the internal state is bitwise XORed with a round-dependent constant (generated with an LFSR);
- *SubCells:* the PRESENT [7] S-box is applied to each 4-bit nibble of the internal state;
- *ShiftRows:* nibble row i of the internal state is cyclically shifted by i positions to the left;
- *MixColumnsSerial:* each nibble column of the internal state is transformed by multiplying it once with MDS matrix χ^4 (or two times with matrix χ^2, or four times with matrix χ).

$$\chi = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 4\,1\,2\,2 \end{pmatrix}; \quad (\chi)^2 = \begin{pmatrix} 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 4\,1\,2\,2 \\ 8\,6\,5\,6 \end{pmatrix}; \quad (\chi)^4 = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 4\,1\,2\,2 \end{pmatrix}^4 = \begin{pmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix}$$

The key schedule of LED is very simple. In the case of LED-64, the key K is repeatedly XORed to the internal state every 4 rounds (with whitening key operation). In the case of LED-128, the key K is divided into two 64-bit subparts $K = K_1||K_2$, each XORed alternatively to the internal state every 4 rounds. The 4-round operation between two key addition is called a step.

2.2 PHOTON

In this section we describe the PHOTON family of hash functions, for which five versions exist with digest sizes of 80, 128, 160, 224 and 256 bits. PHOTON is based on the sponge construction. First, after padding, the input message is divided into blocks of r-bit each. At each iteration, the t-bit internal state ($t = r + c$) absorbs the incoming message block by simply XORing it to the r-bit *bitrate* part (the remaining c-bit part is called the *capacity*). Then, after the absorption of the message block, one applies a t-bit permutation P to the internal state. Once all message blocks have been processed the squeezing phase starts. During this phase, for each iteration r' bits are output from the internal state and the permutation P is applied. One continues to squeeze until the proper digest size n is reached.

The PHOTON internal permutation P is also AES-like and consists of 12 rounds. The internal state is represented as a $(d \times d)$ matrix of s-bit cells and each round is defined as the application of 4 operations:

- *AddConstants:* the internal state is bitwise XORed with a round-dependent constant (generated with an LFSR);
- *SubCells:* the S-box is applied to each s-bit nibble of the internal state (the PRESENT S-box [7] if $s = 4$, the AES S-box [9] if $s = 8$);
- *ShiftRows:* nibble row i of the internal state is cyclically shifted by i positions to the left;
- *MixColumnsSerial:* each nibble column of the internal state is transformed by multiplying it once with MDS matrix χ^d (or two times with matrix $\chi^{d/2}$, ... , or d times with matrix χ).

The values of t, c, r, r', s and d depend on the hash output size n and we give in Table 1 the 5 versions of PHOTON (we refer to [19] for the various matrices χ depending on the PHOTON versions). Note that one always uses a cell size of 4 bits, except for the PHOTON-256/32/32 version for which one uses 8-bit cells.

Table 1. The 5 versions of PHOTON parameters

	r	r'	c	s	d	t
PHOTON-80/20/16	20	16	80	4	5	100
PHOTON-128/16/16	16	16	128	4	6	144
PHOTON-160/36/36	36	36	160	4	7	196
PHOTON-224/32/32	32	32	224	4	8	256
PHOTON-256/32/32	32	32	256	8	6	288

3 LED implementations

In this section, we present three different architectures for the FPGA implementation of the lightweight block cipher LED. The first one is a round-based implementation, while the second one is a fully serialized implementation, performing operations on a single cell during each clock cycle. The third one is a novel architecture, also fully serial, but based on the SRL16s and aiming at the smallest area possible. As we are interested in the performance of the plain LED core, we did not include any I/O logic implementation such as a UART interface.

We have also investigated the performance of the LED cipher with different trade-offs. Indeed, the diffusion matrix being serial in LED, one can view the *MixColumnsSerial* diffusion layer as a single application of $(\chi)^4$, or two successive applications of $(\chi)^2$ or four successive applications of (χ).

We have implemented both LED versions (the 64-bit key version LED-64 and the 128-bit key version LED-128) in VerilogHDL and targeted Xilinx FPGAs Spartan-3 [23] and Artix-7 [25]. We used Mentor Graphics ModelSimPE for simulation purposes and Xilinx ISE v14.4 WebPACK for design synthesis. In Xilinx ISE the design goal is kept balanced and strategy is kept default (unlocked) and the synthesis optimization goal is set to area.

3.1 Round-based

We give in Figure 1 the block diagram of the round-based implementation of LED. Naturally, the data register (Dreg) is updated after every round operation. The keys are selected according to the key length (K_1 is loaded without modification every four rounds in LED-64, while K_1 and K_2 are loaded alternatively every four rounds for LED-128). Table 2 provides the detailed results of our round-based FPGA implementations of LED with three different approaches concerning the computation of the diffusion matrix: we compute χ^4 by either applying 4 times the matrix χ, or by applying 2 times the matrix χ^2, or by directly applying the entire matrix χ^4. As expected, the last option provides a higher throughput (since we directly compute the entire diffusion matrix), but for the price of higher resource consumption. In contrary, the first option allows to save resources, but at the expense of a lower throughput. The second option offers a trade-off in between.

We also added in Table 2 a comparison with known round-based FPGA implementations of other (lightweight) block ciphers on the same FPGA device. One can see that our LED-64 and LED-128 proposed round-based implementations outperform all the previous works in term of area.

3.2 Serialized

Our first serialized implementation of LED is derived from the architecture proposed in [20] for ASICs, but with some architectural modifications for the *MCS* state operations in order to improve the performance. This implementation stores the data and key in the registers (FF) and it has a 4-bit wide datapath, i.e. only 4 bits are processed in one clock cycle (see Figure 2). It consists of 4 states: *Init*, *Sbox*, *Srow* and *MCS*:

The *Init* state initial data and key values are stored in the data registers and key registers, respectively.

The *Sbox* state is for the simultaneous execution of the *SubCells* (SC) operations, *AddConstants* (AC) operations and XORing the roundkey (AK) every fourth round. It requires 16 clock cycles.

The *Srow* state is for the execution of the *ShiftRows* operation. It can be performed in 3 clock cycles with no additional hardware cost, because it just shifts the row positions of the state matrix.

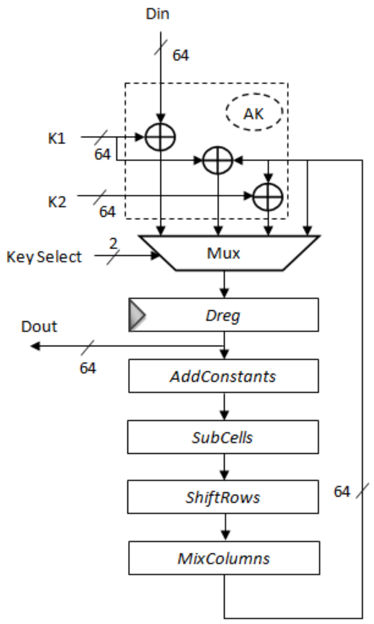

Fig. 1. Architecture of the `LED` round based encryption module

Table 2. FPGA round-based implementation results of `LED` block cipher with different approaches for diffusion matrix computation.

Design	MDS approach	Block	Key	No. of	No. of	No. of	Clock	Max. freq	T/put	Eff.	FPGA Device

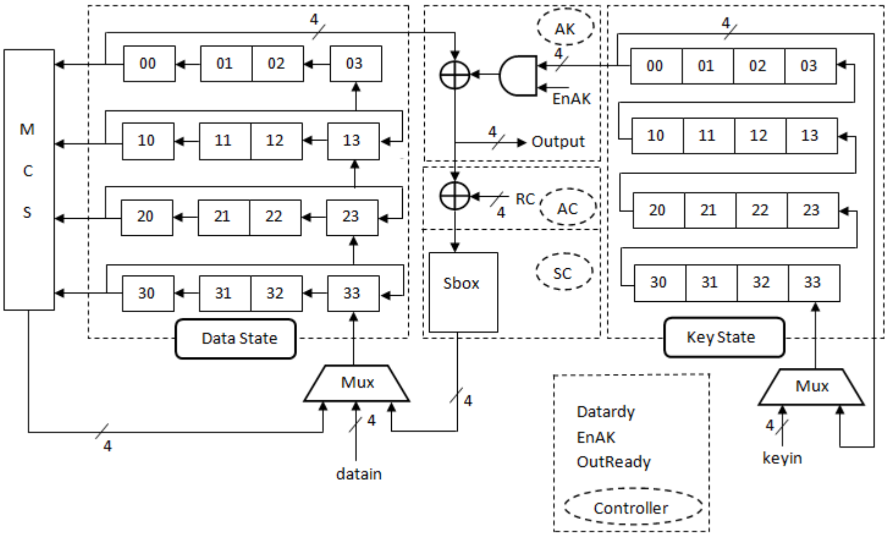

Fig. 2. A serialized architecture of the LED encryption module

proposed in [20]. We give in the first row of Table 3 the detailed results of our serialized implementations. For a (χ) version of the diffusion matrix computation, we obtain for LED-64 and LED-128 140 slices and 167 slices respectively, while the throughput reaches 9.11 Mbps and 5.2 Mbps, respectively. One can see that LED-64 and LED-128 seem to require much less area than most ciphers [27, 21, 10, 28] while having a higher throughput than SIMON [3]. Furthermore, an increased throughput can be reached by scaling the datapath to 16 bits and by computing the diffusion matrix in a less serial manner, i.e. by applying two times $(\chi)^2$ or direct $(\chi)^4$. Moreover, our proposed serialized implementations when using directly $(\chi)^4$ outperforms most ciphers [28, 3] implementations in terms of throughput per area ratio (Eff.). Using device-dependent building blocks, such as BRAMs and DSPs, are a great way to enhance performance and optimize implementations for a specific target device. However, it also, obviously, makes a fair comparison of the hardware costs (area) much more difficult. Therefore we do not use any additional building blocks and instead compare the number of slices. In the next section we will explain how to further reduce area and latency.

3.3 Serialized using SRL16s

Our second serialized implementation of LED is based on the use of a building block of Xilinx Spartan-3 FPGAs called SRL16s [24]. More precisely, SRL16 are look up tables (LUT) that are used as 16-bit shift registers that allow to access (or output) bits of its internal state in two ways (as shown in Figure 3): the last bit of its 16 stages (Q_{15}) is always available, while a multiplexer allows to access one additional bit from any of its internal stages.

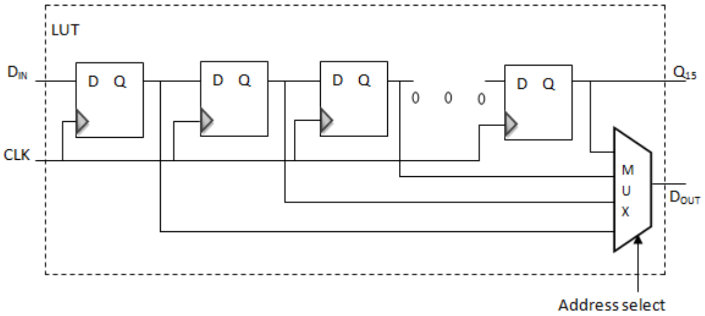

Fig. 3. LUT configured as a shift register

The Configurable Logic Blocks (CLBs) are the basic logic units in an FPGA. Each CLB has four slices, but only the two at the left-hand of the CLB can be used as shift registers. Spartan-3 FPGAs can configure some LUTs as a 16-bit shift register without using the flip-flops available in each slice. When a shift register

Table 3. FPGA serialized implementation results of LED block cipher with different approaches for diffusion matrix computation.

Design	MDS approach	Data-path (bits)	Block Size (bits)	Key Size (bits)	Area (slices)	No. of FFs	No. of LUTs	Clock Cycles	Max. freq (MHz)	T/put (Mbps)	Eff. (Mbps/ slices)	FPGA Device
LED our paper (Section 3.2)	(χ)	4	64	64	140	151	255	1120	159.43	9.11	0.07	Spartan-3 XC3S50-5
				128	167	216	302	1680	137.34	5.2	0.03	
	$(\chi)^2$	8	64	64	169	157	332	608	157.43	16.6	0.10	
				128	203	219	388	912	142.01	9.97	0.05	
	$(\chi)^4$	16	64	64	180	162	342	352	137.5	24.99	0.14	
				128	219	227	414	528	128.73	15.6	0.07	
	(χ)	4	64	64	37	52	78	1120	378	21.6	0.58	Artix-7 XC7A100T-3
				128	40	57	82	1680	368	14.02	0.35	
	$(\chi)^2$	8	64	64	58	95	135	608	380.3	40.03	0.69	
				128	61	104	141	912	356.5	25.02	0.41	
	$(\chi)^4$	16	64	64	78	110	162	352	367.4	66.8	0.86	
				128	82	175	188	528	375.6	45.53	0.56	
LED our paper (Section 3.3)	(χ)	16	64	64	111	80	215	640	119.62	11.96	0.11	Spartan-3 XC3S50-5
				128	122	72	233	960	118.25	7.88	0.06	
	$(\chi)^2$	8	64	64	77	44	148	768	119.19	9.93	0.13	
				128	86	48	167	1152	120.75	6.71	0.08	
	$(\chi)^4$	16	64	64	119	76	228	256	119.27	29.82	0.25	
				128	127	70	248	384	117.87	19.65	0.15	
	(χ)	16	64	64	51	45	113	640	303.9	30.39	0.60	Artix-7 XC7A100T-3
				128	59	55	121	960	308.5	20.57	0.35	
	$(\chi)^2$	8	64	64	40	36	100	768	275.2	22.93	0.57	
				128	50	40	107	1152	302.64	16.81	0.34	
	$(\chi)^4$	16	64	64	63	43	133	256	284.83	71.21	1.13	
				128	69	53	138	384	286.5	47.75	0.70	
PRESENT [40]			64	128	117	—	—	256	114.8	28.46	0.24	Spartan-3 XC3S50-5
HIGHT [40]			64	128	91	—	—	160	163.7	65.48	0.72	Spartan-3 XC3S50-5
xTEA [27]			64	128	254	—	—	112	62.6	35.78	0.14	Spartan-3 XC3S50-5
PRESENT [21]			64	80	271	—	—	—	—	—	—	Spartan-3E XC3S500
SIMON [3]			128	128	36	—	—	—	136	3.60	0.10	Spartan-3E XC3S500
AES [10]			128	128	184	—	—	160	45.6	36.5	0.20	Spartan-3 XC3S50-5
AES [28]			128	128	393	—	—	534	—	16.86	0.04	Spartan-3 XC3S50-5

is described in generic HDL code with the global reset signal, it has no impact on shift registers and synthesis tools infer the use of the SRL16s. Moreover, SRL16 is present in almost all XILINX FPGA families and [22] describes a way to use SRL16s on ALTERA devices.

We have investigated possible area reductions by scaling the 64-bit implementation to an 8-bit (when using $(\chi)^2$) and 16-bit datapath (when using χ and $(\chi)^4$) using SRL16s. As *MixColumnsSerial* requires 16-bit inputs (4 times 4-bit) in every clock cycle, but each SRL16 only allows access to 2 bits, we have to use eight and sixteen SRL16s to store the state, respectively.

Figure 4 shows the block diagram for the SRL16s based implementation of LED with 8-bit datapath when using $(\chi)^2$. It consists of 4 states: Init, SrSc, Re-update and MCS, where the content of each SRL16 is indicated in Table 4 for all the state operations. We also give in Table 7 and 8 of Appendix A the SRL16 content for 16-bit datapath implementations when using χ and $(\chi)^4$ respectively.

The *Init* state: initial data and key values are stored in the data SRL16s and key SRL16s, respectively. A special ordering of the nibbles is required as shown in Table 4 and in Figure 4.

The *SrSc* state: performs *ShiftRows*, *SubCells*, *AddConstants* and *AddRoundKey* simultaneously by clever memory (SRL16) addressing schedule. Table 4 depicts in bold the bits that are selected in every clock cycle to achieve this. The round operation starts by bitwise XORing the incoming data with the round key and round constants, then applying this result to two S-boxes (8-bit datapath) or four S-boxes (16-bit datapath), respectively. The first nibbles processed are 00 and 11 (8-bit datapath) and 00, 11, 22, and 33 (16-bit datapath), respectively. In order to perform *ShiftRows*, *SubCells*, *AddConstants* and *AddRoundKey* operations on the whole state, it takes 8 clock cycles (clk 9-16 in Table 4) using an 8-bit datapath, and 4 clock cycles (clk 5-8 in Table 8) using a 16-bit datapath, respectively.

The *Re-update* state: when using the 8-bit datapath, the 8-bit output from the S-boxes needs to be duplicated within the SRL16s. This is because the *MixColumnsSerial* operation reads four input vectors simultaneously and thus the leftmost bits of the SRL16s must be used. 8 clock cycles (clk 17-24 in Table 4) are required for this step. Note that this state only applies to 8-bit datapath, this is why it is not present in Table 7 and 8 of Appendix A.

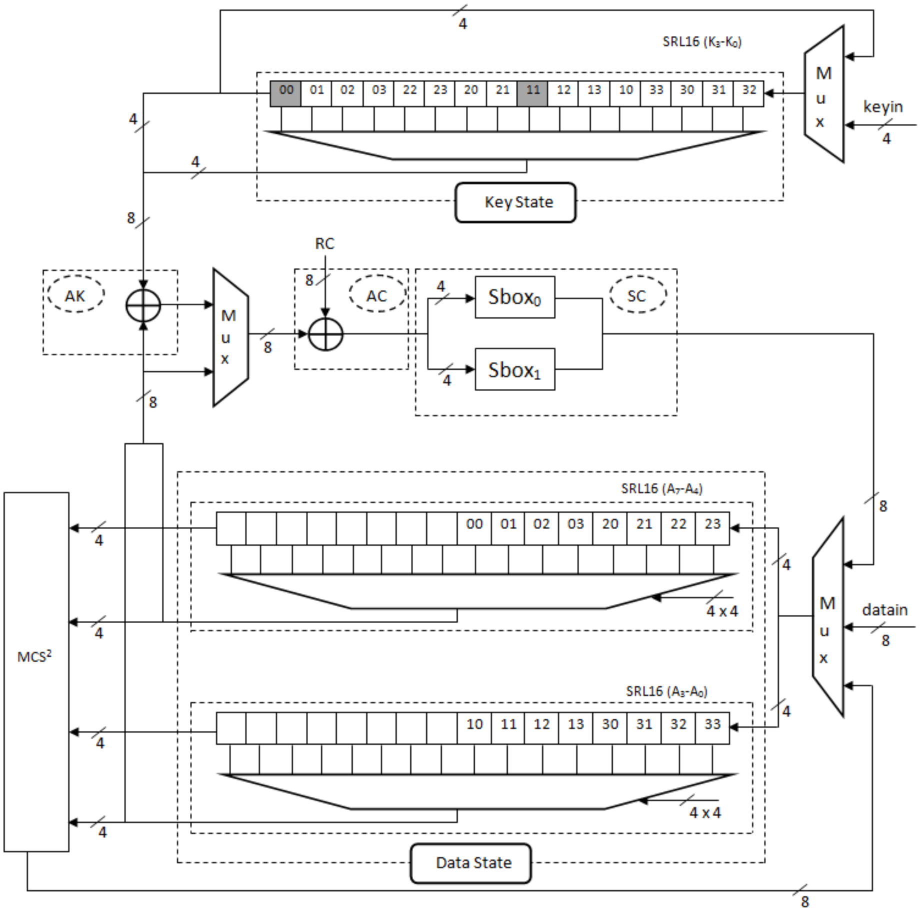

Fig. 4. The block diagram for the SRL16s based implementation of LED with 8-bit datapath when using $(\chi)^2$

The *MCS* state: the 4 x 4-bit input data is read from the bits indicated in bold in Table 4. It starts with the four 4-bit blocks 00, 11, 22 and 33, and using $(\chi)^2$, the resulting 8-bit output is stored in the SRL16s labeled as $00'$, $10'$ (and $20'$, and $30'$, respectively) to indicate the indices of the next round. In the next clock cycle, the input data is 01, 12, 23, and 30, and the corresponding result is labeled as $01'$, $11'$ (and $21'$, and $31'$) and so on. In total 8 clock cycles (clk 25-32) are required to complete the *MixColumnsSerial* layer using $(\chi)^2$, 4 clock cycles (clk 9-12 in Table 8) when using $(\chi)^4$, and 16 clock cycles (clk 9-24 in Table 7) when using (χ), respectively. The next round starts with the *SrSc* state (clk 9) and inputs $00'$ and $11'$.

Concerning the key incorporation, we give in Table 9 (resp. Table 10) of Appendix A the SRL16s positions for the key when using 8-bit datapath with $(\chi)^2$ (resp. when using $(\chi)^4$ or (χ) for the 16-bit datapath).

For the 8-bit datapath, four and eight SRL16s are required in order to store the entire 64-bit and 128-bit key, respectively. The keys are always read 8-bit at a time from the 4-bit blocks indicated in bold in Table 9 with a grey background in Figure 4. Then, the key blocks of SRL16s are rotated by one position. Eight clock cycles (clk 17-24 in Table 9) are required for the 8-bit datapath, but extra 8 clock cycles (clk 25-32 in Table 9) are required for 64-bit key blocks so as to reach the initial position. The next *AddRoundKey* starts with the *SrSc* state (clk 17 in Table 9) and inputs $00'$ and $11'$.

We have used sixteen SRL16s in order to store the 64-bit or 128-bit key for the 16-bit datapath. Initially, the key values are stored in the key SRL16s 4 times for the 64-bit (2 times for the 128-bit). 16 clock cycles (clk 1-16 in Table 10) are required for this step. The keys are read 16-bit at a time from the 4-bit blocks of SRL16s by selecting address taps based on the *ShiftRows* position (clk 17-20 in Table 10). After every 16-bit keys read, the key blocks of SRL16s are rotated by one position. The next *AddRoundKey* starts with the *SrSc* state (clk 17 in Table 10) and inputs $00'$, $11'$, $22'$ and $33'$.

For the 8-bit datapath, 24 clock cycles are required in order to complete one round of LED (clk 9-32 in Table 4), resulting in a total latency of 768 clock cycles for LED-64 and 1152 clock cycles for LED-128. Table 3 shows the detailed results of our implementations of LED based on SRL16s for various MDS matrix computation approaches. Our design $(\chi)^2$ only occupies 77 slices for LED-64 and 86 slices for LED-128 respectively, with a corresponding throughput of 9.93Mbps and 6.71Mbps respectively. The throughput can be increased

Table 4. Content of SRL16s for one round of LED when using $(\chi)^2$ for the 8-bit datapath. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation (see also Figure 4). The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s
	Init		*Re-update*
1	00 10	17	01 02 03 20 21 22 23 00 **01** 02 03 22 23 20 21 00 11 12 13 30 31 32 33 11 **12** 13 10 33 30 31 32 11
2	00 01 10 11	18	02 03 20 21 22 23 00 01 **02** 03 22 23 20 21 00 01 12 13 30 31 32 33 11 12 **13** 10 33 30 31 32 11 12
3	00 01 02 10 11 12	19	03 20 21 22 23 00 01 02 **03** 22 23 20 21 00 01 02 13 30 31 32 33 11 12 13 **10** 33 30 31 32 11 12 13
4	00 01 02 03 10 11 12 13	20	20 21 22 23 00 01 02 03 **22** 23 20 21 00 01 02 03 30 31 32 33 11 12 13 10 **33** 30 31 32 11 12 13 10
5	00 01 02 03 20 10 11 12 13 30	21	21 22 23 00 01 02 03 22 **23** 20 21 00 01 02 03 22 31 32 33 11 12 13 10 33 **30** 31 32 11 12 13 10 33
6	00 01 02 03 20 21 10 11 12 13 30 31	22	22 23 00 01 02 03 22 23 **20** 21 00 01 02 03 22 23 32 33 11 12 13 10 33 30 **31** 32 11 12 13 10 33 30
7	00 01 02 03 20 21 22 10 11 12 13 30 31 32	23	23 00 01 02 03 22 23 20 **21** 00 01 02 03 22 23 20 33 11 12 13 10 33 30 31 **32** 11 12 13 10 33 30 31
8	**00** 01 02 03 20 21 22 23 10 **11** 12 13 30 31 32 33	24	**00** 01 02 03 22 23 20 21 00 01 02 03 **22** 23 20 21 **11** 12 13 10 33 30 31 32 11 12 13 10 **33** 30 31 32
	SrSc		*MCS*
9	00 **01** 02 03 20 21 22 23 00 10 11 **12** 13 30 31 32 33 11	25	**01** 02 03 22 23 20 21 00 01 02 03 22 **23** 20 21 00′ **12** 13 10 33 30 31 32 11 12 13 10 33 **30** 31 32 10′
10	00 01 **02** 03 20 21 22 23 00 01 10 11 12 **13** 30 31 32 33 11 12	26	**02** 03 22 23 20 21 00 01 02 03 22 23 **20** 21 00′ 01′ **13** 10 33 30 31 32 11 12 13 10 33 30 **31** 32 10′ 11′
11	00 01 02 **03** 20 21 22 23 00 01 02 **10** 11 12 13 30 31 32 33 11 12 13	27	**03** 22 23 20 21 00 01 02 03 22 23 20 **21** 00′ 01′ 02′ **10** 33 30 31 32 11 12 13 10 33 20 31 **32** 10′ 11′ 12′
12	00 01 02 03 20 21 **22** 23 00 01 02 03 10 11 12 13 30 31 32 **33** 11 12 13 10	28	**22** 23 20 21 00 01 02 03 22 23 20 21 **00′** 01′ 02′ 03′ **33** 30 31 32 11 12 13 10 33 30 31 32 **10′** 11′ 12′ 13′
13	00 01 02 03 20 21 22 **23** 00 01 02 03 22 10 11 12 13 **30** 31 32 33 11 12 13 10 33	29	**23** 20 21 00 01 02 03 22 23 20 21 00′ **01′** 02′ 03′ 20′ **30** 31 32 11 12 13 10 33 30 31 32 10′ **11′** 12′ 13′ 30′
14	00 01 02 03 **20** 21 22 23 00 01 02 03 22 23 10 11 12 13 30 **31** 32 33 11 12 13 10 33 30	30	**20** 21 00 01 02 03 22 23 20 21 00′ 01′ **02′** 03′ 20′ 21′ **31** 32 11 12 13 10 33 30 31 32 10′ 11′ **12′** 13′ 30′ 31′
15	00 01 02 03 20 **21** 22 23 00 01 02 03 22 23 20 10 11 12 13 30 31 **32** 33 11 12 13 10 33 30 31	31	**21** 00 01 02 03 22 23 20 21 00′ 01′ 02′ **03′** 20′ 21′ 22′ **32** 11 12 13 10 33 30 31 32 10′ 11′ 12′ **13′** 30′ 31′ 32′
16	00 01 02 03 20 21 22 23 **00** 01 02 03 22 23 20 21 10 11 12 13 30 31 32 33 **11** 12 13 10 33 30 31 32	32	00 01 02 03 22 23 20 21 **00′** 01′ 02′ 03′ 20′ 21′ 22′ 23′ 11 12 13 10 33 30 31 32 10′ **11′** 12′ 13′ 30′ 31′ 32′ 33′

to 29.82Mbps by scaling the 8-bit to a 16-bit datapath and by directly computing the $(\chi)^4$ matrix. It is noteworthy to point out that our SRL16 based implementation on Artix-7 FPGA only occupies 40 slices for LED-64 and 50 slices for LED-128, respectively, with a throughput almost three times increased compared to Spartan-3 devices.

We also give in Table 3 the performance of existing FPGA implementations of some other lightweight block ciphers. As can be seen from the table, our work seems to require much less area than most ciphers [40, 27, 21, 10, 28] while having a higher throughput than AES [28] implementations and also yields a better throughput per area ratio (Eff.) compared to most ciphers [27, 28]. Compared to FPGA implementations of the lightweight block cipher SIMON [3], we get bigger area requirements but for a higher throughput (and also achieves the better throughput per area ratio (Eff.) when using direct matrix $(\chi)^4$). We remark that HIGHT [40] has a better throughput per area ratio than LED, but in this article our goal with serialised implementations is to reduce area, and not to improve throughput per area ratio. More importantly, one can see in the table that our SRL16 implementation technique both saves area and increases throughput compared to a classical optimized serial implementation. Therefore, we believe this technique is very interesting in order to implement serial-matrix based cryptographic primitives in FPGA technology.

4 PHOTON implementations

In this section, we present three different architectures for the FPGA implementation of the lightweight hash function PHOTON. As in the previous section, the first architecture is a round-based implementation, the second one a fully serialized implementation, and the third one our new serial architecture based on SRL16s. The diffusion layer in PHOTON is based on a similar serial MDS matrix as in LED, thus we also tested different trade-offs concerning its implementation. We have implemented all PHOTON versions (PHOTON-80/20/16, PHOTON-128/16/16, PHOTON-160/36/36, PHOTON-224/32/32 and PHOTON-256/32/32) in VerilogHDL and targeted Xilinx FPGAs Spartan-3 [23] and Artix-7 [25]. Again, we used Mentor Graphics ModelSimPE for simulation purposes and Xilinx ISE v14.4 WebPACK for design synthesis.

4.1 Round-based

In order to fully implement the sponge construction, the input data must be padded according to the sponge padding rule [19], and this is handled by the padding unit. A 2×1 multiplexer drives r bits of the data input from message registers and applies the XOR operation with r bits of the input blocks. After the padding procedure, this multiplexer operates as a feedback multiplexer in order to apply the 12 rounds of the internal permutation of PHOTON. The data register Treg is updated every round, that is after processing *AddConstants*, *SubCells*, *ShiftRows*, and *MixColumnsSerial* in one clock cycle. Another 2×1 multiplexer is devoted to drive either the IV value or the internal state. Finally, during the squeezing phase, r' bits are output from the internal state after every application of the permutation P, until the length of the hash digest size n is reached.

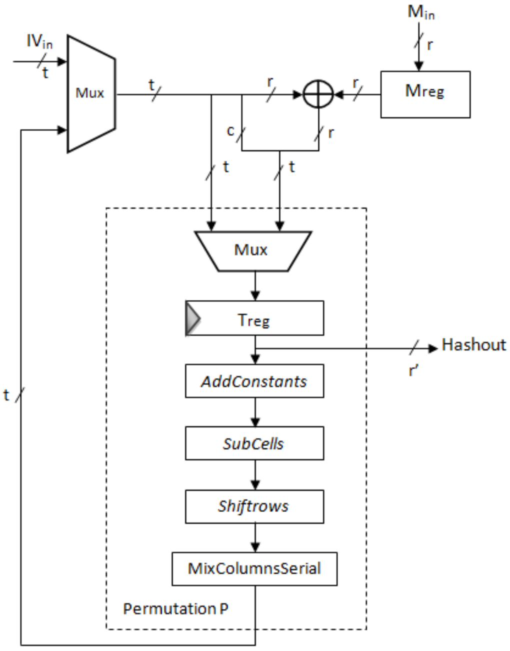

Fig. 5. Architecture of the PHOTON round-based implementations.

The round-based hardware architecture of the PHOTON hash function implementations is shown in Figure 5. The architectures were optimized for high throughput and minimal FPGA area resource consumption. The resulting design fits in the smallest Xilinx devices such as Spartan-3 XC3S50 for variants PHOTON-80/20/16, PHOTON-128/16/16 and Spartan-3 XC3S400 for variants PHOTON-160/36/36, PHOTON-224/32/32 and PHOTON-256/32/32 (because Spartan-3 XC3S50 has only 768 Slices). The major interest was to examine if this method is appropriate to obtain a high throughput implementation of PHOTON hash function. In Table 5, our results are compared to other hardware implementations [1, 4]. One can see that our proposed round-based implementations outperform all the previous works in terms of throughput per area ratio (Eff.).

4.2 Serialized

Similarly to our work on LED in Section 3.2, we have built a serialized implementation of the different PHOTON versions. One can see in Figure 6 that our serialized implementation consists of 6 modules: MCS, IO, AC, SC,

Table 5. FPGA round-based implementation results of PHOTON hash function.

Design	MDS approach	Data-path (bits)	Area (slices)	No. of FFs	No. of LUTs	Clock Cycles	Max. freq (MHz)	T/put (Mbps)	Eff. (Mbps/ slices)	FPGA Device
PHOTON-80/20/16	(χ)	100	285	127	565	12	78.53	130.88	0.46	Spartan-3 XC3S50-5
	(χ)	100	142	111	336	12	232.65	387.75	2.73	Artix-7 XC7A100T-3
SPONGENT-88 [1]		88	157	—	—	45	—	17.78	0.11	Spartan-3
PHOTON-128/16/16	(χ)	144	549	172	1022	12	65.39	87.19	0.16	Spartan-3 XC3S50-5
	(χ)	144	204	156	590	12	189.03	252.04	1.24	Artix-7 XC7A100T-3
SPONGENT-128 [1]		136	208	—	—	70	—	11.43	0.06	Spartan-3
PHOTON-160/36/36	(χ)	196	846	243	1534	12	61.03	183.09	0.22	Spartan-3 XC3S400-5
	(χ)	196	429	207	800	12	155.75	467.25	1.10	Artix-7 XC7A100T-3
SPONGENT-160 [1]		176	264	—	—	90	—	8.89	0.03	Spartan-3
PHOTON-224/32/32	(χ)	256	1235	279	2241	12	51.73	137.95	0.11	Spartan-3 XC3S400-5
	(χ)	256	616	267	1292	12	150.79	402.11	0.65	Artix-7 XC7A100T-3
SPONGENT-224 [1]		240	322	—	—	120	—	6.67	0.02	Spartan-3
PHOTON-256/32/32	(χ)	288	2067	300	3673	12	35.34	94.24	0.05	Spartan-3 XC3S400-5
	(χ)	288	865	300	2126	12	112.43	299.81	0.35	Artix-7 XC7A100T-3
SPONGENT-256 [1]		272	357	—	—	140	—	5.71	0.02	Spartan-3
CUBEHASH-256 [4]		—	2883	—	—	—	59	50	0.017	Spartan-3 XC3S5000-5

ShR, and Controller. These modules and the general hardware architecture that we propose are almost the same as the one described in [19] for ASICs. Yet, we applied the same optimization for *MixColumnSerial* that we have described for LED in detail in Section 3.2. It takes $d \cdot d$ clock cycles to perform *MixColumnsSerial* operation instead of the usual $d \cdot (d+1)$ clock cycles [19].

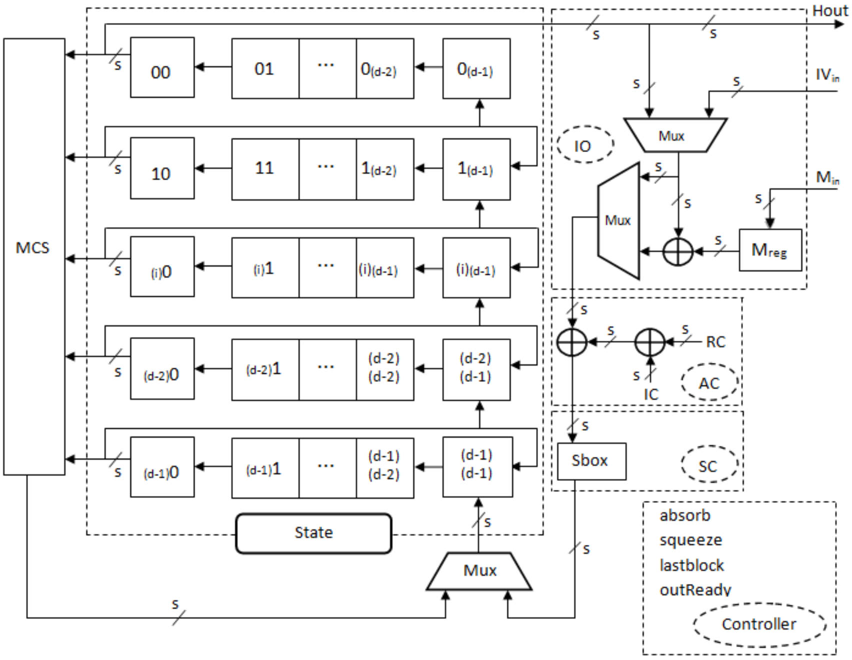

Fig. 6. A serialized architecture of the PHOTON hash function.

Overall, our implementation requires $d \cdot d + (d-1) + d \cdot d$ clock cycles to perform one round of the PHOTON internal permutation P, instead of the $d \cdot d + (d-1) + d \cdot (d+1)$ clock cycles required in [19]. Therefore, we obtain a total latency of $12 \cdot (2 \cdot d \cdot d + d - 1)$ clock cycles, which is $12 \cdot d$ clock cycles faster. We give in Table 6 our implementation results for PHOTON-80/20/16, PHOTON-128/16/16, PHOTON-160/36/36, PHOTON-224/32/32 and PHOTON-256/32/32. One can see that when compared to previous FPGA implementations [13, 32, 12, 26, 29], we have greatly reduced the area and increased the throughput (and also obtained a better throughput per area ratio (Eff.)) as compared to PHOTON-80/20/16 [13] and PHOTON-128/16/16 [32]. Compared to FPGA

Table 6. FPGA serialized implementation results of the PHOTON hash function.

Design	impl. approach	MDS approach	Data-path (bits)	Area (slices)	No. of FFs	No. of LUTs	Clock Cycles	Max. freq (MHz)	T/put (Mbps)	Eff. (Mbps/ slices)	FPGA Device
PHOTON-80/20/16	serial	(χ)	4	146	137	256	648	100.43	3.10	0.02	Spartan-3 XC3S50-5
	SRL16	(χ)	20	112	68	203	360	118.19	6.57	0.06	
	serial	(χ)	4	67	134	167	648	329.51	10.17	0.15	Artix-7 XC7A100T-3
	SRL16	(χ)	20	58	89	144	360	329.95	18.33	0.32	
	serial	(χ)	4	82	135	188	648	302.68	9.34	0.11	Virtex-5 XC5VLX50-1
	SRL16	(χ)	20	69	89	159	360	285.2	15.84	0.22	
PHOTON-80/20/16 [13]			4	149	—	—	708	250	7	0.05	Virtex-5
SPONGENT-88 [1]			4	116	—	—	900	—	.81	0.01	Spartan-3
PHOTON-128/16/16	serial	(χ)	4	183	173	317	924	101.60	1.76	0.01	Spartan-3 XC3S50-5
	SRL16	(χ)	24	137	79	250	504	115.67	3.67	0.03	
	serial	(χ)	4	84	179	212	924	360.33	6.24	0.07	Artix-7 XC7A100T-3
	SRL16	(χ)	24	72	99	137	504	342.36	10.87	0.20	
PHOTON-128/16/16 [32]			4	469	—	—	948	30.2	.551	0.001	Spartan-3
SPONGENT-128 [1]			4	144	—	—	2380	—	.34	0.002	Spartan-3
PHOTON-160/36/36	serial	(χ)	4	233	257	407	1248	72.4	2.01	0.01	Spartan-3 XC3S50-5
	SRL16	(χ)	28	164	138	314	672	122.76	6.58	0.04	
	serial	(χ)	4	117	252	296	1248	328.52	9.47	0.08	Artix-7 XC7A100T-3
	SRL16	(χ)	28	89	135	204	672	328.17	17.58	0.20	
SPONGENT-160 [1]			4	193	—	—	3960	—	.2	0.001	Spartan-3
PHOTON-224/32/32	serial	(χ)	4	274	311	493	1620	69.04	1.36	0.005	Spartan-3 XC3S50-5
	SRL16	(χ)	32	176	135	339	864	123.33	4.57	0.03	
	serial	(χ)	4	130	305	328	1620	382.52	7.55	0.06	Artix-7 XC7A100T-3
	SRL16	(χ)	32	96	131	181	864	327.28	12.12	0.13	
SPONGENT-224 [1]			4	225	—	—	7200	—	.11	0.0005	Spartan-3
GRØSTL-224 [26]			64	1276	—	—	—	60	192	—	Spartan-3
PHOTON-256/32/32	serial	(χ)	8	327	335	577	924	42.56	1.47	0.004	Spartan-3 XC3S50-5
	SRL16	(χ)	48	416	160	806	504	58.95	3.74	0.009	
	serial	(χ)	8	157	331	373	924	132.49	4.59	0.03	Artix-7 XC7A100T-3
	SRL16	(χ)	48	159	100	384	504	166.41	10.75	0.07	
SPONGENT-256 [1]			4	241	—	—	9520	—	0.08	.0003	Spartan-3 XC3S200-5
SHABAL-256 [12]			—	499	—	—	—	100	.8	1.60	Spartan-3 XC3S200-5
BLAKE-256 [29]			—	631	—	—	—	—	216.3	0.34	Spartan-3 XC3S50-5
GRØSTL [29]			—	766	—	—	—	—	192.6	0.25	Spartan-3 XC3S50-5
JH [29]			—	558	—	—	—	—	63.7	0.11	Spartan-3 XC3S50-5
KECCAK [29]			—	766	—	—	—	—	46.2	0.06	Spartan-3 XC3S50-5
SKEIN [29]			—	766	—	—	—	—	16.6	0.02	Spartan-3 XC3S50-5
SHA-2 [29]			—	745	—	—	—	—	137.8	0.19	Spartan-3 XC3S50-5

implementations [1] of the lightweight hash function SPONGENT [6], we get bigger area requirements but for a much higher throughput per area (Eff.). We will see in the next section that SRL16 based implementations of PHOTON will lead to lower area and much higher throughput and yield a better throughput per area ratio (Eff.) than SPONGENT.

4.3 Serialized using SRL16s

As for LED in Section 3.3, we considered a second serialized implementation of PHOTON hash function based on the use of SRL16s [24]. Our architecture is based on a 20-bit datapath that uses χ. It it is depicted in Figure 7 and consists of 3 states: Init, SrSc and MCS, where the content of each SRL16 is indicated in Table 11 of Appendix B for all the state operations.

The *Init* state: after the padding procedure, the IV value is stored into the data SRL16s ($z = s \cdot d$ bits) using a 3×1 multiplexer which drives either the IV input value, updates SrSc state value, or updates MCS state value.

The *SrSc* state: it reads the data values from SRL16s by selecting address taps according to the ShiftRows positions. The round operation starts by bitwise XORing the incoming data with r bits of the message input if applicable, and then adding the constants (round constants and internal constants). Next, the result goes through d S-boxes for a z-bit datapath. Finally, the output of the 4-bit S-boxes is given as input to the blocks 00, 11, 22, 33 and 44 of SRL16s for PHOTON-80/20/16, to the blocks 00, 11, 22, 33, 44 and 55 of SRL16s for PHOTON-128/16/16 and PHOTON-256/32/32, to the blocks 00, 11, 22, 33, 44, 55 and 66 of SRL16s for PHOTON-160/36/36 and to the blocks 00, 11, 22, 33, 44, 55, 66 and 77 of SRL16s for PHOTON-224/32/32. Thus, it takes

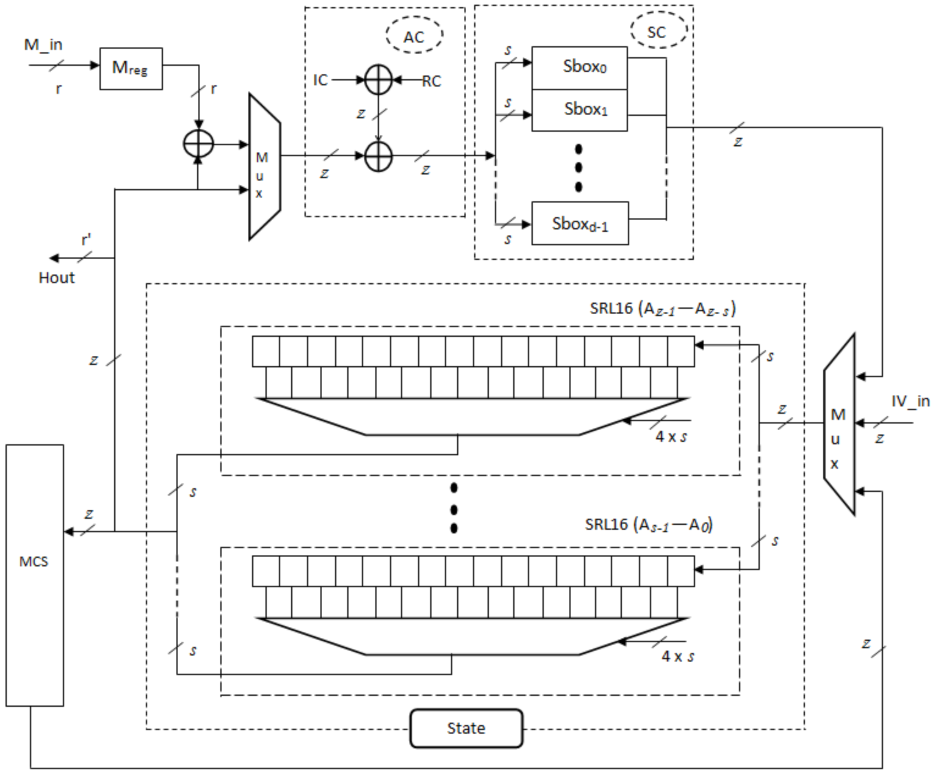

Fig. 7. A serialized architecture of the PHOTON hash function based on SRL16s

d clock cycles (clk 6-10 in Table 11 for PHOTON-80/20/16) for a z-bit datapath to perform *AddConstants*, *ShiftRows* and *SubCells* operations on the entire state.

The *MCS* state: the z-bit data is read from the bits indicated in bold in Table 11 for PHOTON-80/20/16. It starts with the five 4-bit blocks 00, 11, 22, 33 and 44, and using (χ), the resulting 20-bit output is stored in the SRL16s labeled as 11, 22, 33, 44 and $00'$. In the next clock cycle, the input is 01, 12, 23, 34 and 40, and the corresponding result is labeled as 12, 23, 34, 40, $01'$ and so on similar to Table 7. In total 25 clock cycles (clk 11-35 in Table 11) are required to complete the *MixColumnsSerial* operation for PHOTON-80/20/16. We have also implemented the remaining 4 versions of PHOTON using same architecture and give below the MCS state input(x) SRL16s labeled and output(y) SRL16s labeled for the first clock cycle.

- **PHOTON-128/16/16:** x = 00, 11, 22, 33, 44, 55; y = 11, 22, 33, 44, 55, $00'$
- **PHOTON-160/36/36:** x = 00, 11, 22, 33, 44, 55, 66; y = 11, 22, 33, 44, 55, 66, $00'$
- **PHOTON-224/32/32:** x = 00, 11, 22, 33, 44, 55, 66, 77 ; y = 11, 22, 33, 44, 55, 66, 77, $00'$
- **PHOTON-256/32/32:** x = 00, 11, 22, 33, 44, 55; y = 11, 22, 33, 44, 55, $00'$

$d \cdot d$ clock cycles are required for a z-bit datapath in order to complete the *MixColumnsSerial* operation. Overall, we require $d + d \cdot d$ clock cycles to compute a single round. Since PHOTON has 12 rounds, the total number of cycles required to process one block of message is $12(d + d \cdot d)$. Table 6 describes the performance results of our implementations and compares it with existing FPGA implementations of PHOTON and other lightweight hash functions. Concerning KECCAK-f[200], perhaps we just add that KECCAK-f[200] is not included in this table as no FPGA implementation of this function has been published so far. As seen from the table, our work provides the smallest area among all known implementations of lightweight hash functions while having a higher throughput and yields a better throughput per area ratio (Eff.) than PHOTON-80/20/16 [13], PHOTON-128/16/16 [32] and the implementation of SPONGENT [1]. We remark that SHABAL [12] has a better throughput per area ratio than PHOTON, but in this article our goal with serialised implementations is to reduce area, and not to improve throughput per area ratio.

5 Conclusion

In this paper, we have analyzed the feasibility of creating a very compact, low cost FPGA implementation of LED and PHOTON. For both primitives, we studied round-based and serial architectures and we implemented several possible tradeoffs when computing the diffusion matrix. In particular, we proposed an SRL16 based architecture, that seems to be very well suited for all cryptographic primitives that use serial matrices. Our results show that LED and PHOTON are very good candidates for lightweight applications, our implementations yield for example the best area of all lightweight hash functions implementations published so far. Future work will include the investigation of side-channel analysis on our implementations and apply countermeasures [18, 34, 33] in order to resist these attacks.

Acknowledgements

Authors would like to thank the anonymous reviewers for their helpful comments. The first author wishes to thank Prof. R. Balsubramanian, Executive Director (SETS) and Sri. S. Thiagarajan, Registrar (SETS) for their support. Thomas Peyrin is supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Marwan Adas. On The FPGA Based Implementation of SPONGENT, 2011. http://ece.gmu.edu/coursewebpages/ECE/ECE646/F11/project/F11_presentations/Marwan.pdf.
2. Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A Lightweight Hash. *Journal of Cryptology*, 26(2):313–339, 2013.
3. Aydin Aysu, Ege Gulcan, and Patrick Schaumont. SIMON Says, Break the Area Records for Symmetric Key Block Ciphers on FPGAs. *IACR Cryptology ePrint Archive*, 2014, Available at:. http://eprint.iacr.org/2014/237.
4. Brian Baldwin, Andrew Byrne, Mark Hamilton, Neil Hanley, Robert P McEvoy, Weibo Pan, and William P Marnane. FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 783–790. IEEE, 2009.
5. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. *IACR Cryptology ePrint Archive*, 2013, Available at:. http://eprint.iacr.org/2013/404.
6. Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. SPONGENT: A Lightweight Hash Function. In *Cryptographic Hardware and Embedded Systems–CHES*, pages 312–325. Springer, 2011.
7. Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 450–466. Springer, 2007.
8. Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In *Progress in Cryptology–AFRICACRYPT 2008*, pages 16–26. Springer, 2008.
9. David Canright. A Very Compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 441–455. Springer, 2005.
10. Junfeng Chu and Mohammed Benaissa. Low area memory-free FPGA implementation of the AES algorithm. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 623–626. IEEE, 2012.
11. Christophe De Canniere, Orr Dunkelman, and Miroslav Knežević. KATAN and KTANTAN – A Family of Small and Efficient Hardware-Oriented Block Ciphers. In *CHES 2009*, pages 272–288. Springer, 2009.
12. Jérémie Detrey, Pierrick Gaudry, and Karim Khalfallah. A Low-Area Yet Performant FPGA Implementation of Shabal. In *Selected Areas in Cryptography - 17th International Workshop, SAC*, pages 99–113. Springer, 2010.
13. Susana Eiroa and Iluminada Baturone. FPGA implementation and DPA resistance analysis of a lightweight HMAC construction based on photon hash family. In *FPL*, pages 1–4. IEEE, 2013.
14. Andreas Engel, Björn Liebig, and Andreas Koch. Feasibility Analysis of Reconfigurable Computing in Low-Power Wireless Sensor Applications. In *ARC*, pages 261–268. Springer, 2011.
15. Martin Feldhofer, Manfred Josef Aigner, Thomas Baier, Michael Hutter, Thomas Plos, and Erich Wenger. Semi-passive RFID development platform for implementing and attacking security tags. In *ICITST*, pages 1–6. IEEE, 2010.
16. Zheng Gong, Svetla Nikova, and Yee Wei Law. KLEIN: A New Family of Lightweight Block Ciphers. In *RFID. Sec*, pages 1–18. Springer, 2012.
17. Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 427–440. Springer, 2005.
18. Tim Güneysu and Amir Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 33–48. Springer, 2011.
19. Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON Family of Lightweight Hash Functions. In *Advances in Cryptology–CRYPTO 2011*, pages 222–239. Springer, 2011.

20. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED Block Cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.
21. Xu Guo, Zhimin Chen, and Patrick Schaumont. Energy and Performance Evaluation of an FPGA-Based SoC Platform with AES and PRESENT Coprocessors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 106–115. Springer, 2008.
22. Xilinx Inc. AN 307: Altera Design Flow for Xilinx Users, March, 2013. Available at:. `http://www.altera.com/literature/an/an307.pdf`.
23. Xilinx Inc. Spartan-3 Generation FPGA User Guide, August, 2010. Available at:. `http://www.xilinx.com/support/documentation/user_guides/ug331.pdf/`.
24. Xilinx Inc. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs, May, 2005. Available at:. `http://www.xilinx.com/support/documentation/application_notes/xapp465.pdf`.
25. Xilinx Inc. Xilinx 7 Series FPGAs FPGA User Guide, February, 2014. Available at:. `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.
26. Bernhard Jungk and Steffen Reith. On FPGA-based implementations of Grøstl. In *IACR Cryptology ePrint Archive*, volume 2010, Available at:. `http://eprint.iacr.org/2010/260`.
27. Jens-Peter Kaps. Chai-Tea, Cryptographic Hardware Implementations of xTEA. In *Progress in Cryptology-INDOCRYPT 2008*, pages 363–375. Springer, 2008.
28. Jens-Peter Kaps and Berk Sunar. Energy Comparison of AES and SHA-1 for Ubiquitous Computing. In *EUC Workshops*, pages 372–381. Springer, 2006.
29. Jens-Peter Kaps, Panasayya Yalla, Kishore Kumar Surapathi, Bilal Habib, Susheel Vadlamudi, and Smriti Gurung. Lightweight Implementations of SHA-3 Candidates on FPGAs. In *The Third SHA-3 Candidate Conference*, 2012.
30. Lars Knudsen, Gregor Leander, Axel Poschmann, and Matthew JB Robshaw. PRINTcipher: A Block Cipher for IC-Printing. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 16–32. Springer, 2010.
31. François Macé, François-Xavier Standaert, and Jean-Jacques Quisquater. FPGA Implementation(s) of a Scalable Encryption Algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):212–216, 2007.
32. Pavan Kumar Malka. Compact Hardware Implementation of PHOTON Hash Function in FPGA, 2011. `http://ece.gmu.edu/coursewebpages/ECE/ECE646/F11/project/F11_presentations/Pavan.pdf`.
33. Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Advances in Cryptology–EUROCRYPT 2011*, pages 69–88. Springer, 2011.
34. Axel Poschmann, Amir Moradi, Khoongming Khoo, Chu-Wee Lim, Huaxiong Wang, and San Ling. Side-Channel Resistant Crypto for Less than 2,300 GE. *Journal of Cryptology*, 24(2):322–345, 2011.
35. Axel York Poschmann. LIGHTWEIGHT CRYPTOGRAPHY: Cryptographic Engineering for a Pervasive World. In *PH. D. THESIS*. Citeseer, 2009.
36. Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: An Ultra-Lightweight Blockcipher. In *Cryptographic Hardware and Embedded Systems–CHES*, pages 342–357. Springer, 2011.
37. François-Xavier Standaert, Gilles Piret, Gaël Rouvroy, and Jean-Jacques Quisquater. FPGA implementations of the ICEBERG block cipher. *Integration, the VLSI Journal*, 40(1):20–27, 2007.
38. Tim Tuan, Arif Rahman, Satyaki Das, Steven Trimberger, and Sean Kao. A 90-nm Low-Power FPGA for Battery-Powered Applications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):296–300, 2007.
39. Wenling Wu and Lei Zhang. LBlock: A Lightweight Block Cipher. In *Applied Cryptography and Network Security*, pages 327–344. Springer, 2011.
40. Panasayya Yalla and Jens-Peter Kaps. Lightweight Cryptography for FPGAs. In *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, pages 225–230. IEEE, 2009.

A SRL16s positions for `LED`

Table 7. Content of SRL16s after every state of `LED` when using (χ) for the 16-bit datapath. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation. The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s
	Init		*SrSc*
1	00 10 20 30	5	00 **01** 02 03 00 10 11 **12** 13 11 20 21 22 **23** 22 **30** 31 32 33 33
2	00 01 10 11 20 21 30 31	6	00 01 **02** 03 00 01 10 11 12 **13** 11 12 **20** 21 22 23 22 23 30 **31** 32 33 33 30
3	00 01 02 10 11 12 20 21 22 30 31 32	7	00 01 02 **03** 00 01 02 **10** 11 12 13 11 12 13 20 **21** 22 23 22 23 20 30 31 **32** 33 33 30 31
4	**00** 01 02 03 10 **11** 12 13 20 21 **22** 23 30 31 32 **33**	8	00 01 02 03 **00** 01 02 03 10 11 12 13 **11** 12 13 10 20 21 22 23 **22** 23 20 21 30 31 32 33 **33** 30 31 32
	MCS		*MCS*
9	00 01 02 03 00 **01** 02 03 11 10 11 12 13 11 **12** 13 10 22 20 21 22 23 22 **23** 20 21 33 30 31 32 33 33 **30** 31 32 00′	17	01 02 03 00 01 02 03 11 12 13 10 22 **23** 20 21 33 11 12 13 11 12 13 10 22 23 20 21 33 **30** 31 32 00′ 21 22 23 22 23 20 21 33 30 31 32 00′ **01′** 02′ 03′ 10′ 31 32 33 33 30 31 32 00′ 01′ 02′ 03′ 10′ **11′** 12′ 13′ 20′
10	00 01 02 03 00 01 **02** 03 11 12 10 11 12 13 11 12 **13** 10 22 23 20 21 22 23 22 23 **20** 21 33 30 30 31 32 33 33 30 **31** 32 00′ 01′	18	02 03 00 01 02 03 11 12 13 10 22 23 **20** 21 33 30 12 13 11 12 13 10 22 23 20 21 33 30 **31** 32 00′ 01′ 22 23 22 23 20 21 33 30 31 32 00′ 01′ **02′** 03′ 10′ 11′ 32 33 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ **12′** 13′ 20′ 21′
11	00 01 02 03 00 01 02 **03** 11 12 13 10 11 12 13 11 12 13 **10** 22 23 20 20 21 22 23 22 23 20 **21** 33 30 31 30 31 32 33 33 30 31 **32** 00′ 01′ 02′	19	03 00 01 02 03 11 12 13 10 22 23 20 **21** 33 30 31 13 11 12 13 10 22 23 20 21 33 30 31 **32** 00′ 01′ 02′ 23 22 23 20 21 33 30 31 32 00′ 01′ 02′ **03′** 10′ 11′ 12′ 33 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ **13′** 20′ 21′ 22′
12	00 01 02 03 00 01 02 03 **11** 12 13 10 10 11 12 13 11 12 13 10 **22** 23 20 21 20 21 22 23 22 23 20 21 **33** 30 31 32 30 31 32 33 33 30 31 32 **00′** 01′ 02′ 03′	20	00 01 02 03 11 12 13 10 22 23 20 21 **33** 30 31 32 11 12 13 10 22 23 20 21 33 30 31 32 **00′** 01′ 02′ 03′ 22 23 20 21 33 30 31 32 00′ 01′ 02′ 03′ **10′** 11′ 12′ 13′ 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ **20′** 21′ 22′ 23′
13	00 01 02 03 00 01 02 03 11 **12** 13 10 22 10 11 12 13 11 12 13 10 22 **23** 20 21 33 20 21 22 23 22 23 20 21 33 **30** 31 32 00′ 30 31 32 33 33 30 31 32 00′ **01′** 02′ 03′ 10′	21	01 02 03 11 12 13 10 22 23 20 21 33 **30** 31 32 00′ 12 13 10 22 23 20 21 33 30 31 32 00′ **01′** 02′ 03′ 10′ 23 20 21 33 30 31 32 00′ 01′ 02′ 03′ 04′ **11′** 12′ 13′ 20′ 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ 20′ **21′** 22′ 23′ 30′
14	00 01 02 03 00 01 02 03 11 12 **13** 10 22 23 10 11 12 13 11 12 13 10 22 23 **20** 21 33 30 20 21 22 23 22 23 20 21 33 30 **31** 32 00′ 01′ 30 31 32 33 33 30 31 32 00′ 01′ **02′** 03′ 10′ 11′	22	02 03 11 12 13 10 22 23 20 21 33 30 **31** 32 00′ 01′ 13 10 22 23 20 21 33 30 31 32 00′ 01′ **02′** 03′ 10′ 11′ 20 21 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ **12′** 13′ 20′ 21′ 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ 20′ 21′ **22′** 23′ 30′ 31′
15	00 01 02 03 00 01 02 03 11 12 13 **10** 22 23 20 10 11 12 13 11 12 13 10 22 23 20 **21** 33 30 31 20 21 22 23 22 23 20 21 33 30 31 **32** 00′ 01′ 02′ 30 31 32 33 33 30 31 32 00′ 01′ 02′ **03′** 10′ 11′ 12′	23	03 11 12 13 10 22 23 20 21 33 30 31 **32** 00′ 01′ 02′ 10 22 23 20 21 33 30 31 32 00′ 01′ 02′ **03′** 10′ 11′ 12′ 21 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ **13′** 20′ 21′ 22′ 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ 20′ 21′ 22′ **23′** 30′ 31′ 32′
16	00 01 02 03 00 01 02 03 11 12 13 10 **22** 23 20 21 10 11 12 13 11 12 13 10 22 23 20 21 **33** 30 31 32 20 21 22 23 22 23 20 21 33 30 31 32 **00′** 01′ 02′ 03′ 30 31 32 33 33 30 31 32 00′ 01′ 02′ 03′ **10′** 11′ 12′ 13′	24	11 12 13 10 22 23 20 21 33 30 31 32 **00′** 01′ 02′ 03′ 22 23 20 21 33 30 31 32 00′ 01′ 02′ 03′ 10′ **11′** 12′ 13′ 33 30 31 32 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ 20′ 21′ **22′** 23′ 00′ 01′ 02′ 03′ 10′ 11′ 12′ 13′ 20′ 21′ 22′ 23′ 30′ 31′ 32′ **33′**

Table 8. Content of SRL16s after every state of `LED` when using $(\chi)^4$ for the 16-bit datapath. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation. The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s	clk	content of SRL16s
	Init		*SrSc*		*MCS*
1	00 10 20 30	5	00 **01** 02 03 00 10 11 **12** 13 11 20 21 22 **23** 22 **30** 31 32 33 33	9	00 01 02 03 00 **01** 02 03 00′ 10 11 12 13 11 **12** 13 10 10′ 20 21 22 23 22 **23** 20 21 20′ 30 31 32 33 33 **30** 31 32 30′
2	00 01 10 11 20 21 30 31	6	00 01 **02** 03 00 01 10 11 12 **13** 11 12 **20** 21 22 23 22 23 30 **31** 32 33 33 30	10	00 01 02 03 00 01 **02** 03 00′ 01′ 10 11 12 13 11 12 **13** 10 10′ 11′ 20 21 22 23 22 23 **20** 21 20′ 21′ 30 31 32 33 33 30 **31** 32 30′ 31′
3	00 01 02 10 11 12 20 21 22 30 31 32	7	00 01 02 **03** 00 01 02 **10** 11 12 13 11 12 13 20 **21** 22 23 22 23 20 30 31 **32** 33 33 30 31	11	00 01 02 03 00 01 02 **03** 00′ 01′ 02′ 10 11 12 13 11 12 13 **10** 10′ 11′ 12′ 20 21 22 23 22 23 20 **21** 20′ 21′ 22′ 30 31 32 33 33 30 31 **32** 30′ 31′ 32′
4	**00** 01 02 03 10 **11** 12 13 20 21 **22** 23 30 31 32 **33**	8	00 01 02 03 **00** 01 02 03 10 11 12 13 **11** 12 13 10 20 21 22 23 **22** 23 20 21 30 31 32 33 **33** 30 31 32	12	00 01 02 03 00 01 02 03 **00′** 01′ 02′ 03′ 10 11 12 13 11 12 13 10 10′ **11′** 12′ 13′ 20 21 22 23 22 23 20 21 20′ 21′ **22′** 23′ 30 31 32 33 33 30 31 32 30′ 31′ 32′ **33′**

Table 9. Content of KEY SRL16s after every state of `LED`-64 when using $(\chi)^2$ for the 8-bit datapath. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation (see also Figure 4). The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s
	Init		*Init*
1	00	9	00 01 02 03 22 23 20 21 11
2	00 01	10	00 01 02 03 22 23 20 21 11 12
3	00 01 02	11	00 01 02 03 22 23 20 21 11 12 13
4	00 01 02 03	12	00 01 02 03 22 23 20 21 11 12 13 10
5	00 01 02 03 22	13	00 01 02 03 22 23 20 21 11 12 13 10 33
6	00 01 02 03 22 23	14	00 01 02 03 22 23 20 21 11 12 13 10 33 30
7	00 01 02 03 22 23 20	15	00 01 02 03 22 23 20 21 11 12 13 10 33 30 31
8	00 01 02 03 22 23 20 21	16	**00** 01 02 03 22 23 20 21 **11** 12 13 10 33 30 31 32
	SrSc		*MCS(Rotate)*
17	**01** 02 03 22 23 20 21 11 **12** 13 10 33 30 31 32 00	25	**12** 13 10 33 30 31 32 00 **01** 02 03 22 23 20 21 11
18	**02** 03 22 23 20 21 11 12 **13** 10 33 30 31 32 00 01	26	**13** 10 33 30 31 32 00 01 **02** 03 22 23 20 21 11 12
19	**03** 22 23 20 21 11 12 13 **10** 33 30 31 32 00 01 02	27	**10** 33 30 31 32 00 01 02 **03** 22 23 20 21 11 12 13
20	**22** 23 20 21 11 12 13 10 **33** 30 31 32 00 01 02 03	28	**33** 30 31 32 00 01 02 03 **22** 23 20 21 11 12 13 10
21	**23** 20 21 11 12 13 10 33 **30** 31 32 00 01 02 03 22	29	**30** 31 32 00 01 02 03 22 **23** 20 21 11 12 13 10 33
22	**20** 21 11 12 13 10 33 30 **31** 32 00 01 02 03 22 23	30	**31** 32 00 01 02 03 22 23 **20** 21 11 12 13 10 33 30
23	**21** 11 12 13 10 33 30 31 **32** 00 01 02 03 22 23 20	31	**32** 00 01 02 03 22 23 20 **21** 11 12 13 10 33 30 31
24	**11** 12 13 10 33 30 31 32 **00** 01 02 03 22 23 20 21	32	**00′** 01 02 03 22 23 20 21 **11′** 12 13 10 33 30 31 32

Table 10. Content of KEY SRL16s when using $(\chi)^4$ or (χ) for the 16-bit datapath. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation. The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s
	Init		*Init*
1	00 10 20 30	9	00 01 02 03 00 01 02 03 00 10 11 12 13 10 11 12 13 10 20 21 22 23 20 21 22 23 20 30 31 32 33 30 31 32 33 30
2	00 01 10 11 20 21 30 31	10	00 01 02 03 00 01 02 03 00 01 10 11 12 13 10 11 12 13 10 11 20 21 22 23 20 21 22 23 20 21 30 31 32 33 30 31 32 33 30 31
3	00 01 02 10 11 12 20 21 22 30 31 32	11	00 01 02 03 00 01 02 03 00 01 02 10 11 12 13 10 11 12 13 10 11 12 20 21 22 23 20 21 22 23 20 21 22 30 31 32 33 30 31 32 33 30 31 32
4	00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33	12	00 01 02 03 00 01 02 03 00 01 02 03 10 11 12 13 10 11 12 13 10 11 12 13 20 21 22 23 20 21 22 23 20 21 22 23 30 31 32 33 30 31 32 33 30 31 32 33
5	00 01 02 03 00 10 11 12 13 10 20 21 22 23 20 30 31 32 33 30	13	00 01 02 03 00 01 02 03 00 01 02 03 00 10 11 12 13 10 11 12 13 10 11 12 13 10 20 21 22 23 20 21 22 23 20 21 22 23 20 30 31 32 33 30 31 32 33 30 31 32 33 30
6	00 01 02 03 00 01 10 11 12 13 10 11 20 21 22 23 20 21 30 31 32 33 30 31	14	00 01 02 03 00 01 02 03 00 01 02 03 00 01 10 11 12 13 10 11 12 13 10 11 12 13 10 11 20 21 22 23 20 21 22 23 20 21 22 23 20 21 30 31 32 33 30 31 32 33 30 31 32 33 30 31
7	00 01 02 03 00 01 02 10 11 12 13 10 11 12 20 21 22 23 20 21 22 30 31 32 33 30 31 32	15	00 01 02 03 00 01 02 03 00 01 02 03 00 01 02 10 11 12 13 10 11 12 13 10 11 12 13 10 11 12 20 21 22 23 20 21 22 23 20 21 22 23 20 21 22 30 31 32 33 30 31 32 33 30 31 32 33 30 31 32
8	00 01 02 03 00 01 02 03 10 11 12 13 10 11 12 13 20 21 22 23 20 21 22 23 30 31 32 33 30 31 32 33	16	00 01 02 03 00 01 02 03 **00** 01 02 03 00 01 02 03 10 11 12 13 10 11 12 13 10 **11** 12 13 10 11 12 13 20 21 22 23 20 21 22 23 20 21 **22** 23 20 21 22 23 30 31 32 33 30 31 32 33 30 31 32 **33** 30 31 32 33
	SrSc		*SrSc*
17	01 02 03 00 01 02 03 00 **01** 02 03 00 01 02 03 00 11 12 13 10 11 12 13 10 11 **12** 13 10 11 12 13 10 21 22 23 20 21 22 23 20 21 22 **23** 20 21 22 23 20 31 32 33 30 31 32 33 **30** 31 32 33 30 31 32 33 30	19	03 00 01 02 03 00 01 02 **03** 00 01 02 03 00 01 02 13 10 11 12 13 **10** 11 12 13 10 11 12 13 10 11 12 23 20 21 22 23 20 **21** 22 23 20 21 22 23 20 21 22 33 30 31 32 33 30 31 **32** 33 30 31 32 33 30 31 32
18	02 03 00 01 02 03 00 01 **02** 03 00 01 02 03 00 01 12 13 10 11 12 13 10 11 12 **13** 10 11 12 13 10 11 22 23 20 21 22 23 **20** 21 22 23 20 21 22 23 20 21 32 33 30 31 32 33 30 **31** 32 33 30 31 32 33 30 31	20	00 01 02 03 00 01 02 03 **00′** 01 02 03 00 01 02 03 10 11 12 13 10 11 12 13 10 **11′** 12 13 10 11 12 13 20 21 22 23 20 21 22 23 20 21 **22′** 23 20 21 22 23 30 31 32 33 30 31 32 33 30 31 32 **33′** 30 31 32 33

B SRL16s positions for PHOTON-80/20/16

Table 11. Indicates the SRL16s position after every state of PHOTON-80/20/16. Every cell of the content shows the index of a nibble of the state. Printed in bold is the input to the subsequent operation (see also Figure 7). The indices of the next round are indicated with a ′.

clk	content of SRL16s	clk	content of SRL16s
	Init		*SrSc*
1	00	6	00 **01** 02 03 04 00
	10		10 11 **12** 13 14 11
	20		20 21 22 **23** 24 22
	30		30 31 32 33 **34** 33
	40		**40** 41 42 43 44 44
2	00 01	7	00 01 **02** 03 04 00 01
	10 11		10 11 12 **13** 14 11 12
	20 21		20 21 22 23 **24** 22 23
	30 31		**30** 31 32 33 34 33 34
	40 41		40 **41** 42 43 44 44 40
3	00 01 02	8	00 01 02 **03** 04 00 01 02
	10 11 12		10 11 12 13 **14** 11 12 13
	20 21 22		**20** 21 22 23 24 22 23 24
	30 31 32		30 **31** 32 33 34 33 34 30
	40 41 42		40 41 **42** 43 44 44 40 41
4	00 01 02 03	9	00 01 02 03 **04** 00 01 02 03
	10 11 12 13		**10** 11 12 13 14 11 12 13 14
	20 21 22 23		20 **21** 22 23 24 22 23 24 20
	30 31 32 33		30 31 **32** 33 34 33 34 30 31
	40 41 42 43		40 41 42 **43** 44 44 40 41 42
5	**00** 01 02 03 04	10	00 01 02 03 04 **00** 01 02 03 04
	10 **11** 12 13 14		10 11 12 13 14 **11** 12 13 14 10
	20 21 **22** 23 24		20 21 22 23 24 **22** 23 24 20 21
	30 31 32 **33** 34		30 31 32 33 34 **33** 34 30 31 32
	40 41 42 43 **44**		40 41 42 43 44 **44** 40 41 42 43
	MCS		*MCS*
11	00 01 02 03 04 00 **01** 02 03 04 11	34	20 21 33 34 30 31 32 44 40 41 42 **43** 00′ 01′ 02′ 03′
	10 11 12 13 14 11 **12** 13 14 10 22		31 32 44 40 41 42 43 00′ 01′ 02′ 03′ **04′** 10′ 11′ 12′ 13′
	20 21 22 23 24 22 **23** 24 20 21 33		42 43 00′ 01′ 02′ 03′ 04′ 10′ 11′ 12′ 13′ **14′** 20′ 21′ 22′ 23′
	30 31 32 33 34 33 **34** 30 31 32 44		03′ 04′ 10′ 11′ 12′ 13′ 14′ 20′ 21′ 22′ 23′ **24′** 30′ 31′ 32′ 33′
	40 41 42 43 44 44 **40** 41 42 43 00′		13′ 14′ 20′ 21′ 22′ 23′ 24′ 30′ 31′ 32′ 33′ **34′** 40′ 41′ 42′ 43′
12	00 01 02 03 04 00 01 **02** 03 04 11 12	35	21 33 34 30 31 32 44 40 41 42 43 **00′** 01′ 02′ 03′ 04′
	10 11 12 13 14 11 12 **13** 14 10 22 23		32 44 40 41 42 43 00′ 01′ 02′ 03′ 04′ 10′ **11′** 12′ 13′ 14′
	20 21 22 23 24 22 23 **24** 20 21 33 34		43 00′ 01′ 02′ 03′ 04′ 10′ 11′ 12′ 13′ 14′ 20′ 21′ **22′** 23′ 24′
	30 31 32 33 34 33 34 **30** 31 32 44 40		04′ 10′ 11′ 12′ 13′ 14′ 20′ 21′ 22′ 23′ 24′ 30′ 31′ 32′ **33′** 34′
	40 41 42 43 44 44 40 **41** 42 43 00′ 01′		14′ 20′ 21′ 22′ 23′ 24′ 30′ 31′ 32′ 33′ 34′ 40′ 41′ 42′ 43′ **44′**