Compact Hardware Implementations of ChaCha, BLAKE, Threefish, and Skein on FPGA

Nuray At*, Jean-Luc Beuchat†, Eiji Okamoto‡, İsmail San*, and Teppei Yamazaki‡
*Department of Electrical and Electronics Engineering, Anadolu University, Eskişehir, Turkey
Email: {nat, isan}@anadolu.edu.tr
†ELCA Informatique SA, Av. de la Harpe 22–24, Case postale 519, 1001 Lausanne, Switzerland
Email: jeanluc.beuchat@gmail.com
‡Graduate School of Systems and Information Engineering,
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan
Email: okamoto@risk.tsukuba.ac.jp, yamazaki@cipher.risk.tsukuba.ac.jp

***Abstract*—The cryptographic hash functions BLAKE and Skein are built from the ChaCha stream cipher and the tweakable Threefish block cipher, respectively. Interestingly enough, they are based on the same arithmetic operations, and the same design philosophy allows one to design lightweight coprocessors for hashing and encryption. The key element of our approach is to take advantage of the parallelism of the algorithms to deeply pipeline our Arithmetic an Logic Units, and to avoid data dependencies by interleaving independent tasks. We show for instance that a fully autonomous implementation of BLAKE and ChaCha on a Xilinx Virtex-6 device occupies 144 slices and three memory blocks, and achieves competitive throughputs. In order to offer the same features, a coprocessor implementing Skein and Threefish requires a substantial higher slice count.**

I. Introduction

The cryptographic hash functions BLAKE [1] and Skein [2] are built from the ChaCha stream cipher [3] and the tweakable Threefish block cipher [2], respectively. It is therefore tempting to design compact unified hardware architectures able to hash and encrypt a message. Such processors are for instance valuable for constrained environments, where some security protocols mainly rely on cryptographic hash functions [4]. Furthermore, 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" [5]. In order to have a deeper understanding of the computational efficiency of ChaCha, BLAKE, Threefish, and Skein, we extend here our work presented in [6], [7] and propose novel lightweight coprocessors. The key element of our approach is to take advantage of the parallelism of the algorithms to deeply pipeline our Arithmetic an Logic Units (ALUs), and to avoid data dependencies by interleaving independent tasks.

Throughout this article, all operands are w-bit unsigned integers and the following notation is adopted:

- $\boxplus$: addition modulo 2^w;
- $\boxminus$: subtraction modulo 2^w;
- $\wedge$: bitwise AND;
- $\vee$: bitwise OR;
- $\oplus$: bitwise exclusive OR;
- $\ggg k$: rotation by k bits to the right;
- $\lll k$: rotation by k bits to the left.

The rest of the article is organized as follows: after a brief overview of Threefish (Section II), Skein (Section III), ChaCha (Section IV), and BLAKE (Section V), we describe our design philosophy and compact hardware implementations (Section VI). We discuss our implementation results on several Xilinx Field-Programmable Gate Arrays (FPGAs) in Section VII and conclude in Section VIII.

II. The Threefish Block Cipher

The design philosophy of Threefish is that "a larger number of simple rounds is more secure than fewer complex rounds" [2]. The key schedule can be computed in a few clock cycles, which is an important consideration in order to build a compression function from a block cipher.

Threefish operates entirely on unsigned 64-bit integers and involves only three operations: rotation of k bits to the left, bitwise exclusive OR, and addition modulo 2^{64}. 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 size of a plaintext block is given by $N_b = 8 \cdot N_w$ bytes.

Table I
Number of rounds of Threefish for different key sizes.

Key size [bits]	# 64-bit words N_w	# rounds N_r	Block size N_b [bytes]
256	4	72	32
512	8	72	64
1024	16	80	128

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 II
PERMUTATIONS USED BY THE SKEIN FUNCTIONS (REPRINTED FROM [2]).

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	$N_w = 4$	0	3	2	1												
$\pi(i)$	$N_w = 8$	2	1	4	7	6	5	0	3								
	$N_w = 16$	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

Algorithm 1 Key schedule of Threefish.

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 $k_{s,0}$, $k_{s,1}$, $\ldots$, k_{s,N_w-1}, where $0 \le s \le N_r/4$.

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** $k_{s,0}$, $k_{s,1}$, $\ldots$, k_{s,N_w-1}, where $0 \le s \le N_r/4$;

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 $\text{Mix}_{d,j}$ (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 [2, Table 4]), and a bitwise exclusive OR. A word permutation $\pi(i)$ (defined in Table II) 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 2 describes a decryption round of Threefish-256. It consists of the inverse word permutation followed by the inverse MIX functions. Note that subkeys are injected in reverse order.

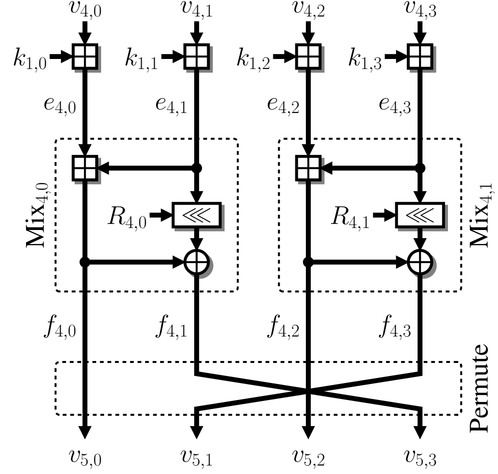

Figure 1. One of the 72 encryption rounds of Threefish-256.

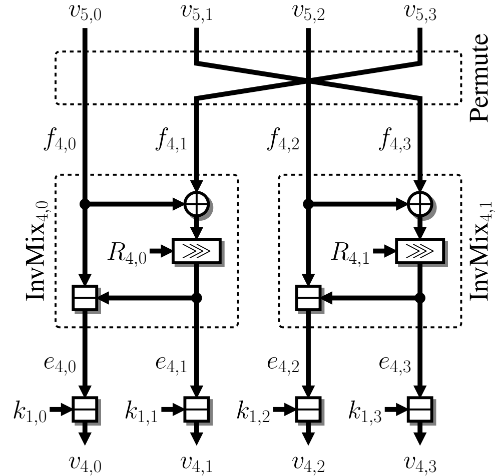

Figure 2. One of the 72 decryption rounds of Threefish-256.

III. THE SKEIN FAMILY OF HASH FUNCTIONS

The Unique Block Iteration (UBI) chaining mode allows one to build a compression function out of a tweakable encryption function. Let M be a message of arbitrary length up to $2^{99} - 8$ bits. If the number of bits in M is not a multiple of 8, we append a bit 1 followed by a (possibly empty) string of 0's. This step guarantees that M contains N_M bytes. Then, we pad M with p zero bytes so that $N_M + p$ is a multiple of the block size N_b. We can now split M into N_b-byte blocks M_0, $\ldots$, M_{k-1}, where $k = (N_M + p)/N_b$. Each block M_i is processed with a unique tweak value T_i encoding how many bytes have been processed so far, a type field (see [2] 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_b bytes.

In this work, we consider the normal hashing mode and refer the reader to [2] for a description of Skein-MAC and tree hashing with Skein. Skein is built on three invocations of UBI (Figure 3):

- Define a 32-byte configuration string C that contains the length of the digest size (in bits), a schema identifier,

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 $k_{s,0}$, $k_{s,1}$, …, k_{s,N_w-1}, where $0 \le s \le N_r/4$; $4N_w$ rotation constants $R_{i,j}$, where $0 \le i \le 7$ and $0 \le j \le N_w/2$.

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** $C = (c_0, c_1, \ldots, c_{N_w-1})$;

and a version number [2, Table 7]. Compute the N_b-byte block G_0:

$$G_0 \leftarrow \text{UBI}(0, C, T_{\text{cfg}}2^{120}).$$

Note that G_0 only depends on the digest size and can easily be precomputed.

- The message is then processed as follows:

$$G_1 \leftarrow \text{UBI}(G_0, M, T_{\text{msg}}2^{120}).$$

- A third call to UBI is required to achieve hashing-appropriate randomness:

$$H \leftarrow \text{UBI}(G_1, 0, T_{\text{out}}2^{120}).$$

This transform allows one to produce arbitrary digest sizes (up to 2^{64} bits). If a single output block H is not enough, one can use Threefish in counter mode to produce the digest.

IV. The ChaCha Stream Cipher

The ChaCha family of stream ciphers was designed by Bernstein [3] to improve the diffusion per round of Salsa20 [8], while preserving the encryption rate. ChaCha operates on 32-bit words, and expands a 256-bit key $(k_0, \ldots, k_7)$ and a 64-bit nonce $(\text{IV}_0, \text{IV}_1)$ into a 2^{70}-byte stream. A b-byte message is

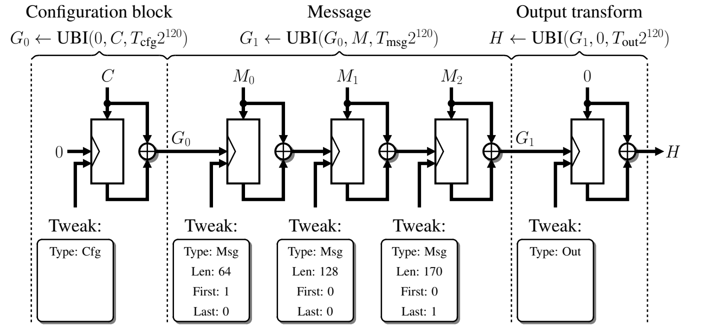

Figure 3. Skein in normal hashing mode.

then encrypted (or decrypted) by XORing it with the first b bytes of the stream.

ChaCha generates the stream by blocks of 64 bytes. In order to process the ith block, ChaCha acts on a 4×4 matrix M of 32-bit integers defined as follows:

$$\begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & \text{IV}_0 & \text{IV}_1 \end{pmatrix},$$

where

- c_0 = `61707865`, c_1 = `3320646E`, c_2 = `79622D32`, and c_3 = `6B206574` are predefined constants;
- $t = (t_0, t_1)$ is a 64-bit counter encoding the index i (*i.e.* $i = 2^{32}t_1 + t_0$).

ChaCha transforms the matrix M through a series of N_r rounds (Algorithm 3). The algorithm is based on a nonlinear operation called quarter-round function and described by Algorithm 4. Matrix M is copied into matrix V. Then, the even- and odd-numbered rounds of ChaCha apply the quarter-round function to each row and northwest-to-southeast diagonal of V, respectively. Eventually, a new block of the stream is generated by adding V to the original matrix M (Algorithm 3, line 15), and the block counter is incremented (Algorithm 3, lines 17 to 20).

Bernstein proposed 8-, 12-, and 20-round variants of ChaCha. Aumasson *et al.* introduced a novel method for differential cryptanalysis of ChaCha and broke the 7-round variant [9]. Ishiguro *et al.* [10], [11] improved the attack and concluded that Salsa20 and Chacha "are not presently under threat".

V. The BLAKE Family of Hash Functions

The BLAKE family combines three previously studied components, chosen by Aumasson *et al.* for their complementarity [1]: the iteration mode HAIFA, the internal structure of the hash function LAKE, and a modified version of Bernstein's stream cipher ChaCha as compression function. BLAKE is a family of four hash functions, namely BLAKE-224, BLAKE-256, BLAKE-384, and BLAKE-512 (Table III). The main differences lie in the length of words w, the number of rounds N_r, and in some constants involved in the algorithm. In the

Table III
PROPERTIES OF THE BLAKE HASH FUNCTIONS.

Algorithm	Word size w [bits]	Message [bits]	Block [bits]	Digest [bits]	Salt [bits]	# rounds N_r	Rotation distances δ_0	δ_1	δ_2	δ_3
BLAKE-224	32	$< 2^{64}$	512	224	128	14	16	12	8	7
BLAKE-256	32	$< 2^{64}$	512	256	128	14	16	12	8	7
BLAKE-384	64	$< 2^{128}$	1024	384	256	16	32	25	16	11
BLAKE-512	64	$< 2^{128}$	1024	512	256	16	32	25	16	11

Algorithm 3 Computation of a 64-byte block of the stream of ChaCha.

Input: A key, a nonce, and a block counter stored in a matrix M.
Output: A 64-byte block of the stream.

1. **for** $i \leftarrow 0$ **to** 15 **do**
2. $\quad v[i] \leftarrow m[i]$;
3. **end for**
4. **for** $i \leftarrow 0$ **to** $N_r/2-1$ **do**
5. $\quad$ QUARTERROUND(v_0, v_4, v_8, v_{12});
6. $\quad$ QUARTERROUND(v_1, v_5, v_9, v_{13});
7. $\quad$ QUARTERROUND$(v_2, v_6, v_{10}, v_{14})$;
8. $\quad$ QUARTERROUND$(v_3, v_7, v_{11}, v_{15})$;
9. $\quad$ QUARTERROUND$(v_0, v_5, v_{10}, v_{15})$;
10. $\quad$ QUARTERROUND$(v_1, v_6, v_{11}, v_{12})$;
11. $\quad$ QUARTERROUND(v_2, v_7, v_8, v_{13});
12. $\quad$ QUARTERROUND(v_3, v_4, v_9, v_{14});
13. **end for**
14. **for** $i \leftarrow 0$ **to** 15 **do**
15. $\quad v[i] \leftarrow v[i] \boxplus m[i]$;
16. **end for**
17. $m_{12} \leftarrow m_{12} \boxplus 1$;
18. **if** $m_{12} = 0$ **then**
19. $\quad m_{13} \leftarrow m_{13} \boxplus 1$;
20. **end if**
21. **Return** M and V;

Algorithm 4 The ChaCha quarter-round function.

Input: Four 32-bit integers a, b, c, and d.
Output: QUARTERROUND(a, b, c, d).

1. $a \leftarrow a \boxplus b$;
2. $d \leftarrow (d \oplus a) \lll 16$;
3. $c \leftarrow c \boxplus d$;
4. $b \leftarrow (b \oplus c) \lll 12$;
5. $a \leftarrow a \boxplus b$;
6. $d \leftarrow (d \oplus a) \lll 8$;
7. $c \leftarrow c \boxplus d$;
8. $b \leftarrow (b \oplus c) \lll 7$;
9. **Return** a, b, c, and d;

following, we denote by BLAKE-n the algorithm with an n-bit digest.

BLAKE-n involves only two arithmetic operations: the addition modulo 2^w of two w-bit unsigned integers and the bitwise exclusive OR of two w-bit words. The latter is sometimes followed by a rotation of δ_j bits to the right. The four possible rotation distances depend on the digest size and are defined in Table III. The compression function of BLAKE-n produces a new chain value $h' = h'_0, \ldots, h'_7$ from a message block $m = m_0, \ldots, m_{15}$, a chain value $h = h_0, \ldots, h_7$, a salt $s = s_0, \ldots, s_3$, a counter $t = t_0, t_1$, and 16 constants c_i defined in [1, p. 8]. This process consists of three steps. First, a 16-word internal state $v = v_0, \ldots, v_{15}$ is initialized as follows:

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

Then, a series of N_r rounds is performed. Each of them consists of a transformation of the internal state v based on the G_i function described by Algorithm 5, where σ_r denotes a permutation of $\{0, \ldots, 15\}$ parametrized by the round index r (see Table IV). A column step updates the four columns of matrix v as follows: $G_0(v_0, v_4, v_8, v_{12})$, $G_1(v_1, v_5, v_9, v_{13})$, $G_2(v_2, v_6, v_{10}, v_{14})$, and $G_3(v_3, v_7, v_{11}, v_{15})$. Note that each call to G_i updates a distinct column of matrix v. Since we focus on compact implementations of BLAKE in this work, we interleave the computation of G_0, G_1, G_2, and G_3. This approach allows us to design an ALU with four pipeline stages and to achieve high clock frequencies. Then, a diagonal step updates the four diagonals of v: $G_4(v_0, v_5, v_{10}, v_{15})$, $G_5(v_1, v_6, v_{11}, v_{12})$, $G_6(v_2, v_7, v_8, v_{13})$, and $G_7(v_3, v_4, v_9, v_{14})$. Here again, each call to G_i modifies a distinct diagonal of the matrix, allowing us to interleave the computation of G_4, G_5, G_6, and G_7.

At the end of the last round, a new chain value $h' = h'_0, \ldots, h'_7$ is computed from the internal state v and the previous chain value h (finalization step):

$$\begin{aligned} h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8, & h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12}, \\ h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9, & h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13}, \\ h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10}, & h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14}, \\ h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11}, & h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}. \end{aligned}$$

In order to guarantee that the length ℓ of a message is a multiple of the block size b, Aumasson *et al.* define the following padding scheme [1]:

Table IV
PERMUTATIONS OF $\{0,\ldots,15\}$ USED BY THE BLAKE FUNCTIONS (REPRINTED FROM [1]).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_1(i)$	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
$\sigma_2(i)$	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
$\sigma_3(i)$	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
$\sigma_4(i)$	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
$\sigma_5(i)$	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
$\sigma_6(i)$	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
$\sigma_7(i)$	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
$\sigma_8(i)$	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
$\sigma_9(i)$	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Algorithm 5 The G_i function.

Input: A function index i and four w-bit integers a, b, c, and d.
Output: $G_i(a,b,c,d)$.
1. $a \leftarrow a \boxplus b$;
2. $a \leftarrow a \boxplus (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$;
3. $d \leftarrow (d \oplus a) \ggg \delta_0$;
4. $c \leftarrow c \boxplus d$;
5. $b \leftarrow (b \oplus c) \ggg \delta_1$;
6. $a \leftarrow a \boxplus b$;
7. $a \leftarrow a \boxplus (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$;
8. $d \leftarrow (d \oplus a) \ggg \delta_2$;
9. $c \leftarrow c \boxplus d$;
10. $b \leftarrow (b \oplus c) \ggg \delta_3$;

- append a bit 1 followed by a sufficient number of 0 bits such that the length is congruent to $b - 2w - 1$ modulo b;
- a padding bit followed by the $2w$-bit unsigned big-endian representation of ℓ is then added; in the case of BLAKE-256 and BLAKE-512, the padding bit is equal to 1; otherwise, it is set to 0.

The hash can now be computed iteratively (Algorithm 6): the padded message is divided into N 16-word blocks $m^{(0)},\ldots,m^{(N-1)}$ and the chain value $h^{(0)}$ is set to the same initial value as SHA-n. The counter $t^{(i)}$ denotes the number of message bits in $m^{(0)},\ldots,m^{(i)}$ (*i.e.* excluding padding bits). Note that, if the last block contains only padding bits, then $t^{(N-1)}$ is set to zero. The message digest consists of the n least significant bits of the output $h^{(N)}$.

Algorithm 6 Iterated hash.

Input: A padded message split into N 16-word blocks and a salt s.
Output: A n-bit digest.
1. $\left(h_0^{(0)},\ldots,h_7^{(0)}\right) \leftarrow (\mathrm{IV}_0,\ldots,\mathrm{IV}_7)$;
2. **for** $i \leftarrow 0$ **to** $N-1$ **do**
3. $\quad h^{(i+1)} \leftarrow$ **compress** $\left(h^{(i)}, m^{(i)}, s, t^{(i)}\right)$;
4. **end for**
5. **return** $h^{(N)}$;

VI. HARDWARE IMPLEMENTATION

All of our architectures consist of a register file organized into w-bit words and implemented by means of dual-ported memory, an ALU, and a control unit (Figure 4). The user loads messages, plaintext blocks or ciphertext blocks into port A. A few control bits allows her to select the algorithm and the desired level of security. When the coprocessors are hashing or encrypting a message, the intermediates results are always written to port B. In the following, we assume that our coprocessors are provided with padded messages. A hardware wrapper interface for BLAKE, Skein, and several other hash functions comprising communication and padding is described in [12].

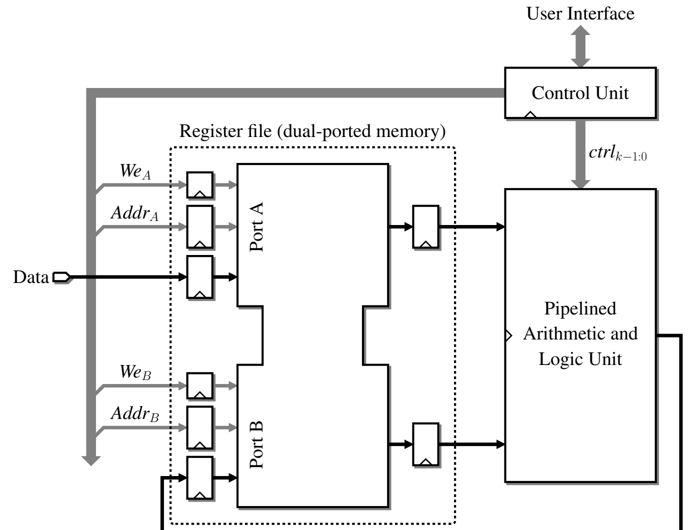

Figure 4. General architecture of our coprocessors.

We follow here the design strategy outlined in [6], [7], [13]–[15]. The first step consists in defining the minimal instruction set to implement a block cipher and a hash function. Then, an in-depth study of the scheduling allows us to build the ALU and organize the data in the register file. During this step, we

- try to minimize the number of control bits to keep the instruction memory as compact as possible;
- take advantage of FPGA specifics to optimize the slice count;

- identify the available parallelism and pipeline the ALU accordingly.

Eventually, we design the control unit. The instruction memory is automatically generated by a C program. In order to keep the instruction ROM as compact as possible, our C program is able to compress the code, and to generate the VHDL description of the decompression unit.

A. Arithmetic and Logic Units for Threefish and Skein

Our first ALU implements Threefish encryption and Skein. In the following, Ri denotes a 64-bit register. Figure 5 illustrates our scheduling of the two mixing functions $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ of the fifth round of Threefish-256:

- 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 $\text{R3} \leftarrow \text{R1} \boxplus \text{R2}$ and obtain $f_{4,0}$.
- 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 $f_{4,1}$.

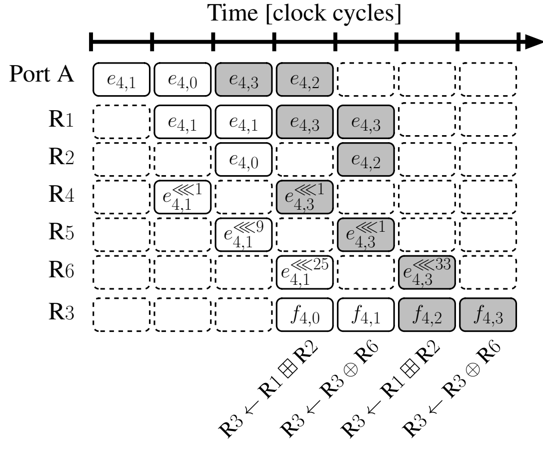

Figure 5. Computation of $\text{Mix}_{4,0}$ and $\text{Mix}_{4,1}$ (Threefish-256).

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 6):

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

where ctrl_{10} denotes a control bit. 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 $a \oplus b = (a \vee b) \boxminus (a \wedge b)$ [16]. 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 7 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.

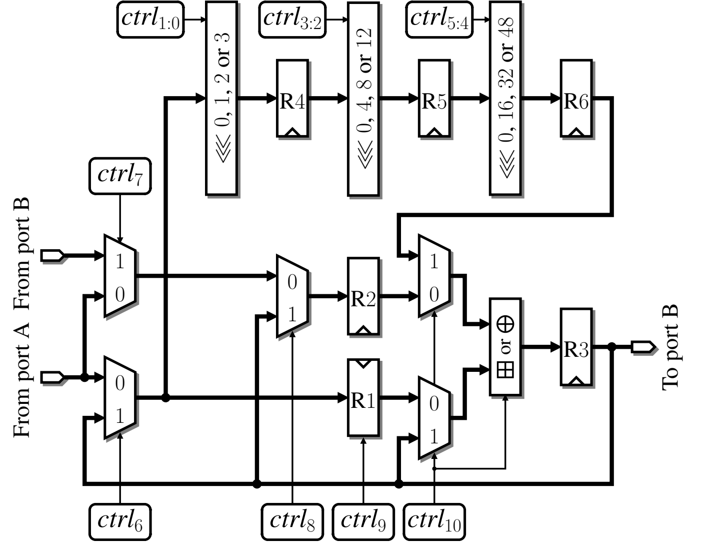

Figure 6. Arithmetic and logic unit for Threefish encryption.

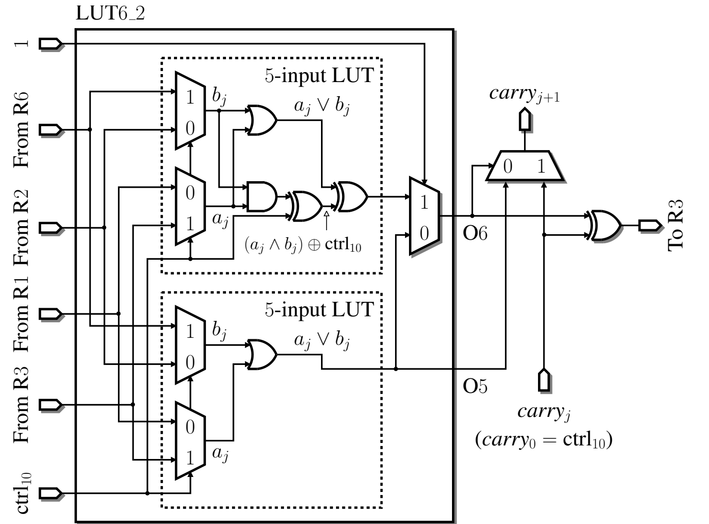

Figure 7. Computation of $\text{R3} \leftarrow \text{R1} \boxplus \text{R2}$ or $\text{R3} \leftarrow \text{R3} \oplus \text{R6}$ on a Virtex-6 device.

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 R3 ← R1⊕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:

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

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 6). 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:

$$\begin{aligned} &\mathrm{R1} \leftarrow k_2, \quad \mathrm{R2} \leftarrow t_1, \\ &\mathrm{R3} \leftarrow \mathrm{R1} \boxplus \mathrm{R2} \\ &\mathrm{R1} \leftarrow \mathrm{R3}, \quad \mathrm{R2} \leftarrow p_2, \\ &\mathrm{R3} \leftarrow \mathrm{R1} \boxplus \mathrm{R2}. \end{aligned}$$

Figures 8 and 9 describe how we schedule the instructions of Threefish-256.

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:

$$\begin{aligned} e_{N_r,i} &\leftarrow v_{N_r,i} \boxplus k_{N_r/4,i}; \\ c_i &\leftarrow e_{N_r,i} \oplus p_i. \end{aligned}$$

The only difference between this operation and the mixing function $\mathrm{MIX}_{d,j}$ is that no permutation is applied to the second operand of the bitwise exclusive OR.

The inverse of the MIX function being purely sequential, Threefish decryption has less parallelism than encryption. We suggest to modify our ALU as follows to fully support both encryption and decryption (Figure 10):

- The inverse of the Mix function and the inverse of the key injection require a subtraction modulo 2^{64}. Our modified ALU is therefore able to perform a new operation: R3 ← R1⊟R2. Because of the additional control bit required to select the operation, it is not possible to implement our arithmetic operator by means of 64 LUT6_2 anymore. Thus, the slice count and the critical path are expected to increase.
- The output of the inverse Mix function is provided either by the arithmetic operator (e.g. $e_{4,0}$ on Figure 2) or the rotation unit (e.g. $e_{4,1}$ on Figure 2). The multiplexer controlled by ctrl_{17} allows us to select the word we store in the register file.
- Since the inverse of the Mix function is sequential, we have to perform the rotation in a single clock cycle. We suggest to take advantage of the SRL16E primitive available on Xilinx devices to implement a FIFO whose depth is dynamically adjusted according to the algorithm selected by the user: one and three stages for decryption and encryption, respectively.

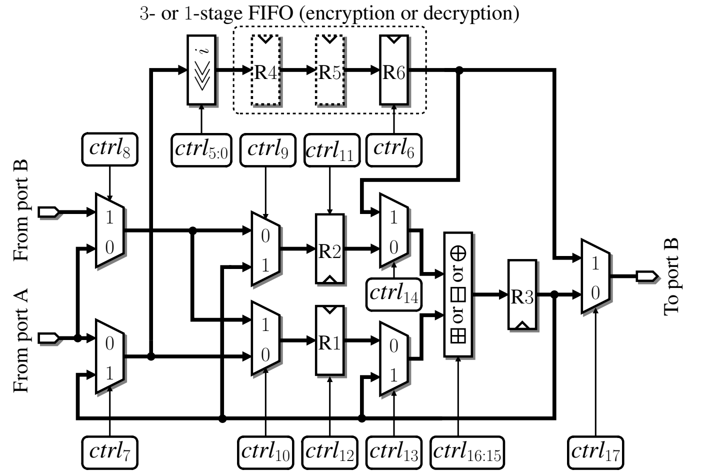

Figure 10. Arithmetic and logic unit for Threefish encryption and decryption.

B. Arithmetic and Logic Units for BLAKE and ChaCha

Let us consider the G_i function of BLAKE-n to define the instruction set of our coprocessors. Since we focus on compact coprocessors for the BLAKE family in this article, we perform a single step of Algorithm 5 at each clock cycle. We will show later that the input operand b is already stored in an internal register of our ALU when we start the computation of $G_i(a, b, c, d)$. Therefore, each operation involves the result of the previous one, and our ALU will include a feedback mechanism to bypass the register file of the coprocessor.

Assume that the w-bit word computed by the ALU is stored in register R5, and denote by RF_A and RF_B the operands provided by the register file. From the data flow diagram of Algorithm 5, we easily identify three operations (Figure 11):

1) Save the content of R5 in the register file and compute $\mathrm{R5} \leftarrow \mathrm{R5} \boxplus \mathrm{RF}_A$.
2) Compute $\mathrm{R5} \leftarrow \mathrm{R5} \boxplus (\mathrm{RF}_A \oplus \mathrm{RF}_B)$.
3) Save the content of R5 in the register file and compute $\mathrm{R5} \leftarrow (\mathrm{R5} \oplus \mathrm{RF}_A) \ggg \delta_j$.

Recall now that the four calls to G_i in a column step or a diagonal step can be computed in parallel. In order to keep the critical path as short as possible, we suggest to design an ALU

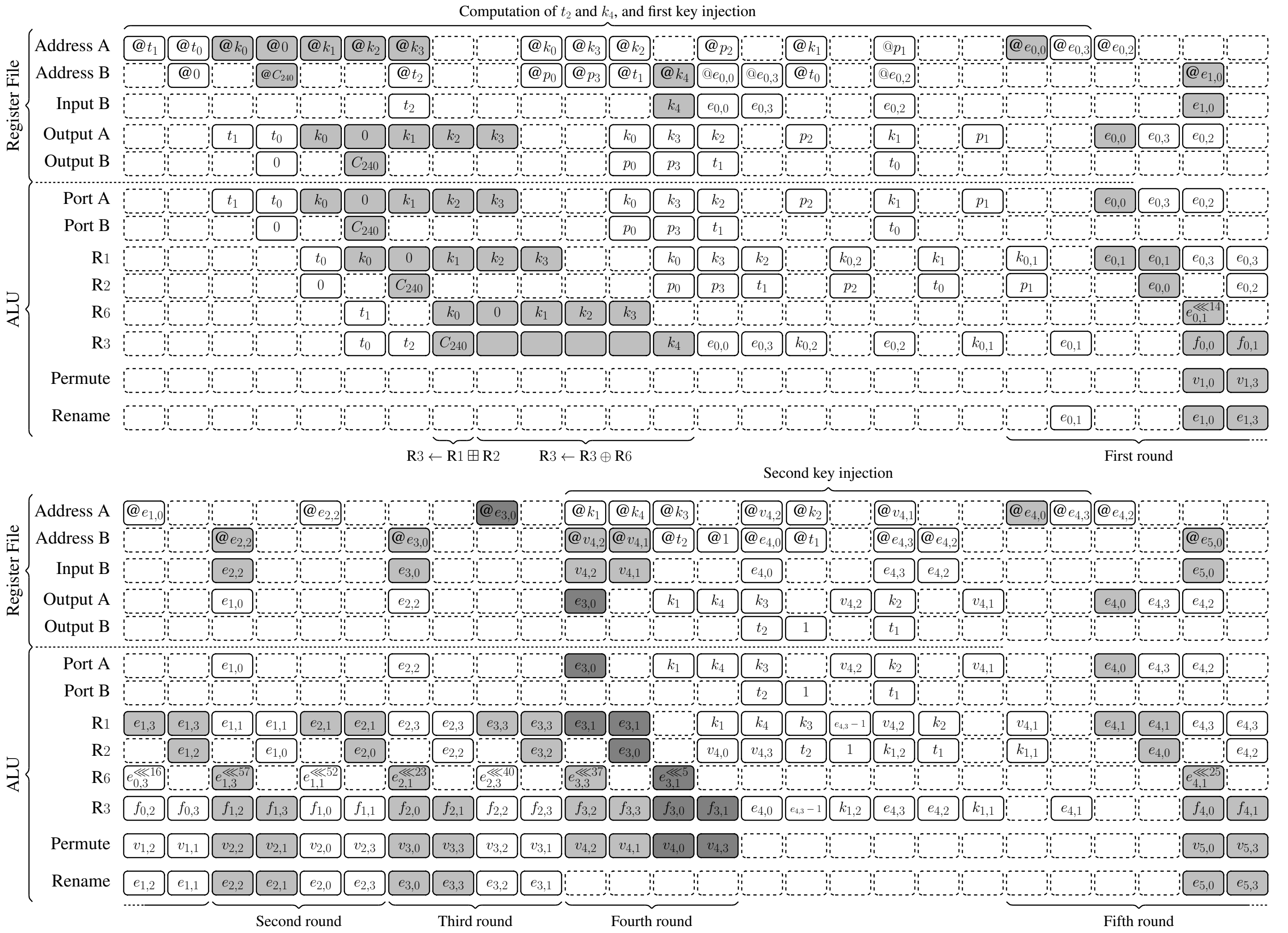

Figure 8. Scheduling of Threefish-256 encryption. $@d$ denotes the address of the 64-bit word d in the register file. "Rename" and "Permute" refer to lines 9 and 17 of Algorithm 2, respectively.

with four pipeline stages and to interleave the computation of four G_i functions (Figure 12). The heart of the ALU is the arithmetic operator performing the addition or the bitwise XOR of two w-bit words described in Section VI-A. Our operator computes:

$$\begin{aligned} \mathrm{R3} \leftarrow & \begin{cases} \mathrm{R1} \boxplus \mathrm{R2} & \text{when } \mathrm{ctrl}_2 = 0, \\ \mathrm{R1} \oplus \mathrm{R2} & \text{otherwise} \end{cases} \\ & = (\mathrm{R1} \vee \mathrm{R2}) \boxplus ((\mathrm{R1} \wedge \mathrm{R2}) \oplus \mathrm{ctrl}_2) \boxplus \mathrm{ctrl}_2, \end{aligned}$$

where

- R1 stores the data provided by the register file. Since a flip-flop is always associated with a LUT, we can perform some simple pre-processing without increasing the number of slices of the ALU: a control bit ctrl_0 selects either RF_A or $\mathrm{RF}_A \oplus \mathrm{RF}_B$. This allows us to compute $m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)}$ and $m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)}$ for free (Algorithm 5, lines 2 and 7).
- R2 almost always stores the result of a previous operation. However, we have to disable the feedback mechanism during the initialization step: the computation of $v_8 \leftarrow s_0 \oplus c_0$ involves for instance only two words stored in the register file. An array of AND gates controlled by ctrl_1 allows us to force the second operand to zero in such cases.

If needed, the content of register R3 is then rotated to the left in two steps. Our implementation is based on the following observation:

$$\mathrm{R3} \ggg \delta_i = (\mathrm{R3} \ggg (\delta_i - \delta_3)) \ggg \delta_3,$$

where $0 \leq i \leq 3$. At first glance, this design choice may look awkward. However, it will allow us to easily build a unified processor for the BLAKE family. The key point is that the content of R3 is copied into R5 when the three control bits $\mathrm{ctrl}_{5:3}$ are equal to 0 (Figure 12).

Note that the pipeline has three possible configurations, denoted by ①, ②, and ③ in Figure 12:

① In order to minimize the area of our ALU, we can insert a w-bit register after the first stage of the rotation. Since the latter involves w LUTs, there is no hardware overhead on a Virtex-6 device.

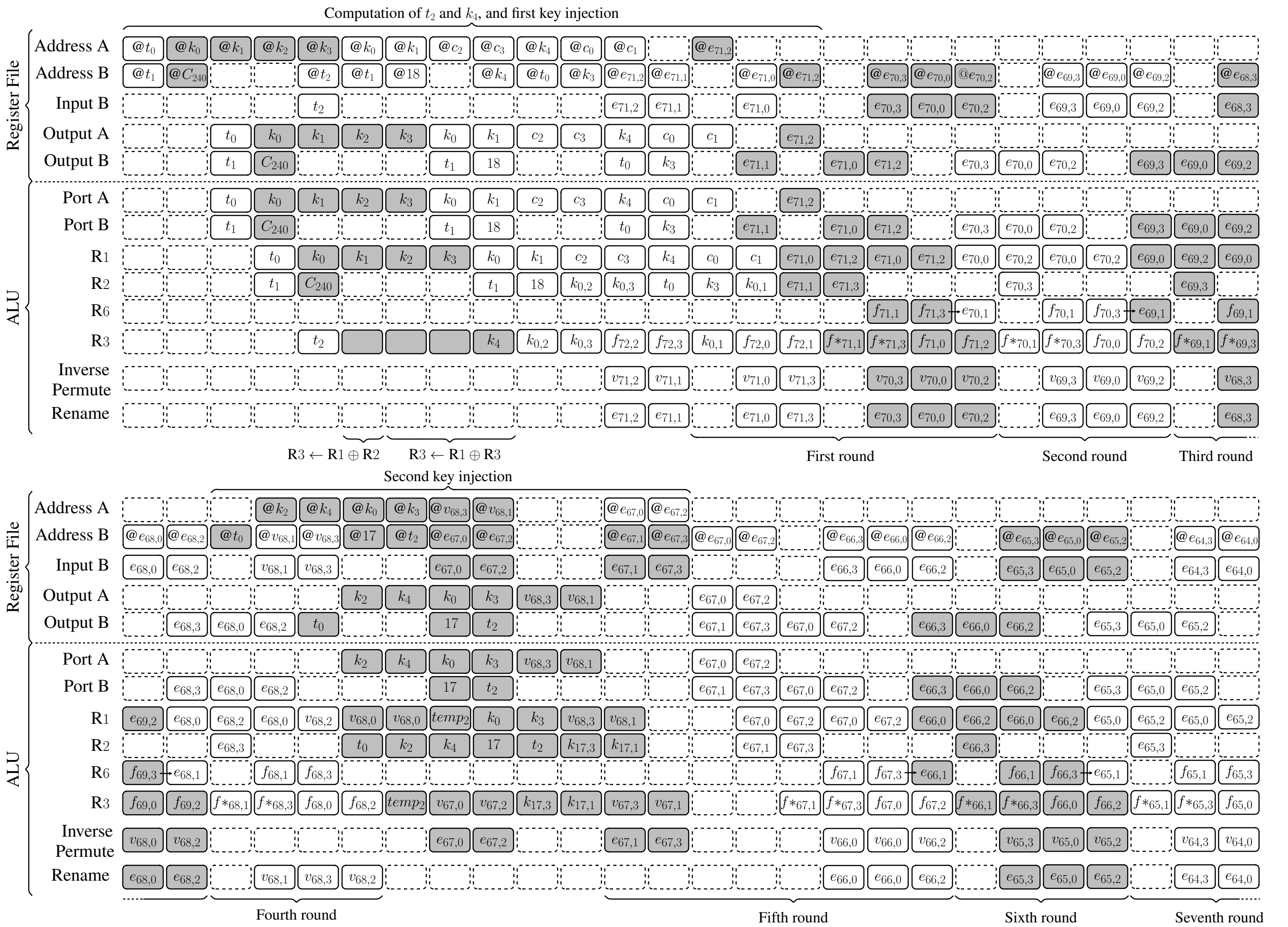

Figure 9. Scheduling of Threefish-256 decryption.

② The addition modulo 2^w can be computed in two clock cycles. Let $a = a_{\text{low}} + 2^{\frac{w}{2}} a_{\text{high}}$ and $b = b_{\text{low}} + 2^{\frac{w}{2}} b_{\text{high}}$. We store a_{high} and b_{high} in two $\frac{w}{2}$-bit registers, and compute a sum word s_{low} and a carry bit such that $2^{\frac{w}{2}} c + s_{\text{low}} = a_{\text{low}} + b_{\text{low}}$. A flip-flop and a $\frac{w}{2}$-bit register store c and s_{low}, respectively. The most significant bits of the sum are then given by $s_{\text{high}} = a_{\text{high}} + b_{\text{high}} + c$. This approach allows us to reduce the worst-case carry path at the price of three $\frac{w}{2}$-bit registers and a flip-flop.

③ Routing a signal from a memory block to a slice is sometimes expensive in terms of wire delay. If the critical path is located between the register file and register R1, this pipeline configuration will help boosting the clock frequency. The output data path of a Virtex-6 memory block has an optional internal pipeline register. Therefore, the only hardware overhead is the w-bit register between R5 and the array of AND gates controlled by ctrl_1.

In order to avoid pipeline bubbles between column and diagonal steps, it suffices to process the four calls to G_i of the diagonal step in the following order: G_7, G_4, G_5, and G_6. We check for instance that the ALU outputs the new value of v_4 (last instruction of G_0) at time $\tau+3$. If we load v_3 from the register file, we can start the computation of G_7 at time $\tau + 4$ (Figure 13). We easily check that this scheduling also avoids pipeline bubbles between a diagonal step and a column step (Figure 14). Since each call to G_i involves ten instructions, we need 80 clock cycles to perform a round of BLAKE-n.

Our first architecture can be modified to support the four algorithms of the BLAKE family (Figure 15). The 64-bit datapath is built out of two 32-bit datapaths, thus allowing us to perform a single 64-bit operation or two 32-bit operations at each clock cycle. The mode of operation is selected according to an additional control bit ctrl_6, the latter being provided by the user. The ALU includes two 32-bit adders. Let a_{low}, a_{high}, b_{low}, b_{high}, s_{low}, and s_{high} denote unsigned 32-bit integers. When the user chooses BLAKE-224 or BLAKE-256 ($\text{ctrl}_6 = 0$), two messages are processed in parallel and the ALU performs two 32-bit additions:

$$s_{\text{low}} \leftarrow a_{\text{low}} + b_{\text{low}} + \text{ctrl}_2,$$
$$s_{\text{high}} \leftarrow a_{\text{high}} + b_{\text{high}} + \text{ctrl}_2.$$

When the coprocessor executes BLAKE-384 or BLAKE-512

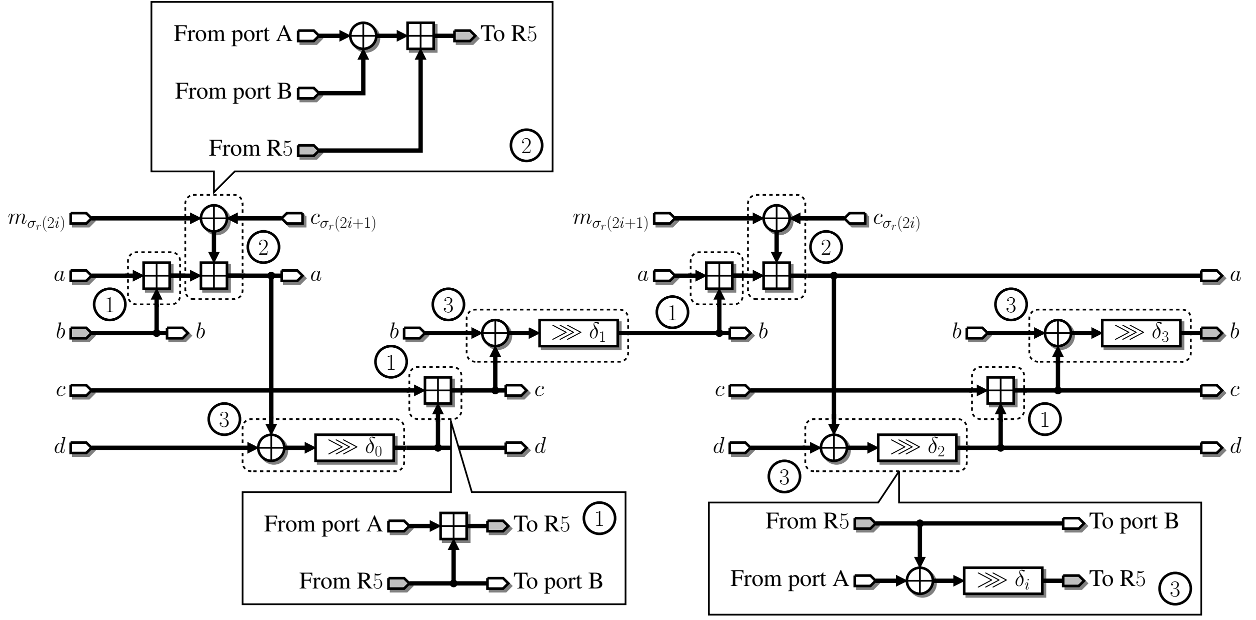

Figure 11. Implementation of the G_i function of BLAKE-n by means of three instructions. R5 denotes an internal register of the ALU.

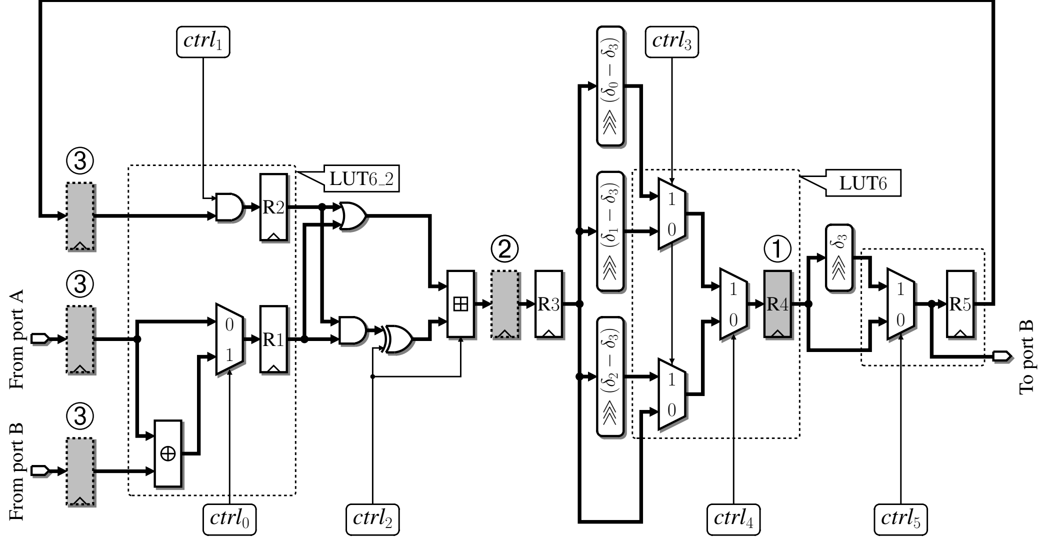

Figure 12. Arithmetic and Logic Unit for BLAKE-n.

($\mathrm{ctrl}_6 = 1$), the ALU carries out a 64-bit addition. The first adder generates the least significant bits of the sum and a carry bit c such that:

$$2^{32} \cdot c + s_{\text{low}} = a_{\text{low}} + b_{\text{low}} + \mathrm{ctrl}_2.$$

The second adder computes the most significant bits of the sum:

$$s_{\text{high}} \leftarrow a_{\text{high}} + b_{\text{high}} + c.$$

We use the rotation unit of our first processor to deal with BLAKE-224 and BLAKE-256. Note that the content of R3 is always copied into R6 when $\mathrm{ctrl}_{5:3} = (000)_2$. Thus, we share this datapath between all algorithms of the BLAKE family, and need only 64 LUTs to implement the rotation unit of BLAKE-384 and BLAKE-512. When ctrl_5 is equal to one, $\mathrm{ctrl}_{4:3}$ encodes the index i of the rotation distance δ_i (Table V). Consequently, we can use the same instruction flow for all algorithms and select the width of the datapath according to ctrl_6. Note that the three pipeline configurations defined for our first coprocessor are also available here.

The QUARTERROUND function of ChaCha requires only two of the instructions we defined for the G_i function. Thus, the design of a ChaCha coprocessor is rather straightforward (Figure 16). Since it is not necessary to compute $\mathrm{RF}_A \oplus \mathrm{RF}_B$ anymore, the ALU has a single 32-bit input. The only difficulty is to increment the 64-bit counter $2^{32}m_{13}+m_{12}$ (Algorithm 3, lines 17 to 20). Assume that the constant 1 is stored in the

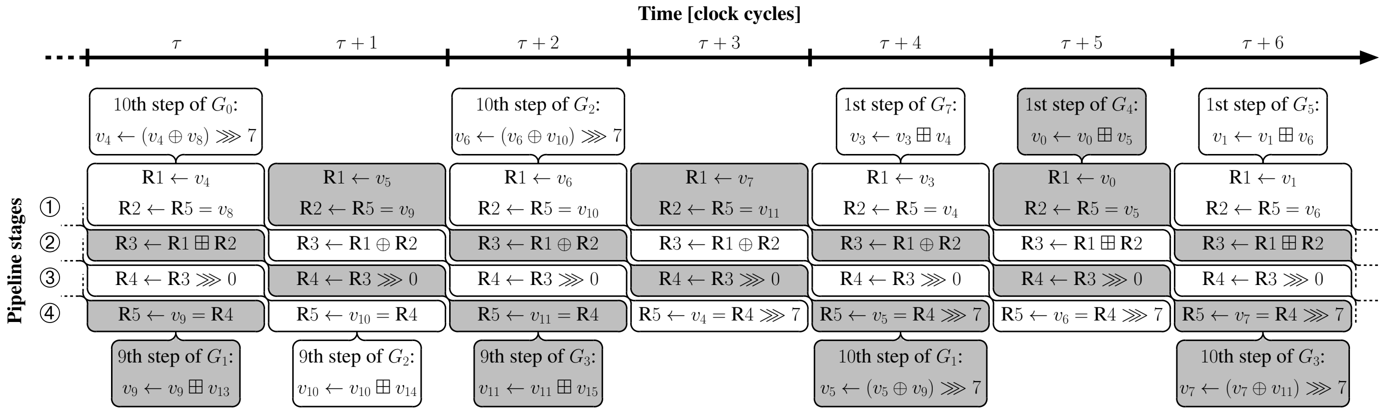

Figure 13. Avoiding pipeline bubbles between a column step and a diagonal step.

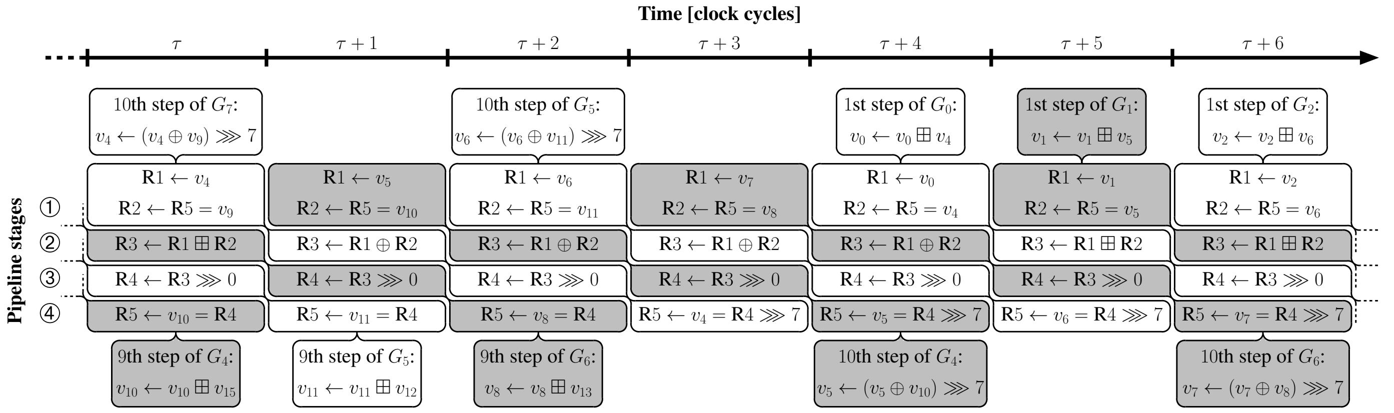

Figure 14. Avoiding pipeline bubbles between a diagonal step and a column step.

Table V
ROTATION DISTANCES OF THE UNIFIED BLAKE COPROCESSOR.

$\mathbf{ctrl_{5:3}}$	Rot. dist.	BLAKE-224/256	BLAKE-384/512
$(000)_2$	0	$R_6 \leftarrow R_3$ (common datapath)	
$(100)_2$	δ_0	$R_6 \leftarrow R_3 \ggg 7$	$R_6 \leftarrow R_3 \ggg 11$
$(101)_2$	δ_1	$R_6 \leftarrow R_3 \ggg 8$	$R_6 \leftarrow R_3 \ggg 16$
$(110)_2$	δ_2	$R_6 \leftarrow R_3 \ggg 12$	$R_6 \leftarrow R_3 \ggg 25$
$(111)_2$	δ_3	$R_6 \leftarrow R_3 \ggg 16$	$R_6 \leftarrow R_3 \ggg 32$

register file. The control bit ctrl_0 allows us to disable the feedback mechanism and to load the constant 0 in register R2. Execute the following instructions:

$$\begin{aligned} &\text{R1} \leftarrow 1 && \text{R2} \leftarrow 0, \\ &\text{R3} \leftarrow \text{R1} \oplus \text{R2}, \\ &\text{R4} \leftarrow \text{R3}, \\ &\text{R5} \leftarrow \text{R4}, \\ &\text{R1} \leftarrow m_{12} && \text{R2} \leftarrow \text{R5}. \end{aligned}$$

Registers R1 and R2 store m_{12} and the constant 1, respectively. Note that the output carry of the 32-bit adder can now be stored in a flip-flop F. Furthermore, when ctrl_2 is set to one, our ALU performs an "add with carry" instruction. We can now compute $m_{12} + 1$, save the output carry in F, and increment m_{13} if necessary:

$$\begin{aligned} (\text{F}, \text{R3}) &\leftarrow \text{R1} \boxplus \text{R2} && \text{R1} \leftarrow m_{13} \quad \text{R2} \leftarrow 0, \\ \text{R3} &\leftarrow \text{R1} \boxplus \text{R2} \boxplus \text{F} && \text{R4} \leftarrow \text{R3}, \\ \text{Register file} &\leftarrow \text{R4}\ (m_{12}) && \text{R4} \leftarrow \text{R3}, \\ \text{Register file} &\leftarrow \text{R4}\ (m_{13}). \end{aligned}$$

Three pipeline configurations are again available. The second one needs specific attention: since the adder is pipelined, the computation of $m_{12} \boxplus 1$ requires two clock cycles. It is therefore mandatory to introduce a NOP before loading m_{13} into register R1.

It is of course possible to build a unified coprocessor for ChaCha and the BLAKE family (Figure 17). A new control bit ctrl_7 allows the user to select the mode of operation of the ALU: encryption or hashing. Since the coprocessor has a 64-bit datapath to support BLAKE-384 and BLAKE-512, it is possible to encrypt two messages in parallel with ChaCha.

C. Register Files and Control Units

We will consider our unified coprocessor for the BLAKE and ChaCha algorithms to describe how we design our control units. The same approach can easily be applied to the other coprocessors considered in this work. Virtex-6 FPGAs embed several configurable memory blocks that can for instance

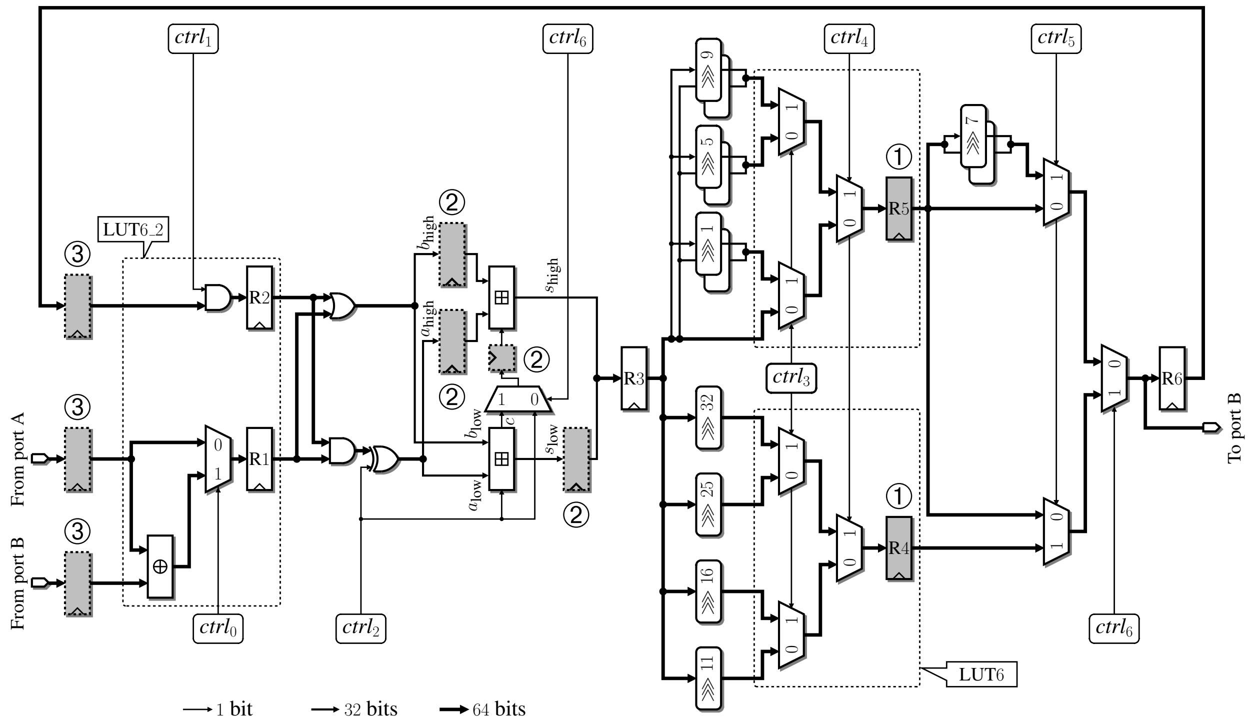

Figure 15. Unified Arithmetic and Logic Unit for the BLAKE family.

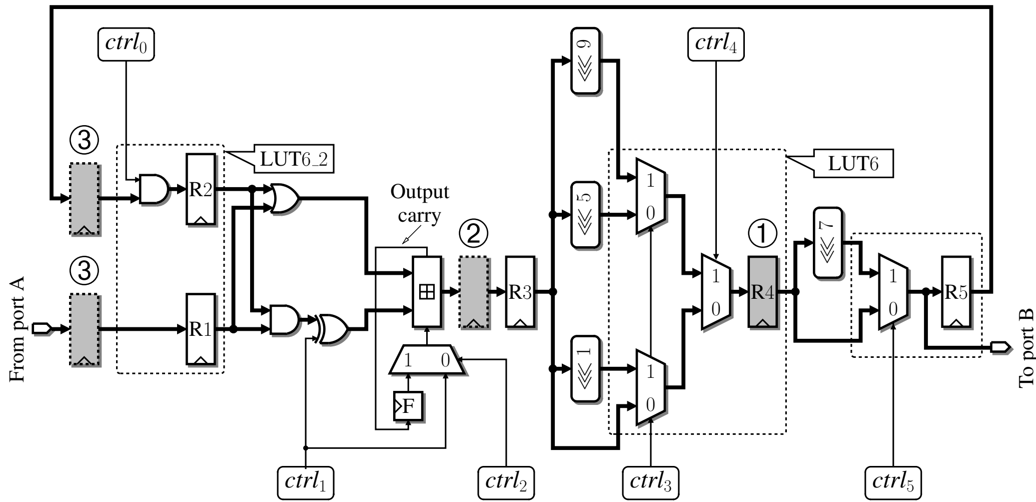

Figure 16. Arithmetic and Logic Unit for ChaCha.

store 1024 36-bit words or 2048 18-bit words. Our control unit mainly consists of a program counter that addresses an instruction memory implemented by means of a memory block.

A straightforward way to deal with the permutations involved in the BLAKE family is to unroll the round loop. Table VI summarizes the number of instructions required by the algorithms supported by our coprocessor if we follow this approach. Note that it suffices to store the code of BLAKE-384/512 and 20-round ChaCha (2039 instructions): a simple finite-state machine allows us to jump to the finalization step when the desired number of rounds has been performed[1]. The

Table VI
NUMBER OF INSTRUCTIONS OF THE ALGORITHMS OF THE BLAKE AND CHACHA FAMILIES.

Algorithm	# instructions
BLAKE-224/256	1184
BLAKE-384/512	1344
8-round ChaCha	311
12-round ChaCha	439
20-round ChaCha	695

[1]It is possible to reduce the size of the code by storing the table defining the permutation of $\{0 \ldots, 15\}$ parametrized by the round index r (Table IV) and by generating the addresses of $m_{\sigma_r(2i)}$ and $c_{\sigma_r(2i+1)}$ on-the-fly. However, this approach would require a more complex control unit. As long as the micro-code fits into a single block of memory, there is no need to try to reduce the number of instructions.

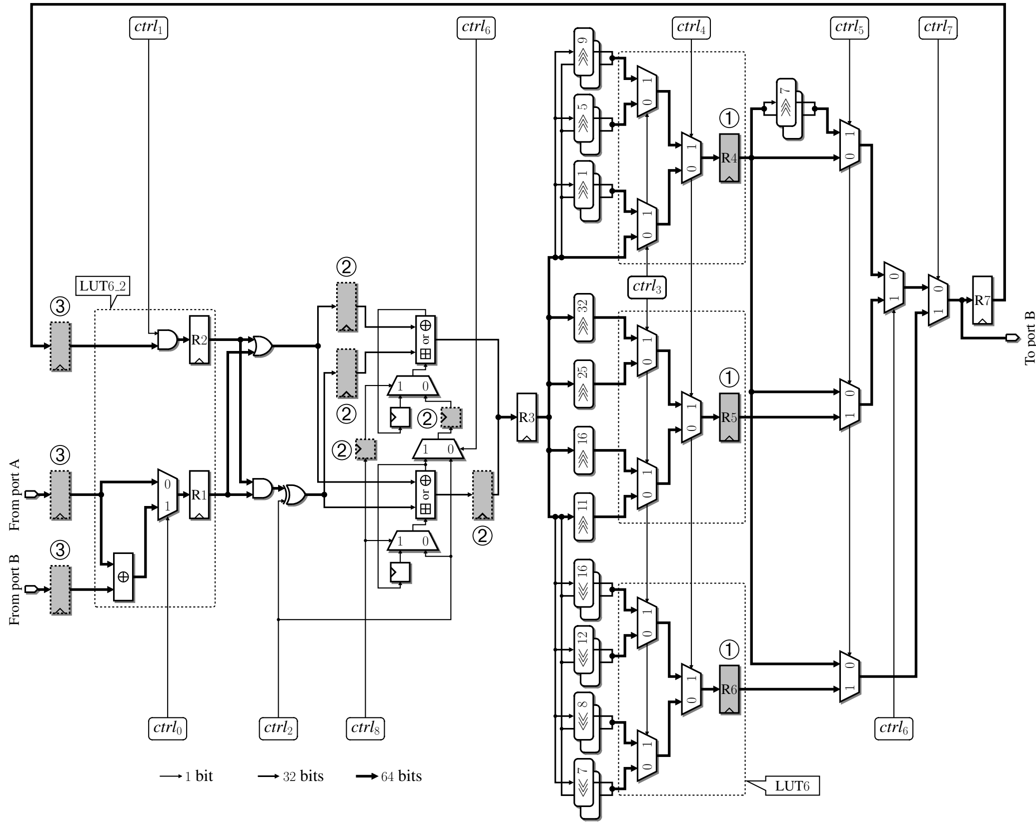

Figure 17. Unified Arithmetic and Logic Unit for the BLAKE and ChaCha families.

main challenge is therefore to define control words of at most 18 bits in order to implement our instruction memory by means of a single memory block. A clever organization of the register file (Figure 18) and a simple compression algorithm allows us to achieve this goal. Two blocks of dual-ported memory configured as 256 entries of 32 bits store the message, the chaining value, the constants, and all the intermediate variables of BLAKE and ChaCha. Thus, our coprocessor requires 26 control bits (Figure 19a):

- 8 address bits and a write enable signal for port A of the register file;
- 8 address bits and a write enable signal for port B of the register file;
- 8 control bits for the ALU.

Two control bits are provided by the user: ctrl_7 allows her to select between BLAKE and ChaCha, and ctrl_6 specifies the configuration of the datapath (2×32 bits or 64 bits). Our organization of the data in the register file enables us to define a 20-bit instruction:

- The most significant address bit depends on the algorithm being executed, and is therefore provided by the user.
- We use ports A and B to load new data (message, salt, and counter) and save the intermediate variables computed by the ALU, respectively. Consequently, the write enable signal of port A is also given by the user.
- Let us denote by $a_{7:0}$ the eight address bits of ports A. Note that a_6 is equal to one only when we read an initial vector and assume that the digest size is selected according to an additional control bit ctrl_8. The address bit a_5 is computed as follows:

$$a_5 \leftarrow \begin{cases} a_5 & \text{when } a_6 = 0, \\ \text{ctrl}_8 & \text{otherwise.} \end{cases}$$

  Thanks to this simple mechanism, the instruction flow does not depend on the digest size. Initial vectors are always read from port A.
- Since the initial vectors are neither modified nor read from port B, the second most significant address bit is always equal to zero.

Consequently, we can store 20-bit words in the instruction memory (Figure 19b). We designed a simple compression algorithm to encode the write enable signal of port B and the six control bits $\text{ctrl}_{5:0}$ by means of five bits. A C program generates the content of the instruction memory and the VHDL description of the decompression circuit. The latter involves only seven 5-input LUTs, and stores the control bits of the ALU and the write enable signal of port B in a register. Because of this pipeline stage, it is necessary to generate the write enable signal one clock cycle in advance when we have to store a word in the register file. Our C program takes this parameter into account and organizes the control bits in the instruction memory according to the pipeline

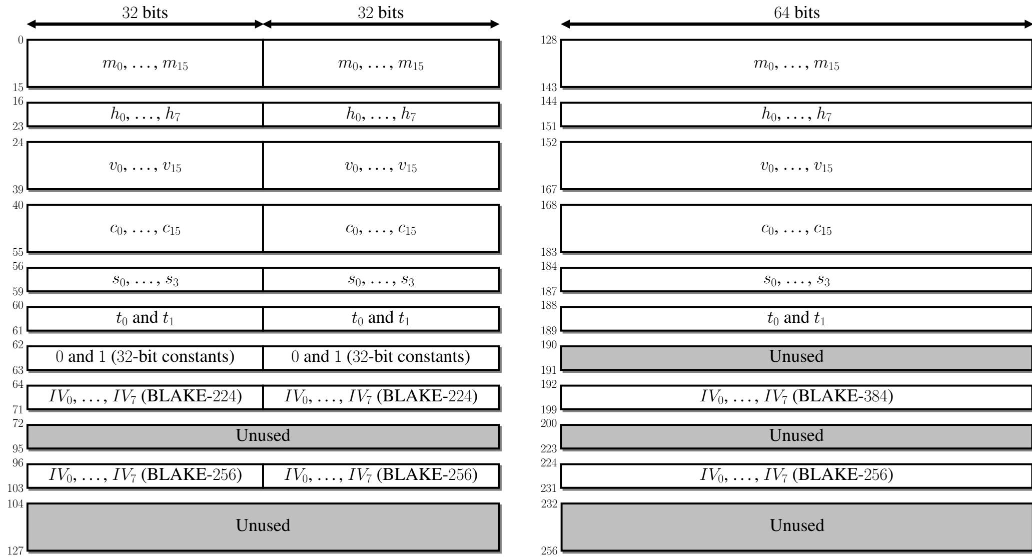

Figure 18. Register file of the unified coprocessor for the BLAKE and ChaCha families.

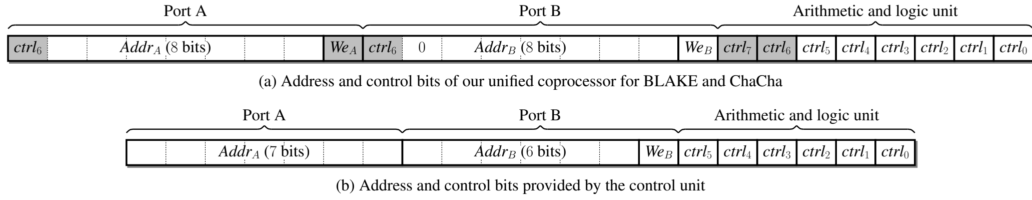

Figure 19. From 26- to 18-bit instructions. Shaded cells denote control bits provided by the user.

configuration. Then, it generates the compressed instruction memory. Figure 20 describes the instruction flow for the first pipeline configuration of our coprocessor:

- As explained above, the write enable signal is generated one clock cycle in advance to take the internal pipeline stage of the decompression unit into account.
- All inputs of the register file are registered, and the two control bits ctrl_0 and ctrl_1 must therefore be generated one clock cycle after the addresses. We take advantage of the latency of our decompression unit to synchronize the control signals.

We followed the same approach to build our control units for Threefish and Skein. 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 (Figure 21). 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. The number of clock cycles required for Threefish encryption and decryption according to the key size is summarized in Table VII. Because of the output transform, $k+1$ invocations of the tweakable encryption function are necessary to hash a k-block message with Skein. There is a latency of 5 clock cycles between two consecutive Threefish encryption. Thus, the throughput of Skein is given by:

$$\textit{throughput} = \frac{8 \cdot N_b \cdot k \cdot f}{(k+1) \cdot \textit{latency of Threefish with UBI} + 5 \cdot k},$$

where f denotes the clock frequency.

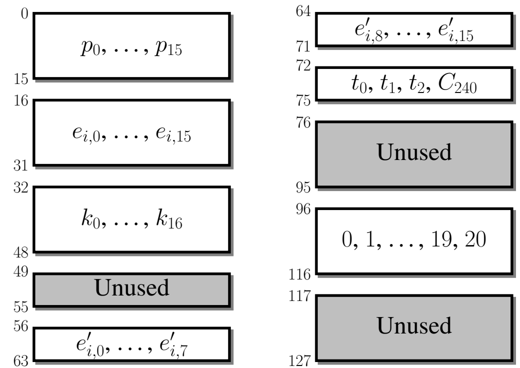

Figure 21. Register file of our Threefish and Skein architectures.

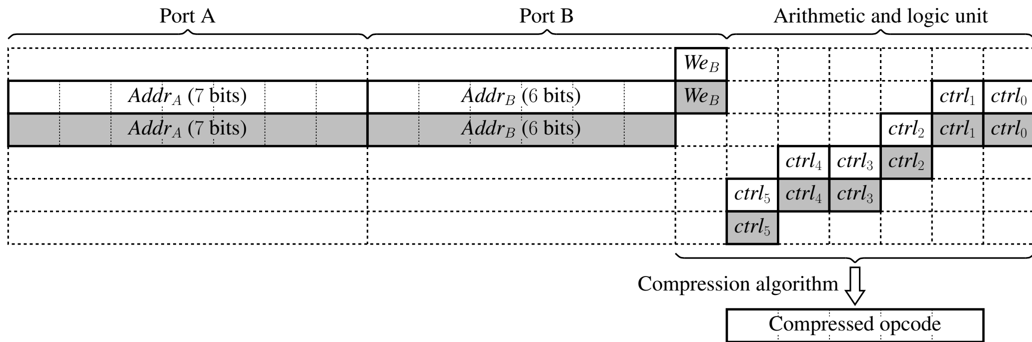

Figure 20. Generation of the compressed instruction memory.

Table VII
NUMBER OF INSTRUCTIONS OF THE ALGORITHMS OF THE THREEFISH FAMILY.

Algorithm	# instructions
Threefish-256 encryption	490
Threefish-256 encryption with UBI	501
Threefish-256 decryption	469
Threefish-512 encryption	860
Threefish-512 encryption with UBI	882
Threefish-512 decryption	1092
Threefish-1024 encryption	1874
Threefish-1024 encryption with UBI	1912
Threefish-1024 decryption	2351

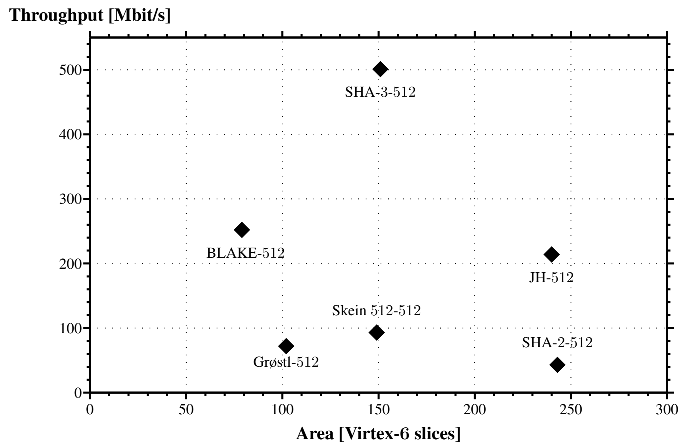

Figure 22. Compact implementations of several cryptographic hash functions on Virtex-6 FPGAs (512-bit digests).

VII. RESULTS AND COMPARISONS

We captured our architecture in the VHDL language and prototyped our coprocessors on a Xilinx Virtex-6 FPGA with average speedgrade. Tables VIII and IX summarize our place-and-route results measured with ISE 14.2. Note that we considered the least favorable case, where the message consists of a single block, to compute the throughput of Skein.

Most of the architectures described in the open literature focus on a single level of security (Table X). We took advantage of the intrinsic parallelism of BLAKE to interleave the computation of four instances of the G_i function. Thanks to this approach, we designed an ALU with four pipeline stages and achieved higher clock speeds than the coprocessors listed in Table X. A careful scheduling allowed us to totally avoid pipeline bubbles and memory collisions. We also addressed FPGA-specific issues and described how to share slices between addition and bitwise exclusive OR of two operands. We followed the same strategy to design our coprocessors for Threefish and Skein. As a consequence, our coprocessors provide the end-user with hashing and encryption at all levels of security, while offering a better area–time trade-off.

We report in Figure 22 the latest lightweight implementation results of several cryptographic hash functions. Besides our coprocessors for BLAKE-512 and Skein-512-512, we selected Grøstl [14], JH [5], SHA-2-512 [21], and SHA-3-512 (Keccak $[r = 1024, c = 576]$) [13]. In this context, BLAKE is obviously the best choice for lightweight implementations on FPGA. Since our unified architecture for the BLAKE family (Figure 15) requires less than 100 Virtex-6 slices, BLAKE is also an excellent candidate for cryptographic coprocessors supporting several levels of security.

We already proposed lightweight implementations of ECHO & AES [15] and Grøstl & AES[2] [14] (Table XI). According to our results, the unified coprocessor for BLAKE and ChaCha offers the best area–time trade-off. However, given that all symmetric cryptographic functions (including authenticated encryption) can be efficiently implemented with Keccak, we would get the following figures with a unified architecture based on [13]:

- hashing with arbitrary length at a security level of 256 bits: 501 Mbits/s;
- authenticated encryption at a security level of 256 bits: more than 501 Mbits/s (the generic security of keyed sponges allows one to use less capacity than for hashing, hence a larger rate and a proportionally larger throughput) [24].

[2]Note that Järvinen [22] proposed the first unified coprocessor for AES-128 (encryption and key expansion) and Grøstl-256. Recently, Rogawski & Gaj [23] designed a parallel coprocessor for Grøstl-based HMAC and AES in the counter mode. Both architectures are optimized for high-speed implementations, and it is therefore difficult to make a comparison with our lightweight coprocessors.

Table VIII
PLACE-AND-ROUTE RESULTS FOR OUR THREEFISH AND SKEIN COPROCESSORS ON A VIRTEX-6 FPGA (XC6VLX75T-2). ALL DESIGNS REQUIRE THREE MEMORY BLOCKS.

Supported Algorithms	Area [Slices]	Freq. [MHz]	Throughput [Mbits/s]							
			Skein-256-256	Skein-512-512	Threefish-256		Threefish-512		Threefish-1024	
					Enc.	Dec.	Enc.	Dec.	Enc.	Dec.
Threefish encryption	145	294	–	–	153	–	175	–	160	–
Threefish	277	267	–	–	139	145	158	125	145	116
Skein and Threefish encryption	150	295	75	85	154	–	175	–	161	–
Skein and Threefish	292	279	70	80	145	152	166	130	152	121

Table IX
PLACE-AND-ROUTE RESULTS FOR OUR CHACHA AND BLAKE COPROCESSORS ON A VIRTEX-6 FPGA (XC6VLX75T-2).

Supported algorithms	Pipeline config.	Area [slices]	# block RAMs	Freq. [MHz]	Throughput [Mbits/s]				
					BLAKE-224 and BLAKE-256	BLAKE-384 and BLAKE-512	8-round ChaCha	12-round ChaCha	20-round ChaCha
ChaCha	①	49	2	362	–	–	595	422	266
	②	77	2	316	–	–	520	368	232
	③	77	2	345	–	–	569	403	254
BLAKE-224 and BLAKE-256	①	47	2	338	146	–	–	–	–
	②	49	2	341	147	–	–	–	–
	③	50	2	349	150	–	–	–	–
BLAKE-384 and BLAKE-512	①	79	3	331	–	252	–	–	–
	②	91	3	331	–	252	–	–	–
	③	91	3	329	–	250	–	–	–
BLAKE (all levels of security)	①	94	3	312	2×134	237	–	–	–
	②	126	3	332	2×143	252	–	–	–
	③	129	3	343	2×148	261	–	–	–
BLAKE and ChaCha (all levels of security)	①	144	3	335	2×144	255	2×551	2×390	2×246
	②	156	3	289	2×124	220	2×475	2×337	2×212
	③	168	3	304	2×131	231	2×500	2×354	2×223

VIII. CONCLUSION

The stream cipher ChaCha, the block cipher Threefish, and the hash functions BLAKE and Skein are based on the same arithmetic operations. In this work, we showed that the same design philosophy allows one to design lightweight coprocessors for hashing and encryption. The key element of our approach is to take advantage of the parallelism of the algorithms to:

- deeply pipeline the ALU to achieve a high clock frequency;
- avoid data dependencies by interleaving independent tasks.

Furthermore, we described how to design compact control units thanks to a careful organization of the register file, loop unrolling, and a simple compression algorithm. Our architectures are mainly designed for embedded systems. Thus, it would be interesting to conduct side-channel and fault injection attacks in future work.

Our results show that BLAKE and ChaCha are excellent candidates for lightweight coprocessors. However, since all symmetric cryptographic functions can be implemented by means of keyed sponges, we are planning to design hardware architectures based on the new SHA-3 algorithm. According to the preliminary results reported in [13], SHA-3 could outperform BLAKE and ChaCha.

ACKNOWLEDGEMENTS

The authors would like to thank Daniel J. Bernstein and Ray Cheung for their valuable comments. This work was partially supported by the Japanese Society of Promotion of Science (JSPS) through the A3 Foresight Program (Research on Next Generation Internet and Network Security). Additionally the authors would like to acknowledge Xilinx and the Xilinx University Program for its generous donation of materials in terms of design tools.

REFERENCES

[1] J.-P. Aumasson, L. Henzen, W. Meier, and R. Phan, “SHA-3 proposal BLAKE (version 1.4),” Jan. 2011, available at http://www.131002.net/blake.

[2] 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, available at http://www.skein-hash.info.

[3] D. Bernstein, “ChaCha, a variant of Salsa20,” Jan. 2008, available at http://cr.yp.to/papers.html#chacha.

Table X
COMPACT IMPLEMENTATIONS OF BLAKE AND SKEIN ON VIRTEX-5 AND VIRTEX-6 FPGAS. THE THROUGHPUT IS COMPUTED FOR A ONE-BLOCK MESSAGE.

	Supported algorithm(s)	FPGA	Area [slices]	36k memory blocks	Frequency [MHz]	Throughput [Mbits/s]
Latif *et al.* [17]‡	Skein-256-256	xc5vlx110-3	821	Not specified	119	1610
Jungk [18]‡	Skein-512-256	xc5v	555	–	271	237
Jungk [19]‡	Skein-512-256	xc6v	406	–	318	277
Kaps *et al.* [20]	Skein-512-256	xc6vlx75t-1	207	1	166	17
Kaps *et al.* [20]	Skein-512-256	xc6vlx75t-1	193	–	193	21
Kerckhof *et al.* [5]‡	Skein-512-512	xc6vlx75t-1	240	–	160	179
Aumasson *et al.* [1]	BLAKE-256	xc5vlx110	390	–	91	412
Jungk [19]	BLAKE-256	xc6v	235	–	231	518
Jungk [19]	BLAKE-256	xc6v	404	–	185	823
Kaps *et al.* [20]	BLAKE-256	xc6vlx75t-1	163	1	197	327
Kaps *et al.* [20]	BLAKE-256	xc6vlx75t-1	166	–	268	445
Aumasson *et al.* [1]	BLAKE-512	xc5vlx110	939	–	59	468
Kerckhof *et al.* [5]	BLAKE-512	xc6vlx75t-1	192	–	240	183

†Without output transformation ‡Single call to Threefish-512

Table XI
PLACE-AND-ROUTE RESULTS FOR HASHING AND AES ENCRYPTION ON A VIRTEX-6 FPGA (XC6VLX75T-2).

FPGA	Area [slices]	Frequency [MHz]	Throughput [Mbits/s]				
			AES-128	AES-192	AES-256	256-bit digest	512-bit digest
AES & ECHO [15]	155	397	219	186	161	92	48
AES & Grøstl [14]	169	393	217	184	159	92	69

[4] 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, M. Gavrilova, O. Gervasi, V. Kumar, C. K. Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, Eds., no. 3983. Springer, 2006, pp. 296–305.

[5] 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.

[6] N. At, J.-L. Beuchat, and İ. San, "Compact implementation of Threefish and Skein on FPGA," in *Proceedings of the Fifth IFIP International Conference on New Technologies, Mobility and Security–NTMS 2012*, A. Levi, M. Badra, M. Cesana, M. Ghassemian, O. Gürbüz, N. Jabeur, M. Klonowski, A. Maña, S. Sargento, and S.Zeadally, Eds. IEEE eXpress Conference Publishing, 2012.

[7] 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.

[8] D. Bernstein, "The Salsa20 family of stream ciphers," Dec. 2007, available at http://cr.yp.to/snuffle/salsafamily-20071225.pdf.

[9] J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier, and C. Rechberger, "New features of Latin dances: Analysis of Salsa, ChaCha, and Rumba," in *Fast Software Encryption–FSE 2008*, ser. Lecture Notes in Computer Science, K. Nyberg, Ed., vol. 5086. Springer, 2008, pp. 470–488.

[10] T. Ishiguro, S. Kiyomoto, and Y. Miyake, "Latin dances revisited: New analytic results of Salsa20 and ChaCha," in *Information and Communications Security–ICICS 2011*, ser. Lecture Notes in Computer Science, S. Qing, W. Susilo, G. Wang, and D. Liu, Eds., vol. 7043. Springer, 2011, pp. 255–266.

[11] T. Ishiguro, "Modified version of "Latin dances revisited: New analytic results of Salsa20 and ChaCha"," 2012, cryptology ePrint Archive, Report 2012/065.

[12] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. Marnane, "A hardware wrapper for the SHA-3 hash algorithms," 2010, cryptology ePrint Archive, Report 2010/124.

[13] İ. San and N. At, "Compact Keccak hardware architecture for data integrity and authentication on FPGAs," *Information Security Journal: A Global Perspective*, vol. 21, no. 5, pp. 231–242, 2012.

[14] N. At, J.-L. Beuchat, E. Okamoto, İ. San, and T. Yamazaki, "A low-area unified hardware architecture for the AES and the cryptographic hash function Grøstl," 2012, cryptology ePrint Archive, Report 2012/535.

[15] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "A low-area unified hardware architecture for the AES and the cryptographic hash function ECHO," *Journal of Cryptographic Engineering*, vol. 1, no. 2, pp. 101–121, 2011.

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

[17] 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.

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

[19] ——, "Evaluation of compact FPGA implementations for all SHA-3 finalists," in *The Third SHA-3 Candidate Conference*, Mar. 2012.

[20] J.-P. Kaps, P. Yalla, K. Surapathi, B. Habib, S. Vadlamudi, and S. Gurung, "Lightweight implementations of SHA-3 finalists on FPGAs," in *The Third SHA-3 Candidate Conference*, Mar. 2012.

[21] H. Technology, "FULL DATASHEET–Tiny hash core family for Xilinx FPGA," revision 2.0 (11/06/2010).

[22] K. Järvinen, "Sharing resources between AES and the SHA-3 second round candidates Fugue and Grøstl," in *The Second SHA-3 Candidate Conference*, Aug. 2010.

[23] M. Rogawski and K. Gaj, "A high-speed unified hardware architecture for AES and the SHA-3 candidate Grøstl," in *Proceedings of the 15th Euromicro Conference on Digital System Design*, Sep. 2012.

[24] The Keccak Team, Personal communication, Sep. 2012.