A Compact FPGA Implementation of the SHA-3 Candidate 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 architecture of the SHA-3 candidate ECHO for the Virtex-5 FPGA family. Our architecture is built around a 8-bit datapath. We show that a careful organization of the chaining variable and the message block in the register file allows one to design a compact control unit based on a 4-bit counter, an 8-bit counter, and a simple Finite State Machine.**

A fully autonomous implementation of ECHO on a Xilinx Virtex-5 FPGA requires 127 slices and a single memory block to store the internal state, and achieves a throughput of 72Mbps.

I. Introduction

We describe a compact architecture of the SHA-3 candidate ECHO, proposed by Benadjila *et al.* [1], on a Virtex-5 Field-Programmable Gate Array (FPGA). Such an implementation is for instance 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 [2]).

After a short introduction to the ECHO family of hash functions (Section II), we describe a compact coprocessor based on an 8-bit datapath (Section III). We have prototyped our architecture on a Xilinx Virtex-5 FPGA and discuss our results in Section IV.

II. Algorithm Specification

The ECHO family of hash functions [1] is built around the round function of the Advanced Encryption Standard (AES) [3]. 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 [4]. ECHO is a family of four hash functions, namely ECHO-224, ECHO-256, ECHO-384, and ECHO-512 (Table I). The main differences lie in the length of the chaining variable and in the number of rounds.

TABLE I
PROPERTIES OF THE ECHO FAMILY OF HASH FUNCTIONS (REPRINTED FROM [1]). 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 this work, we assume that our coprocessor is provided with padded messages M. We refer the reader to [1, Section 2.2] for a description of the padding step. A hardware wrapper interface for the SHA-3 candidates comprising communication and padding is described in [5]. 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, $\ldots$, 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).

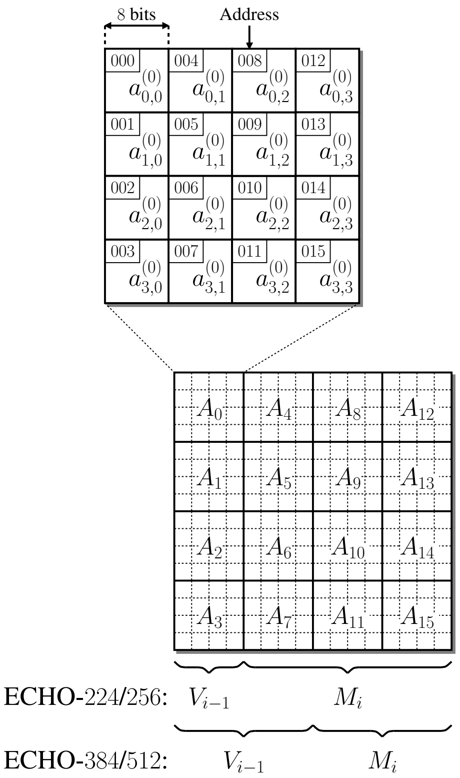


Fig. 1. Internal state of the ECHO family.

The internal state S_i of the ECHO family can be viewed as a 4×4 array of 128-bit words (Figure 1), each of them being considered as an AES state [3] (*i.e.* a 4×4 array of bytes).

- 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 4 and 12 128-bit words, respectively. V_{i-1} is stored in the

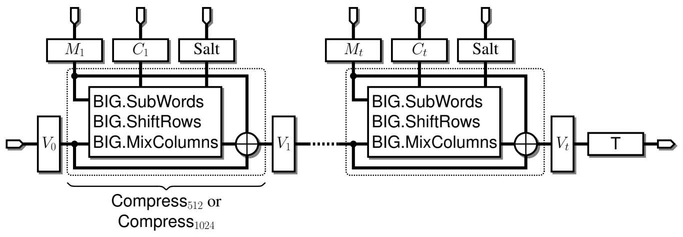


Fig. 2. Chained iteration of the compression function. T denotes the optional truncation described in [1, Section 3.5] and [1, Section 4.1].

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 8 128-bit words. V_{i-1} occupies the first half of the internal state and M_i the second one.

The initial chaining variable V_0 encodes the intended hash output size [1, Section 2.1].

ECHO applies iteratively a compression function to update the chaining variable V_i, $0 \leq i \leq t$ (Figure 2). Compress$_{512}$ and Compress$_{1024}$ perform eight and ten iterations of BIG.Round, respectively. BIG.Round is the sequential composition of:

1) BIG.SubWords. This transformation applies two AES rounds to each 128-bit word A_j, $0 \leq j \leq 15$, of the internal state defined on Figure 1:

$$\tilde{A}_j \leftarrow \mathsf{AESENC}(\mathsf{AESENC}(A_j, k_1), k_2),$$

where AESENC denotes one round of the AES encryption flow. 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.

2) BIG.ShiftRows. This operation 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.

3) BIG.MixColumns. The four 128-bit columns of the internal state can also be seen as 64 8-bit columns. The MixColumns operation of the AES is applied to each of them.

In this work, BIG.ShiftRows is implemented by accordingly addressing the register file, and this operation is therefore virtually for free. In the following, we will always combine the BIG.ShiftRows and BIG.MixColumns steps. The internal state is then updated as follows:

$$\left(\tilde{a}_{i,j}^{(k)}, \tilde{a}_{i,j}^{(k+1)}, \tilde{a}_{i,j}^{(k+2)}, \tilde{a}_{i,j}^{(k+3)}\right) \leftarrow \mathsf{MixColumns}\left(a_{i,j}^{(k)}, a_{i,j}^{((k+5) \bmod 16)}, a_{i,j}^{((k+10) \bmod 16)}, a_{i,j}^{((k+15) \bmod 16)}\right),$$

where $0 \leq i, j \leq 3$ and $k \in \{0, 4, 8, 12\}$. Since we focus on a compact coprocessor for ECHO, we will implement a single MixColumns unit and perform the BIG.MixColumns step sequentially. 16 calls to $\mathsf{MixColumns}\left(a_{i,j}^{(0)}, a_{i,j}^{(5)}, a_{i,j}^{(10)}, a_{i,j}^{(15)}\right)$, $0 \leq i, j \leq 3$, will update the first column of the state matrix. However, A_1, A_2, and A_3 are still involved in the forthcoming calls to MixColumns and we have to be careful not to overwrite their current values. A solution consists in having two blocks of 256 bytes in the memory: the operands are read from the first one and the results written in the second one. The role of both blocks is interchanged at the end of the BIG.MixColumns step (note that we apply the same strategy when performing the calls to AESENC during the BIG.SubWords steps).

After eight (Compress$_{512}$) or ten (Compress$_{1024}$) rounds, the BIG.Final step generates the new value of the chaining variable V_i from V_{i-1}, M_i, and the internal state. It is therefore necessary to keep a copy of V_{i-1} and M_i in the register file of the coprocessor. We give here the BIG.Final step of the compression function Compress$_{512}$ and refer the reader to [1, Section 4] for Compress$_{1024}$:

$$\begin{aligned}
V_i^{(0)} &= V_{i-1}^{(0)} \oplus M_i^{(0)} \oplus M_i^{(4)} \oplus M_i^{(8)} \oplus \\
&\quad A_0 \oplus A_4 \oplus A_8 \oplus A_{12}, \\
V_i^{(1)} &= V_{i-1}^{(1)} \oplus M_i^{(1)} \oplus M_i^{(5)} \oplus M_i^{(9)} \oplus \\
&\quad A_1 \oplus A_5 \oplus A_9 \oplus A_{13}, \\
V_i^{(2)} &= V_{i-1}^{(2)} \oplus M_i^{(2)} \oplus M_i^{(6)} \oplus M_i^{(10)} \oplus \\
&\quad A_2 \oplus A_6 \oplus A_{10} \oplus A_{14}, \\
V_i^{(3)} &= V_{i-1}^{(3)} \oplus M_i^{(3)} \oplus M_i^{(7)} \oplus M_i^{(11)} \oplus \\
&\quad A_3 \oplus A_7 \oplus A_{11} \oplus A_{15},
\end{aligned}$$

where $V_i = V_i^{(0)} \parallel V_i^{(1)} \parallel V_i^{(2)} \parallel V_i^{(3)}$ and $M_i = M_i^{(0)} \parallel \ldots \parallel M_i^{(11)}$.

Once one has a coprocessor for ECHO-256, writing a VHDL description of another member of the ECHO family is straightforward: it suffices to slightly modify the control unit in order to select the proper number of rounds and the BIG.Final operation. Therefore, we will only focus on ECHO-256 in the following.

III. A Compact Coprocessor for the ECHO Family of Hash Functions

Figure 3 describes our compact coprocessor based on an 8-bit datapath. To our best knowledge, the first 8-bit Application Specific Instruction Processor (ASIP) for the AES was proposed by Good and Benaissa [6]. 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. In this work, we show that a careful organization of the chaining variable V_i and the message block M_i in the register file allows one to design a control unit based on a 4-bit counter, an 8-bit counter, and a simple Finite State Machine (FSM). ShiftRows and BIG.ShiftRows operations are implemented by accordingly addressing the register file and do not require dedicated hardware. We slightly modified an AES encryption round in order to support the BIG.MixColumns and BIG.Final steps.

A. The SubBytes Step

The SubBytes step is the only non-linear transformation of the AES. Each byte $a_{i,j}$ of the state is considered as a polynomial belonging to $\mathbb{F}_{2^8}$. An S-Box computes the modular inverse of $a_{i,j}^{-1}$ (the value 00 is mapped onto itself) and then applies an affine transformation [3]. This step is often considered as the most critical part of the AES and several architectures for the S-Box have already been described in the literature (see for instance [7] for a comprehensive bibliography). On Xilinx Virtex-5 FPGAs, the best design strategy consists in implementing the S-Box as a large multiplexer controlled by $a_{i,j}$ [8]. An S-Box fits in only 32 6-input LUTs.

B. The MixColumns and BIG.MixColumns steps

Since the BIG.MixColumns step involves only MixColumns operations, a multiplexer allows us to bypass the S-Box. Recall that the MixColumns step is a permutation operating on the AES state column by column: each column is considered as a polynomial over $\mathbb{F}_{2^8}$ and multiplied modulo x^4+1 by $c(x) = 03\cdot x^3+01\cdot x^2+01\cdot x+02$ [3]. This operation is performed by multiplying each column of the AES state by the following circulant matrix (all operations are performed in $\mathbb{F}_{2^8} \cong \mathbb{F}_2[y]/(y^8+y^4+y^3+y+1)$):

$$\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

Since we emphasize reducing the usage of FPGA resources, we designed a Multiply-Accumulate unit and compute the above equation in four clock cycles (Figure 4). Therefore, we need 16 clock cycles to update the four columns of an AES state. A single control bit allows one to enable or disable the feedback loop.

C. The AddRoundKey and BIG.Final steps

Our MixColumns unit outputs four bytes that we store in a shift register. This approach allows us to write the result in the register file byte by byte. Since we focus on Virtex-5 FPGAs, we can take advantage of the 6-input LUT associated with each flip-flop and perform an optional AddRoundKey step without increasing the slice count (Figure 5). When the control bit $ctrl_5$ is set to 1, a subkey is combined with the AES state by means of a bitwise exclusive OR operation. Eight additional 6-input LUTs derive the new output value of the chaining variable. During the BIG.Final step, two bytes are read from the register file, added and accumulated thanks to the feedback mechanism enabled by the control bit $ctrl_3$.

Shift register LUTs with a clock enable (SRL16E primitive) store the subkeys while minimizing the number of slices of our coprocessor (Figure 6). The choice of a 32-bit datapath enables to provide the AddRoundKey with four 8-bit sukbeys sk_i, $0 \leq i \leq 3$ at each clock cycle, and to increment the internal counter κ in two clock cycles, thus keeping the critical path as short as possible.

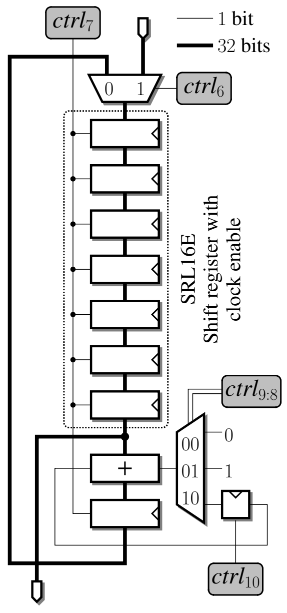


Fig. 6. Implementation of the key schedule. Control bits $ctrl_7$ and $ctrl_{10}$ denote clock enable signals.

D. Register File

Recall that we need to store three blocks of 256 bytes in the memory of the coprocessor. The first one stores the chaining variable V_{i-1} and the message block M_i needed to perform the BIG.Final step. Two further blocks are allocated to the internal state: during MixColumns operations, data are read from one of them and results are written into the other one. The total amount of memory is therefore 768 bytes, and a single block of dual-ported memory allows us to implement our register file on a Virtex-5 FPGA (RAMB18 primitive). The initial chaining variable V_0 and the message blocks M_i, $1 \leq i \leq t$, are written on port A. In order to avoid multiplexers (and to keep the size of the circuit as small as possible), the data computed during the BIG.SubWords, BIG.MixColumns, and BIG.Final steps are written on port B. Since our MixColumns unit processes a single input byte at each clock cycle, we need a single port to read data during the BIG.SubWords and BIG.MixColumns operations. However, in order to speed up the BIG.Final step, we read data from ports A and B.

Our control unit generates 10-bit addresses: the byte $a_{i,j}^{(k)}$, where $0 \leq k \leq 15$ and $0 \leq i,j \leq 3$, is stored at position $16k+$

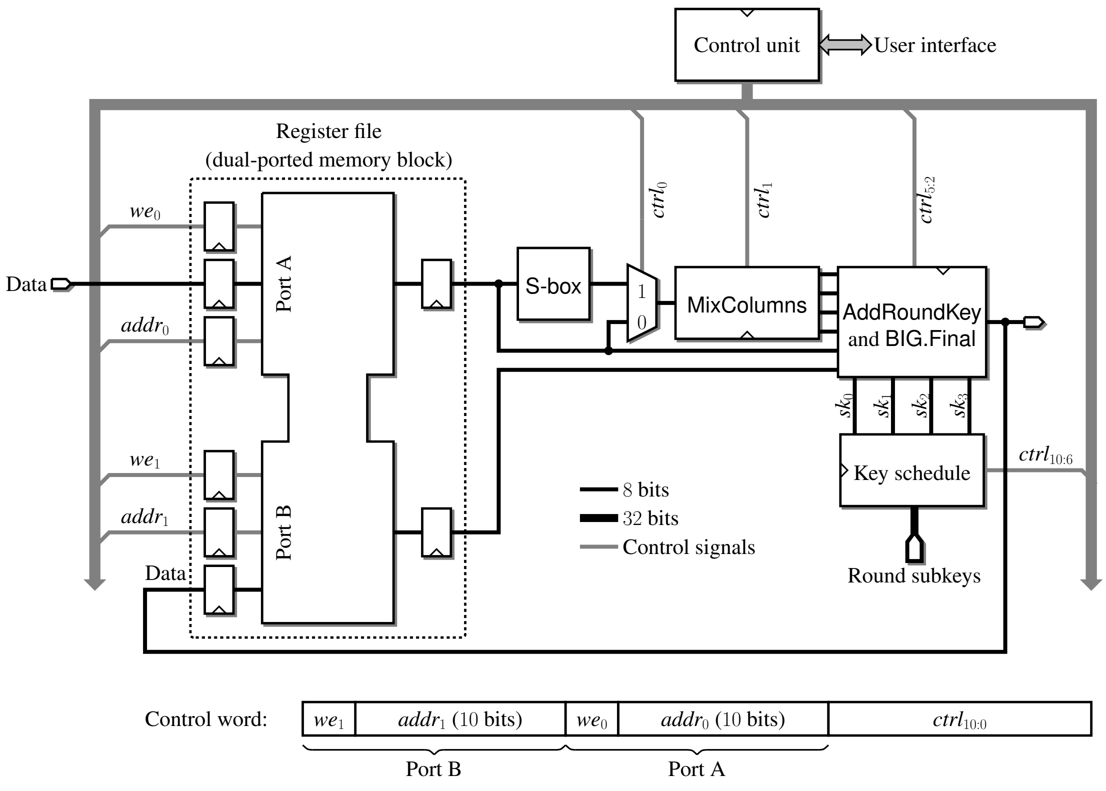


Fig. 3. General architecture of a 8-bit ECHO coprocessor.

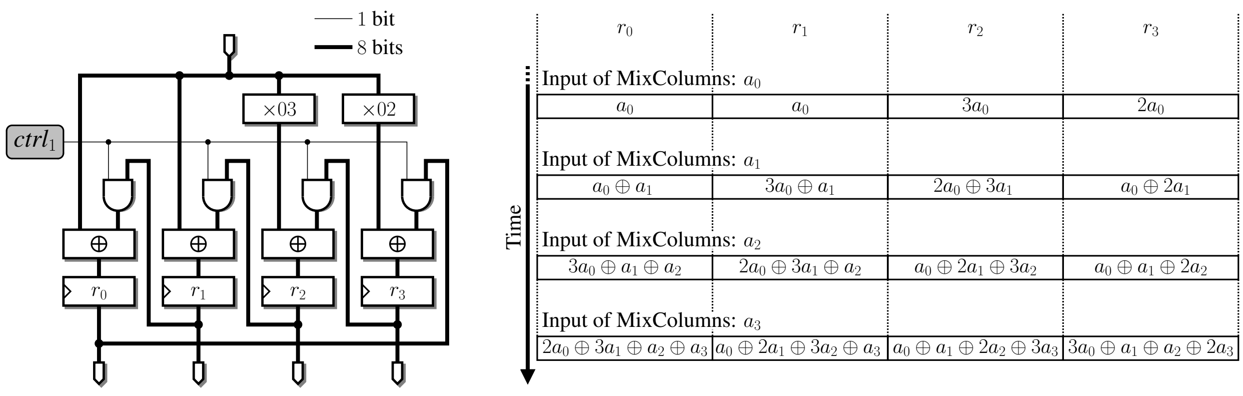

1 bit
8 bits
×03
×02
$ctrl_1$
r_0 r_1 r_2 r_3
Time
Input of MixColumns: a_0
a_0 | a_0 | $3a_0$ | $2a_0$
Input of MixColumns: a_1
$a_0 \oplus a_1$ | $3a_0 \oplus a_1$ | $2a_0 \oplus 3a_1$ | $a_0 \oplus 2a_1$
Input of MixColumns: a_2
$3a_0 \oplus a_1 \oplus a_2$ | $2a_0 \oplus 3a_1 \oplus a_2$ | $a_0 \oplus 2a_1 \oplus 3a_2$ | $a_0 \oplus a_1 \oplus 2a_2$
Input of MixColumns: a_3
$2a_0 \oplus 3a_1 \oplus a_2 \oplus a_3$ | $a_0 \oplus 2a_1 \oplus 3a_2 \oplus a_3$ | $a_0 \oplus a_1 \oplus 2a_2 \oplus 3a_3$ | $3a_0 \oplus a_1 \oplus a_2 \oplus 2a_3$

Fig. 4. Implementation of MixColumns.

$4j + i$ in the the block specified by the two most significant bits of the address (Figure 1).

E. Control Unit

Our control unit generates the eleven control bits *ctrl*$_{10:0}$ of the processing units, and addresses *addr*$_0$ and *addr*$_1$. It mainly consists of a simple FSM (13 states), a 3-bit round counter, and two specific counters that output the eight least significant bits of the read and write addresses (*i.e.* the position of the byte $a_{i,j}^{(k)}$ in a block of 256 bytes; the two most significant bits defining in which block we read/write the data are easily computed by the FSM). We will focus here on the address generation process, which was the most challenging task in the design of our compact architecture: the addressing schemes of the BIG.SubWords, BIG.MixColumns, and BIG.Final steps seem very different at first. However, we found a way to generate all read and write addresses with only one counter by 5 modulo 16 and one modulo-256 counter.

Since we combine ShiftRows and MixColumns operations during the BIG.SubWords step, we compute:

$$\left(\tilde{a}_{0,j}^{(k)}, \tilde{a}_{1,j}^{(k+1)}, \tilde{a}_{2,j}^{(k+2)}, \tilde{a}_{3,j}^{(k+3)}\right) \leftarrow \mathsf{MixColumns}\left(a_{0,j}^{(k)}, a_{1,(j+1) \bmod 4}^{(k)}, a_{2,(j+2) \bmod 4}^{(k)}, a_{3,(j+3) \bmod 4}^{(k)}\right),$$

where $0 \le j \le 3$ and $0 \le k \le 15$. Let us consider the four least significant bits of the address of $a_{0,j}^{(k)}$ (they give the

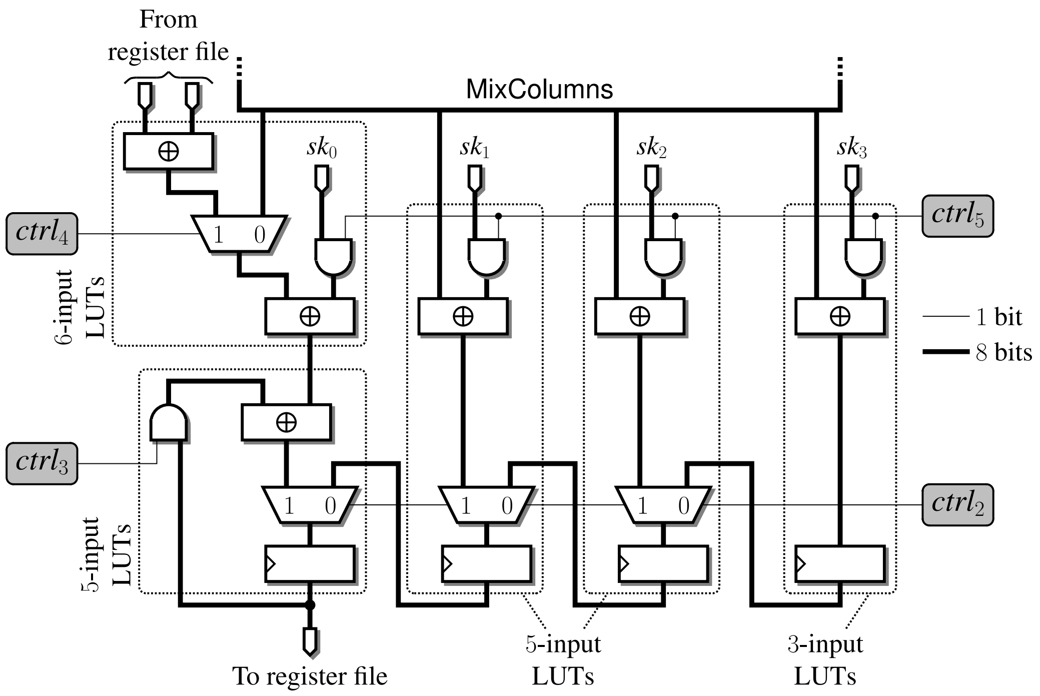


Fig. 5. Implementation of AddRoundKey and BIG.Final.

position of a $a_{0,j}^{(k)}$ in the AES state A_k). We check that we have to increment this address by 5 modulo 16 in order to obtain the four least significant bits of the read addresses of $a_{1,j}^{(k)}$, $a_{2,j}^{(k)}$, and $a_{3,j}^{(k)}$ (Figure 7a). Since a MixColumns operation takes 16 clock cycles, we have to increment the four most significant bits of the read address every 16 clock cycles. It suffices to consider the four most significant bits of a modulo-256 counter to achieve this task. Recalll that MixColumns steps update each AES state A_i, $0 \leq i \leq 15$, column by column. Thus, thanks to our organization of the bytes $a_{i,j}^{(k)}$ in the memory, the same modulo-256 counter allows us to generate write addresses.

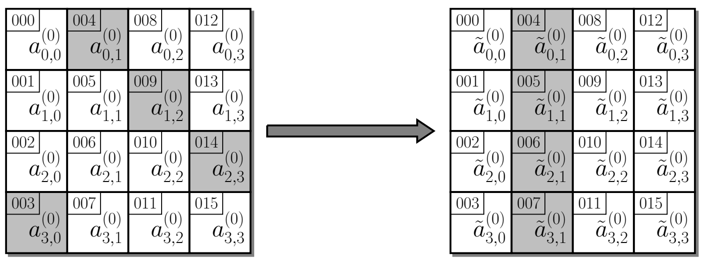
During the BIG.MixColumns step, we have to increment the read addresses by 80 modulo 256 (Figure 7b). Since $80 = 5 \cdot 16$, we can re-use our counter by 5 modulo 16 to compute the four most significant bits of the read address. The write address is incrementd by 16 modulo 256 at each clock cycles. The four least significant bit of the modulo 256 counter allow us to perform this operation. After 16 clock cycles, these counters go back to their initial value and we have to increment the read and write addresses by one. Thus, the four most significant bits of the modulo-256 counter provide us with the four least significant bits of these addresses.

Recall that we read two bytes at each clock cycle during the BIG.Final step. Let us denote by $B^{(k)}$, $0 \leq k \leq 15$, the 128-bit words of the 4×4 array storing the chaining variable V_{i-1} and the message block M_i. In order to update a byte of the chaining variable, we compute the bitwise exclusive OR of $a_{i,j}^{(k)}$, $a_{i,j}^{(k+4)}$, $a_{i,j}^{(k+8)}$, $a_{i,j}^{(k+12)}$, $b_{i,j}^{(k)}$, $b_{i,j}^{(k+4)}$, $b_{i,j}^{(k+8)}$, and $b_{i,j}^{(k+12)}$, where $0 \leq k \leq 3$. Starting from the address of $a_{i,j}^{(k)}$ (or $b_{i,j}^{(k)}$ since we only consider the address of a byte in an array of 256 bytes), we have to increment it by 64 at each clock cycles. The least significant bit is incremented by one every four clock cycles in order to process the next byte of the internal state. Since we overwrite V_{i-1}, the generation of write addresses is straightforward during the BIG.Final step.

These observations allow for the design of a compact address generator for our coprocessor (Figure 8). It mainly consists of a counter by 5 modulo 16 and a modulo-256 counter. Two multiplexers select read and write addresses according to two control bits specifying the current operation (BIG.ShiftRows, BIG.MixColumns, or BIG.Final). Recall that our coprocessor embeds several pipeline stages (Figure 3):

- The read address is stored in an internal register of the dual-ported memory block.
- The data at the specified address is stored in the output register on port A.
- The MixColumns operations is performed in an iterative fashion and requires four clock cycles.
- Our AddRoundKey unit stores its outcome in a shift-register before sending it to the register file.

Consequently, the write operation occurs seven clock cycles after the corresponding read operation. It is therefore necessary to delay write addresses by seven clock cycles. Figure 9 describes our address generation scheme. For each operation, we have to generate 256 read and write addresses (except in the BIG.final step where only 64 bytes are written in the register file). Since our two counters are incremented modulo 16 and 256, respectively, they automatically return to their initial state at the end of the process, and we can start a new operation without introducing pipeline bubbles (one easily checks that


(a) ShiftRows and MixColumns

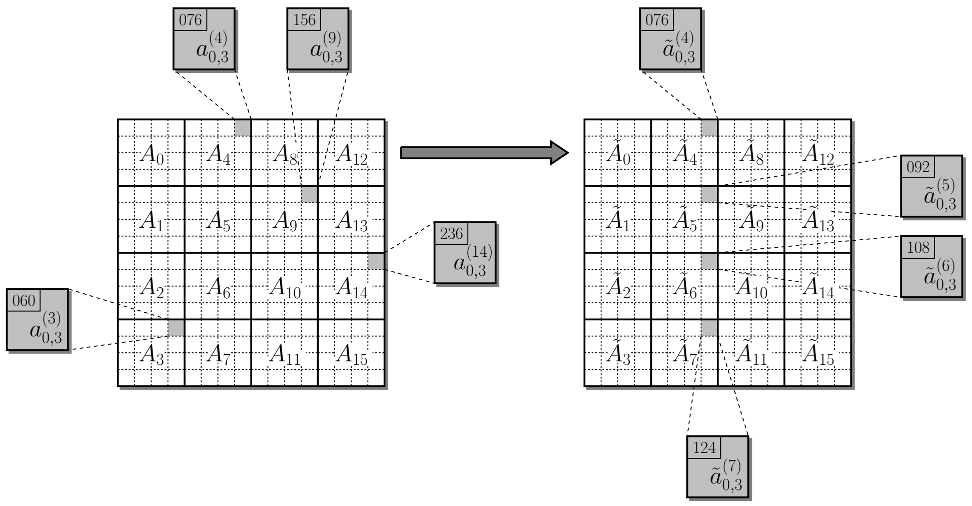

(b) BIG.ShiftRows and BIG.MixColumns

Fig. 7. Address generation during BIG.SubWords and BIG.MixColumns.

our address generation scheme avoids memory collisions). Each round involves two MixColumns steps (BIG.SubWords) and a BIG.MixColumns step. Therefore, the total number of clock cycles for eight rounds is equal to $8 \cdot 3 \cdot 256 = 6144$.

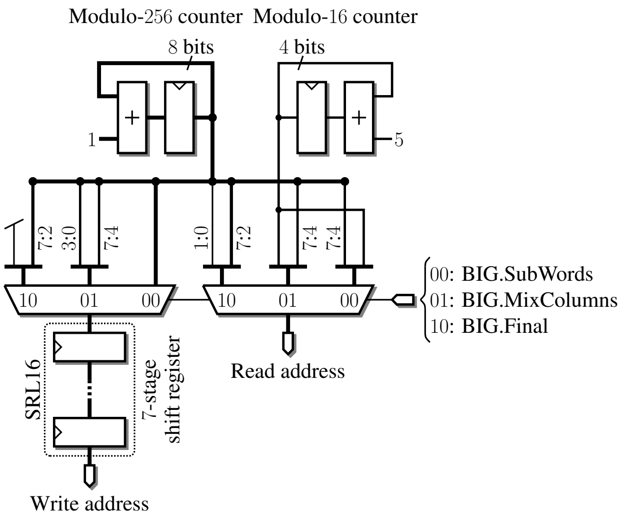


Fig. 8. Address generator based on a modulo-16 counter and a modulo-256 counter.

The BIG.Final step requires careful attention: in order to speed up this operation, we read a byte of V_{i-1} or M_i on port A, and a byte of the internal state (*i.e.* the output of the eighth round) on port B at each clock cycle. Due to the internal pipeline stages of our architecture, the computation of a byte of V_i also requires seven clock cycles. However, we have to wait for the end of the write cycle before processing the next byte and the total number of clock cycles is given by $64 \cdot 7 + 1 = 449$ (one additional cycle is required to complete the last write cycle and reset the FSM before processing the next compression function).

IV. Results, Comparisons, and Future Work

We captured our architectures in the VHDL language and prototyped our coprocessors on a Virtex-5 FPGA with average speedgrade. Table II summarizes our place-and-route results. A bunch of articles about ASIC and FPGA implementations of ECHO are available in the SHA-3 Zoo [9]. However, most of them focus on high-speed parallel implementations [10]–[13], and it is difficult to compare our work against such designs. To the best of our knowledge, the only compact coprocessor reported in the literature is the one by Lu *et al.* [12]. Unfortunately for our comparison, they only describe an ASIC implementation.

A few researchers proposed compact implementations of other SHA-3 candidates on Virtex-5 FPGAs (see for instance [9]). In terms of slice count, ECHO is one of the best candidates: only BLAKE-32 allows for the design of a more

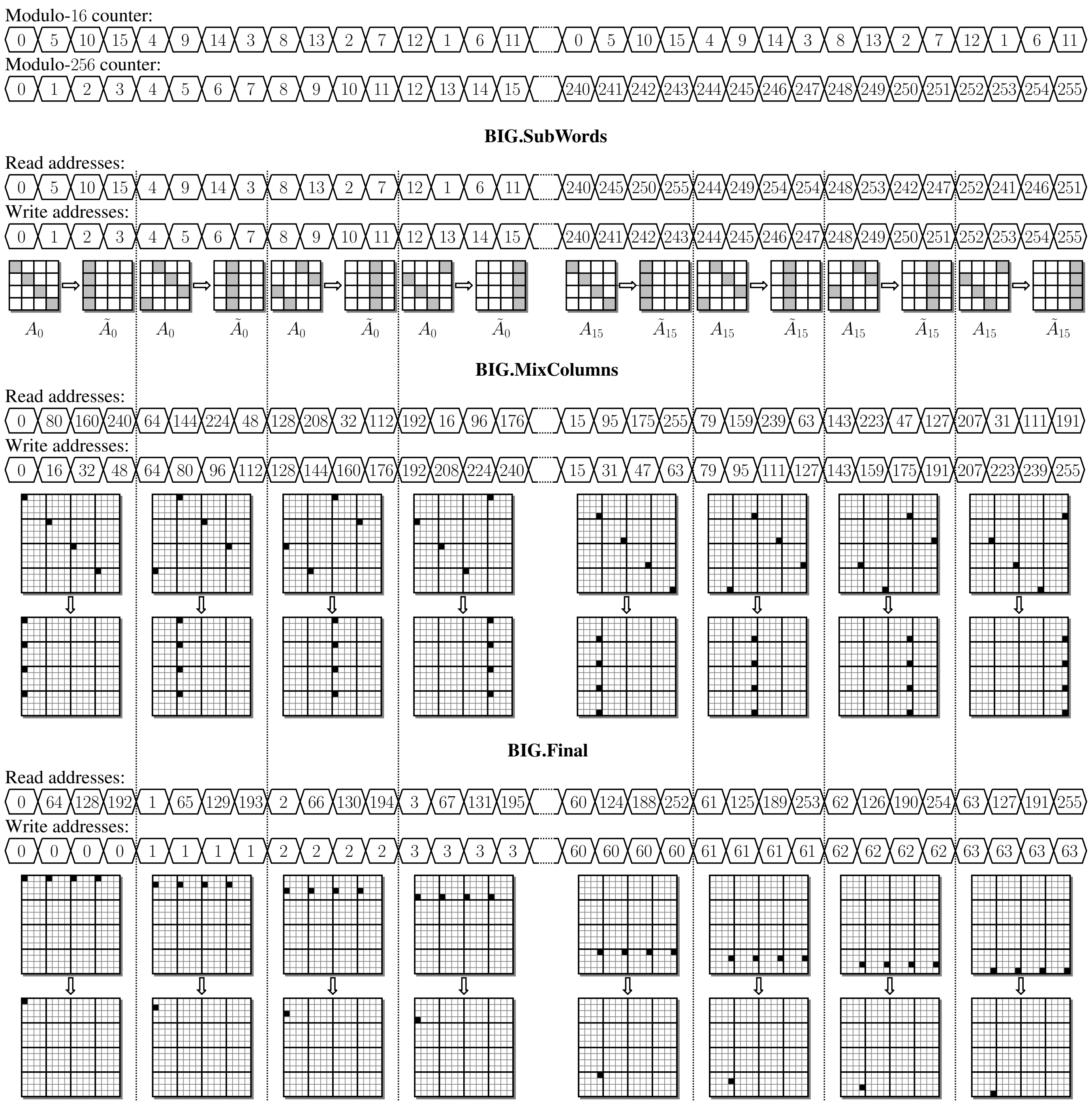


Fig. 9. Address generation during BIG.SubWords, BIG.MixColumns, and BIG.Final steps.

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

	Algorithm	FPGA	Area [slices]	Memory blocks	Frequency [MHz]	Throughput [Mbps]
This work	ECHO-224/256	xc5vlx50-2	127	1	352	72
Beuchat *et al.* [14]	BLAKE-32	xc5vlx50-2	56	2	372	225
Aumasson *et al.* [15]	BLAKE-32	xc5vlx110	390	–	91	575
Beuchat *et al.* [14]	BLAKE-64	xc5vlx50-2	108	3	358	314
Aumasson *et al.* [15]	BLAKE-64	xc5vlx110	939	–	59	533
Bertoni *et al.* [16]	Keccak	xc5vlx50-3	448	–	265	52
Baldwin *et al.* [17]	Shabal	xc5vlx220-2	2307	–	222.22	1330
Feron and Francq [18]	Shabal	not specified	596	–	109	1142
Detrey *et al.* [19]	Shabal	xc5vlx30-2	153	–	256	2051

compact coprocessor (at the price of an extra memory block). In spite of a high clock frequency, the throughput is however quite disappointing when compared to other candidates.

The main advantage of ECHO is that it is based on the round function of the AES. We plan to modify our architecture to support AES encryption, AES decryption, and ECHO. Figure 10 describes the general architecture of our processing unit. On Virtex-5 devices, the MixColumns and InvMixColumns units require the same number of slices. Thus, there is no need to exploit the relation between the MixColumns polynomial $c(x)$ and the InvMixColumns polynomial $d(x)$ [3, p. 55]. A few control bits allow one to configure the datapath of the coprocessor in order to perform the desired operation. We will design a control unit for this architecture in future work.

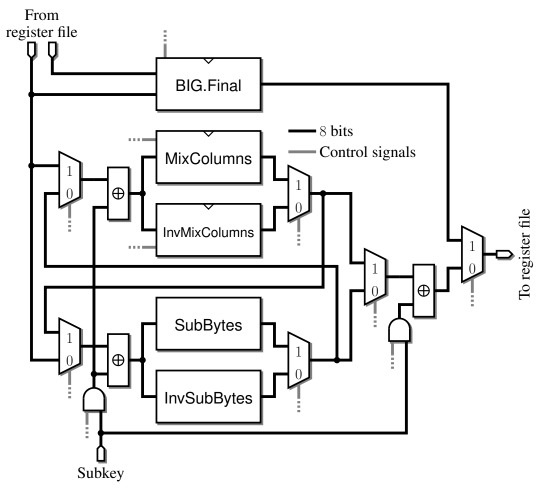


Fig. 10. A coprocessor for AES encryption, AES decryption, and ECHO.

REFERENCES

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

[2] J. Zhai, C. Park, and G.-N. Wang, "Hash-based RFID security protocol using randomly key-changed identification procedure," in *Computational Science and Its Applications–ICCSA 2006*, ser. Lecture Notes in Computer Science, M. Gavrilova, O. Gervasi, V. Kumar, C. K. Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, Eds., no. 3983. Springer, 2006, pp. 296–305.

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

[4] R. Benadjila, O. Billet, S. Gueron, and M. Robshaw, "The Intel AES instructions set and the SHA-3 candidates," in *Advances in Cryptology–ASIACRYPT 2009*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., no. 5912. Springer, 2009, pp. 162–178.

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

[6] T. Good and M. Benaissa, "AES on FPGA from the fastest to the smallest," in *Cryptographic Hardware and Embedded Systems–CHES 2005*, ser. Lecture Notes in Computer Science, J. R. Rao and B. Sunar, Eds., no. 3659. Springer, 2005, pp. 427–440.

[7] K. Gaj and P. Chodowiec, "PGA and ASIC implementations of the AES," in *Cryptographic Engineering*, Ç.K. Koç, Ed. Springer, 2009, pp. 235–294.

[8] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy, "Implementation of the AES-128 on Virtex-5 FPGAs," in *Progress in Cryptology–AFRICACRYPT 2008*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., no. 5023. Springer, 2008, pp. 16–26.

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

[10] M. Kinsy and R. Uhler, "SHA-3: FPGA implementation of ESSENCE and ECHO hash algorithm candidates using Bluespec," available at http://csg.csail.mit.edu/6.375/6_375_2009_www/projects/group1_report.pdf.

[11] K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta, "Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII," 2010, cryptology ePrint Archive, Report 2010/010.

[12] L. Lu, M. O'Neill, and E. Swartzlander, "Hardware evaluation of SHA-3 hash function candidate ECHO," available at http://www.ucc.ie/en/crypto/CodingandCryptographyWorkshop/TheClaudeShannonWorkshoponCodingCryptography2009/DocumentFile,75649,en.pdf.

[13] S. Tillich, M. Feldhofer, M. Kirschbaum, T. Plos, J.-M. Schmidt, and A. Szekely, "High-speed hardware implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein," 2009, cryptology ePrint Archive, Report 2009/510.

[14] J.-L. Beuchat, E. Okamoto, and T. Yamazaki, "Compact implementations of BLAKE-32 and BLAKE-64 on FPGA," 2010, cryptology ePrint Archive, Report 2010/173.

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

[16] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Keccak sponge function family main document (version 2.0)," 2009, available online at http://keccak.noekeon.org.

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

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

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