EFFICIENT BYTE PERMUTATION REALIZATIONS FOR COMPACT AES IMPLEMENTATIONS

Tuomas Järvinen, Perttu Salmela, Panu Hämäläinen, and Jarmo Takala

Tampere University of Technology, P.O.Box 553, FIN-33101 Tampere, Finland
Email: forename.surname@tut.fi

ABSTRACT

Advanced Encryption Standard (AES) algorithm incorporates a byte permutation operation which reorders the bytes within a 128-bit data block. This permutation can be described by reading the input data bytes into a 4×4 matrix called state in column wise and shifting the rows by one, two, or three bytes to the left. In decryption, the shifting is reversed, i.e., the rows are shifted to the right. While such shifting operations are straightforward if the computation is done with 128-bit data blocks at a time, they become more complex in area-efficient folded implementations where smaller than 128-bit data blocks are used. In such cases, a storage of data is required, either in the form of registers or memories. In this paper, efficient realizations of the byte permutations in AES algorithm, where the size of simultaneously computed data can be 1, 2, 4, or 8 bytes, are presented. All the realizations use the minimum number of storage elements implying area-efficiency.

1. INTRODUCTION

In various embedded applications, there is a need for a simple low-power implementation of AES algorithm [1] to guarantee a secure data transmission. However, current AES implementations are targeted mainly to high-speed data thus they exploit the parallelism in the algorithm for increasing the throughput. On the other hand, low-rate networks exist, e.g., for short range monitoring and control in automation, safety, healthcare, and entertainment at homes, public buildings and industry. For these applications, IEEE 802.15.4 standard [2] was ratified in the end of 2003 defining the radio and medium access control layer of low-rate, low-power wireless networks. The core of 802.15.4 security is the AES algorithm, thus its implementation has a great impact on area- and power-efficiency of the overall security system.

The current AES implementations can be divided into iterated, pipelined, and loop unrolled structures [3]. In iterated structures, computation of a 128-bit data block requires multiple iterations. In pipelined implementations, additional register stages are placed in the data path for enhancing the throughput. In the loop unrolled implementations, two or more rounds are computed at a clock cycle.

An efficient method for decreasing area and power is obtained with the iterated and folded structures where computation is done with the data blocks of size less than 128-bits. An example of such structure is proposed in [4] where 128-bit data block is computed iteratively in 32-bit blocks. Similar folded structure has been proposed in [5, 6]. The key advantage in folded structures is that the area can be decreased in proportion of the folding factor. However, the byte permutations become more complex since they span over the whole 128-bit block and either registers or memories are needed for delaying certain data bytes. Reducing the complexity of byte permutation realizations is the principal problem considered in this paper.

The AES byte permutations are illustrated in Fig. 1(a,b) with a 128-bit data organized into a 4×4 matrix called state. Each entry in the matrix represents a byte index within a 128-bit data block. These indices are permuted by shifting the row i cyclically to left/right by i, $i = \{0, 1, 2, 3\}$, thus these byte permutations are often called as ShiftRows operations. The permuted indices are $(0, 5, 10, 15, 1, 6, 11, 12, 2, 7, 8, 13, 3, 4, 9, 14)$ for left shift and $(0, 7, 10, 13, 1, 4, 11, 14, 2, 5, 8, 15, 3, 6, 9, 12)$ for right shift permutations.

The principal block diagram of the folded AES structure is depicted in Fig. 2. It consists of ShiftRows, SubBytes, MixColumns, and AddRoundKey operations. Various area-efficient implementations of SubBytes operation have been proposed, e.g., in [6, 7, 8], for MixColumns operation in [4, 6], and for AddRoundKey operation in folded AES implementation in [4]. None of the existing papers have considered efficient ShiftRows implementations for folded AES structures.

In current folded AES structures, the byte permutations are implemented with registers or FPGA-specific dual-port memories. For example, in [4], an FPGA implementation of a folded AES implementation is proposed where the byte permutations are suggested to be performed with separate dual-port memories for read and write operations. At the end of each round, the memories are swapped. This method uses twice the amount of memory that is actually needed. The authors also proposed an alternative realization, which uses shift registers. Also in [5, 6] registers were used for byte permutations. However, none of the previous register-based permutation unit resulted in the minimum number of registers.

In this paper, efficient realizations for the byte permutations in AES algorithm of any cipher key length are proposed. These realizations can be divided into register- and memory-based structures. The advantage of the given structures is that only the minimum amount of storage is needed implying area efficiency. The proposed structures are well suited for folded AES implementations where the computation is done with Q bytes at a time, $Q = \{1, 2, 4, 8\}$. In addition, some application-specific processors may benefit from the proposed structures in AES implementations if they are included as a custom unit. This way some performance gain is obtained since no permutations need to be done with processor's own resources, which can be time time-consuming since the permutations are done with bytes while typical processors operate with 16- or 32-bit words.

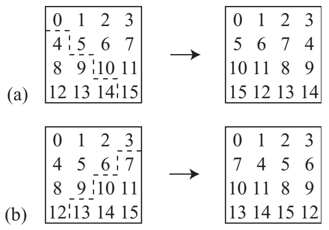


Figure 1: Byte permutations (ShiftRows operation) in AES algorithm: (a) shift rows left, (b) shift rows right.

2. MEMORY-BASED BYTE PERMUTATION UNITS

The principal idea in memory-based byte permutation unit (BPU) is to exploit an in-place storage method such that no conflicts are allowed and only the minimum amount of memory is used. In this way, there is no need to use separate write and read memories but a dual-port memory is needed due to simultaneous read and write operations. In general, the row address of a certain data byte is likely to be different in successive rounds but, however, the row address sequences are cyclic, i.e., they repeat after certain number of rounds.

Next, the address sequences are defined. The module addresses *ma* are constant determined by the following equation where Q is the number of memories and h is the index for 16 data bytes

$$ma(h) = h \mod Q. \tag{1}$$

In this case, there is no need for the multiplexing of data bytes between the memory modules and computation units thus the input and output ports of the memories can be straight connected to other computation units.

The row address sequences are slightly different for each parallel memory configuration. Let us assume that the number of memory modules Q is 1, 2, or 4. In this case, the row address *ra* of module *ma* for the left shift operation is given as

$$ra(j,k) = j + 4j(k+1) + (4/Q)(k+1)ma \mod 16/Q \tag{2}$$

where j is the access index, $j = (0, 1, \ldots, 16/Q-1)$, and k is the round, $k = (0, 1, 2, \ldots)$.

For the right shift operation, the row address for module *ma* is given as

$$ra(j,k) = j - 4j(k+1) - (4/Q)(k+1)ma \mod 16/Q. \tag{3}$$

Implementation of the row address generator is very simple: as the access index j and round index k need to be provided anyway, no separate counters are needed. It is also worth noting that the number of memory modules Q is constant in actual realization, thus no division need to be implemented. In addition, since all the computations are modulo $16/Q$, the word width for the generated row address is $\log_2(16/Q)$.

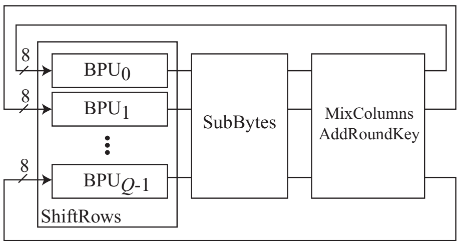


Figure 2: Principal block diagram of folded AES structure. BPU: byte permutation unit. Q: number of bytes processed at a time. $Q = \{1, 2, 4, 8\}$.

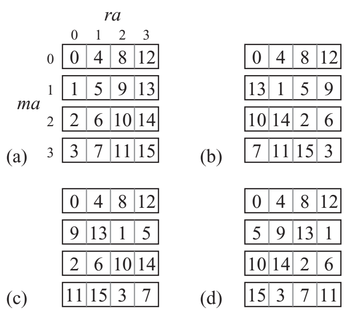


Figure 3: Evolution of memory contents: (a) initial state, (b) after 1st round ($k = 0$), (c) after 2nd round ($k = 1$), and (d) after 3rd round ($k = 2$).

Next, the operation of memory-based BPUs is illustrated with an example where the number of parallel memories is four, which is targeted to AES structures where four bytes are computed at a time, as in [4, 5]. First, the 128-bit data block is initialized into four dual-port memories as depicted in Fig. 3(a). As shown, the data bytes are written into consecutive memory locations. Consider the left shift operation where the first four bytes to be processed are $0, 5, 10, 15$. These bytes are read out, processed, and the results with the new indices $0, 1, 2, 3$ are stored into the same locations. The row addresses for that are obtained from (2). After all the bytes are processed, the contents of the memories is as depicted in Fig. 3(b). Continuing in a similar way, the contents of memories after 2nd and 3rd rounds are shown in Fig. 3(c) and Fig. 3(d), respectively. Thereafter, the memory contents return to its initial state and the cycle starts over.

3. REGISTER-BASED BYTE PERMUTATION UNITS

Register-based BPUs are constructed based on Parhi's design methodology for data format converters proposed in [9]. Such design methodology results in one-dimensional permutation units, where a shift register is used for delaying data. The data is reordered by circulating the bytes from the last register backwards with the aid of multiplexers. This backward allocation is possible since certain bytes are read out earlier, i.e., bypassed with multiplexers placed at the output. Since the backward allocation and bypassing are in balance, there exist no deadlocks, and each time a byte needs to be backward allocated there is an empty slot available.

Let us begin with a one-port byte permutation unit. By following the design methodology in [9], the structure depicted in Fig. 4(a) is obtained. The one-port BPU is capable of performing both left and right shift byte permutations with the given control signals c_i. Note that with c_i value of zero, the uppermost byte in the input is passed to the output. The latency of the network is 12 cycles, which is the maximum distance a data byte has to be moved in the permutation. Thus, 12 registers are needed in minimum, which is the lower bound for register complexity. The operation of the network is continuous so there is no need for empty cycles and the next 128-bit sequence can be fed in after the last byte of the previous 128-bit sequence. The depicted control signals are given for permuting a single 128-bit sequence where clock cycle $t = 0$ is the time instant when the first byte is at the input of BPU.

The two-port BPU depicted in Fig. 4(b) takes in two bytes at a time and has a latency of six cycles. Also this BPU is capable of performing both the left and right shift permutations with the given control signals. Similarly, we obtain four- and eight-port BPUs, depicted in Fig. 4(c) and Fig. 4(d), respectively. Both the structures support left and right shift permutations.

The presented register-based byte permutation units were obtained based on the general design methodology for data format converters in [9]. This methodology is well suitable for ShiftRows permutations since there is no need to exchange bytes between parallel delay lines. All the resulting structures require only the minimum number of registers, as stated in [9]. As the number of registers in the proposed BPUs is less than required for storing all the 16 bytes, some pipeline stages need to be placed into final AES structures. Pipelining is however often used in AES implementations and is likely to provide some performance gain by shortening the critical path.

4. COMPARISON

In this section, a short comparison of the 4-port BPUs is made and the results are shown in Table 1. The Proposed 1 corresponds to 4-port memory-based BPU and the Proposed 2 to 4-port register-based BPU depicted in Fig. 4(c). None of the structures except the proposed ones are scalable to the other number of ports. In addition, the structures proposed in [6, 5] are not applicable to decryption since no right shift operation is supported. If such support were adopted, it would increase the complexity of multiplexing. In Table 1, the multiplexing complexity is estimated by converting all multiplexers to equivalent 8-bit 2-to-1 multiplexers, e.g., a 32-bit 4-to-1 multiplexer is equivalent to 12 8-bit 2-to-1 muxes. As seen, the number of registers in the Proposed 1 is less than in the other realization. Similarly, the size of memory in the Proposed 2 is half of the scheme in [4]. The register-based BPU in [4] has less multiplexers than the Proposed 1 because the bytes are not backward allocated. This requires more registers and implies also greater latency. Also in [6, 5] less multiplexers is needed than in Proposed 2, which is due to the fact that they do not support both left and right ShiftRows permutations. It is possible to configure the Proposed 2 BPU in Fig. 4(c) for left shift permutations which would reduce the number of multiplexers to 10. In that case the multiplexers corresponding the control signals c_0, c_1, c_2 are left out and c_3 is reduced to an 8-bit 2-to-1 multiplexer.

5. CONCLUSIONS

In this paper, efficient byte permutation units for compact AES structures of any cipher key length were proposed. The units are divided into memory- and register-based structures based on the type of storage. All units are area-efficient since they use only the minimum amount of storage elements. They support folded AES structures where the number of parallel computed bytes Q can be $Q = \{1,2,4,8\}$. It was also shown that compared to other existing approaches for

Table 1: Comparison of 4-port BPUs. # regs: number of 8-bit registers. mem: size of memory in bytes. # muxes: number of 8-bit 2-to-1 multiplexers.

	# regs	mem	# muxes	shift left/right	scalable
[4] (memory)	–	32	–	yes/yes	no
[4] (register)	28	–	9	yes/yes	no
[6]	28	–	12	yes/no	no
[5]	16	–	12	yes/no	no
Proposed 1	–	16	–	yes/yes	yes
Proposed 2	12	–	14	yes/yes	yes

byte permutations in compact AES implementations, the proposed units resulted in less number of registers and memory.

REFERENCES

[1] *Advanced Encryption Standard*, National Institute of Standards and Technology Std., Nov. 2001.

[2] *Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for low-rate wireless personal area networks*, IEEE Std., 2003.

[3] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 4, pp. 545–557, Aug. 2001.

[4] P. Chodowiec and K. Gaj, "Very compact FPGA implementation of the AES algorithm." in *CHES*, ser. Lecture Notes in Computer Science, C. D. Walter, Ç. K. Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 319–333.

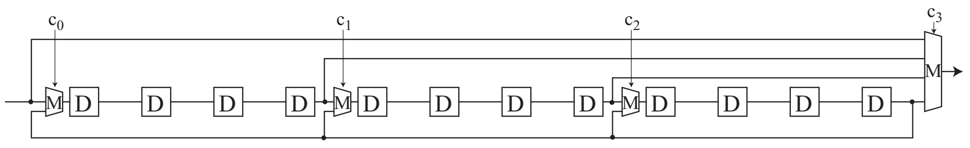
[5] S. McMillan and C. Patterson, "Jbits^{tm} implementations of the advanced encryption standard (Rijndael)." in *FPL*, ser. Lecture Notes in Computer Science, G. J. Brebner and R. Woods, Eds., vol. 2147. Springer, 2001, pp. 162–171.

[6] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A compact Rijndael hardware architecture with S-box optimization," in *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*. Springer-Verlag, 2001, pp. 239–254.

[7] V. Rijmen. Efficient implementation of the Rijndael S-box. [Online]. Available: www.esat.kuleuven.ac.be/∼rijmen/rijndael/sbox.pdf

[8] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, "Efficient Rijndael encryption implementation with composite field arithmetic," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. Springer-Verlag, 2001, pp. 171–184.

[9] K. K. Parhi, "Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation," *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, vol. 39, no. 7, pp. 423–440, Jul. 1992.

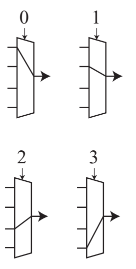


shift left

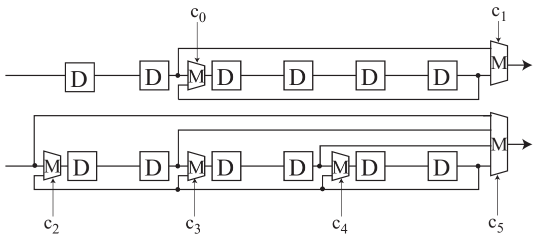
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
c_0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
c_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0
c_2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0	0
c_3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1	0	3	2	1	1	3	2	3	2	3	3	3	3

shift right

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
c_0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
c_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0
c_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	0	0	0	0
c_3	3	3	3	3	3	3	3	3	3	3	3	3	3	0	1	2	3	1	1	2	3	2	3	2	3	3	3	3



(a)



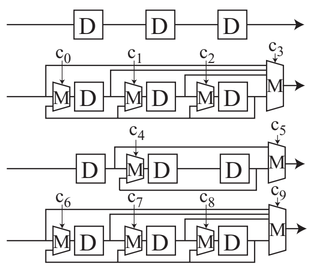
shift left

	0	1	2	3	4	5	6	7	8	9	10	11	12
c_0	0	0	0	0	0	0	1	0	1	0	0	0	0
c_1	2	2	2	2	2	2	0	2	0	2	2	2	2
c_2	0	0	0	0	0	0	0	1	0	0	0	0	0
c_3	0	0	0	0	0	0	0	0	0	1	0	0	0
c_4	0	0	0	0	0	0	1	0	1	0	1	1	0
c_5	3	3	3	3	3	3	2	0	2	1	2	2	3

shift right

	0	1	2	3	4	5	6	7	8	9	10	11	12
c_0	0	0	0	0	0	0	1	0	1	0	0	0	0
c_1	2	2	2	2	2	2	0	2	0	2	2	2	2
c_2	0	0	0	0	0	0	1	0	0	0	0	0	0
c_3	0	0	0	0	0	0	0	0	1	0	0	0	0
c_4	0	0	0	0	0	0	0	1	0	1	1	1	0
c_5	3	3	3	3	3	3	0	2	1	2	2	2	3

(b)



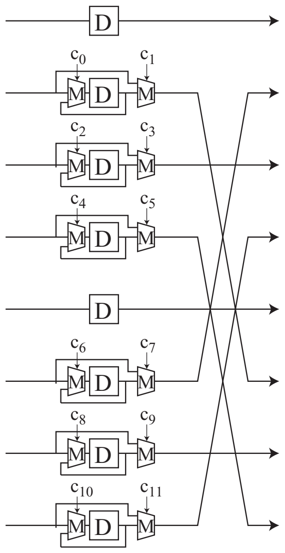
shift left

	0	1	2	3	4	5	6
c_0	0	0	0	0	0	0	0
c_1	0	0	0	0	0	0	0
c_2	0	0	0	1	1	1	0
c_3	1	1	1	0	0	0	1
c_4	0	0	0	1	1	0	0
c_5	1	1	1	0	0	1	1
c_6	0	0	0	1	0	0	0
c_7	0	0	0	0	1	0	0
c_8	0	0	0	0	0	1	0
c_9	3	3	3	0	1	2	3

shift right

	0	1	2	3	4	5	6
c_0	0	0	0	1	0	0	0
c_1	0	0	0	0	1	0	0
c_2	0	0	0	0	0	1	0
c_3	3	3	3	0	1	2	3
c_4	0	0	0	1	1	0	0
c_5	1	1	1	0	0	0	1
c_6	0	0	0	0	0	1	0
c_7	0	0	0	0	0	0	0
c_8	0	0	0	0	0	0	0
c_9	3	3	3	0	1	2	3

(c)



shift left

	0	1	2
c_0	0	1	0
c_1	1	0	1
c_2	0	1	0
c_3	1	0	1
c_4	0	0	0
c_5	1	1	1
c_6	0	0	0
c_7	1	1	1
c_8	0	1	0
c_9	1	0	1
c_{10}	0	1	0
c_{11}	1	0	1

shift right

	0	1	2
c_0	0	0	0
c_1	1	1	1
c_2	0	1	0
c_3	1	0	1
c_4	0	1	0
c_5	1	0	1
c_6	0	1	0
c_7	1	0	1
c_8	0	1	0
c_9	1	0	1
c_{10}	0	0	0
c_{11}	1	1	1

(d)

Figure 4: Block diagrams of register-based byte permutation units (BPU) with multiplexer control. (a) 1-port BPU, (b) 2-port BPU, (c) 4-port BPU, and (d) 8-port BPU. D: 8-bit register. M: 8-bit multiplexer. c$_i$: control signal. t: clock cycle.