PAPER

# A Design of AES Encryption Circuit with 128-bit Keys Using Look-Up Table Ring on FPGA

Hui QIN[†], *Nonmember*, Tsutomu SASAO[†a)], *and* Yukihiro IGUCHI[††], *Members*

**SUMMARY** This paper addresses a pipelined partial rolling (PPR) architecture for the AES encryption. The key technique is the PPR architecture. With the proposed architecture on the Altera Stratix FPGA, two PPR implementations achieve 6.45 Gbps throughput and 12.78 Gbps throughput, respectively. Compared with the unrolling implementation that achieves a throughput of 22.75 Gbps on the same FPGA, the two PPR implementations improve the memory efficiency (i.e., throughput divided by the size of memory for core) by 13.4% and 12.3%, respectively, and reduce the amount of the memory by 75% and 50%, respectively. Also, the PPR implementation has a up to 9.83% higher memory efficiency than the fastest previous FPGA implementation known to date. In terms of resource efficiency (i.e., throughput divided by the equivalent logic element or slice), one PPR implementation offers almost the same as the rolling implementation, and the other PPR implementation offers a medium value between the rolling implementation and the unrolling implementation that has the highest resource efficiency. However, the two PPR implementations can be implemented on the minimum-sized Stratix FPGA while the unrolling implementation cannot. The PPR architecture fills the gap between unrolling and rolling architectures and is suitable for small and medium-sized FPGAs.

***key words:*** *AES encryption, pipelined partial rolling (PPR), FPGA*

## 1. Introduction

The Advanced Encryption Standard (AES) [1] was accepted as a FIPS (Federal Information Processing Standards) in Nov. 2001, and became effective on May 26, 2002 by NIST (National Institute of Standards and Technology) to replace DES (Data Encryption Standard). With the increasing requirements for secure communications, the AES has a broad range of applications, including smart cards, cellular phones, Web servers, automated teller machines (ATMs), and digital video recorders. Since Nov. 2001, various AES implementations using ASICs or FPGAs have been reported. Some focus on the small chip area by using the rolling architecture whereby the data are iteratively passed through the round transformations [2]–[4], and others focus on high throughput by using the unrolling architecture whereby processing multiple blocks of data simultaneously. To achieve a high throughput, partition of each round by inserting pipeline registers is necessary. However, this will increase the cycles (or stages) and registers in the AES round blocks. For example, Saggese et al. [5] achieved 20.3 Gbps with 50 cycles, while Zambreno et al. [6] achieved 23.50 Gbps with 30 cycles.

Manuscript received June 29, 2005.
Manuscript revised September 29, 2005.

[†]The authors are with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
[††]The author is with the Department of Computer Science, Meiji University, Kawasaki-shi, 214–8571 Japan.
a) E-mail: sasao@cse.kyutech.ac.jp

DOI: 10.1093/ietisy/e89–d.3.1139

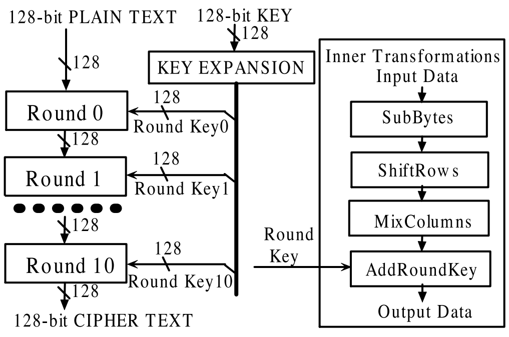


**Fig. 1** AES encryption with 128-bit key.

In this paper, we have developed a pipelined partial rolling (PPR) architecture to achieve a high throughput with small memory area. It performs partial-rolling in the round while adopting the pipeline technique. In addition, this architecture is well suited to operate in an online key generation manner. Using the PPR architecture, a fully online key generated AES encryption processor of 128-bit key as shown in Fig. 1 can be implemented on the minimum size Altera Stratix device, but the unrolling implementation is unable to fit into the same device. Furthermore, the proposed implementation on the Altera Stratix EP1S20F780C5 device increases the memory efficiency up to 13.4% compared with the unrolling implementation on the same device. It has 9.83% higher memory efficiency than the prior fastest FPGA implementation [6], and it can reduce the amount of memory up to 75%. The rest of the paper is organized as follows: Section 2 explains the AES algorithm. Section 3 presents conventional AES implementation. Section 4 introduces the AES design based on LUT ring. Section 5 presents the AES implementation using an FPGA. Section 6 shows the experimental results. And finally, Sect. 7 concludes the paper.

The preliminary version of this paper was presented at GLSVLSI'05 [7].

## 2. AES Algorithm

The AES algorithm is based on arithmetic in a finite Galois field, GF($2^8$), and is a symmetric block cipher that encrypts 128-bit plain text data with a 128-bit, 192-bit, or 256-bit cipher key [1]. The 128-bit plain text data is divided into 16-byte data. These byte data are mapped to a 4×4 matrix called the State, and all the internal operations of the AES algorithm are performed on the State. The algorithm consists of four basic transformations that make up a round which is iterated 10 times for a 128-bit length key, 12 times for a 192-bit key, and 14 times for a 256-bit key.

- SubBytes - Replaces each byte of the data block with another byte by using an S-box lookup table. The contents of the S-box is the multiplicative inverse in GF ($2^8$), combined with an affine permutation over GF(2).
- ShiftRows - Cyclically shifts *i* bytes of the state in each row, where *i* is the row number.
- MixColumns - Groups 4-bytes of each column in the state matrix together forming 4-term polynomials and multiplies the polynomials with a fixed polynomial mod ($x^4 + 1$).
- AddRoundKey - Adds the round key to the state using a bit-wise XOR operation.

The round key for each round are generated through the key expansion process that is described by the pseudo code listed in the Fig. 2 [1], where $N_r$ is 10, 12 or 14 and $N_k$ is 4, 6, or 8, when the key length is 128, 192 or 256-bit, respectively.

In the key expansion, SubBytes transformation is applied to each of the four bytes in the SubWord, while the RotWord cyclically shifts each byte in a word one byte to the left. The Rcon is a constant word array, and only the leftmost byte in each word is nonzero [1]. The key expansion generates a total of $4(N_r + 1)$ 4-byte words ($w_0$, $w_1$,..., $w_{4(N_r+1)-1}$). The initial key, which is divided into $N_k$ words, is used as the initial $N_k$ words ($w_0, w_1, \dots, w_{N_k-1}$), and the rest of the words are generated from the initial key iteratively. Each round key has 128 bits, and is formed by concatenating four words: Round Key(i) = ($w_{4i}$, $w_{4i+1}$, $w_{4i+2}$, $w_{4i+3}$).

In this paper, we focus on the AES encryption using a 128-bit key shown in Fig. 1 which requires 11 rounds (i.e., logic operations). The first round performs only the AddRoundKey transformation, the middle 9 rounds perform all the four transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey, and the final round performs three transformations: SubBytes, ShiftRows, and AddRoundKey, omitting the MixColumns transformation. The round keys for each round are generated from the original 128-bit input key through the key expansion block. In general, two methods exist to generate the round keys. In the first method, the round keys are stored in a register or memory, and then used for all incoming plain text data. However, this method requires a large register or memory for the round keys, and it needs a preprocessing phase every time the key is changed. The second method is an online key generation algorithm, where the round keys are generated concurrently with the encryption process. Since the online key generation method allows the block cipher to work at full speed even if the key is changed, we adopted this method in this work.

```
KeyExpansion(byte key[4*Nk], word w[4*(Nr+1)], Nk)
begin
word temp
i = 0
while (i < Nk)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
     i = i+1
end while
i = Nk
while (i < 4 * (Nr+1)]
    temp = w[i-1]
    if (i mod Nk = 0)
       temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if (Nk > 6 and i mod Nk = 4)
       temp = SubWord(temp)
   end if
   w[i] = w[i-Nk] xor temp
    i = i + 1
end while
end
```

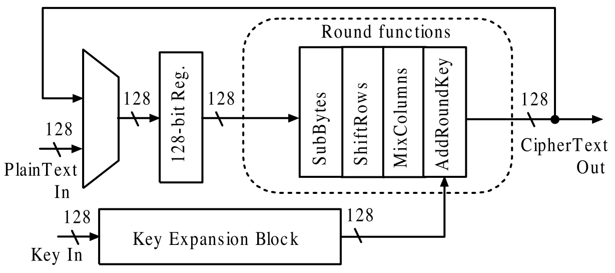
**Fig. 2** Pseudo code for key expansion.

## 3. Existing AES Architecture

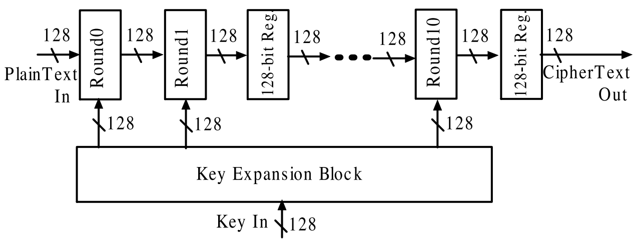
Various architectures exist to realize the AES encryption. Among them, the rolling architecture and the unrolling architecture shown in Fig. 3 (a) and (b) are the two basic architectures.

The **rolling architecture shown** in Fig. 3 (a) uses a feedback structure where the data are iteratively transformed by the round functions. This approach occupies small area, but achieves low throughput. Existing rolling implementations [2], [3], [6], have the throughput of approximately 1 to 1.4 G-bit/s, and the size of the memory for the core is just 32 K bits.

In the **unrolling architecture** shown in Fig. 3 (b), the



(a) Rolling architecture of AES Encryption with 128-bit key.



(b) Unrolling architecture of AES Encryption with 128-bit key.

**Fig. 3** Existing AES architectures.

round blocks are pipelined and the inserted pipeline registers allow simultaneous operation of all 11 round blocks. Due to the pipeline, this approach achieves a high throughput, but requires large area. Existing unrolling implementations [5], [6], [8], [9], have the throughput of approximately 10 to 23 G-bit/s, and the size of the memory for the core is up to 320 K bits.

## 4. Pipelined Partial Rolling (PPR) Architecture

This section describes the AES design with a new type of memory-based architecture called **pipelined partial rolling** (**PPR**) based on LUT ring [10], [11].

In the AES round, among the four inner transformations, the SubBytes requires the largest area and latency. It consists of 16 S-Boxes that is the most complicated function block in the entire circuit. For the SubBytes, both of the rolling implementation and the unrolling implementation use 16 S-Boxes. In the PPR, we use a special mechanism for the ShiftRows and the SubBytes to reduce the number of S-Boxes and areas for shifters and multiplexers.

Two approaches exist to realize an S-Box, the first one uses Galois Field operations. In this method, since the direct calculation of the multiplicative inverse in GF($2^8$) is very expensive, the inventors of the AES algorithm suggest an algorithm that calculates the multiplicative inverse in GF($2^8$) using the GF($2^4$) operations [12]. Reference [13] presents one implementation of such algorithm. In this paper, we called it combinational S-Box using Galois Field operations. The other method uses look-up table that is faster but consumes larger amount of area compared with the former, as the studies in [14] and [15] indicate. Since the PPR requires the minimum latency for the S-Box, we use a ROM to realize the S-Box in the SubBytes. As for the MixColumns, since the multiplication over GF($2^8$) in MixColumns uses a constant as one operand, and this constant multiplication can be simply converted into a bit-wise XOR operation, the matrix multiplication can be replaced by several XOR operations. Hence, the MixColumns can be realized as XOR operations. Also, the AddRoundKey operation uses an XOR operation to add the round key. Thus, we can easily implement the MixColumns and the AddRoundKey by using bit-wise XOR operations.

Figure 4 shows the architecture of the PPR. We used pipeline to increase the throughput. The key expansion block consists of 10 round key circuits that used to generate the round key for each round. A round unit consists of the following components:

**Rolling Part:** Performs the ShiftRows and the SubBytes transformations.

**128-bit Pipeline Reg.:** Stores the states of each round.

**MixColumns:** Performs the MixColumns using several bit-wise XOR operations.

**128-bit XOR:** Performs the AddRoundKey using 128-bit bit-wise XOR operations.

The rolling part is the most important part in the round unit. In this paper, we show two designs for the rolling part: The 4SM and the 8SM.

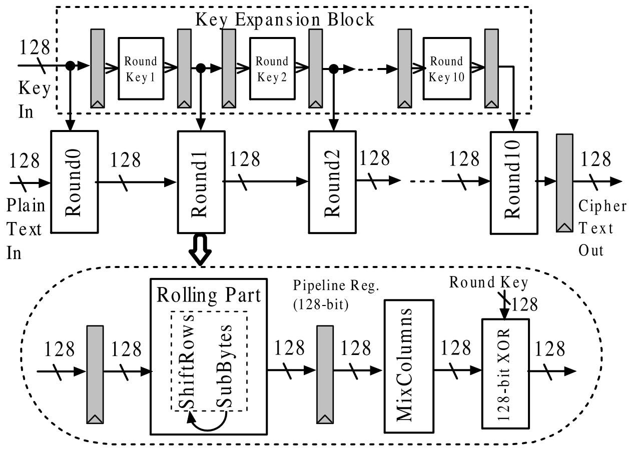


**Fig. 4** Architecture of PPR.

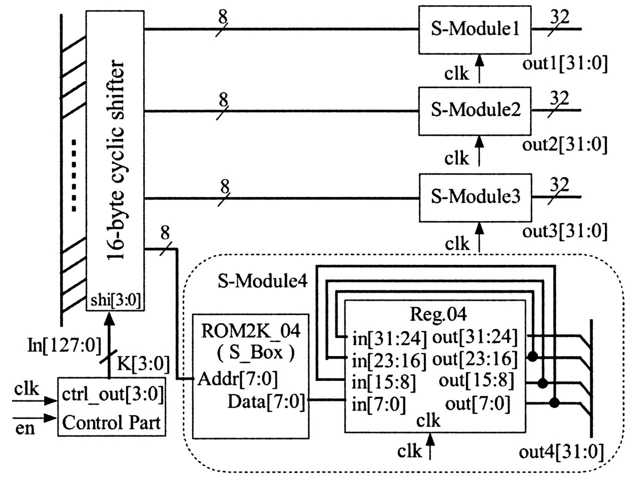


**Fig. 5** Rolling part: 4SM.

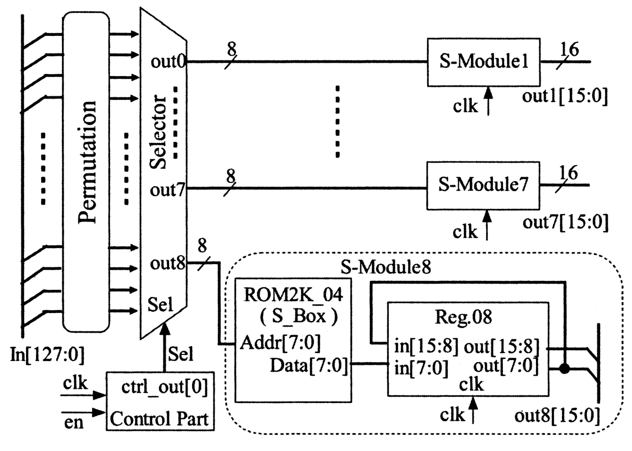


**Fig. 6** Rolling part: 8SM.

The **4SM** shown in Fig. 5 consists of a 16-byte cyclic shifter and four copies of **S-Module (SM)**. The 16-byte cyclic shifter have 16-byte (128-bit) inputs and 4-byte outputs. Let **F** be the input-to-output mapping function of the

**Table 1** Relationship between the inputs and the outputs of the permutation network in 8SM.

| Inputs | $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $X_8$ | $X_9$ | $X_{10}$ | $X_{11}$ | $X_{12}$ | $X_{13}$ | $X_{14}$ | $X_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Outputs | $X_0$ | $X_{10}$ | $X_4$ | $X_{14}$ | $X_8$ | $X_2$ | $X_{12}$ | $X_6$ | $X_5$ | $X_{15}$ | $X_9$ | $X_3$ | $X_{13}$ | $X_7$ | $X_1$ | $X_{11}$ |

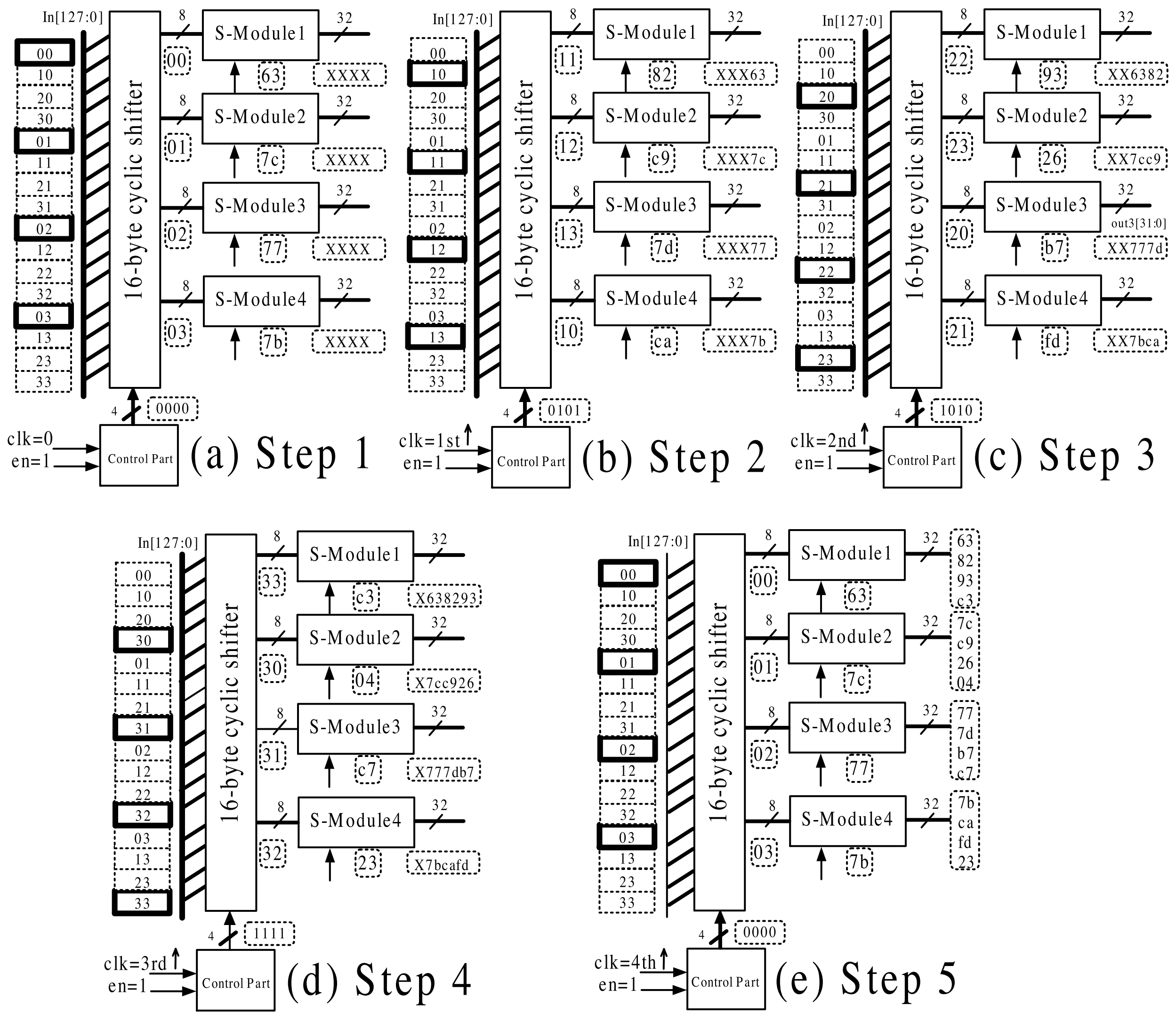


**Fig. 7** Operations of rolling part: 4SM.

cyclic shifter, then

$$\mathbf{F}(X_0, X_1, \ldots, X_{15}, K) = ( X_{K(mod16)}, X_{(K+4)(mod16)}, X_{(K+8)(mod16)}, X_{(K+12)(mod16)}),$$

where $K$ is 0, 5, 10 or 15. $K$ is represented by the 4-bit output signals from the control part, and $X_i$ denotes byte data. For example,

$$\mathbf{F}(X_0, X_1, \ldots, X_{15}, 5) = (X_5, X_9, X_{13}, X_1).$$

Each S-Module consists of a 2 K-bit ROM and a 32-bit feed-back register, where the ROM stores the table for the S-Box, and the feed-back register stores the outputs of the S-Box.

The **8SM** shown in Fig. 6 consists of a 16-byte to 8-byte selector and eight copies of S-Module. In front of the selector, the permutation network is used to arrange the input data in the required order. Table 1 shows the permutation, where both inputs and outputs are 16-byte data. When the output Sel is 0, the upper 8 bytes are selected. On the other hand, when the output Sel is 1, the lower 8 bytes are selected. Each S-Module consists of a 2 K-bit ROM and a 16-bit feed-back register, where the ROM stores the table for the S-Box, and the feed-back register stores the outputs of the S-Box. In the round operation, the S-Modules of the 4SM are used four times, while the S-Modules of the 8SM are used twice.

For the 4SM, we can also adopt the architecture of the 8SM. That is, to use permutation network and selector instead of the 16-byte cyclic shifter. In our FPGA implementation, Quartus II 4.1 simulator shows that a 16-byte cyclic shifter produces higher memory efficiency than the permutation and a selector, while the areas for the two implementations are almost the same.

Both of the 4SM and the 8SM, the whole circuits of the rolling parts form the special LUT rings where feedback is unnecessary [10], [11].

**Example 4.1:** This part illustrates the operations for the 4SM. Let the hexadecimal representation of the 128-bit original data be 00 10 20 30 01 11 21 31 02 12 22 32 03 13 23 33. After SubBytes and ShiftRows, the output data become 63 82 93 c3 7c c9 26 04 77 7d b7 c7 7b ca fd 23. Figures 7 (a) - (b) show the operations of the 4SM.

**In Step 1**, the outputs of the control part are 0000, and then the outputs of the 16-byte cyclic shifter become 00, 01, 02, 03. By using the S-Boxes, the outputs of the ROMs become 63, 7c, 77, 7b, respectively.

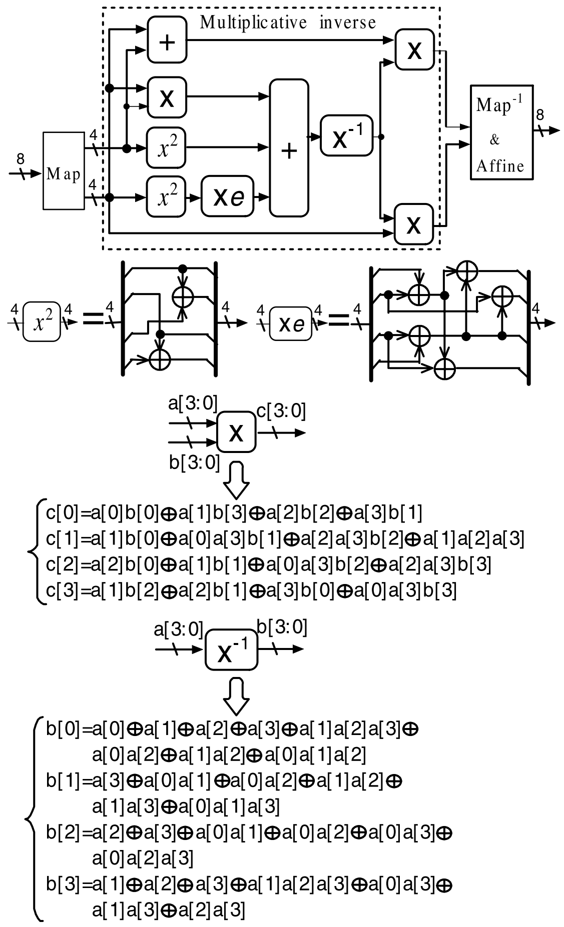
**In Step 2**, when a positive clock is applied, the outputs of the control part become 0101. At the same time, the previous outputs of ROMs (63, 7c, 77, 7b) are stored in the registers, and also sent to the output terminals. And then the outputs of the 16-byte cyclic shifter become 11, 12, 13, 10. By using the S-Boxes, the outputs of the ROMs become 82,c9,7d,ca, respectively.

**In Step 3**, when a positive clock is applied, the outputs of the control part become 1010. At the same time, the previous outputs of ROMs (82, c9, 7d, ca) are stored in the registers, and also sent to the output terminals. And then the outputs of the 16-byte cyclic shifter become 22, 23, 20, 21. By using the S-Boxes, the outputs of the ROMs become 93, 26, b7, fd, respectively.

**In Step 4**, when a positive clock is applied, the outputs of the control part become 1111. At the same time, the previous outputs of ROMs (93, 26, b7, fd) are stored in the registers, and also they are sent to the output terminals. And then the outputs of the 16-byte cyclic shifter become 33, 30, 31, 32. By using the S-Boxes, the outputs of the ROMs become c3, 04, c7, 23, respectively.

**In Step 5**, when a positive clock is applied, the outputs of the control part become 0000. At the same time, the previous outputs of ROMs (c3, 04, c7, 23) are stored in the registers, and also sent to the output terminals. And then the outputs of the 16-byte cyclic shifter become 00, 01, 02, 03. In this way, the rolling part implements the SubBytes and ShiftRows transformations. (End of Example)

Since the round keys are generated on the fly, we divide the Key Expansion Block into 10 round key circuits while inserting the same number of pipeline registers as the round circuit as shown in Fig. 4. From Fig. 2, we can observe that the critical path of the round key circuit consists of one S-Box and four XOR gates. Since the critical path of the key expansion is shorter than that of a Rolling Part, we use combinational S-Box [13] in each round key circuit to reduce the aera. Figure 8 shows the detailed implementation of the S-Box using Galois Field operations. First, the input byte (an element of GF($2^8$)) is mapped to two elements of GF($2^4$). Next, the multiplicative inverse is calculated using GF($2^4$) operators. Then the two GF($2^4$) elements are inversely mapped to one element in GF($2^8$). In the end, the affine transformation is performed to complete the SubBytes transformation.



$$\begin{cases} c[0]=a[0]b[0]\oplus a[1]b[3]\oplus a[2]b[2]\oplus a[3]b[1] \\ c[1]=a[1]b[0]\oplus a[0]a[3]b[1]\oplus a[2]a[3]b[2]\oplus a[1]a[2]a[3] \\ c[2]=a[2]b[0]\oplus a[1]b[1]\oplus a[0]a[3]b[2]\oplus a[2]a[3]b[3] \\ c[3]=a[1]b[2]\oplus a[2]b[1]\oplus a[3]b[0]\oplus a[0]a[3]b[3] \end{cases}$$

$$\begin{cases} b[0]=a[0]\oplus a[1]\oplus a[2]\oplus a[3]\oplus a[1]a[2]a[3]\oplus \\ \quad a[0]a[2]\oplus a[1]a[2]\oplus a[0]a[1]a[2] \\ b[1]=a[3]\oplus a[0]a[1]\oplus a[0]a[2]\oplus a[1]a[2]\oplus \\ \quad a[1]a[3]\oplus a[0]a[1]a[3] \\ b[2]=a[2]\oplus a[3]\oplus a[0]a[1]\oplus a[0]a[2]\oplus a[0]a[3]\oplus \\ \quad a[0]a[2]a[3] \\ b[3]=a[1]\oplus a[2]\oplus a[3]\oplus a[1]a[2]a[3]\oplus a[0]a[3]\oplus \\ \quad a[1]a[3]\oplus a[2]a[3] \end{cases}$$

**Fig. 8** Combinational S-Box using Galois field operations.

## 5. FPGA Implementation

We use the Altera Stratix FPGA to implement the AES encryption circuit. The Altera Stratix FPGAs offer special RAM blocks called M4K that can store 4096 bits. The M4K can be configured at ratios between 4096×1 to 256×16, and may have dual-port functionality. The M4Ks are also suitable for implementing synchronous ROMs.

As mentioned in Sect. 4, in the round, the MixColumns and the AddRoundKey can be realized as a network of XOR gates and are implemented by Logic Elements (LEs) on the FPGA. In the rolling part, the 16-byte cyclic shifter for the 4SM and selector for the 8SM are implemented by LEs. To implement S-Boxes, we used the M4K. Each M4K is configured as a dual-port synchronous 256×8-bit words ROM to implement two separate S-Boxes. The values in the look-up tables for S-Boxes are loaded into the M4Ks at the con-

**Table 2** Comparison of AES-4SM, AES-8SM, UNROLLING, UNROLL_ROM and the published works.

| Design | Device | LE / | Memory | Memory | Cycle | F_clk | Th. | Mem-eff | Eq-LE / | LE-eff / | Design |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Slice | for key | for core | | (MHz) | (Gbps) | $(\frac{\text{Mbps}}{\text{K-bit}})$ | Eq-Sli | $\frac{\text{Sli-eff}}{2}$ | Rule |
| AES-4SM | 1S25 | 10530 LEs | 0 | 80 K-bit (20 M4Ks) | 21 | 49.80 | 6.37 | 79.68 | 15650 LEs | 0.407 Mbps/LE | 0.13 $\mu$m |
| AES-8SM | 1S25 | 10730 LEs | 0 | 160 K-bit (40 M4Ks) | 21 | 98.31 | 12.58 | 78.65 | 20970 LEs | 0.600 Mbps/LE | 0.13 $\mu$m |
| UNROLLING | 1S25 | 8220 LEs | 0 | 320 K-bit (80 M4Ks) | 21 | 179.95 | 23.03 | 71.98 | 28700 LEs | 0.803 Mbps/LE | 0.13 $\mu$m |
| UNROLL_ROM | 1S25 | 7600 LEs | 80 K-bit (20 M4Ks) | 320 K-bit (80 M4Ks) | 21 | 182.48 | 23.36 | 72.99 | 33200 LEs | 0.704 Mbps/LE | 0.13 $\mu$m |
| AES-4SM | 1S20 | 10645 LEs | 0 | 80 K-bit (20 M4Ks) | 21 | 50.41 | 6.45 | 80.66 | 15765 LEs | 0.409 Mbps/LE | 0.13 $\mu$m |
| AES-8SM | 1S20 | 10538 LEs | 0 | 160 K-bit (40 M4Ks) | 21 | 99.84 | 12.78 | 79.87 | 20778 LEs | 0.615 Mbps/LE | 0.13 $\mu$m |
| UNROLLING | 1S20 | 8515 LEs | 0 | 320 K-bit (80 M4Ks) | 21 | 177.75 | 22.75 | 71.10 | 28995 LEs | 0.785 Mbps/LE | 0.13 $\mu$m |
| UNROLL_ROM | 1S20 | —— | oversize | | | | | | | | 0.13 $\mu$m |
| AES-4SM | 1S10 | 10220 LEs | 0 | 80 K-bit (20 M4Ks) | 21 | 48.24 | 6.18 | 77.19 | 15340 LEs | 0.403 Mbps/LE | 0.13 $\mu$m |
| AES-8SM | 1S10 | 10237 LEs | 0 | 160 K-bit (40 M4Ks) | 21 | 96.10 | 12.30 | 76.88 | 20477 LEs | 0.601 Mbps/LE | 0.13 $\mu$m |
| UNROLLING | 1S10 | —— | 0 | oversize | | | | | | | 0.13 $\mu$m |
| UNROLL_ROM | 1S10 | —— | oversize | | | | | | | | 0.13 $\mu$m |
| Standaert et al. [8](unrolling) | XCV3 200E8 | 2784 Slices | 80 K-bit (20 BRAMs) | 320 K-bit (80 BRAMs) | 21 | | 11.77 | 36.78 | 15584 Slices | 0.378 Mbps/Slice | 0.18 $\mu$m |
| Saggese et al. [5](unrolling) | XVE 2000 | 5810 Slices | 80 K-bit (20 BRAMs) | 320 K-bit (80 BRAMs) | 50 | | 20.30 | 63.44 | 18610 Slices | 0.545 Mbps/Slice | 0.18 $\mu$m |
| UF10-PP3B [6] (unrolling) | XC2V 4000 | 5142 Slices | 80 K-bit (20 BRAMs) | 320 K-bit (80 BRAMs) | 30 | | 23.50 | 73.44 | 17942 Slices | 0.655 Mbps/Slice | 0.12 / 0.15 $\mu$m |
| UF1-PP0B [6] (rolling) | XC2V 4000 | 387 Slices | 8 K-bit (2 BRAMs) | 32 K-bit (8 BRAMs) | 10 | | 1.41 | 44.06 | 1667 Slices | 0.423 Mbps/Slice | 0.12 / 0.15 $\mu$m |
| Helion [2] (rolling) | Stratix -C5 | 1023 LEs | 8 K-bit (2 M4Ks) | 32 K-bit (8 M4Ks) | 10 | | 1.40 | 43.75 | 3583 LEs | 0.391 Mbps/LE | 0.13 $\mu$m |

1S10: Altera Stratix EP1S10F780C5;1S20: Altera Stratix EP1S20F780C5; 1S25: Altera Stratix EP1S25F780C5
LE: Contains one 4-input look-up tables; Slice: Contains two 4-input look-up tables; BRAM: Block Selected RAM (4 K-bit)
Eq-LE: All equivalent LEs where one M4K is equivalent to 256 LEs; Eq-Sli: All equivalent slices where one BRAM is equivalent to 128 slices
LE-eff: Throughput/Eq-LE; Sli-eff: Throughput/Eq-Sli

figuration time. Since an M4K implements two separate S-Boxes, 8 copies of the M4K are sufficient for each SubBytes that contains 16 S-Boxes. As for the key expansion block, each round key circuit consists of 6 XOR gates and 4 combinational S-Boxes that are implemented by LEs, where the combinational S-box is realized by an average of 45 LEs for the EP1S20F780C5 device. Note that the number of LEs for the combinational S-box would be slightly different for the different devices.

We also designed the unrolling implementation with the same FPGA for comparison. In this design, the SubBytes was implemented by M4Ks, the MixColumns and the AddRoundKey were implemented by LEs, and the ShiftRows was simply realized by hardwiring. To realize the S-Box of the key expansion block in the unrolling implementation, we use both the look up table method where two S-Boxes were implemented by one M4K, and the Galois Field operations method where the combinational S-box was realized by LEs.

## 6. Performance and Comparisons

In this section, we evaluate the performance of the AES-4SM and the AES-8SM, and compare with the unrolling implementations called UNROLLING, UNROLL_ROM (designed by us) and other published works.

### 6.1 Various Implementations

The AES-4SM is implemented with rolling part 4SM, while the AES-8SM is implemented with rolling part 8SM. Both the UNROLLING and the UNROLL_ROM are implemented with unrolling architecture. The former adopts the combinational S-Boxes in the key expansion block but adopts ROMs for the S-Boxes in the round circuit, and the latter adopts ROMs for the S-Boxes in both the key expansion block and the round circuit. To compare the performance of different architectures, we designed both the proposed implementations and the unrolling implementations on the same FPGA. For each implementation, first we described the circuit by Verilog HDL, and then used Quartus II 4.1 for synthesis, place & route and timing analysis. Finally, we used the Quartus II 4.1 simulator to test the logical operation and to do the worst-case timing analysis for the design in the target FPGA with the test vectors available from CSRC (computer security resource ceneter) of NIST [16]. The maximum clock rate (F_clk) was obtained by the Quartus II 4.1 simulator.

In Table 2, the upper twelve rows show the results of our implementations AES-4SM, AES-8SM, UNROLLING and UNROLL_ROM on three Altera Stratix devices, while the lower five rows show the previous results done by other groups. The column "Device" denotes the FPGA used. The column "LE / Slice" denotes the number of LEs utilization of Altera FPGA or the number of occupied slices of Xilinx FPGA. The column "Memory for key" denotes the amount of memory utilized for the key expansion block, and "Memory for core" denotes the amount of memory utilized for the core, where one M4K is equivalent to 4096 bits. The column "Cycle" denotes the number of clock cycles for the process of the whole AES rounds. The column "Th." denotes the maximum **throughput** calculated by:

$$\text{Th.} = 128 \cdot \text{F\_clk} \ .$$

The column "Mem-eff" shows the **memory efficiency** calculated by:

$$\text{Mem-eff} = \frac{\text{Throughput (Mbps)}}{\text{Memory for core (K bits)}} \ . \tag{1}$$

## 6.2 Memory Efficiency

In the Altera Stratix Device family [17], EP1S10F780C5 is the minimum-sized device, containing 10570 LEs and 60 M4Ks. EP1S20F780C5 is the medium-sized device, containing 18460 LEs and 82 M4Ks. EP1S25F780C5 is the large-sized device, containing 25660 LEs and 138 M4Ks.

On EP1S25F780C5 (the large-sized device), all of our designs can be implemented. The UNROLL_ROM has a slightly higher throughput than the UNROLLING. On this device, the UNROLL_ROM uses 80 K-bits more memory than the UNROLLING, however, the UNROLL_ROM uses just 620 fewer LEs than the UNROLLING. Since the memory efficiency only considers the memory for the core, the UNROLL_ROM has a slightly higher memory efficiency than the UNROLLING, but has still lower memory efficiency than both the AES-4SM and the AES-8SM.

On EP1S20F780C5 (the medium-sized device), the AES-4SM achieves a throughput of 6.45 Gbps by using 20 M4Ks, and the AES-8SM achieves a throughput of 12.78 Gbps by using 40 M4Ks. Compared with the UNROLLING that achieves a throughput of 22.75 Gbps using 80 M4Ks, the AES-4SM and the AES-8SM improve the memory efficiency by 13.4% and 12.3%, respectively, and reduce the amount of the memory by 75% and 50%, respectively. This device is too small for the UNROLL_ROM because both of the memory for key and the memory for core requires 100 copies of the M4K, which exceeds the maximum number of the M4Ks of the EP1S20F780C5.

On EP1S10F780C5 (the minimum-sized device), both of the AES-4SM and the AES-8SM can be implemented where 96% of LEs are utilized. However, this device is too small for both the UNROLL_ROM and the UNROLLING due to the limitation of the number of M4Ks.

Helion [2] designed the rolling implementation on Altera Stratix FPGA as shown in the last row of Table 2. Since the LEs and memory used by Helion is less than that of the EP1S10F780C5, we can conjecture that this design fits into the same device. Although our proposed implementations on the EP1S10F780C5 achieve lower throughputs than the implementations on other two devices, compared with Helion that achieves 1.4 Gbps, the AES-4SM and the AES-8SM increase the throughput by 4.41 times and 8.79 times, respectively, and improve the memory efficiency by 13.4% and 12.3%, respectively.

The upper part of the Table 2 shows that the AES-4SM always has the highest memory efficiency, and the AES-8SM has a slightly lower memory efficiency than that of the AES-4SM where the implemented devices are different. Also, only the AES-4SM, the AES-8SM and the rolling implementation [2] fit into the smallest Strtix device. Hence, the proposed implementations are also suitable for the smaller FPGA implementation. In addition, the AES-4SM and the AES-8SM fill the gap between the rolling implementation and the unrolling implementation.

Direct comparison among various FPGA implementations of the AES algorithms is difficult, since FPGA target devices are usually different. However, many AES implementations provide the maximum throughputs and the amount of the memory utilized for the core. Thus, we can compare the memory efficiency defined in (1).

Compared with the published unrolling implementations signified with "(unrolling)" in the column "Design", the AES-4SM and the AES-8SM on the EP1S20F780C5 have 9.83% and 8.76% higher memory efficiencies than the fastest implementation (UF10-PP3B). Note that the number of cycles for UF10-PP3B is 30, while the number of cycles for our proposed implementations is 21. Besides, the amounts of the memory utilized for the core of the AES-4SM and the AES-8SM are reduced by 75% and 50%, respectively.

Compared with the published rolling implementations signified with "(rolling)" in the column "Design", both the throughputs and the memory efficiencies of the proposed implementations are much higher than the fastest rolling implementation (UF1-PP0B).

Figure 9 illustrates the relation of Memory for core and Throughput in Table 2. The gradient of the line shows the

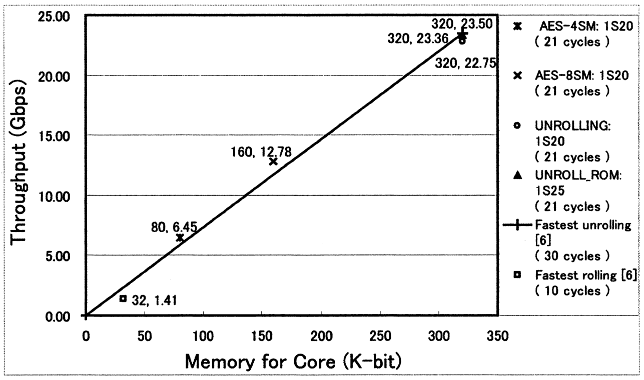


**Fig. 9** An illustration of the memory efficiency in Table 2.

memory efficiency of the fastest design UF1-PP0B [6]. Both the AES-4SM and the AES-8SM have a higher memory efficiency than others.

### 6.3 Resource Efficiency

Here, we consider the efficiency of the utilized resource considering both LEs or slices and the block RAMs, i.e., the relation between the throughput with the utilized resource. Since the FPGA companies do not disclose the actual area for LEs or slices and the block RAMs, the exact evaluation of the efficiency of the utilized resource is difficult. Among the publications on the AES, the authors of [5] and [18] compare the efficiency of utilized resource (both slice and BRAM) where one BRAM is equivalent to 128 slices. Although Altera FPGA is different from Xilinx FPGA, both the LE and the slice consist of 4-input look up tables (LUTs), and both the M4K and the BRAM can be used to form the dual-port $256 \times 8$-bit RAM that is adopted in the AES design. Thus, we assume that one BRAM is equivalent to 128 slices, and one M4K is equivalent to 256 LEs, since the number of LUTs in a slice is twice of that in a LE.

In Table 2, the column Eq-LE and Eq-Sli denote the number of the equivalent LEs and the number of the equivalent Slices, respectively, where one M4K is replaced by 256 LEs and one BRAM is replaced by 128 slices. In the column "LE-eff / $\frac{\text{Sli-eff}}{2}$", LE-eff and Sli-eff show the **resource efficiency** of Altera FPGA and Xilinx FPGA, respectively. They are calculated by:

$$\text{LE-eff} = \frac{\text{Throughput (Mbps)}}{\text{Eq-LE (LE)}},$$
$$\text{Sli-eff} = \frac{\text{Throughput (Mbps)}}{\text{Eq-Sli (slice)}}.$$

Since one slice is equivalent to two LEs with regards the number of LUTs, Sli-eff is divided by two as shown in Table 2. In this regards, the UNROLLING on the EP1S25F780C5 (large-sized FPGA) has the highest resource efficiency. Although the AES-8SM and the AES-4SM are less efficient than the highest resource efficiency, both of them can be implemented on the EP1S10F780C5 (minimum-sized FPGA) while the UNROLLING cannot.

### 6.4 Features of PPR

Both of the AES-4SM and the AES-8SM have much higher throughput than software implementations. An AES rolling implementation achieves 1.538 Gbps on a 3.2 GHz Pentium4 processor [19] and a 640 Mbps on a 1 GHz embedded processor [20].

Table 3 compares the features of the three different architectures: unrolling, PPR, and rolling. In the columns "Memory efficiency" and "Resource efficiency", the values of Helion [2] are set to be 1. We can see that the PPR architecture offers a high memory efficiency, and the AES-8SM offers a medium resource efficiency, but the AES-4SM offers the same resource efficiency as rolling architecture.

**Table 3** Comparison of the different architectures.

| Architecture | Memory area | Through-put | Memory efficiency | Resource efficiency |
|---|---|---|---|---|
| unrolling | Large | High | 0.84~1.68 | 0.97~2.05 |
| PPR(AES-4SM) | Small | Medium | 1.76~1.84 | 1.03~1.05 |
| PPR(AES-8SM) | Medium | High | 1.76~1.84 | 1.54~1.57 |
| rolling | Small | Low | 1 | 1~1.08 |

## 7. Conclusions

In this paper, we presented the pipelined partial rolling (PPR) architecture for an AES encryption processor. We implemented two different designs: AES-4SM and AES-8SM on three different-sized Altera Stratix FPGAs. On the medium-sized FPGA, the AES-4SM achieves a throughput of 6.45 Gbps by using 20 M4Ks, and the AES-8SM achieves a throughput of 12.78 Gbps by using 40 M4Ks. Compared with the unrolling implementation that achieves a throughput of 22.75 Gbps by using 80 M4Ks on the same FPGA, the AES-4SM and the AES-8SM improve the memory efficiency by 13.4% and 12.3%, respectively, and reduce the amount of the memory by 75% and 50%, respectively. Compared with the existing FPGA designs, the proposed implementations have a higher memory efficiency than all the prior implementations known to the authors. Although the AES-8SM offers a medium resource efficiency and the AES-4SM offers almost the same resource efficiency as rolling implementation, both of them can be implemented on the minimum-sized FPGA while the unrolling implementation cannot. The PPR architecture fills the gap between unrolling and rolling architectures and is suitable for small and medium-sized FPGAs.

## Acknowledgments

This research is partly supported by JSPS, the Grant in Aid for Scientific Research, and MEXT, the Kitakyushu area innovative cluster project.

### References

[1] National Institute of Standards and Technology (NIST), Advanced Encryption Standard (AES), Federal Information Processing Standards Publications 197 (FIPS197), Nov. 2001.
[2] HELION Technology Limited, "High performance AES (Rijndael) cores for Altera FPGA," available at http://www.heliontech.com/core2.htm
[3] Amphion Semiconductor, "CS5210-40: High performance AES encryption cores," 2003, available at http://www.amphion.com/cs5210.htm
[4] N. Pramstaller and J. Wolkerstorfer, "A universal and efficient AES co-processor for field programmable logic arrays," FPL 2004, LNCS3203, pp.565–574, 2004.
[5] G.P. Saggese, A. Mazzeo, N. Mazzocca, and A.G.M. Strollo, "An FPGA-based performance analysis of the unrolling, tiling, and pipelining of the AES algorithm," FPL 2003, LNCS 2778, pp.292–302, 2003.

[6] J. Zambreno, D. Nguyen, and A.N. Choudhary, "Exploring area/delay tradeoffs in an AES FPGA implementation," FPL 2004, LNCS3203, pp.575–585, 2004.
[7] H. Qin, T. Sasao, and Y. Iguchi, "An FPGA design of AES encryption circuit with 128-bit keys," 15th IEEE /ACM Great Lakes Symposium on VLSI (GLSVLSI'05), pp.147–152, Chicago, IL, April 2005.
[8] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs," Proc. CHES 2003, LN CS2523, pp.334–350, Cologne, Germany, Springer-Verlag, Sept. 2003.
[9] F. Charot, E. Yahya, and C. Wagner, "Efficient modular-pipelined AES implementation in counter mode on ALTERA FPGA," FPL 2003, pp.282–291, Lisbon, Portugal, 2003.
[10] H. Qin, T. Sasao, M. Matsuura, S. Nagayama, K. Nakamura, and Y. Iguchi, "A realization of multiple-output functions by a look-up table ring," IEICE Trans. Fundamentals, vol.E87-A, no.12, pp.3141–3150, Dec. 2004.
[11] T. Sasao, M. Kusano, and M. Matsuura, "Optimization methods in look-up table rings," International Workshop on Logic and Synthesis (IWLS-2004), pp.431–437, Temecula, CA, June 2004.
[12] V. Rijmen, "Efficient implemenation of the Rijndael S-box," available at http://www.esat.kuleuven.ac.be/˜rijmen/rijndael/sbox.pdf
[13] J. Wolkerstorfer, E. Oswald, and M. Lamberger, "An ASIC implementation of the AES S-boxes," RSA 2002, LNCS 2271, pp.67–78, San Jose, CA, Feb. 2002.
[14] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and performance testing of a 2.29 Gb/s Rijndael processor," IEEE J. Solid-State Circuits, vol.38, no.3, pp.569–572, 2003.
[15] S. Morioka and A. Satoh, "An optimized S-Box circuit architecture for low power AES design," CHES 2002, LNCS 2523, pp.172–186, 2003.
[16] http://csrc.nist.gov/CryptoToolkit/aes/rijndael/
[17] http://www.altera.com
[18] X. Zhang and K.K. Parhi, "High-speed VLSI architectures for the AES algorithm," IEEE Trans. Very Large Scale Intergr. (VLSI) Syst., vol.12, no.9, pp.957–967, Sept. 2004.
[19] H. Lipmaa, "AES implementation speed comparison," available at http://www.tsc.hut.fi./ aes/rijndael.html,2003.
[20] K. Nadehara, M. Ikekawa, and I. Kuroda "Extended instructions for the AES cryptography and their efficient implementation," IEEE Workshop on Signal Processing System (SIPS'04), FA-1.3, Oct. 2004.

**Hui Qin** received the B.E. degree from Beijing University of Aeronautics and Astronautics, China, in 1994 and the M.E. degree from Kyushu Institute of Technology, Japan, in 2004. From 1994 to 2001, he worked at China Academy of Space Technology, involved in the design of China-Brazil Earth Resources Satellite. He received the Scientific Technology Progress Award for National Defense in China in 2000. Now he is working toward the Ph.D. degree in the Department of Computer Science and Electronics, Kyushu Institute of Technology. His research interests include reconfigurable architecture, complex systems design, software synthesis.

**Tsutomu Sasao** received the B.E., M.E., and Ph.D. degrees in Electronics Engineering from Osaka University, Osaka Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, IBM T.J. Watson Research Center, Yorktown Height, NY and the Naval Postgraduate School, Monterey, CA. Now, he is a Professor of Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than 9 books on logic design including, Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis, and Logic Synthesis and Verification, Kluwer Academic Publishers 1993, 1996, 1999, 2001 respectively. He has served Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan in 1998. He received the NIWA Memorial Award in 1979, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC in 1987, 1996, 2003, and 2004 for papers presented at ISMVLs, and Takeda Techno-Entrepreneurship Award in 2001. He has served an associate editor of the IEEE Transactions on Computers. Currently, he was the Chairman of the Technical Committee on Multiple-Valued Logic, IEEE Computer Society. He is a Fellow of the IEEE.

**Yukihiro Iguchi** was born in Tokyo, and received the B. E., M. E., and Ph. D. degree in electronic engineering from Meiji University, Kanagawa Japan, in 1982, 1984, and 1987, respectively. He is now an associate professor of Meiji University. His research interest includes logic design, switching theory, and reconfigurable systems. In 1996, he spent a year at Kyushu Institute of Technology. He received Takeda Techno-Entrepreneurship Award in 2001.