<table>
<thead>
<tr>
<th>著者</th>
<th>アカデミック サイエンス レポート</th>
<th>説明書の内容</th>
</tr>
</thead>
<tbody>
<tr>
<td>優先順位</td>
<td>1</td>
<td>説明書の内容</td>
</tr>
<tr>
<td>言語</td>
<td>日本語</td>
<td>説明書の内容</td>
</tr>
<tr>
<td>権利</td>
<td>アカデミック サイエンス レポート</td>
<td>説明書の内容</td>
</tr>
</tbody>
</table>

doi: 10.1109/TCSI.2013.2278385
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: jeanj luc.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]. 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 discuss 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

<table>
<thead>
<tr>
<th>Key size [bits]</th>
<th># 64-bit words ( N_w )</th>
<th># rounds ( N_r )</th>
<th>Block size ( N_b ) [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4</td>
<td>72</td>
<td>32</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>72</td>
<td>64</td>
</tr>
<tr>
<td>1024</td>
<td>16</td>
<td>80</td>
<td>128</td>
</tr>
</tbody>
</table>

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 \( T = (t_0, t_1) \). Each subkey is a combination of \( N_w \) words of the extended key, two words of the extended tweak, and a counter \( s \) (Algorithm I 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 I lines [1] to [2]). 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]).

<table>
<thead>
<tr>
<th>i</th>
<th>( \pi(i) )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>2</td>
<td>13</td>
<td>6</td>
<td>11</td>
<td>4</td>
<td>15</td>
</tr>
</tbody>
</table>

Algorithm 1 Key schedule of Threefish.

**Input:** A block cipher key \( K = (k_0, k_1, \ldots, k_{N_w-1}); \)
\( N_t \) a tweak \( T = (t_0, t_1); \) the constant \( C_{240} = 1BD1BDA9FC1A22. \)

**Output:** \( N_r/4 + 1 \) subkeys \( k_{s,0}, k_{s,1}, \ldots, k_{s,N_w-1}, \) where 
\( 0 \leq s \leq N_r/4. \)

1. \( k_{N_w} \leftarrow C_{240} \oplus \bigoplus_{i=0}^{N_w-1} k_i; \)
2. \( t_0 \leftarrow t \oplus t_1; \)
3. for \( s \leftarrow 0 \) to \( N_r/4 \) do
4. for \( i \leftarrow 0 \) to \( N_w - 4 \) do
5. \( k_{s,i} \leftarrow k_{i(s+1) \mod (N_w+1)}; \)
6. end for
7. \( k_{s,N_w-3} \leftarrow k_{i(s+N_w-3) \mod (N_w+1)} \oplus t_{s \mod 3}; \)
8. \( k_{s,N_w-2} \leftarrow k_{i(s+N_w-2) \mod (N_w+1)} \oplus t_{(s+1) \mod 3}; \)
9. \( k_{s,N_w-1} \leftarrow k_{i(s+N_w-1) \mod (N_w+1)} \oplus s; \)
10. end for
11. return \( k_{s,0}, k_{s,1}, \ldots, k_{s,N_w-1}, \) 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 \( \text{Mix}_{d,j} \) (Algorithm 2 lines 13 and 14). It consists of an addition, a rotation by a constant \( R_d \mod 8, j \) (repeated every eight rounds and defined in Table 2), 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 \( \text{Mix} \) functions. Note that subkeys are injected in reverse order.

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 \( 2^{64} - 8 \) 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 \( N_r \)-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:

\[
H_0 \leftarrow G,
\]

\[
H_{i+1} \leftarrow M_i \oplus E(T_i, M_i),
\]

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 \( P = (p_0, p_1, \ldots, p_{N_s-1}); N_s/4 + 1 \) subkeys \( k_{s,0}, k_{s,1}, \ldots, k_{s,N_s-1} \), where \( 0 \leq s \leq N_r/4; \) 4\(N_r\) rotation constants \( R_{i,j} \), where \( 0 \leq i \leq 7 \) and \( 0 \leq j \leq N_r/2 \).

Output: A ciphertext block \( C = (c_0, c_1, \ldots, c_{N_r-1}) \).

1. for \( i \leftarrow 0 \) to \( N_r - 1 \) do
2. \( v_0,i \leftarrow p_i; \)
3. end for
4. for \( d \leftarrow 0 \) to \( N_r - 1 \) do
5. for \( i \leftarrow 0 \) to \( N_r - 1 \) do
6. if \( d \mod 4 = 0 \) then
7. \( e_{d,i} \leftarrow v_{d,i} \oplus k_{d/4,i}; \)
8. else
9. \( e_{d,i} \leftarrow v_{d,i}; \)
10. end if
11. end for
12. for \( j \leftarrow 0 \) to \( N_r/2 - 1 \) do
13. \( f_{d,2j} \leftarrow e_{d,2j} \oplus e_{d,2j+1}; \)
14. \( f_{d,2j+1} \leftarrow f_{d,2j} \otimes (e_{d,2j+1} \ll R_{d \mod s,j}); \)
15. end for
16. for \( i \leftarrow 0 \) to \( N_r - 1 \) do
17. \( v_{d+1,i} \leftarrow f_{d,\pi(i)}; \)
18. end for
19. end for
20. for \( i \leftarrow 0 \) to \( N_r - 1 \) do
21. \( c_i \leftarrow v_{N_{r-i},i} \oplus k_{N_r/4,i}; \)
22. end for
23. return \( C = (c_0, c_1, \ldots, c_{N_r-1}) \);

and a version number [2 Table 7]. Compute the \( N_r \)-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 \((k_{0,0}, \ldots, k_{7,7})\) and a 64-bit nonce \((IV_0, IV_1)\) into a \(2^{70}\)-byte stream. A \( b \)-byte message is 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 \( i \)-th block, ChaCha acts on a \( 4 \times 4 \) matrix \( M \) of 32-bit integers defined as follows:
\[
\begin{bmatrix}
    m_0 & m_1 & m_2 & m_3 \\
    m_4 & m_5 & m_6 & m_7 \\
    m_8 & m_9 & m_{10} & m_{11} \\
    m_{12} & m_{13} & m_{14} & m_{15}
\end{bmatrix} =
\begin{bmatrix}
    c_0 & c_1 & c_2 & c_3 \\
    k_0 & k_1 & k_2 & k_3 \\
    k_4 & k_5 & k_6 & k_7 \\
    t_0 & t_1 & IV_0 & IV_1
\end{bmatrix},
\]

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
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. \( v[i] \leftarrow m[i]; \)
3. end for
4. for \( i \leftarrow 0 \) to \( N_r/2 - 1 \) do
5. QUARTERROUND\((v_0, v_4, v_8, v_{12});\)
6. QUARTERROUND\((v_1, v_5, v_9, v_{13});\)
7. QUARTERROUND\((v_2, v_6, v_{10}, v_{14});\)
8. QUARTERROUND\((v_3, v_7, v_{11}, v_{15});\)
9. end for
10. for \( i \leftarrow 0 \) to \( 15 \) do
11. \( v[i] \leftarrow v[i] \oplus m[i]; \)
12. end for
13. \( m_{12} \leftarrow m_{12} \ll 1; \)
14. \( m_{13} \leftarrow m_{13} \ll 1; \)
15. end if
16. 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. \( a \leftarrow a \oplus b; \)
2. \( d \leftarrow (d \oplus a) \ll 16; \)
3. \( c \leftarrow c \oplus d; \)
4. \( b \leftarrow (b \oplus c) \ll 12; \)
5. \( a \leftarrow a \oplus b; \)
6. \( d \leftarrow (d \oplus a) \ll 8; \)
7. \( c \leftarrow c \oplus d; \)
8. \( b \leftarrow (b \oplus c) \ll 7; \)
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 \( \delta_j \) 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 \( h' = h'_0, \ldots, h'_2 \) 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:

\[
\begin{align*}
&v_0 \quad v_1 \quad v_2 \quad v_3 \\
v_4 \quad v_5 \quad v_6 \quad v_7 \\
v_8 \quad v_9 \quad v_{10} \quad v_{11} \\
v_{12} \quad v_{13} \quad v_{14} \quad v_{15} \\
&\leftarrow \begin{pmatrix}
h_0 & h_1 & h_2 & h_3 \\
h_4 & h_5 & h_6 & h_7 \\
s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\
l_0 \oplus c_4 & l_0 \oplus c_5 & l_1 \oplus c_6 & l_1 \oplus c_7
\end{pmatrix}
\end{align*}
\]

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 \( \sigma_r \) 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 \( v \), four pipeline stages and to achieve high clock frequencies.

At the end of the last round, a new chain value \( h' = h'_0, \ldots, h'_2 \) is computed from the internal state \( v \) and the previous chain value \( h \) (finalization step):

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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Word size ( w ) [bits]</th>
<th>Message size [bits]</th>
<th>Block size [bits]</th>
<th>Digest size [bits]</th>
<th>Salt size [bits]</th>
<th># rounds ( N_r )</th>
<th>Rotation distances ( \delta_0, \delta_1, \delta_2, \delta_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE-256</td>
<td>32</td>
<td>(&lt; 2^{256})</td>
<td>512</td>
<td>128</td>
<td>14</td>
<td>16</td>
<td>12, 8, 7</td>
</tr>
<tr>
<td>BLAKE-256</td>
<td>32</td>
<td>(&lt; 2^{64})</td>
<td>512</td>
<td>256</td>
<td>14</td>
<td>16</td>
<td>8, 7</td>
</tr>
<tr>
<td>BLAKE-284</td>
<td>64</td>
<td>(&lt; 2^{128})</td>
<td>1024</td>
<td>384</td>
<td>16</td>
<td>32</td>
<td>25, 16, 11</td>
</tr>
<tr>
<td>BLAKE-512</td>
<td>64</td>
<td>(&lt; 2^{128})</td>
<td>1024</td>
<td>256</td>
<td>16</td>
<td>32</td>
<td>25, 16, 11</td>
</tr>
</tbody>
</table>
\begin{table}[h]
\centering
\caption{Permutations of \{0, \ldots, 15\} used by the BLAKE functions (reprinted from [1]).}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\hline
\sigma_0(i) & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\sigma_1(i) & 14 & 10 & 4 & 8 & 9 & 15 & 13 & 6 & 1 & 12 & 0 & 2 & 11 & 7 & 5 & 3 \\
\hline
\sigma_2(i) & 11 & 8 & 12 & 0 & 5 & 2 & 15 & 13 & 10 & 14 & 3 & 6 & 7 & 1 & 9 & 4 \\
\hline
\sigma_3(i) & 7 & 9 & 3 & 1 & 13 & 12 & 11 & 14 & 2 & 6 & 5 & 10 & 4 & 0 & 15 & 8 \\
\hline
\sigma_4(i) & 9 & 0 & 5 & 7 & 2 & 4 & 10 & 15 & 14 & 1 & 11 & 12 & 6 & 8 & 3 & 13 \\
\hline
\sigma_5(i) & 2 & 12 & 6 & 10 & 0 & 11 & 8 & 3 & 4 & 13 & 7 & 5 & 15 & 14 & 1 & 9 \\
\hline
\sigma_6(i) & 12 & 5 & 1 & 15 & 14 & 13 & 4 & 10 & 0 & 7 & 6 & 3 & 9 & 2 & 8 & 11 \\
\hline
\sigma_7(i) & 13 & 11 & 7 & 14 & 12 & 1 & 3 & 9 & 5 & 0 & 15 & 4 & 8 & 6 & 2 & 10 \\
\hline
\sigma_8(i) & 6 & 15 & 14 & 9 & 11 & 3 & 0 & 8 & 12 & 2 & 13 & 7 & 1 & 4 & 10 & 5 \\
\hline
\sigma_9(i) & 10 & 2 & 8 & 4 & 7 & 6 & 1 & 5 & 15 & 11 & 9 & 14 & 3 & 12 & 13 & 0 \\
\hline
\end{tabular}
\end{table}

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. \( a \leftarrow a \oplus b \);
2. \( a \leftarrow a \oplus (m_{\sigma_i(2i)} \oplus c_{\sigma_i(2i+1)}) \);
3. \( d \leftarrow (d \oplus a) \gg 60 \);
4. \( c \leftarrow c \oplus d \);
5. \( b \leftarrow (b \oplus c) \gg 2 \delta_1 \);
6. \( a \leftarrow a \oplus b \);
7. \( a \leftarrow a \oplus (m_{\sigma_i(2i+1)} \oplus c_{\sigma_i(2i)}) \);
8. \( d \leftarrow (d \oplus a) \gg 2 \delta_2 \);
9. \( c \leftarrow c \oplus d \);
10. \( b \leftarrow (b \oplus c) \gg 2 \delta_3 \);

\section{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].

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{General architecture of our coprocessors.}
\end{figure}

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, R1 denotes a 64-bit register. Figure 5 illustrates our scheduling of the two mixing functions Mix\(_{4,0}\) and 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\(_{4,1}\) \(\ll\) 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 R3 \(\leftarrow\) R1 \(\oplus\) R2 and obtain f\(_{4,0}\).
- R3 and R6 contain f\(_{4,0}\) and e\(_{4,1}\) \(\ll\) R\(_{4,0}\), respectively. The instruction R3 \(\leftarrow\) R3 \(\oplus\) R6 allows us to compute f\(_{4,1}\).

It is well-known that \(a \oplus b = (a \lor b) \oplus (a \land b)\) and \(a \oplus b = (a \lor b) \oplus (a \land b)\) [16]. Thus, Equation (1) can be rewritten as follows:

\[
R3 \leftarrow (a \lor b) \oplus ((a \land b) \oplus c_{\text{tr}1}) \oplus c_{\text{tr}10}.
\] (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-LUT\(_2\) primitives and dedicated carry logic.

![Figure 5. Computation of Mix\(_{4,0}\) and Mix\(_{4,1}\) (Threefish-256).](image)

![Figure 6. Arithmetic and logic unit for Threefish encryption.](image)

![Figure 7. Computation of R3 \(\leftarrow\) R1 \(\oplus\) R2 or R3 \(\leftarrow\) R3 \(\oplus\) R6 on a Virtex-6 device.](image)
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 $R3 \leftarrow R1 \oplus R2$. 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 $t_2 = (t_0 \oplus t_0) \oplus (t_1 \ll 0)$, it suffices to execute the following instructions:

\[
\begin{align*}
R4 &= t_1 \ll 0, \\
R1 &= t_0, \\
R2 &= 0, \\
R5 &= R4 \ll 0, \\
R3 &= R1 \oplus R2, \\
R6 &= R5 \ll 0, \\
R3 &= R3 \oplus R6.
\end{align*}
\]

This approach assumes that we can read simultaneously two values from the register file. Thanks to the multiplexer controlled by ctrl7, we can load data from port A or port B into register R2 (Figure 6). A similar strategy allows us to compute $k_{N_r}$.

The implementation of the key injection is more straightforward. Note that the multiplexers controlled by ctrl6 and ctrl7 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 $p_2 \oplus k_{0,2} = p_2 \oplus k_{2} \oplus t_1$ and is computed as follows:

\[
\begin{align*}
R1 &= k_2, \\
R2 &= t_1, \\
R3 &= R1 \oplus R2, \\
R1 &= R3, \\
R2 &= p_2, \\
R3 &= R1 \oplus R2.
\end{align*}
\]

Figures 8 and 9 describe how we schedule the instructions of Threefish encryption. It suffices to modify line 21 of Algorithm 3 as follows:

\[
\begin{align*}
e_{N_i, r, i} &= v_{N_i, r, i} \oplus k_{N_i, a, i}; \\
c_i &= e_{N_i, r, i} \oplus p_i.
\end{align*}
\]

The only difference between this operation and the mixing function $MIX_{a, j}$ 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 $2^{w}$. Our modified ALU is therefore able to perform a new operation: $R3 \leftarrow R1 \oplus R2$. 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. $c_{4,0}$ on Figure 3) or the rotation unit (e.g. $c_{3,1}$ on Figure 2). The multiplexer controlled by ctrl12 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.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig10.png}
\caption{3- or 1-stage FIFO (encryption or decryption).}
\end{figure}

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 $R5 \leftarrow R5 \oplus RF_A$.
2. Compute $R5 \leftarrow R5 \oplus (RF_A \oplus RF_B)$.
3. Save the content of R5 in the register file and compute $R5 \leftarrow (R5 \oplus RF_A) \gg \delta_j$.

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

-
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 \( \omega \)-bit words described in Section VI-A. Our operator computes:

\[
R_3 = \begin{cases} 
R_1 \oplus R_2 & \text{when } \text{ctrl}_2 = 0, \\
R_1 \oplus R_2 & \text{otherwise}
\end{cases}
\]

\[=(R_1 \lor R_2) \oplus ((R_1 \land R_2) \oplus \text{ctrl}_2) \oplus \text{ctrl}_2,\]

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 \( \text{ctrl}_0 \) selects either \( RF_A \) or \( RF_B \). This allows us to compute \( m_{\sigma_i}(2i) \oplus c_{\sigma_i}(2i+1) \) and \( m_{\sigma_i}(2i+1) \oplus c_{\sigma_i}(2i) \) 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 \( \text{ctrl}_1 \) 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:

\[R_3 \gg \delta_i = (R_3 \gg (\delta_i - \delta_3)) \gg \delta_3,\]

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 \( \text{ctrl}_{1:3} \) 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 \( \omega \)-bit register after the first stage of the rotation. Since the latter involves \( \omega \) LUTs, there is no hardware overhead on a Virtex-6 device.
We check for instance that the ALU outputs the new value of \( G \) as given by

\[
\begin{align*}
\text{rename} & \Rightarrow \text{permute} \\
\text{high} & \Rightarrow \text{low} \\
\text{low} & \Rightarrow \text{high}
\end{align*}
\]

A flip-flop and an \( n \)-bit register can be computed in two clock cycles. The mode of operation is selected according to an additional control bit \( c_{\text{low}} \), the latter being provided by the user. The ALU includes two 32-bit adders. Let \( a_{\text{low}}, b_{\text{low}}, a_{\text{high}}, b_{\text{high}}, s_{\text{low}}, \) and \( s_{\text{high}} \) denote unsigned 32-bit integers. When the user chooses BLAKE-224 or BLAKE-256 (\( c_{\text{low}} = 0 \), two messages are processed in parallel and the ALU performs two 32-bit additions:

\[
\begin{align*}
\text{low} & \leftarrow a_{\text{low}} + b_{\text{low}} + c_{\text{low}}, \\
\text{high} & \leftarrow a_{\text{high}} + b_{\text{high}} + c_{\text{low}}.
\end{align*}
\]

When the coprocessor executes BLAKE-384 or BLAKE-512
From port A To R
From port B
From R
From port A
To port B From R
From port A
To port B
To R

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-$n$.

(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:

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

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 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$_6$ is equal to one, ctrl$_{4:3}$ encodes the index $i$ of the rotation distance $\delta_i$ (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 $R_{A} \oplus R_{B}$ 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
Pipeline stages

\[ \begin{align*}
\text{1st step of } G_5: & \quad \tau + 1 \\
& \quad \tau + 2 \\
& \quad \tau + 3 \\
& \quad \tau + 4 \\
& \quad \tau + 5 \\
& \quad \tau + 6
\end{align*} \]

Time [clock cycles]

\[ \begin{align*}
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_3 \leftarrow R_1 & \quad \llcorner R_2 \\
& \quad R_1 \leftarrow v_1 \\
& \quad R_2 \leftarrow R_5 \leftarrow v_6 \\
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_5 \leftarrow v_11 \leftarrow R_4 \\
& \quad R_3 \leftarrow R_1 \llcorner R_2 \\
& \quad R_5 \leftarrow v_10 \leftarrow R_4 \\
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_5 \leftarrow v_9 \leftarrow R_4
\end{align*} \]

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

Pipeline stages

\[ \begin{align*}
\text{1st step of } G_5: & \quad \tau + 1 \\
& \quad \tau + 2 \\
& \quad \tau + 3 \\
& \quad \tau + 4 \\
& \quad \tau + 5 \\
& \quad \tau + 6
\end{align*} \]

Time [clock cycles]

\[ \begin{align*}
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_3 \leftarrow R_1 \llcorner R_2 \\
& \quad R_1 \leftarrow v_1 \\
& \quad R_2 \leftarrow R_5 \leftarrow v_6 \\
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_5 \leftarrow v_{11} \leftarrow R_4 \\
& \quad R_3 \leftarrow R_1 \llcorner R_2 \\
& \quad R_5 \leftarrow v_{10} \leftarrow R_4 \\
& \quad R_4 \leftarrow R_3 \gg 0 \\
& \quad R_5 \leftarrow v_9 \leftarrow R_4
\end{align*} \]

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

Table V

<table>
<thead>
<tr>
<th>ctrl_{5:3}</th>
<th>Rot. dist.</th>
<th>BLAKE-224/256</th>
<th>BLAKE-384/512</th>
</tr>
</thead>
<tbody>
<tr>
<td>(000)_{2}</td>
<td>0</td>
<td>R_6 \leftarrow r_3 (common datapath)</td>
<td>R_6 \leftarrow r_3 \gg 7</td>
</tr>
<tr>
<td>(100)_{2}</td>
<td>\delta_0</td>
<td>R_6 \leftarrow r_3 \gg 7</td>
<td>R_6 \leftarrow r_3 \gg 11</td>
</tr>
<tr>
<td>(101)_{2}</td>
<td>\delta_1</td>
<td>R_6 \leftarrow r_3 \gg 8</td>
<td>R_6 \leftarrow r_3 \gg 16</td>
</tr>
<tr>
<td>(110)_{2}</td>
<td>\delta_2</td>
<td>R_6 \leftarrow r_3 \gg 12</td>
<td>R_6 \leftarrow r_3 \gg 25</td>
</tr>
<tr>
<td>(111)_{2}</td>
<td>\delta_3</td>
<td>R_6 \leftarrow r_3 \gg 16</td>
<td>R_6 \leftarrow r_3 \gg 32</td>
</tr>
</tbody>
</table>

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:

\[ \begin{align*}
R_1 & \leftarrow 1 \\
R_3 & \leftarrow R_1 \llcorner R_2 \\
R_4 & \leftarrow R_3 \\
R_5 & \leftarrow R_4 \\
R_1 & \leftarrow m_{12} \\
R_2 & \leftarrow 0,
\end{align*} \]

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_3 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:

\[ \begin{align*}
(F, R_3) & \leftarrow R_1 \llcorner R_2 \quad R_1 \leftarrow m_{13} \\
R_2 & \leftarrow 0, \\
R_3 & \leftarrow R_1 \llcorner R_2 \llcorner F \\
R_4 & \leftarrow R_3 \\
R_5 & \leftarrow R_4 \\
\text{Register file} & \leftarrow R_4 \leftarrow (m_{12}) \\
R_4 & \leftarrow R_3, \\
\text{Register file} & \leftarrow R_4 \leftarrow (m_{13}).
\end{align*} \]

Three pipeline configurations are again available. The second one needs specific attention: since the adder is pipelined, the computation of \( m_{12} + 1 \) 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
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. The

1It 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 V) and by generating the addresses of \( m_{r,2i} \) and \( c_{r,2i+1} \) 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAKE-224/256</td>
<td>1184</td>
</tr>
<tr>
<td>BLAKE-384/512</td>
<td>1344</td>
</tr>
<tr>
<td>8-round ChaCha</td>
<td>311</td>
</tr>
<tr>
<td>12-round ChaCha</td>
<td>439</td>
</tr>
<tr>
<td>20-round ChaCha</td>
<td>695</td>
</tr>
</tbody>
</table>
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: \( \text{ctrl}_7 \) allows her to select between BLAKE and ChaCha, and \( \text{ctrl}_6 \) specifies the configuration of the datapath (2 \( \times \) 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 \( \text{ctrl}_8 \). 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.
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 \( \text{ctrl}_0 \) and \( \text{ctrl}_1 \) 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 (\( C_{d,i} \), 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 3 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 \( N_e = 80 \)). 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:

\[
\text{throughput} = \frac{8 \cdot N_h \cdot k \cdot f}{(k + 1) \cdot \text{latency of Threefish with UBI} + 5 \cdot k},
\]

where \( f \) denotes the clock frequency.
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-512, JH-512, SHA-2-512, 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 (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).

\[ \text{Throughput [Mbit/s]} \]

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.

![Figure 20. Generation of the compressed instruction memory.](image)

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


### Table X

**COMPACT IMPLEMENTATIONS OF BLAKE AND SKEIN ON VIRTEX-5 AND VIRTEX-6 FPGAS.** The throughput is computed for a one-block message.

<table>
<thead>
<tr>
<th>Supported algorithm(s)</th>
<th>FPGA</th>
<th>Area [slices]</th>
<th>30k memory blocks</th>
<th>Frequency [MHz]</th>
<th>Throughput [Mbits/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latif et al. [17]†</td>
<td>xc6vlx75t-1</td>
<td>207</td>
<td>1</td>
<td>166</td>
<td>17</td>
</tr>
<tr>
<td>Jungk [18]‡</td>
<td>xc6vlx75t-1</td>
<td>300</td>
<td>1</td>
<td>91</td>
<td>412</td>
</tr>
<tr>
<td>Jungk [19]‡</td>
<td>xc6vlx75t-1</td>
<td>404</td>
<td>1</td>
<td>85</td>
<td>823</td>
</tr>
<tr>
<td>Kaps et al. [20]</td>
<td>xc6vlx75t-1</td>
<td>512</td>
<td>1</td>
<td>39</td>
<td>468</td>
</tr>
<tr>
<td>Auamasson et al. [1]‡</td>
<td>xc6vlx75t-1</td>
<td>512</td>
<td>1</td>
<td>39</td>
<td>468</td>
</tr>
<tr>
<td>Kercikho et al. [1]‡</td>
<td>xc6vlx75t-1</td>
<td>512</td>
<td>1</td>
<td>39</td>
<td>468</td>
</tr>
</tbody>
</table>

† 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).**

<table>
<thead>
<tr>
<th>FPGA</th>
<th>Area [slices]</th>
<th>Frequency [MHz]</th>
<th>Throughput [Mbits/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES &amp; ECHO [13]</td>
<td>155</td>
<td>303</td>
<td>219 156 161 92 45</td>
</tr>
<tr>
<td>AES &amp; Grøssl [13]</td>
<td>160</td>
<td>303</td>
<td>217 184 150 92 40</td>
</tr>
</tbody>
</table>


