A Low-Area Unified Hardware Architecture for the AES and the Cryptographic Hash Function Grøstl

Nuray At*, Jean-Luc Beuchat†, Eiji Okamoto†, İsmail San*, and Teppei Yamazaki†
*Department of Electrical and Electronics Engineering, Anadolu University, Eskişehir, Turkey
†Faculty of Engineering, Information and Systems,
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan

Abstract—**This article describes the design of an 8-bit coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function Grøstl on several Xilinx FPGAs. Our Arithmetic and Logic Unit performs a single instruction that allows for implementing AES encryption, AES decryption, AES key expansion, and Grøstl at all levels of security. Thanks to a careful organization of AES and Grøstl internal states in the register file, we manage to generate all read and write addresses by means of a modulo-128 counter and a modulo-256 counter. A fully autonomous implementation of Grøstl and AES on a Virtex-6 FPGA requires 169 slices and a single 36k memory block, and achieves a competitive throughput. Assuming that the security guarantees of Grøstl are at least as good as the ones of the other SHA-3 finalists, our results show that Grøstl is the best candidate for low-area cryptographic coprocessors.**

I. Introduction

In response to the successful cryptanalysis of the MD4, MD5, and SHA-0 hash functions, the National Institute of Standards and Technology (NIST) has decided to develop a new algorithm to augment the Secure Hash Standard (FIPS 180-2). The public SHA-3 competition was announced in November 2007. After two rounds of internal reviews and feedback from the cryptographic community, the NIST selected 5 candidates out of 64 to advance to the final round. If security remains the main criterion, computational efficiency, memory requirement, flexibility, and simplicity are also of great significance.

Since the SHA-3 finalist Grøstl [13] is strongly inspired by the Advanced Encryption Standard (AES) [11], it is tempting to design a compact unified hardware architecture which supports both algorithms. Such an implementation is valuable for constrained environments, where some security protocols mainly rely on cryptographic hash functions (see for instance [29]). 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" [19]. In order to have a deeper understanding of the computational efficiency of several SHA-3 candidates (resource sharing, memory access scheme, scheduling, etc.), we already designed five low-area coprocessors [1], [7]–[9], [24]. In particular, we proposed a compact unified architecture for the SHA-3 round 2 candidate ECHO [4] and the AES. The main originality of our work was to describe the AES by means of a single instruction [9]. Since ECHO is built around the round function of the AES, it is rather straightforward to design a unified Arithmetic and Logic Unit (ALU) for both algorithms. In the conclusion of our article, we stated that our design strategy could be applied to Grøstl and AES. However, even if Grøstl borrows the the S-box of the AES, the construction of the diffusion layers is only based on the design philosophy of the AES. Contrary to ECHO, Grøstl can not be implemented with the AES instruction set of Intel Westmere processors [5], and it seems much more challenging to build a compact unified coprocessor. We bring a solution to this problem in this work.

The rest of the article is organized as follows: Section II provides the reader with a short description of the AES and a summary of the design strategy we proposed in [9]. In Section III, we give an alternative description of Grøstl showing how to implement the algorithm with our single instruction set architecture. Then, we describe our unified 8-bit coprocessor, focusing on its control unit (Section IV). We discuss our implementation results on several Xilinx Field-Programmable Gate Arrays (FPGAs) in Section V and conclude in Section VI.

II. The Advanced Encryption Standard

The round transformation of the AES operates on a 128-bit intermediate result, called state. The state is internally represented as a $N_l \times N_c$ array of bytes A, where N_l and N_c denotes the number of lines and columns, respectively. In the case of the AES, $N_l = N_c = 4$. Each byte $a_{i,j}$, $0 \leq i, j \leq 3$, is considered as an element of $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/(m(x))$, where the irreducible polynomial is given by $m(x) = x^8 + x^4 + x^3 + x + 1$. In the following, we encode an element of $\mathbb{F}_{2^8}$ by two hexadecimal digits: for instance, 95 is equivalent to $x^7 + x^4 + x^2 + 1$ in the polynomial basis representation. We denote the jth column of A by A_j. The number of rounds N_r as well as the number of 32-bit blocks in the cipher key N_k of the AES depend on the desired security level (Table I).

The AES involves four byte-oriented transformations and their inverses for encryption and decryption, respectively [11]:

- The SubBytes step updates each byte of the state using an 8-bit S-box, denoted by $\mathrm{S_{RD}}$. The inverse transformation is called InvSubBytes and denoted by $\mathrm{S_{RD}^{-1}}$.
- The ShiftRows step simply consists of a cyclical left shift of the three bottom rows of the state by 1, 2, and 3 bytes, respectively.

Table I
BLOCK LENGTH, KEY LENGTH, NUMBER OF 32-BIT BLOCKS OF THE KEY (N_k), AND NUMBER OF ROUNDS (N_r) OF AES-128, AES-192, AND AES-256.

Algorithm	Block length [bits]	Key length [bits]	N_k	N_r
AES-128	128	128	4	10
AES-192	128	192	6	12
AES-256	128	256	8	14

- The MixColumns step is a permutation operating on the AES state column by column. Each column of the AES state is considered as a polynomial over $\mathbb{F}_{2^8}$, and is multiplied modulo $y^4 + \texttt{01}$ by the constant polynomial $c(y) = \texttt{03}\cdot y^3 + \texttt{01}\cdot y^2 + \texttt{01}\cdot y + \texttt{02}$ [11]. This operation is performed by multiplying each column of the state A by a circulant matrix $\mathcal{M}_E = \mathrm{circ}(\texttt{02}, \texttt{03}, \texttt{01}, \texttt{01})$. During the inverse operation, called InvMixColumns, each column of the state is multiplied by $\mathcal{M}_D = \mathrm{circ}(\texttt{0E}, \texttt{0B}, \texttt{0D}, \texttt{09})$.
- The AddRoundKey step combines the state A with a 128-bit round key. Let r denote the round index. Each byte $k_{i,4r+j}$ of the round key and its corresponding byte $a_{i,j}$ are added in $\mathbb{F}_{2^8}$ by a simple bitwise XOR operation. AddRoundKey is therefore its own inverse.

After an initial AddRoundKey step, an AES encryption involves $N_r - 1$ repetitions of a round composed of the four byte-oriented transformations described above. Eventually, a final encryption round, in which the MixColumns step is omitted, produces the ciphertext (Figure 1). We consider here the equivalent decryption algorithm described in [11, Section 3.7.3]. Its main advantage over the straightforward decryption process is that encryption and decryption rounds share the same datapath (Figure 1). Nevertheless, the round keys are introduced in reverse order for decryption. A key expansion algorithm allows one to derive the round keys involved in the AddRoundKey steps from the cipher key. Let us consider an array consisting of 4 rows and $4 \cdot (N_r + 1)$ columns. The cipher key is copied in the first N_k columns of the array, and the next columns are defined recursively (see [11, Section 3.6] for details).

If an AES coprocessor is built around an 8-bit datapath, the ShiftRows and InvShiftRows steps are implemented by accordingly addressing the register file organized into bytes. As a result, these operations are virtually for free and do not require dedicated hardware in the ALU. It is then possible to describe encryption, decryption, and key expansion with a single instruction [9]:

$$R_k \leftarrow \mathcal{A} \cdot g(R_i) \oplus \mathcal{B} \cdot R_j, \qquad (1)$$

where

- R_i, R_j, and R_k are vectors of N_l bytes.
- $\mathcal{A}$ and $\mathcal{B}$ are matrices of $N_l \times N_l$ bytes. Let us define the identity matrix $\mathcal{I}_{\mathrm{AES}} = \mathrm{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00})$ and the permutation matrix $\mathcal{P}_{\mathrm{AES}} = \mathrm{circ}(\texttt{00}, \texttt{01}, \texttt{00}, \texttt{00})$. The latter matrix is essential to the key schedule. We showed in [9] that $\mathcal{A}$ can be any of the four matrices we introduced so far, whereas $\mathcal{B}$ is either $\mathcal{M}_D$ or the identity matrix $\mathcal{I}_{\mathrm{AES}}$.
- g is a function applied to each byte of R_i. In addition to S_{RD} and $\mathrm{S}_{\mathrm{RD}}^{-1}$, we need the identity function to implement the key expansion and the first AddRoundKey step of AES encryption or decryption. The first instruction of the decryption process (Figure 1) is for instance

$$A_0 \leftarrow \mathcal{I}_{\mathrm{AES}} \cdot A_0 \oplus \mathcal{I}_{\mathrm{AES}} \cdot K_{4N_r}.$$

Since the ALU processes the operands byte by byte, the computation of Equation (1) involves at least N_l cycles. In order to achieve a high clock frequency on FPGA, it is however necessary to make the pipeline deeper. The main challenge is to schedule the instructions to avoid pipeline bubbles as much as possible.

III. THE HASH FUNCTION GRØSTL

Grøstl is a family of cryptographic hash functions able to compute message digests from 8 to 512 bits [13]. We denote by Grøstl-n the algorithm with a n-bit output and focus on the digest sizes specified for the SHA-3 competition (224, 256, 384, and 512 bits). The original message is padded, split into t message blocks of ℓ bits, and organized as an array of $N_l \times N_c$ bytes, where ℓ and N_c depend on the desired level of security (Table II). The number of lines N_l is always equal to 8. In this work, we assume that our coprocessor is provided with a padded message and refer the reader to [13, Section 3.6] for a description of the padding algorithm. A hardware wrapper interface for Grøstl (and several other hash functions) comprising communication and padding is for instance described in [3]. Starting from an initial chaining input $H^{(0)} = IV_n$, the message blocks $M^{(1)}, \ldots, M^{(t)}$ are processed by a compression function f as:

$$H^{(i)} \leftarrow f(H^{(i-1)}, M^{(i)}),$$

where $1 \leq i \leq t$. Figure 2 describes the datapath of the compression function f that consists of a key schedule, and two permutations P_ℓ and Q_ℓ operating on two $8 \times N_c$ array of bytes P and Q. Each byte is considered as an element of $\mathbb{F}_{2^8} \cong \mathbb{F}_2[x]/(m(x))$, where $m(x) = x^8 + x^4 + x^3 + x + 1$ is the irreducible polynomial of the AES.

Table II
BLOCK LENGTH, NUMBER OF COLUMN OF THE INTERNAL STATE, AND NUMBER OF ROUNDS OF GRØSTL-n.

Digest size n [bits]	Block length ℓ [bits]	# columns N_c	# rounds N_r
8 to 256	512	8	10
264 to 512	1024	16	14

Each permutation involves a first key injection followed by N_r rounds consisting of four byte-oriented transformations similar to those of the AES (N_r depends on the digest size and is defined in Table II):

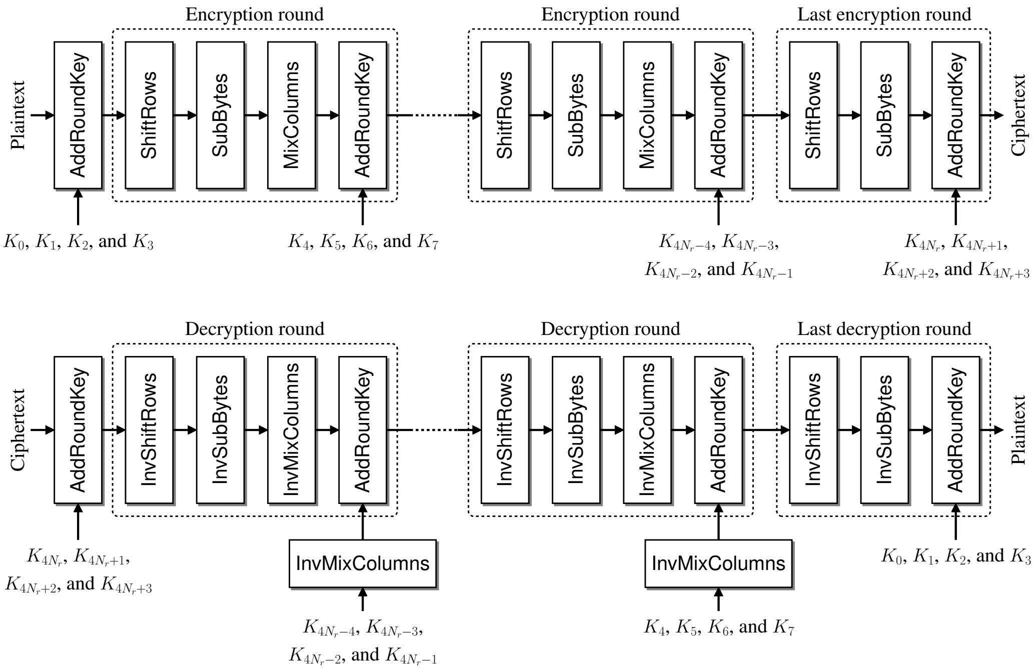


Figure 1. AES encryption and decryption flowcharts (reprinted from [9]).

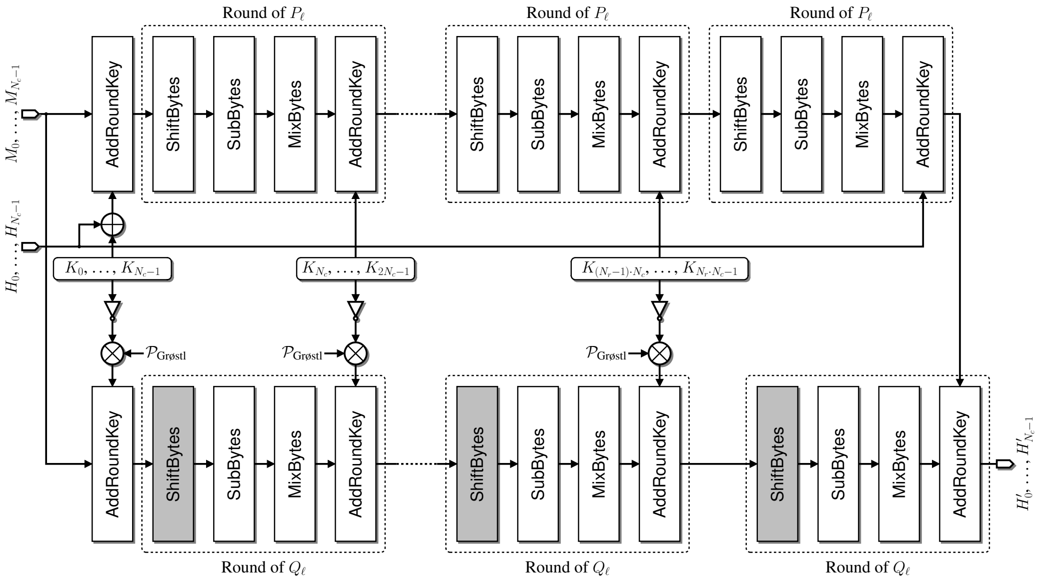


Figure 2. Flowchart of the compression function f of Grøstl.

- The ShiftBytes step cyclically rotates the ith row of P and Q to the left by $\sigma_P(i)$ and $\sigma_Q(i)$ bytes, respectively. Let $T = \mathsf{ShiftBytes}(P)$ and $U = \mathsf{ShiftBytes}(Q)$. We have:
$$t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c},$$
$$u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c},$$
where $0 \leq i \leq 7$ and $0 \leq j \leq N_c - 1$. The offsets $\sigma_P(i)$ and $\sigma_Q(i)$ depend on the row index i and the number of columns N_c, and are defined in Table III.
- The SubBytes step updates each byte of P and Q using the AES S-box, denoted by S_{RD}.

Table III
Offsets of the ShiftBytes transformation according to the row index i and the number of columns N_c.

i	0	1	2	3	4	5	6	7
$\sigma_P(i)$	0	1	2	3	4	5	6	$\frac{N_c}{2}+3$
$\sigma_Q(i)$	1	3	5	$\frac{N_c}{2}+3$	0	2	4	6

- The MixBytes step is performed by multiplying each column of P and Q by the circulant matrix $\mathcal{M}_{\text{Grøstl}} = \text{circ}(\texttt{02}, \texttt{02}, \texttt{03}, \texttt{04}, \texttt{05}, \texttt{03}, \texttt{05}, \texttt{07})$.

- The AddRoundKey step combines P and Q with two ℓ-bit round keys. The key expansion requires $N_r \cdot N_c$ 64-bit constants that can be computed on-the-fly according to Algorithm 1. Since the round counter r and the loop index j are 4-bit numbers ($N_r \leq 14$ and $N_c - 1 \leq 15$), each $k_{i,r\cdot N_c+j}$ can be seen as an element of $\mathbb{F}_{2^8}$.

Algorithm 1 Computation of the round constants.

```
Input: $N_r$ (number of rounds of the permutation $P$) and $N_c$
    (number of columns of the internal state).
Output: The $N_r \cdot N_c$ round constants required to compute the
    permutation $P$.
for $r \leftarrow 0$ to $N_r - 1$ do
  for $j \leftarrow 0$ to $N_c - 1$ do
    $k_{0,r\cdot N_c+j} \leftarrow j \parallel r$;
    for $i \leftarrow 1$ to $7$ do
      $k_{i,r\cdot N_c+j} \leftarrow \texttt{00}$;
    end for
  end for
end for
Return $K_0, K_1, \ldots, K_{N_r\cdot N_c-1}$;
```

Besides the round constants, the key expansion involves the chaining input H and a permutation matrix $\mathcal{P}_{\text{Grøstl}} = \text{circ}(\texttt{00}, \texttt{01}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00})$ (Figure 2). Finally, note that the output of P_ℓ serves as the key of the last round of Q_ℓ.

Algorithm 2 describes how we implement Grøstl using the instruction defined by Equation (1). The ShiftBytes step (*i.e.* computation of T_j and U_j on lines 10, 17, 25, and 32) is performed by accordingly addressing the register file organized into bytes. As a result, these operations are virtually for free and do not require dedicated hardware in the ALU. Since the ShiftBytes transformations performs cyclical left shifts of the rows of the state, we have to be careful not to overwrite bytes that are still involved in the forthcoming MixBytes steps ($p_{1,0}$ is for instance needed to update the eighth column of P, and should not be overwritten when updating the first column). We solve this problem by introducing two $8 \times N_c$ arrays of bytes P' and Q' to store intermediate results. The coprocessor we will describe in Section IV embeds a number of pipeline stages, and several clock cycles are required to process a byte of the state. In order to avoid data dependency issues between two consecutive rounds of a given permutation, we interleave the computation of P_ℓ and Q_ℓ.

After the last message block has been processed, an output transformation based on P_ℓ generates the n-bit digest D (Algorithm 3). The function $\text{trunc}_n(P')$ (line 19) discards all but the n trailing bits of P'. Table IV provides the reader with a summary of the instructions involved in the compression function and the output transformation.

Algorithm 2 Compression function f of Grøstl.

```
Input: A $\ell$-bit message block $M$ and a chaining value $H$.
Output: A new chaining value $H'$.
 1. for $j \leftarrow 0$ to $N_c - 1$ do
 2.   $P_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot M_j \oplus \mathcal{I}_{\text{Grøstl}} \cdot (K_j \oplus H_j)$;
 3. end for
 4. for $j \leftarrow 0$ to $N_c - 1$ do
 5.   $Q_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot M_j \oplus \mathcal{P}_{\text{Grøstl}} \cdot \neg K_j$;
 6. end for
 7. for $r \leftarrow 1$ to $N_r - 1$ do
 8.   for $j \leftarrow 0$ to $N_c - 1$ do
 9.     for $i \leftarrow 0$ to $7$ do
10.       $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
11.     end for
12.     $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_{r\cdot N_c+j}$;
13.   end for
14.   $P \leftarrow P'$;
15.   for $j \leftarrow 0$ to $N_c - 1$ do
16.     for $i \leftarrow 0$ to $7$ do
17.       $u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c}$;
18.     end for
19.     $Q'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(U_j) \oplus \mathcal{P}_{\text{Grøstl}} \cdot \neg K_{r\cdot N_c+j}$;
20.   end for
21.   $Q \leftarrow Q'$;
22. end for
23. for $j \leftarrow 0$ to $N_c - 1$ do
24.   for $i \leftarrow 0$ to $7$ do
25.     $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
26.   end for
27.   $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot H_j$;
28. end for
29. $P \leftarrow P'$;
30. for $j \leftarrow 0$ to $N_c - 1$ do
31.   for $i \leftarrow 0$ to $7$ do
32.     $u_{i,j} \leftarrow q_{i,(j+\sigma_Q(i)) \bmod N_c}$;
33.   end for
34.   $H'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(U_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot P_j$;
35. end for
36. Return $H'_0, \ldots, H'_{N_c-1}$;
```

IV. A Compact Unified Coprocessor for the AES and the Grøstl Family of Hash Functions

Figure 3 describes how we modified the 8-bit coprocessor proposed by Beuchat *et al.* [9] in order to share the same datapath between Grøstl and the AES. The architecture is built around an 8-bit datapath and consists of three main components:

- a register file and a key memory implemented by means of a single dual-ported memory block; address bits and write enable signals are denoted by $a_{39:0}$ and $we_{3:0}$, respectively;
- a control unit responsible for the address generation and the selection of the parameters $\mathcal{A}$, $\mathcal{B}$, and g (Table IV);
- an ALU implementing the instruction defined by Equation (1), the key expansion mechanism of the AES, and the computation of the round constants of Grøstl.

Table IV
IMPLEMENTATION OF GRØSTL WITH A SINGLE INSTRUCTION.

Operation	R_k	$\mathcal{A}$	g	R_i	$\mathcal{B}$	R_j
Algorithm 2, line 2	P_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	M_j	$\mathcal{I}_{\text{Grøstl}}$	$K_j \oplus H_j$
Algorithm 2, line 5	Q_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	M_j	$\mathcal{I}_{\text{Grøstl}}$	$\neg K_j$
Algorithm 2, line 12 and Algorithm 3, line 9	P'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	T_j	$\mathcal{I}_{\text{Grøstl}}$	$K_{r \cdot N_c + j}$
Algorithm 2, line 19	Q'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	U_j	$\mathcal{P}_{\text{Grøstl}}$	$\neg K_{r \cdot N_c + j}$
Algorithm 2, line 27 and Algorithm 3, line 17	P'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	T_j	$\mathcal{I}_{\text{Grøstl}}$	H_j
Algorithm 2, line 34	Q'_j	$\mathcal{M}_{\text{Grøstl}}$	S_{RD}	U_j	$\mathcal{I}_{\text{Grøstl}}$	P_j
Algorithm 3, line 2	P_j	$\mathcal{I}_{\text{Grøstl}}$	Identity	H_j	$\mathcal{I}_{\text{Grøstl}}$	K_j

Algorithm 3 Output transformation.

```
Input: An intermediate hash value $H$
Output: A $n$-bit digest $D$
1.  for $j \leftarrow 0$ to $N_c - 1$ do
2.      $P_j \leftarrow \mathcal{I}_{\text{Grøstl}} \cdot H_j \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_j$;
3.  end for
4.  for $r \leftarrow 1$ to $N_r - 1$ do
5.      for $j \leftarrow 0$ to $N_c - 1$ do
6.          for $i \leftarrow 0$ to $7$ do
7.              $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
8.          end for
9.          $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot K_{r \cdot N_c + j}$;
10.     end for
11.     $P \leftarrow P'$;
12. end for
13. for $j \leftarrow 0$ to $N_c - 1$ do
14.     for $i \leftarrow 0$ to $7$ do
15.         $t_{i,j} \leftarrow p_{i,(j+\sigma_P(i)) \bmod N_c}$;
16.     end for
17.     $P'_j \leftarrow \mathcal{M}_{\text{Grøstl}} \cdot \text{S}_{\text{RD}}(T_j) \oplus \mathcal{I}_{\text{Grøstl}} \cdot H_j$;
18. end for
19. $D \leftarrow \text{trunc}_n(P')$;
20. Return $D$;
```

A. Memory Organization

Since we consider an 8-bit datapath, the memory of our coprocessor is organized into bytes. We will show below that ten address bits are needed to access message blocks and intermediate data, thus allowing us to implement the register file and the key memory by means of a single Virtex-6 block RAM configured as two independent 18 Kb RAMs (Figure 3).

Recall that each variable of Grøstl-n is an array of $8 \times N_c$ bytes and that the compression function of Grøstl requires six variables: a chaining value H, a message block M, P, P', Q, and Q' (Algorithm 2). Since $N_c \leq 16$ (Table II), we define six chunks of 128 bytes in the register file (Figure 4), and address each byte with 10 bits:

- the three most significant bits select the desired variable;
- the next four bits encode the column index j;
- the three least significant bits define the row index i.

The addresses of $h_{i,j}$ and $p_{i,j}$ are for instance given by $8j + i$ and $256 + 8j + i$, respectively. The key memory stores:

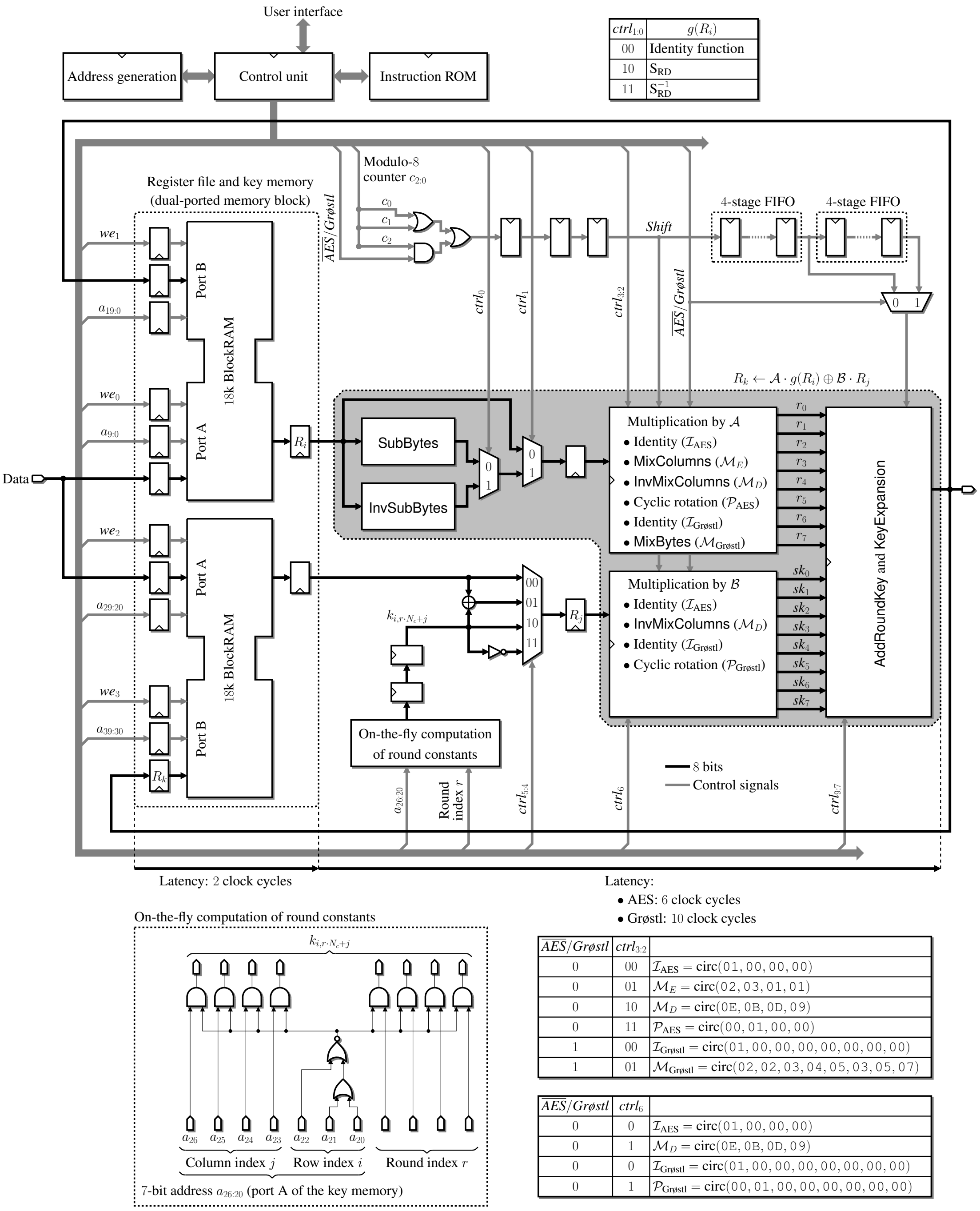
- a copy of the chaining value H required to implement the key schedule (lines 2 and 27 of Algorithm 2, and line 17 of Algorithm 3);
- the initial chaining values IV_{224}, IV_{256}, IV_{384}, and IV_{512};
- a copy of P needed to perform the AddRoundKey step of the last round of Q_ℓ (line 34 of Algorithm 2).

We keep the memory organization proposed in [9] for the AES. Note that the execution of Grøstl-n does not overwrite the round keys of the AES. As long as the AES master key is not modified, it is therefore possible to switch between the hash function and the block cipher with no need for the AES KeyExpansion step. In the following, we show that our careful organization of the data in the register file and in the key memory allows one to design a control unit based on a 256-bit counter, a 128-bit counter, and a simple Finite State Machine (FSM).

B. Control Unit

The control bits of the ALU, the read and write addresses of the register file and the key memory, and the write enable signals are computed by a control unit that mainly consists of an address generator and an instruction memory. At first glance, it seems that each algorithm (AES key expansion, AES encryption, AES decryption, P_ℓ, and Q_ℓ) requires a different addressing scheme. However, we described a way to generate all read and write addresses of the AES and the hash function ECHO [4] by means of a modulo-16 counter and a modulo-256 counter in our previous work [9]. The same design philosophy allows us to generate the addresses of Grøstl. Since the internal state contains up to 128 bytes, we have to replace the modulo-16 counter by a modulo-128 counter. Our control unit generates a read address and its corresponding write address at each clock cycle. Since our coprocessor embeds several pipeline stages, it is mandatory to delay write addresses and write enable signals accordingly. Furthermore, the latency depends on the algorithm being executed (Figure 3). On Xilinx devices, an efficient solution consists in synchronizing control signals by means of SRL16 primitives, whose depth can be dynamically adjusted.

The three most significant bits of read and write addresses select a block of 128 bytes in the memory, and their generation is quite straightforward. We refer the reader to our open source VHDL code and to [9] for further details. Therefore, we focus only on the generation of the seven least significant bits (*i.e.*


User interface
Address generation
Control unit
Instruction ROM
Register file and key memory
(dual-ported memory block)
Modulo-8 counter $c_{2:0}$
c_0
c_1
c_2
$\overline{AES}/Grøstl$
Shift
4-stage FIFO
4-stage FIFO
we_1
$a_{19:0}$
we_0
$a_{9:0}$
we_2
$a_{29:20}$
we_3
$a_{39:30}$
Data
Port B
Port A
18k BlockRAM
Port A
Port B
18k BlockRAM
R_i
R_j
R_k
$ctrl_0$
$ctrl_1$
$ctrl_{3:2}$
$\overline{AES}/Grøstl$
$R_k \leftarrow \mathcal{A} \cdot g(R_i) \oplus \mathcal{B} \cdot R_j$
SubBytes
InvSubBytes
0
1
0
1
0
1
Multiplication by $\mathcal{A}$
• Identity ($\mathcal{I}_{\text{AES}}$)
• MixColumns ($\mathcal{M}_E$)
• InvMixColumns ($\mathcal{M}_D$)
• Cyclic rotation ($\mathcal{P}_{\text{AES}}$)
• Identity ($\mathcal{I}_{\text{Grøstl}}$)
• MixBytes ($\mathcal{M}_{\text{Grøstl}}$)
Multiplication by $\mathcal{B}$
• Identity ($\mathcal{I}_{\text{AES}}$)
• InvMixColumns ($\mathcal{M}_D$)
• Identity ($\mathcal{I}_{\text{Grøstl}}$)
• Cyclic rotation ($\mathcal{P}_{\text{Grøstl}}$)
r_0 r_1 r_2 r_3 r_4 r_5 r_6 r_7
sk_0 sk_1 sk_2 sk_3 sk_4 sk_5 sk_6 sk_7
AddRoundKey and KeyExpansion
00
01
10
11
$k_{i,r \cdot N_c + j}$
On-the-fly computation of round constants
$a_{26:20}$
Round index r
$ctrl_{5:4}$
$ctrl_6$
$ctrl_{6:7}$
8 bits
Control signals
Latency: 2 clock cycles
Latency:
• AES: 6 clock cycles
• Grøstl: 10 clock cycles
On-the-fly computation of round constants
$k_{i,r \cdot N_c + j}$
a_{26} a_{25} a_{24} a_{23} a_{22} a_{21} a_{20}
Column index j
Row index i
Round index r
7-bit address $a_{26:20}$ (port A of the key memory)

$ctrl_{1:0}$	$g(R_i)$
00	Identity function
10	S_{RD}
11	$\text{S}_{\text{RD}}^{-1}$

$\overline{AES}/Grøstl$	$ctrl_{3:2}$	
0	00	$\mathcal{I}_{\text{AES}} = \text{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00})$
0	01	$\mathcal{M}_E = \text{circ}(\texttt{02}, \texttt{03}, \texttt{01}, \texttt{01})$
0	10	$\mathcal{M}_D = \text{circ}(\texttt{0E}, \texttt{0B}, \texttt{0D}, \texttt{09})$
0	11	$\mathcal{P}_{\text{AES}} = \text{circ}(\texttt{00}, \texttt{01}, \texttt{00}, \texttt{00})$
1	00	$\mathcal{I}_{\text{Grøstl}} = \text{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00})$
1	01	$\mathcal{M}_{\text{Grøstl}} = \text{circ}(\texttt{02}, \texttt{02}, \texttt{03}, \texttt{04}, \texttt{05}, \texttt{03}, \texttt{05}, \texttt{07})$

$\overline{AES}/Grøstl$	$ctrl_6$	
0	0	$\mathcal{I}_{\text{AES}} = \text{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00})$
0	1	$\mathcal{M}_D = \text{circ}(\texttt{0E}, \texttt{0B}, \texttt{0D}, \texttt{09})$
0	0	$\mathcal{I}_{\text{Grøstl}} = \text{circ}(\texttt{01}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00})$
0	1	$\mathcal{P}_{\text{Grøstl}} = \text{circ}(\texttt{00}, \texttt{01}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00}, \texttt{00})$

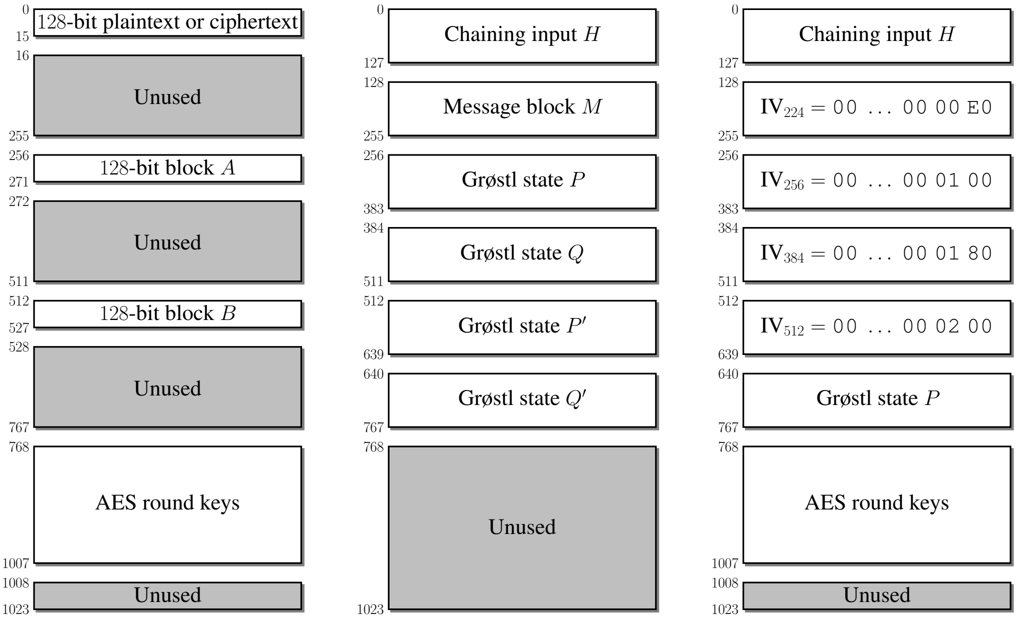
Figure 3. General architecture of our unified 8-bit coprocessor for AES and Grøstl.

the location of a byte in the internal state) in the following. Note that we

- interleave the computation of P_ℓ and Q_ℓ in order to avoid data dependencies between two consecutive rounds and
- implement the ShiftBytes step by accordingly addressing the register file.

Figures 5 and 6 summarize the address generation process of Grøstl-256 and Grøstl-512, respectively. At each clock cycle, a new read address is generated by adding a 7-bit offset to the current read address. The rules summarized in Table V allow one to compute the offset according to the permutation being executed. They involve the following signals:

- depending on the value of ℓ, *Counter* is a modulo-64 counter or a modulo-128 counter used to enumerate the bytes of the internal state (it simply consists of the six or seven least significant bits of the modulo-256 counter, and defines the write address of the register file);
- *Switch* is a 1-bit signal equal to one if and only if we are performing the last step of P_ℓ or Q_ℓ;
- Ω is a 1-bit flag indicating that the output transformation



(a) Register file (AES) (b) Register file (Grøstl) (c) Key memory (AES and Grøstl)

Figure 4. Memory organization.

is carried out.

Consider for instance Q_{1024} and assume that *Counter* $= (0010001)_2 = 17$. We compute the offset according to Table V and obtain:

$$\begin{aligned} \textit{Offset} &= 0 \wedge 0 \wedge 1 \parallel (\neg 0) \wedge 0 \parallel 0 \vee (\neg 0) \vee (\neg 1) \parallel \\ &\quad\; 0 \wedge 1 \wedge (\neg 0) \parallel 0 \parallel 0 \parallel 1 \\ &= (0010001)_2 = 17. \end{aligned}$$

Since the current read address is equal to 41, the next read address is given by $(41 + \textit{Offset}) \bmod 128 = 58$. *Counter* is now equal to $(0010010)_2 = 18$ and the new offset is given by:

$$\begin{aligned} \textit{Offset} &= 0 \wedge 1 \wedge 0 \parallel (\neg 0) \wedge 1 \parallel 0 \vee (\neg 1) \vee (\neg 0) \parallel \\ &\quad\; 1 \wedge 0 \wedge (\neg 0) \parallel 0 \parallel 0 \parallel 1 \\ &= (0110001)_2 = 49. \end{aligned}$$

The computation of $\textit{Offset}_4$ involves seven inputs: Ω, *Switch*, the three least significant bits of *Counter*, and two bits to define the permutation (P_{512}, Q_{512}, P_{512} or Q_{512}). On modern Xilinx devices, it is implemented by means of two 6-input Look-Up Tables (LUTs) and a dedicated multiplexer (F7AMUX or F7BMUX). Since $\textit{Offset}_5$ and $\textit{Offset}_6$ share the same five inputs, they are generated thanks to a LUT with two independent outputs (LUT6_2 primitive). A fourth LUT allows us to compute $\textit{Offset}_3$. Thus, we defined an extremely lightweight address generation process for Grøstl. It can easily be combined with the addressing scheme of the AES described in [9].

The output transformation requires special attention: since it involves only P_ℓ, five idle clock cycles between two consecutive rounds are mandatory to avoid memory collisions. Let us consider the ith round of Grøstl-256 to describe the problem (Figure 7). The control unit generates the address of $p_{7,6}$ (read operation) and $p_{7,7}$ (write operation) at time t. However, our coprocessor includes $D = 12$ pipeline stages and we write the new value of $p_{7,7}$ in the register file at time $t + D$. In order to update the first column of the internal state P, we have to read $p_{0,0}$, $p_{1,1}$, $p_{2,2}$, $p_{3,3}$, $p_{4,4}$, $p_{5,5}$, $p_{6,6}$, and $p_{7,7}$. The latter is available on port A of the register file at time $t + D + 1$, which means that $p_{0,0}$ can be read at time $t + D - 6 = t + 6$. It is therefore necessary to introduce five idle clock cycles between two rounds.

Table VI summarizes the number of clock cycles required for the AES and Grøstl. In the case of the AES, we obtain exactly the same results as in [9]. Thanks to our careful organization of the memory, we achieve a perfectly tight scheduling (no idle cycle) for the compression function of Grøstl.

Table VI
NUMBER OF CLOCK CYCLES REQUIRED FOR THE AES AND GRØSTL.

Algorithm		**# cycles**
AES-128	Key expansion	365
	Encryption/decryption	231
AES-192	Key expansion	421
	Encryption/decryption	273
AES-256	Key expansion	476
	Encryption/decryption	315
Grøstl-256	Compression function	1411
	Output transformation	757
Grøstl-512	Compression function	3843
	Output transformation	1993

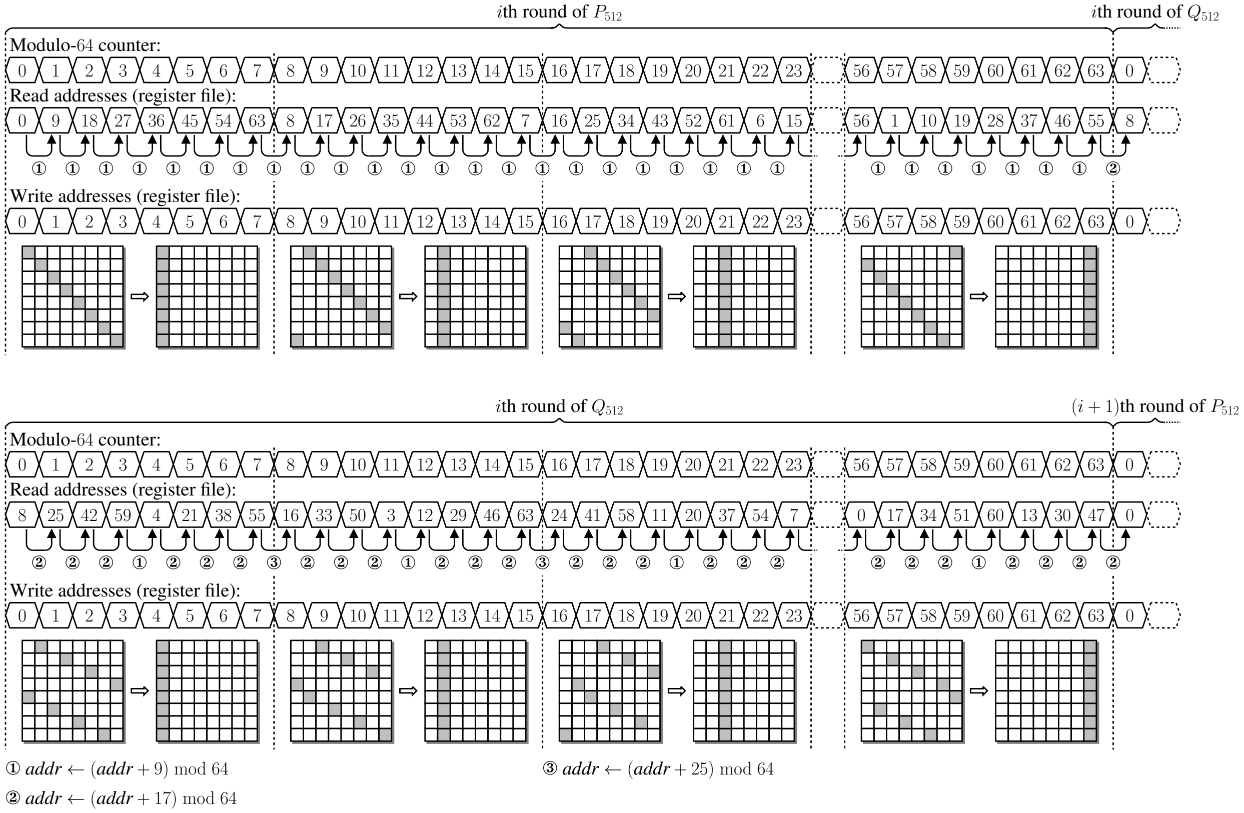


Figure 5. Address generation of P_{512} and Q_{512}.

Table V
COMPUTATION OF THE OFFSET ACCORDING TO THE PERMUTATION BEING EXECUTED.

	P_{512}	P_{1024}	Q_{512}	Q_{1024}
$\textit{Offset}_6$	0	0	0	$\textit{Counter}_2 \wedge \textit{Counter}_1 \wedge \textit{Counter}_0$
$\textit{Offset}_5$	0	$\textit{Counter}_2 \wedge \textit{Counter}_1$	0	$(\neg\textit{Counter}_2) \wedge \textit{Counter}_1$
$\textit{Offset}_4$	$(\neg\Omega) \wedge \textit{Switch}$		$\textit{Counter}_2 \vee \neg\textit{Counter}_1 \vee (\neg\textit{Counter}_0)$	
$\textit{Offset}_3$	$\Omega \vee (\neg\textit{Switch})$		$\textit{Counter}_1 \wedge \textit{Counter}_0 \wedge (\neg\textit{Switch})$	
$\textit{Offset}_{2:0}$	$0 \parallel 0 \parallel 1$			

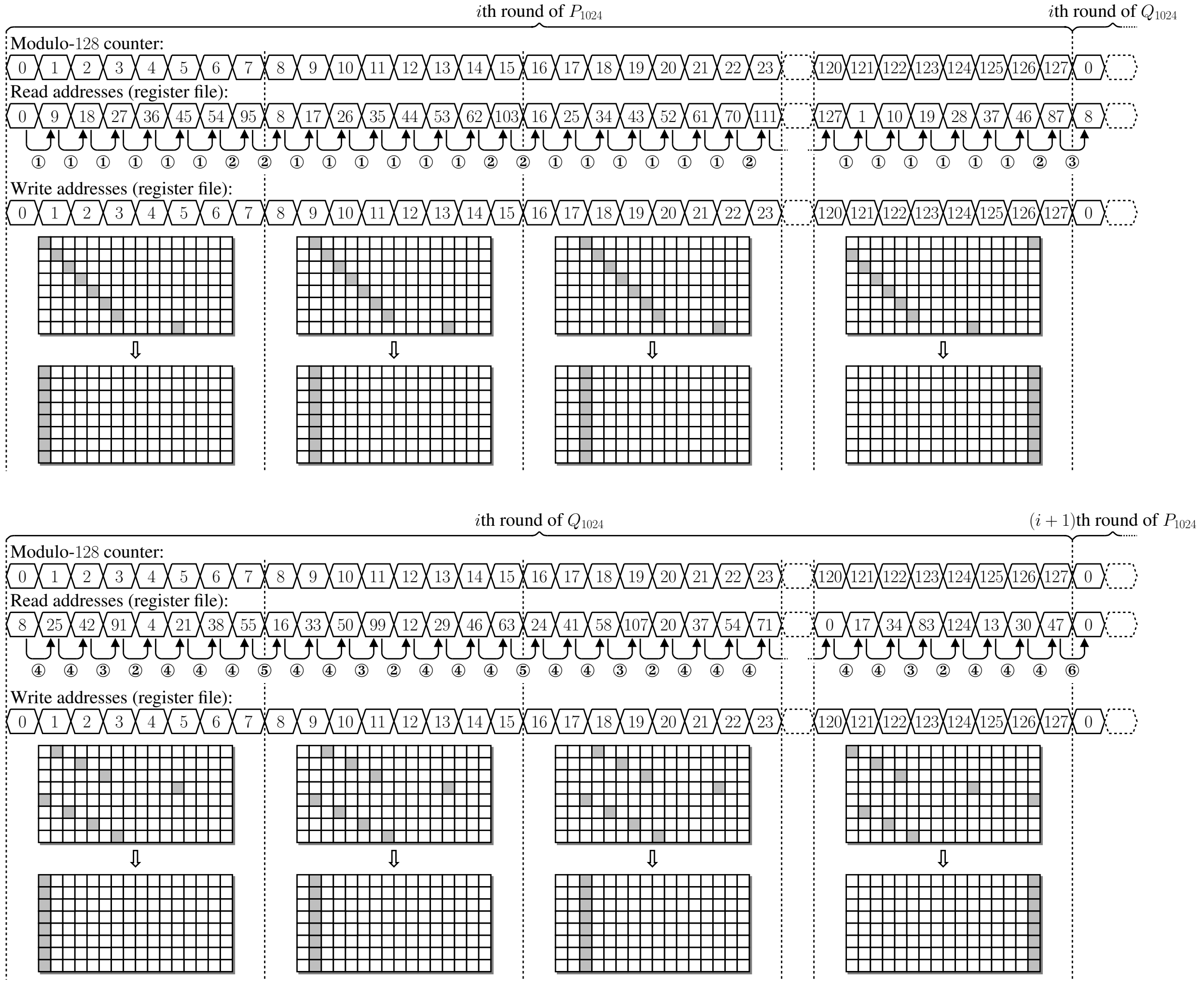
C. Arithmetic and Logic Unit

The SubBytes and InvSubBytes steps are often considered as the most critical part of the AES, and several architectures for S_{RD} and $\mathrm{S}_{\mathrm{RD}}^{-1}$ have already been described in the open literature (see for instance [12] for a comprehensive bibliography). On Xilinx Virtex-6 FPGAs, the best design strategy consists in implementing the AES S-boxes as 8-input tables [10]. Two control bits $\textit{ctrl}_{1:0}$ allow us to perform SubBytes, InvSubBytes, or to bypass this stage when g is the identity function.

In the case of the AES, the first matrix multiplication of Equation (1) can involve any of the four circulant matrices defined in Section II. Grøstl requires only $\mathcal{I}_{\text{Grøstl}}$ and $\mathcal{M}_{\text{Grøstl}}$. Let us define the control signal $\overline{\text{AES}}/\text{Grøstl}$ whose role is to identify the algorithm being executed. Together with two control bits $\textit{ctrl}_{3:2}$, this signal allows us to select matrix $\mathcal{A}$. The choice of matrix $\mathcal{B}$ turns out to be simpler and requires a single extra control bit $\textit{ctrl}_6$ (Figure 3).

Since we emphasize reducing the usage of FPGA resources, we adopt the multiply-and-accumulate approach proposed by Hämäläinen *et al.* [14], and need N_l clock cycles to multiply one column of the state or the round key array by a circulant matrix (Figures 8 and 9). We compute a first partial product and store the result in registers r_0 to r_{N_l}. Then, at each clock cycle, the intermediate result is rotated and accumulated with a new partial product. This process involves a *Shift* control signal to distinguish between the first step and the subsequent ones. Such a signal can be generated by computing the bitwise OR of the bits of a modulo-N_l counter. Let us consider the three bits $c_{2:0}$ of a modulo 8 counter. Since $N_l = 4$ and $N_l = 8$ for AES and Grøstl, respectively, we define

$$\begin{aligned}\textit{Shift} \leftarrow &\begin{cases} c_1 \vee c_0 & \text{if } \overline{\text{AES}}/\text{Grøstl} = 0,\\ c_2 \vee c_1 \vee c_0 & \text{if } \overline{\text{AES}}/\text{Grøstl} = 1\end{cases}\\ &= (c_2 \wedge \overline{\text{AES}}/\text{Grøstl}) \vee c_1 \vee c_0.\end{aligned}$$



① $addr \leftarrow (addr + 9) \bmod 128$

② $addr \leftarrow (addr + 41) \bmod 128$

③ $addr \leftarrow (addr + 49) \bmod 128$

④ $addr \leftarrow (addr + 17) \bmod 128$

⑤ $addr \leftarrow (addr + 89) \bmod 128$

⑥ $addr \leftarrow (addr + 81) \bmod 128$

Figure 6. Address generation of P_{1024} and Q_{1024}.

Note that the rotation mechanism depends on the algorithm being executed: the AES involves registers r_0, r_1, r_2, and r_3, whereas Grøstl requires eight registers to store intermediate results. Therefore, the feedback mechanism is implemented by means of a multiplexer controlled by $\overline{\text{AES}}$/Grøstl. We describe in Appendix how to optimize the MixColumns and MixBytes steps on the latest Xilinx FPGAs.

Figure 10 describes the component we designed to perform the AddRoundKey and KeyExpansion steps. Since our matrix multiplication units output N_l bytes, we perform N_l additions over $\mathbb{F}_{2^8}$ in parallel and store the result in a shift register. Then, we write data byte by byte in the register file, and a modulo-N_l counter controls the process. Therefore, it suffices to delay our *Shift* signal by a total of N_l clock cycles, which is the latency of a matrix-vector multiplication. Additional hardware resources allow us to deal with the round constant RC involved in the key expansion of the AES (see [9] for details).

The last operation we have to consider is the AddRoundKey step of Grøstl. In order to compute $p'_{i,j}$ (Algorithm 2, line 12), we generate $k_{i,r\cdot N_c+j}$ on-the-fly. Recall that:

- i is a 3-bit row index;
- j is a 4-bit column index;
- r is a 4-bit round index.

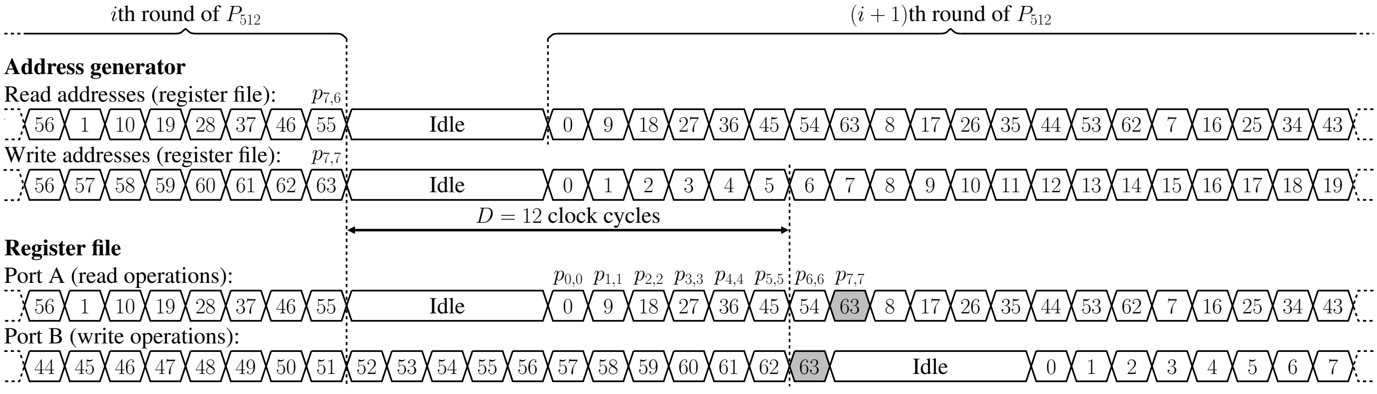
The indices i and j are given by the control signal $a_{26:20} = j \parallel i = 8j + i$ (Figure 3). According to Algorithm 1, we have:

$$k_{i,r\cdot N_c+j} \leftarrow \begin{cases} j \parallel r & \text{when } i = 0, \\ \texttt{00} & \text{otherwise.} \end{cases}$$

Since $i = a_{22:20}$ and $j = a_{26:23}$, we can rewrite the above equation as follows:

$$k_{i,r\cdot N_c+j} \leftarrow \begin{cases} a_{26:23} \parallel r & \text{when } a_{20} \vee a_{21} \vee a_{22} = 0, \\ \texttt{00} & \text{otherwise,} \end{cases}$$

and compute $k_{i,r\cdot N_c+j}$ by means of a 3-input NOR gate and



ith round of P_{512}
$(i+1)$th round of P_{512}
Address generator
Read addresses (register file):
$p_{7,6}$
56 1 10 19 28 37 46 55 Idle 0 9 18 27 36 45 54 63 8 17 26 35 44 53 62 7 16 25 34 43
Write addresses (register file):
$p_{7,7}$
56 57 58 59 60 61 62 63 Idle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
$D = 12$ clock cycles
Register file
Port A (read operations):
$p_{0,0}$ $p_{1,1}$ $p_{2,2}$ $p_{3,3}$ $p_{4,4}$ $p_{5,5}$ $p_{6,6}$ $p_{7,7}$
56 1 10 19 28 37 46 55 Idle 0 9 18 27 36 45 54 63 8 17 26 35 44 53 62 7 16 25 34 43
Port B (write operations):
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 Idle 0 1 2 3 4 5 6 7

Figure 7. Latency between two consecutive rounds of P_{512} during the output transformation.

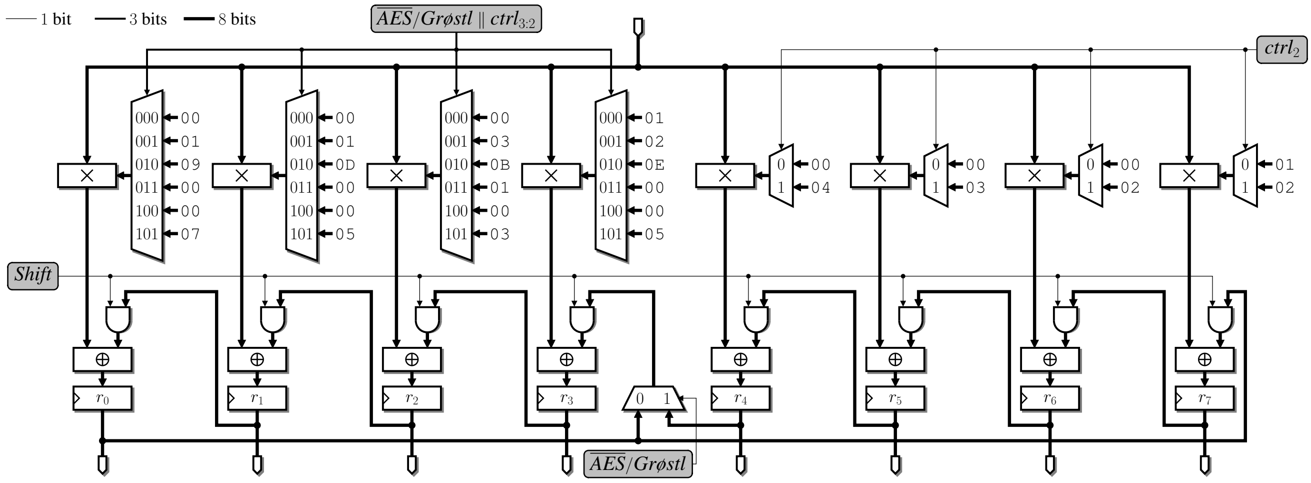


Figure 8. Implementation of MixColumns and MixBytes.

eight 2-input AND gates. A multiplexer controlled by *ctrl*$_{5:4}$ allows us to inject $k_{i,r \cdot N_c+j}$, $\neg k_{i,r \cdot N_c+j}$, $k_{i,j} \oplus h_{i,j}$ (first key injection of P_ℓ), or a variable stored in the key memory.

V. Results and Comparisons

We captured our architecture in the VHDL language and prototyped our coprocessor on several Xilinx FPGAs with average speedgrade. Table VII summarizes our place-and-route results measured with ISE 14.2. In order to evaluate the hardware overhead introduced by the AES, we designed a second coprocessor that implements only Grøstl-256 and Grostl-512 (Table VIII). It is possible to reduce the number of slices by implementing a subset of the functionalities (*e.g.* a single level of security, AES without key expansion, etc.).

Our coprocessor requires a similar number of slices and achieves the same clock frequency as the architecture we designed for ECHO and AES [9]. However, the implementation of Grøstl on our architecture involves a smaller number of instructions and the throughput turns out to be slightly higher than the one of ECHO. Therefore, two conclusions we drew in our previous work can be transposed here:

- Helion Technology [26] is selling a tiny AES core that implements encryption, decryption, and key expansion at all levels of security. The coprocessor occupies only 88 Virtex-6 slices and achieves a throughput of 83 Mbps in the case of AES-128. Our unified coprocessor is almost twice as big, but we achieve a better encryption/decryption rate and improve the area–time product compared to the tiny AES core designed by Helion Technology. Thus, combining the hash function Grøstl with the AES does not impact the overall performance of the latter.
- The unified core for SHA-1, SHA-224/256, and SHA-384/512 designed by Helion Technology [25] turns out to be larger and slightly slower than our coprocessor. Furthermore, the Helion commercial core must be supplemented with an AES core to provide the same functionalities as our architecture. Assuming that the security of Grøstl is at least as good as the one of SHA-2, Grøstl is a clear winner for resource-constrained devices.

Järvinen [15] 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

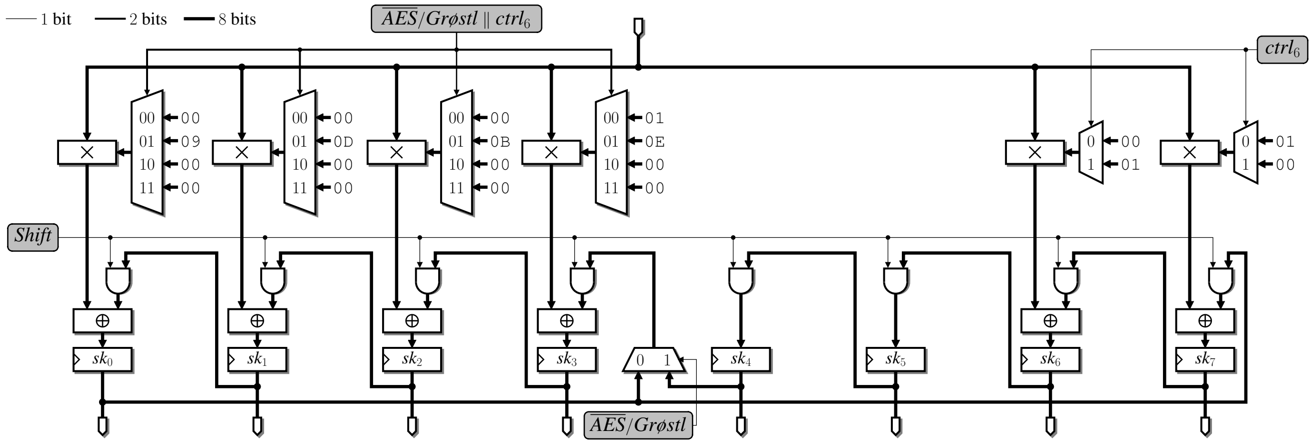


Figure 9. Multiplication by $\mathcal{I}_{\text{AES}}$, $\mathcal{M}_D$, $\mathcal{I}_{\text{Grøstl}}$, and $\mathcal{P}_{\text{Grøstl}}$.

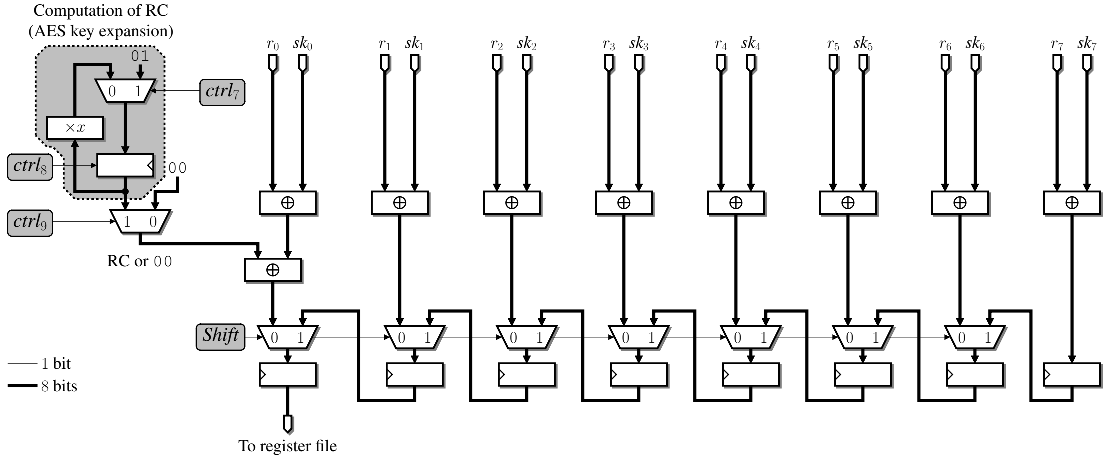


Figure 10. Implementation of AddRoundKey and KeyExpansion.

Table VII
PLACE-AND-ROUTE RESULTS FOR OUR UNIFIED COPROCESSOR ON VIRTEX-6, ARTIX-7, KINTEX-7, AND VIRTEX-7 FPGAS. THE THROUGHPUT OF GRØSTL IS COMPUTED FOR A ONE-BLOCK MESSAGE.

FPGA	**Area [slices]**	**Frequency [MHz]**	**Throughput [Mbits/s]**				
			AES-128	**AES-192**	**AES-256**	**Grøstl-256**	**Grøstl-512**
Virtex-6 (xc6vlx75t-2)	169	393	217	184	159	92	69
Artix-7 (xc7a100t-2)	188	265	146	124	107	62	46
Kintex-7 (xc7k70t-2)	172	438	242	205	177	103	76
Virtex-7 (7vx330t-2)	185	415	229	194	168	98	72

unified coprocessor.

We report in Table IX the latest FPGA implementation results of the five SHA-3 finalists (see for instance [20], [21] for a survey of parallel architectures). We consider here the least favorable case for Grøstl, in which a single block is processed. The throughput of Grøstl-256 is for instance given by:

$$\textit{throughput} = \frac{512 \cdot \#\textit{blocks}}{(1411 \cdot \#\textit{blocks} + 757) \cdot T},$$

where T denotes the clock period. When the number of blocks increases, one can neglect the cost of the output transformation and the throughput tends asymptotically to $512/(1411 \cdot T)$.

Most of the architectures described in the open literature focus on a single level of security. In this context, BLAKE [2] is obviously the best choice for low-area implementations on FPGA. However, as soon as a circuit must support several levels of security, Grøstl will offer the most compact solution.

VI. CONCLUSION

The design philosophy we proposed in [9] allowed us to develop a low-area coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function Grøstl at all levels of security. Our architecture is built around an 8-bit datapath and the ALU performs a

Table VIII
PLACE-AND-ROUTE RESULTS FOR OUR GRØSTL COPROCESSOR ON VIRTEX-6, ARTIX-7, KINTEX-7, AND VIRTEX-7 FPGAS. THE THROUGHPUT IS COMPUTED FOR A ONE-BLOCK MESSAGE.

FPGA	Area [slices]	Frequency [MHz]	Throughput [Mbits/s]	
			Grøstl-256	Grøstl-512
Virtex-6 (xc6vlx75t-2)	102	413	97	72
Artix-7 (xc7a100t-2)	131	331	78	58
Kintex-7 (xc7k70t-2)	111	450	106	78
Virtex-7 (7vx330t-2)	108	480	113	84

single instruction that allows for implementing both algorithms. Despite the various addressing schemes required for the different steps of Grøstl and the AES, our control unit remains compact: all addresses are generated by means of a modulo-128 counter and a modulo-256 counter. Thanks to an alternative description of Grøstl and a meticulous organization of the memory, we manage to implement the compression function f (Algorithm 2) without any pipeline stall. The key element of our approach is to take advantage of the parallelism of Grøstl to

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

At the cost of 67 Virtex-6 slices, one can add the AES functionalities to a Grøstl coprocessor. Despite of the differences between the two algorithms (size of the internal state, coefficients of the circulant matrices, key schedule, etc.), resource sharing is possible. Assuming that the security guarantees of Grøstl are at least as good as the ones of the other SHA-3 finalists, Grøstl is the best candidate for low-area cryptographic coprocessors.

Our architecture is mainly designed for embedded systems. Thus, it would be interesting to conduct side-channel and fault injection attacks in future work. Since the ALU executes the same instruction at each clock cycle, our design strategy could offer a protection against some attacks.

ACKNOWLEDGEMENTS

The authors would like to thank Ray Cheung for his 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] N. At, J.-L. Beuchat, and İ. San. Compact implementation of Threefish and Skein on FPGA. In A. Levi, M. Badra, M. Cesana, M. Ghassemian, Ö. Gürbüz, N. Jabeur, M. Klonowski, A. Maña, S. Sargento, and S.Zeadally, editors, *Proceedings of the Fifth IFIP International Conference on New Technologies, Mobility and Security–NTMS 2012*. IEEE eXpress Conference Publishing, 2012.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R.C.-W. Phan. SHA-3 proposal BLAKE (version 1.4). Available at http://www.131002.net/blake, January 2011.

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

[4] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 proposal: ECHO. Available at http://crypto.rd.francetelecom.com/echo, 2009.

[5] R. Benadjila, O. Billet, S. Gueron, and M.J.B. Robshaw. The Intel AES instructions set and the SHA-3 candidates. In M. Matsui, editor, *Advances in Cryptology–ASIACRYPT 2009*, number 5912 in Lecture Notes in Computer Science, pages 162–178. Springer, 2009.

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

[7] J.-L. Beuchat, E. Okamoto, and T. Yamazaki. A compact FPGA implementation of the SHA-3 candidate ECHO. Cryptology ePrint Archive, Report 2010/364, 2010.

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

[9] 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*, 1(2):101–121, 2011.

[10] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In S. Vaudenay, editor, *Progress in Cryptology–AFRICACRYPT 2008*, number 5023 in Lecture Notes in Computer Science, pages 16–26. Springer, 2008.

[11] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002.

[12] K. Gaj and P. Chodowiec. FPGA and ASIC implementations of the AES. In Ç.K. Koç, editor, *Cryptographic Engineering*, pages 235–294. Springer, 2009.

[13] P. Gauravaram, L.R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S.S. Thomsen. Grøstl – a SHA-3 candidate. Available at http://www.groestl.info, 2011.

[14] P. Hämäläinen, T. Alho, M. Hännikäinen, and T.D. Hämäläinen. Design and implementation of low-area and low-power AES encryption hardware core. In *Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools–DSD'06*, pages 577–583. IEEE Computer Society, 2006.

[15] K. Järvinen. Sharing resources between AES and the SHA-3 second round candidates Fugue and Grøstl. In *The Second SHA-3 Candidate Conference*, August 2010.

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

[17] B. Jungk. Evaluation of compact FPGA implementations for all SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, March 2012.

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

[19] 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.

[20] M. Knežević, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, Ü. Kocabaş, J. Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki. Fair and consistent hardware evaluation of fourteen round two SHA-3 candidates. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(5):827–840, May 2012.

Table IX
COMPACT IMPLEMENTATIONS OF THE FIVE SHA-3 FINALISTS 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]**
Aumasson *et al.* [2]	BLAKE-256	xc5vlx110	390	–	91	412
Beuchat *et al.* [8]*	BLAKE-256	xc6vlx75t-2	52	2	456	194
Jungk [17]	BLAKE-256	xc6v	235	–	231	518
Jungk [17]	BLAKE-256	xc6v	404	–	185	823
Kaps *et al.* [18]	BLAKE-256	xc6vlx75t-1	163	1	197	327
Kaps *et al.* [18]	BLAKE-256	xc6vlx75t-1	166	–	268	445
Aumasson *et al.* [2]	BLAKE-512	xc5vlx110	939	–	59	468
Beuchat *et al.* [8]*	BLAKE-512	xc6vlx75t-2	81	3	374	280
Kerckhof *et al.* [19]	BLAKE-512	xc6vlx75t-1	192	–	240	183
Yamazaki *et al.* [28]	BLAKE-256 and BLAKE-512	xc5vlx50-2	138	3	342	2×150 (BLAKE-256) 264 (BLAKE-512)
Jungk [16]†	Grøstl-256	xc5v	470	–	354	1132
Jungk [17]†	Grøstl-256	xc6v	328	–	365	1168
Kaps *et al.* [18]	Grøstl-256	xc6vlx75t-1	241	1	244	115
Kaps *et al.* [18]	Grøstl-256	xc6vlx75t-1	263	–	359	240
Yamazaki [27]†	Grøstl-256	xc6vlx75t-2	82	1	410	154
Kerckhof *et al.* [19]†	Grøstl-512	xc6vlx75t-1	260	–	280	640
This work	Grøstl-256 and Grøstl-512	xc6vlx75t-2	102	1	413	97 (Grøstl-256) 72 (Grøstl-512)
This work	Grøstl-256, Grøstl-512, AES-128, AES-192, and AES-256	xc6vlx75t-2	169	1	393	92 (Grøstl-256) 69 (Grøstl-512)
Jungk [16]	JH-256	xc5v	205	–	341	27
Jungk [17]	JH-256	xc6v	193	–	385	29
Jungk [17]	JH-256	xc6v	424	–	365	1112
Kaps *et al.* [18]	JH-256	xc6vlx75t-1	196	1	243	148
Kaps *et al.* [18]	JH-256	xc6vlx75t-1	171	–	252	154
Kerckhof *et al.* [19]	JH-512	xc6vlx75t-1	240	–	288	214
Jungk [17]	Keccak$[r = 1088, c = 512]$	xc6v	397	–	197	1071
Kaps *et al.* [18]	Keccak$[r = 1088, c = 512]$	xc6vlx75t-1	129	1	260	74
Kaps *et al.* [18]	Keccak$[r = 1088, c = 512]$	xc6vlx75t-1	106	–	299	135
Bertoni *et al.* [6]	Keccak$[r = 576, c = 1024]$	xc5vlx50-3	448	–	265	52
Kerckhof *et al.* [19]	Keccak$[r = 576, c = 1024]$	xc6vlx75t-1	144	–	250	68
San & At [24]	Keccak$[r = 576, c = 1024]$	xc5vlx50-2	151	3	520	501
Latif *et al.* [22]‡	Skein-256-256	xc5vlx110-3	821	Not specified	119	1610
Jungk [16]‡	Skein-512-256	xc5v	555	–	271	237
Jungk [17]‡	Skein-512-256	xc6v	406	–	318	277
Kaps *et al.* [18]	Skein-512-256	xc6vlx75t-1	207	1	166	17
Kaps *et al.* [18]	Skein-512-256	xc6vlx75t-1	193	–	193	21
At *et al.* [1]	Skein-512-512	xc6vlx75t-1	132	2	276	80
Kerckhof *et al.* [19]‡	Skein-512-512	xc6vlx75t-1	240	–	160	179

*Modified to implement the tweaked version submitted for the final round of the SHA-3 competition.
†Without output transformation.
‡Single call to Threefish-512.

[21] K. Latif, M.M. Rao, A. Aziz, and A. Mahboob. Efficient hardware implementations and hardware performance evaluation of SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, March 2012.

[22] 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.

[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*, September 2012.

[24] İ. San and N. At. Compact Keccak hardware architecture for data integrity and authentication on FPGAs. *Information Security Journal: A Global Perspective*, 21(5):231–242, 2012.

[25] Helion Technology. FULL DATASHEET–Tiny hash core family for Xilinx FPGA. Revision 2.0 (11/06/2010).

[26] Helion Technology. OVERVIEW DATASHEET–Ultra-low resource AES (Rijndael) cores for Xilinx FPGA. Revision 1.3.0.

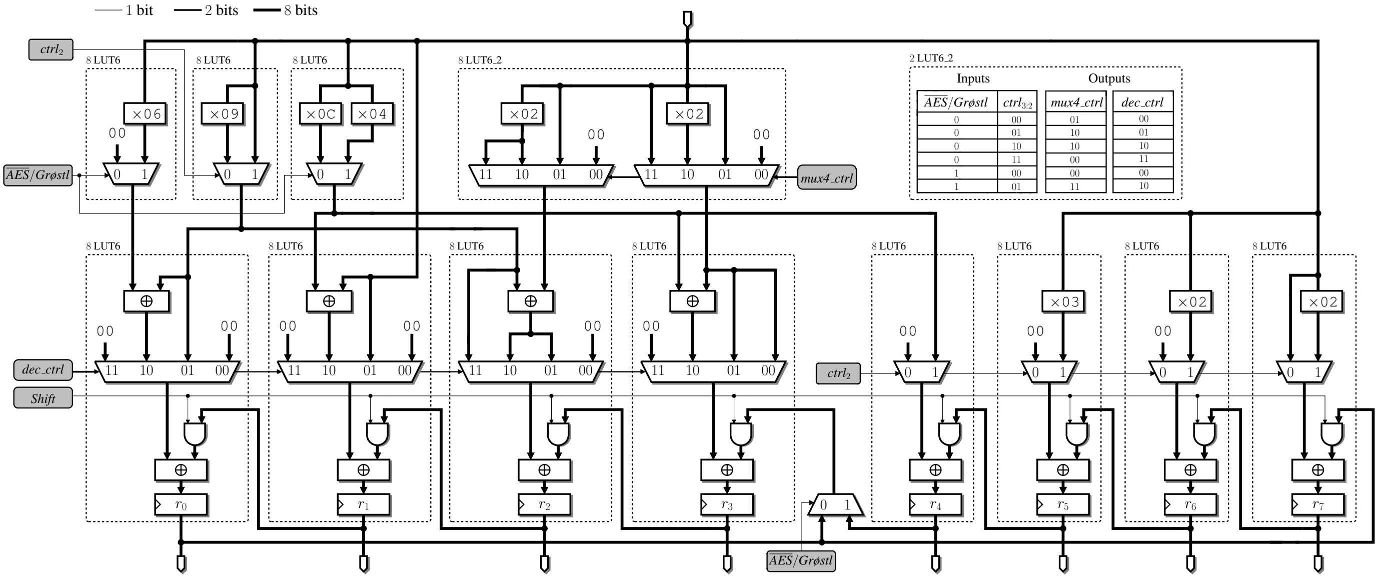
[27] T. Yamazaki. Compact implementation of hash functions on FPGA. Master's thesis, Graduate School of Systems and Information Engineering, University of Tsukuba, 2012.

[28] T. Yamazaki, J.-L. Beuchat, and E. Okamoto. BLAKE-256, BLAKE-512のコンパクトな統合実装. *IEICE*暗号と情報セキュリティ実装技術小特集号, J-95A(5):416–424, 2012.

[29] J. Zhai, C.M. Park, and G.-N. Wang. Hash-based RFID security protocol using randomly key-changed identification procedure. In *Computational Science and Its Applications–ICCSA 2006*, number 3983 in Lecture Notes in Computer Science, pages 296–305. Springer, 2006.

APPENDIX

Figure 11 describes how we take advantage of the LUT6_2 primitive in order to optimize MixColumns and MixBytes steps on the latest Xilinx FPGAs. The first step of a multipli-



Inputs		Outputs	
$\overline{AES}/Grøstl$	$ctrl_{3:2}$	$mux4\_ctrl$	dec_ctrl
0	00	01	00
0	01	10	01
0	10	10	10
0	11	00	11
1	00	00	00
1	01	11	10

Figure 11. Implementation of MixColumns and MixBytes on the latest Xilinx FPGAs.

cation by $\mathcal{M}_{\text{Grøstl}}$ is for instanced performed as follows:

$$r_0 \leftarrow \texttt{09} \cdot a_0,$$
$$r_1 \leftarrow \texttt{0C} \cdot a_0 + \texttt{01} \cdot a_0 = \texttt{0D} \cdot a_0,$$
$$r_2 \leftarrow \texttt{09} \cdot a_0 + \texttt{02} \cdot a_0 = \texttt{0B} \cdot a_0,$$
$$r_3 \leftarrow \texttt{0C} \cdot a_0 + \texttt{02} \cdot a_0 = \texttt{0E} \cdot a_0,$$
$$r_4 \leftarrow \texttt{04} \cdot a_0,$$
$$r_5 \leftarrow \texttt{03} \cdot a_0,$$
$$r_6 \leftarrow \texttt{02} \cdot a_0,$$
$$r_7 \leftarrow \texttt{02} \cdot a_0.$$