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

Jean-Luc Beuchat, Eiji Okamoto, and Teppei Yamazaki

Graduate School of Systems and Information Engineering
University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573, Japan
jeanluc.beuchat@gmail.com, okamoto@risk.tsukuba.ac.jp, yamazaki@cipher.risk.tsukuba.ac.jp

***Abstract*—We propose a compact coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function ECHO on Virtex-5 and Virtex-6 FPGAs. Our architecture is built around an 8-bit datapath. The Arithmetic and Logic Unit performs a single instruction that allows for implementing AES encryption, AES decryption, AES key expansion, and ECHO at all levels of security. Thanks to a careful organization of AES and ECHO internal states in the register file, we manage to generate all read and write addresses by means of a modulo-16 counter and a modulo-256 counter. A fully autonomous implementation of ECHO and AES on a Virtex-5 FPGA requires 193 slices and a single 36k memory block, and achieves competitive throughputs. Assuming that the security guarantees of ECHO are at least as good as the ones of the SHA-3 finalists BLAKE and Keccak, our results show that ECHO is a better candidate for low-area cryptographic coprocessors. Furthermore, the design strategy described in this work can be applied to combine the AES and the SHA-3 finalist Grøstl.**

I. Introduction

We describe a compact unified architecture for the Advanced Encryption Standard (AES) [13] and the cryptographic hash function ECHO [5] on Virtex-5 and Virtex-6 Field-Programmable Gate Arrays (FPGAs). Our coprocessor implements AES encryption, AES decryption, AES key expansion, and ECHO at all levels of security. This architecture might for instance be extremely valuable for constrained environments such as wireless sensor networks or radio frequency identification technology, where some security protocols mainly rely on cryptographic hash functions (see for example [30]). Several cryptographic protocols combine public-key cryptography (PKC) (*e.g.* RSA, elliptic curve cryptography (ECC), etc.), hash functions, random number generators, and symmetric encryption/decryption algorithms. Consider for instance the BLS short signature scheme [10]: in order to verify a signature, one has to hash the message and compute two bilinear pairings on an elliptic curve. Each pairing constitutes a time-consuming task: the best coprocessors for embedded systems compute the Tate pairing over 128-bit-security curves in more than 2ms [1], [17]. Therefore, one has more than 4ms in order to hash the next message while computing the two bilinear pairings for the current message. In this context, it is also important to keep the amount of hardware resources for the hash function as small as possible (*i.e.* it is pointless to design a massively parallel coprocessor able to hash a message in far less than 4ms).

After a short description of the AES (Section II) and the ECHO family of hash functions (Section III), we propose a unified coprocessor built around an 8-bit datapath (Section IV). We have prototyped our architecture on Virtex-5 and Virtex-6 FPGAs and discuss our results in Section V.

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 4×4 array of bytes A:

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}.$$

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, 89 is equivalent to $x^7 + x^3 + 1$ 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).

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 AES involves four byte-oriented transformations and their inverses for encryption and decryption, respectively [13]:

- The SubBytes step is the only non-linear transformation of the AES. Its role is to introduce confusion to the data so that the relationship between the secret key and the ciphertext is obscured. It updates each byte of the state using an 8-bit S-box, denoted by S_{RD}. Internally, the AES S-box computes the modular inverse of $a_{i,j}$ (the value 00 is mapped onto itself) and then applies an affine transformation.

The inverse transformation, called InvSubBytes and denoted by $\mathrm{S}_{\mathrm{RD}}^{-1}$, performs the inverse affine transformation followed by an inversion over $\mathbb{F}_{2^8}$.

- 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:

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \leftarrow \begin{pmatrix} a_{0,j} \\ a_{1,(j+1) \bmod 4} \\ a_{2,(j+2) \bmod 4} \\ a_{3,(j+3) \bmod 4} \end{pmatrix},$$

where $0 \leq j \leq 3$. The inverse operation is called InvShiftRows:

$$\begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} \leftarrow \begin{pmatrix} b_{0,j} \\ b_{1,(j+3) \bmod 4} \\ b_{2,(j+2) \bmod 4} \\ b_{3,(j+1) \bmod 4} \end{pmatrix}.$$

- The MixColumns step is a permutation operating on the AES state column by column. Together with ShiftRows, this step provides diffusion in the cipher: if a single bit of the plaintext is flipped, then the whole ciphertext should be changed. Each column of the AES state is considered as a polynomial over $\mathbb{F}_{2^8}$, and is multiplied modulo $y^4 + \mathtt{01}$ by the constant polynomial $c(y) = \mathtt{03} \cdot y^3 + \mathtt{01} \cdot y^2 + \mathtt{01} \cdot y + \mathtt{02}$ [13]. This operation is performed by multiplying each column of the state A by a circulant matrix $\mathcal{M}_E$:

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \leftarrow \underbrace{\begin{pmatrix} \mathtt{02} & \mathtt{03} & \mathtt{01} & \mathtt{01} \\ \mathtt{01} & \mathtt{02} & \mathtt{03} & \mathtt{01} \\ \mathtt{01} & \mathtt{01} & \mathtt{02} & \mathtt{03} \\ \mathtt{03} & \mathtt{01} & \mathtt{01} & \mathtt{02} \end{pmatrix}}_{\mathcal{M}_E} \cdot \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix},$$

where $0 \leq j \leq 3$.
During the inverse operation, called InvMixColumns, each column of the state is multiplied by $d(y) = \mathtt{0B} \cdot y^3 + \mathtt{0D} \cdot y^2 + \mathtt{09} \cdot y + \mathtt{0E}$. One easily checks that $d(y)$ is the multiplicative inverse of $c(y)$ modulo $y^4 + 1$:

$$c(y) \cdot d(y) \equiv \mathtt{01} \pmod{y^4 + 1}.$$

Here again, the modular multiplication by a constant polynomial can be defined by a matrix multiplication:

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} \leftarrow \underbrace{\begin{pmatrix} \mathtt{0E} & \mathtt{0B} & \mathtt{0D} & \mathtt{09} \\ \mathtt{09} & \mathtt{0E} & \mathtt{0B} & \mathtt{0D} \\ \mathtt{0D} & \mathtt{09} & \mathtt{0E} & \mathtt{0B} \\ \mathtt{0B} & \mathtt{0D} & \mathtt{09} & \mathtt{0E} \end{pmatrix}}_{\mathcal{M}_D} \cdot \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix}.$$

- 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.

A. Key Expansion

The round keys involved in the AddRoundKey steps are derived from the cipher key as follows. Let us consider an array consisting of 4 rows and $4 \cdot (N_r + 1)$ columns. Each column K_j contains four elements of $\mathbb{F}_{2^8}$ denoted by $k_{0,j}$, $k_{1,j}$, $k_{2,j}$, and $k_{3,j}$. The round key of the jth round of the AES encryption algorithm is given by columns K_{4j} to K_{4j+3} (Figure 1).

The cipher key is copied in the first N_k columns of the array, and the next columns are defined recursively. The process, summarized by Algorithms 1 and 2, involves an intermediate variable $\mathrm{RC} \in \mathbb{F}_{2^8}$ and a permutation matrix $\mathcal{P}$ defining a cyclic rotation of the bytes within a column:

$$\mathcal{P} = \begin{pmatrix} \mathtt{00} & \mathtt{01} & \mathtt{00} & \mathtt{00} \\ \mathtt{00} & \mathtt{00} & \mathtt{01} & \mathtt{00} \\ \mathtt{00} & \mathtt{00} & \mathtt{00} & \mathtt{01} \\ \mathtt{01} & \mathtt{00} & \mathtt{00} & \mathtt{00} \end{pmatrix}.$$

We denote the identity matrix by $\mathcal{I}$. This matrix notation will be useful to pinpoint a unified 8-bit datapath for key expansion, encryption, and decryption in Section IV-A.

Algorithm 1 AES key expansion for $N_k \leq 6$.

Input: A cipher key $K_0, \ldots, K_{N_k-1}$.
Output: Expanded key.

1. $\mathrm{RC} \leftarrow x^0$;
2. **for** $j = N_k$ to $4N_r + 3$ **do**
3. **if** $j \bmod N_k = 0$ **then**
4. $K_j \leftarrow \mathcal{P} \cdot \begin{pmatrix} \mathrm{S_{RD}}(k_{0,j-1}) \\ \mathrm{S_{RD}}(k_{1,j-1}) \\ \mathrm{S_{RD}}(k_{2,j-1}) \\ \mathrm{S_{RD}}(k_{3,j-1}) \end{pmatrix} \oplus \mathcal{I} \cdot K_{j-N_k}$;
5. $k_{0,j} \leftarrow k_{0,j} \oplus \mathrm{RC}$;
6. $\mathrm{RC} \leftarrow x \cdot \mathrm{RC}$;
7. **else**
8. $K_j \leftarrow \mathcal{I} \cdot K_{j-1} \oplus \mathcal{I} \cdot K_{j-N_k}$;
9. **end if**
10. **end for**
11. **Return** $K_{N_k}, \ldots, K_{4N_r+3}$;

B. Encryption

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 (Algorithm 3). Noting that the order of ShiftRows and SubBytes is indifferent [13], we obtain the datapath depicted on Figure 2.

Algorithm 3 updates the AES state column by column. Since the ShiftRows transformations performs cyclical left shifts of the three bottom rows of the state, we have to be careful not to overwrite bytes that are still involved in the forthcoming MixColumns steps ($a_{1,0}$ is for instance needed to update the fourth column of the AES state, and should not be overwritten when updating the first column). Thus, the

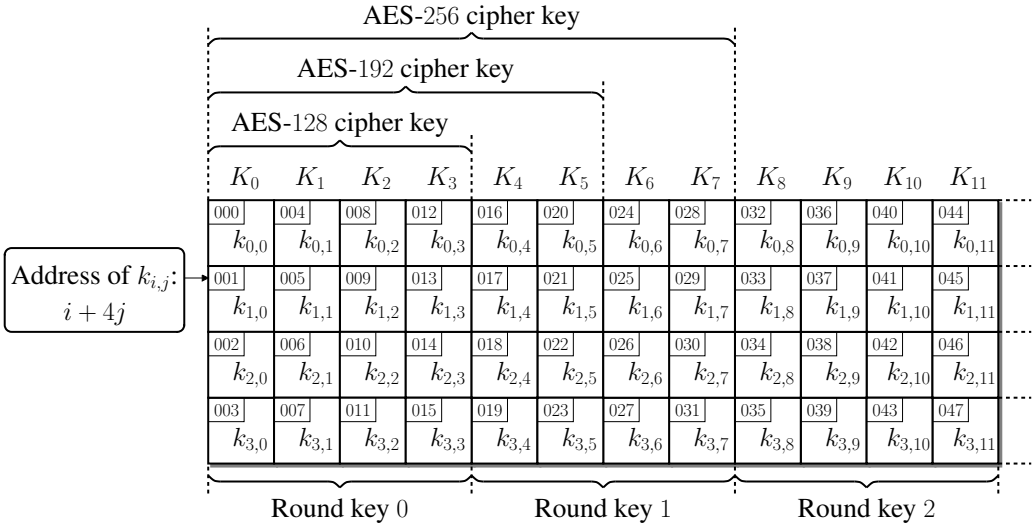


Fig. 1. Key expansion and round selection.

Algorithm 2 AES key expansion for $N_k > 6$.

Input: A cipher key $K_0, \ldots, K_{N_k-1}$.
Output: Expanded key.

1. RC $\leftarrow x^0$;
2. **for** $j = N_k$ to $4N_r + 3$ **do**
3. **if** $j \bmod N_k = 0$ **then**
4. $K_j \leftarrow \mathcal{P} \cdot \begin{pmatrix} \mathrm{S_{RD}}(k_{0,j-1}) \\ \mathrm{S_{RD}}(k_{1,j-1}) \\ \mathrm{S_{RD}}(k_{2,j-1}) \\ \mathrm{S_{RD}}(k_{3,j-1}) \end{pmatrix} \oplus \mathcal{I} \cdot K_{j-N_k}$;
5. $k_{0,j} \leftarrow k_{0,j} \oplus$ RC;
6. RC $\leftarrow x \cdot$ RC;
7. **else if** $j \bmod N_k = 4$ **then**
8. $K_j \leftarrow \mathcal{I} \cdot \begin{pmatrix} \mathrm{S_{RD}}(k_{0,j-1}) \\ \mathrm{S_{RD}}(k_{1,j-1}) \\ \mathrm{S_{RD}}(k_{2,j-1}) \\ \mathrm{S_{RD}}(k_{3,j-1}) \end{pmatrix} \oplus \mathcal{I} \cdot K_{j-N_k}$;
9. **else**
10. $K_j \leftarrow \mathcal{I} \cdot K_{j-1} \oplus \mathcal{I} \cdot K_{j-N_k}$;
11. **end if**
12. **end for**
13. **Return** $K_{N_k}, \ldots, K_{4N_r+3}$;

Algorithm 3 AES encryption.

Input: A 128-bit plaintext A and $N_r + 1$ round keys.
Output: A 128-bit ciphertext B.

1. **for** $j = 0$ to 3 **do**
2. $A_j \leftarrow \mathcal{I} \cdot A_j \oplus \mathcal{I} \cdot K_j$;
3. **end for**
4. **for** $i = 1$ to $N_r - 1$ **do**
5. **for** $j = 0$ to 3 **do**
6. $B_j \leftarrow \mathcal{M}_E \cdot \begin{pmatrix} \mathrm{S_{RD}}(a_{0,j}) \\ \mathrm{S_{RD}}(a_{1,(j+1) \bmod 4}) \\ \mathrm{S_{RD}}(a_{2,(j+2) \bmod 4}) \\ \mathrm{S_{RD}}(a_{3,(j+3) \bmod 4}) \end{pmatrix} \oplus \mathcal{I} \cdot K_{4i+j}$;
7. **end for**
8. $A \leftarrow B$;
9. **end for**
10. **for** $j = 0$ to 3 **do**
11. $B_j \leftarrow \mathcal{I} \cdot \begin{pmatrix} \mathrm{S_{RD}}(a_{0,j}) \\ \mathrm{S_{RD}}(a_{1,(j+1) \bmod 4}) \\ \mathrm{S_{RD}}(a_{2,(j+2) \bmod 4}) \\ \mathrm{S_{RD}}(a_{3,(j+3) \bmod 4}) \end{pmatrix} \oplus \mathcal{I} \cdot K_{4N_r+j}$;
12. **end for**
13. **Return** B;

encryption algorithm requires an internal 4×4 array of bytes B.

C. Decryption

We consider here the equivalent decryption algorithm described in [13, Section 3.7.3] (Algorithm 4). Its main advantage over the straightforward decryption process is that encryption and decryption rounds share the same datapath (Figure 2). Nevertheless, the round keys are introduced in reverse order for decryption.

III. The Hash Function ECHO

The ECHO family of hash functions [5] is built around the round function of the AES. This design strategy allows one to easily exploit advances in the implementation of the AES, such as the new AES instruction set of Intel Westmere processors [6]. ECHO is a family of four hash functions, namely ECHO-224, ECHO-256, ECHO-384, and ECHO-512 (Table II). The main differences lie in the length of the chaining variable and in the number of rounds.

In this work, we assume that our coprocessor is provided with a padded message M. We refer the reader to [5, Section 2.2] for a description of the padding step. A hardware wrapper

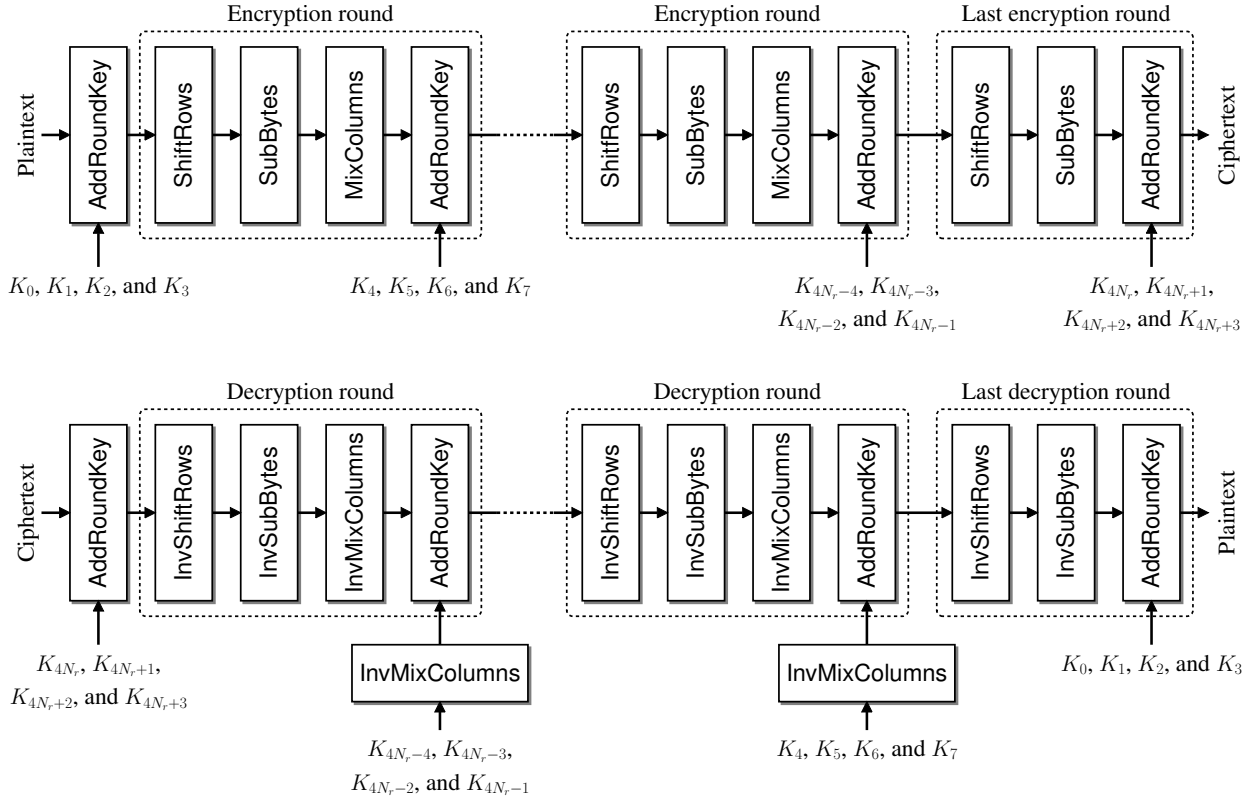


Fig. 2. AES encryption and decryption flowcharts.

Algorithm 4 AES decryption.

Input: A 128-bit ciphertext A and $N_r + 1$ round keys.
Output: A 128-bit plaintext B.

1. **for** $j = 0$ to 3 **do**
2. $\quad A_j \leftarrow \mathcal{I} \cdot A_j \oplus \mathcal{I} \cdot K_{4N_r+j}$;
3. **end for**
4. **for** $i = 1$ to $N_r - 1$ **do**
5. $\quad$ **for** $j = 0$ to 3 **do**
6. $$B_j \leftarrow \mathcal{M}_D \cdot \begin{pmatrix} \mathrm{S}_{\mathrm{RD}}^{-1}(a_{0,j}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{1,(j+3) \bmod 4}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{2,(j+2) \bmod 4}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{3,(j+1) \bmod 4}) \end{pmatrix} \oplus \mathcal{M}_D \cdot K_{4N_r-4i+j};$$
7. $\quad$ **end for**
8. $\quad A \leftarrow B$;
9. **end for**
10. **for** $j = 0$ to 3 **do**
11. $$B_j \leftarrow \mathcal{I} \cdot \begin{pmatrix} \mathrm{S}_{\mathrm{RD}}^{-1}(a_{0,j}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{1,(j+3) \bmod 4}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{2,(j+2) \bmod 4}) \\ \mathrm{S}_{\mathrm{RD}}^{-1}(a_{3,(j+1) \bmod 4}) \end{pmatrix} \oplus \mathcal{I} \cdot K_j;$$
12. **end for**
13. **Return** B;

interface for ECHO (and several other hash functions) comprising communication and padding is for instance described

TABLE II
PROPERTIES OF THE ECHO FAMILY OF HASH FUNCTIONS (REPRINTED FROM [5]). ALL SIZES ARE GIVEN IN BITS.

Algorithm	Chaining variable	Message block	Digest	Counter	Salt
ECHO-224	512	1536	224	64 or 128	128
ECHO-256	512	1536	256	64 or 128	128
ECHO-384	1024	1024	384	64 or 128	128
ECHO-512	1024	1024	512	64 or 128	128

in [4]. A padded message is divided into 1536-bit (ECHO-224 and ECHO-256) or 1024-bit (ECHO-384 and ECHO-512) message blocks M_1, M_2, ..., M_t that are iteratively processed using a compression function $\mathsf{Compress}_{512}$ (ECHO-224 and ECHO-256) or $\mathsf{Compress}_{1024}$ (ECHO-384 and ECHO-512).

The internal state S_i of the ECHO family can be viewed as a 4×4 array of 128-bit words (Figure 3), each of them being considered as an AES state $A^{(k)}$, $0 \leq k \leq 15$:

- ECHO-224/256. The 512-bit chaining variable V_{i-1} and the 1536-bit message block M_i, $1 \leq i \leq t$, are split into $N_v = 4$ and $N_m = 12$ AES states, respectively. V_{i-1} is stored in the first column of the internal state, and M_i in the remaining columns.
- ECHO-384/512. Both V_{i-1} and M_i are 1024-bit values that can be split into $N_v = N_m = 8$ AES states. V_{i-1} occupies the first half of the internal state and M_i the second one.

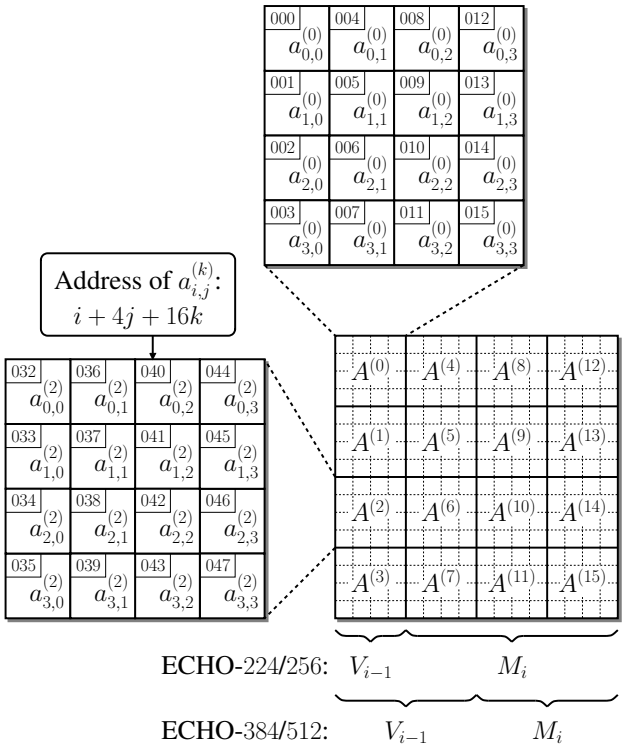


Fig. 3. Internal state of the ECHO family.

The initial chaining variable V_0 encodes the intended hash output size [5, Section 2.1].

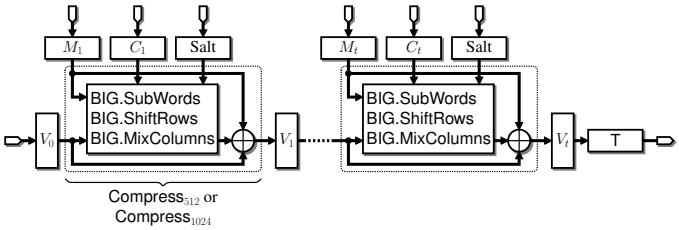


Fig. 4. Chained iteration of the compression function. T denotes the optional truncation described in [5, Section 3.5] and [5, Section 4.1].

ECHO applies iteratively a compression function to update the chaining variable V_i, $0 \leq i \leq t$ (Figure 4 and Algorithm 5). Compress$_{512}$ and Compress$_{1024}$ perform $N_r^{(\text{ECHO})} = 8$ and 10 iterations of BIG.Round, respectively. BIG.Round is the sequential composition of three transformations:

- The BIG.SubWords transformation applies two AES rounds to each 128-bit word $A^{(j)}$, $0 \leq j \leq 15$, of the internal state defined on Figure 3:

  $$A^{(j)} \leftarrow \mathsf{AESROUND}(\mathsf{AESROUND}(A^{(j)}, k_1), k_2),$$

  where AESROUND denotes one round of the AES encryption flow. As explained in Section II-B, an internal 4×4 array of bytes $B^{(j)}$ is needed to solve data dependency issues (Algorithm 5, lines 11 and 12).
  The key schedule for the derivation of the two 128-bit subkeys k_1 and k_2 is much simpler than the one of the AES. k_1 is related to the number of unpadded message bits C_i hashed at the end of the current iteration. An internal 64-bit counter κ is initialized with the value of C_i, and k_1 is defined as follows:

  $$k_1 = \kappa \parallel \underbrace{0 \ldots 0}_{64\times}.$$

  κ is incremented at the end of each AES round involving k_1. If the size of the message exceeds $2^{64} - 1$, one has the flexibility to use a 128-bit counter C_i. k_2 is equal to the 128-bit salt value that enables ECHO to support randomized hashing.
- The BIG.ShiftRows step is the analogue of the ShiftRows step of the AES. The first line of the internal state is left unchanged. Each 128-bit word of the second, third, and fourth lines is left-rotated by one, two, and three positions, respectively. At the byte level, this transformation is given by:

  $$\begin{pmatrix} b_{i,j}^{(4k)} \\ b_{i,j}^{(4k+1)} \\ b_{i,j}^{(4k+2)} \\ b_{i,j}^{(4k+3)} \end{pmatrix} \leftarrow \begin{pmatrix} a_{i,j}^{(4k)} \\ a_{i,j}^{((4k+5) \bmod 16)} \\ a_{i,j}^{((4k+10) \bmod 16)} \\ a_{i,j}^{((4k+15) \bmod 16)} \end{pmatrix},$$

  where $0 \leq i, j, k \leq 3$.
- The BIG.MixColumns step operates on the ECHO state column by column. We build a polynomial over $\mathbb{F}_{2^8}$ by picking the $(i + 4j)$th byte of each AES state in the kth column, and apply to it the MixColumns transformation:

  $$\begin{pmatrix} b_{i,j}^{(4k)} \\ b_{i,j}^{(4k+1)} \\ b_{i,j}^{(4k+2)} \\ b_{i,j}^{(4k+3)} \end{pmatrix} \leftarrow \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} a_{i,j}^{(4k)} \\ a_{i,j}^{(4k+1)} \\ a_{i,j}^{(4k+2)} \\ a_{i,j}^{(4k+3)} \end{pmatrix},$$

  where $0 \leq i, j, k \leq 3$. We combine the BIG.ShiftRows and BIG.MixColumns steps (Algorithm 5, line 18), and avoid data dependency issues thanks to intermediate variables $B^{(j)}$, $0 \leq j \leq 15$.

After $N_r^{(\text{ECHO})}$ calls to the compression function, the BIG.Final step generates the new value of the chaining variable V_i from V_{i-1}, M_i, and the internal state. Note that this step depends on the selected level of security (Algorithm 5, lines 26 to 34).

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

A. A Unified Arithmetic and Logic Unit

Since our objective is to develop a low-area coprocessor for the AES and the ECHO family of hash functions, it seems natural to consider an 8-bit datapath (Figure 5). Above all, note that 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 Arithmetic and Logic

Algorithm 5 The ECHO hash function.

Input: A chaining variable V (N_v 128-bit words), a message block (N_m 128-bit blocks), κ, and salt. AESROUND denotes an encryption round of the AES.

Output: A new chaining variable.

1. **for** $i = 0$ to $N_v - 1$ **do**
2. $\quad A^{(i)} \leftarrow V^{(i)}$;
3. **end for**
4. **for** $i = 0$ to $N_m - 1$ **do**
5. $\quad A^{(N_v+i)} \leftarrow M^{(i)}$;
6. **end for**
7. $k_1 \leftarrow \kappa \parallel \underbrace{0 \ldots 0}_{64\times}$;
8. $k_2 \leftarrow$ Salt;
9. **for** $r = 1$ to $N_r^{(\text{ECHO})}$ **do**
10. $\quad$ **for** $j = 0$ to 15 **do**
11. $\quad\quad B^{(j)} \leftarrow \mathsf{AESROUND}(A^{(j)}, k_1)$;
12. $\quad\quad A^{(j)} \leftarrow \mathsf{AESROUND}(B^{(j)}, k_2)$;
13. $\quad\quad k_1 \leftarrow (k_1 + 1) \bmod 2^{64}$;
14. $\quad$ **end for**
15. $\quad$ **for** $k = 0$ to 3 **do**
16. $\quad\quad$ **for** $i = 0$ to 3 **do**
17. $\quad\quad\quad$ **for** $j = 0$ to 3 **do**
18. $$\begin{pmatrix} b_{i,j}^{(4k)} \\ b_{i,j}^{(4k+1)} \\ b_{i,j}^{(4k+2)} \\ b_{i,j}^{(4k+3)} \end{pmatrix} \leftarrow \mathcal{M}_E \cdot \begin{pmatrix} a_{i,j}^{(4k)} \\ a_{i,j}^{((4k+5) \bmod 16)} \\ a_{i,j}^{((4k+10) \bmod 16)} \\ a_{i,j}^{((4k+15) \bmod 16)} \end{pmatrix};$$
19. $\quad\quad\quad$ **end for**
20. $\quad\quad$ **end for**
21. $\quad$ **end for**
22. $\quad$ **for** $j = 0$ to 15 **do**
23. $\quad\quad A^{(j)} \leftarrow B^{(j)}$;
24. $\quad$ **end for**
25. **end for**
26. **if** Algorithm is ECHO-224/256 **then**
27. $\quad$ **for** $i = 0$ to 3 **do**
28. $\quad\quad V^{(i)} \leftarrow V^{(i)} \oplus M^{(i)} \oplus M^{(i+4)} \oplus M^{(i+8)} \oplus A^{(i)} \oplus A^{(i+4)} \oplus A^{(i+8)} \oplus A^{(i+12)}$;
29. $\quad$ **end for**
30. **else**
31. $\quad$ **for** $i = 0$ to 7 **do**
32. $\quad\quad V^{(i)} \leftarrow V^{(i)} \oplus M^{(i)} \oplus A^{(i)} \oplus A^{(i+8)}$;
33. $\quad$ **end for**
34. **end if**
35. **Return** V;

Unit (ALU). We can now describe key expansion (Algorithms 1 and 2), encryption (Algorithm 3), and decryption (Algorithm 4) using a single instruction:

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

where

- R_i, R_j, and R_k are vectors of four bytes;
- f is a function applied to each byte of R_i;
- $\mathcal{A}$ and $\mathcal{B}$ are 4×4 matrices of bytes.

The values of these parameters for the different steps of Algorithms 1, 2, 3, and 4 are summarized in Table III. The hash function ECHO benefits from the same instruction: the BIG.SubWords consists of AES rounds, and the BIG.MixColumns step involves the circulant matrix $\mathcal{M}_E$. Only the key schedule and the BIG.Final step require a small additional amount of hardware.

1) The SubBytes and InvSubBytes Steps: The SubBytes and InvSubBytes steps are often considered as the most critical part of the AES and several architectures for S_{RD} and S_{RD}^{-1} have already been described in the literature (see for instance [20] for a comprehensive bibliography). On Xilinx Virtex-5 and Virtex-6 FPGAs, the best design strategy consists in implementing the AES S-boxes as 8-input tables [12]. Two control bits $ctrl_{1:0}$ allow us to perform SubBytes, InvSubBytes, or to bypass this stage when f is the identity function.

2) Matrix Multiplication: A quick look at Table III indicates that matrix $\mathcal{A}$ in Equation (1) can be any of the four matrices introduced in Section II. Two control bits $ctrl_{3:2}$ are therefore necessary to select the desired operation. Since we emphasize reducing the usage of FPGA resources, we adopt the multiply-and-accumulate approach proposed by Hämäläinen *et al.* [24], and need 4 clock cycles to multiply one column of the state or the round key array by a 4×4 circulant matrix (Figure 6). Let us consider the product $\mathcal{M}_E \cdot A_j$. We compute a first partial product by multiplying each coefficient of the fixed polynomial $01 + 01 \cdot y + 03 \cdot y^2 + 02 \cdot y^3$ by $a_{0,j}$, and store the result in registers r_0, r_1, r_2, and r_3. Then, at each clock cycle, the intermediate result is rotated and accumulated with a new partial product. This process involves a 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 two bits of a modulo-4 counter.

A standard way to implement the AES consists in taking advantage of the well-known relation between the MixColumns and InvMixColumns polynomials [13, p. 55]:

$$d(y) = (04y^2 + 05) \cdot c(y) \bmod (y^4 + 01).$$

However, multiplication by $04y^2 + 05$ would incur extra clock cycles for decryption (*i.e.* a different instruction flow for encryption and decryption). In order to keep the instruction memory of our coprocessor as small as possible, it is crucial to use the same code for encryption and decryption. A status register indicates which algorithm is currently executed, and the control unit generates the control bits $ctrl_{3:0}$ accordingly.

Our algorithm for multiplication by $\mathcal{M}_D$ is based on the following observation [29]:

$$\mathcal{M}_\mathcal{D} = \mathcal{M}_\mathcal{E} + \begin{pmatrix} 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \\ 0C & 08 & 0C & 08 \\ 08 & 0C & 08 & 0C \end{pmatrix}.$$

Table IVa defines the multiplication of an element $a(x) = \sum_{i=0}^{7} a_i x^i \in \mathbb{F}_{2^8}$ by 08 and 0C. Note that each line of the table involves at most 5 bits of $a(x)$ and can therefore be

TABLE III
IMPLEMENTATION OF AES KEY EXPANSION, AES ENCRYPTION, AES DECRYPTION, AND BIG.MIXCOLUMNS WITH A SINGLE INSTRUCTION.

Algorithm	Operation	R_k	$\mathcal{A}$	f	R_i	$\mathcal{B}$	R_j
Key expansion	Algorithm 1, line 4	K_i	$\mathcal{P}$	S_{RD}	K_{i-1}	$\mathcal{I}$	K_{i-N_k}
	Algorithm 1, line 8	K_i	$\mathcal{I}$	Identity	K_{i-1}	$\mathcal{I}$	K_{i-N_k}
	Algorithm 2, line 8	K_i	$\mathcal{I}$	S_{RD}	K_{i-1}	$\mathcal{I}$	K_{i-N_k}
AES encryption	Algorithm 3, line 2	A_j	$\mathcal{I}$	Identity	A_j	$\mathcal{I}$	K_j
	Algorithm 3, line 6	B_j	$\mathcal{M}_E$	S_{RD}	$\begin{pmatrix} a_{0,j} \\ a_{1,(j+1) \bmod 4} \\ a_{2,(j+2) \bmod 4} \\ a_{3,(j+3) \bmod 4} \end{pmatrix}$	$\mathcal{I}$	K_{4i+j}
	Algorithm 3, line 11	B_j	$\mathcal{I}$	S_{RD}	$\begin{pmatrix} a_{0,j} \\ a_{1,(j+1) \bmod 4} \\ a_{2,(j+2) \bmod 4} \\ a_{3,(j+3) \bmod 4} \end{pmatrix}$	$\mathcal{I}$	K_{4N_r+j}
AES decryption	Algorithm 4, line 2	A_j	$\mathcal{I}$	Identity	A_j	$\mathcal{I}$	K_{4N_r+j}
	Algorithm 4, line 6	B_j	$\mathcal{M}_D$	S_{RD}^{-1}	$\begin{pmatrix} a_{0,j} \\ a_{1,(j+3) \bmod 4} \\ a_{2,(j+2) \bmod 4} \\ a_{3,(j+1) \bmod 4} \end{pmatrix}$	$\mathcal{M}_D$	K_{4N_r-4i+j}
	Algorithm 4, line 11	B_j	$\mathcal{I}$	S_{RD}^{-1}	$\begin{pmatrix} a_{0,j} \\ a_{1,(j+3) \bmod 4} \\ a_{2,(j+2) \bmod 4} \\ a_{3,(j+1) \bmod 4} \end{pmatrix}$	$\mathcal{I}$	K_j
BIG.MixColumns	Algorithm 5, line 18	$\begin{pmatrix} b_{i,j}^{(4k)} \\ b_{i,j}^{(4k+1)} \\ b_{i,j}^{(4k+2)} \\ b_{i,j}^{(4k+3)} \end{pmatrix}$	$\mathcal{M}_E$	Identity	$\begin{pmatrix} a_{i,j}^{(4k)} \\ a_{i,j}^{((4k+5) \bmod 16)} \\ a_{i,j}^{((4k+10) \bmod 16)} \\ a_{i,j}^{((4k+15) \bmod 16)} \end{pmatrix}$	$\mathcal{I}$	$\begin{pmatrix} 00 \\ 00 \\ 00 \\ 00 \end{pmatrix}$

implemented by means of a LUT with 5 inputs and 2 outputs (*i.e.* a LUT6_2 primitive if we consider Virtex-5 or Virtex-6 FPGAs). A second table computes $(00 \cdot y^2 + 00 \cdot y^3) \cdot a(x)$, $(00 \cdot y^2 + 01 \cdot y^3) \cdot a(x)$, $(01 \cdot y^2 + 00 \cdot y^3) \cdot a(x)$, or $(03 \cdot y^2 + 02 \cdot y^3) \cdot a(x)$ according to the 2 control bits $ctrl_{3:2}$. Since the computation of each digit of $02 \cdot a(x)$ and $03 \cdot a(x)$ requires at most 3 coefficients of $a(x)$ (Table IVb), this operation can be implemented by means of 8 LUT6_2 primitives.

Figure 7 describes how we implement multiplication by $\mathcal{I}$, $\mathcal{M}_E$, $\mathcal{M}_D$, and $\mathcal{P}$ by combining the outputs of those tables. One easily checks that this circuit is equivalent to the one illustrated in Figure 6. In particular, note that the content of registers r_2 and r_3 is given by:

$$r_2 \leftarrow \begin{cases} 00 & \text{if } ctrl_{3:2} = 00, \\ 03 \cdot a(x) & \text{if } ctrl_{3:2} = 01, \\ (08 \oplus 03) \cdot a(x) = 0B \cdot a(x) & \text{if } ctrl_{3:2} = 10, \\ 01 \cdot a(x) & \text{if } ctrl_{3:2} = 11, \end{cases}$$

and

$$r_3 \leftarrow \begin{cases} 01 \cdot a(x) & \text{if } ctrl_{3:2} = 00, \\ 02 \cdot a(x) & \text{if } ctrl_{3:2} = 01, \\ (0C \oplus 02) \cdot a(x) = 0D \cdot a(x) & \text{if } ctrl_{3:2} = 10, \\ 00 & \text{if } ctrl_{3:2} = 11. \end{cases}$$

Our matrix multiplication unit involves 16 LUT6_2 primitives and 32 LUT6 primitives, resulting in a total requirement of 12 slices on a Virtex-5 FPGA. Compared to the MixColumns operator of ECHO-256 coprocessor described in [8], where only multiplication by $\mathcal{M}_E$ is needed, the hardware overhead amounts to 4 Virtex-5 slices. Matrix $\mathcal{B}$ is either the identity matrix $\mathcal{I}$ or the InvMixColumns matrix $\mathcal{M}_D$ (Table III). We followed a similar strategy to implement multiplication by $\mathcal{B}$.

3) Addition over $\mathbb{F}_{2^8}$: Figure 8 describes the component we designed to perform the AddRoundKey step. Since our matrix multiplication units output 4 bytes, we perform 4 additions over $\mathbb{F}_{2^8}$ in parallel and store the result in a shift register. This approach allows us to write data byte by byte in the register file. Here again, a simple modulo-4 counter controls the process: a new result is loaded during the first clock cycle, and then shifted in the three subsequent clock cycles.

The same component performs the additions involved in the round key derivation. However, additional hardware resources are needed to:

- initialize RC (Algorithm 1, line 1 and Algorithm 2, line 1);
- add RC to $k_{0,i}$ when the column index i is a multiple of N_k (Algorithm 1, line 5 and Algorithm 2, line 5);
- update RC (Algorithm 1, line 6 and Algorithm 2, line 6).

A multiplexer controlled by $ctrl_6$ selects the operand loaded in the register when the clock enable signal $ctrl_7$ is equal to 1: the initial value 01 or $x \cdot$ RC. When $i \bmod N_k = 0$, the control unit sets $ctrl_8$ to 1 so that RC is added to $k_{0,i}$.

Recall that the BIG.MixColumns step does not involve any round key addition (see Algorithm 5, line 18 and Table III). In order to use the same datapath for this operation, we add

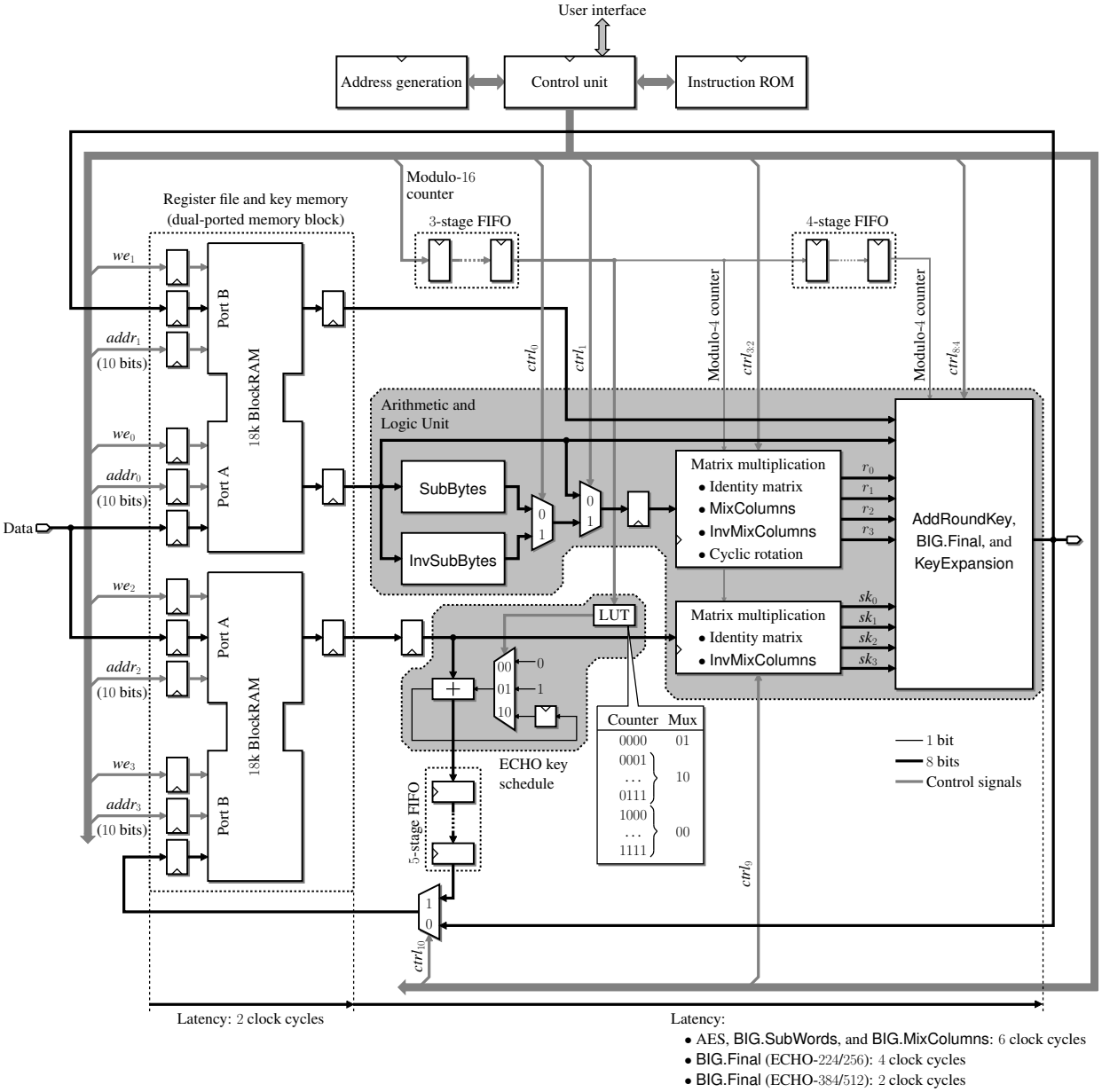
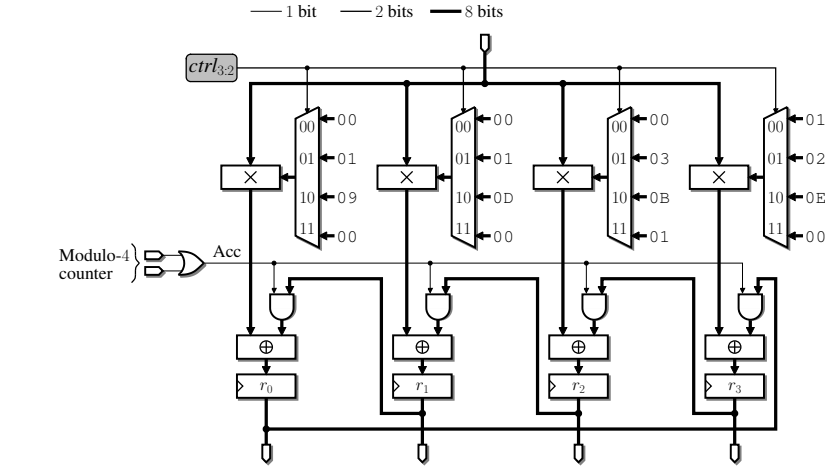


Fig. 5. General architecture of our unified 8-bit coprocessor for AES and ECHO.

the constant 00 stored in the key memory of the coprocessor.

During the BIG.Final step, two bytes are read from the register file at each clock cycle, and accumulated thanks to the feedback mechanism controlled by $ctrl_4$ and $ctrl_5$ (here again, the signal sk_0 is obtained by reading the constant 00 from the key memory). Thus, the computation of each byte of $V^{(i)}$ involves four and two clock cycles for ECHO-224/256 and ECHO-384/512, respectively. All other operations require six clock cycles (Figure 5). Therefore, special attention must be paid to the design of the control unit in order to take the latency of each operation into account.

4) ECHO key schedule: The choice of an 8-bit datapath enables to increment the internal 64-bit counter κ in 8 clock cycles, thus keeping the critical path of the adder as small as possible. Figure 5 describes the pipelined adder implementing ECHO key schedule. k_1 is stored in the key memory and is read byte by byte. During the first clock cycle, we add the constant 1 to the least significant byte of k_1 and store the output carry in a flip-flop. This carry bit is then added to the second byte of k_1, and the content of the flip-flop is updated accordingly. We repeat this process until the 64 least significant bits of k_1 are updated. Since the 8 most significant bytes of k_1 are not modified, we simply add the constant 0 in the remaining clock cycles. A modulo-16 counter and a small look-up table allow us to select the input carry of the 8-bit adder at each clock cycle.



$ctrl_{3:2}$	Operation
00	Identity: multiplication by $\mathcal{I} = \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix}$
01	MixColumns: multiplication by $\mathcal{M}_E = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$
10	InvMixColumns: multiplication by $\mathcal{M}_D = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$
11	Cyclic rotation: multiplication by $\mathcal{P} = \begin{pmatrix} 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \end{pmatrix}$

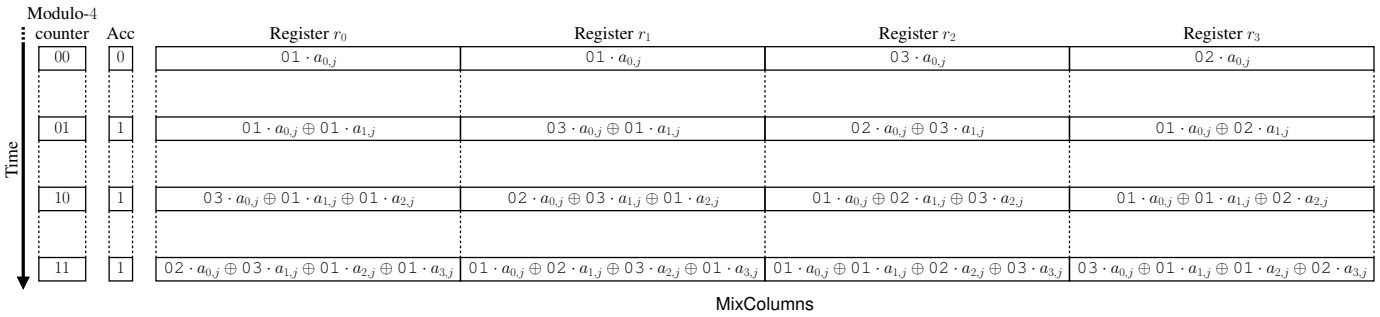


Fig. 6. Multiplication by a circulant matrix.

TABLE IV
MULTIPLICATION OVER $\mathbb{F}_{2^8}$ OF $a(x)$ BY SEVERAL CONSTANTS.

Inputs	$\times$08	$\times$0C
a_5, a_6	a_5	$a_5 \oplus a_6$
a_5, a_6, a_7	$(a_5 \oplus a_6)x$	$(a_5 \oplus a_7)x$
a_0, a_6, a_7	$(a_6 \oplus a_7)x^2$	$(a_0 \oplus a_6)x^2$
a_0, a_1, a_5, a_6, a_7	$(a_0 \oplus a_5 \oplus a_7)x^3$	$(a_0 \oplus a_1 \oplus a_5 \oplus a_6 \oplus a_7)x^3$
a_1, a_2, a_5, a_6, a_7	$(a_1 \oplus a_5 \oplus a_6)x^4$	$(a_1 \oplus a_2 \oplus a_5 \oplus a_7)x^4$
a_2, a_3, a_6, a_7	$(a_2 \oplus a_6 \oplus a_7)x^5$	$(a_2 \oplus a_3 \oplus a_6)x^5$
a_3, a_4, a_7	$(a_3 \oplus a_7)x^6$	$(a_3 \oplus a_4 \oplus a_7)x^6$
a_4, a_5	a_4x^7	$(a_4 \oplus a_5)x^7$

(a) Multiplication of $a(x)$ by 08 and 0C.

Inputs	$\times$01	$\times$02	$\times$03
a_0, a_7	a_0	a_7	$a_0 \oplus a_7$
a_0, a_1, a_7	a_1x	$(a_0 \oplus a_7)x$	$(a_0 \oplus a_1 \oplus a_7)x$
a_1, a_2	a_2x^2	a_1x^2	$(a_1 \oplus a_2)x^2$
a_2, a_3, a_7	a_3x^3	$(a_2 \oplus a_7)x^3$	$(a_2 \oplus a_3 \oplus a_7)x^3$
a_3, a_4, a_7	a_4x^4	$(a_3 \oplus a_7)x^4$	$(a_3 \oplus a_4 \oplus a_7)x^4$
a_4, a_5	a_5x^5	a_4x^5	$(a_4 \oplus a_5)x^5$
a_5, a_6	a_6x^6	a_5x^6	$(a_5 \oplus a_6)x^6$
a_6, a_7	a_7x^7	a_6x^7	$(a_6 \oplus a_7)x^7$

(b) Multiplication of $a(x)$ by 02 and 03.

B. Memory Organization

Since we consider an 8-bit datapath, the memory of our coprocessor is organized into bytes. We will show below that 10 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-5 or Virtex-6 block RAM configured as two independent 18 Kb RAMs (Figure 5).

a) Register file.: Recall that an ECHO state is an array of 256 bytes $a_{i,j}^{(k)}$, where $0 \leq i, j \leq 3$ and $0 \leq k \leq 15$ (Figure 3). Let us define the 8-bit address of $a_{i,j}^{(k)}$ as $16k+4j+i$ (*i.e.* the 4 most significant bits encode the index k, and the 4 least significant bits define the location of the byte in the AES state $A^{(k)}$). We decided to organize the register file into four blocks of 256 bytes selected by two additional address bits (Figure 9). In order to implement ECHO according to Algorithm 5, we need a first 4×4 array of AES states to store the chaining variable and the message block. The compression function involves two additional arrays (ECHO states A and B in Algorithm 5). We use the 128 least significant bits of A and B as intermediate variables for the AES. The key expansion algorithm computes K_i from K_{i-1} and K_{i-N_k} (Algorithms 1 and 2). In order to access a byte of K_{i-1} and K_{i-N_k} at each clock cycle, we keep two copies of the round keys. The first one is located in the 4th block of 256 bytes of the register file, and the second one is stored in the key memory. Since $N_r \leq 14$, we have to memorize at most $4N_r+4 = 60$ columns K_i, *i.e.* 240 bytes. The 8 least significant address bits of a round key byte $k_{i,j}$, $0 \leq i \leq 3$ and $0 \leq j \leq N_r \leq 14$, are given by $i + 4j$ (Figure 1).

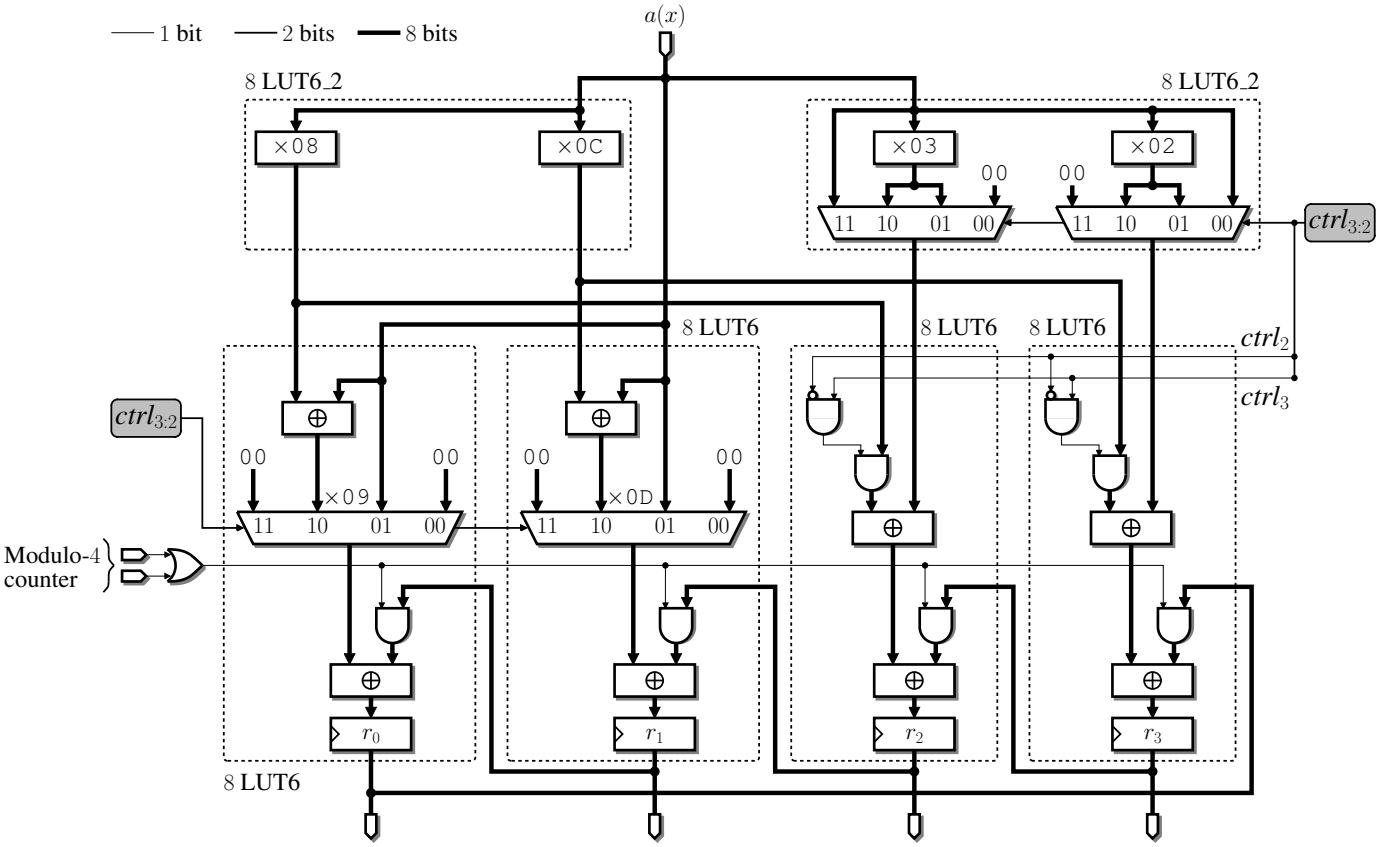


Fig. 7. Multiplication by $\mathcal{I}$, $\mathcal{M}_E$, $\mathcal{M}_D$, and $\mathcal{P}$.

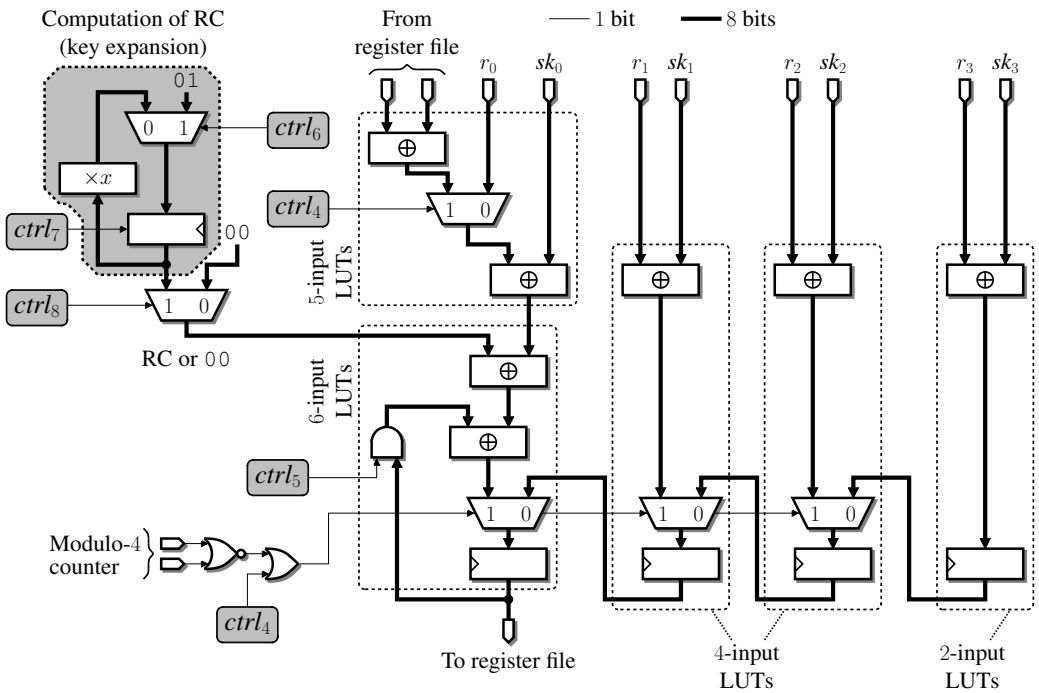
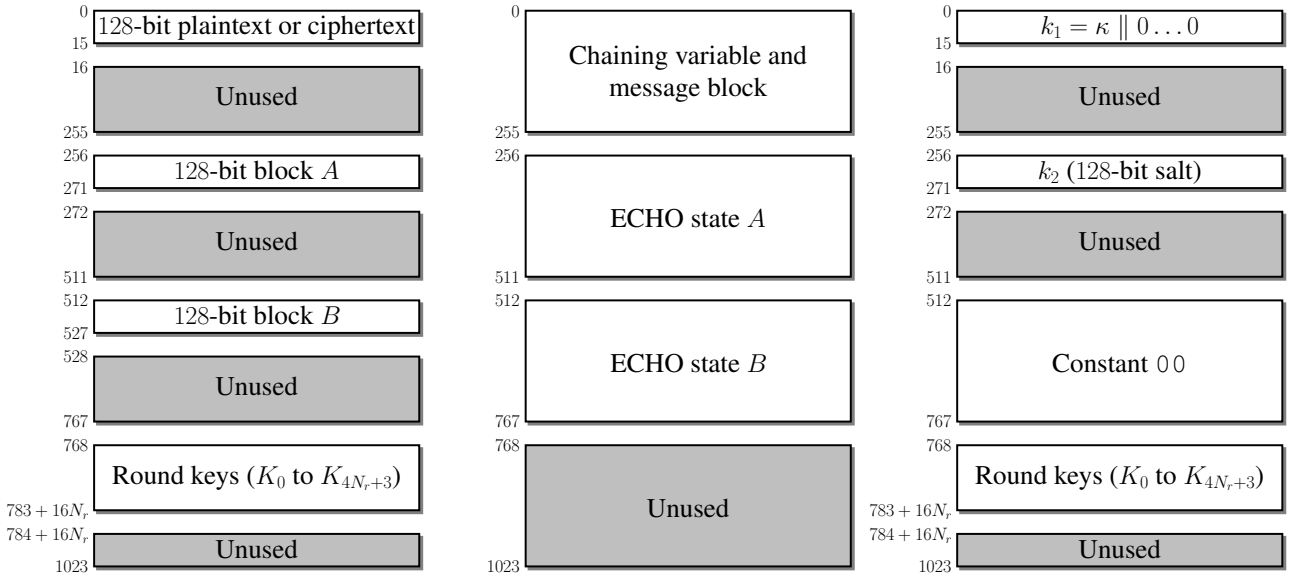


Fig. 8. Implementation of AddRoundKey, KeyExpansion, and BIG.Final.



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

Fig. 9. Memory organization.

b) Key memory.: Besides a copy of the AES round keys, the key memory contains k_1, k_2, and a block whose all bytes are set to zero which provides us with the constant 00 needed for the BIG.MixColumns and BIG.Final steps (Section IV-A3). Thus, no dedicated hardware is needed to force sk_0, sk_1, sk_2, and sk_3 to 00.

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 4-bit counter, an 8-bit counter, and a simple Finite State Machine (FSM).

C. Control Unit

The control bits of our unified 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. A FSM, four internal registers, and a stack allow us to select and execute the algorithm specified by the user.

1) Address Generation: The address generation process is the most challenging task in the design of a low-area unified coprocessor for the AES and the hash function ECHO: at first glance, it seems that each task (AES key expansion, AES encryption, AES decryption, BIG.MixColumns, etc.) requires a different addressing scheme. However, we described a way to generate the eight least significant bits of all read and write addresses of ECHO-256 by means of a counter by 5 modulo 16 and a modulo-256 counter [8]. We show here that our address generator can be slightly modified in order to support ECHO-512 and the AES (Figures 10 and 11). Note that our control unit generates at each clock cycle a read address and its corresponding write address. Since our coprocessor embeds several pipeline stages (Figure 5), it is necessary to delay write addresses and write enable signals accordingly. Shift registers allow us to synchronize signals in our coprocessor. On Xilinx devices, they are efficiently implemented by means of SRL16 primitives, whose depth is dynamically adjusted according to the algorithm being executed (Figure 10c): the latency of the BIG.Final step is equal to six and four clock cycles for ECHO-224/256 and ECHO-384/512, respectively. In all other cases, the datapath includes eight pipeline stages.

Figure 10 describes the generation of the write enable signals and the two most significant bits of read and write addresses. The architecture is fairly simple in the case of the key memory: two control bits $ctrl_{6:5}$ allows for selecting one of the four blocks of 256 bytes. For a given algorithm, read and write operations always occur in the same block and share the same two most significant address bits. Since the BIG.Final step does not modify the key memory, an 8-stage FIFO allows for synchronizing the write address and the write enable signal.

The register file needs a more careful attention. Recall that 128-bit plaintext or ciphertext blocks, chaining variables and message blocks are stored in the first block of 256 byte of the register file (Figure 9). The first intermediate variables are written in the second block. Thus, the two most significant bits of read and write addresses must be set to 00 and 01, respectively. This task is performed thanks to two multiplexers controlled by $ctrl_{10:9}$ and $ctrl_{8:7}$. Then, read and write operations alternate between the second and the third blocks of 256 bytes. It suffices to flip the bits of the write address. In the case of the read address, we wish to generate the sequence $00 \rightarrow 01 \rightarrow 10 \rightarrow 01 \rightarrow \ldots$. Let $a_{1:0}$ denote the two most significant bits of the current read address. We easily check that we obtain the next read address $b_{1:0}$ by computing $b_0 \leftarrow \bar{a}_0 \vee a_1$ and $b_1 \leftarrow a_0$. Of course it would have been possible to add a fourth input to the multiplexer controlled by $ctrl_{10:9}$ in order to set the read address to 01. Then, it suffices to flip the address bits to switch between the second

and the third memory block. However, this approach would imply two distinct instructions to switch from the first to the second block, and between the second and the third blocks, thus increasing the size of the instruction memory.

Figure 11 describes how we generate the eight least significant bits of read and write addresses (*i.e.* the location of a byte in a block of 256 bytes).

a) AES key schedule.: Figure 12 illustrates the scheduling of the AES-128 key expansion algorithm. Since $N_r \leq 14$, the round key array contains at most 240 bytes, and we can use the modulo-256 counter to process it byte by byte (Algorithm 1): a new byte $k_{j,i}$ of the array is computed from $k_{j,i-N_k}$ and $k_{j,i-1}$. Recall that the address of $k_{j,i-N_k}$ is given by $j + 4i - 4N_k$ and assume that it is provided by the modulo-256 counter. It suffices to increment the counter by $4 \cdot (N_k - 1)$ and $4N_k$ to obtain the addresses of $k_{j,i-1}$ and $k_{j,i}$, respectively. Our address generator is provided by $N_k - 1$ and a 6-bit adder allows us to increment the current value of the modulo-256 counter by $4 \cdot (N_k - 1)$ (Figure 11). Since

$$N_k = 2 \cdot (((N_k - 1) \text{ div } 2) + 1),$$

it suffices to add $8 \cdot (((N_k - 1) \text{ div } 2) + 1) = 4N_k$ to the address of $k_{j,i-N_k}$ in order to obtain the address of $k_{j,i}$ (5-bit adder on Figure 11).

b) AES encryption.: Recall that the ShiftRows step is implemented by accordingly addressing the register file (Section IV-A) and that the order in which bytes are processed during the first AddRoundKey step does not matter. In order to update a column of the AES state, we have to read $a_{0,j}$, $a_{1,(j+1) \bmod 4}$, $a_{2,(j+2) \bmod 4}$, and $a_{3,(j+3) \bmod 4}$, where $0 \leq j \leq 3$ (Algorithm 3). During an encryption round, the control unit performs the following tasks (Figure 13):

- Read a byte of the AES state from the register file. Starting from 0 (*i.e.* the address of $a_{0,0}$), we generate all read addresses thanks to a counter by 5 modulo 16.
- Read a byte of the round key from the key memory. The modulo-256 counter allows us to process the round key array column by column.
- Update one byte of the AES state. Since the AES state is updated column by column, the address is given by the 4 least significant bits of the modulo-256 counter.

In order to update the value of $a_{3,3}$, we have to provide our ALU with $a_{0,3}$, $a_{1,0}$, $a_{2,1}$, and $a_{3,2}$. Our control unit will generate the address of $a_{3,2}$ (read operation) and $a_{3,3}$ (write operation) at time t. Since our coprocessor includes $D = 8$ pipeline stages, we will write the new value of $a_{3,3}$ in the register file at time $t + D$ (Figure 14). Therefore, we have to wait $D-3 = 5$ clock cycles before starting the next encryption round. Then, we read $a_{0,0}$ at time $t + D - 2$, $a_{1,1}$ at time $t + D - 1$, $a_{2,2}$ at time $t + D$, and $a_{3,3}$ at time $t + D + 1$, thus satisfying constraints implied by data dependencies. Each encryption round requires $16 + D - 3 = 21$ clock cycles. It is possible to relax this constraint by interleaving two (or more) AES encryptions. However, this approach works only in the case of a chaining mode without output feedback or during the BIG.SubWords step of ECHO, where we process 16 AES states.

c) AES decryption.: Two simple modifications of the AES encryption addressing scheme allow us to decrypt a ciphertext block (Figure 15):

- In order to perform InvShiftRows instead of ShiftRows, it suffices to increment the modulo-16 counter by 13 instead of 5. Therefore, only the most significant bit of the offset depends on the algorithm.
- The 128-bit round keys must be introduced in reverse order: the jth step of decryption involves the $(N_r - j)$th round key ($0 \leq j \leq N_r$). Since the 16 bytes of round key j are stored from address $16j$ to $16j+15$ (Figure 1), we have to modify the four most significant bits of the address in order to perform decryption. Furthermore, N_r is always even, and the least significant bit of $N_r - j$ has the same value as the one of j. Thus, we can compute the three most significant bits of $N_r - j$ by means of three look-up tables addressed by j.

The control unit embeds an internal register that indicates which algorithm is executed. The most significant bit of the offset as well as the control signals of the multiplexers selecting the read address of the key memory depend only on the content of this register. Thanks to this design strategy, the instruction memory contains a single algorithm to perform either encryption or decryption.

d) ECHO.: Figure 16 describes the address generation process of ECHO. The only difference between BIG.SubWords and AES encryption is that we now have to process 16 AES states. The four most significant bits of the address are therefore given by the four most significant bits of the modulo-256 counter.

During the BIG.MixColumns step, we have to increment the read addresses by 80 modulo 256. Consider the read addresses of the BIG.SubWords state: it suffices to swap the first four bits with the last four bits in order to obtain a counter by 80 modulo 256 (since $80 = 16 \cdot 5$, we can re-use our counter by 5 modulo 16). One easily checks that the write addresses are obtained by swapping the first four bits with the last four bits of our modulo-256 counter.

The BIG.Final step requires careful attention: in order to speed up this operation, we read a byte of the chaining variable or of the message block on the first port of the register file, and a byte of the internal state (*i.e.* the output of the last round) on the second one. We describe this process on Figure 16 in the case of ECHO-224/256. Modifying the scheduling for ECHO-384/512 is straightforward.

2) Instruction Memory: We implemented two mechanisms in our control unit in order to keep the size of the instruction memory as small as possible:

- Nested loops. Consider for instance AES encryption: since the number of rounds N_r depends on the desired level of security, we need a loop instruction in order to share the same code between AES-128, AES-192, and AES-256. When encryption starts, the value of N_r

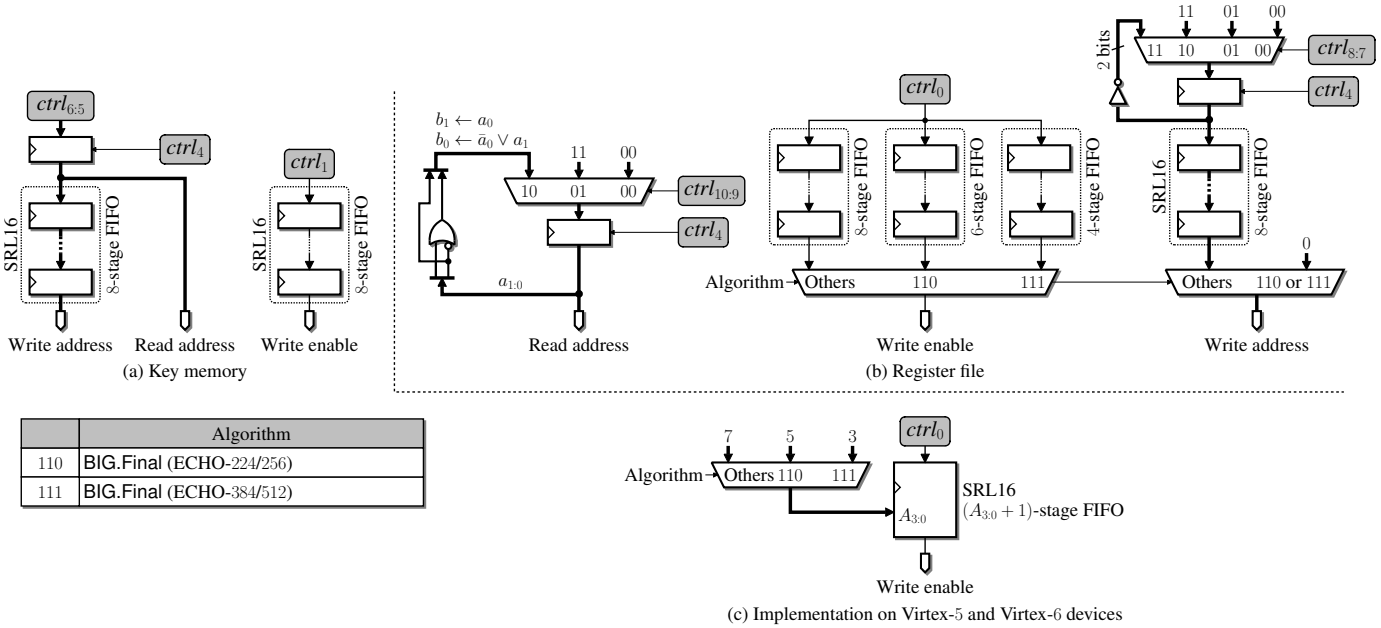


Fig. 10. Generation of the 2 most significant bits of read and write addresses, and generation of the write enable signals.

is loaded in one of the four internal registers of the control unit. The loop instruction will therefore include the address of the register. A nested loop is then needed to process all the columns of the AES state. The number of iterations is the same, regardless of the chosen security level, and can be specified in the loop instruction. Therefore, we implemented two addressing modes (absolute and register indirect). Each time a loop instruction is executed, the return address and the number of iterations are pushed onto a stack.

- Conditional branch. Compared to AES-128 and AES-192, the key expansion algorithm for AES-256 requires specific instructions to compute K_i when $i \bmod N_k = 4$ (Algorithm 2). Thanks to a conditional branch mechanism, we can write a single key expansion algorithm and skip the instructions specific to AES-256 when we target a lower level of security. Conditional instructions are also useful to select the code of the BIG.Final step of ECHO (*i.e.* lines 27 to 29 or lines 31 to 33 of Algorithm 5).

Thanks to these mechanisms, the instruction memory contains only 3 algorithms: AES key expansion (58 instructions), AES encryption/decryption (26 instructions), and ECHO (36 instructions).

V. Results and Comparisons

We captured our architecture in the VHDL language and prototyped our coprocessor on Virtex-5 and Virtex-6 FPGAs with average speedgrade. Table V and VI summarize the place-and-route results measured with ISE 12.3 and the throughput of each algorithm implemented, respectively. It is of course 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.).

TABLE V
Place-and-route results.

FPGA	Area [slices]	18k memory blocks	Frequency [MHz]
xc5vlx50-2	193	2	359
xc6vlx75t-2	155	2	397

A. Low-Resource AES Cores

Several articles describe AES cores built around an 8-bit datapath:

- Feldhofer *et al.* [18] have introduced a protocol based on the AES for authenticating an RFID tag to a reader device. The challenge was to propose a low-power AES-128 encryption core suitable for RFID tags. In order to keep the number of registers as small as possible, round keys are computed just in time by using the S-box and the XOR functionality of the datapath. The coprocessor needs 1016 clock cycles for the encryption of a 128-bit plaintext block (including key expansion). Our approach involves a smaller number of clock cycles, however it would be unfair to make a comparison between an architecture optimized for RFID tags (0.35 μm CMOS process) and a coprocessor taking advantage of the features of today's FPGA technology.
- Good and Benaissa [22], [23] have proposed an 8-bit Application Specific Instruction Processor (ASIP) for AES-128. They defined a minimal set of instructions to perform the operations required by the AES and the control unit mainly consists of a program ROM, an instruction decoder, and a program counter. Their coprocessor needs 122 Spartan-II slices and is therefore more compact than

	Algorithm
000	AddRoundKey
001	AES round (encryption or decryption)
010	Last AES round (encryption or decryption)
011	Key expansion

	Algorithm
100	BIG.SubWords
101	BIG.MixColumns
110	BIG.Final (ECHO-224/256)
111	BIG.Final (ECHO-384/512)

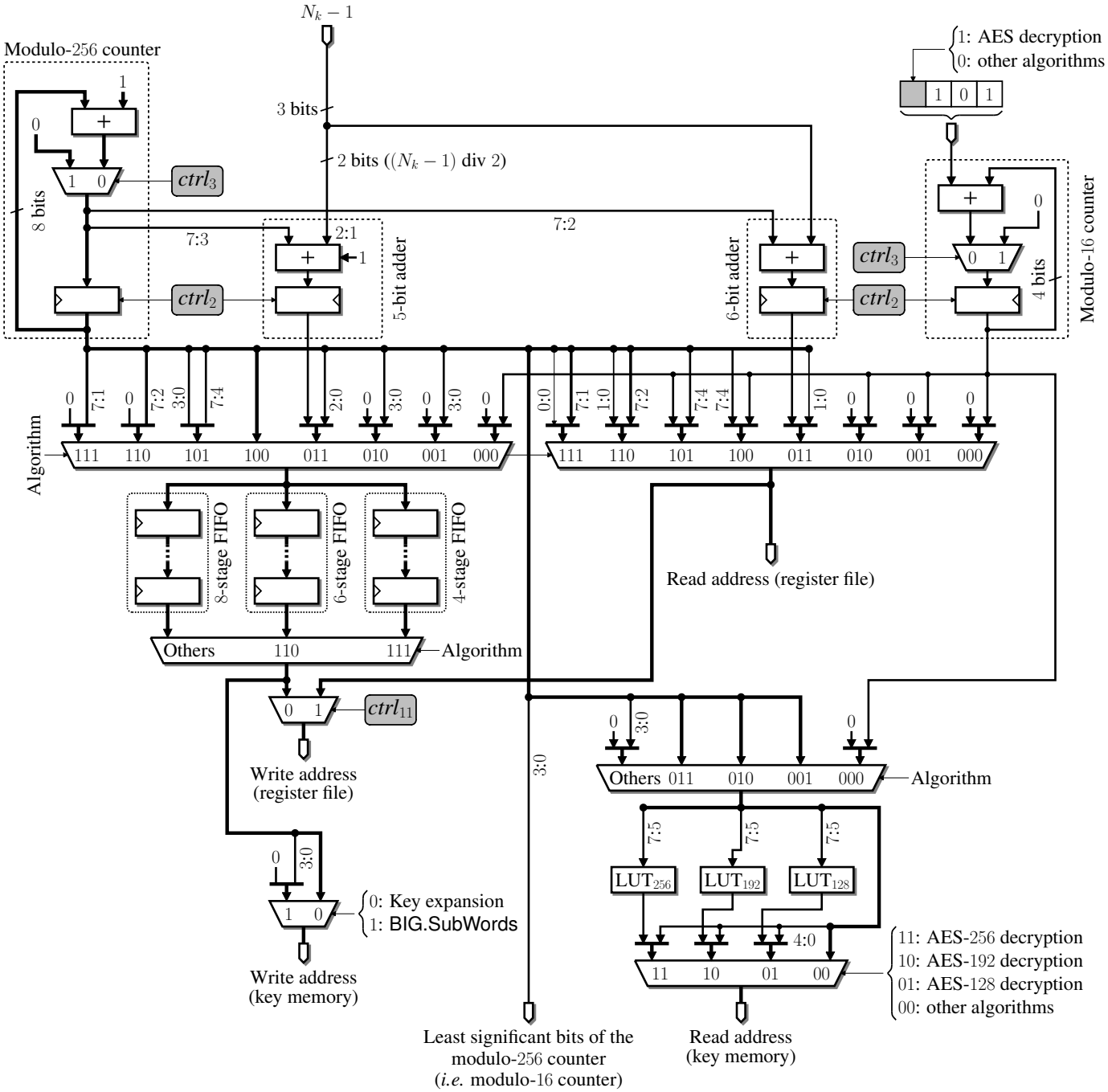


Fig. 11. Generation of the 8 least significant bits of read and write addresses.

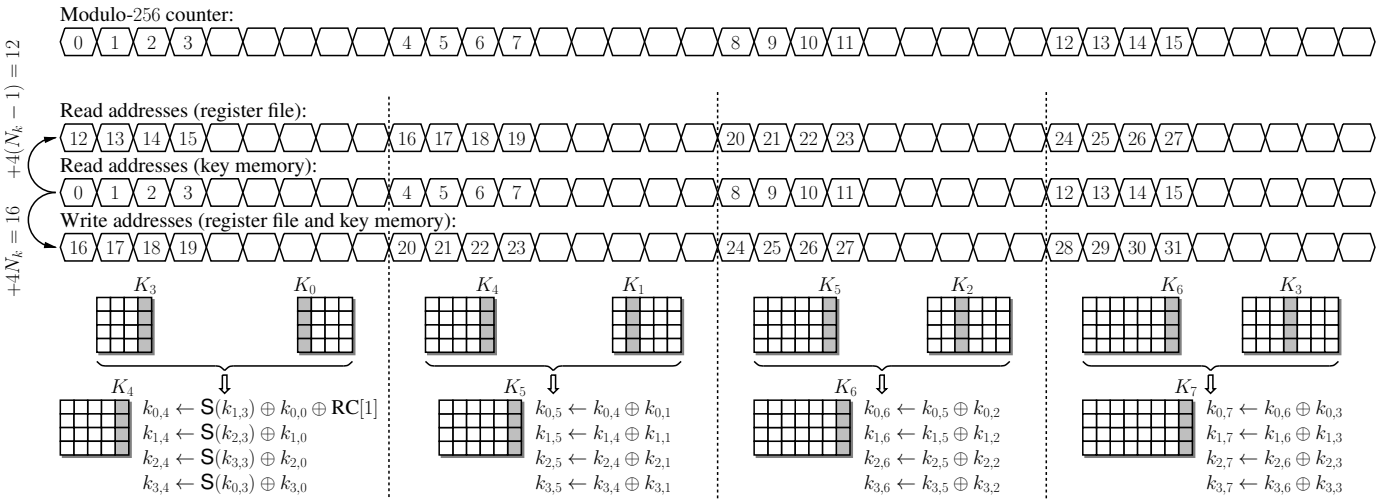


Fig. 12. Address generation during AES-128 key expansion.

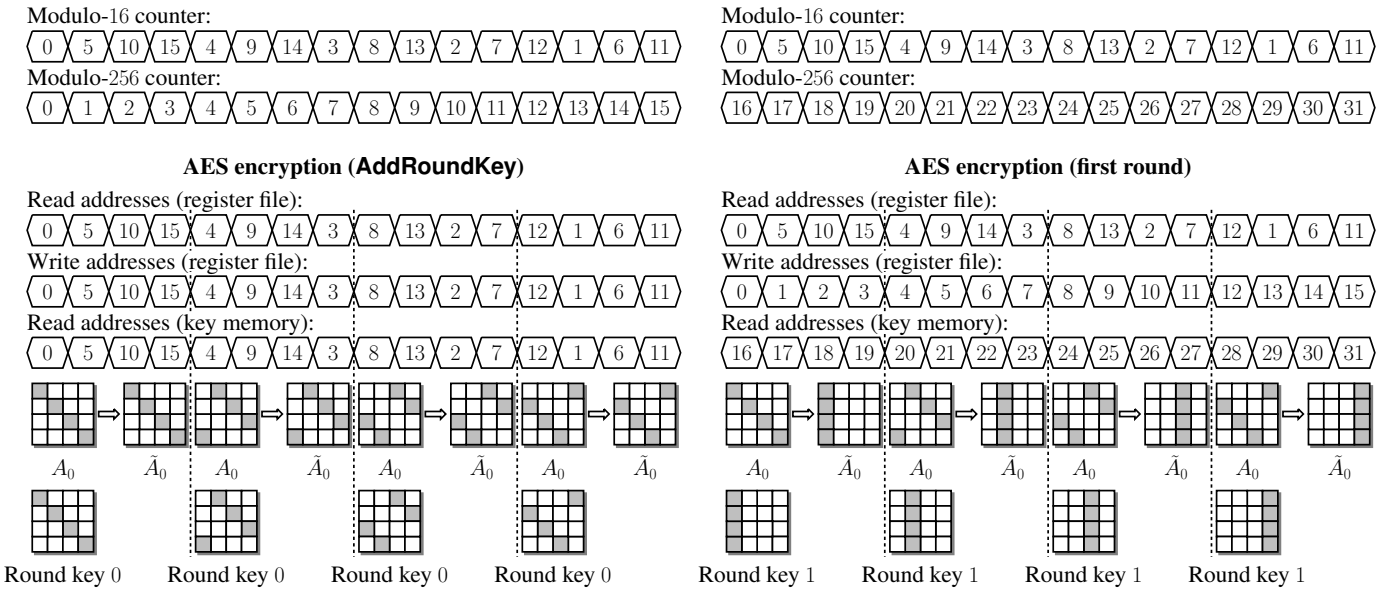


Fig. 13. Address generation during AES encryption.

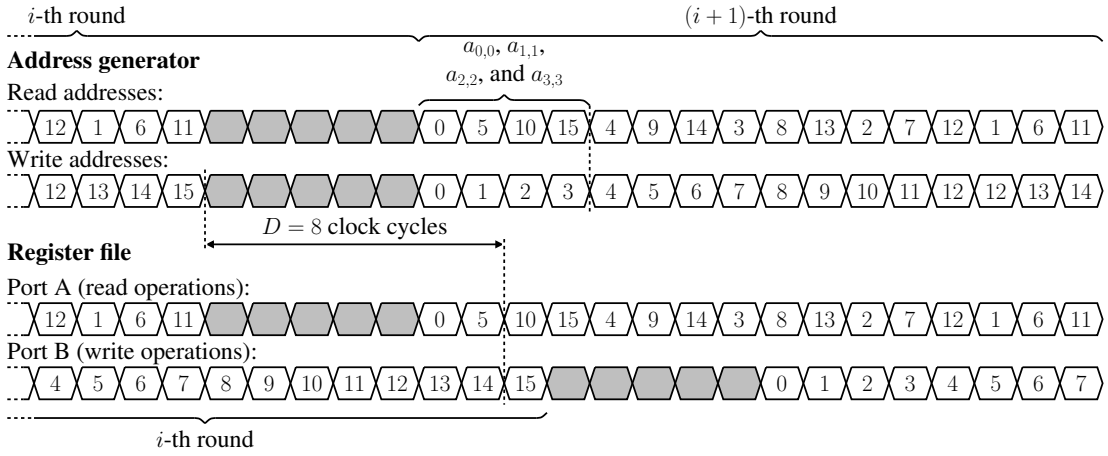


Fig. 14. Latency between two encryption rounds.

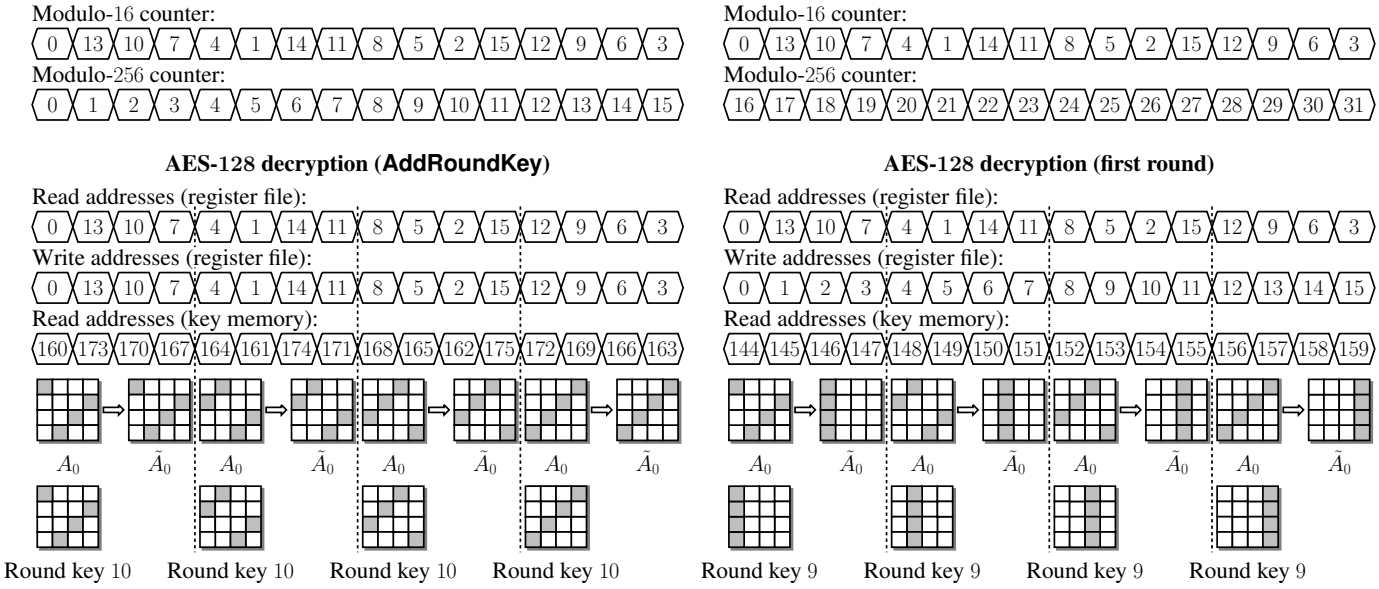


Fig. 15. Address generation during AES-128 decryption.

TABLE VI
TIMINGS ACHIEVED ON VIRTEX-5 AND VIRTEX-6 FPGAS.

Algorithm		# cycles	Throughput [Mbps]	
			Virtex-5	Virtex-6
AES-128	Key expansion	365	–	–
	Encryption/decryption	231	198.9	219.9
AES-192	Key expansion	421	–	–
	Encryption/decryption	273	168.3	186.1
AES-256	Key expansion	476	–	–
	Encryption/decryption	315	145.8	161.3
ECHO-256		6605	83.4	92.3
ECHO-512		8333	44.1	48.7

our architecture. The average throughput for encryption and decryption (including the key schedule that is performed on-the-fly) is equal to 2.18 Mbps (3691 clock cycles are needed to encrypt a 128-bit plaintext block). On a Virtex-5 FPGA, the same design would achieve much better performance: the clock frequency would be higher (Xilinx produces the Virtex-5 family in a 65 nm CMOS process, whereas the Spartan-II family was based on a 0.18 μm CMOS technology) and the number of slices would be roughly divided by two (a Virtex-5 slice contains four function generators configurable as 6-input LUTs or dual-output 5-input LUTS, whereas a Spartan-II slice includes only two 4-input LUTs). Therefore, the 8-bit ASIP should have a slightly better area–time trade-off than our coprocessor for short messages (according to Table VI, our coprocessor requires 596 clock cycles to perform the key expansion step and encrypt a 128-bit plaintext block). For long messages, our architecture should be a better choice.

- Hämäläinen *et al.* [24] have designed several AES-128 cores implementing encryption and key expansion. The throughput varies between 121 Mbps and 232 Mbps according to the optimization criterion (area, power, or speed). Since they have synthesized their core to gate level using a 0.13 μm standard-cell CMOS technology, it is again difficult to make a comparison between their work and our architecture.
- Helion Technology [28] is selling a tiny AES core that implements encryption, decryption, and key expansion at all levels of security. The coprocessor occupies only 97 Virtex-5 slices and achieves a throughput of 78 Mbps in the case of AES-128. The slice count is reduced to 88 on a Virtex-6 device, and the throughput of AES-128 is equal to 83 Mbps. Our coprocessor is 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 ECHO with the AES does not impact the overall performance of the latter.

B. *Low-Resource SHA-1 and SHA-2 Cores*

Table VII summarizes the result reported by Helion Technology [27] for their family of compact SHA-1 and SHA-2 cores on Virtex-6 FPGAs. The unified core for SHA-1,

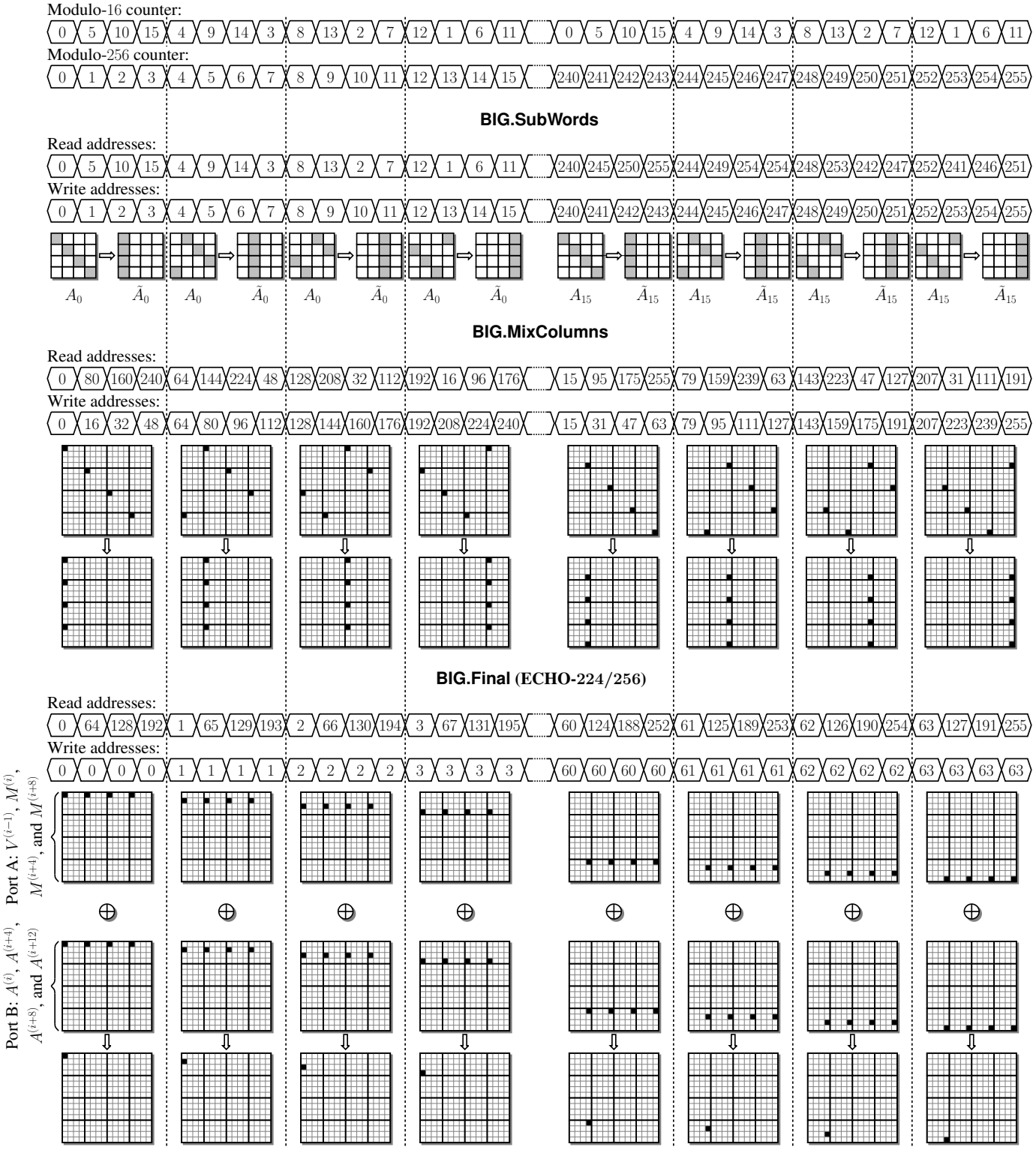


Fig. 16. Address generation during the BIG.SubWords, BIG.MixColumns, and BIG.Final steps.

SHA-224/256, and SHA-384/512 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. If we assume that the security of ECHO is at least as good as the one of SHA-2, ECHO is a clear winner for resource-constrained devices.

C. Round Two SHA-3 Candidates

A few researchers have proposed compact implementations of a subset of round two SHA-3 candidates. Table VIII provides the reader with a comparison of coprocessors optimized for Virtex-5 devices (note that BLAKE and Keccak have been selected as finalists in December 2010).

We have designed a low-area ALU for BLAKE and Blue Midnight Wish (BMW) on Xilinx devices [9]. Thanks to our approach, the BLAKE and BMW implementations reported in [9] and [15], respectively, rank among the smallest SHA-3 coprocessors. However, the datapath depends on the level of security one wishes to achieve: both algorithms involve arithmetic operations on 32-bit unsigned integers to produce 224- or 256-bit digests. The computation of 384- and 512-bit digests requires a 64-bit datapath. To the best of our knowledge, no one has proposed yet a low-area coprocessor for BLAKE or BMW providing the user with all levels of security. However, combining the 32 datapath with the 64 datapath will almost certainly increase the area (additional multiplexers to select the datapath, more complex control unit, etc.). Assuming that ECHO offers at least the same security guarantees as BLAKE or BMW, ECHO seems to be a better choice when all digest sizes and the AES are required.

Compared to the ECHO-256 coprocessor we described in [8], our new architecture also provides the user with ECHO-512 and the AES (encryption, decryption, and key expansion at all levels of security) at the cost of 66 slices. Thanks to a better pipelining, we also managed to achieve a slightly higher clock frequency in this work.

Shabal [11] ranks first in terms of throughput and area–time trade-off. Detrey *et al.* [14] noted that only a small fraction of the internal state of Shabal is used at any step of the algorithm. They exploited this fact and minimized the area of the circuit by taking advantage of the dedicated shift register resources available in the recent Xilinx devices (SRL16 primitive). Combined with a tiny AES core, Shabal is an excellent candidate for low-area implementations on Xilinx devices. However, porting this coprocessor to FPGAs that do not embed SRL16-like primitives might have an important impact on the overall performance. The architecture described in this work includes only a small number of SRL16 primitives in order to synchronize control signals. Therefore, it should be more portable than the Shabal coprocessor designed by Detrey *et al.* [14].

Several researchers provided the scientific community with comparisons of parallel architectures for the 14 round two SHA-3 candidates (see for instance [25]). The main criticism leveled at ECHO is its poor throughput to area ratio when compared to most of the round two SHA-3 candidates. Our results contradict previous studies: as long as compact implementations are concerned, ECHO offers for instance a better area–time trade-off than Keccak or BMW. When the coprocessor must offer several digest sizes and AES encryption/decryption, ECHO should also perform better than BLAKE.

VI. Conclusion

We described a low-area coprocessor for the AES (encryption, decryption, and key expansion) and the cryptographic hash function ECHO at all levels of security. Our architecture is built around an 8-bit datapath and the ALU performs a single instruction that allows for implementing both algorithms. Thanks to a careful organization of AES and ECHO internal states in the register file, the control unit remains simple, despite the various addressing schemes required for the different steps of the AES and ECHO: all read and write addresses are generated by means of a modulo-16 counter and a modulo-256 counter. Our results show that:

- At the cost of 66 slices, one can modify the ECHO-256 coprocessor we described in [8] in order to include ECHO-512 and the AES (encryption, decryption, and key expansion at all levels of security). Thanks to a better pipelining, the throughput of our novel architecture is even slightly improved.
- Our coprocessor improves the area–time product compared to the tiny AES core designed by Helion Technology [28]. Combining ECHO with the AES does not impact the overall performance of the latter.
- Assuming that the security guarantees of ECHO are at least as good as the ones of the SHA-3 finalists BLAKE and Keccak, ECHO is a better candidate for low-area cryptographic coprocessors.

Furthermore, we believe that the design strategy we proposed in this work can be applied to the SHA-3 finalist Grøstl [21]. We expect to obtain a much more compact unified coprocessor (AES and Grøstl) than the one described by Järvinen [26].

Acknowledgements

The authors would like to thank Francisco Rodríguez-Henríquez for his valuable comments.

References

[1] D.F. Aranha, J.-L. Beuchat, J. Detrey, and N. Estibals. Optimal Eta pairing on supersingular genus-2 binary hyperelliptic curves. Cryptology ePrint Archive, Report 2010/559, 2010.

[2] J.-P. Aumasson, L. Henzen, W. Meier, and R.C.-W. Phan. SHA-3 proposal BLAKE (version 1.3). Available at http://www.131002.net/blake, 2009.

[3] B. Baldwin, A. Byrne, M. Hamilton, N. Hanley, R.P. McEvoy, W. Pan, and W.P. Marnane. FPGA implementations of SHA-3 candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash. Cryptology ePrint Archive, Report 2009/342, 2009.

[4] 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.

TABLE VII
COMPACT IMPLEMENTATIONS OF SHA-1 AND SHA-2 ON A VIRTEX-6 DEVICE [27]. EACH COPROCESSOR EMBEDS A SINGLE 36K MEMORY BLOCK.

Algorithm(s)	Area [slices]	Frequency [MHz]	Throughput [Mbps]		
			SHA-1	SHA-224/256	SHA-384/512
SHA-1	81	298	74	–	–
SHA-224/256	110	277	–	65	–
SHA-1/224/256	149	256	63	60	–
SHA-1/224/256/384/512	243	273	67	64	46

TABLE VIII
COMPACT IMPLEMENTATIONS OF SHA-3 CANDIDATES ON VIRTEX-5 FPGAS.

	Algorithm	FPGA	Area [slices]	36k memory blocks	Frequency [MHz]	Throughput [Mbps]
Beuchat *et al.* [9]	BLAKE-32	xc5vlx50-2	56	2	372	225
Aumasson *et al.* [2]	BLAKE-32	xc5vlx110	390	–	91	575
Beuchat *et al.* [9]	BLAKE-64	xc5vlx50-2	108	3	358	314
Aumasson *et al.* [2]	BLAKE-64	xc5vlx110	939	–	59	533
El-Hadedy *et al.* [16]	BMW-256	xc5vlx110	84	2	116	28
El-Hadedy *et al.* [15]	BMW-256	xc5vlx110	51	3	141	68
El-Hadedy *et al.* [15]	BMW-512	xc5vlx110	105	3	115	112
Beuchat *et al.* [8]	ECHO-256	xc5vlx50-2	127	1	352	72
Bertoni *et al.* [7]	Keccak	xc5vlx50-3	448	–	265	52
Baldwin *et al.* [3]	Shabal	xc5vlx220-2	2307	–	222.22	1330
Feron and Francq [19]	Shabal	not specified	596	–	109	1142
Detrey *et al.* [14]	Shabal	xc5vlx30-2	153	–	256	2051

[5] 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.

[6] 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.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document (version 2.0). Available at http://keccak.noekeon.org, 2009.

[8] 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.

[9] 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.

[10] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In C. Boyd, editor, *Advances in Cryptology–ASIACRYPT 2001*, number 2248 in Lecture Notes in Computer Science, pages 514–532. Springer, 2001.

[11] E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.R. Reinhard, C. Thuillet, and M. Videau. Shabal, a submission to NIST's cryptographic hash algorithm competition. Available at http://www.shabal.com, 2008.

[12] 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.

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

[14] J. Detrey, P. Gaudry, and K. Khalfallah. A low-area yet performant FPGA implementation of Shabal. Cryptology ePrint Archive, Report 2010/292, 2010.

[15] M. El-Hadedy, D. Gligoroski, and S.J. Knapskog. Single core implementation of Blue Midnight Wish hash function on VIRTEX 5 platform. Available at http://tinyurl.com/3xhvx6c, October 2010.

[16] M. El-Hadedy, M. Margala, D. Gligoroski, and S.J. Knapskog. Resource-efficient implementation of Blue Midnight Wish-256 hash function on Xilinx FPGA platform. In *The Second SHA-3 Candidate Conference*, August 2010.

[17] N. Estibals. Compact hardware for computing the Tate pairing over 128-bit-security supersingular curves. In M. Joye, A. Miyaji, and A. Otsuka, editors, *Pairing-Based Cryptography–Pairing 2010*, Lecture Notes in Computer Science. Springer, 2010. To appear.

[18] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems–CHES 2004*, number 3156 in Lecture Notes in Computer Science, pages 357–370. Springer, 2004.

[19] R. Feron and J. Francq. FPGA implementation of Shabal: Our first results. Available at http://www.shabal.com, 2010.

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

[21] 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, 2008.

[22] T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems–CHES 2005*, number 3659 in Lecture Notes in Computer Science, pages 427–440. Springer, 2005.

[23] T. Good and M. Benaissa. Very small FPGA application-specific instruction processor for AES. *IEEE Transactions on Circuits and Systems–I: Regular Papers*, 53(7):1477–1486, July 2006.

[24] 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.

[25] E. Homsirikamol, M. Rogawski, and K. Gaj. Comparing hardware performance of fourteen round two SHA-3 candidates using FPGAs. Cryptology ePrint Archive, Report 2010/445, 2010.

[26] 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.

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

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

[29] J. Wolkerstorfer. An ASIC implementation of the AES-MixColumn operation. In P. Rössler and A. Döderlein, editors, *Proceedings of Austrochip 2001*, pages 129–132, 2001.

[30] J. Zhai, C.M. Park, and G.-N. Wang. Hash-based RFID security protocol using randomly key-changed identification procedure. In M. Gavrilova, O. Gervasi, V. Kumar, C.J. Kenneth Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, editors, *Computational Science and Its Applications–ICCSA 2006*, number 3983 in Lecture Notes in Computer Science, pages 296–305. Springer, 2006.