International Journal of Internet, Broadcasting and Communication Vol.8 No.1 42-52 (2016)
http://dx.doi.org/10.7236/IJIBC.2016.8.1.42

IJIBC 16-1-6

Optimization of ARIA Block-Cipher Algorithm for Embedded Systems with 16-bits Processors

Wan Yeon Lee[1*], Yun-Seok Choi[2]

[1] *Dept. of Computer Science, Dongduk Women's University, Seoul 136-714, South Korea,*
Email: wanlee@dongduk.ac.kr
[2] *Dept. of Computer Science, Dongduk Women's University, Seoul 136-714, South Korea,*
Email: cooling@ dongduk.ac.kr

Abstract

In this paper, we propose the 16-bits optimization design of the ARIA block-cipher algorithm for embedded systems with 16-bits processors. The proposed design adopts 16-bits XOR operations and rotated shift operations as many as possible. Also, the proposed design extends 8-bits array variables into 16-bits array variables for faster chained matrix multiplication. In evaluation experiments, our design is compared to the previous 32-bits optimized design and 8-bits optimized design. Our 16-bits optimized design yields about 20% faster execution speed and about 28% smaller footprint than 32-bits optimized code. Also, our design yields about 91% faster execution speed with larger footprint than 8-bits optimized code.

Keywords: *Block cipher, ARIA, 16-bits Processor, Optimization, Embedded System*

1. Introduction

With rapid growth of information technology and science of encryption, many cryptographic products have been developed for secure communications [1,2,3]. Symmetric-key cryptographic algorithms apply a shared secret key to both encryption and decryption processes. Asymmetric-key cryptographic algorithms apply a public key to encryption process and a private key to decryption process. Popular symmetric-key algorithms are data encryption standard (DES), triple encryption with DES (Triple-DES) and advanced encryption standard (AES) [4]. The AES algorithm [5] is the most prevalently used for commercial markets. Popular asymmetric-key algorithms are RSA, Diffie-Hellman and elliptic-curve cryptography (ECC) [4].

Recently, a 128-bits block cipher called ARIA [6] is proposed. The ARIA is a symmetric-key cryptography algorithm and includes an involutional substitution and permutation encryption network (SPN) which is not totally involutional. Also, the ARIA contains a diffusion layer to resist against powerful attacks such as collision attacks, partial sum attacks and truncated differential attacks [6].

Manuscript Received: Dec. 7, 2015 / Revised: Dec. 23, 2015/ Accepted: Jan. 8, 2016
Corresponding Author: wanlee@dongduk.ac.kr
Tel:, +82-2-940-4685 Fax: +82-2-940-4170
Dept. of Computer Science, Dongduk Women's University, Seoul 136-714, South Korea

The original design of ARIA algorithm has many 8-bits computations and thus suitable for 8-bits processors. Another design is given to be suitable for 32-bits processors [7]. However, there is no design suitable for 16-bits processors although 16-bits processors are widely employed by many embedded systems [8,9].

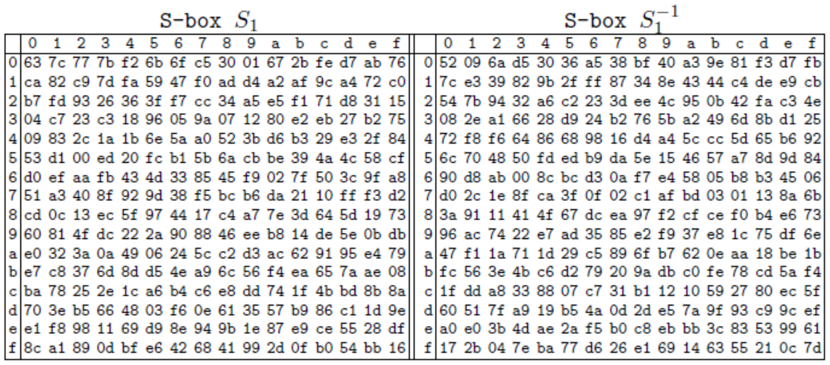
The proposed design optimizes the execution speed of the ARIA algorithm for embedded systems with 16-bits processors. The ARIA algorithm consists of three parts: round key addition, substitution layer, and diffusion layer. The proposed design first modifies the diffusion layer so as to reduce the number of XOR operations requested for the chained matrix multiplication calculation of the diffusion layer. Our design applies 16-bits XOR operation instead of 8-bits XOR operations. This modification of the diffusion layer requires 16-bits extensions of 8-bits array variables used in the substitution layer. The proposed design extends 8-bits array variables into 16-bits array variable for faster matrix multiplication calculation of the diffusion layer. Also, the proposed design employs 16-bits XOR operations and rotated shift operations as many as possible using the extended 16-bits array variables.

In experiments with ATmega2560 micro-controllers, yields about 20% faster execution speed and about 28% smaller size (footprint) of translated machine code than the 32-bits optimized code. Compared to the 8-bits optimized code, our code yields about 91% faster execution speed with a larger space requirement.

The rest of this paper is organized as follows; Section 2 explains the outline of the original ARIA algorithm. Section 3 describes the proposed design in detail. Section 4 shows evaluation results of the proposed design. Section 5 provides concluding remarks.

2. Outline of ARIA Algorithm

The ARIA algorithm consists of three parts: round key addition, substitution layer, and diffusion layer. The round key addition part performs XOR operations of the 128-bit round key. The XOR operation is denoted as ⊕. The substitution layer part replaces inputs with their position-corresponding values in two types of S-Boxes (S_1, S_2) and their inverses (S_1^{-1}, S_2^{-1}). The pre-calculated values of S_1, S_2, S_1^{-1}, and S_2^{-1} are stored in constant array variables, as shown in Figure 1 and Figure 2. For example, S_1(0x01)=0x7c, S_2(0x02)=0x54, S_1^{-1} (0xf0)=0x17, and S_2^{-1}(0xf1)=0x8a.



S-box S_1

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

S-box S_1^{-1}

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 1. Values of S-box S_1 and S_1^{-1}

S-box S_2

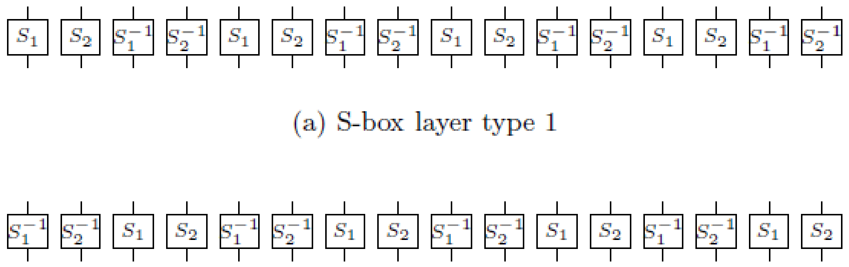
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	e2	4e	54	fc	94	c2	4a	cc	62	0d	6a	46	3c	4d	8b	d1
1	5e	fa	64	cb	b4	97	be	2b	bc	77	2e	03	d3	19	59	c1
2	1d	06	41	6b	55	f0	99	69	ea	9c	18	ae	63	df	e7	bb
3	00	73	66	fb	96	4c	85	e4	3a	09	45	aa	0f	ee	10	eb
4	2d	7f	f4	29	ac	cf	ad	91	8d	78	c8	95	f9	2f	ce	cd
5	08	7a	88	38	5c	83	2a	28	47	db	b8	c7	93	a4	12	53
6	ff	87	0e	31	36	21	58	48	01	8e	37	74	32	ca	e9	b1
7	b7	ab	0c	d7	c4	56	42	26	07	98	60	d9	b6	b9	11	40
8	ec	20	8c	bd	a0	c9	84	4	49	23	f1	4f	50	1f	13	dc
9	d8	c0	9e	57	e3	c3	7b	65	3b	02	8f	3e	e8	25	92	e5
a	15	dd	fd	17	a9	bf	d4	9a	7e	c5	39	67	fe	76	9d	43
b	a7	e1	d0	f5	68	f2	1b	34	70	05	a3	8a	d5	79	86	a8
c	30	c6	51	4b	1e	a6	27	f6	35	d2	6e	24	16	82	5f	da
d	e6	75	a2	ef	2c	b2	1c	9f	5d	6f	80	0a	72	44	9b	6c
e	90	b	5b	33	7d	5a	52	f3	61	a1	f7	b0	d6	3f	7c	6d
f	ed	14	e0	a5	3d	22	b3	f8	89	de	71	1a	af	ba	b5	81

S-box S_2^{-1}

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	30	68	99	1b	87	b9	21	78	50	39	db	e1	72	9	62	3c
1	3e	7e	5e	8e	f1	a0	cc	a3	2a	1d	fb	b6	d6	20	c4	8d
2	81	65	f5	89	cb	9d	77	c6	57	43	56	17	d4	40	1a	4d
3	c0	63	6c	e3	b7	c8	64	6a	53	aa	38	98	0c	f4	9b	ed
4	7f	22	76	af	dd	3a	0b	58	67	88	06	c3	35	0d	01	8b
5	8c	c2	e6	5f	02	24	75	93	66	1e	e5	e2	54	d8	10	ce
6	7a	e8	8	2c	12	97	32	ab	b4	27	0a	23	df	ef	ca	d9
7	b8	fa	dc	31	6b	d1	ad	19	49	bd	51	96	ee	e4	a8	41
8	da	ff	cd	55	86	36	be	61	52	f8	bb	0e	82	48	69	9a
9	e0	47	9e	5c	04	4b	34	15	79	26	a7	de	29	ae	92	d7
a	84	e9	d2	ba	5d	f3	c5	b0	bf	a4	3b	71	44	46	2b	fc
b	eb	6f	d5	f6	14	fe	7c	70	5a	7d	fd	2f	18	83	16	a5
c	91	1f	05	95	74	a9	c1	5b	4a	85	6d	13	07	4f	4e	45
d	b2	0f	c9	1c	a6	bc	ec	73	90	7b	cf	59	8f	a1	f9	2d
e	f2	b1	00	94	37	9f	d0	2e	9c	6e	28	3f	80	f0	3d	d3
f	25	8a	b5	e7	42	b3	c7	ea	f7	4c	11	33	03	a2	ac	60

Figure 2. Values of S-box *S*$_2$ and *S*$_2^{-1}$

There are two types of S-box layers as shown in Figure 3. Type 1 is used in odd rounds and type 2 is used in even rounds.



(a) S-box layer type 1

(b) S-box layer type 2

Figure 3. Two Types of S-box Layers

The diffusion layer generates an output vector (y_0, …, y_{15}) with an input vector of (x_0, …, x_{15}), where

$$
\begin{aligned}
y_0 &= x_3 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_9 \oplus x_{13} \oplus x_{14}, & y_8 &= x_0 \oplus x_1 \oplus x_4 \oplus x_7 \oplus x_{10} \oplus x_{13} \oplus x_{15},\\
y_1 &= x_2 \oplus x_5 \oplus x_7 \oplus x_8 \oplus x_9 \oplus x_{12} \oplus x_{15}, & y_9 &= x_0 \oplus x_1 \oplus x_5 \oplus x_6 \oplus x_{11} \oplus x_{12} \oplus x_{14},\\
y_2 &= x_1 \oplus x_4 \oplus x_6 \oplus x_{10} \oplus x_{11} \oplus x_{12} \oplus x_{15}, & y_{10} &= x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_8 \oplus x_{13} \oplus x_{15},\\
y_3 &= x_0 \oplus x_5 \oplus x_7 \oplus x_{10} \oplus x_{11} \oplus x_{13} \oplus x_{14}, & y_{11} &= x_2 \oplus x_3 \oplus x_4 \oplus x_7 \oplus x_9 \oplus x_{12} \oplus x_{14},\\
y_4 &= x_0 \oplus x_2 \oplus x_5 \oplus x_8 \oplus x_{11} \oplus x_{14} \oplus x_{15}, & y_{12} &= x_1 \oplus x_2 \oplus x_6 \oplus x_7 \oplus x_9 \oplus x_{11} \oplus x_{12},\\
y_5 &= x_1 \oplus x_3 \oplus x_4 \oplus x_9 \oplus x_{10} \oplus x_{14} \oplus x_{15}, & y_{13} &= x_0 \oplus x_3 \oplus x_6 \oplus x_7 \oplus x_8 \oplus x_{10} \oplus x_{13},\\
y_6 &= x_0 \oplus x_2 \oplus x_7 \oplus x_9 \oplus x_{10} \oplus x_{12} \oplus x_{13}, & y_{14} &= x_0 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_9 \oplus x_{11} \oplus x_{14},\\
y_7 &= x_1 \oplus x_3 \oplus x_6 \oplus x_8 \oplus x_{11} \oplus x_{12} \oplus x_{13}, & y_{15} &= x_1 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_8 \oplus x_{10} \oplus x_{15}.
\end{aligned}
$$

The diffusion layer can be expressed with an equivalent matrix multiplication as shown in Figure 4.

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{pmatrix} = \begin{pmatrix} 0&0&0&1&1&0&1&0&1&1&0&0&0&1&1&0 \\ 0&0&1&0&0&1&0&1&1&1&0&0&1&0&0&1 \\ 0&1&0&0&1&0&1&0&0&0&1&1&1&0&0&1 \\ 1&0&0&0&0&1&0&1&0&0&1&1&0&1&1&0 \\ 1&0&1&0&0&1&0&0&1&0&0&1&0&0&1&1 \\ 0&1&0&1&1&0&0&0&0&1&1&0&0&0&1&1 \\ 1&0&1&0&0&0&0&1&0&1&1&0&1&1&0&0 \\ 0&1&0&1&0&0&1&0&1&0&0&1&1&1&0&0 \\ 1&1&0&0&1&0&0&1&0&0&1&0&0&1&0&1 \\ 1&1&0&0&0&1&1&0&0&0&0&1&1&0&1&0 \\ 0&0&1&1&0&1&1&0&1&0&0&0&0&1&0&1 \\ 0&0&1&1&1&0&0&1&0&1&0&0&1&0&1&0 \\ 0&1&1&0&0&0&1&1&0&1&0&1&1&0&0&0 \\ 1&0&0&1&0&0&1&1&1&0&1&0&0&1&0&0 \\ 1&0&0&1&1&1&0&0&0&1&0&1&0&0&1&0 \\ 0&1&1&0&1&1&0&0&1&0&1&0&0&0&0&1 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{pmatrix}$$

Figure 4. Matrix Multiplication for the Diffusion Layer

The encryption and decryption processes of an *n*-round ARIA are shown in Figure 5, when *n* is 10, 12 and 14 for 128-bit, 192-bit and 256-bit secret keys, respectively. The encryption and decryption processes are identical except in the use of round keys, ek_i.

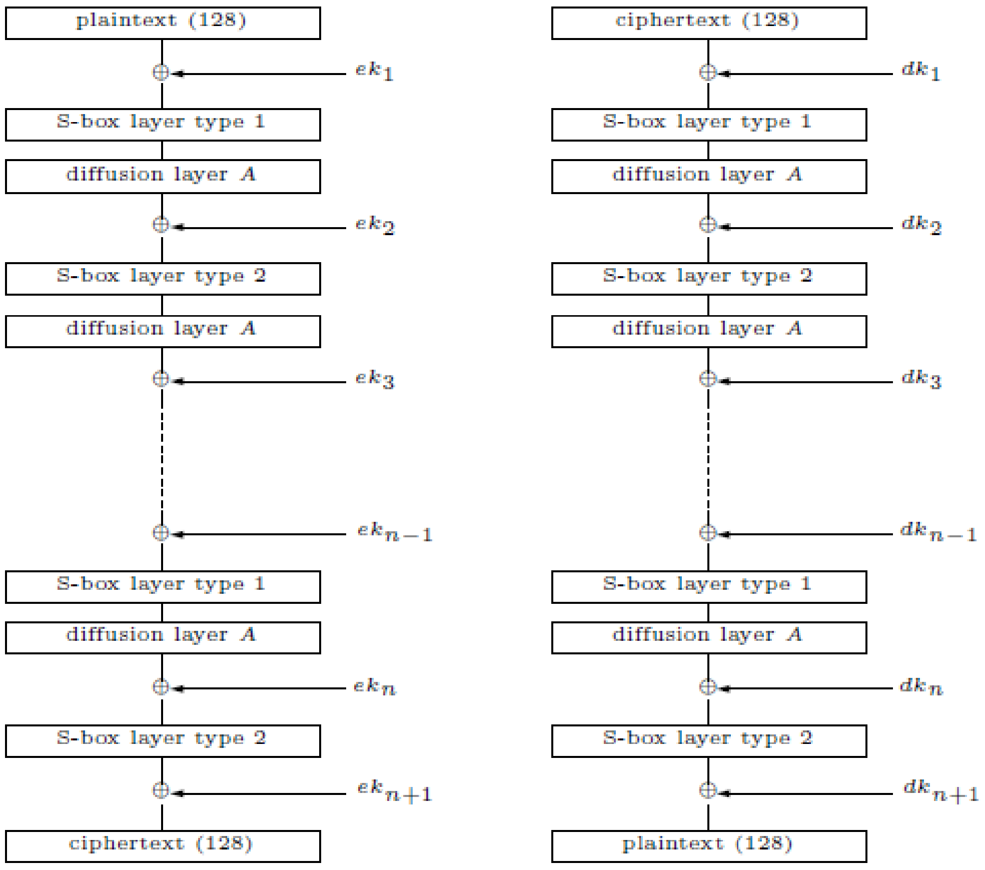


Figure 5. Encryption and Decryption Processes

The round keys, ek_i, are generated according to the following operation, where "<<< n" denotes the left circular rotation by n bits and ">>> n" denotes the right circular rotation by n bits. Four 128-bits values W_1, W_2, W_3 and W_4 are generated from the mater key by using 3-round 256-bits Feistel cipher [3].

$$
\begin{aligned}
ek_1 &= (W_0^{\ggg 7}) \oplus (W_1^{\lll 11}), & ek_2 &= (W_1^{\lll 22}) \oplus (W_2),\\
ek_3 &= (W_2^{\ggg 17}) \oplus (W_3^{\lll 16}), & ek_4 &= (W_0^{\ggg 14}) \oplus (W_3^{\lll 32}),\\
ek_5 &= (W_0^{\ggg 21}) \oplus (W_2^{\ggg 34}), & ek_6 &= (W_1^{\lll 33}) \oplus (W_3^{\lll 48})\\
ek_7 &= (W_1^{\lll 44}) \oplus (W_2^{\ggg 51}), & ek_8 &= (W_0^{\ggg 28}) \oplus (W_3^{\lll 64}),\\
ek_9 &= (W_1^{\lll 55}) \oplus (W_3^{\lll 80}), & ek_{10} &= (W_0^{\ggg 35}) \oplus (W_2^{\ggg 68}),\\
ek_{11} &= (W_0^{\ggg 42}) \oplus (W_1^{\lll 66}), & ek_{12} &= (W_1^{\lll 77}) \oplus (W_2^{\ggg 85}) \oplus (W_3^{\lll 96}),\\
ek_{13} &= (W_0^{\ggg 49}) \oplus (W_2^{\ggg 102}), & ek_{14} &= (W_2^{\ggg 119}) \oplus (W_3^{\lll 112}) \oplus (KR^{\lll 64}),\\
ek_{15} &= (W_0^{\ggg 56}) \oplus (W_1^{\lll 88}) \oplus (KR). & &
\end{aligned}
$$

Figure 6. Calculation for the Round Key Generation

3. Proposed Design

The proposed method modifies the diffusion layer design of the 32-bits optimized method [3] so as to work with 16-bits computations as follows; The matrix multiplication of the diffusion shown in Figure 4 can be reformulated with a matrix form $A = M_1^{-1} \cdot M_2 \cdot M_1$, where M_2 is a 16×16 involutional block diagonal matrix and M_1 is a 16×16 involutional matrix (i.e., $M_1^{-1}=M_1$). The matrix M_2 has a form $M_2 = P \cdot M$.

$$
M_1 = \begin{pmatrix} I & I & I & 0\\ I & 0 & I & I\\ I & I & 0 & I\\ 0 & I & I & I \end{pmatrix},\quad
P = \begin{pmatrix} I & 0 & 0 & 0\\ 0 & P_1 & 0 & 0\\ 0 & 0 & P_2 & 0\\ 0 & 0 & 0 & P_3 \end{pmatrix},\quad
M = \begin{pmatrix} T & 0 & 0 & 0\\ 0 & T & 0 & 0\\ 0 & 0 & T & 0\\ 0 & 0 & 0 & T \end{pmatrix}
$$

$$
T = \begin{pmatrix} 0 & 1 & 1 & 1\\ 1 & 0 & 1 & 1\\ 1 & 1 & 0 & 1\\ 1 & 1 & 1 & 0 \end{pmatrix},\quad
P_1 = \begin{pmatrix} 0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0 \end{pmatrix},
$$

$$
P_2 = \begin{pmatrix} 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 \end{pmatrix},\quad
P_3 = \begin{pmatrix} 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0 \end{pmatrix}
$$

When $\boldsymbol{S}$ denotes the S-box substitution, the one round except for key addition can be written as $A \cdot \boldsymbol{S} = M_1^{-1} \cdot M_2 \cdot M_1 \cdot \boldsymbol{S} = M_1 \cdot M_2 \cdot M_1 \cdot \boldsymbol{S}$ [3]. Because the form $M_1 \cdot \boldsymbol{S}$ is not implemented efficiently by 8×32 table lookups, the formulation of the diffusion layer is modified as follows:

$$
A \cdot \boldsymbol{S} = M_1 \cdot M_2 \cdot M_1 \cdot \boldsymbol{S} = M_1 \cdot P \cdot M \cdot M_1 \cdot \boldsymbol{S} = M_1 \cdot P \cdot M \cdot M_1 \cdot M \cdot M \cdot \boldsymbol{S}
$$

$$= M_1 \cdot P \cdot M \cdot M \cdot M_1 \cdot M \cdot \boldsymbol{S} = M_1 \cdot P \cdot M_1 \cdot M \cdot \boldsymbol{S}.$$

Then $M \cdot \boldsymbol{S}$ can be expressed respectively for odd rounds and even rounds as follows:

$$\begin{pmatrix}
0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&0
\end{pmatrix} \cdot \boldsymbol{S}$$

where $\boldsymbol{S}$ in odd rounds is

$$\boldsymbol{S} = [S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}]^T$$

and $\boldsymbol{S}$ in even rounds is

$$\boldsymbol{S} = [S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2]^T.$$

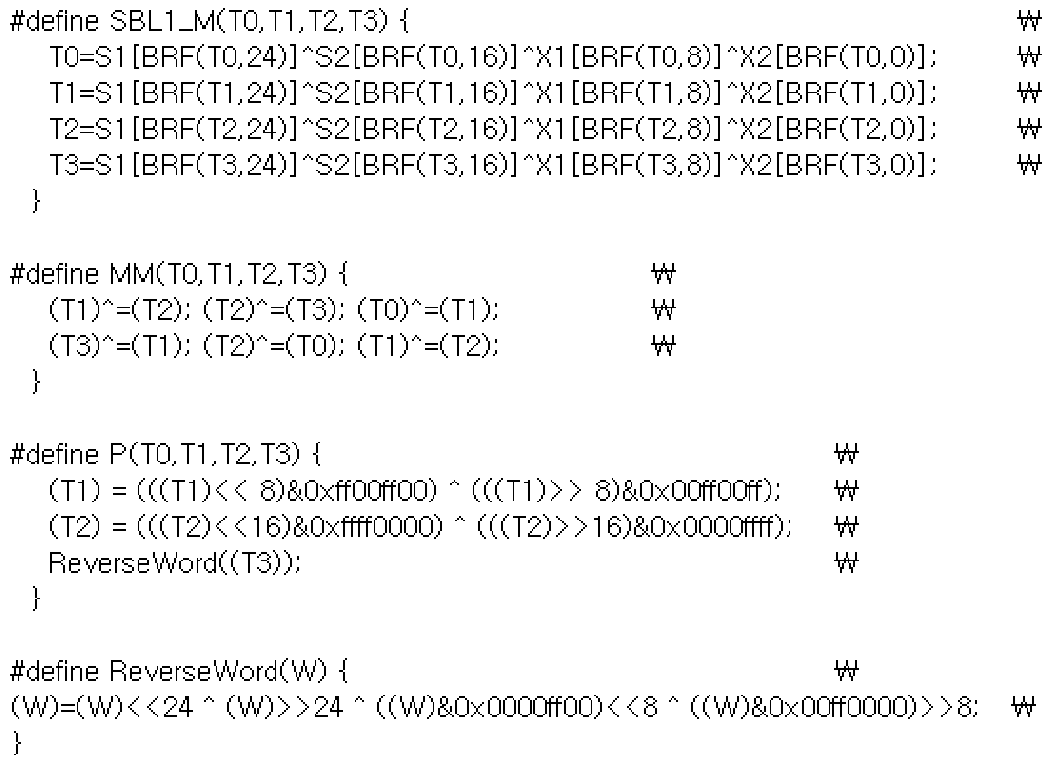
Let us check the left-upper quarter part of the above $M \cdot S$ calculation with $\boldsymbol{S} = [S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}, S_1, S_2, S_1^{-1}, S_2^{-1}]^T$. This part performs 24 XOR operations of 8-bits values such as S_1, S_2, S_1^{-1}, and $\underline{S}_2^{-1}$ as below:

$$\begin{pmatrix}
0&1&1&1&0&0&0&0\\
1&0&1&1&0&0&0&0\\
1&1&0&1&0&0&0&0\\
1&1&1&0&0&0&0&0\\
0&0&0&0&0&1&1&1\\
0&0&0&0&1&0&1&1\\
0&0&0&0&1&1&0&1\\
0&0&0&0&1&1&1&0
\end{pmatrix} \cdot \begin{pmatrix} S_1\\ S_2\\ S_1^{-1}\\ S_2^{-1}\\ S_1\\ S_2\\ S_1^{-1}\\ S_2^{-1} \end{pmatrix} = \begin{pmatrix}
0 \oplus S_2 \oplus S_1^{-1} \oplus S_2^{-1}\\
S_1 \oplus 0 \oplus S_1^{-1} \oplus S_2^{-1}\\
S_1 \oplus S_2 \oplus 0 \oplus S_2^{-1}\\
S_1 \oplus S_2 \oplus S_1^{-1} \oplus 0\\
0 \oplus S_2 \oplus S_1^{-1} \oplus S_2^{-1}\\
S_1 \oplus 0 \oplus S_1^{-1} \oplus S_2^{-1}\\
S_1 \oplus S_2 \oplus 0 \oplus S_2^{-1}\\
S_1 \oplus S_2 \oplus S_1^{-1} \oplus 0
\end{pmatrix}$$

For optimized operation of 16-bits processors, the above equation can be reformulated with 16-bits values such as 0x00S_1, 0x$S_2$00, 0x$S_1^{-1}S_1^{-1}$, 0x$S_2^{-1}S_2^{-1}$, 0xS_1S_1, 0xS_2S_2, 0x00S_1^{-1}, and 0xS_2^{-1}00 as below:

$$= \begin{pmatrix} \begin{bmatrix} 0 \\ S_1 \end{bmatrix}^T \oplus \begin{bmatrix} S_2 \\ 0 \end{bmatrix}^T \oplus \begin{bmatrix} S_1^{-1} \\ S_1^{-1} \end{bmatrix}^T \oplus \begin{bmatrix} S_2^{-1} \\ S_2^{-1} \end{bmatrix}^T \\ \begin{bmatrix} S_1 \\ S_1 \end{bmatrix}^T \oplus \begin{bmatrix} S_2 \\ S_2 \end{bmatrix}^T \oplus \begin{bmatrix} 0 \\ S_1^{-1} \end{bmatrix}^T \oplus \begin{bmatrix} S_2^{-1} \\ 0 \end{bmatrix}^T \\ \begin{bmatrix} 0 \\ S_1 \end{bmatrix}^T \oplus \begin{bmatrix} S_2 \\ 0 \end{bmatrix}^T \oplus \begin{bmatrix} S_1^{-1} \\ S_1^{-1} \end{bmatrix}^T \oplus \begin{bmatrix} S_2^{-1} \\ S_2^{-1} \end{bmatrix}^T \\ \begin{bmatrix} S_1 \\ S_1 \end{bmatrix}^T \oplus \begin{bmatrix} S_2 \\ S_2 \end{bmatrix}^T \oplus \begin{bmatrix} 0 \\ S_1^{-1} \end{bmatrix}^T \oplus \begin{bmatrix} S_2^{-1} \\ 0 \end{bmatrix}^T \end{pmatrix}$$

The above formulation performs 12 XOR operations of 16-bits values, whereas the original formulation performs 24 XOR operations of 8-bits values. As results, the modified formulation enhances the processing speed of 16-bits processors. On the contrary, the modified formulation requires the 16-bits extension of 8-bits values: for example, 0xS_1 → 0x00S_1 and 0xS_1 → 0xS_1S_1. For faster operation, the 16-bits extensions (such as 0x00S_1, 0x$S_2$00, 0x$S_1^{-1}S_1^{-1}$, 0x$S_2^{-1}S_2^{-1}$, 0xS_1S_1, 0xS_2S_2, 0x00S_1^{-1}, and 0xS_2^{-1}00) are pre-calculated and stored in array variables. For example, 0x00S_1(0x01)=0x007c, 0xS_2S_2(0x02)=0x5454, 0x00S_1^{-1}(0xf0)=0x0017, and 0x$S_2^{-1}S_2^{-1}$(0xf1)=0x8a8a.



```
#define SBL1_M(T0,T1,T2,T3) {                                          ₩
   T0=S1[BRF(T0,24)]^S2[BRF(T0,16)]^X1[BRF(T0,8)]^X2[BRF(T0,0)];      ₩
   T1=S1[BRF(T1,24)]^S2[BRF(T1,16)]^X1[BRF(T1,8)]^X2[BRF(T1,0)];      ₩
   T2=S1[BRF(T2,24)]^S2[BRF(T2,16)]^X1[BRF(T2,8)]^X2[BRF(T2,0)];      ₩
   T3=S1[BRF(T3,24)]^S2[BRF(T3,16)]^X1[BRF(T3,8)]^X2[BRF(T3,0)];      ₩
 }

#define MM(T0,T1,T2,T3) {                    ₩
   (T1)^=(T2); (T2)^=(T3); (T0)^=(T1);       ₩
   (T3)^=(T1); (T2)^=(T0); (T1)^=(T2);       ₩
 }

#define P(T0,T1,T2,T3) {                                          ₩
   (T1) = (((T1)<< 8)&0xff00ff00) ^ (((T1)>> 8)&0x00ff00ff);     ₩
   (T2) = (((T2)<<16)&0xffff0000) ^ (((T2)>>16)&0x0000ffff);     ₩
   ReverseWord((T3));                                             ₩
 }

#define ReverseWord(W) {                                          ₩
(W)=(W)<<24 ^ (W)>>24 ^ ((W)&0x0000ff00)<<8 ^ ((W)&0x00ff0000)>>8;   ₩
}
```

Figure 7. Codes for Matrix Operation $P \cdot M_1 \cdot M$ *before* Modification

The modified formulation requires the matrix operation $A \cdot \boldsymbol{S} = M_1 \cdot P \cdot M_1 \cdot M \cdot \boldsymbol{S}$ to be modified to calculate with the 16-bits extended values. Figure 7 shows the code before the modification. The matrix operation M is denoted with a macro code, **SBL1_M(T0, T1, T2, T3)**, where T0, T1, T2 and T3 are 32-

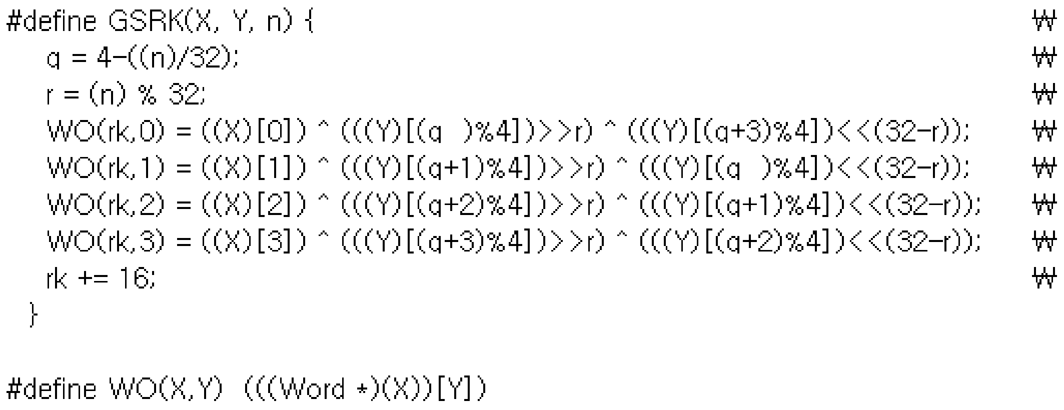
bits variables. The matrix operations M_1 is denoted with a macro code, **MM(T0, T1, T2, T3)**, and the matrix operation P is denoted with a macro code, **P(T0, T1, T2, T3)**.

Figure 8 shows the code after the modification, where t0, t1, t2 and t3 are the pointer of 16-bit array variables t0[2], t1[2], t2[2] and t3[2]. In the modified codes, __t[2] is a temporary 16-bit array variable.

```
#define SBL1_M( t0, t1, t2, t3 ) {                                                                    ₩
    __t[0] = t0[0]>>8; __t[1] = t0[0]&0x00ff; __t[2] = t0[1]>>8; __t[3] = t0[1]&0x00ff; ₩
    t0[0]=S1[__t[0]][0] ^ S2[__t[1]][0] ^ X1[__t[2]][0] ^ X2[__t[3]][0];               ₩
    t0[1]=S1[__t[0]][1] ^ S2[__t[1]][1] ^ X1[__t[2]][1] ^ X2[__t[3]][1];               ₩
    __t[0] = t1[0]>>8; __t[1] = t1[0]&0x00ff; __t[2] = t1[1]>>8; __t[3] = t1[1]&0x00ff; ₩
    t1[0]=S1[__t[0]][0] ^ S2[__t[1]][0] ^ X1[__t[2]][0] ^ X2[__t[3]][0];               ₩
    t1[1]=S1[__t[0]][1] ^ S2[__t[1]][1] ^ X1[__t[2]][1] ^ X2[__t[3]][1];               ₩
    __t[0] = t2[0]>>8; __t[1] = t2[0]&0x00ff; __t[2] = t2[1]>>8; __t[3] = t2[1]&0x00ff; ₩
    t2[0]=S1[__t[0]][0] ^ S2[__t[1]][0] ^ X1[__t[2]][0] ^ X2[__t[3]][0];               ₩
    t2[1]=S1[__t[0]][1] ^ S2[__t[1]][1] ^ X1[__t[2]][1] ^ X2[__t[3]][1];               ₩
    __t[0] = t3[0]>>8; __t[1] = t3[0]&0x00ff; __t[2] = t3[1]>>8; __t[3] = t3[1]&0x00ff; ₩
    t3[0]=S1[__t[0]][0] ^ S2[__t[1]][0] ^ X1[__t[2]][0] ^ X2[__t[3]][0];               ₩
    t3[1]=S1[__t[0]][1] ^ S2[__t[1]][1] ^ X1[__t[2]][1] ^ X2[__t[3]][1]; }

#define MM( t0, t1, t2, t3 ) {                                                                        ₩
    (t1[0])^=(t2[0]);  (t1[1])^=(t2[1]);   (t2[0])^=(t3[0]);   (t2[1])^=(t3[1]);           ₩
    (t0[0])^=(t1[0]);  (t0[1])^=(t1[1]);   (t3[0])^=(t1[0]);   (t3[1])^=(t1[1]);           ₩
    (t2[0])^=(t0[0]);  (t2[1])^=(t0[1]);   (t1[0])^=(t2[0]);   (t1[1])^=(t2[1]); }

#define P( t0, t1, t2, t3 ) {                                                                         ₩
    t1[0] = (((t1[0])<< 8)&0xff00) ^ (((t1[0])>> 8)&0x00ff);                                   ₩
    t1[1] = (((t1[1])<< 8)&0xff00) ^ (((t1[1])>> 8)&0x00ff);                                   ₩
    __t[0] = t2[0];  t2[0] = t2[1];  t2[1] = __t[0];                                            ₩
    __t[0] = t3[0];  t3[0] = swap16(t3[1]);   t3[1] = swap16(__t[0]);  }

#define swap16( x )   ( (x) >> 8 | (x) << 8 )
```

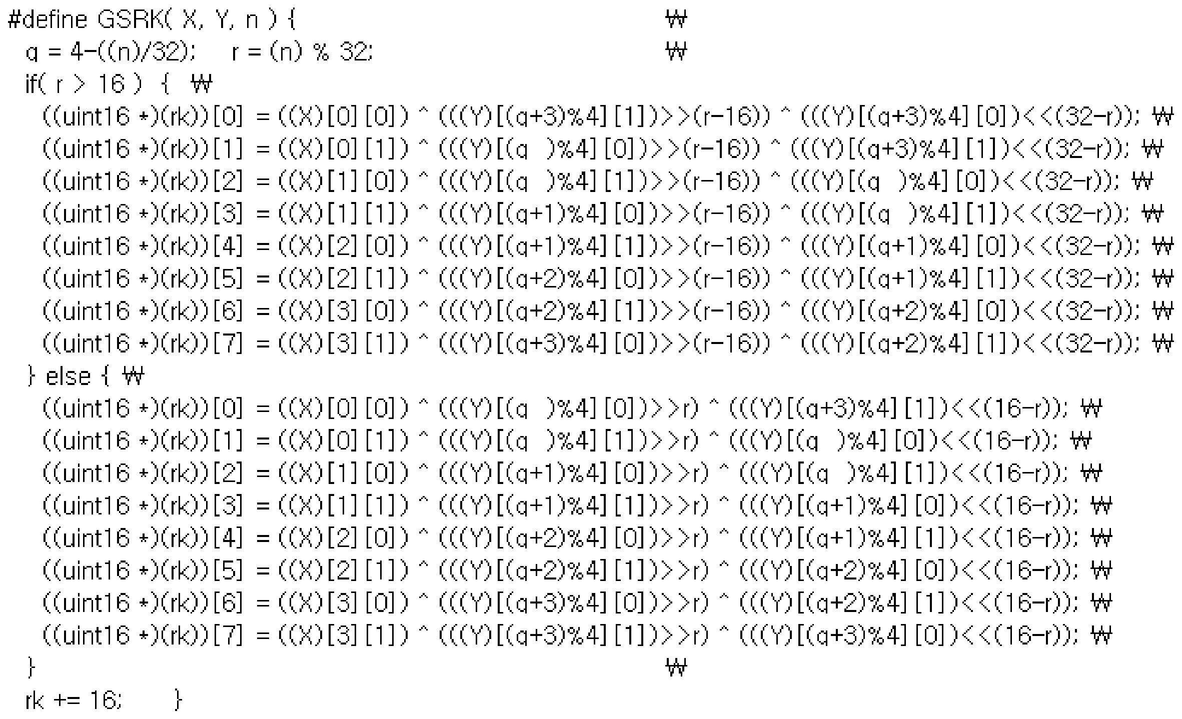
Figure 8. Codes for Matrix Operation $P \cdot M_1 \cdot M$ *after* Modification

Also the modified formulation requires the round key generation process, shown in Figure 6, to be modified to calculate with the 16-bits extended values. Figure 9 shows the circular rotation code after the modification, and Figure 10 shows the circular rotation code after the modification.



```
#define GSRK(X, Y, n) {                                                      ₩
    q = 4-((n)/32);                                                          ₩
    r = (n) % 32;                                                            ₩
    WO(rk,0) = ((X)[0]) ^ (((Y)[(q  )%4])>>r) ^ (((Y)[(q+3)%4])<<(32-r));    ₩
    WO(rk,1) = ((X)[1]) ^ (((Y)[(q+1)%4])>>r) ^ (((Y)[(q  )%4])<<(32-r));    ₩
    WO(rk,2) = ((X)[2]) ^ (((Y)[(q+2)%4])>>r) ^ (((Y)[(q+1)%4])<<(32-r));    ₩
    WO(rk,3) = ((X)[3]) ^ (((Y)[(q+3)%4])>>r) ^ (((Y)[(q+2)%4])<<(32-r));    ₩
    rk += 16;                                                                ₩
  }

#define WO(X,Y)  (((Word *)(X))[Y])
```

Figure 9. Code for Round Key Generation *before* Modification



```
#define GSRK( X, Y, n ) {                                  ₩
  q = 4-((n)/32);    r = (n) % 32;                         ₩
  if( r > 16 )  {  ₩
    ((uint16 *)(rk))[0] = ((X)[0] [0]) ^ (((Y)[(q+3)%4] [1])>>(r-16)) ^ (((Y)[(q+3)%4] [0])<<(32-r)); ₩
    ((uint16 *)(rk))[1] = ((X)[0] [1]) ^ (((Y)[(q  )%4] [0])>>(r-16)) ^ (((Y)[(q+3)%4] [1])<<(32-r)); ₩
    ((uint16 *)(rk))[2] = ((X)[1] [0]) ^ (((Y)[(q  )%4] [1])>>(r-16)) ^ (((Y)[(q  )%4] [0])<<(32-r)); ₩
    ((uint16 *)(rk))[3] = ((X)[1] [1]) ^ (((Y)[(q+1)%4] [0])>>(r-16)) ^ (((Y)[(q  )%4] [1])<<(32-r)); ₩
    ((uint16 *)(rk))[4] = ((X)[2] [0]) ^ (((Y)[(q+1)%4] [1])>>(r-16)) ^ (((Y)[(q+1)%4] [0])<<(32-r)); ₩
    ((uint16 *)(rk))[5] = ((X)[2] [1]) ^ (((Y)[(q+2)%4] [0])>>(r-16)) ^ (((Y)[(q+1)%4] [1])<<(32-r)); ₩
    ((uint16 *)(rk))[6] = ((X)[3] [0]) ^ (((Y)[(q+2)%4] [1])>>(r-16)) ^ (((Y)[(q+2)%4] [0])<<(32-r)); ₩
    ((uint16 *)(rk))[7] = ((X)[3] [1]) ^ (((Y)[(q+3)%4] [0])>>(r-16)) ^ (((Y)[(q+2)%4] [1])<<(32-r)); ₩
  } else { ₩
    ((uint16 *)(rk))[0] = ((X)[0] [0]) ^ (((Y)[(q  )%4] [0])>>r) ^ (((Y)[(q+3)%4] [1])<<(16-r)); ₩
    ((uint16 *)(rk))[1] = ((X)[0] [1]) ^ (((Y)[(q  )%4] [1])>>r) ^ (((Y)[(q  )%4] [0])<<(16-r)); ₩
    ((uint16 *)(rk))[2] = ((X)[1] [0]) ^ (((Y)[(q+1)%4] [0])>>r) ^ (((Y)[(q  )%4] [1])<<(16-r)); ₩
    ((uint16 *)(rk))[3] = ((X)[1] [1]) ^ (((Y)[(q+1)%4] [1])>>r) ^ (((Y)[(q+1)%4] [0])<<(16-r)); ₩
    ((uint16 *)(rk))[4] = ((X)[2] [0]) ^ (((Y)[(q+2)%4] [0])>>r) ^ (((Y)[(q+1)%4] [1])<<(16-r)); ₩
    ((uint16 *)(rk))[5] = ((X)[2] [1]) ^ (((Y)[(q+2)%4] [1])>>r) ^ (((Y)[(q+2)%4] [0])<<(16-r)); ₩
    ((uint16 *)(rk))[6] = ((X)[3] [0]) ^ (((Y)[(q+3)%4] [0])>>r) ^ (((Y)[(q+2)%4] [1])<<(16-r)); ₩
    ((uint16 *)(rk))[7] = ((X)[3] [1]) ^ (((Y)[(q+3)%4] [1])>>r) ^ (((Y)[(q+3)%4] [0])<<(16-r)); ₩
  }                                                        ₩
  rk += 16;       }
```

Figure 10. Code for Round Key Generation *after* Modification

4. Evaluation

We compare the proposed 16-bits optimized code with open source codes optimized for 32-bits processors and 8-bits processors. The 32-bits optimized code and 8-bits optimized code are distributed by KISA [10]. The three codes are compiled with the AVR compiler and translated into executable machine codes. The translated machine codes are applied to Atmel ATmega2560 micro-controllers. Table 1 shows the execution time of the three codes. When encrypting a 128-bits block message, each code requires both encryption key setup procedure and encryption procedure. When decrypting a 128-bit encrypted block message, each code requires both decryption key setup procedure and decryption procedure. Compare to the 32-bits optimized code, the proposed 16-bits optimized code enhance the execution speed by about 20%. Also, compared to the 8-bits optimized code, the proposed code enhances the execution speed by 91%.

Table 1. Execution Speed of Three Codes

	32-bits Optimize Code	**8-bits Optimize Code**	**16-bits Optimize Code**
Encryption Key Setup Ex Time	864 µsec.	14,252 µsec.	480 µsec.
Encryption Exec. Time	532 µsec.	828 µsec.	600 µsec.
Decryption Key Setup Ex Time	1,264 µsec.	14,768 µsec.	860 µsec.
Decryption Exec. Time	532 µsec.	816 µsec.	600 µsec.
Total Exec. Time	3,192 µsec.	30,664 µsec.	2,540 µsec.

Table 2 shows the space requirements of the three codes. Compare to the 32-bits optimized code, the proposed 16-bits optimized code reduces the translated machine code size (footprint) by about 28%. The memory usages of two codes are equal. Compare to the 8-bits optimized code, the proposed code requires a larger machine code size and memory usage. However, our 16-bits optimized code is faster than the 8-bits by more than 10 times. Consequently, our 16-bits optimized code is the best choice for systems with fast speedup requirements and the 8-bits optimized code is the best choice for systems with small memory usage requirements.

Table 2. Space Requirements of Three Codes

	32-bits Optimized C	**8-bits Optimized C**	**16-bits Optimize Code**
Machine Code Size	31,094 bytes	6,518 bytes	22,366 bytes
Memory Usage	4,458 bytes	1,386 bytes	4,458 bytes

5. Conclusions

The proposed design optimizes the execution speed of the 128-bits block cipher ARIA for 16-bits processors, whereas the previous studies handled only the 32-bits optimization and the 8-bits optimization of the ARIA algorithm. The proposed design extends 8-bits array variables into 16-bits array variables for faster chained matrix multiplication. Also, the proposed design adopts 16-bits XOR operations and rotated shift operations as many as possible. In evaluation experiments, our 16-bits optimized code yields about 20% faster execution speed and about 28% smaller footprint, compared to the 32-bits optimized code. Compared to the 8-bits optimized code, our code yields about 91% faster execution speed with a larger space requirement.

Acknowledgements

This research was supported by the Dongduk Women’s University Grant, 2015.

References

[1] M. Ebrahim, S. Khan and U. B. Khalid, “Symmetric Algorithm Survey: A Comparative Analysis,” International Journal of Computer Applications **(Jan. 2013)**, vol. 61, no. 20, pp. 12-19.

[2] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann and L. Uhsadel, “A Survey of Lightweight-Cryptography Implementations,” IEEE Design & Test of Computers **(Nov.-Dec. 2007)**, vol. 24, no. 6, pp. 522-533.

[3] S. B. Sasi and N. Sivan, “A Survey on Cryptography Using Optimization Algorithms in WSNs,” Indian Journal of Science and Technology **(Feb. 2015)**, vol. 8, no. 3, pp. 216-221.

[4] William Stalling, Cryptography and Network Security: Principles and Practices, 6th edition, Prentice Hall, **(2013)**.

[5] National Institute of Standards and Technology (NIST), “Advanced Encryption Standards (AES),” Federal Information Processing Standards Publication 197 **(Nov. 2001)**, pp. 1-26.

[6] KS X 1213:2004, “128 bit Block Encryption Algorithm ARIA,” Korean Agency for Technology and Standards **(Dec. 2004)**.

[7] D. Kwon, J. Kim, S. Park, S. H. Sung, Y. Sohn, J. W. Song, Y. Yeom, E. Yoon, S. Lee, J. Lee, S. Chee, D. Han and J. Hong, “New Block Cipher: ARIA,” International Conference on Information Security and Cryptology **(Nov. 2003)**, Lecture Notes in Computer Science 2971, pp. 432-445.

[8] H. I. Kim, C. Park, D. Hong, and C. Seo, “A LEA Implementation Study on UICC-16bit”, Journal of The Korea Institute of Information Security & Cryptology **(Aug. 2004)**, vol. 24, no. 4, pp. 585-591.

[9] Y. W. Law, J. Doumen and P. Hartel, “Survey and Benchmark of Block Cipher for Wireless Sensor Networks,” ACM Transactions on Sensor Networks **(Feb. 2006)**, vol. 2, no. 1, pp. 65-93.

[10] Korea Internet & Security Agency (KISA), Available at http://seed.kisa.or.kr/iwt/ko/bbs/EgovReferenceDetail.do?bbsId=BBSMSTR_000000000002&nttId=39&pageIndex=1&searchCnd=&searchWrd=