SPEEDY on Cortex–M3: Efficient Software Implementation of SPEEDY on ARM Cortex–M3

Hyunjun Kim[1], Kyungbae Jang[1], Gyeongju Song[1], Minjoo Sim[1], Siwoo Eum[1], Hyunji Kim[1], Hyeokdong Kwon[1], Wai-Kong Lee[2], and Hwajeong Seo[1][0000−0003−0069−9061]

[1]IT Department, Hansung University, Seoul (02876), South Korea,
{khj930704, starj1023, thdrudwn98, minjoos9797, shuraatum, khj1594012, korlethean, hwajeong84}@gmail.com
[2]Department of Computer Engineering, Gachon University, Seongnam, Incheon (13120), Korea,
waikonglee@gachon.ac.kr

Abstract. The SPEEDY block cipher suite announced at CHES 2021 shows excellent hardware performance. However, SPEEDY was not designed to be efficient in software implementations. SPEEDY's 6-bit S-box and bit permutation operations generally do not work efficiently in software. We implemented SPEEDY block cipher by applying the implementation technique of bit-slicing. As an implementation technique of bit-slicing, SPEEDY can be operated in software very efficiently and can be applied in microcontroller. By calculating the round key in advance, the performance on ARM Cortex-M3 for SPEEDY-5-192, SPEEDY-6-192, and SPEEDY-7-192 are 65.7, 75.25, and 85.16 clock cycles per byte (i.e. cpb), respectively. It showed better performance than AES-128 constant-time implementation and GIFT constant-time implementation in the same platform. Through this, we conclude that SPEEDY can show good performance on embedded environments.

Keywords: Software Implementation · SPEEDY · ARM Cortex–M3.

1 Introduction

SPEEDY is a very low-latency block cipher designed for hardware implementation in high-performance CPUs. With gate and transistor level considerations, it has been shown that they can run faster in hardware than other block ciphers. The author of SPEEDY [1] provides reference code implemented in C. However, this software implementation does not show the superior performance like the hardware implementation. The performance of SPEEDY's software implementation has not been confirmed, but hardware-oriented block cipher is generally less efficient in software implementation than other software-oriented block ciphers. For this reason, efficient implementation should be considered for SPEEDY block

cipher. In [2], Reis et al. shows that PRESENT can be implemented efficiently in software. PRESENT consists of a 4-bit S-box and a 64-bit permutation, which is far from being implemented in efficient software. However, they used a bit-slicing implementation of the S-box using new permutations and optimized boolean formulas instead of lookup tables, improving the best assembly implementation in Cortex-M3 by 8× than previous works. The implementation is close to competing with the software implementation of AES. In [3], Adomnicai et al. show a very efficient software implementation of GIFT using only a few rotations with a new technique called fix-slicing. It showed faster performance than the best AES [4] constant time at the time when it was reported that PRESENT [2] implementation 1,617 cycles were required in Cortex-M3 microcontrollers. Based on these studies, it is possible for a hardware-friendly cipher to implement in software efficiently. We expected that SPEEDY would be able to achieve sufficient performance in software through the previous technique. NIST's nominations for cryptographic standards include hardware and software evaluations, and ciphers with high performance in hardware and software are considered competitive over other ciphers. If the efficient implementation of SPEEDY's software is possible, it is thought that it will enhance the competitiveness of SPEEDY.

1.1 Contributions

We have achieved excellent performance by implementing SPEEDY in software on a 32-bit ARM processors. SPEEDY's 6-bit S-box and bit permutations seem to make the software implementation inefficient, but we use the bit-slicing implementation technique to resolve this issue. By implementing bit-slicing, all blocks of SPEEDY can be operated in a parallel way. It can also achieve constant time implementation, leading to the prevention of timing attacks. Bit-slicing technique is applied efficiently due to the bit permutation of the simple structure. The barrel shift of the Cortex-M3 maximizes these advantages. In ARM Cortex-M3, SPEEDY-5-192, SPEEDY-6-192, and SPEEDY-7-192 achieved 65.7, 75.25, and 85.16 clock cycles per byte, respectively. This is faster than the constant time implementation of GIFT-128 and AES-128 block ciphers in same architecture.

2 SPEEDY Algorithm

SPEEDY is a very low-latency block cipher. SPEEDY prioritizes speed and high security. It is primarily designed for hardware security solutions built into high-end CPUs that require significantly higher performance in terms of latency and throughput. A 6-bit S-box is used and 192 bits, which is the least common multiple of 6 and 64, is used as the block and key size considering the 64-bit CPU. 192 bits can be expressed in 32 rows of 6 bits each. SPEEDY Family consists of SPEEDY-5-192, SPEEDY-6-192, and SPEEDY-7-192 according to the number of rounds. It is noted that round 6 achieves 128-bit security, 7 round achieves 192-bit security, and round 5 provides a sufficient level of security for many practical applications.The SPEEDY-r-6ℓ an instance of this family with a

block and key size of 6ℓ bits. It can be seen as a $\ell \times 6$ rectangular arrangement. The round function function is as follows.

– SubBox (SB): SPEEDY's 6-bit S-box is based on NAND gates and is designed to be very fast in CMOS hardware while at the same time providing excellent cryptographic properties. S-boxes are applied to each row of states. The Disjunctive Normal Form (DNF) of S-box is as follows. In our implementation, the operation of DNF is followed, and S-box is performed by AND operation and OR operation.

$$
\begin{aligned}
y_0 &= (x_3 \wedge \neg x_5) \vee (x_3 \wedge x_4 \wedge x_2) \vee (\neg x_3 \wedge x_1 \wedge x_0) \vee (x_5 \wedge x_4 \wedge x_1) \\
y_1 &= (x_5 \wedge x_3 \wedge \neg x_2) \vee (\neg x_5 \wedge x_3 \wedge \neg x_4) \vee (x_5 \wedge x_2 \wedge x_0) \vee (\neg x_3 \wedge \neg x_0 \wedge x_1) \\
y_2 &= (\neg x_3 \wedge x_0 \wedge x_4) \vee (x_3 \wedge x_0 \wedge x_1) \vee (\neg x_3 \wedge \neg x_4 \wedge x_2) \vee (\neg x_0 \wedge \neg x_2 \wedge \neg x_5) \\
y_3 &= (\neg x_0 \wedge x_2 \neg x_3) \vee (x_0 \wedge x_2 \wedge x_4) \vee (x_0 \wedge \neg x_2 \wedge x_5) \vee (\neg x_4 \wedge \neg x_2 \wedge x_1) \\
y_4 &= (x_0 \wedge \neg x_3) \vee (x_0 \wedge \neg x_4 \wedge \neg x_2) \vee (\neg x_0 \wedge x_4 \wedge x_5) \vee (\neg x_4 \wedge \neg x_2 \wedge x_1) \\
y_5 &= (x_2 \wedge x_5) \vee (\neg x_2 \wedge \neg x_1 \wedge x_4) \vee (x_2 \wedge x_1 \wedge x_0) \vee (\neg x_1 \wedge x_0 \wedge x_3)
\end{aligned}
$$

– ShiftColumns (SC) : The j-th column of the state is rotated upside by j bits. In hardware implementation, ShiftColumnsdms is free with simple wiring, but additional operation is required in software.

$$y_{[i,j]} = x_{[i+j,j]}$$

– MixColumns (MC) : A cyclic binary matrix is multiplied to each column of the state. In hardware, it can be implemented only with XOR gate, but similar to ShiftColumns, the additional operation is required in the software implementation. $\alpha = (\alpha 1, ..., \alpha 6)$ is the parameterized value for each version

$$y_{[i,j]} = x_{i,j} \oplus x_{[i+\alpha_1,j]} \oplus x_{[i+\alpha_2,j]} \oplus x_{[i+\alpha_3,j]} \oplus x_{[i+\alpha_4,j]} \oplus x_{[i+\alpha_5,j]} \oplus x_{[i+\alpha_6,j]}$$

– AddRoundKey (AK) : The 6ℓ-bit round key k_r is XORed to the whole of the state.

$$y_{[i,j]} = x_{[i,j]} \oplus k_{r[i,j]}$$

– AddRoundConstant (AC) : The 6ℓ-bit constant c_r is XORed to the whole of the state. round constants are chosen as the binary digits of the number $\pi - 3 = 0.1415....$

$$y_{[i,j]} = x_{[i,j]} \oplus c_{r[i,j]}$$

Encryption operates in the order of $\mathrm{A}_k \rightarrow$ SB $\rightarrow$ SC $\rightarrow$ SB $\rightarrow$ SC $\rightarrow$ MC $\rightarrow \mathrm{A}_c$ in one round, and operates in the order of $\mathrm{A}_k \rightarrow$ SB $\rightarrow$ SC $\rightarrow$ SB $\rightarrow \mathrm{A}_k$ in the last round. In the decoding, an inverse operation is performed in the reverse order. In the first round, it operates in the order of $\mathrm{A}_k \rightarrow$ InverseSB $\rightarrow$ InverseSC $\rightarrow$ InverseSB $\rightarrow \mathrm{A}_k$, and from the next round, it repeats in the order $\mathrm{A}_c \rightarrow$ InverseMC $\rightarrow$ InverseSC $\rightarrow$ InverseSB $\rightarrow$ InverseSC $\rightarrow$ InverseSB $\rightarrow \mathrm{A}_k$.

3 Proposed Method

This chapter describes the proposed SPEEDY implementation method. First, looking at the round function of SPEEDY from a software implementation point of view, in the case of ShiftColumns, it is computed for free in hardware, but the bit permutation in a column unit is inefficient in software. A block of 6-bit does not fit the 8-bit, 32-bit, and 64-bit blocks used in typical processor architectures. Implementing SPEEDY in software is inefficient due to 6-bit S-boxes and bit permutations. As a solution to this, we used the bit-slicing technique. Due to the effect of this alignment, the round function can be implemented efficiently in software.

We show that SPEEDY can work well in an embedded environment by implementing it on ARM Cortex-M3 microcontrollers. In our implementation, the 32 blocks of 6 bits are converted to a bit-slicing representation and stored in 6 32-bit registers. As a result, the round function is able to execute 32 blocks in parallel. Specifically, ShiftColumns and MixColumns work efficiently with the Cortex-M3 barrel shifter. Since we modified the logical operation process of S-box, S-BOX operation operates using few instructions and implemented it efficiently. Considering the case where the key is used repeatedly, the RoundConstant value is calculated in advance with the round key, and AddRoundConstant is omitted.

3.1 SPEEDY on ARM Cortex–M3

The Cortex-M3 is ARM's family of 32-bit processors for use in embedded microcontrollers. It is designed to be inexpensive and energy-efficient, so it has very effective characteristics for implementing IoT services. Arithmetic and logic operations take one clock cycle. However, branches, loads, and stores can take more cycles. A distinctive feature is that it supports a barrel shifter. By using the barrel shifter, rotation or shift can be performed at no additional cost in arithmetic or logical operations. This microprocessor haves 16 32-bit registers, 3 of which are for program counters, stack pointers and link registers, for a total of 14 registers available to developers (R_0-R_{12}, R_14). The first thing to consider to increase computational performance is to minimize access to memory.

Therefore, in our implementation, the address value of the periodically used round key is stored in one register and used repeatedly, and the intermediate value of the operation is stored by fixing 6 registers. In order to operate without access to memory except for the AddRoundKey function, 7 temporary storage spaces were required for SubBox operation, and 6 temporary storage spaces were needed for ShiftColumns and MixColumns respectively. For the efficient operation, all 14 registers are used and the value stored in the ciphertext address is called once at the end. It is stored on the stack at the start of the operation and loaded at the end.

3.2 Using Bit-slicing in SPEEDY

Bit-slicing was the first technique used by Biham [5] instead of lookup tables to speed up the software implementation of DES. The basic method of bit-slicing

Table 1. Bit-slicing representation from using 6 32-bit registers R_0, ... , R_5 to process 8 blocks $b^0, , b^7$ in parallel where b^i_j refers to the j-th bit of the i-th block.

	$Block0$	$Block1$	$Block2$	$Block3$	$\cdots$ $\cdots$	$Block28$	$Block29$	$Block30$	$Block31$
R_0	b_0^0	b_0^1	b_0^2	b_0^3	$\cdots$ $\cdots$	b_0^{28}	b_0^{29}	b_0^{30}	b_0^{31}
R_1	b_1^0	b_1^1	b_1^2	b_1^3	$\cdots$ $\cdots$	b_1^{28}	b_1^{29}	b_1^{30}	b_1^{31}
R_2	b_2^0	b_2^1	b_2^2	b_2^3	$\cdots$ $\cdots$	b_2^{28}	b_2^{29}	b_2^{30}	b_2^{31}
R_3	b_3^0	b_3^1	b_3^2	b_3^3	$\cdots$ $\cdots$	b_3^{28}	b_3^{29}	b_3^{30}	b_3^{31}
R_4	b_4^0	b_4^1	b_4^2	b_4^3	$\cdots$ $\cdots$	b_4^{28}	b_4^{29}	b_4^{30}	b_4^{31}
R_5	b_5^0	b_5^1	b_5^2	b_5^3	$\cdots$ $\cdots$	b_5^{28}	b_5^{29}	b_5^{30}	b_5^{31}

is to express n-bit data by 1 bit in n registers. In this way, multiple blocks can be processed in parallel with bitwise operation instructions. In the case of AES, 128-bit plaintext is expressed in 8 registers and operates. Larger registers allow more blocks to be operated. They work more efficiently on processors using large registers. In the case of SPEEDY, since 192 bits are divided into 32 blocks of 6 bits each, it can be expressed with 6 32-bit registers. The 192-bit plaintext is relocated to bitslcicing representation as shown in Table a and stored in 6 32-bit registers. With this expression method, the blocks of SPEEDY can be operated in parallel in all functions and operate efficiently. In S-Box operation is performed by combining bitwise operators, and all operations are processed in parallel with 32 blocks. In particular, SC and MC operations can be operated very simply and quickly with the barrel shift operation of Cortex-M3.

In general, when rearranging the input into a bit-slicing representation, this can be done using the SWAPMOVE technique [6].

$$\begin{gathered}\text{SWAPMOVE(A, B, M, n)}: \\ \text{T} = (\text{B} \oplus (A \ll n)) \wedge M \\ \text{B} = \text{B} \oplus T \\ \text{A} = \text{A} \oplus (T \gg n)\end{gathered}$$

However, SPEEDY block cipher could not make a bit-slicing representation using only SWAPMOVE. In a 32-bit processor, 192 bits of plaintext are stored in six segments. At this time, it is inefficient to rearrange the 6-bit blocks in a bit-slicing representation because they are stored in different spaces. Considering this, we implemented it in three steps to make the most of SWAPMOVE technology. There are 5 blocks of 6 bits that can be completely stored in one register. Therefore, SWAPMOVE technology is applied to the blocks of 0-th to 29-th indexes, and the rest are implemented by moving 1 bit at a time. First, as shown in Figure a, step 1 is arranged in 6-bit blocks, and in step 2, indexes 30 and 31 are rearranged by bit-slicing expression and rearranged by moving one bit at a time. Finally, in step 3, SWAPMOVE rearranges the blocks from the 0-th to the 29-th index.

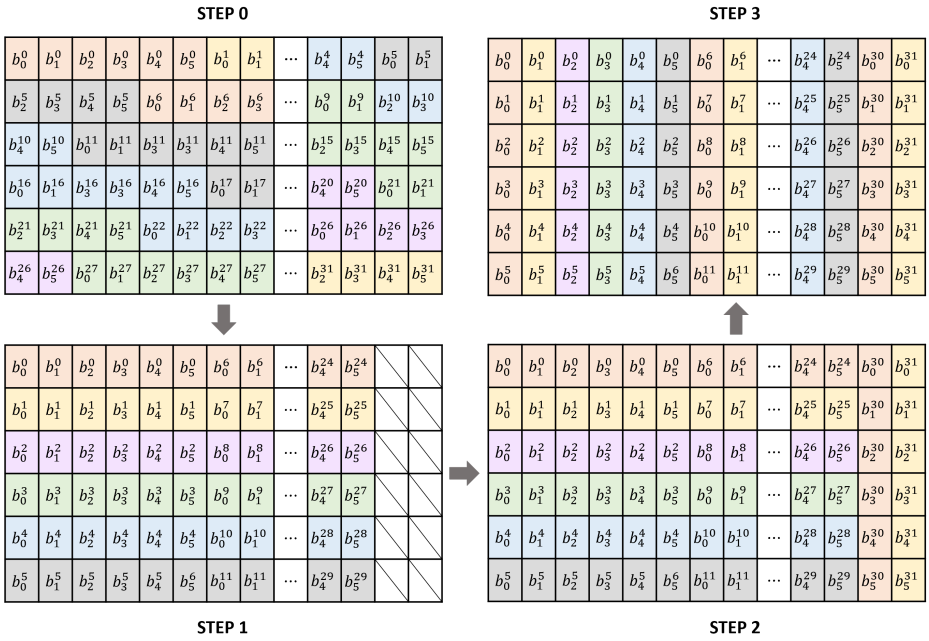

Fig. 1. Reorders plain text consisting of 32 blocks of 6 bits in 6 32-bit registers into a bit-slicing representation. A block of 6 bits is expressed as b_j^i, where i is the index of the block, and j is the position of the bit.

3.3 SubBox

SubBox layer operation is performed by combining logical operators instead of lookup table method. In the implementation of the expression by bit-slicing, 32 blocks of 6 bits can be operated in parallel by a combination of logical operators. It can be operated efficiently. Additionally, we use the rule of logical operators to reduce the number of logical operations. In this way, 8 instruction are reduced in the SubBox layer. The S-Box operation can be transformed into the following formula.

$$
\begin{aligned}
y_0 &= x_3 \wedge (\neg x_5 \vee (x_4 \wedge x_2) \vee (x_1 \wedge ((\neg x_3 \wedge x_0) \vee (x_5 \wedge x_4))) \\
y_1 &= x_5 \wedge ((x_3 \wedge \neg x_2) \vee (x_2 \wedge x_0) \vee (\neg x_5 \wedge x_3 \wedge \neg x_4) \vee (\neg x_3 \wedge \neg x_0 \wedge x_1) \\
y_2 &= x_0 \wedge ((\neg x_3 \wedge x_4) \vee (x_3 \wedge x_1) \vee (\neg x_3 \wedge \neg x_4 \wedge x_2) \vee (\neg x_0 \wedge \neg x_2 \wedge x_5) \\
y_3 &= x_2 \wedge ((\neg x_0 \wedge \neg x_3) \vee (x_0 \wedge x_4) \vee (x_0 \wedge \neg x_2 \wedge x_5) \vee (\neg x_0 \wedge x_3 \wedge x_1) \\
y_4 &= (x_0 \wedge \neg x_3) \vee (\neg x_0 \wedge x_4 \wedge x_5) \vee ((\neg x_4 \wedge \neg x_2) \wedge (x_0 \vee x_1) \\
y_5 &= x_2 \wedge (x_5 \vee (x_1 \wedge x_0)) \vee (\neg x_1 \wedge ((\neg x_2 \wedge x_4) \vee (x_0 \wedge x_3)
\end{aligned}
$$

There is no Cortex-M3 assembly instruction corresponding to the $\neg$ operation or the operation $a \wedge \neg b$ used here. However, the operation of $a \vee \neg b$ can be performed

with the ORN instruction as used in Algorithm a. For efficient implementation, the not operation is performed using the ORN instruction. For example, in the case of $(x_3 \wedge x_4 \wedge x_2) \vee (\neg x_3 \wedge x_1 \wedge x_0)$, use the rule of logical operators to convert it to $(x_3 \wedge x_4 \wedge x_2) \vee \neg((x_3 \vee \neg x_1) \vee_0)$.

Algorithm 1 bit-slicing implementations of S-box in ARMv6 assembly.

Input: input registers
`x0-x5` (`r4-r9`),
temporal register `t` (`r14`)
Output: output registers
`y0-y5` (`r1-r3, r10-r12`)

```
 1: AND y3, x2, x4
 2: ORN y3, y3, x5
 3: AND y3, y3, x3
 4: AND y4, x5, x4
 5: ORN y5, x3, x0
 6: ORN y4, y4, y5
 7: AND y4, x1, y4
 8: ORR y0, y4, y3

 9: AND y3, x0, x2
10: ORN y4, x2, x3
11: ORN y3, y3, y4
12: AND y3, y3, x5
13: ORR y4, x5, x4
14: ORN y4, y4, x3
15: ORN y3, y3, y4
16: ORR y4, x0, x3
17: ORN y4, y4, x1
18: ORN y1, y3, y4

19: AND y3, x1, x3
20: ORN y4, x3, x4
21: ORN y3, y3, y4
22: AND y3, x0, y3
23: ORR y4, x3, x4
24: ORN y4, y4, x2
25: ORN y3, y3, y4
26: ORR y4, x0, x2
27: ORR y4, y4, x5
28: ORN y2, y3, y4

29: AND y3, x0, x4
30: ORR y4, x0, x3
31: ORN y3, y3, y4
32: AND y3, y3, x2
33: AND y4, x0, x5
34: ORN y4, x2, y4
35: ORN y3, y3, y4
36: AND y4, x1, x3
37: ORN y4, x0, y4
38: ORN y3, y3, y4

39: MOV t, #s0
40: ORR y4, x4, x2
41: ORR y5, x0, x1
42: ORN y4, y4, y5
43: ORN y4, t, y4
44: ORN y5, x3, x0
45: ORN y4, y4, y5
46: AND y5, x4, x5
47: ORN y5, x0, y5
48: ORN y4, y4, y5

49: AND t, x0, x3
50: ORN y5, x2, x4
51: ORN t, t, y5
52: ORN t, x1, t
53: AND y5, x1, x0
54: ORR y5, y5, x5
55: AND y5, y5, x2
56: ORN y5, y5, t
```

3.4 ShiftColumns

In the bit-slicing implementation, ShiftColumns can be implemented efficiently. In ShiftColumns, bits of the block are shifted in the column direction. In the bit-

Algorithm 2 bit-slicing implementations of ShiftColumns in ARMv6 assembly.

Input: input registers
`x0-x5` (`r1-r3, r10-r12`)
Output: output register
`y0-y5` (`r4-r9`)

```
1: MOV y0, x0
2: MOV y1, x1, ROR #31
3: MOV y2, x2, ROR #30
4: MOV y3, x3, ROR #29
5: MOV y4, x4, ROR #28
6: MOV y5, x5, ROR #27
```

slicing representation, bits are converted into transposition in the row direction because rows and columns are switched. Therefore, it can be implemented with rotation operation. 32 blocks are operated in parallel and can be implemented with 6 mov instructions. At this time, the value moved after rotation is stored in another register. And the value stored in the existing register is used again in the MC operation for operation.

3.5 MixColumns

In a bit-slicing implementation similar to ShiftColumns, the operation of MixColumns can be implemented efficiently. Since rows and columns are switched, rotate each row as much as a_i and perform XOR as shown below.

$$y[i] = x[i] \oplus (x[i] <<< a_0) \oplus (x[i] <<< a_1) \oplus (x[i] <<< a_2) \\ \oplus (x[i] <<< a_3) \oplus (x[i] <<< a_4) \oplus (x[i] <<< a_5)$$

For this operation, the value of $y[i]$ must be stored. As in Algorithm 3, this value reuses the value stored in the existing register in the previous SC process. Since the value stored in the existing register is the value before the SC operation, the additional rotation is required as much as the SC operation. This operation can be implemented with only the XOR instruction, since the rotation operation can be operated with a barrel-shifter. At this time, as a result of the SC operation, 32 blocks are operated in parallel with 36 EOR instruction.

3.6 AddRoundKey and AddRoundConstant

In the case of AC operation, the process of XORing each bit with a constant value is performed in the same way as in AR operation. Therefore, it is implemented to XOR the constant value and the round key value in advance. In consideration of the bit-slicing expression, the round key must also be packed in the same form, and as in Algorithm a, load and xor are each executed 6 times. When encryption starts after the key schedule operates first, the encryption process is calculated by omitting the AC process .

Algorithm 3 bit-slicing implementations MixColumns in ARMv6 assembly.

Input: input registers
`x0-x5` (`r1-r3, r10-r12`)
`y0-y5` (`r4-r9`)
Output: output registers
`y0-y5` (`r4-r9`)

```
 1: EOR y0, y0, x0, ROR #31
 2: EOR y0, y0, x0, ROR #27
 3: EOR y0, y0, x0, ROR #23
 4: EOR y0, y0, x0, ROR #17
 5: EOR y0, y0, x0, ROR #11
 6: EOR y0, y0, x0, ROR #6

 7: EOR y1, y1, x1, ROR #30
 8: EOR y1, y1, x1, ROR #26
 9: EOR y1, y1, x1, ROR #22
10: EOR y1, y1, x1, ROR #16
11: EOR y1, y1, x1, ROR #10
12: EOR y1, y1, x1, ROR #5

13: EOR y2, y2, x2, ROR #29
14: EOR y2, y2, x2, ROR #25
15: EOR y2, y2, x2, ROR #21
16: EOR y2, y2, x2, ROR #15
17: EOR y2, y2, x2, ROR #9
18: EOR y2, y2, x2, ROR #4

19: EOR y3, y3, x3, ROR #28
20: EOR y3, y3, x3, ROR #24
21: EOR y3, y3, x3, ROR #20
22: EOR y3, y3, x3, ROR #14
23: EOR y3, y3, x3, ROR #8
24: EOR y3, y3, x3, ROR #3

25: EOR y4, y4, x4, ROR #27
26: EOR y4, y4, x4, ROR #23
27: EOR y4, y4, x4, ROR #19
28: EOR y4, y4, x4, ROR #13
29: EOR y4, y4, x4, ROR #7
30: EOR y4, y4, x4, ROR #2

31: EOR y5, y5, x5, ROR #26
32: EOR y5, y5, x5, ROR #22
33: EOR y5, y5, x5, ROR #18
34: EOR y5, y5, x5, ROR #12
35: EOR y5, y5, x5, ROR #6
36: EOR y5, y5, x5, ROR #1
```

4 Results

In this chapter, we compare the results for our implementation. The software was developed with Arduino IDE on the ArduinoDUE (AT91SAM3X8E) development board equipped with an ARM Cortex-M3 processor. The operating clock is 84 MHz, and it has 512 KB of flash memory and 96 KB of RAM. Performance comparison measured the average cycle when encrypting. Key scheduling is not taken into account as it is assumed that round keys are pre-computed and stored in RAM. We implemented SPEEDY-5-192, SPEEDY-6-192, and SPEEDY-7-192, and for comparison, AES-128 and GIFT-128 implemented in constant time in the same environment were compared together. In general, 128-bit blocks are encrypted, but since SPEEDY encrypts 192-bit blocks, the performance difference was compared based on cycle per byte (cpb) to compare other encryptions. And the key schedule was calculated in advance, and the average was measured when it was operated in ECB mode. As shown in Table 2, compared to the reference C implementation of SPEEDY-7-192, the speed difference was about 180 times. Although the optimization level of the reference C implementation is not performed in assembly level, it showed a noticeable high performance improvement. In addition, when comparing our implemented SPEEDY-6-192 with the

same security level AES-128 and GIFT-128, the result of 75.2 cpb is 1.6× faster than 120.4 cpb of AES-128 and 1.3× faster than 104.1 cpb of GIFT-128. Considering that SPEEDY is designed to be hardware-friendly, this is a remarkable result.

Table 2. Comparison of SPEEDY implementation results and various constant-time implementation results on ARM Cortex-M3. The performance is evaluated in clock cycles per byte (cpb).

Implementation	Speed (cpb)	Block size
AES-128 encryption	120.4	128
GIFT-128 encryption	104.1	128
SPEEDY-7-192 encryption (reference)	15,407	192
SPEEDY-5-192 encryption (ours)	65.7	192
SPEEDY-6-192 encryption (ours)	75.2	192
SPEEDY-7-192 encryption (ours)	85.1	192

5 Conclusion

We implemented SPEEDY by applying the implementation technique of bitslicing. For the case where the round key is calculated in advance, in ARM Cortex-M3, SPEEDY-5-192 achieves 65.7 cpb, SPEEDY-6-192 achieves 75.25 cpb, and SPEEDY-7-192 achieves 85.16 cpb, respectively. In the same environment, it showed better performance than 120.4 cpb of constant time implementation GIFT-128 and 104.1 cpb of constant time implementation AES-128. Through this, we showed that SPEEDY can be run very efficiently in software and can be applied in microcontrollers. The proposed technique is likely to be applicable to other processors, and in the future, we plan to implement other platforms (e.g. Cortex-M4). The proposed implementation is working in constant timing, which has an advantage against timing attacks. In the future work, we intend to apply an efficient masking technique for additional side-channel security.

6 Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00540, Development of Fast Design and Implementation of Cryptographic Algorithms based on GPU/ASIC, 50%) and this work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00264, Research on Blockchain Security Technology for IoT Services, 50%).

References

1. G. Leander, T. Moos, A. Moradi, and S. Rasoolzadeh, "The SPEEDY family of block ciphers: Engineering an ultra low-latency cipher from gate level for secure processor architectures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, p. 510–545, Aug. 2021.
2. T. Reis, D. Aranha, and J. López, "PRESENT runs fast," pp. 644–664, 08 2017.
3. A. Adomnicai, Z. Najm, and T. Peyrin, "Fixslicing: A new GIFT representation: Fast constant-time implementations of GIFT and GIFT-COFB on ARM Cortex-M," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, p. 402–427, Jun. 2020.
4. P. Schwabe and K. Stoffelen, "All the AES you need on Cortex-M3 and M4," pp. 180–194, 10 2017.
5. E. Biham, "A fast new DES implementation in software," in *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, vol. 1267 of *Lecture Notes in Computer Science*, pp. 260–272, Springer, 1997.
6. L. May, L. Penna, and A. Clark, "An implementation of bitsliced DES on the pentium MMX," pp. 112–122, 01 2000.