Compact Implementation of Threefish and Skein on FPGA

Nuray At*, Jean-Luc Beuchat†, and İsmail San*
*Department of Electrical and Electronics Engineering, Anadolu University, Eskişehir, Turkey
Email: {nat, isan}@anadolu.edu.tr
†Faculty of Engineering, Information and Systems,
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan
Email: jeanluc.beuchat@gmail.com

***Abstract*—The SHA-3 finalist Skein is built from the tweakable Threefish block cipher. In order to have a better understanding of the computational efficiency of Skein (resource sharing, memory access scheme, scheduling, etc.), we design a low-area coprocessor for Threefish and describe how to implement Skein on our architecture. We harness the intrinsic parallelism of Threefish to design a pipelined ALU and interleave several tasks in order to achieve a tight scheduling. From our point of view, the main advantage of Skein over other SHA-3 finalists is that the same coprocessor allows one to encrypt or hash a message.**

I. Introduction

In this article, we propose a novel hardware implementation of the SHA-3 finalist Skein [1]. As emphasized by Kerckhof *et al.*, "fully unrolled and pipelined architectures may sometimes hide a part of the algorithms' complexity that is better revealed in compact implementations" [2]. In order to have a deeper understanding of the computational efficiency of Skein (resource sharing, memory access scheme, scheduling, etc.), we decided to design a low-area coprocessor on a Field-Programmable Gate Array (FPGA). Furthermore, such an implementation is valuable for constrained environments, where some security protocols mainly rely on cryptographic hash functions (see for instance [3]).

Skein is built from the tweakable Threefish block cipher, defined with a 256-, 512-, and 1024-bit block size. The main contribution of this article is a lightweight implementation of Threefish (Section II). Then, we describe how to implement Skein on our coprocessor (Section III). We have prototyped our architecture on a Xilinx Virtex-6 device and discuss our results in Section IV.

II. The Threefish Block Cipher

A. Algorithm Specification

Threefish operates entirely on unsigned 64-bit integers and involves only three operations: rotation of k bits to the left (denoted by $\lll k$), bitwise exclusive OR (denoted by $\oplus$), and addition modulo 2^{64} (denoted by $\boxplus$). Therefore, the plaintext P and the cipher key K are converted to N_w 64-bit words. Note that the number of words N_w and the number of rounds N_r depend on the key size (Table I).

The key schedule generates the subkeys from a block cipher key $K = (k_0, k_1, \ldots, k_{N_w-1})$ and a 128-bit tweak $T = (t_0, t_1)$. K and T are extended with one parity word (Algorithm 1, lines 1 and 2). Each subkey is a combination of N_w words of the extended key, two words of the extended tweak, and a counter s (Algorithm 1, lines 5 to 9). Note that the extended key and the extended tweak are rotated by one word position between two consecutive subkeys.

Table I
Number of rounds for different key sizes (reprinted from [1]).

Key size [bits]	# words N_w	# rounds N_r
256	4	72
512	8	72
1024	16	80

Algorithm 1 Key schedule.

Input: A block cipher key $K = (k_0, k_1, \ldots, k_{N_w-1})$; a tweak $T = (t_0, t_1)$; the constant C_{240} = `1BD11BDAA9FC1A22`.
Output: $N_r/4 + 1$ subkeys $k_{s,0}$, $k_{s,1}$, ..., k_{s,N_w-1}, where $0 \leq s \leq N_r/4$.

1. $k_{N_w} \leftarrow C_{240} \oplus \bigoplus_{i=0}^{N_w-1} k_i$;
2. $t_2 \leftarrow t \oplus t_1$;
3. **for** $s \leftarrow 0$ **to** $N_r/4$ **do**
4.   **for** $i \leftarrow 0$ **to** $N_w - 4$ **do**
5. $k_{s,i} \leftarrow k_{(s+i) \bmod (N_w+1)}$;
6.   **end for**
7. $k_{s,N_w-3} \leftarrow k_{(s+N_w-3) \bmod (N_w+1)} \boxplus t_{s \bmod 3}$;
8. $k_{s,N_w-2} \leftarrow k_{(s+N_w-2) \bmod (N_w+1)} \boxplus t_{(s+1) \bmod 3}$;
9. $k_{s,N_w-1} \leftarrow k_{(s+N_w-1) \bmod (N_w+1)} \boxplus s$;
10. **end for**
11. **return** $k_{s,0}$, $k_{s,1}$, ..., k_{s,N_w-1}, where $0 \leq s \leq N_r/4$;

A series of N_r rounds (Figure 1 and Algorithm 2, lines 4 to 19) and a final subkey addition (Algorithm 2, line 21) are applied to produce the ciphertext. The core of a round is the simple non-linear mixing function $\mathrm{Mix}_{d,j}$ (Algorithm 2, lines 13 and 14). It consists of an addition, a rotation by a constant $R_{d \bmod 8,j}$ (repeated every eight rounds and defined

in [1, Table 4]), and a bitwise exclusive OR. A word permutation $\pi(i)$ (defined in [1, Table 3]) is then applied to obtain the output of the round (Algorithm 2, line 17). Furthermore, a subkey is injected every four rounds (Algorithm 2, line 7).

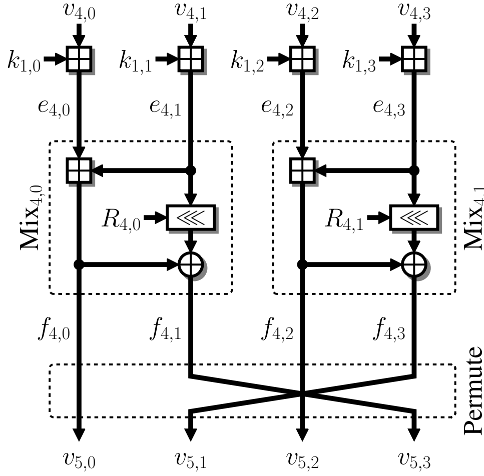


Figure 1. One of the 72 rounds of Threefish-256.

Algorithm 2 Encryption with the Threefish block cipher.

Input: A plaintext block $P = (p_0, p_1, \ldots, p_{N_w-1})$; $N_r/4+1$ subkeys $k_{s,0}$, $k_{s,1}$, $\ldots$, k_{s,N_w-1}, where $0 \leq s \leq N_r/4$; $4N_w$ rotation constants $R_{i,j}$, where $0 \leq i \leq 7$ and $0 \leq j \leq N_w/2$.
Output: A ciphertext block $C = (c_0, c_1, \ldots, c_{N_w-1})$.

1. **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
2. $\quad v_{0,i} \leftarrow p_i$;
3. **end for**
4. **for** $d \leftarrow 0$ **to** $N_r - 1$ **do**
5. $\quad$ **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
6. $\quad\quad$ **if** $d \bmod 4 = 0$ **then**
7. $\quad\quad\quad e_{d,i} \leftarrow v_{d,i} \boxplus k_{d/4,i}$; (Key injection)
8. $\quad\quad$ **else**
9. $\quad\quad\quad e_{d,i} \leftarrow v_{d,i}$; (Rename)
10. $\quad\quad$ **end if**
11. $\quad$ **end for**
12. $\quad$ **for** $j \leftarrow 0$ **to** $N_w/2 - 1$ **do**
13. $\quad\quad f_{d,2j} \leftarrow e_{d,2j} \boxplus e_{d,2j+1}$; ($\text{Mix}_{d,j}$)
14. $\quad\quad f_{d,2j+1} \leftarrow f_{d,2j} \oplus (e_{d,2j+1} \lll R_{d \bmod 8,j})$;
15. $\quad$ **end for**
16. $\quad$ **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
17. $\quad\quad v_{d+1,i} \leftarrow f_{d,\pi(i)}$; (Permute)
18. $\quad$ **end for**
19. **end for**
20. **for** $i \leftarrow 0$ **to** $N_w - 1$ **do**
21. $\quad c_i \leftarrow v_{N_r,i} \boxplus k_{N_r/4,i}$; (Key injection)
22. **end for**
23. **return** $C = (c_0, c_1, \ldots, c_{N_w-1})$;

B. The UBI Chaining Mode

Let $E(K, T, P)$ be a tweakable encryption function. The Unique Block Iteration (UBI) chaining mode allows one to build a compression function out of E. Each block M_i of the message is processed with a unique tweak value T_i encoding how many bytes have been processed so far, a type field (see [1] for details), and two bits specifying whether it is the first and/or last block. The UBI chaining mode is computed as:

$$H_0 \leftarrow G,$$
$$H_{i+1} \leftarrow M_i \oplus E(H_i, T_i, M_i),$$

where G is a starting value of N_w words.

C. Hardware Implementation

This short description of Threefish gives us the first hints on designing a dedicated coprocessor (Figure 2). Our architecture consists of a register file implemented by means of dual-ported memory, an ALU, and a control unit. The register file is organized into 64-bit words, and stores a plaintext block, an internal state ($e_{d,i}$, where $0 \leq i \leq N_w - 1$), an extended block cipher key, an extended tweak, the constant C_{240}, and all possible values of s involved in the key schedule. Thanks to this approach, the word permutation $\pi(i)$ and the word rotation of the key schedule are conveniently implemented by addressing the register file accordingly. Since the round constants repeat every eight rounds (Algorithm 2, line 14), we decided to unroll eight iterations of the main loop of Threefish (Algorithm 2, lines 4 to 19). The rotation constants $R_{d,i}$ are included in the microcode executed by the control unit. Note that our register file is designed for Threefish-1024 (*i.e.* $N_w = 16$ and $N_r = 80$). It is therefore straightforward to implement the two other variants of the algorithm on our architecture.

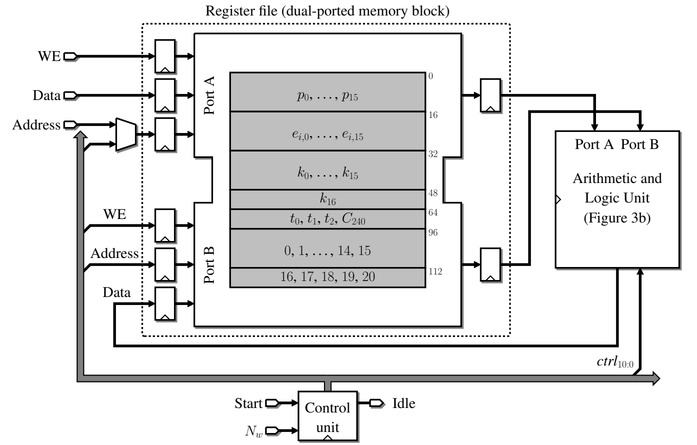
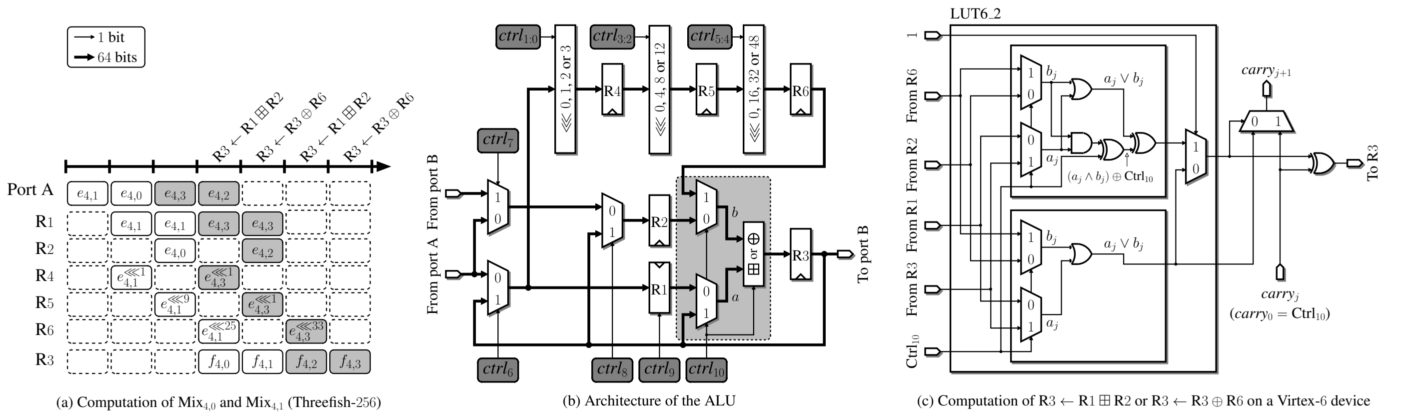


Figure 2. Architecture of the Threefish coprocessor.

The next step consists in defining the architecture of the ALU and the instruction set of our coprocessor. In the following, Ri denotes a 64-bit register. Figure 3a illustrates our scheduling of the two mixing functions $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ of the fifth round of Threefish-256:



(a) Computation of $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ (Threefish-256)

(b) Architecture of the ALU

(c) Computation of $\text{R3} \leftarrow \text{R1} \boxplus \text{R2}$ or $\text{R3} \leftarrow \text{R3} \oplus \text{R6}$ on a Virtex-6 device

Figure 3. Arithmetic and logic unit for Threefish encryption.

- The operand $e_{4,1}$ is loaded in register R1; at the same time, we start the computation of $e_{4,1} \lll R_{4,0}$; this operation requires three clock cycles and intermediate results are stored in R4, R5, and R6.
- Then, $e_{4,0}$ is loaded in register R2; the content of R1 is not modified (*i.e.* R1 must be controlled by an enable signal).
- We execute the instruction $\text{R3} \leftarrow \text{R1} \boxplus \text{R2}$.
- R3 and R6 contain $f_{4,0}$ and $e_{4,1} \lll R_{4,0}$, respectively. The instruction $\text{R3} \leftarrow \text{R3} \oplus \text{R6}$ allows us to compute $f_{4,1}$.

We schedule $\text{Mix}_{4,1}$ as soon as $e_{4,0}$ has been read, and manage to keep the pipeline continuously busy. In summary, our ALU must be able to carry out any rotation of a 64-bit word and to perform the following operation (Figure 3b):

$$\text{R3} \leftarrow \begin{cases} \text{R1} \boxplus \text{R2} & \text{when } \text{Ctrl}_{10} = 0, \\ \text{R3} \oplus \text{R6} & \text{otherwise,} \end{cases} \tag{1}$$

where Ctrl_{10} denotes a control bit. Our ALU is therefore similar to the one proposed by Beuchat *et al.* [4] for the SHA-3 finalist BLAKE, and we can take advantage of their strategy to share hardware resources between the adder and the array of XOR gates (we propose here a more general approach that does not require a low-level VHDL description relying on libraries provided by Xilinx). Let us define two 64-bit operands a and b such that:

$$(a, b) = \begin{cases} (\text{R1}, \text{R2}) & \text{when } \text{Ctrl}_{10} = 0, \\ (\text{R3}, \text{R6}) & \text{otherwise.} \end{cases}$$

It is well-known that $a \boxplus b = (a \vee b) \boxplus (a \wedge b)$ and $a \oplus b = (a \vee b) \boxminus (a \wedge b)$, where $\vee$, $\wedge$, and $\boxminus$ denote the bitwise OR, the bitwise AND, and the subtraction modulo 2^{64} of two operands, respectively [5]. Thus, Equation (1) can be rewritten as follows:

$$\text{R3} \leftarrow (a \vee b) \boxplus ((a \wedge b) \oplus \text{Ctrl}_{10}) \boxplus \text{Ctrl}_{10}. \tag{2}$$

Figure 3c describes the implementation of Equation (2) on a Virtex-6 device. Since there is a single control signal to choose the arithmetic operation and to select a and b, Equation (2) involves only five variables, and is advantageously implemented by 64 LUT6_2 primitives and dedicated carry logic.

In order to reduce the number of operands stored in the register file, we interleave the key schedule (Algorithm 1) and the encryption process (Algorithm 2). This approach allows us to generate the subkeys on-the-fly. It is however necessary to compute t_2 and k_{N_w} before the first key injection. The easiest way to compute t_2 would be to load t_0 and t_1 in registers R1 and R2, respectively, and to execute the instruction $\text{R3} \leftarrow \text{R1} \oplus \text{R2}$. Unfortunately, this solution requires one more control bit to select the inputs of the arithmetic operator, and it is not possible to implement the multiplexers and the adder on the same LUT6_2 primitive anymore. Since the critical path of our coprocessor is located in the 64-bit adder, an extra level of LUTs would decrease the clock frequency. However, we are able to compute t_2 using only the functionalities defined by Equation (1). Since $t_2 = (t_0 \boxplus 0) \oplus (t_1 \lll 0)$, it suffices to execute the following instructions:

$$\begin{array}{lll} \text{R4} \leftarrow t_1 \lll 0, & & \\ \text{R1} \leftarrow t_0, & \text{R2} \leftarrow 0, & \text{R5} \leftarrow \text{R4} \lll 0, \\ \text{R3} \leftarrow \text{R1} \boxplus \text{R2}, & \text{R6} \leftarrow \text{R5} \lll 0, & \\ \text{R3} \leftarrow \text{R3} \oplus \text{R6}. & & \end{array}$$

This approach assumes that we can read simultaneously two values from the register file. Thanks to the multiplexer controlled by Ctrl_7, we can load data from port A or port B into register R_2 (Figure 3b). A similar strategy allows us to compute k_{N_w}.

The implementation of the key injection is more straightforward. Note that the multiplexers controlled by Ctrl_6 and Ctrl_8 allow us to bypass the register file and to use the content of R3 as an input to the ALU. Let us consider for instance the first key injection of Threefish-256: $e_{0,2}$ is defined as $p_2 \boxplus k_{0,2} = p_2 \boxplus k_2 \boxplus t_1$ and is computed as follows:

$$\begin{array}{ll} \text{R1} \leftarrow k_2, & \text{R2} \leftarrow t_1, \\ \text{R3} \leftarrow \text{R1} \boxplus \text{R2} & \\ \text{R1} \leftarrow \text{R3}, & \text{R2} \leftarrow p_2, \\ \text{R3} \leftarrow \text{R1} \boxplus \text{R2}. & \end{array}$$

Figure 4 describes how we schedule the instructions of Threefish-256.

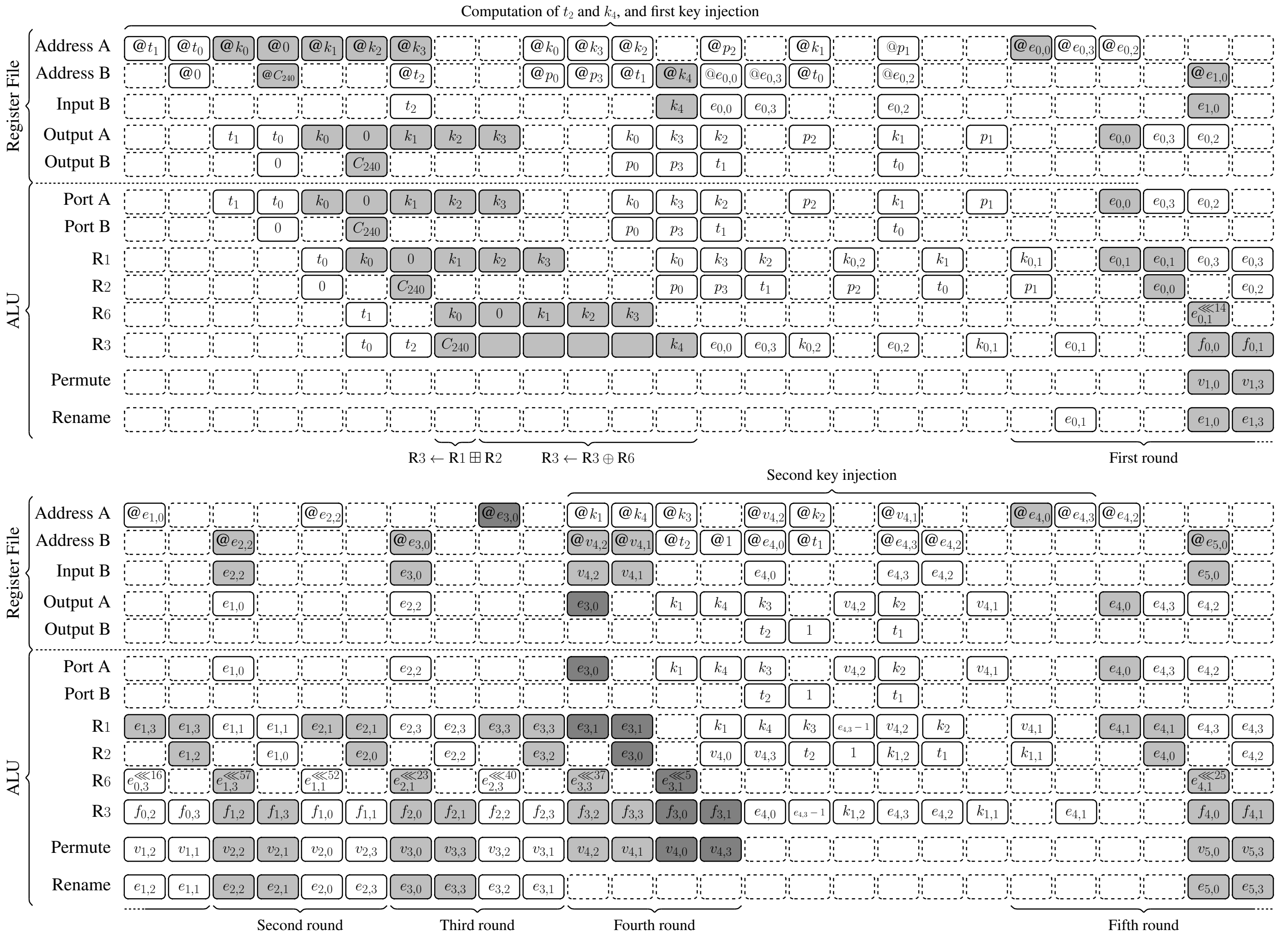


Figure 4. Scheduling of Threefish-256. $@d$ denotes the address of the 64-bit word d in the register file. "Rename" and "Permute" refer to lines 9 and 17 of Algorithm 2, respectively.

The UBI chaining mode can be combined with the final key injection of Threefish encryption. It suffices to modify line 21 of Algorithm 2 as follows: $e_{N_r,i} \leftarrow v_{N_r,i} \boxplus k_{N_r/4,i}$ and $c_i \leftarrow e_{N_r,i} \oplus p_i$. The only difference between this operation and the mixing function $\text{MIX}_{d,j}$ is that no permutation is applied to the second operand of the bitwise exclusive OR.

The control unit consists of an instruction memory, a small table to generate the address of $k_{(s+i) \bmod (N_w+1)}$ during the key schedule (Algorithm 1), and a simple finite-state machine. Since we unroll only eight rounds, we manage to keep the instruction memory compact. In the case of Threefish-512, we need for instance 145 instructions and 881 clock cycles to encrypt a plaintext block in UBI mode.

III. Skein Hashing

In this work, we focus on the simple Skein hash computation and refer the reader to [1] for a description of the other modes of operation. Three UBI invocations allow one to compute the digest of a message $M = (M_0, M_1, \ldots, M_{b-1})$ of up to $2^{99}-8$ bits, where each M_i is a block of N_w 64-bit word. We explain this process with an example: how to hash a message of 170 bytes $M = (M_0, M_1, M_2)$ with Skein-512-512 (Figure 5):

1) Configuration block. The 32-byte string C encodes the desired output length and several parameters defined in [1]. In the simple hashing mode, $G_0 = \text{UBI}(0, C, T_{\text{cfg}}2^{120})$ depends only on the output size, and can be precomputed (see [1, Appendix B]).
2) Message. Note that M_0 and M_1 contain 64 bytes of data each, and M_2 is the padded final block with 42 bytes of data. The tweak encodes whether M_i is the first or last block of M, and the number of bytes processed so far. In this example, $\text{UBI}(G_0, M, T_{\text{msg}}2^{120})$ requires three calls to Threefish-512.
3) Output transform. The call to $\text{UBI}(G_1, 0, T_{\text{out}}2^{120})$ achieves hashing-appropriate randomness. If a single output block is not sufficient, several output transforms can be run in parallel [1].

In order to process a b-block message M, we load the precomputed value of G_0 in the register file of our coprocessor. Then, $b+1$ calls to Threefish allow us to process the message

and to perform the output transform. In the case of Skein-512-512, the throughput is given by $T = \frac{512 \cdot b \cdot f}{(b+1) \cdot 881}$ bits/s, where f denotes the clock frequency of our architecture.

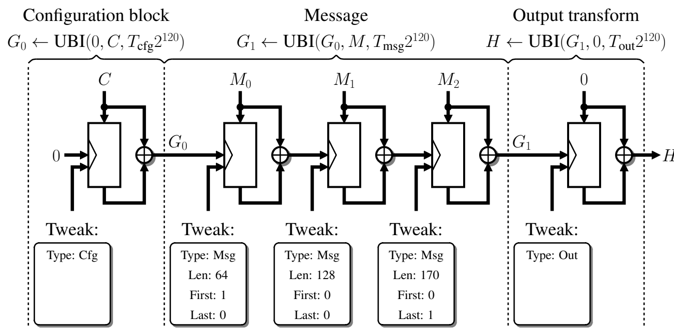


Figure 5. Processing a 3-block message using Skein-512-512 in the simple hashing mode.

IV. Results and Perspectives

We captured our architecture in the VHDL language and prototyped a fully autonomous implementation of Skein-512-512 on a Xilinx Virtex-6 FPGA. Table II summarizes our results and the figures published by other researchers focusing on compact coprocessors (we refer the reader to the SHA-3 Zoo [12] for an overview of high-speed designs). Note that we considered the least favorable case, where the message consists of a single block, to compute the throughput. If we increase the size of the message, the throughput of our coprocessor converges asymptotically to 160 Mbits/s. The other hardware architectures of Skein reported in Table II make a single call to Threefish-512 and do not perform the output transform. Let us assume that all SHA-3 finalists provide the levels of security expected by the NIST. Then, according to Table II, BLAKE, Keccak, and Skein seem to be the best candidates for compact implementations on FPGA. Beuchat *et al.* [4] designed a low-area ALU for BLAKE on Xilinx devices. However, the datapath depends on the level of security one wishes to achieve. In order to overcome this drawback, Yamazaki *et al.* [7] proposed a unified coprocessor for the BLAKE family. Their ALU is built around a 64-bit datapath, and can process a 512-bit block (BLAKE-512) or two 256-bit blocks in parallel (BLAKE-256). From our point of view, the main advantage of Skein over other SHA-3 finalists is that the same coprocessor allows one to encrypt or hash a message. We plan to improve our architecture in order to support Threefish decryption, Skein-MAC, and tree hashing with Skein.

References

[1] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family (version 1.3)," Oct. 2010.

[2] S. Kerckhof, F. Durvaux, N. Veyrat-Charvillon, F. Regazzoni, G. Meurice de Dormale, and F.-X. Standaert, "Compact FPGA implementations of the five SHA-3 finalists," in *Proceedings of the ECRYPT II Hash Workshop*, 2011.

[3] J. Zhai, C. Park, and G.-N. Wang, "Hash-based RFID security protocol using randomly key-changed identification procedure," in *Computational Science and Its Applications–ICCSA 2006*, ser. Lecture Notes in Computer Science, no. 3983. Springer, 2006, pp. 296–305.

[4] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "Compact implementations of BLAKE-32 and BLAKE-64 on FPGA," in *Proceedings of the 2010 International Conference on Field-Programmable Technology–FPT 2010*, J. Bian, Q. Zhou, and K. Zhao, Eds. IEEE Press, 2010, pp. 170–177.

[5] H. Warren, *Hacker's Delight*. Addison-Wesley, 2002.

[6] J.-P. Aumasson, L. Henzen, W. Meier, and R. Phan, "SHA-3 proposal BLAKE (version 1.4)," Jan. 2011.

[7] T. Yamazaki, J.-L. Beuchat, and E. Okamoto, "BLAKE-256, BLAKE-512のコンパクトな統合実装," *IEICE*暗号と情報セキュリティ実装技術小特集号, vol. J-95A, no. 5, 2012.

[8] B. Jungk, "Compact implementations of Grøstl, JH and Skein for FPGAs," in *Proceedings of the ECRYPT II Hash Workshop*, 2011.

[9] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "Keccak implementation overview (version 3.1)," Sep. 2011.

[10] İ. San and N. At, "Compact Keccak hardware architecture for data integrity and authentication on FPGAs," *Information Security Journal: A Global Perspective*, 2012.

[11] K. Latif, M. Tariq, A. Aziz, and A. Mahboob, "Efficient hardware implementation of secure hash algorithm (SHA-3) finalist - Skein," in *Proceedings of the International Conference on Computer, Communication, Control and Automation–3CA2011*, 2011.

[12] "The SHA-3 zoo," http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

Table II
COMPACT IMPLEMENTATIONS OF THE FIVE SHA-3 FINALISTS ON VIRTEX-5 AND VIRTEX-6 FPGAS.

	Algorithm	**FPGA**	**Area [slices]**	**36k memory blocks**	**Frequency [MHz]**	**Throughput [Mbits/s]**
Aumasson *et al.* [6]	BLAKE-256	xc5vlx110	390	–	91	412
Beuchat *et al.* [4][†]	BLAKE-256	xc6vlx75t-2	52	2	456	194
Aumasson *et al.* [6]	BLAKE-512	xc5vlx110	939	–	59	468
Beuchat *et al.* [4][†]	BLAKE-512	xc6vlx75t-2	81	3	374	280
Kerckhof *et al.* [2]	BLAKE-512	xc6vlx75t-1	192	–	240	183
Yamazaki *et al.* [7]	BLAKE (unified coprocessor)	xc5vlx50-2	138	3	342	2×150 (BLAKE-256) 264 (BLAKE-512)
Jungk [8]	Grøstl-256	xcv5	470	–	354	1132
Kerckhof *et al.* [2]	Grøstl-512	xc6vlx75t-1	260	–	280	640
Jungk [8]	JH-256	xcv5	205	–	341	27
Kerckhof *et al.* [2]	JH-512	xc6vlx75t-1	240	–	288	214
Bertoni *et al.* [9]	Keccak$[r = 1024, c = 576]$	xc5vlx50-3	448	–	265	52
Kerckhof *et al.* [2]	Keccak$[r = 1024, c = 576]$	xc6vlx75t-1	144	–	250	68
San & At [10]	Keccak$[r = 1024, c = 576]$	xc5vlx50-2	151	3	520	501
This Work	Skein-512-512	xc6vlx75t-1	132	2	276	80
Jungk [8][‡]	Skein-512-256	xcv5	555	–	271	237
Kerckhof *et al.* [2][‡]	Skein-512-512	xc6vlx75t-1	240	–	160	179
Latif *et al.* [11][‡]	Skein-256-256	xc5vlx110-3	821	Not specified	119	1610

[†]Modified to implement the tweaked version submitted for the final round of the SHA-3 competition.
[‡]Single call to Threefish-512.