



# Efficient Implementation of SHA-3 Hash Function on 8-Bit AVR-Based Sensor Nodes

YoungBeom Kim, Hojin Choi, and Seog Chung Seo<sup>(✉)</sup> 

Department of Information Security, Cryptology, and Mathematics,  
Kookmin University, Seoul, South Korea  
{[darania](mailto:darania@kookmin.ac.kr), [ondoli0312](mailto:ondoli0312@kookmin.ac.kr), [scseo](mailto:scseo@kookmin.ac.kr)}@kookmin.ac.kr

**Abstract.** The Keccak algorithm was selected by NIST as the standard SHA-3 hash algorithm for replacing currently used SHA-2 algorithm in 2015. Despite SHA-3's improved security compared to SHA-2, its low performance in software implementation limits its wide use. In this paper, we propose an optimized SHA-3 implementation on 8-bit AVR microcontrollers (MCU) which are dominantly used for sensor devices in WSNs. Until now, there are only a few researches on optimization of SHA-3 in spite of its security importance. Furthermore, it is very challenging to optimize hash function, especially, SHA-3, on 8-bit AVR MCUs. This is because the internal state of SHA-3 is 1,600-bit which is much larger than internal state of symmetric algorithms (typically, 128-bit) like AES, ARIA, and so on. In other words, it is difficult to accommodate the whole of SHA-3's internal state on the registers of AVR MCUs, which incurs heavy memory accesses during computation. Thus, we analyzed the structure of SHA-3 algorithm and found that each lane of the internal state can be executed independently for each process in SHA-3. By using this fact, we propose an optimization method which can reduce efficiently the times of memory accesses to the internal state. With this proposed method minimizing the memory accesses, our implementation of SHA3-256 achieves around 25.0% of performance improvement when hashing 500 bytes message compared with the previous best work on 8-bit AVR MCU. To the best of our knowledge, our software is the fastest SHA-3 implementation on AVR platforms until now. In addition, the proposed optimization method can be easily extended to other embedded MCUs such as 16-bit MSP430, 32-bit RISC-V and ARM-based MCUs.

**Keywords:** SHA-3 · Keccak algorithm · 8-bit AVR MCUs · Embedded · Microcontroller · WSN

## 1 Introduction

As the recent Internet of Things (IoT) era has arrived, many applications are being implemented with Wireless Sensor Networks (WSNs) [1]. However, due

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1058494).

to the characteristics of wireless communication in WSNs, the transmitted data can be easily modified or forged. Hash function is cryptographic algorithm that provides data integrity and be used to prove whether the data have been modified or not. Furthermore, it is also used in other various cryptographic applications, e.g., PBKDF2, HMAC, DRBG, and digital signature algorithms such as DSA, ECDSA, and RSA-PSS. Hash functions (e.g., SHA-1 or SHA2) are the most significant cryptographic algorithms from an integrity perspective. However, several vulnerabilities in SHA-1 have been identified; thus, the US National Institute of Standards and Technology (NIST) currently does not recommend the use of SHA-1 [2–6]. Public attacks on the SHA-2 family have continued since 2008. Note that SHA-1 and SHA-2 differ significant; however, they share a similar algorithm (i.e., SHA) [7, 8]. Therefore, SHA-1 and SHA-2 are vulnerable to attacks with some identical digests. In some scenarios, the security of SHA-2 is superior to that of SHA-1, except SHA-2 uses larger inputs and outputs [7, 8]. In addition, several theoretical preimage attacks on SHA-2 have been found [9–11]. Prior to the discovery of preimage attacks on SHA-2, NIST held the SHA-3 competition, concerning that SHA-2 had a similar structure to SHA-1. The Keccak algorithm was the winner and selected to be the next-generation hash function standard, i.e., SHA-3 [12].

Through numerous verification, the security of SHA-3(Keccak) has been to be secure against existing attacks threatening the security of SHA-1 and SHA-2. However, in spite of higher security of SHA-3, it is currently used less in the field. One of the biggest reasons is SHA-3 has low performance in software [13–20]. In the software implementation, SHA-3 (256-bit) is three times slower than SHA-256 in the 8-bit AVR platform [12, 15–17] and SHA-3 (512-bit) is two times slower than SHA-512 in a CPU environment [2, 18–20]. Therefore, software optimization studies on SHA-3 applicable to various platforms are essential in the future.

In this paper, we propose optimization methods for SHA-3 software implementation. Also, we demonstrate the efficiency of the proposed optimization method, for typical low-end 8-bit AVR microcontrollers which are mainly used for sensor devices in WSNs. The proposed SHA-3 optimization method exploits the fact that each of the 25 lanes of the internal state can be operated independently. Unlike the general implementation order of the existing implementation methods, we combine  $\theta$  process and  $\rho$  process into a single process to reduce memory accesses to the internal state in an efficient manner. By using this fact,  $\pi$  process can be executed implicitly. In addition, the proposed optimization method does not require additional operations or lookup tables. This advantage is especially efficient for limited embedded devices which used in WSNs.

Using the propose method, we present a carefully-optimized assembler implementation of SHA-3 for 8-bit AVR microcontrollers, specifically the ATmega128. When hashing a 500 bytes message, we obtained an execution time of 1073 Cycles Per Byte (*CPB*) using the Assembly implementation. This is 79.1% and 25.0% faster than the latest implementations proposed by Otte et al. [15] and Balasch et al. [16], respectively. In addition, our optimized implementation of SHA-3 is faster than the implementation proposed by Balasch et al., which is currently

the fastest SHA-3 software implementation on the same platform. Furthermore, the proposed method can be applied to various platforms, e.g., 16-bit MSP430, 32-bit ARM-based MCUs, 32-bit RISC-V, CPUs, GPUs [16], and so on.

The remainder of this paper is organized as follows. In Sect. 2, we briefly review SHA-3 and provides an overview of the 8-bit AVR microcontroller. Section 3 analysis a related work on the 8-bit AVR microcontroller. Section 4 propose a new optimization method for SHA-3. Section 5 compares the performance. Finally, Sect. 6 concludes the paper.

## 2 Background

### 2.1 Overview of SHA-3

In 1993, NIST proposed Secure Hash Algorithm 0 (SHA-0) hash function. Later, SHA-1 and SHA-2 were proposed and standardized. However, Stevens et al. proposed the collision pair of SHA-1 [2], and, as the security of SHA-2 hash algorithm has improved [8, 21, 22], NIST selected the Keccak algorithm by Bertoni et al. as SHA-3 in order to substitute SHA-2 in a competition on August 5, 2015 [12]. The Keccak algorithm is based on a sponge construction structure, which differs from the structure of SHA-2; thus, the Keccak algorithm is resistant against attacks that are applicable to SHA-2 [12].

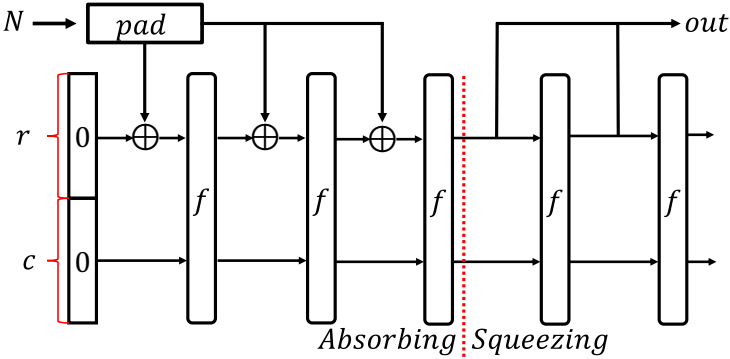


Fig. 1. Overview of sponge structure

**Sponge Structure.** Figure 1 shows the sponge structure of SHA-3. The operational process of SHA-3 assumes a sponge structure comprising absorbing and squeezing processes. The absorbing process compresses the message using the  $b$ -bit permutation  $f$ -function and a padding function to  $pad$  the message. Then, using the exclusive-OR (XOR) operation, the padded message is calculated with the output of  $f$ -function. The digest is calculated in the squeezing process. In SHA-3,  $b$ -bit permutation means the size of the state and is fixed with  $b \in \{25,$

50, 100, 200, 400, 800, 1600}. Here,  $b$  comprises bitrate ( $r$ ) and capacity ( $c$ ), and satisfies  $b = r + c$ . In the squeezing process, if the required digest length is greater than  $r$ , then  $f$ -function is called to update the internal state. In this paper, the values of the SHA-3 parameters are  $b = 1600$ ,  $r = 1088$ , and  $c = 512$ , and the digest length is 256 bits. These parameters are generally used for safety [12].

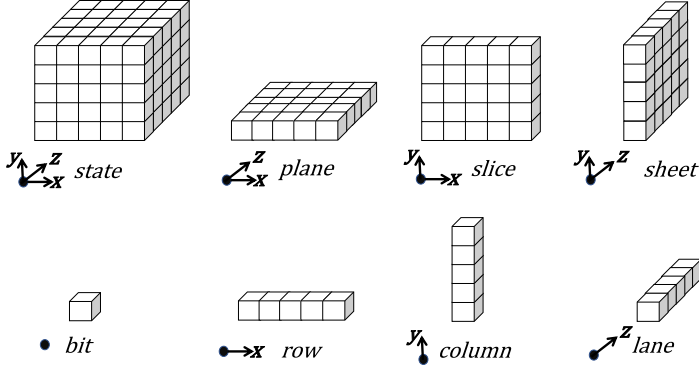


Fig. 2. State of SHA-3[12]

Table 1. the values of  $w$  and  $l$  for each  $b$

| $b$ (bit) | 25 | 50 | 100 | 200 | 400 | 800 | 1600 |
|-----------|----|----|-----|-----|-----|-----|------|
| $w$ (bit) | 1  | 2  | 4   | 8   | 16  | 32  | 64   |
| $l$       | 0  | 1  | 2   | 3   | 4   | 5   | 6    |

**State of SHA-3.**  $f$ -function is the primary process of SHA-3.  $f$ -function takes the state as input data. Figure 2 shows the structure of the state in SHA-3.

The *state* of SHA-3 is a three-dimensional  $x \times y \times z$  matrix, where row  $x$  and column  $y$  are both fixed to five. The *state* comprises 25 lanes and the length of each lane depends on parameters  $b$ ,  $w$ , and  $l$  in SHA-3. According to  $b$ , the *state* is composed of  $5 \times 5 \times w$ . Table 1 shows the  $w$  and  $l$  values for each value of  $b$ . As shown in Table 1,  $l$  is equal to  $\log_2(b/25)$ , and  $w$  is equal to  $2^l$ . In SHA-3,  $f$ -function repeats the same process by the number of rounds (denoted  $n_r$ ). Using parameter  $l$ ,  $n_r$  is represented by  $12 + 2 \times l$ .

**$f$ -function.**  $f$ -function is an internal function of the sponge structure.  $f$ -function is a  $b$ -bit permutation. In SHA-3,  $f$ -function comprise five processes:  $\theta$ ,  $\pi$ ,  $\rho$ ,  $\chi$  and  $\iota$ . These five processes are used to update state, and  $f$ -function repeats the processes in times of  $n_r$  rounds.

The effect of  $\theta$  process is to XOR each bit in the *state* with parties of two columns in the array. In the  $\theta$  process, if this process operates for the bit  $(x_0, y_0, z_0)$ , the  $x$ -coordinates of the required columns are  $(x_0 + 1) \bmod 5$ ,  $(x_0 - 1) \bmod 5$ . In addition, in this process, the  $z$ -coordinate of the column with  $x$ -coordinate  $(x_0 - 1) \bmod 5$  is  $(z_0 - 1) \bmod w$ . Algorithm 1 shows the pseudocode of the  $\theta$  process. Line 2 in Algorithm 1 is referred to as the initial  $\theta$ , which generates  $D[x, z]$ .

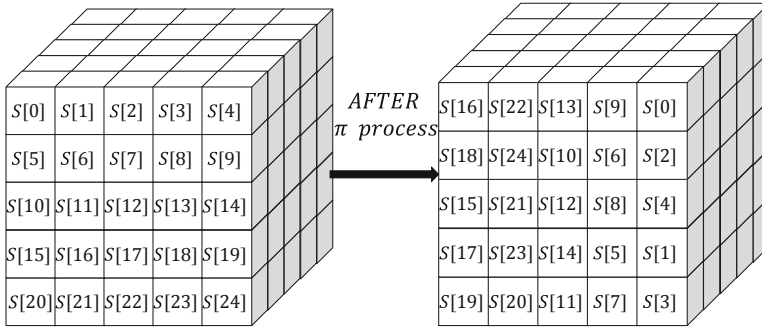
---

**Algorithm 1.**  $\theta$  Process

---

**Require:** *state A*
**Ensure:** *state A'*

- 1: For all pairs  $(x, z)$  such that  $0 \leq x \leq 4$  and  $0 \leq z \leq w-1$   
 $C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z]$ ;
  - 2: For all pairs  $(x, z)$  such that  $0 \leq x \leq 4$  and  $0 \leq z \leq w-1$   
*//This step is initial  $\theta$*   
 $D[x, z] = C[(x - 1) \bmod 5, z] \oplus C[(x + 1) \bmod 5, (z - 1) \bmod w]$ ;
  - 3: For all triples  $(x, y, z)$  such that  $0 \leq x, y \leq 4$  and  $0 \leq z \leq w-1$   
 $A'[x, y, z] = A[x, y, z] \oplus D[x, z]$ ;
  - 4: **return**  $A'$
- 



**Fig. 3.** Overview of  $\pi$  process

The effect of  $\pi$  process is to rearrange the positions of the lanes, as shown in Fig. 3. Here,  $S[i]$  ( $i \in [0, 24]$ ) is each lane of *state*. Note that  $S[12]$  is a lane of  $(x = 0, y = 0)$  in *state* [12].

The effect of  $\rho$  process is to right-rotate the bits of each lane by length (referred to as the offset), which depends on the fixed  $x$ - and  $y$ -coordinates of the lane [12]. Equivalently, for each bit in the lane, the  $z$ -coordinate is modified by adding the offset, modulo the lane size. The offsets for each lane that require the operation are listed in Table 2.

**Table 2.** Offsets of  $\rho$  Process

|       | x = 3 | x = 4 | x = 0 | x = 1 | x = 2 |
|-------|-------|-------|-------|-------|-------|
| y = 2 | 153   | 231   | 3     | 10    | 171   |
| y = 1 | 55    | 276   | 36    | 300   | 6     |
| y = 0 | 28    | 91    | 0     | 1     | 190   |
| y = 4 | 120   | 78    | 210   | 66    | 253   |
| y = 3 | 21    | 136   | 105   | 45    | 15    |

The effect of  $\chi$  process is to XOR each bit with a nonlinear function of two other bits in its row [12]. Note that the difference between  $\chi$  process and the other processes (i.e.,  $\theta$ ,  $\pi$ , and  $\rho$  processes) is that  $\chi$  process should be operated in row form and implemented accordingly.

$\iota$  process execute an XOR operation, for the lane of  $(x = 0, y = 0)$  of the *state* and constants RC [12]. Since  $\iota$  process operates only one lane, in the most implementations,  $\chi$  process and  $\iota$  process are combined into a single process.

## 2.2 Overview of 8-Bit AVR MCUs

The 8-bit AVR microcontroller is an embedded device made into a single integrated circuit by adding memory and I/P to the microprocessor. And this microcontroller is currently the most used worldwide in the WSNs environment [1].

**Table 3.** 8-bit AVR Assembly Instructions [23,24], cc means clock cycles.

| Asm | Operands | Description                          | Operation                       | cc |
|-----|----------|--------------------------------------|---------------------------------|----|
| ADD | Rd, Rr   | Add without Carry                    | $Rd \leftarrow Rd + Rr$         | 1  |
| ADC | Rd, Rr   | Add with Carry                       | $Rd \leftarrow Rd + Rr + C$     | 1  |
| MOV | Rd, Rr   | Copy Register                        | $Rd \leftarrow Rr$              | 1  |
| LDI | Rd, K    | Load Immediate                       | $Rd \leftarrow K$               | 1  |
| LD  | Rd, X    | Load Indirect                        | $Rd \leftarrow (X)$             | 2  |
| ST  | Z, Rr    | Store Indirect                       | $(Z) \leftarrow Rr$             | 2  |
| EOR | Rd, Rr   | Exclusive OR                         | $Rd \leftarrow Rd \oplus Rr$    | 1  |
| LSL | Rd       | Logical Shift Left                   | $C Rd \leftarrow Rd \ll 1$      | 1  |
| LSR | Rd       | Logical Shift Right                  | $Rd C \leftarrow 1 \gg Rd$      | 1  |
| ROL | Rd       | Rotate Left Through Carry            | $C Rd \leftarrow Rd \ll 1   C$  | 1  |
| ROR | Rd       | Rotate Right Through Carry           | $Rd C \leftarrow C    1 \gg Rd$ | 1  |
| BST | Rd, b    | Bit store from Bit in Reg to T Flag  | $T \leftarrow Rd(b)$            | 1  |
| BLD | Rd, b    | Bit load from T Flag to a Bit in Reg | $Rd(b) \leftarrow T$            | 1  |

The 8-bit AVR microcontroller's commands comprise operation codes and an operand. Table 3 shows the operands and clock cycles of the commands used in this paper. The 8-bit AVR-MCU consists of flash memory, SRAM, and EEPROM with a Harvard architecture. Our target device is ATmega128 which is widely used for sensor nodes in Wireless Sensor Networks [25]. ATmega128 has a 128 KB of flash memory, 4 KB SRAM, and 4 KB EEPROM. The device supports throughput of 16 MIPS 16 MHz and operates between 4.5-5.5 volts [26]. The AVR-MCU has 32 8-bit general-purpose registers, which are used for various purposes, e.g., basic private operations and bit operations. Specifically, the R26-R31 registers can be combined and used as three 16-bit registers, i.e., X, Y, and Z registers. These registers (X, Y, and Z) are used as pointers to indirectly specify a 16-bit address for data memory. The Status REGISTER (SREG) shows the status and result after Arithmetic Logic Unit (ALU) calculations.

### 3 Analysis of Existing Implementations of Hash Functions on 8-Bit AVR MCUs

Since Keccak algorithm was selected as SHA-3 standard in 2012, it has been implemented in various low-end-processors. Basically, the hardware implementation of SHA-3 is faster than SHA-2 and SHA-1 [27]. In particular, ARM-v8 architectures include special commands to increase the speed of the SHA-3 algorithm [28]. However, the software implementation of SHA-3 is much slower than SHA-2 [13–17]. Currently, the existing software-implemented SHA-3 algorithm in various IoT devices, including 8-bit AVR MCUs, follows the order of the standard SHA-3 implementation listed by NIST [13–16]. In the implementations based on order of standard way [13–16],  $\pi$  and  $\rho$  processes were implemented in once ( $\pi \sim \rho$  process), because rotate-operation can be executed while executing  $\pi$  process. Note that standard implementation execute  $f$ -function as follows:  $\theta$  process  $\rightarrow \pi \sim \rho$  process  $\rightarrow \chi \sim \iota$  process.

We found two representative SHA-3 implementations in 8-bit AVR MCUs [15, 16]. First, we analyzed the implementation of Otte et al. in AVR-Crypto-Lib; however, this implementation is based on the C-language (not 8-bit AVR assembly) [15]. The Otte et al.'s implementation of SHA-3 (256-bit) from AVR-Crypto-Lib requires 2,570,828 clock cycles to compute the digest of a 500 bytes message, which corresponds to a hash rate of 5,142 (CPB). This incurs nearly seven times the cost of 783 (CPB) of the SHA-2 implementation proposed by Otte et al. [15]. Therefore, we analyzed the SHA-3 implementation presented by Balasch et al., which is the most popular hash function implementations with an assembly on the 8-bit AVR microcontroller [16]. The Balash et al.'s implementation of SHA-3 (256-bit) requires 716,483 (CPB) to compute the digest of a 500 bytes message, which corresponds to a hash rate of 1,432 (CPB). This is the fastest implementation among existing SHA-3 implementations.

Balash et al. implemented the efficient shift-rotation for operating  $\pi \sim \rho$  process. The only rotation instructions in the 8-bit AVR microcontroller are rotations of 8-bit operands by 1 bit to the left (ROL) and right (ROR). In SHA-3,

for  $b = 1600$  and  $w = 64$ , the length of a single lane is 64 bits. to rotate 64-bit in  $\rho$  process, eight general-purpose registers are required. Typically, in an 8-bit AVR MCUs, a 1-bit left-rotation is implemented via a 1-bit logical left-shift (LSL) followed by seven 1-bit left-rotations through carry (ROL) and an addition with carry (ADC) [29]. In contrast, a 1-bit right-rotation is implemented by a 1-bit store to T in SREG ((BST)) followed by eight 1-bit right-rotations through carry (ROR) and a 1-bit load from T in SREG (BLD). Rotation of 64-bit data by  $n$  bits, where  $1 < n < 8$ , can be calculated by repeating the sequence of instructions to rotate a 64-bit quantity by one bit in the same direction  $n$  times. However, the execution time is not equal when executing 1-bit left-rotation and 1-bit right-rotation for 64-bit data; therefore, for efficient implementation, the  $n$ -bit rotation or  $(64-n)$ -bit rotation is employed as required [29]. In addition, for  $n$ -bit rotations greater than 8, the actual execution time of all  $n$ -bit shift-rotation operations can be reduced to 40 clock cycles or fewer if  $x = (x \ggg n)$  operations are replaced by  $x = (x \ggg n\%8)$ . In this process, operation of  $x = (x \ggg n/8)$  directly allocate and store in memory. When storing to memory ( $\rho$  process), the implementation of Balasch et al. combines  $\pi$  and  $\rho$  processes into a single process ( $\pi \sim \rho$ ). Note that Balasch et al. implemented SHA-3 based on the order of standard implementation [12].

However, even though the implementation presented by Balasch et al. is efficient for the 8-bit AVR MCUs, SHA-3 remains much slower than SHA-2. In low-end processors, accessing memory requires longer execution time compared to arithmetic operations. In addition, the *state* in SHA-3 where  $b = 1600$  requires at least 200 bytes, which is a heavier memory requirement compared to symmetric ciphers, where the *state* is only 128 bits. Therefore, in 8-bit AVR MCUs, it is important to optimize memory access to the *state* during SHA-3 execution.

**Table 4.** Number of times to memory accesses to the *state* of previous implementation, e.g., Balasch et al. and Otte et al. [13–17]

| Standard Method | initial $\theta$ | $\theta$ process | $\pi \sim \rho$ process | $\chi \sim \iota$ process | Total Access |
|-----------------|------------------|------------------|-------------------------|---------------------------|--------------|
| Load            | O                | O                | O                       | O                         | 7            |
| Store           | X                | O                | O                       | O                         |              |

Table 4 shows the number of times to memory access to *state* in the standard implementation of SHA-3 recommended by NIST. Here,  $\pi \sim \rho$  process indicates that  $\pi$  and  $\rho$  processes are combined, and  $\theta \sim \rho$  process indicates that  $\theta$  process and  $\rho$  process are combined.  $\chi$  and  $\iota$  processes are also combined with the same logic. Note that initial  $\theta$ , which creates  $D[x, z]$  in Algorithm 1, is part of  $\theta$  process. Table 4 shows that *state* stored in memory is accessed three times during  $\theta$  process and is accessed twice when  $\pi \sim \rho$  process is executed. In addition, the *state* is access twice in the  $\chi \sim \iota$  process. Therefore, existing standard implementations require seven memory accesses to the *state*. The size of *state* is

200 bytes ( $25 \times 64/8$ ), where  $b = 1600$ ,  $w = 64$ , and  $l = 16$ . In addition, the  $\theta$ ,  $\pi$ ,  $\rho$ ,  $\chi$ , and  $\iota$  processes should be executed in 24 rounds ( $12 + 6 \times 2$ ). Therefore, each execution of  $f$ -function of SHA-3 results in  $168 = 24 \times 7$  memory accesses for 200 bytes (200 bytes are *state* stored in memory).

In the embedded device, currently used in the WSNs, the high number of memory accesses to *state* causes low performance of the algorithm; therefore, in Sect. 4, we propose a new optimization method that reduces memory access without additional computation and lookup tables.

## 4 Proposed SHA-3 Implementations in 8-Bit AVR MCUs

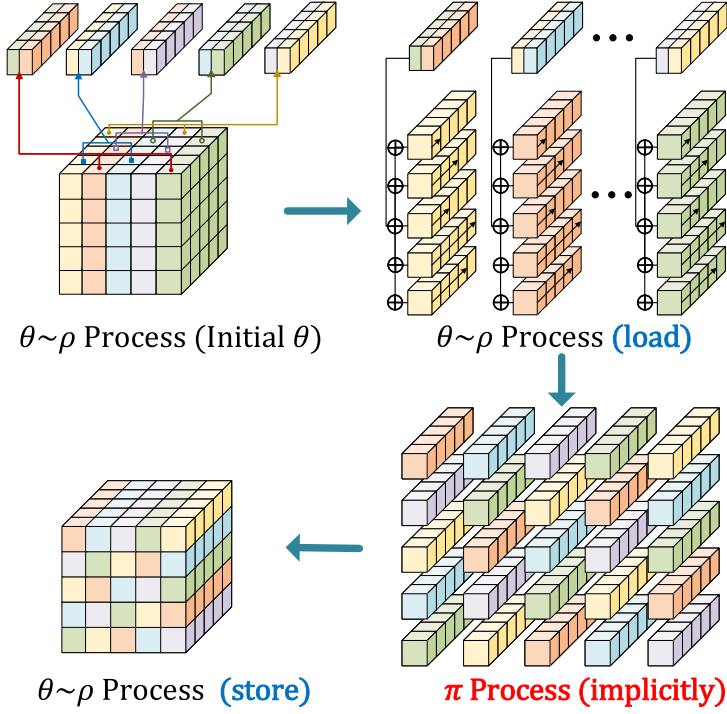
### 4.1 Main Idea

Here, we propose a generic SHA-3 optimization method that can be used for various platforms, e.g., low-end and high-end processors. In addition, we present the optimized techniques of SHA-3 on 8-bit AVR microcontroller. As mentioned in Sect. 3, in low-end-processors, memory access to *state* requires longer execution time compared to arithmetic operations. Therefore, it is important to schedule the use of general-purpose registers for the *state* efficiently and optimize memory access during SHA-3.

The main idea is to execute  $\pi$  process implicitly at minimum cost and combine the  $\theta$  and  $\rho$  processes into a single process ( $\theta \sim \rho$  process) to reduce accesses to the state. Figure 4 shows an overview of proposed SHA-3 method. Each lane of the *state* is operated independently in the  $\theta$  process; thus,  $\rho$  process can be applied to each lane. In other words, after generating  $D[x, z]$  through the initial  $\theta$ , we can execute the remained  $\theta$  process and  $\rho$  process. Note that initial  $\theta$  is part of  $\theta$  process. Twice memory accesses (Load and Store) occur for *state* executing  $\theta \sim \rho$  process.  $\pi$  process is performed implicitly while executing  $\theta \sim \rho$  process; also, changes only the position of the lane without directly affecting the value of the lane in the *state*. Therefore, our main idea implicitly executes  $\pi$  process, when storing the calculated *state* in the memory.

**Table 5.** Number of memory accesses to the *state*

| Standard Method | initial $\theta$ | $\theta$ process           | $\pi \sim \rho$ process | $\chi \sim \iota$ process | Total Access |
|-----------------|------------------|----------------------------|-------------------------|---------------------------|--------------|
| Load            | O                | O                          | O                       | O                         | 7            |
| Store           | X                | O                          | O                       | O                         |              |
| Proposed Method | initial $\theta$ | $\theta \sim \rho$ process | $\pi$ process           | $\chi \sim \iota$ process | Total Access |
| Load            | O                | O                          | X(Implied)              | O                         | 5            |
| Store           | X                | O                          | X(Implied)              | O                         |              |



**Fig. 4.** Proposed main idea

Table 5 shows the number of accesses to the *state* for the proposed implementation, compared the previous implementations based on the order of standard methods. The proposed method reduces memory accesses twice compared to the standard method. Therefore, a total of five accesses to the *state* occur, and when *f*-function is called, 120 ( $24 \times 5$ ) memory accesses to the 200-byte *state* occur. In other words, the proposed implementation reduces the amount of memory access 48 ( $168 - 120$ ) compared to the standard implementation.

Implementations of SHA-3 on the 8-bit AVR MCUs depend on the value of *b*. If *b* is less than 200 (*w* is less than 7), the general-purpose register can store all lanes of the state. However, if *b* is 400, 800, or 1600, the length of a single lane is 16-bit, 32-bit, and 64-bit, respectively; therefore, it is difficult to store all lanes of state in a general-purpose register. Therefore, an important point in implementing SHA-3 is how efficiently memory is accessed when the length of the lane is more than 8-bit. In this section, we present the SHA-3 implementation technique where *b* = 1600, which conforms to the Korean Cryptographic Module Validation Program (KCMVP) [30].

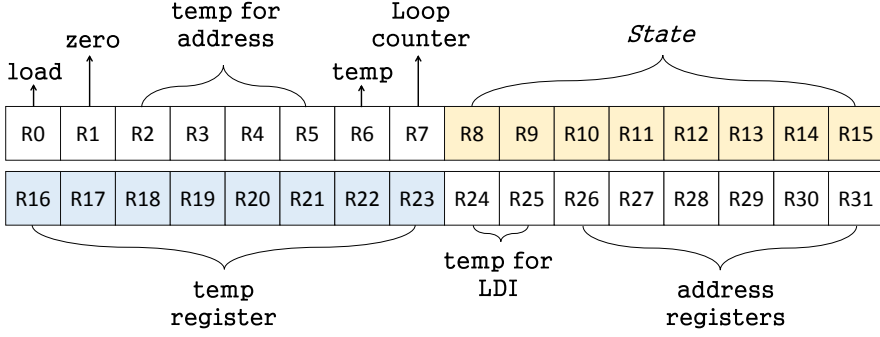


Fig. 5. Register scheduling for proposed implementation in 8-bit AVR MCUs

## 4.2 Proposed Implementation Technique on 8-Bit AVR MCUs

Figure 5 shows the register scheduling for the proposed method on the 8-bit AVR microcontroller. Here, eight registers are required to compute a single lane (64-bit) within the *state*. Here, registers R8-R15 are used to perform operations on the single lane. Note that registers R16-R23 and R8-R15 can also store a single lane. Registers R2-R5 are required to handle the *state* address value of in  $\theta \sim \rho$  process. R26:R27 keep the address value of *state* of  $f$ -function. R28:R29 keep the address value of  $D[x, z]$  stored initial  $\theta$ . In addition, R30:R31 keep the address value of the constant data used in  $\iota$  process.

Algorithm 2 shows the pseudocode for computation of the initial four lanes of the *state* in  $\theta \sim \rho$  process. Here, `load_state` is a macro that uses registers R24 and R25 as offsets to perform an XOR operation using the  $D[x, z]$  made in initial  $\theta$ , and the results are stored in registers R8-R15. `load_state` comprises assembly instructions as 2 ADD, 2 ADC, 16 LD, and 8 EOR. First, `load_state` loads the value of the lane to the registers R8-R15. Then, `load_state` loads 8-bit of  $D[x, z]$  to R0 and execute the EOR instruction for registers R8 (EOR R8, R0). In `load_state` macro, the sequence of loading  $D[x, z]$  and executing EOR instruction is repeated eight times, because one of lane is 64-bit: thus, `load_state` completes the computational part of  $\theta$  process for a single lane. In other words, for  $\theta$  process, only the memory update to the *state* remains. `load_temp` is a macro with the same logic as `load_state` that stores the computational value in registers R16-R23. `rotate_store_s` is a macro that executes  $\rho$  process for the eight registers that have completed  $\theta$  process stored in registers R8-R15 and updates registers R8-R15 to *state* using the offset stored in register R24. `rotate_store_t` is a macro of logic such as `rotate_store_s` that uses R16-R23.

In line 1 of Algorithm 2, `load_state` stores the results of  $\theta$  process for  $S[0]$ . This value is  $S'[0]$ . Here,  $\theta$  process has been completed; however, to execute  $\pi$  process implicitly,  $S[4]$  must be loaded to registers R16-R23, because the lane of  $S[0]$  move to  $S[4]$  in  $\pi$  process. Line 2 (Algorithm 2) sets the offset of  $S[4]$ , and line 3 sets offset  $D[4]$ . Then, line 4 makes  $S'[4]$  into the same logic as line 1, and, at line 5,  $\rho$  process is executed to make  $S'[0]$  to  $\bar{S}[0]$ . Line 5 updates  $S[4]$

---

**Algorithm 2.** AVR Assembly Codes for proposed combined  $\theta \sim \rho$  process for implicitly executing  $\pi$  process with initial four lane,  $D[i]$  ( $i \in [0, 4]$ ) :  $D[i, z]$  of initial  $\theta$ ,  $S[j]$  : 64-bit data of one lane of *state*,  $S'[j]$  : 64-bit data with  $\theta$  process calculated,  $S''[j]$  : 64-bit data with  $\theta$  and  $\rho$  process calculated,  $j \in [0, 24]$ .

---

|                                                                                                                                                                                                                                                                                                                                                                                                                          |                                                                                                                                                                                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p style="color: red; margin: 0;"><math>S[4] \leftarrow \bar{S}[0]</math> computation</p> <pre style="margin: 0;"> 1: load_state    //R8-R15 : <math>S'[0] \leftarrow (S[0] \oplus D[0])</math> 2: LDI R24, 32 // <math>S[4]</math> 3: LDI R25, 32 // <math>D[4]</math> 4: load_temp    //R16-R23 : <math>S'[4] \leftarrow (S[4] \oplus D[4])</math> 5: rotate_store_s // <math>S[4] \leftarrow \bar{S}[0]</math> </pre> | <p style="color: red; margin: 0;"><math>S[17] \leftarrow \bar{S}[14]</math> computation</p> <pre style="margin: 0;"> 10: LDI R17, 136 // <math>S[17]</math> 11: LDI R17, 16 // <math>D[2]</math> 12: load_temp    //R16-R23 : <math>S'[17] \leftarrow (S[17] \oplus D[2])</math> 13: rotate_store_s // <math>S[17] \leftarrow \bar{S}[14]</math> </pre>  |
| <p style="color: red; margin: 0;"><math>S[14] \leftarrow \bar{S}[4]</math> computation</p> <pre style="margin: 0;"> 6: LDI R24, 112 // <math>S[14]</math> 7: LDI R25, 32 // <math>D[4]</math> 8: load_state    //R8-R15 : <math>S'[14] \leftarrow (S[14] \oplus D[4])</math> 9: rotate_store_t // <math>S[14] \leftarrow \bar{S}[4]</math> </pre>                                                                        | <p style="color: red; margin: 0;"><math>S[15] \leftarrow \bar{S}[17]</math> computation</p> <pre style="margin: 0;"> 14: LDI R24, 120 // <math>S[15]</math> 15: EOR R25, R25 // <math>D[0]</math> 16: load_state    //R8-R15 : <math>S'[15] \leftarrow (S[15] \oplus D[0])</math> 17: rotate_store_t // <math>S[15] \leftarrow \bar{S}[17]</math> </pre> |

---

to  $\bar{S}[0]$  using the offset stored in register R24. Here, where  $\pi$  process is executed implicitly. In the standard method, the implementation order is  $\theta \rightarrow \pi \rightarrow \rho$ . In proposed method,  $\pi$  process is executed through the step of store in memory; thus, the offset used in  $\rho$  process, which is executed before  $\pi$  process, is the offset where  $\pi$  process should be applied. Our implementation is actually implemented in  $\theta \rightarrow \rho \rightarrow \pi$  order. Since the offset in table applied  $\pi$  process can be used rather than the offset in Table 2, no additional memory cost and generating cost is incurred. When computation of line 5 is completed,  $S[4]$  is updated to  $\bar{S}[0]$  with  $\theta$ ,  $\pi$ ,  $\rho$  processes applied. The proposed implementation repeats this sequence. In lines 6–17, the proposed implementation repeats same logic of lines 2–5. If Algorithm 2 is within same logic, executing  $\pi$  process implicitly, and  $\theta \sim \rho$  process for all lanes of *state* is completed.

The advantage of the proposed implementation on the 8-bit AVR MCUs is that there is no additional memory cost, which is significant for resource-limited environments. In other words, with the proposed implementation, memory accesses are reduced without having to generate a lookup table. In addition, no additional clock cycles occur in the proposed implementation. As with line 2 and 3, the cost of setting offsets is the same as the offset setup costs required by  $\theta \sim \rho$  process in the standard implementation method [15, 16].

## 5 Performance Analysis

We compared the performance of the proposed implementation method to that of existing implementation method on 8-bit AVR MCUs. In this evaluation, we used the following of SHA-3 (256-bit) parameters:  $b = 1600$ ,  $r = 1088$ , and  $c = 512$ . Thus, in the sponge structure absorbing process, the message block was divided into 136 bytes. Therefore, if the length of the message was a multiple of 136 bytes, the maximum performance relative to clock cycles per byte can be achieved. The performance was measured relative to execution time (*CPB*) by hash rate when hashing a byte of various messages in the 8-bit AVR microcontroller. The software was implemented using *Atmel Studio 7*, and the code was compiled using the `-O2` option in *avr-gcc version 5.4.0*. All reported execution times were determined using the cycle-accurate instruction set simulator in *Atmel Studio 7*, where we used the ATmega128 microcontroller as the target device.

**Table 6.** Performance of proposed SHA-3 Implementations by hash rate when hashing a byte of various message in 8-bit AVR microcontroller, hash rate represent cyc/byte (*CPB*) [15, 16]

| Reference           | Algorithm        | Language | Length of message byte  |                         |                         |
|---------------------|------------------|----------|-------------------------|-------------------------|-------------------------|
|                     |                  |          | 50 byte                 | 100 byte                | 500 byte                |
| <b>This Paper</b>   | SHA-3(256-bit)   | Asm      | <b>2667</b><br>(+25.1%) | <b>1333</b><br>(+25.7%) | <b>1073</b><br>(+25.0%) |
| Otte et al. [15]    | SHA-3 (256-bit)  | C        | 12854                   | 6427                    | 5142                    |
| Balasch et al. [16] | SHA-3 (256-bit)  | Asm      | 3560 (-)                | 1795 (-)                | 1432 (-)                |
| Balasch et al. [16] | SHA-256          | Asm      | 672                     | 668                     | 532                     |
| Balasch et al. [16] | Blake (256-bit)  | Asm      | 714                     | 708                     | 562                     |
| Balasch et al. [16] | Grøstl (256-bit) | Asm      | 1220                    | 1012                    | 686                     |
| Balasch et al. [16] | Photon (256-bit) | Asm      | 9723                    | 7982                    | 4788                    |

Table 6 compares the proposed implementation to previous SHA-3 implementations, e.g., the implementations proposed by Otte et al. [15] and Balasch et al. [16]. In addition, we compared the proposed implementation to a previous SHA-2 implementation and three SHA-3 candidates. When hashing a 500 bytes message, we obtained an overall execution time of 1073 *CPB* for the Assembly implementation. The proposed implementation obtained a 79.1% performance improvement over the implementation method proposed by Otte et al. and a 25.0% performance improvement over the method proposed by Balasch et al. The existing implementation [16] incurs nearly three times the *CPB* cost for SHA-256. However, the proposed implementation requires the *CPB* by nearly two times to SHA-256 [16], compared to Balash et al.'s implementation [16].

## 6 Concluding Remarks

In this paper, we have proposed software optimization method for SHA-3 on 8-bit AVR MCUs. The proposed method optimizes SHA-3 process without requiring a lookup table and additional operations from a memory access perspective. With the proposed optimization method, our implementation on an 8-bit AVR microcontroller obtained 25.0% performance improvement over the previous best results on the same condition [16]. The proposed method is applicable to a variety of algorithms using SHA-3, e.g., HASH\_DRBG, HMAC\_DRBG, PBKDF2, and digital signature algorithms. Furthermore, our method is a generic optimization method that can be applied to various platforms, e.g., both high-end and low-end processor environments.

## References

1. Park, S.-E., Hwang, C.-G., Park, D.-C.: Internet of Things (IoT) on system implementation with minimal Arduino based appliances standby power using a smart-phone alarm in the environment. *JKIECS* **10**, 1175–1182 (2015)
2. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. In: Katz, J., Shacham, H. (eds.) *CRYPTO 2017*. LNCS, vol. 10401, pp. 570–596. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-63688-7\\_19](https://doi.org/10.1007/978-3-319-63688-7_19)
3. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) *CRYPTO 2005*. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005). [https://doi.org/10.1007/11535218\\_2](https://doi.org/10.1007/11535218_2)
4. Rijmen, V., Oswald, E.: Update on SHA-1. *IACR Cryptology ePrint Archive* 2005:10 (2005)
5. De Cannière, C., Rechberger, C.: Finding SHA-1 characteristics: general results and applications. In: Lai, X., Chen, K. (eds.) *ASIACRYPT 2006*. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006). [https://doi.org/10.1007/11935230\\_1](https://doi.org/10.1007/11935230_1)
6. Manuel, S.: Classification and generation of disturbance vectors for collision attacks against SHA-1. *Des. Codes Cryptogr.* **59**(1–3), 247–263 (2011)
7. Khovratovich, D., Rechberger, C., Savelieva, A.: Bicliques for preimages: attacks on Skein-512 and the SHA-2 family. In: Canteaut, A. (ed.) *FSE 2012*. LNCS, vol. 7549, pp. 244–263. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-34047-5\\_15](https://doi.org/10.1007/978-3-642-34047-5_15)
8. Lamberger, M., Mendel, F.: Higher-order differential attack on reduced SHA-256. *IACR Cryptology ePrint Archive* 2011:37 (2011)
9. Mendel, F., Nad, T., Schl  ffer, M.: Improving local collisions: new attacks on reduced SHA-256. In: Johansson, T., Nguyen, P.Q. (eds.) *EUROCRYPT 2013*. LNCS, vol. 7881, pp. 262–278. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38348-9\\_16](https://doi.org/10.1007/978-3-642-38348-9_16)
10. Dobraunig, C., Eichlseder, M., Mendel, F.: Analysis of SHA-512/224 and SHA-512/256. *IACR Cryptology ePrint Archive* 2016:374 (2016)
11. Sasaki, Y., Wang, L., Aoki, K.: Preimage attacks on 41-step SHA-256 and 46-step SHA-512. *IACR Cryptology ePrint Archive* 2009:479 (2009)
12. Dworkin Morris, J.: SHA-3 standard: permutation-based hash and extendable-output functions (2015). <https://doi.org/10.6028/NIST.FIPS.202>