Copyright ⓒ 2018 KSII

w-Bit Shifting Non-Adjacent Form Conversion

Doo-Hee Hwang[1] and Yoon-Ho Choi[1]
[1] School of Computer Science and Engineering, Pusan National University
Busan, Korea
[e-mail: dooheeh@gmail.com; yhchoi@pusan.ac.kr]
*Corresponding author: Yoon-Ho Choi

Received October 11, 2017; revised November 17, 2017; accepted March 9, 2018; published July 31, 2018

Abstract

As a unique form of signed-digit representation, non-adjacent form (NAF) minimizes Hamming weight by removing a stream of non-zero bits from the binary representation of positive integer. Thanks to this strong point, NAF has been used in various applications such as cryptography, packet filtering and so on. In this paper, to improve the NAF conversion speed of the NAF_w algorithm, we propose a new NAF conversion algorithm, called w-bit Shifting Non-Adjacent Form($SNAF_w$), where w is width of scanning window. By skipping some unnecessary bit comparisons, the proposed algorithm improves the NAF conversion speed of the NAF_w algorithm. To verify the excellence of the $SNAF_w$ algorithm, the NAF_w algorithm and the $SNAF_w$ algorithm are implemented in the 8-bit microprocessor ATmega128. By measuring CPU cycle counter for the NAF conversion under various input patterns, we show that the $SNAF_2$ algorithm not only increases the NAF conversion speed by 24% on average but also reduces deviation in the NAF conversion time for each input pattern by 36%, compared to the NAF_2 algorithm. In addition, we show that $SNAF_w$ algorithm is always faster than NAF_w algorithm, regardless of the size of w.

Keywords: signed-digit representation, non-adjacent form, Hamming weight, encoding, public key cryptography

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2017-2014-1-00743) supervised by the IITP(Institute for Information & communications Technology Promotion) and basic science research program through national research foundation korea (NRF) funded by the ministry of science, ICT and future planning (NRF-2015R1D1A1A01057888).

http://doi.org/10.3837/tiis.2018.07.025 **ISSN : 1976-7277**

1. Introduction

In mathematical notation of numbers, signed-digit representation means a positional system for a signed digit. Since signed-digit representation can remove dependency on carry propagation, it is widely used for the fast addition of integers.

One integer can have various signed-digit representations. For example, the positive integer of 15 has five types of signed-digit representations as below. Here, $\bar{1}$ means -1.

$$(01111)_2 = 8 + 4 + 2 + 1$$

$$(1\bar{1}111)_2 = 16 - 8 + 4 + 2 + 1$$

$$(10\bar{1}11)_2 = 16 - 4 + 2 + 1$$

$$(100\bar{1}1)_2 = 16 - 2 + 1$$

$$(1000\bar{1})_2 = 16 - 1$$

Integer encoding methods for signed-digit representation, which remove a stream of non-zero bits, include Booth encoding [1], a Fibonacci encoding [2] and Non-Adjacent Form (NAF). Compared to the other encoding methods, NAF has been used in diverse applications such as public key cryptography [3-6], packet filtering [7-8], constructing a ternary FCSRs [9-10] and analysis for medical predictive models [11]. In particular, NAF has been actively used in some exponentiation-based public-key cryptographic algorithms including RSA [12] and Elliptic Curve Cryptography (ECC) [13-14].

Non-adjacent form (NAF) is a particular form of signed-digit representation, which is used in the binary numeral system [15-17]. As a unique signed binary representation, NAF of a positive integer k is generally expressed into equation (1) below:

$$\mathrm{NAF}(k) = \sum_{i=0}^{m} k_i 2^i \text{ , where } k_i \in \{-1, 0, 1\}, k_m \neq 0 \tag{1}$$

Table 1. Terms and Notation

Term	Notation
k	positive integer
$k_{(2)}$	binary representation of k
l	length of $k_{(2)}$
w	width of scanning window for bit stream search, where $w \geq 2$
m	length of NAF, where $l - w + 2 \leq m \leq l + 1$
k_i or u_i	a nonzero coefficient, where $\lvert k_i \rvert, \lvert u_i \rvert < 2^{w-1}$ and $0 \leq i \leq m$
$\mathrm{NAF}_w(k)$	NAF of $k_{(2)}$, i.e., $(k_m, k_{m-1}, \ldots, k_1, k_0)$
x	decimal value of the rightmost w bits

As a method for integer encoding that minimizes Hamming weight, NAF removes a stream of non-zero bits from the binary representation of an integer. For example, in the case of the aforementioned signed-digit representation of positive integer 15, NAF is $(1000\bar{1})_2$ with

$k_i \cdot k_{i+1} = 0$ and the minimum Hamming weight. On average, binary representation of k in the length of l consists of non-zero bits that account for half of the total number of bits. On the other hand, NAF reduces the number of non-zero bits to $(l/3)$ on average in binary representation that has the same length. In **Table 1**, we summarize terms and their notation used in this paper.

As the NAF_w algorithm [18-19] is most commonly used for NAF conversion in many application fields, it is denoted into the canonical NAF conversion algorithm. By removing a stream of non-zero bits from $k_{(2)}$, the NAF_w algorithm returns $\text{NAF}_w(k)$, which has the minimum Hamming weight. Specifically, the Hamming weight of $\text{NAF}_w(k)$ in the length of m is $m/(w+1)$. Also k_i of $\text{NAF}(k)$ described in equation (1) can only be {-1, 0, 1}. On the other hand, k_i of $\text{NAF}_w(k)$ can be any odd integer satisfying $|k_i| < 2^{w-1}$. For example, when w is 3, k_i value has one of {-3, -1, 0, 1, 3}. $\text{NAF}_w(k)$ in the length of m is expressed into equation (2) below:

$$\text{NAF}_w(k) = \sum_{i=0}^{m} k_i 2^i \text{, where } |k_i| < 2^{w-1}, w \geq 2, k_m \neq 0. \quad (2)$$

Algorithm 1. NAF_w algorithm

Input : k, w $(w \geq 2)$

Output : $\text{NAF}_w(k) = k_m, k_{m-1}, \ldots, k_1, k_0$

```
1 :  i ← 0
2 :  while k ≥ 1 do
3 :      if k is odd then
4 :          k_i ← k mod 2^w
5 :          if k_i ≥ 2^(w-1) then
6 :              k_i ← k_i − 2^w
7 :          end if
8 :          k ← k − k_i
9 :      else
10 :         k_i ← 0
11 :     end if
12 :     k ← k/2 , i ← i + 1
13 : end while
14 : Return (k_m, k_(m-1), ..., k_1, k_0)
```

In **Algorithm 1**, we describe a pseudocode for implementing the NAF_w algorithm. If the value of least significant bit (LSB) in k is 0, k_i is simply set into 0 in line 10. Otherwise, the rightmost w bits in k are set into k_i in line 4. However, k_i should be expressed into the signed digit representation with the minimum Hamming weight. Thus, if k_i is greater than or equal to 2^{w-1}, k_i is set into a negative coefficient $k_i - 2^w$ in line 6 and then, k is set into $k - k_i$ in line 8. We denote the operations in lines 3 to 12 into comparison-and-shift encoding. Also we denote the operation in line 2 into bit comparison for conditional NAF conversion. If the length of $k_{(2)}$ in NAF_w algorithm is l, l numbers of comparison-and-shift encoding are

required and l numbers of bit comparison for conditional NAF conversion are required.

In this paper, to convert k into $\mathrm{NAF}_w(k)$ faster than the NAF_w algorithm, we propose a new NAF conversion algorithm, called the *w*-bit Shifting Non-Adjacent Form (SNAF_w) algorithm. When converting k into $\mathrm{NAF}_w(k)$, where the length of $k_{(2)}$ is l, the NAF_w algorithm respectively requires l numbers of bit comparison for conditional NAF conversion and comparison-and-shift encoding. Compared to the NAF_w algorithm, since $k_{(2)}$ consists of $l/2$ non-zero bits in probability, the SNAF_w algorithm checks whether the NAF conversion of $k_{(2)}$ is completed only when the rightmost w bits in $k_{(2)}$ is less than 2^{w-1}. Therefore, the SNAF_w algorithm requires $l/2(\mathrm{w}+1)$ numbers of bit comparisons on average to check whether the NAF conversion is completed. That is, the SNAF_w algorithm reduces the number of bit comparisons for conditional NAF conversion.

Also, when computing k_i with the minimum Hamming weight, the NAF_w algorithm sets the rightmost w bits into 0s in line 8 and then, shifts $k_{(2)}$ by one bit to the right side in line 12 and compare whether the bit value is 0. The comparison-and-shift encoding is repeated for the remaining $w-2$ bits within *w*-width scanning window. Compared to the $\mathrm{NAF}_w(k)$ algorithm, the SNAF_w algorithm shifts $k_{(2)}$ by w bits to the right side without setting the w bits in the *w*-width scanning window into 0s. That is, by skipping comparison-and-shift encoding for the leftmost $w-1$ bits in the *w*-width scanning window, the SNAF_walgorithm requires $2l/(w+1)$ numbers of comparison-and-shift encoding on average. As a result, the SNAF_w algorithm converts k into NAF faster than the NAF_w algorithm by reducing the number of bit comparisons for conditional NAF conversion and comparison-and-shift encoding. The SNAF_w algorithm also shows the faster conversion speed than the other well-known NAF conversion algorithms [20-24].

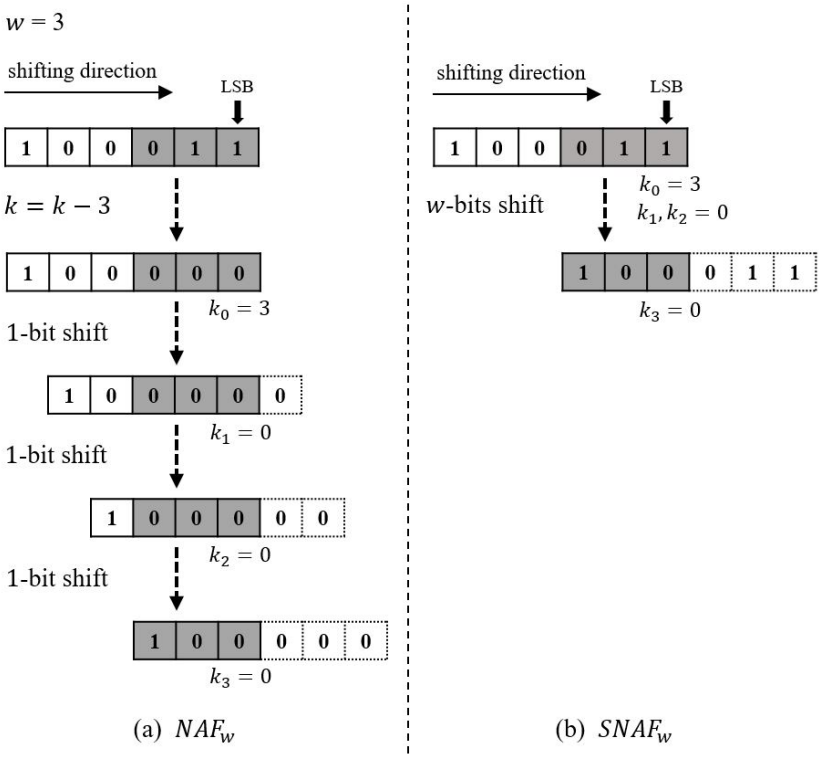

Fig. 1. NAF conversion examples using (a) $\mathbf{NAF}_w$ and (b) $\mathbf{SNAF}_w$

In **Fig. 1**, let us consider an example that shows how the NAF_3 algorithm and the SNAF_3 algorithm convert a positive odd integer k(=35) into $\mathrm{NAF}_3(35)$ respectively. We assume $k_{(2)} = (100011)_2$. In **Fig. 1(a)**, since the value of the rightmost 3 bits, marked into the grey color, is odd and less than 2^2, the NAF_3 algorithm sets k_0 into 3 and then, the rightmost 3 bits into 0. After shifting $k_{(2)}$ by one bit to the right side, k is not odd. Thus, the NAF_3 algorithm shifts $k_{(2)}$ by one bit to the right side repeatedly and thus, k_1, k_2 and k_3 are computed into 0. In this example, while computing k_0 to k_3, the NAF_3 algorithm requires one memory access for setting 3 bits into 0s and three numbers of comparison-and-shift encodings. On the other hand, as shown in **Fig. 1(b)**, the SNAF_3 algorithm shifts $k_{(2)}$ by 3 bits to the right side without setting the rightmost 3 bits into 0s.

The SNAF_3 algorithm also computes k_0 to k_3 by only one comparison-and-shift encoding. That is, in the example of **Fig. 1**, the NAF_3 algorithm respectively requires 4 numbers of bit comparison for conditional NAF conversion and comparison-and-shift encoding. On the other hand, SNAF_3 algorithm respectively requires 2 numbers of bit comparison for conditional NAF conversion and comparison-and-shift encoding. Thus, the SNAF_3 algorithm converts $(100011)_2$ into NAF much faster than the NAF_3 algorithm. In section 3, we show how the SNAF_w algorithm operates in details.

Contributions of this paper can be summarized as follows. First, we propose a new NAF conversion algorithm that improves the speed of NAF conversion regardless of the performance of microprocessors. Second, we show theoretical analysis results that explain why the computational speed of the proposed algorithm is faster than the other NAF conversion algorithms. Third, from the experimental results in the 8-bit microprocessor ATmega128, we show that compared to the NAF_w algorithm, the SNAF_w algorithm increases the speed of NAF conversion by 20% on average and 25% at the maximum.

This paper consists of as follows. Section 2 overview the characteristics of the recently proposed NAF conversion algorithms. In section 3, we describe the SNAF_w algorithm in details. In section 4, we show the evaluation results of the proposed algorithm. Finally, we summarize this paper in section 5.

2. Related Work

In this section, we overview the well-known NAF conversion algorithms, which improve the NAF_w algorithm in terms of memory efficiency and the NAF conversion speed in scalar multiplication and so on.

2.1 FAN

FAN algorithm is proposed to reduce memory usage of the NAF_w algorithm [21]. Specifically, since the NAF_w algorithm converts $k_{(2)}$ into NAF by checking every bit in the right-to-left direction, called the right-to-left encoding. In the NAF_w algorithm, the converted NAF values are buffered in memory until scalar multiplication or modular exponentiation is completed. That is, the additional memory is required because the buffer with the NAF values cannot be reused until completing scalar multiplication or modular exponentiation.

When implementing scalar multiplication or modular exponentiation in devices such as smart cards, the size of memory is an important issue. This is because the embedded system has a low memory resource. To resolve the aftermentioned unnecessary memory usage, the FAN algorithm is designed to check every bit in the left-to-right direction, called the

left-to-right encoding. This is because in general, scalar multiplication and modular exponentiation are conducted in the left-to-right direction. By checking each bit in the left-to-right direction, the buffer for storing a single NAF, i.e., k_i, value can be reused.

2.2 Compact Encoding NAF

To convert $k_{(2)}$ in the length l into the NAF for w=2, two bits for expressing each k_i are used to represent "$-$" sign for encoding. Thus, to store the NAF values converted from $k_{(2)}$ in the length of l, $2(l+1)$ bits are required. In order to reduce the required memory size, the compact encoding NAF algorithm was proposed [22].

The compact encoding NAF is a simple right-to-left encoding method based on the characteristics of $(k_i \cdot k_{i+1} = 0)$ that "1" or "-1" of NAF is always adjacent to "0".

$$R = \begin{cases} 01 \rightarrow 01 \\ 0\bar{1} \rightarrow 11 \\ 0 \rightarrow 0 \end{cases} \qquad R^{-1} = \begin{cases} 01 \rightarrow 01 \\ 11 \rightarrow 0\bar{1} \\ 0 \rightarrow 0 \end{cases} \tag{3}$$

As R conversion rules in equation (3) are used, the NAF of positive integer is encoded in binary representation without the bit of "-1". For example, $(1000\bar{1})_2$, which is the NAF of positive integer 15, is changed to $(10011)_2$. On the contrary, R^{-1} conversion rules can be used also for conversion from compact encoding NAF to NAF.

Algorithm 2. MOF_w : Left-to-Right Encoding for $w = 2$

Input : a non-zero n-bit binary string, $d = d_{n-1}|d_{n-2}|\ldots|d_1|d_0$

Output : $u_n|u_{n-1}|\ldots|u_1|u_0$ of d

1 : $u_n \leftarrow d_{n-1}$
2 : **for** i from $n-1$ down to 1 **do**
3 : $u_i \leftarrow d_{i-1} - d_i$
4 : **end for**
5 : $u_0 \leftarrow -d_0$
6 : Return $(u_n, u_{n-1}, \ldots, u_1, u_0)$

2.3 Mutual Opposite Form

The Mutual Opposite Form (MOF_w) algorithm [23] is designed to convert $k_{(2)}$ into NAF using the right-to-left encoding. Also, by using the right-to-left encoding, the MOF_w algorithm converts $k_{(2)}$ into MOF, which is a signed binary representation with the same Hamming weight as NAF. By following the below conditions, n-bit $k_{(2)}$ is expressed into $(n+1)$ numbers of MOFs, i.e., u_i.

1. Sign values of adjacent non-zero bits are opposite to each other.
2. If all bits are not 0s, the most significant bit (MSB) and LSB are 1 and $\bar{1}$ respectively.

By converting $k_{(2)}$ into NAF using the left-to-right encoding, the MOF_w algorithm can reduce memory space, which is required to store the NAF conversion value, by much as n bits.

In **Algorithm 2**, we show the detailed operation of the MOF_w algorithm using the left-to-right encoding for $w = 2$.

2.4 Complementary Canonical Sliding Window Recoding

The CCS(Complementary Canonical Sliding window) recoding algorithm expresses NAF into an extension of complement expressions [24]. The CCS recoding algorithm converts $k_{(2)}$ into NAF by consecutively using the method of calculating 1's complement, the NAF_w algorithm and a sliding window method. In the elliptic curve cryptosystem, the CCS representation can be applied to reduce the average number of the point addition operation in scalar multiplication.

When being used for scalar multiplication in ECC, the FAN algorithm, the compact encoding NAF algorithm and the MOF_w algorithm are implemented by using less memory than the NAF_w algorithm. However, the NAF conversion speed of the FAN algorithm and the compact encoding NAF algorithm is much slower than the NAF_w algorithm due to many numbers of memory accesses. Also, the FAN algorithm and the compact encoding NAF algorithm can be used only when $w = 2$. In the MOF_w algorithm, if w is larger than two, many numbers of memory access occur in the encoding process. Due to many numbers of memory access, the conversion speed of the MOF_w algorithm is much slower than the NAF_w algorithm. In the CCS recoding algorithm, since the methods of calculating 1's complement, the NAF_w algorithm and a sliding window method are consecutively used, the speed of the CCS recoding algorithm is also slower than the NAF_w algorithm.

In addition to the algorithms mentioned above, there are various algorithms that generate signed-digit representation. First, the algorithm proposed in [25] combines the {0, 1, 3}-NAF algorithm and the {-1, 0, 1}-NAF algorithm. The {-1, 0, 1}-NAF algorithm uses the pre-generated look up table for conversion. That is, the {-1, 0, 1}-NAF algorithm and the algorithm proposed in [25] have many number of memory access. Therefore, they are slower than the NAF_w algorithm. Second, algorithms proposed in [26] and [27] are aim at reducing the Hamming weight of Radix-r representation. The NAF_w algorithm is the same as Radix-2 representation, and the Hamming weight decreases as size of w increases. When Radix is 2, Hamming weight of the algorithms proposed in [26] and [27] is same as Hamming weight of the NAF_w algorithm. However, they have many number of memory access. Therefore, the algorithms proposed in [26] and [27] are slower than the NAF_w algorithm. Most NAF conversion algorithms are slower than the NAF_w algorithm.

An improve signed-digit representation for the multiplier-free implementation of constant vector multiplication was proposed in [28]. This approach is intended to make it suitable for circuit design unlike the NAF conversion algorithms mentioned above. In this paper, we target on improving the NAF conversion speed of the NAF_w algorithm through improvement of the algorithm itself.

3. Proposed Algorithm

In this section, we describe how the SNAF_w algorithm converts k into $\mathrm{NAF}_w(k)$. We also show the theoretical performance analysis results.

Algorithm 3. SNAF$_w$ algorithm

Input : positive integer k, w $(w \geq 2)$

Output : NAF$_w(k) = k_m, k_{m-1}, \ldots, k_1, k_0$

```
1 :  i ← 0 , x
2 :  while true do
3 :      while k is even do
4 :          k_i ← 0, k ← k ≫ 1, i ← i + 1
5 :      end while
6 :      x ← k&(2^w − 1)
7 :      if x < 2^(w−1) then
8 :          k_i ← x
9 :          if x = k then
10 :             i ← i + 1, break
11 :         end if
12 :     else
13 :         k_i ← x − 2^w, k ← k + 2^w
14 :     end if
15 :     k_(i+w−1) ← 0, ..., k_(i+1) ← 0, k ← k ≫ w, i ← i + w
16 : end while
17 : Return (k_m, k_(m−1), ..., k_1, k_0)
```

3.1 Operation of $SNAF_w$

In **Algorithm 3**, we describe the pseudocode of the SNAF$_w$ algorithm. The SNAF$_w$ algorithm consists of two functional modules: (1) single comparison-and-shift encoding for w bits and (2) conditional completion of the NAF conversion.

3.1.1 Single Comparison-and-Shift Encoding for w Bits : When k is even, the SNAF$_w$ algorithm sets k_i into 0 and then, shifts k to the right side by one bit. The one-bit comparison-and-shift encoding is repeated until k is an odd number as shown in lines 3 and 5. When k is odd, we compute k_i by following the procedure in lines 6 to 15. At line 6, x is set into the decimal value of the rightmost w bits in k. Next, as shown in lines 7 to 10, if x is less than 2^{w-1}, k_i is set into x. If x is equal to k, the NAF conversion is completed because $x = k$ implies that the other bits except for the rightmost w bits in k are 0s. However, if x is larger than or equal to 2^{w-1}, k_i is set into $x - 2^w$ because 2's complement of x should be set into k_i. Also, as k value increases by 2^w, carry is generated.

This is because even after 2's complement of x is set into k_i, k should be equal to NAF$_w(k)$. Finally, as shown in line 15, k_{i+1} to k_{i-w+1} are set into 0s. That is, the SNAF$_w$ algorithm does a single comparison-and-shift encoding for w bits instead of w numbers of one-bit comparison-and-shift encoding. As a result, the SNAF$_w$ algorithm reduces the number of comparison-and-shift encoding compared to the NAF$_w$ algorithm.

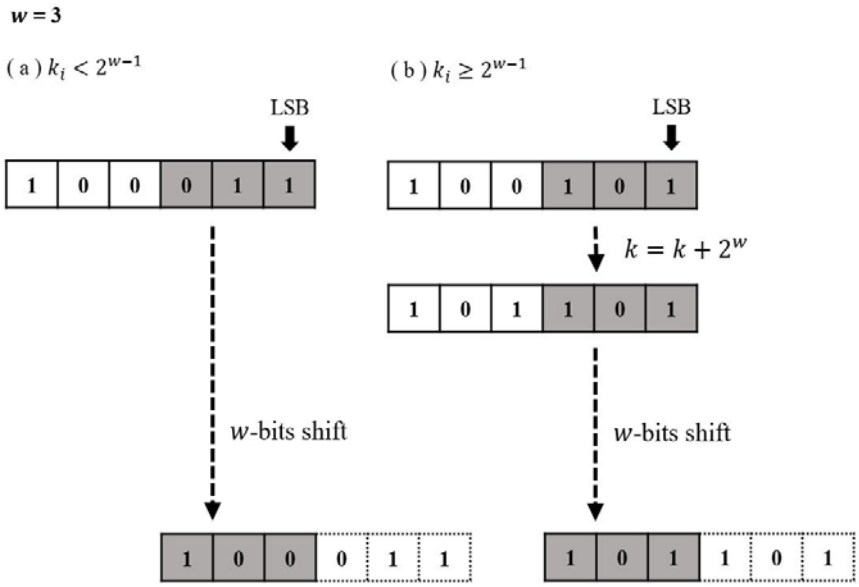

Fig. 2. Operation process of **SNAF$_3$** algorithm

For example, from $(100011)_2$ for k =35 in **Fig. 2(a)**, the binary representation of x is set into $(011)_2$ which is less than 2^{3-1}. Thus, after setting k_i into $(011)_2$ and setting k_{i+1} to k_{i+2} into 0, $(100011)_2$ shifts to the right side by 3 bits and the binary representation of x is set into $(100)_2$. On the other hand, from $(100101)_2$ for k =35 in **Fig. 2(b)**, the binary representation of x is set into $(101)_2$ which is larger than 2^{3-1}. After k_i is set into $\overline{3}$, the binary representation of k is updated into $(101101)_2$ with carry at the w_{th} position. Finally, $(101101)_2$ is shifted to the right side by 3 bits and the binary representation of x is set into $(101)_2$.

Compared to the NAF$_w$ algorithm that requires l numbers of comparison-and-shift encoding, the SNAF$_w$ algorithm takes only $(2 \times l)/(w+1)$ numbers of comparison-and-shift encoding on average. In section 3.2.1, we show the analysis results in details.

3.1.2 Conditional NAF Conversion : The i_{th} element of NAF$_w(k)$, i.e., k_i, is an odd integer, where $|k_i| < 2^{w-1}$. Among k_is, note that k_m should be a positive integer because the value of NAF$_w(k)$, i.e., $(k_m \times 2^m) + \cdots + (k_0 \times 2^0)$, is positive. That is, k_m is a positive odd integer, which is less than 2^{w-1}. Thus, as shown in lines 9 to 11, if k_m is equal to k, the SNAF$_w$ algorithm can complete the NAF conversion.

For example, let us assume that w is equal to 3 and the length of $k_{(2)}$ is equal to 3. Since $k_{(2)}$ is a positive binary number, $k_{(2)}$ can be one of $(001)_2$, $(011)_2$, $(010)_2$, $(100)_2$, $(110)_2$, $(101)_2$ and $(111)_2$. When $k_{(2)}$ is either $(001)_2$ or $(011)_2$, x is set into either $(001)_2$ or $(011)_2$ respectively. That is, since x is equal to k, the NAF conversion is completed. When $k_{(2)}$ is either $(010)_2$, $(100)_2$ or $(110)_2$, x is set into either $(010)_2$, $(100)_2$ or $(110)_2$ respectively. Since x is not the positive odd integer less than 2^{3-1}, comparison-and-shift encoding for w bits is conducted and then, $k_{(2)}$ is set into either $(001)_2$ or $(011)_2$. Since x becomes equal to k, the NAF conversion is completed. Finally, when $k_{(2)}$ is either $(101)_2$ or $(111)_2$, x is set into either $(101)_2$ or $(111)_2$ respectively. After an one-bit

comparison-and-shift encoding is conducted, carry is generated. Thus, $k_{(2)}$ is set into $(001)_2$ and the NAF conversion is completed. From this example, we observe that k_m is a unique positive odd integer less than 2^{w-1}.

Compared to the NAF$_w$ algorithm that requires l numbers of bit comparison to check whether the NAF conversion is completed, the SNAF$_w$ algorithm takes $l/2(w+1)$ numbers of bit comparison. In section 3.2.2, we show the analysis results in details.

3.2 Performance Analysis of the $SNAF_w$ Algorithm

In this section, we analyze how the SNAF$_w$ algorithm takes $(2 \times l)/(w+1)$ numbers of comparison-and-shift encoding and $l/2(w+1)$ numbers of bit comparison on average for conditional completion of the NAF conversion.

For the w-width scanning window, each of 2^w bit patterns is generated with the probability $1/2^w$. Since $k_{(2)}$ consists of $l/2$ numbers of non-zero or zero bits in probability, the probability that the i_{th} bit of $k_{(2)}$ is zero or one is 1/2, i.e., Pr(0)=1/2 and Pr(1)=1/2. Given 2^w bit patterns, we classify them into four pattern groups from P_1 to P_4. First, P_1 consists of 2^{w-2} bit patterns whose LSBs in the w-width scanning window are 1 and whose decimal values are smaller than 2^{w-1}. Second, P_2 consists of 2^{w-2} bit patterns whose LSBs of the w-width scanning window are 1 and whose decimal values are larger than 2^{w-1}. Third, P_3 consists of $2^{w-1}-1$ bit patterns whose LSBs of the w-width scanning window are 0, except for a bit sequence with all zero bits. Finally, P_4 consists of a single bit pattern with all zero bits. For example, when w is 3, $(001)_2$ and $(011)_2$ bit patterns are classified into P_1. $(101)_2$ and $(111)_2$ bit patterns are classified into P_2. Also, $(010)_2$, $(100)_2$ and $(110)_2$ are classified into P_4. Finally, $(000)_2$ is classified into P_4.

3.2.1 Number of Comparison-and-Shift Encoding : From **Algorithm 3**, bit patterns in P_1 to P_4 takes the different types of comparison-and-shift encoding. Bit patterns in P_1 and P_2 take only a single comparison-and-shift encoding. On the other hand, bit patterns in P_4 take w numbers of comparison-and-shift encoding. Bit patterns in P_3 belong to P_1 or P_2 after being shifted to the right side until LSB of the w-width scanning window is 1. In this case, comparison-and-shift encoding by as much as the number of shifts to the right side is taken. Then, P_1 or P_2 take one comparison-and-shift encoding. Also, for patters in P_3, the total number of the right-side shifts until LSB in the w-width scanning window is 1 is given into $2^w - w - 1$.

Theorem. 1. (Number of bit shifts for $\boldsymbol{P_3}$) For patters in $\boldsymbol{P_3}$, the **SNAF$_w$** algorithm takes $\boldsymbol{2^w - w - 1}$ number of the right-side shifts until LSB in the $\boldsymbol{w}$-width scanning window is 1.

$Proof.$

$Since\ f(w=2)=1, f(w=3)=4, f(w=4)=11, f(w=5)=26, f(w=6)=51, \dots,$

$f(n+1)-f(n)=2^{n+1}-1\ for\ n=w-1.$

$$f(n)=f(1)+\sum_{k=1}^{n-1}\left(2^{k+1}-1\right)=1+2\sum_{k=1}^{n-1}2^k-n+1$$

$$=1+2^2(2^{n-1}-1)-n+1=2^{n+1}-n-2.$$

$$\therefore f(w)=2^w-w-1. \quad (4)$$

Also, we can compute the number of comparison-and-shift encoding for P_1 to P_4 as follows:

$$g_1(P_j)\begin{cases} 1, & \text{for } j = 1 \text{ or } 2 \\ 1+\frac{f(w)}{2^{w-1}-1}, & \text{for } j = 3 \\ w, & \text{for } j = 4 \end{cases}, \text{ where } P_j \text{ is a bit pattern} \tag{5}$$

From Equation 5, the average number of comparison-and-shift encoding for converting one bit of $k_{(2)}$ into k_i is computed into:

$$\begin{aligned} h_1(w) &= \frac{2^{w-2}g_1(P_1)+2^{w-2}g_1(P_2)+(2^{w-1}-1)g_1(P_3)+g_1(P_4)}{w2^w+f(w)} \\ &= \frac{2^{w-1}+2^{w-1}-1+2^w-1}{w2^w+2^w-w-1} = \frac{2(2^w-1)}{(w+1)(2^w-1)} = \frac{2}{w+1} \end{aligned} \tag{6}$$

Thus, when the length of $k_{(2)}$ is l, the SNAF$_w$ algorithm takes $2l/(w+1)$ numbers of comparison-and-shift encoding.

Number of Bit Comparison for Conditional NAF Conversion : From **Algorithm 3**, to check whether the NAF conversion is completed, bit patterns in P_1 take a bit comparison. However, bit patterns in P_2 and P_4 do not take a bit comparison. Since bit patterns in P_3 belong to P_1 or P_2 after one-bit shift to the right side, they takes a bit comparison to check whether the NAF conversion is completed. Since $\Pr(0)$ is equal to $\Pr(1)$, the probabilities that bit patterns in P_3 belong to P_1 or P_2 are 50% respectively. Note that such a bit comparison is taken only when bit patterns in P_3 belong to P_1. As a result, to check whether the NAF conversion is completed, each pattern in P_3 takes 1/2 numbers of bit comparison. That is, the number of bit comparison for each pattern group is given into:

$$g_2(P_j)\begin{cases} 1, & \text{for } j = 1 \\ 0, & \text{for } j = 2 \text{ or } 4 \\ 0.5, & \text{for } j = 3 \end{cases}, \text{ where } P_j \text{ is a bit pattern} \tag{7}$$

From Equation 7, the average number of bit comparison for converting one bit of $k_{(2)}$ into k_i is computed into:

$$\begin{aligned} h_2(w) &= \frac{2^{w-2}g_2(P_1)+2^{w-2}g_2(P_2)+(2^{w-1}-1)g_2(P_3)+g_2(P_4)}{w2^w+f(w)} \\ &= \frac{2^{w-2}+2^{-1}(2^{w-1}-1)}{w2^w+2^w-w-1} = \frac{2^{-1}(2^w-1)}{(w+1)(2^w-1)} = \frac{1}{2(w+1)} \end{aligned} \tag{8}$$

Thus, when the length of $k_{(2)}$ is l, the SNAF$_w$ algorithm checks whether the NAF conversion completes after $l/2(w+1)$ numbers of bit comparison for conditional completion.

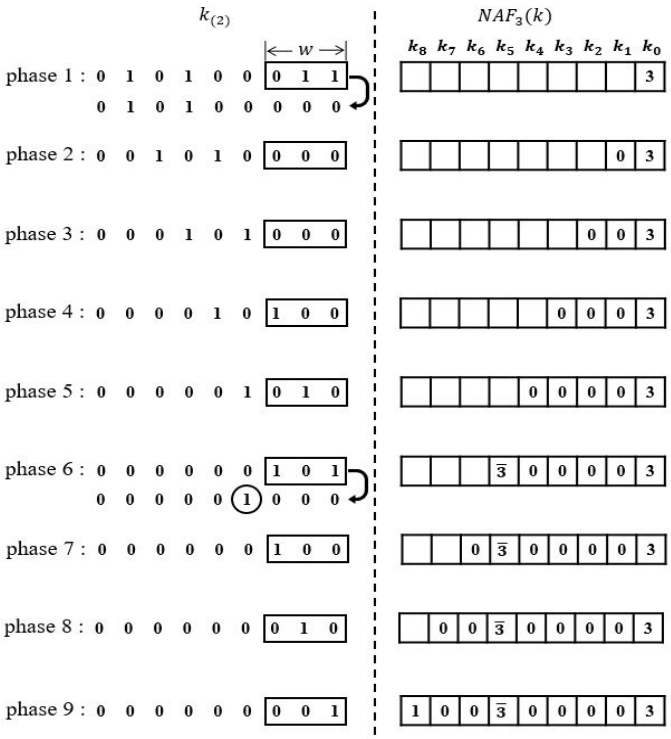

Fig. 3. Example of $\mathbf{NAF_3}$ operation, where the circle 'O' at phase 6 indicates carry

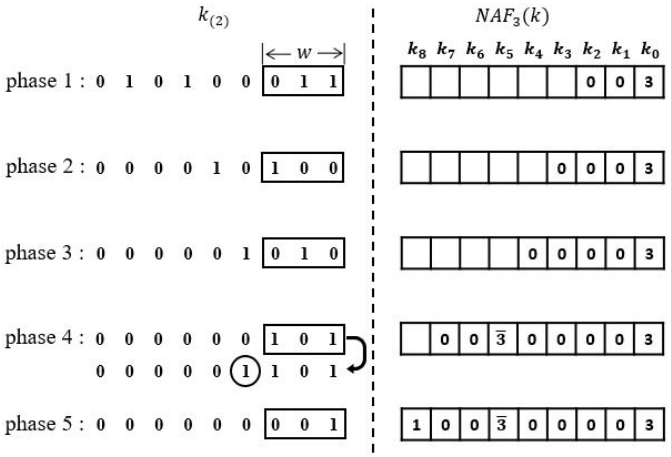

Fig. 4. Example of $\mathbf{SNAF_3}$ operation, where the circle 'O' at phase 4 indicates carry

3.3 Example

By considering a binary representation $(010100011)_2$ for w=3, we show the comparative example of the SNAF_w algorithm and the NAF_w algorithm.

3.3.1 Operational Example of $\mathbf{NAF}_w$: In **Fig. 3**, we observe that to complete the NAF conversion, the NAF_3 algorithm computes $\mathrm{NAF}_3(163)$ after 9 numbers of comparison-and-shift encoding. If LSB in the 3-bit scanning window is 1 as shown in phases 1

and 6, k_0 and k_5 are set into 3 and $\bar{3}$ respectively. After computing k_0 and k_5, w bits within the scanning window are updated into 0 and then, $(010100011)_2$ shifts to the right side by one bit. Also, if LSB in the 3-bit scanning window is 0 as shown in the phases 2 to 5, 7 and 8, k_1 to k_4, k_6 and k_7 are set into 0 respectively. While computing k_1 to k_4, k_6 and k_7, $(010100011)_2$ moves to the right side by one bit respectively. Finally, after k_8 in $(000000001)_2$ is inspected in phase 9, $(010100011)_2$, i.e., k=163, is converted into $(100\bar{3}00003)_{NAF}$.

3.3.2 Operational Example of SNAF$_w$: In **Fig. 4**, we observe that to complete the NAF conversion, the SNAF$_3$ algorithm computes NAF$_w(k)$ after 5 numbers of comparison-and-shift encoding, which is less than the NAF$_3$ algorithm. At phase 1, since x is given into 3, k_0 is set into 3 and k_1 to k_2 are set into 0 and then, $(010100011)_2$ shifts to the right side by w bits. At phases 2 and 3, since LSB within the w-size scanning window are 0s, k_4 to k_3 are set into 0s. In phase 4, x is given into 5, k_5 is set into $\bar{3}$ and k_6 to k_7 are set into 0. In this phase, as x is larger than 2^{3-1}, 2^3 is added to $(101)_2$. That is, since carry occurs, $(000001101)_2$ shifts to the right side by w bits. Finally, after k_8 in $(000000001)_2$ is inspected in phase 5, $(010100011)_2$, i.e., k=163, is converted into $(100\bar{3}00003)_{NAF}$.

4. Performance Evaluation

In this section, we show the performance evaluation results of the SNAF$_w$ algorithm and other NAF conversion algorithms.

4.1 Experimental Environment

To evaluate the performance of the SNAF$_w$ algorithm, we compared the NAF conversion speed of the SNAF$_w$ algorithm with those of the NAF$_w$, MOF$_w$ and scan-NAF algorithms. Our previous research, the scan-NAF algorithm, improves the conversion speed of the NAF$_w$ algorithm through direct assignment instead of calculation for k_is. However, when $w > 2$, the scan-NAF algorithm has a limit that has almost the same performance as the NAF$_w$ algorithm[29].

When comparing the NAF conversion speed, we measured the cycle counter on the low-performance 8-bit microprocessor ATmega128 by using AVR studio 4. This is because AVR studio 4 calculates the cycle counters required to execute assembly instructions regardless of the performance of the microprocessors. Also, it was observed that compared to 16-bit microprocessor and 32-bit processor, the cycle counter difference between the SNAF$_w$ algorithm and the others was minimized in 8-bit ATmeage128. Thus, we measured cycle counters on 8-bit microprocessor ATmega128 by using AVR studio 4.

When converting $k_{(2)}$ into NAF, the performance of the NAF$_w$ and SNAF$_w$ algorithms can vary according to the input patterns, each of which consists of diverse alignment and different numbers of `0' and `1'. This is because the comparison-and-shift encoding for `0' is faster than that for `1'. Thus, we consider the influence of the following input variables on the NAF conversion speed:

• Diverse Patterns : To evaluate the influence of the number of bit 1 on the NAF conversion speed, we generate diverse patterns, each of which consists of different number of bit 0 or bit 1 within bits less than or equal to 3.

• Different Numbers of Repeated Patterns : To evaluate the influence of the length of $k_{(2)}$ on the NAF conversion speed, we change the number of repeated patterns.

• Various Size of w : To evaluate the influence of the scanning width on the NAF conversion speed, we change the value of w.

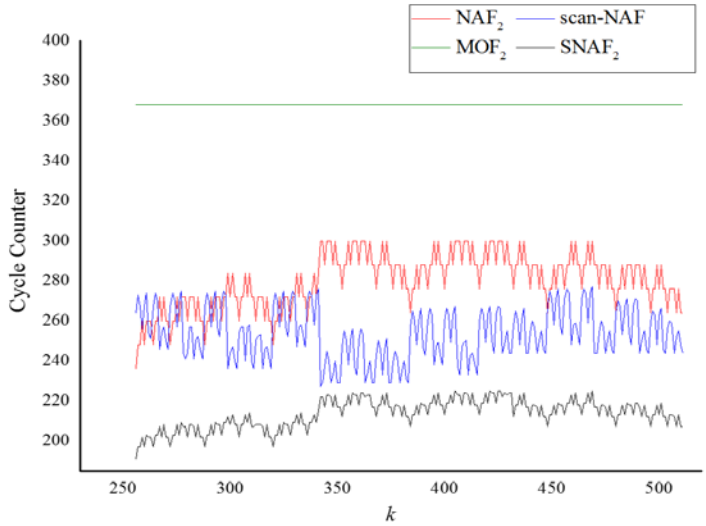

Fig. 5. Cycle counters of $\mathbf{NAF_2}$, $\mathbf{MOF_2}$, **scan-NAF** and $\mathbf{SNAF_2}$ for different $\boldsymbol{k}$s

4.2 $\mathbf{NAF}_w$ *vs.* $\mathbf{MOF}_w$ *vs.* **scan-NAF** *vs.* $\mathbf{SNAF}_w$ $(w = 2)$

For w =2, to measure cycle counters of the NAF_w, MOF_w, scan-NAF and SNAF_w algorithms, we considered all the possible k given from 9 bits, i.e., 256 for $(100000000)_2$ to 511 for $(111111111)_2$. In **Fig. 5**, it is observed that the NAF_2 algorithm takes at least 236 cycle counters, up to 300 cycle counters, and 280 cycle counters on average. The MOF_2 algorithm shows the same cycle counter, i.e., 368, for all ks. The scan-NAF algorithm takes at least 227 cycle counters, up to 277 cycle counters, and 254 cycle counters on average. Compared to the NAF_2, MOF_2 and scan-NAF algorithms, the SNAF_2 algorithm takes at least 191 cycle counters, up to 225 cycle counters, and 213 cycle counters on average. That is, the SNAF_2 algorithm takes the less cycle counter than the MOF_2 algorithm, scan-NAF algorithm and the NAF_2 algorithm for all the possible ks.

In the NAF_2, scan-NAF and SNAF_2 algorithms, as the number of cases in which LSB of scanning window is 1 or 0 varies, the cycle counter for completing the NAF conversion varies. This is because comparison-and-shift encoding time of the NAF_w algorithm, the scan-NAF algorithm and the SNAF_w algorithm is different according to bit 0 and bit 1. However, in the MOF_2 algorithm, the cycle counter for completing the NAF conversion is the same for all ks. This is because the MOF_2 algorithm is designed to do the same comparison-and-shift encoding regardless of bit 0 and bit 1.

Table 2. Average and deviation gain of $\mathbf{SNAF_2}$ algorithm over $\mathbf{NAF_2}$ (unit:%)

	Repeated Patterns							
	1	01	10	001	011	100	101	110
Average gain	21	24.3	18.6	22.3	24.3	21.9	25.3	24.7
Deviation gain	31.3	36.4	31.4	35	36.7	35	37.2	36.7

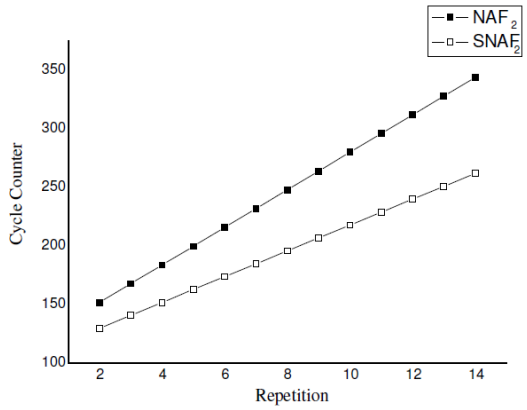

Fig. 6. Cycle counters for repeated pattern $(\mathbf{1})_2$ (w=2)

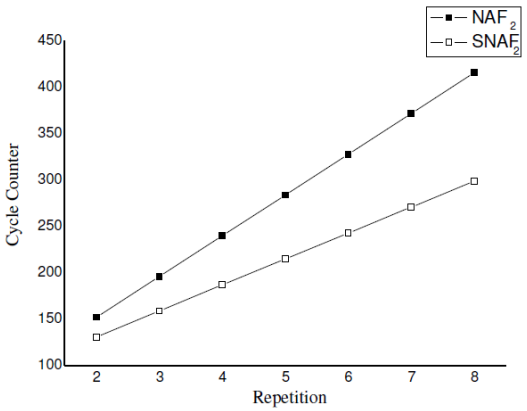

Fig. 7. Cycle counters for repeated pattern $(\mathbf{01})_2$ (w=2)

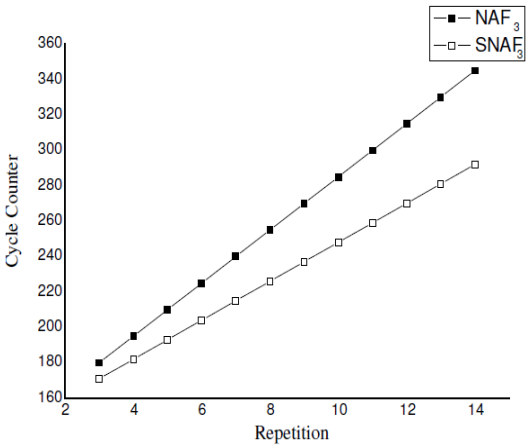

Fig. 8. Cycle counters for repeated pattern $(\mathbf{1})_2$ (w=3)

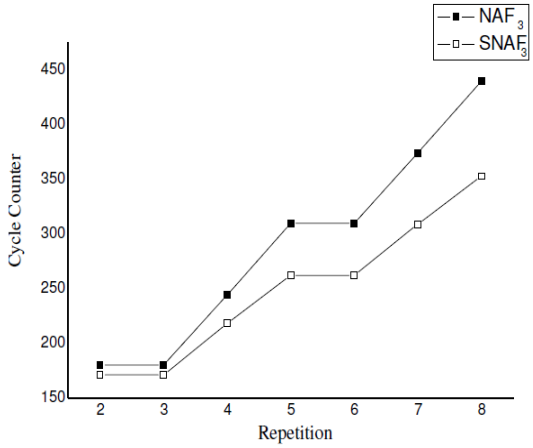

Fig. 9. Cycle counters for repeated pattern $(\mathbf{01})_2$ (w=3)

4.3 $\mathbf{NAF}_w$ $vs.$ $\mathbf{SNAF}_w$ $(w \geq 2)$

We show the measured cycle counter of the NAF_w and SNAF_w algorithms for $w \geq 2$. Since the MOF_w algorithm is designed to operate only for w=2, it is not compared with the other algorithms in this section.

4.3.1 Influence of Diverse Patterns : In this section, by considering all the possible bit patterns which are expressed from w=3 bits, we measured the cycle counters of the NAF_w and SNAF_w algorithms. Note that since we investigate the influence of the repeated bit patterns on the NAF conversion time, we ignore the bit pattern '111'. That is, we evaluated the NAF conversion time by considering diverse patterns with the different ratios of bit 0 and bit 1 in $k_{(2)}$. To consider repetition of the w-size patterns in $k_{(2)}$, we measured cycle counters by varying the value of l from 3 to 15.

In **Table 2**, we summarize the average and deviation gains of the SNAF_2 algorithm over the NAF_2 algorithm. For the repeated patterns whose LSBs are 1 or which include two more 1s, the SNAF_2 algorithm showed the average gain by 24% and the deviation gain by 36% over the NAF_2 algorithm. For the other patterns whose LSBs are 0 or which include a single 1 not in

LSB, the $SNAF_2$ algorithm showed the average gain by 20% and the deviation gain by 31% over the NAF_2 algorithm. In particular, it is observed that the $SNAF_2$ algorithm over the NAF_2 algorithm shows the higher deviation gain than the average gain. This indicates that the $SNAF_w$ algorithm shows the stable NAF conversion speed even when the length l of $k_{(2)}$ increases and various input patterns exist in $k_{(2)}$.

4.3.2 Influence of Number of Repeated Patterns : To investigate the influence of the repeated patterns on the NAF conversion time, cycle counters for the NAF_2 and $SNAF_2$ algorithms were measured under various numbers of repetitions of $(1)_2$ and $(01)_2$ bit patterns. Specifically, cycle counters for the NAF conversion of $k_{(2)}$, where $(1)_2$ pattern repetition is frequently found, are measured to investigate the influence of the repeated patterns whose LSBs of scanning window are 0. Also, cycle counters for the NAF conversion of $k_{(2)}$, where $(01)_2$ pattern repetition is frequently found, are measured to investigate the influence of the repeated patterns whose LSBs of scanning window are 1.

In **Fig. 6** and **Fig. 7**, it is observed that for w=2, the $SNAF_2$ algorithm takes the less cycle counters than the NAF_2 algorithm. That is, the $SNAF_2$ algorithm over the NAF_2 algorithm shows average gain by 21% for $(1)_2$ and 24.3% for $(01)_2$ and deviation gain by 31.3% for $(1)_2$ and 36.4% for $(01)_2$. In **Fig. 8** and **Fig. 9**, it is observed that for w=3, the $SNAF_3$ algorithm also takes the less cycle counters than the NAF_3 algorithm. That is, the $SNAF_3$ algorithm over the NAF_3 algorithm shows average gain by 11.8% for $(1)_2$ and 14.3% for $(01)_2$ and deviation gain by 26.7% for $(1)_2$ and 29.9% for $(01)_2$.

In **Fig. 6** to **Fig. 9**, we observe that cycle counter of the NAF_w algorithm increases more steeply than that of the $SNAF_w$ algorithm. This implies that as the number of repeated patterns increases, the NAF conversion time gap between the NAF_w algorithm and the $SNAF_w$ algorithm steeply increases. Also, from **Fig. 9**, it is observed that when the number of repeated patterns increases from 2 to 3 or from 5 to 6, cycle counter does not vary. This is because the number of bit shifting is the same for these two cases. For example, if the size of w is 3, $(0101)_2$ with two numbers of $(01)_2$ is changed into $(1000)_2$ by carry propagation. After shifting 3 bits to the right side, $(001)_2$ is investigated. Also, $(010101)_2$ with three numbers of $(01)_2$ is changed into $(011000)_2$ by carry propagation. After shifting 3 bits to the right side, $(011)_2$ is investigated.

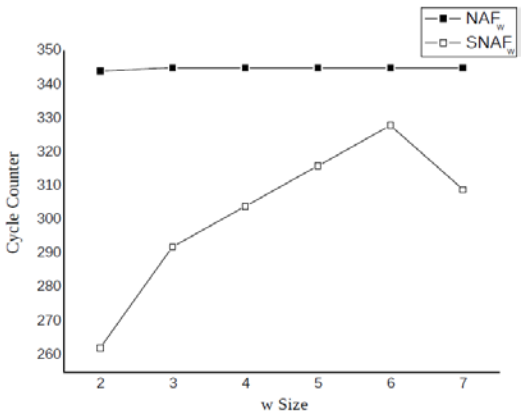

Fig. 10. Cycle counters under the various size of w (k : pattern '1' repeats 14 times)

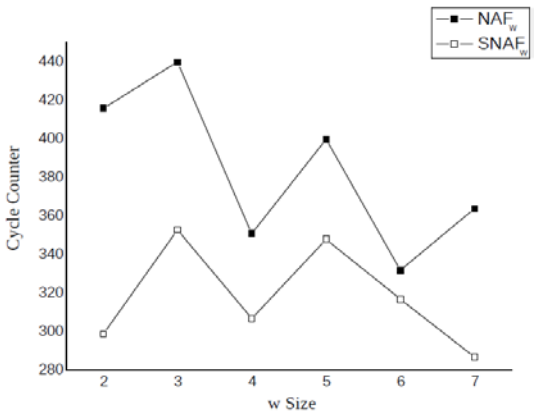

Fig. 11. Cycle counters for the various size of w (k : pattern '01' repeats 8 times)

4.3.3 Influence of Size of w : In **Fig. 10** and **Fig. 11**, we show cycle counters of the NAF_w algorithm and SNAF_w algorithms for completing the NAF conversion when the size of w varies. In **Fig. 10**, we assume that k includes 14 numbers of the pattern '1' and in **Fig. 11**, we assume that k includes 8 numbers of the pattern '01'. From **Fig. 10** and **Fig. 11**, we observe that the SNAF_w algorithm takes the less cycle counter than the NAF_w algorithm regardless of the size of w. Also, it can be evaluated that the performance difference between the NAF_w algorithm and the SNAF_w algorithm is large when the size of w is 2, 3 or 4, which is generally used for the NAF conversion in many applications.

In **Fig. 10**, the NAF_w algorithm takes a constant cycle counter regardless of the size of w. This is because, in the first step of converting $(11111111111111)_2$ to $\text{NAF}_w(k)$, $(11111111111111)_2$ becomes $(100000000000000)_2$. Therefore, the same number of comparison-and-shift encoding are required regardless of the size of w. On the other hand, as the size of w increases, cycle counter of the SNAF_w algorithm gradually increases. This is because the larger the size of w, the more cycle counters take to initialize and delete the data.

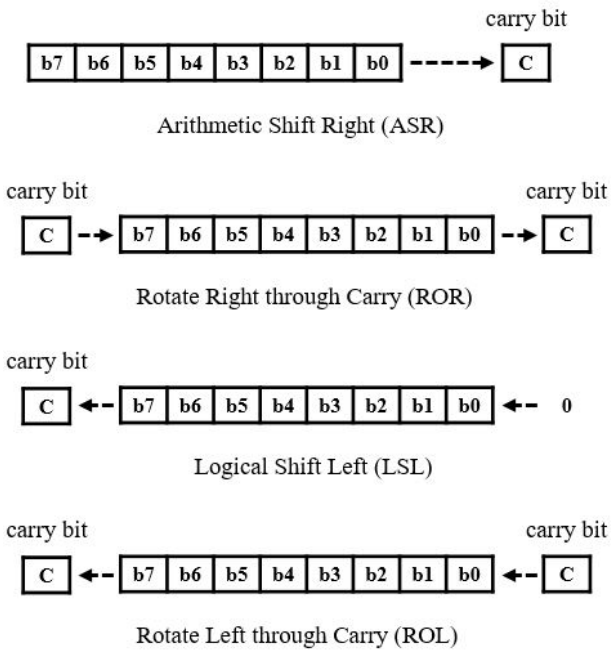

Fig. 12. Instructions used for shift operation

In **Fig. 11**, we observe that as the size of w increases, cycle counter of the NAF_w algorithm decreases. This is because the number of when LSB in w-width scanning window is one decreases as the size of w increases. Also, when w is odd, the NAF_w and SNAF_w algorithms take many cycle counters compared to when w is even. This is because, when w is even, the NAF_w and SNAF_w algorithms do not generate carry while checking w-bit scanning window and thus, do comparison-and-shift encoding for 15 bits equal to $(101010101010101)_2$. However, when w is odd, carry occurs and comparison-and-shift encoding for 16 bits are required. Thus, when w is odd, cycle counter increases.

In **Fig. 10** and **Fig. 11**, cycle counter of the SNAF_w algorithm decreases when the size of w is 7. This is because the register size of ATmega128 is 8 bits. For example, it is assumed that $(00111111\ 11111111)_2$ is stored in the register R19 and R18 by as much as 8 bits

respectively. The $SNAF_w$ algorithm shifts to the right side by the size of w as shown at line 15 in **Algorithm 3**. The 1-bit right-side shift performs the ASR instruction for R19 and then executes the ROR instruction for R18. Thus, if the size of w is 6, repeat such operation 6 times. However, when the size of w is 7, such operation is not repeated. The 7-bit right-side shift in an 8-bit register is equal to discarding all bits other than MSB and shifting MSB to the LSB position. Note that this operation can be changed to a simple 1-bit left-side shift. Thus, the 7-bit right-side shift performs the LSL instruction for R18, copies R19 into R18 and then, performs the ROL instruction for R18. Due to such operations, cycle counter when the size of w is 7 is reduced compared to when the size of w is 6. In **Fig. 12**, we show how the instructions explained in this paragraph work.

5. Conclusion

As a method for integer encoding that minimizes Hamming weight, NAF removes a stream of non-zero bits from the binary representation of an integer. Compared to the other encoding methods, NAF has been used in diverse applications such as public key cryptography, packet filtering, constructing a ternary FCSRs and analysis for medical predictive models. In this paper, we proposed a new NAF conversion algorithm, called $SNAF_w$, which improves the NAF conversion speed of the NAF_w algorithm. The $SNAF_w$ algorithm is designed to skip the unnecessary comparison-and-shift encoding and bit comparison for checking whether the NAF conversion is completed. From the experimental results under various input conditions, the $SNAF_w$ algorithm showed the faster NAF conversion time than the NAF_w algorithm and other NAF conversion algorithms.

Specifically, under diverse bit patterns, the $SNAF_2$ algorithm showed the average gain by 24% and the deviation gain by 36% over the NAF_2 algorithm. Also, under different numbers of repeated patterns, the $SNAF_3$ algorithm over the NAF_3 algorithm showed the average gain by 11.8% to 14.3% and the deviation gain by 26.7% to 29.9%. Also, the $SNAF_w$ algorithm took the less cycle counter than the NAF_w algorithm regardless of the size of w. In summary, the $SNAF_w$ algorithm improves the NAF conversion speed of the current NAF conversion algorithm.

References

[1] A. Booth, "A signed binary multiplication technique," *Journal of Mech. and Applied Math.*, vol. 4, no. 2, pp. 236-240, 1951. Article (CrossRef Link).

[2] Fraenkel, Aviezri S., Klein and Shmuel T, "Robust universal complete codes for transmission and compression," *Discrete Applied Mathematics*, Article (CrossRef Link).

[3] K. Koyama and Y. Tsuruoka, "Speeding Up Elliptic Curve Cryptosystems using a Signed Binary Windows Method," in *Proc. of Crypto 1992. Advances in Cryptology*, pp. 345-357, August 16-20, 1992. Article (CrossRef Link).

[4] J. Adikari, V. S. Dimitrov and K. U. Jarvinen, "A Fast Hardware Architecture for Integer to tauNAF Conversion for Koblitz Curves," *IEEE Transactions on Computers*, vol. 61, no. 5, pp. 732-737, 2012. Article (CrossRef Link).

[5] S. S. Roy, J. Fan and I. Verbauwhede, "Accelerating Scalar Conversion for Koblitz Curve Cryptoprocessors on Hardware Platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vo1. 23, no. 5, pp. 810-818, 2015. Article (CrossRef Link).

[6] N. Meloni and M. A. Hasan, "Random Digit Representation of Integers," in *Proc. of IEEE 23nd Symposium on Computer Arithmetic*, pp. 118-225, July 10-13, 2016. Article (CrossRef Link).

[7] N.-B. Neji and A. Bouhoula, "NAF Conversion: An Efficient Solution for the Range Matching Problem in Packet Filters," *IEEE High Performance Switching and Routing*, pp. 24-29, July 4-6, 2011. Article (CrossRef Link).
[8] N.-B. Neji and A. Bouhoula, "A prefix-based approach for managing hybrid specifications in complex packet filtering," *Computer Networks*, vol. 56, no. 13, pp. 3055-3064, 2012. Article (CrossRef Link).
[9] L.Zhiaiang and P. Dingyi, "Constructing a Ternary FCSR with a Given Connection Integer," Tech. Rep. 2011/358, 2011. Article (CrossRef Link).
[10] P. Dingyi, L.Zhiaiang and X. Zhang, "Construction of Transition Matrices for Ternary Ring Feedback With Carry Shift Registers," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2942-2951, 2015. Article (CrossRef Link).
[11] JH. Cheon, JH. Jeong, JH. Lee and KW. Lee, "Privacy-Preserving Computations of Predictive Medical Models with Minimax Approximation and Non-Adjacent Form," in *Proc. of Springer. WAHC'2017*, April 7, 2017. Article (CrossRef Link).
[12] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120-126, 1978. Article (CrossRef Link).
[13] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Comp*, vol. 48, pp. 203-209, 1987. Article (CrossRef Link).
[14] V.S. Miller, "Use of Elliptic Curves in Cryptography," in *Proc. of Crypto 1985. Advances in Cryptology*, pp. 417-426, 1986. Article (CrossRef Link).
[15] G.W. Reitwiesner, "Binary arithmetic," *Advances in Computers*, vol. 1, pp. 231-308, 1960. Article (CrossRef Link).
[16] IEEE $P_1$363, Standard Specifications for Public-Key Cryptography. Article (CrossRef Link).
[17] H. Prodinger, "On Binary Representations of Integers with Digits {-1,0-1}," *Integers: Electronic Journal of Combinatorial Number Theory*, 2000. Article (CrossRef Link).
[18] D. Hankerson, A. Menezes and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, chapter. 3, 2004.
[19] J. Jedwab and C.J. Mitchell, "Minimum weight modied signed-digit representations and fast exponentiation," *Electronics Letters*, vol. 25, no. 17, pp. 1171-1172, 1989. Article (CrossRef Link).
[20] Yasin, M. Sharifah, M. Ramlan and N.H.N. Rozi, "Performance Analysis of Signed—Digit {0, 1, 3}—NAF Scalar Multiplication Algorithm in Lopez—Dahab Model," *Research Journal of Information Technology*, vol. 7, pp. 80-100, 2015. Article (CrossRef Link).
[21] M. Joye and S.-M. Yen, "Optimal Left-to-Right Binary Signed-digit Exponent Recoding," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 740-748, 2000. Article (CrossRef Link).
[22] M. Joye and T. Christophe, "Compact encoding of non-adjacent forms with applications to elliptic curve cryptography," in *Proc. of 4th Int. Workshop on Public Key Cryptography*, pp. 353-364, February 13-15, 2001. Article (CrossRef Link).
[23] K. Okeya, "Signed binary representations revisited," in *Proc. of Crypto 1992, Advances in Cryptology*, pp. 123-139, August 15-19, 2004. Article (CrossRef Link).
[24] A. Rezai and P. Keshavarzi, "CCS Representation: A New Non-Adjacent Form and its Application in ECC," *Journal of Basic and Applied Scientific Research*, vol. 2, no. 5, pp. 4577-4586, 2012. Article (CrossRef Link).
[25] M. Bafandehkar, S. M. Yasin and R. Mahmod, "Optimizing {0, 1, 3}-NAF Recoding Algorithm Using Block-Method Technique in Elliptic Curve Cryptosystem," *Journal of Computer Science*, vol. 12, no. 11, pp. 534-544, 2016. Article (CrossRef Link).
[26] A. Eghdamian and A. Samsudin, "A Modified Left-to-Right Radix-r Representation," in *Proc. of 2015 International Symposium on Technology Management and Emerging Technologies(ISTMET)*, pp. 254-257, August 25-27, 2015. Article (CrossRef Link).
[27] A. Eghdamian and A. Samsudin, "MGSDNAF-A Modified Signed Digit Generalized Non-Adjacent Form for Integers Representation," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, vol. 9, no. 2-4, pp. 11-13, 2017. Article (CrossRef Link).

[28] C. Fan, Y. Niu, G. Shi, F. Li, X. Xie and D. Jiao, "An Improved Signed Digit Representation Approach for Constant Vector Multiplication," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, no. 10, pp. 999-1003, 2016. Article (CrossRef Link).
[29] D. H. Hwang, J. M. Shin and Y. H. Choi, "Fast Non-Adjacent Form (NAF) Conversion through a Bit-Stream Scan," *Journal of KIISE*, vol. 44, no. 5, pp. 537-544, 2017. Article (CrossRef Link).

Doo-Hee Hwang received his B.S. degrees from the School of Computer Science and Engineering, Pusan National University, Busan, Korea, in Feb. 2017. His research interests include IoT security, blockchain application, and preservation of privacy.

Yoon-Ho Choi is an associate professor at the School of Computer Science and Engineering in Pusan National University, Busan, Korea. He received his M.S. and Ph.D. degrees from the School of Electrical and Computer Engineering, Seoul National University, S. Korea, in Aug. 2004 and Aug. 2008, respectively. He was a postdoctoral scholar at Seoul National University, Seoul, S. Korea from Sep. 2008 to Dec. 2008 and in Pennsylvania State University, University Park, PA, USA from Jan. 2009 to Dec. 2009. He worked as a senior engineer at Samsung Electronics from May 2010 to Feb. 2012. He also worked as an assistant professor at the Department of Convergence Security in Kyonggi University, Suwon, Korea, from May 2012 to Aug. 2014. He has served as a TPC member in various international conferences and journals. His research interests include Deep Packet Inspection (DPI) for high-speed intrusion prevention, mobile computing security, IoT security, vehicular network security for realizing secure things and networks.