JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.17, NO.3, JUNE, 2017
https://doi.org/10.5573/JSTS.2017.17.3.425
ISSN(Print) 1598-1657
ISSN(Online) 2233-4866

Efficient Design and Performance Analysis of a Hardware Right-shift Binary Modular Inversion Algorithm in GF(p)

Piljoo Choi[1], Mun-Kyu Lee[2], Jeong-Taek Kong[3], and Dong Kyue Kim[1]

***Abstract*—For efficient hardware (HW) implementation of elliptic curve cryptography (ECC), various sub-modules for the underlying finite field operations should be implemented efficiently. Among these sub-modules, modular inversion (MI) requires the most computation; therefore, its performance might be a dominant factor of the overall performance of an ECC module. To determine the most efficient MI algorithm for an HW ECC module, we implement various classes of MI algorithms and analyze their performance. In contrast to the common belief in previous research, our results show that the right-shift binary inversion (RS) algorithm performs well when implemented in hardware. In addition, we present optimization methods to reduce the area overhead and improve the speed of the RS algorithm. By applying these methods, we propose a new RS-variant that is both fast and compact. The proposed MI module is more than twice as fast as the other two classes of MI: shifting Euclidean (SE) and left-shift binary inversion (LS) algorithms. It consumes only 15% more area and even 5% less area than SE and LS, respectively. Finally, we show that how our new method can be applied to optimize an HW ECC module.**

Manuscript received Oct. 6, 2016; accepted Dec. 25, 2016
[1] Dept. of Electronic Engineering, Hanyang University, 222 Wangsimni-ro, Seongdong-gu, Seoul 04763, Korea
[2] Dept. of Computer Engineering, Inha University, 100 Inha-ro, Nam-gu, Incheon 22212, Korea
[3] Software College at Sungkyunkwan University, 2066 Seobu-ro, Jangan-gu, Suwon-si, Gyeonggi-do 16419, Korea
E-mail : dqkim@hanyang.ac.kr
Preliminary versions of this paper appeared in MITA 2015 [1] and ISOCC 2015 [2].

***Index Terms*—Elliptic curve cryptosystem (ECC), modular inversion, modular division, computation over finite field, right-shift binary inversion**

I. INTRODUCTION

Public-key cryptographic algorithms such as RSA and elliptic curve cryptography (ECC) are essential for current security systems. Since ECC can provide a similar security level to RSA with much shorter key size [3], implementation of ECC is advantageous in terms of speed and area. Even though shorter keys are used for ECC, the keys are still very long, and this requires significant computation. Thus, there has been extensive research [4-9] on ECC hardware (HW) implementation, where data can be processed in parallel, as well as software (SW) implementation, where data are processed word by word.

To implement an HW ECC module, sub-modules for the underlying finite field operations such as modular multiplication (MM), modular inversion (MI), and modular addition (MA) are necessary. Among the sub-modules, MI requires the most computation; therefore, its performance has a significant effect on the overall performance of an ECC module. Although an MI can be substituted with several MMs using projective coordinates [10, pp. 86-92], this method causes significant area overhead. Moreover, even in this case, an MI module is still necessary since an MI is required to transform projective coordinates back to affine coordinates in the last stage and is necessary for generating and verifying digital signatures in elliptic curve digital signature algorithm (ECDSA) [11]. Therefore, an efficient MI module enables a smaller and faster ECC module and even may get rid of the need for projective coordinates if MI is sufficiently fast.

In this paper, we propose an efficient design of an HW MI module. In the literature, MI algorithms based on the greatest common divisor (GCD) are widely used. We remark that although there is another class of inversion algorithm using Fermat's little theorem that performs $a^{-1} \equiv a^{m-2}$ (mod m) for prime modulus m, this method is not considered in this paper since it just repeats a lot of MMs and consumes a lot of time. There are three classes of MI algorithms using GCD in the literature: Extended Euclidean inversion (EE) algorithm [12], right-shift binary inversion (RS) algorithm [13-15], and left-shift binary inversion (LS) algorithm [16, 17]. EE requires divisions, while RS and LS are based on right-alignment and left-alignment, respectively. LS was proposed later than the others, and it has been believed that its performance is the best among the three classes [16, 18]. However, this optimal performance is confined only to SW implementation. HW implementation has features differentiating it from SW implementation: no cost for shift operations, parallel processing, and limited storage. In this paper, we demonstrate that RS is more appropriate than EE and LS for HW implementation and propose a modified RS algorithm with faster speed and smaller area. Using the modified RS HW module, we also show that a smaller and faster ECC HW module is achievable.

II. Preliminaries

In this section, we explain the three classes of MI algorithms in detail and analyze their speed and area in HW implementation. The three algorithms (Algorithm 1-3) shown in this section are similar to the ones in [18].

1. Three Classes of MI Algorithms

To calculate a^{-1} (mod m), we consider the equation $ax + my = \gcd(a, m)$, where $\gcd(a, m)$ is the greatest common divisor of a and m. If $\gcd(a, m) = 1$, then $ax \equiv 1$ (mod m), so $x \equiv a^{-1}$ (mod m). We determine x using the following equations:

$$aR \equiv U \pmod{m}, \tag{1}$$

and

$$aS \equiv V \pmod{m}. \tag{2}$$

The initial values of R, S, U, and V are 0, 1, m, and a, respectively. Through an operation involving (1) and (2), we obtain a third equation, $aR' \equiv U'$ (mod m); for example, if the operation is subtraction, $R' = R - S$, and $U' = U - V$. Our objective is to reduce the size of U and V by repeating these operations until obtaining $x \equiv a^{-1}$ (mod m). All MI algorithms based on this method have a main loop that maintains (1) and (2) as loop invariants in order to repeat the reduction operations. The main difference between the MI algorithms is their reduction operations. In this section, the three classes of MI algorithms with different reduction operations are explained.

A. Extended Euclidean (EE) Inversion Algorithm

In EE, (1) – $k \times$ (2) is performed for $U > V$, where $k = \lfloor U/V \rfloor$, using the property $\gcd(U, V) = \gcd(V, U \bmod V)$ for $U > V$. The results can be expressed as $U' = U - kV$ and $R' = R - kS$. To calculate k, division is necessary, which is quite complicated. Since division consists of

Algorithm 1: Shifting Euclidean algorithm

```
Input: modulus $m$, divisor $a$.
Output: $a^{-1}$ (mod $m$).
1:  $U \leftarrow m$; $V \leftarrow a$; $R \leftarrow 0$; $S \leftarrow 1$;
2:  while ( $\|V\| > 1$) {     // the bit length of $V$
3:     $f \leftarrow \|U\| - \|V\|$; // the bit length difference
4:     if (sign($U$) = sign($V$)){
5:        $U \leftarrow U - (V \ll f)$;
6:        $R \leftarrow R - (S \ll f)$;
7:     }
8:     else{
9:        $U \leftarrow U + (V \ll f)$;
10:       $R \leftarrow R + (S \ll f)$;
11:    }
12:    if ( $\|U\| < \|V\|$ ) {
13:       $U \leftrightarrow V$;
14:       $R \leftrightarrow S$;
15:    }
16: }
17: if ($V = 0$) return 0;
18: if ($V < 0$) $S \leftarrow -S$
19: if ($S > m$) return $S - m$;
20: if ($S < 0$) return $S + m$;
```

subtractions and shifts, EE can be modified as shown in Algorithm 1.

Algorithm 1 is the modified EE. This shifting Euclidean (SE) algorithm is simpler to implement as an HW module than EE since there is no division involved.

B. Right-shift (RS) Binary Inversion Algorithm

In RS, subtraction is used as a reduction operation; in addition, the property gcd(U, V) = gcd(U/2, V) for even U and odd V is also used. There is no case in which both U and V are even; $U' = U/2$ and $R' = R/2$ are performed for even U, and $V' = V/2$ and $S' = S/2$ are performed for even V. The RS algorithm based on subtraction and halving (right-shift) is shown in Algorithm 2.

U_i is the (i+1)-th bit of U, where U_0 is the least significant bit (LSB). If either U_0 or V_0 is zero, U and R, or V and S are right-shifted until both U_0 and V_0 are one. Then subtraction is performed. These shifts and subtractions are repeated until V = 0. One thing to consider is that, when an even U is right-shifted, R can be odd. To halve an odd R without data loss, modulus m is added before the right-shift, which is also necessary when halving odd S. Due to this, RS is only usable for odd moduli and requires more additions than the other algorithms.

Algorithm 2: Right-shift binary inversion algorithm

```
Input: modulus $m$, divisor $a$.
Output: $a^{-1}$ (mod $m$).
1:  $U \leftarrow m$; $V \leftarrow a$; $R \leftarrow 0$; $S \leftarrow 1$;
2:  while ($V > 0$) {
3:      if ($U_0 = 0$) {                    // even $U$
4:          $U \leftarrow U/2$;                          // halving $U$
5:          if ($R_0 = 0$)   $R \leftarrow R/2$;         // halving even $R$
6:          else          $R \leftarrow (R + m)/2$;   // halving odd $R$
7:      }
8:      else if ($V_0 = 0$) {              // even $V$
9:          $V \leftarrow V/2$;                          // halving $V$
10:         if ($S_0 = 0$)   $S \leftarrow S/2$;         // halving even $S$
11:         else          $S \leftarrow (S + m)/2$;   // halving odd $S$
12:     }
13:     else                              // odd $U$ and $V$
14:         if ($U > V$){
15:             $U \leftarrow U - V$;
16:             $R \leftarrow R - S$;
17:             if ($R < 0$) $R \leftarrow R + m$;
18:         }
19:         else {
20:             $V \leftarrow V - U$;
21:             $S \leftarrow S - R$;
22:             If ($S < 0$) $S \leftarrow S + m$;
23:         }
24: }
25: if ($U > 1$)  return 0;
26: if ($R > m$)  $R \leftarrow R - m$;
27: if ($R < 0$)  $R \leftarrow R + m$;
28: return $R$;
```

C. Left-shift (LS) Binary Inversion Algorithm

In RS, after both U and V are right-aligned, subtraction is performed; while in LS, after both U and V are left-aligned, subtraction or addition is performed. Due to left-alignment, the loop invariants are also a bit different from (1) and (2). The loop invariants for LS are

$$aR\cdot 2^{\min(u,\,v)} \equiv U \pmod{m}, \tag{3}$$

Algorithm 3: Left-shift binary inversion algorithm

```
Input: modulus $m$, divisor $a$.
Output: $a^{-1}$ (mod $m$).
1:  $U \leftarrow m$; $V \leftarrow a$; $R \leftarrow 0$; $S \leftarrow 1$; $u \leftarrow 0$; $v \leftarrow 0$;
2:  while ($U \neq \pm 2^u$ && $V \neq \pm 2^v$) {
3:      if ($|U| < 2^{n-1}$) { // $|U|$ is the absolute value of $U$
4:          $U \leftarrow 2U$;   $u \leftarrow u + 1$;
5:          if ($u > v$)    $R \leftarrow 2R$;
6:          else         $S \leftarrow S/2$;
7:      }
8:      else if ($|V| < 2^{n-1}$) {
9:          $V \leftarrow 2V$;   $v \leftarrow v + 1$;
10:         if ($v > u$)    $S \leftarrow 2S$;
11:         else         $R \leftarrow R/2$;
12:     }
13:     else
14:         if (sign ($U$) = sign ($V$))
15:             if ($u \leq v$)  { $U \leftarrow U - V$;   $R \leftarrow R - S$;  }
16:             else       { $V \leftarrow V - U$;   $S \leftarrow S - R$;  }
17:         else
18:             if ($u \leq v$)  { $U \leftarrow U + V$;   $R \leftarrow R + S$;  }
19:             else       { $V \leftarrow V + U$;   $S \leftarrow S + R$;  }
20:     if ($U = 0$ || $V = 0$) return 0;
21: }
22: if ($V = \pm 2^v$) { $R \leftarrow S$; $U \leftarrow V$; }
23: if ($U < 0$)
24:     if ($R < 0$)   $R \leftarrow -R$;
25:     else        $R \leftarrow m - R$;
26: if ($R < 0$) $R \leftarrow R + m$;
27: return $R$;
```

and

$$aS\cdot 2^{\min(u,\,v)} \equiv V \,(\mathrm{mod}\; m). \qquad (4)$$

When $U = \pm 2^u$ and $u \le v$, or $V = \pm 2^v$ and $v \le u$, the loop is terminated, and the inverse elements are $\pm R$ or $\pm S$, respectively. The detail of LS is shown in Algorithm 3, where n is the length of modulus m, and u and v are the accumulated numbers of left-shifts for U and V, respectively.

While the absolute value of U or V is smaller than 2^{n-1}, U or V is left-shifted, respectively. Then the result of $U + V$, $U - V$, or $V - U$ is used in place of the longer of U and V. These shifts, additions and subtractions are repeated until one of the values left-aligned in U and V becomes ± 1.

2. Speed and Area Analysis of MI Algorithms in HW Implementation

To find out which modular inverse algorithm is the most appropriate for HW implementation, we designed HW MI modules. For fair comparison under the same condition, their basic structures were designed using the same techniques. We explain these structures at first, and then analyze the speed and area of HW MI modules.

A. Basic Structures of HW MI Modules

Algorithm 1-3 spend most time in the main loop, and the main operations in the loop are shift and subtraction (or addition) between U and V, and between R and S. In addition to these operations, RS has an addition of modulus m to odd R or S for halving. We note that since subtraction and addition can be done by an adder, we do not distinguish them when analyzing speed and area.

How to design the structures to process shifts and subtractions affects the speed and area of HW MI modules. If we use only one adder due to confined area, the resulting algorithm will not have significantly different characteristics from SW environment. To fully utilize the merits of HW implementation such as capability of parallel processing, we design the structures of HW MI modules with two desirable features as follows.

The first feature is that a shift and a subtraction in the loop is performed within a single clock cycle to minimize the number of required clock cycles. This does not lengthen the critical path since shift is just shifted wire connection in HW implementation. Note that shift is performed not only right after subtraction, but also by itself. For example, in RS, when both U and V are odd, the difference between them must be even, so a shift is continuously performed. Depending on whether the shift result is odd or even, either another subtraction-and-shift or only shift is performed.

The second feature is that enough adders are used to avoid degradation such as increase of required clock cycles or drop of clock frequency [13, 14]. As a result, at least two adders are required to calculate subtraction between U and V, and between R and S. These adders can be also used to perform subtractions or additions after the loop is finished.

B. Speed Analysis of MI Algorithms

In this subsection, we first explain the relationship between speed and the number of shifts, and then analyze the number of shifts in each algorithm.

1) The Relationship between Speed and the Number of Shifts

The speed is inversely proportional to the processing time, which can be expressed as follows:

$$\text{Processing time} = N_{\mathrm{clk}} / f_{\max} = N_{\mathrm{clk}} \times T_{\mathrm{critical\_path}}$$

where N_{clk}, $f_{\max}$, and $T_{\mathrm{critical\_path}}$ are the number of required clock cycles, the maximum achievable clock frequency, and the time length of critical path, respectively. The lengths of the critical paths in all MI modules are minimized using enough number of adders. As a result, the lengths may become approximately equal to the path length of n-bit addition where n is the length of modulus m. Only several multiplexers are added to the inputs or outputs of each n-bit adder.

If the lengths of critical paths are almost the same, the processing time of MI modules is proportional only to the number of required clock cycles. Due to the first feature mentioned in subsection 2.A, either subtraction-and-shift or shift is always performed every clock cycle in the main loop. The number of shifts is equal to the number of clock cycles consumed in the loop; moreover, most clock cycles are consumed in the loop. Hence, the

Table 1. Simulated Results of Three MI Algorithms

Algorithm	The number of shifts in the loop			
	64-bit	128-bit	256-bit	*n*-bit
SE	128.2	256.3	512.2	2.001-2.003*n*
LS	122.7	250.7	506.6	1.917-1.979*n*
RS	88.1	178.4	359.2	1.377-1.403*n*

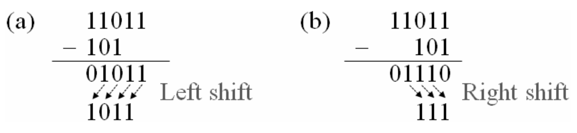


Fig. 1. Subtraction comparison (a) LS, (b) RS.

number of shifts represents the processing time of each MI module.

2) Number of Shifts in Each Algorithm

We can roughly estimate that the expected number of shifts for all MI modules is approximately $2n$. This is because U and V are initialized with n-bit modulus m and divisor a, respectively, either U or V is shortened by one bit due to shift in every clock cycle during the loop, and the loop is finished when both U and V become almost zero.

Table 1 shows the average numbers of shifts for 100000 random samples depending on the length of modulus m: 64, 128, 256, or n bits, which are counted using a Verilog simulator. The numbers of shifts in SE and LS are almost the same because SE repeats left-shifts and subtractions similar to how they are repeated in LS. However, the number of shifts in RS is much smaller. The reason is explained as follows.

Fig. 1 shows the subtraction and shift in LS and RS. In LS, after subtraction, the most significant bit (MSB) always becomes zero, and the LSB becomes zero with a very low probability. In RS, after subtraction, the LSB always becomes zero, and the MSB also becomes zero with a high probability due to carry propagation of subtraction. This means that either U or V is almost always shortened by one bit every clock cycle in SE and LS. Consequentially, almost $2n$ clock cycles are necessary for shifts in SE and LS. In contrast, the number of shifts in RS is much smaller. The exact number of shifts is analyzed in Section IV.

C. Area Analysis of MI Algorithms in HW Implementation

Since variables used in all MI algorithms are almost the same, the sizes of required registers are also similar. Except registers, the number of n-bit adders has significant effect on area. LS and SE need only two n-bit adders, while RS needs six n-bit adders. The reason for this difference is as follows.

In RS, selection of $U - V$ and $V - U$ is determined by the boolean result of the statement, $U > V$. After $V - U$ is performed, if $V - U < 0$, then either $U - V$ or $-(V - U)$ needs to be calculated. This requires another adder or another clock cycle. In order not to increase the number of required clock cycles, two adders for $U - V$ and $V - U$ are necessary.

Addition of modulus m to odd R or S for halving requires more adders as well. Before shifting R, either $R + m$, or $R - S + m$ is calculated. To calculate these without extension of critical path and increase of required clock cycles, a normal adder and a carry save adder (CSA) are required. A CSA transforms a sum of three numbers into a sum of two numbers within the time for 1-bit addition, which is negligible compared to n-bit addition. Selection between $R - S$ and $S - R$ is also determined by the boolean result of the statement, $U > V$, so another pair of an adder and a CSA is required to calculate $S + m$ or $S - R + m$.

Due to the above reasons, RS requires six adders in total including two CSAs. This is a big area overhead compared to SE and LS, but the number of adders in RS can be reduced to three by a simple modification explained in Section III.

There is one more thing to consider for area besides adders. In line 2 of Algorithm 3, the result of $U = 2^u$ can be simply known by checking if only the bits of U except the (u+1)-th LSB of U are zero, but the HW logic for $U \neq -2^u$ is complicated. This can be known by checking if $U + 2^u \neq 0$ or by comparing it with the register that has -2^u. These methods require an n-bit adder or an n-bit register, which increases the area. As a result, the area difference between the LS and RS algorithms may not be significant. The actual area comparison is covered in Section V.

III. Improvement on the RS Algorithm

As explained in the previous section, RS is faster than other methods. Although RS has area overhead, this problem can be solved, and its speed can be improved, as explained in this section.

Table 2. RS vs. RS Variants (When both *U* and *V* are Odd and $U > V$)

RS (Algorithm 2)	RS-alw2 in [18, p. 6]	RS-sel2 in [13]

$U \leftarrow U - V$;
$R \leftarrow R - S$;
if $(R < 0)$
  $R \leftarrow R + m$;
…
$U \leftarrow U/2$;
if $(R_0 = 1)$
  $R \leftarrow R + m$;
$R \leftarrow R/2$;
	if $(U_1 = V_1)${
	  $U \leftarrow U - V$;
	  $R \leftarrow R - S$;
	}
	else {
	  $U \leftarrow U + V$;
	  $R \leftarrow R + S$;
	}
	…
	$U \leftarrow U/4$;
	if $(R_0 = 1)$
	  if $(R_1 = m_1)$
	    $R \leftarrow R - m$;
	  else
	    $R \leftarrow R + m$;
	  else if $(R_1 = 1)$
	    $R \leftarrow R + 2m$;
	$R \leftarrow R/4$;
		$U \leftarrow U - V$;
		$R \leftarrow R - S$;
		…
		if $(U_1 = 1)$ {
		  $U \leftarrow U/2$;
		  if $(R_0 = 1)$
		    $R \leftarrow R + m$;
		  $R \leftarrow R/2$;
		}
		else {
		  $U \leftarrow U/4$;
		  if $(R_0 = 1)$
		    if $(R_1 = m_1)$
		      $R \leftarrow R + 3m$;
		    else
		      $R \leftarrow R + m$;
		  else if $(R_1 = 1)$
		    $R \leftarrow R + 2m$;
		  $R \leftarrow R/4$;
		}

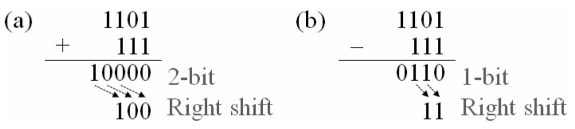


Fig. 2. Operation comparison (a) RS-alw2, (b) RS.

1. Previous RS Variants

There are RS variants to reduce the number of clock cycles consumed in the loop. Their main differences compared to RS are shown in Table 2. In [18, p. 6], the RS variant always performs 2-bit right-shift soon after subtraction or addition; we call this RS-alw2. In [13], the RS variant selectively chooses the 2-bit right-shift after subtraction; we call this RS-sel2. As shown in Table 2, in RS-sel2, 2-bit right-shift can only be done when the second LSB of *U* is also zero. In RS-alw2, by $U + V$, the second LSB of the subtraction or addition result is also always zero, so 2-bit right-shift is always performed after subtraction or addition. However, this is not a good idea. The strength of the RS algorithm is that the subtraction result is sometimes shortened due to carry propagation. The addition result cannot be shortened, and is sometimes prolonged. An example of such a case is shown in Fig. 2. For the same values, addition is selected in RS-alw2; while subtraction is selected in RS. Even though 2-bit is right-shifted after addition in RS-alw2, its result is longer than that of RS. This means that when the second LSB of $U - V$ or $V - U$ is not zero, subtraction and 1-bit shift is better than addition and 2-bit shift. Because of this reason, RS-alw2 may be slower than RS-sel2. The area of RS-alw2 may be bigger than that of RS-sel2 as well. Compared to RS, RS-sel2 needs additional logic only for 2-bit shift, while RS-alw2 has additional 2-bit shifts as well as two additions, $U + V$ and $R + S$.

Table 3. Expression Differences between RS and RS-A

Variable	Previous expression	Modified expression
U, V	$U > V$ $U \leftarrow U - V$ $V \leftarrow V - U$	$V + (-U) < 0$ $(-U) \leftarrow V + (-U)$ $V \leftarrow V + (-U)$
R, S	$R \leftarrow R - S$ $S \leftarrow S - R$	$(-R) \leftarrow S + (-R)$ $S \leftarrow S + (-R)$

2. Proposed RS Variant with Area-improvement (RS-A)

As explained in Section II-2.C, Algorithm 2 needs six adders, but this number can be reduced by storing $-U$ and $-R$ instead of *U* and *R* for the variables *U* and *R*, respectively. This modification is shown in Table 3. In the modified expression, there are no subtractions, but only additions. As a result, the algorithm becomes simpler with only three adders including a CSA, which may reduce the area. We call this RS with area-improvement (RS-A). We note that the speed of RS and RS-A is the same, and that only their areas are different.

In RS-A, the variable *U* is initialized with $-m$ instead

Algorithm 4: RS-A algorithm

Input: modulus *m*, divisor *a*.
Output: a^{-1} (mod *m*).

```
1:  $U \leftarrow -m$; $V \leftarrow a$; $R \leftarrow 0$; $S \leftarrow 1$;
2:  while $(V \neq 1)$ {
3:      if $(V_0 = 1)$         { $TU \leftarrow U$;   $TR \leftarrow R$;  }
4:      else                   { $TU \leftarrow 0$;   $TR \leftarrow 0$;  }
5:      if $(U_0 = 1)$         { $TV \leftarrow V$;   $TS \leftarrow S$;  }
6:      else                   { $TV \leftarrow 0$;   $TS \leftarrow 0$;  }
7:      $VU \leftarrow (TU + TV)/2$;
8:      if $(TR_0$ ^ $TS_0 = 0)$   $SR \leftarrow (TS + TR)/2$;
9:      else if $(TS < 0)$     $SR \leftarrow (TS + TR + m)/2$;
10:     else                   $SR \leftarrow (TS + TR - m)/2$;
11:     if $(VU < 0)$          { $U \leftarrow VU$;   $R \leftarrow SR$;  }
12:     else                   { $V \leftarrow VU$;   $S \leftarrow SR$;  }
13: }
14: if $(V = 0)$ return 0;
15: if $(S > m)$ $S \leftarrow S - m$;
16: if $(S < 0)$  $S \leftarrow S + m$;
17: return $S$;
```

of m; $-m$ is obtained by inverting m except for the LSB of m without any additional adder, since m is odd in RS. As for the variable R, its initial value is zero, requiring no change. The detailed RS-A algorithm is shown in Algorithm 4.

In Algorithm 4, resource sharing is maximized for HW implementation. In line 1, $-m$ instead of m is stored in the variable U. In the loop, one of U, V, and $U + V$ is right-shifted and stored in the variable VU. Depending on the sign of VU, VU is stored in one of U and V. Similarly, R and S are calculated, but they are added or subtracted by m in lines 9 to 10 in order to halve odd R or S. When m is added to positive R and S, the data in the variable SR may overflow. This problem is prevented by subtracting m instead of adding m when the variable TS, where S or zero is stored, is not negative in line 10.

Algorithm 5: RS-AS algorithm

```
Input: modulus $m$, divisor $a$.
Output: $a^{-1}$ (mod $m$).
1:  $U \leftarrow -m$; $V \leftarrow a$; $R \leftarrow 0$; $S \leftarrow 1$;
2:  while ($V \neq 1$) {
3:      if ($V_0 = 1$)              { $TU \leftarrow U$;  $TR \leftarrow R$; }
4:      else                        { $TU \leftarrow 0$;  $TR \leftarrow 0$; }
5:      if ($U_0 = 1$)              { $TV \leftarrow V$;  $TS \leftarrow S$; }
6:      else                        { $TV \leftarrow 0$;  $TS \leftarrow 0$; }
7:      $VU \leftarrow TU + TV$;
8:      $TSR \leftarrow TR_{1:0} + TS_{1:0}$; // $X_{1:0}$ is the two LSBs of $X$
9:      if ($VU_1 = 0$){
10:         $VU \leftarrow VU/4$;
11:         if ($TSR_0 = 0$)
12:             if ($TSR_1 = 1$)    $SR \leftarrow (TR + TS + 2m) / 4$;
13:             else                $SR \leftarrow (TR + TS) / 4$;
14:         else
15:             if ($TSR_1 = m_1$)  $SR \leftarrow (TR + TS - m) / 4$;
16:             else                $SR \leftarrow (TR + TS + m) / 4$;
17:     }
18:     else {
19:         $VU \leftarrow VU/2$;
20:         if ($TSR_0 = 0$)        $SR \leftarrow (TS + TR)/2$;
21:         else if ($TS < 0$)      $SR \leftarrow (TS + TR + m)/2$;
22:         else                    $SR \leftarrow (TS + TR - m)/2$;
23:     }
24:     if ($VU < 0$)               { $U \leftarrow VU$;  $R \leftarrow SR$; }
25:     else                        { $V \leftarrow VU$;  $S \leftarrow SR$; }
26: }
27: if ($V = 0$) return 0;
28: if ($S > m$) $S \leftarrow S - m$;
29: if ($S < 0$) $S \leftarrow S + m$;
30: return $S$;
```

3. Proposed RS Variant with Area- and Speed-Improvement (RS-AS)

RS-sel2, explained in Section III-1, is almost the same as RS except for the codes for the 2-bit shift. Using the idea of RS-sel2, we improve the speed of RS-A. The RS-A with speed-improvement (RS-AS) is shown in Algorithm 5.

Different from RS-sel2 [13], $-m$ is used instead of $3m$ in line 15 since $+3m$ needs two additions, one each for $+m$ and $+2m$, respectively. Compared to RS-A, the codes in lines 9 to 17 in Algorithm 5 are added for 2-bit shift. When the second LSB of $U + V$ is also zero, 2-bit shift is performed instead of 1-bit shift. Except for the codes in lines 9 to 17 in Algorithm 5, RS-AS is the same as RS-A. Therefore, RS-AS needs only three adders as well.

IV. Speed Analysis of the Proposed RS Variants

As explained above, the speed of three algorithms can be roughly compared only using the number of shifts in the loop. Table 4 shows the average number of shifts in RS, RS-A, RS-AS, and RS-alw2 obtained from 100000 random samples with inverse elements. In this section, we mathematically analyze the numbers in Table 4. RS with speed-improvement (RS-S) is the same algorithm as Algorithm 5 (RS-AS) except the area-improvement method. In RS-A and RS-AS, $-U$ and $-R$ instead of U and R are stored for the variables U and R, respectively. Since this may confuse the analysis, we analyze the speed of RS and RS-S instead of RS-A and RS-AS, respectively. Note that the speed of RS and RS-S is the same, and that the speed of RS-A and RS-AS is the same.

1. Number of Shifts in RS

We build a transition model of (U, V) in the loop as follows:

$$(U^1, V^1) \rightarrow (U^2, V^2) \rightarrow \ldots \rightarrow (U^{k-1}, V^{k-1}) \rightarrow (U^k, V^k).$$

The initial value (U^1, V^1) is (m, a) where m is a modulus and a is a divisor. The value on exit (U^k, V^k) is $(1, 0)$ when a^{-1} (mod m) exists. The relationship between

Table 4. Simulated Results of RS Variants

Algorithm	The number of shifts in the loop			
	64-bit	128-bit	256-bit	n-bit
RS or RS-A	88.1	178.4	359.2	1.377–1.403n
RS-S or RS-AS	58.9	119.1	239.6	0.920–0.936n
RS-alw2	64.1	129.4	260.2	1.002–1.016n

Table 5. Relationship between (U^i, V^i) and (U^{i+1}, V^{i+1}) in RS

Case	(U^{i+1}, V^{i+1})	Condition		Probability	
A	$(U^i/2, V^i)$	$U_0^i = 0$ and $V_0^i = 1$		¼	a
B	$(U^i, V^i/2)$	$U_0^i = 1$ and $V_0^i = 0$		¼	b
C	$((U^i - V^i)/2, V^i)$	$U_0^i = 1$ and $V_0^i = 1$	$U^i > V^i$	¼	c
D	$(U^i, (V^i - U^i)/2)$		$U^i \le V^i$	¼	

Table 6. Relationship between $P_i(x, y)$ and $P_{i+1}(x', y')$ for $x, y, x', y' \in \{0, 1\}$

(x',y') \ (x, y)	(0, 1)	(1, 0)	(1,1)	$P_{i+1}(x', y')$
(0, 1)	$a/2$	-	$c/4$	$a/2 + c/4 = a$
(1, 0)	-	$b/2$	$c/4$	$b/2 + c/4 = b$
(1, 1)	$a/2$	$b/2$	$c/2$	$a/2 + b/2 + c/2 = c$
$P_i(x, y)$	a	b	c	1

(U^i, V^i) and (U^{i+1}, V^{i+1}) for any i $(1 \le i < k)$ is shown in Table 5. In Table 5, $P_i(0, 1)$, $P_i(1, 0)$, and $P_i(1, 1)$ are denoted by a, b, and c, respectively, where $P_i(x, y)$ is the probability that $U_0^i = x$ and $V_0^i = y$ for $x, y \in \{0, 1\}$. a, b, and c can be calculated using Table 6, which shows $P_{i+1}(x', y')$ for $x', y' \in \{0, 1\}$ when (x, y) is (0, 1), (1, 0) and (1, 1), respectively. We assume that the probabilities that $U^i > V^i$ and $U^i \le V^i$ are the same, and that the results of subtraction or shift operations are odd or even with the same probability. Since $P_i(x, y) = P_{i+1}(x', y')$ for $(x, y) = (x', y')$, we can obtain $a = b = c/2$.

If U^k is expressed only using k and U^1, we can calculate k. First, we need to modify the expression of Case C in Table 5. To remove V^i from $(U^i - V^i)/2$, we assume $E(U^i - V^i) = U^i/\rho$ where $E(x)$ is the expected value of x. Using ρ, U^k can be expressed as follows, according to Table 5:

$$U^k = \left(\frac{1}{2}\right)^{\frac{k}{4}} \left(\frac{1}{2\rho}\right)^{\frac{k}{4}} U^1. \tag{5}$$

U^k is 1, so (5) is transformed to:

$$U^1 = (4\rho)^{\frac{k}{4}} = 2^{\frac{(2+\log_2 \rho)k}{4}}. \tag{6}$$

Table 7. Relationship between (U^i, V^i) and (U^{i+1}, V^{i+1}) in RS-S

(U^{i+1}, V^{i+1})	Condition			Probability
$(U^i/4, V^i)$	$U_{1:0}^i = 0$ and $V_{1:0}^i = 1$			1/16
$(U^i/2, V^i)$	$U_{1:0}^i = 2$ and $V_{1:0}^i = 1$			1/16
$(U^i, V^i/4)$	$U_{1:0}^i = 1$ and $V_{1:0}^i = 0$			1/16
$(U^i, V^i/2)$	$U_{1:0}^i = 1$ and $V_{1:0}^i = 2$			1/16
$((U^i - V^i)/4, V^i)$	$U_0^i = 1$ and $V_0^i = 1$	$U^i > V^i$	$U_1^i = V_1^i$	3/16
$((U^i - V^i)/2, V^i)$			$U_1^i \neq V_1^i$	3/16
$(U^i, (V^i - U^i)/4)$		$U^i \le V^i$	$U_1^i = V_1^i$	3/16
$(U^i, (V^i - U^i)/2)$			$U_1^i \neq V_1^i$	3/16

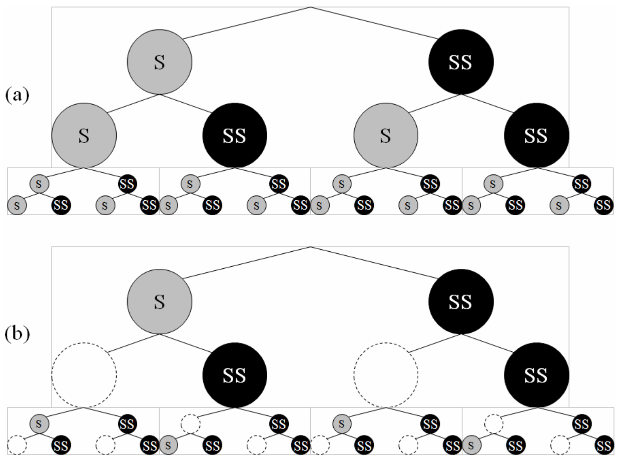


Fig. 3. Shifts and subtraction-and-shifts (a) RS, (b) RS-S.

Since U^1 is the n-bit modulus m, $2^{n-1} < U^1 < 2^n$. If $U^1 \approx 2^n$,

$$k = \frac{4}{(2 + \log_2 \rho)} n. \tag{7}$$

To find out ρ , we examined the ratio of U and V when both U and V are odd during simulation in Table 4. For $U^i > V^i$, $E(V^i)$ is $(4/9) \times U^i$, so we can obtain $\rho = 9/5$ and $k = 1.404n$. This value is very similar to the simulated result of 1.377–1.403n.

2. Number of Shifts in RS-S

In RS-S, the relationship of (U^i, V^i) and (U^{i+1}, V^{i+1}) is different, which is shown in Table 7. In RS, the expected proportion of shifts (Cases A and B in Table 5) to subtraction-and-shifts (Cases C and D in Table 5) is the same; while in RS-S, the proportion is 1/3 as shown in Table 7. The reason is explained using Fig. 3. In Fig. 3, the grey circles with S represent shifts, and the black circles with SS represent subtraction-and-shifts. Fig. 3

describes the selections of shifts or subtraction-and-shifts depending on whether both U and V are odd, or if either U or V is odd. The proportion of S to SS is the same in Fig. 3(a); while all S circles right after S or SS circles are removed in Fig. 3(b) due to 2-bit shifts. As a result, the proportion of S to SS in Fig. 3(b) is 1/3, as shown in Table 7. Using Table 7, U^k can be expressed as follows:

$$U^k = \left(\frac{1}{4}\right)^{\frac{k}{16}} \left(\frac{1}{2}\right)^{\frac{k}{16}} \left(\frac{1}{4\rho}\right)^{\frac{3k}{16}} \left(\frac{1}{2\rho}\right)^{\frac{3k}{16}} U^1. \tag{9}$$

Since $U^k = 1$ and $U^1 \approx 2^n$,

$$k = \frac{8}{3(2+\log_2 \rho)} n. \tag{10}$$

Using $\rho = 9/5$, we obtain $k = 0.936n$, which is very similar to the simulated result of 0.920 to 0.936. This is only about 67% of that in RS, so the RS-S algorithm is about 50% faster than the RS algorithm. In addition, the speed of the RS-S algorithm is almost double speed of the SE and LS algorithms.

3. Number of Shifts in RS-alw2

In RS-alw2, 1-bit shift after subtraction is replaced with 2-bit shift after addition. Therefore, $((U^i - V^i)/2, V^i)$ and $(U^i, (V^i - U^i)/2)$ in Table 7 become $((U^i + V^i)/4, V^i)$ and $(U^i, (V^i + U^i)/4)$ in RS-alw2, respectively. Based on the modified relationship, U^k can be expressed as follows:

$$U^k = \left(\frac{1}{4}\right)^{\frac{k}{16}} \left(\frac{1}{2}\right)^{\frac{k}{16}} \left(\frac{1}{4\rho_1}\right)^{\frac{3k}{16}} \left(\frac{1}{4\rho_2}\right)^{\frac{3k}{16}} U^1, \tag{11}$$

where $E(U^i - V^i) = U^i/\rho_1$, and $E(U^i + V^i) = U^i/\rho_2$. Since $U^k = 1$ and $U^1 \approx 2^n$,

$$k = \frac{16}{15 + 3\log_2 \rho_1 + 3\log_2 \rho_2} n. \tag{12}$$

$\rho_1 = \rho = 9/5$, and $\rho_2 = 9/13$ from $E(U^i - V^i) + E(U^i + V^i) = U^i$. By substituting $\rho_1 = 9/5$ and $\rho_2 = 9/13$, we obtain $k = 1.003n$, which is very similar to the simulated result of 1.002 to 1.016. This is similar to that of RS-S, but slightly slower. This means that the always-2-bit-shift using addition is not as good as the selectively-2-bit-shift.

V. Implementation and Experimental Results

Using 0.18 μm technology, we implemented Algorithms 1-5 and RS-alw2 based on the two features in Section II-2.A. In this section, we show their structures and analyze their speed and area.

1. Structures of the HW MI Modules

The structures of the HW MI modules are shown in Fig. 4, which include their data paths, but not their control logic. RS-AS has additional logic for 2-bit shifts in Fig. 4(f) compared with RS-A in Fig. 4(e). As shown in Fig. 4, SE, LS, RS-A, and RS-AS are simpler and need fewer adders than RS and RS-alw2. The exact area and consumed clock cycles are analyzed in the next sub-section.

2. Experimental Results of Implemented MI Modules

The area and the number of required clock cycles of the implemented modules are shown in Table 8. 100000 samples are used to estimate the average number of required clock cycles. With respect to area, SE is the smallest. As shown in Fig. 4, RS needs four more adders than SE; while RS-A needs only one more adder than SE. As a result, RS-A is about 28% smaller than RS, but only about 4.5% larger than SE at the same clock frequency (100 MHz).

In [16, 18], it was claimed that LS has better performance than other algorithms, but Table 8 shows different results. The number of shifts in RS is smaller than that in LS, but RS has more additions for halving odd R or S. This is why LS is faster than RS in SW implementation. However, in HW implementation, without increase of required clock cycles, halving R or S can be performed by adding more adders. The only weak point of this method is the area overhead, but we could reduce the number of adders in RS using our method explained in Section III-2. In addition, complicated control logic such as $U \neq \pm 2^u$ && $V \neq \pm 2^v$ in LS increases the area of LS; its area is larger than RS-A and RS-AS.

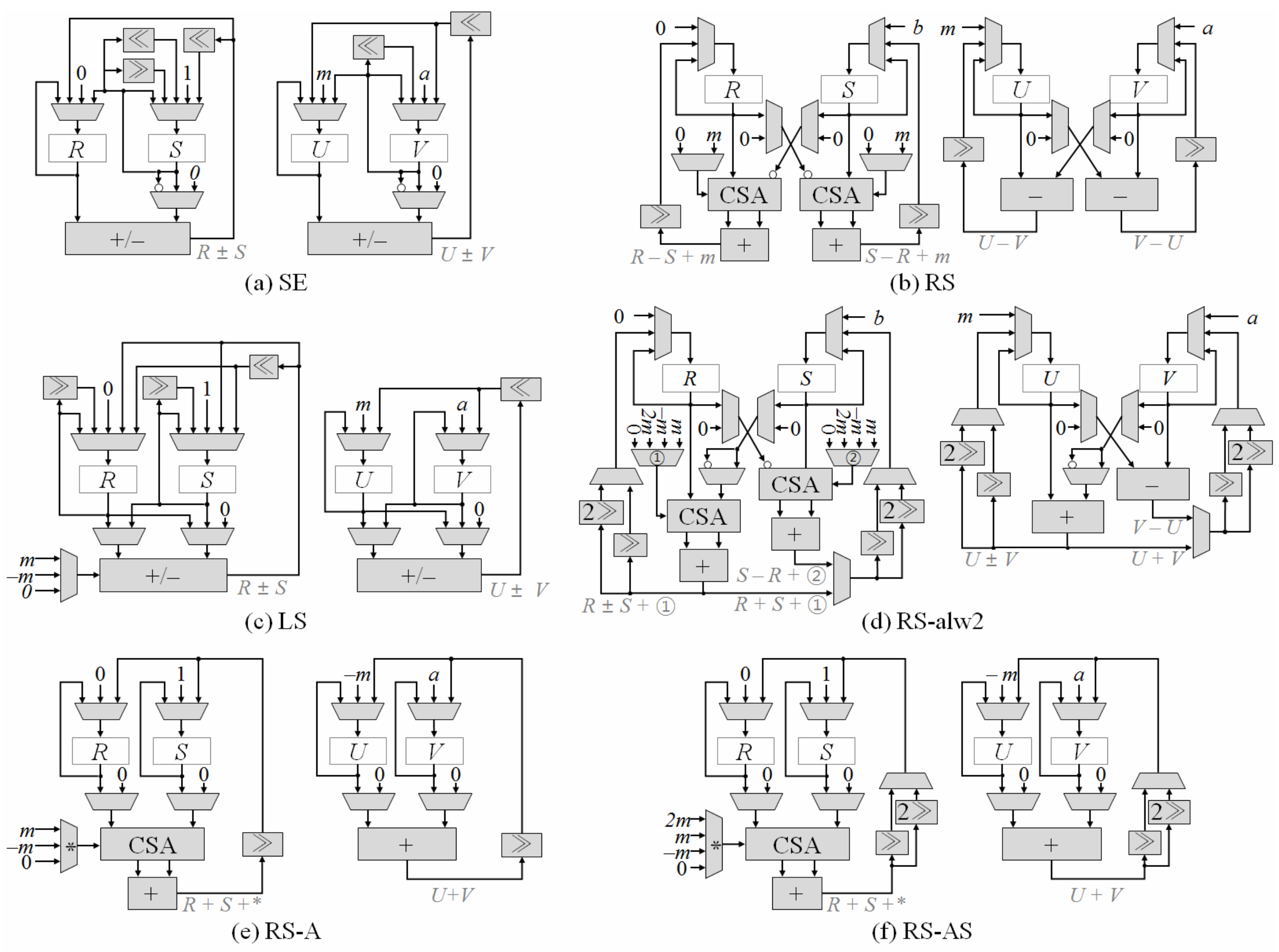


Fig. 4. Structures of HW MI modules (a) SE, (b) RS, (c) LS, (d) RS-alw2, (e) RS-A, (f) RS-AS.

Table 8. Area, Consumed Clock Cycles and Processing Time of Implemented MI Modules

Algorithm		Average number of required clock cycles			Area @100 MHz	Area @ Max. Frequency		Processing time (a) × (b)
		128-bit	256-bit	n-bit (a)		Area	Clock period (b)	
SE	Algorithm 1	258.1	514.1	2.008-2.016n	22179 GE	31130 GE	4.5 ns	9.0n ns
RS	Algorithm 2	180.0	360.7	1.408-1.409n	32541 GE	42862 GE	5.0 ns	7.0n ns
LS	Algorithm 3	252.7	508.6	1.974-1.987n	27367 GE	39793 GE	6.5 ns	12.9n ns
RS-alw2	-	130.1	261.6	1.016-1.022n	35963 GE	57130 GE	6.5 ns	6.6n ns
RS-A	Algorithm 4	180.0	360.7	1.408-1.409n	23405 GE	37274 GE	4.5 ns	6.3n ns
RS-AS	Algorithm 5	120.5	241.0	0.941n	26001 GE	40274 GE	4.5 ns	4.2n ns

VI. Application to ECC

In this section, we analyze the effect of the new inversion algorithm on the overall performance of ECC. Point multiplication (PM) is the main ECC operation, which is a repetition of point doublings (PDs) and point additions (PAs). These point operations are expressed using MM and MI, as shown in Table 9. Modular addition and modular subtraction are not included since each of them can be done in a single clock cycle. PA is only performed when each bit of a scalar is one, so the expected execution number of PA is approximately 0.5n. While this number can be further reduced using some techniques such as signed digit representation and sliding windows methods, 0.5n is used in this paper for simple analysis.

When the MI speed is much slower than MM, an MI is replaced with several MMs using projective coordinates instead of affine coordinates. Jacobian projective coordinates are one of the most efficient projective

Table 9. Necessary Modular Operations for Point Operations

		PD	PA	PM (n × PD + (0.5n) × PA)
Affine	SE, LS	MI + 4MM	MI + 3MM	1.5n × MI + 5.5n × MM
	RS	MD + 3MM	MD + 2MM	1.5n × MD + 4n × MM
Jacobian		8MM	11MM	13.5n × MM

Table 10. Expected Clock Cycles of PM when n = 256

		T(MM)			
		32	64	96	128
Jacobian		110.6k	221.2k	331.8k	442.4k
	RS-AS	125.2k	157.9k	190.7k	223.5k
	RS-A	170.4k	203.2k	235.9k	268.7k
	LS	239.7k	284.8k	329.8k	374.9k

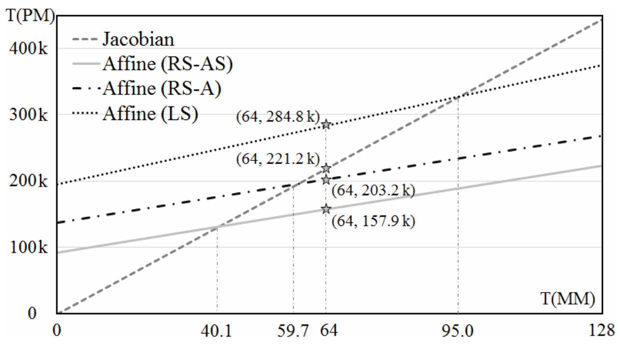


Fig. 5. T(PM) depending on T(MM) and MI algorithms.

coordinates and are widely used. When projective coordinates are used, a point with projective coordinates should be transformed to a point with affine coordinates last. This requires one MI and several MMs, but these operations are not included in Table 9 because they are negligible.

In addition, the necessary modular operations for affine coordinates are different depending on which MI is used. This is because RS can perform modular inversion as well as modular division (MD) differently than SE and LS. As a result, fewer MMs are necessary when RS is used.

Let T(x) be the expected number of clock cycles consumed for an operation x. Table 10 shows T(PM) calculated using Table 9 when n = 256. We assume that all modular operations are performed in the same clock domain, so only the number of required clock cycles is considered.

Fig. 5 shows T(PM) depending on T(MM) and MI algorithms. Now, we show that in some cases, the RS variants may have an additional advantage of simplifying the ECC hardware by avoiding the use of Jacobian coordinates and using only affine coordinates. From Fig. 5, we can find the breakeven points for the coordinate system decision. For example, if we use RS-A, Jacobian coordinates are preferable when T(MM) < 59.7. On the other hand, if T(MM) > 59.7, affine coordinates are better. These breakeven points are T(MM) = 40.1, 59.7, and 95.0 for RS-AS, RS-A, and LS, respectively. The greater the number, the wider the scope where Jacobian coordinates are recommended becomes. The breakeven value for RS-AS is 40.1, which is the smallest among RS-A, RS-AS, and LS; the breakeven value for LS is 95.0, which is the largest. This means that when LS is used for inversion, Jacobian coordinates are required for a wider range of T(MM) than when RS-AS is used. For example, if T(MM) is 64, Jacobian coordinates are better for LS while affine coordinates are better for RS-A and RS-AS. These are marked as starts in Fig. 5. It should be noted that if the use of Jacobian coordinates can be avoided, the design of an HW ECC module becomes much simpler because PM calculation using Jacobian coordinates is very complicated compared to using affine coordinates. Moreover, by the use of RS-AS, T(PM) itself is also reduced. In the above example with T(MM) = 64, Jacobian coordinates should be used only if an LS module is available, resulting in T(PM) = 221.2k cycles. However, if an RS-AS module is available, T(PM) becomes 157.9k cycles with only affine computation. In summary, by adopting RS-AS, a hardware ECC module becomes faster and simpler without using Jacobian coordinates.

VII. Conclusion

In this paper, we analyzed various types of MI algorithms. In contrast to the claims in many previous works, the performance of RS family is good when implemented in hardware. Even though the hardware RS module has area overhead, a large part of the area overhead can be removed using our method. This method is also applicable for speed-improvement to RS variants, resulting in RS with both area- and speed-improvements (RS-AS). Although there is another existing RS variant (RS-alw2) with a similar speed, its algorithm speed is slightly slower and its area is much larger than ours. Finally, we analyzed the ECC performance when using various modular inversion algorithms. When our improved modular inversion modules are used, the modular multiplier's speed scope where affine coordinates are recommended becomes wider, as well as improving the entire ECC performance for some scope.

ACKNOWLEDGMENT

This work was supported by the research fund of Signal Intelligence Research Center supervised by Defense Acquisition Program Administration and Agency for Defense Development of Korea.

REFERENCES

[1] P. Choi, S. Lee, and D. K. Kim, "Design of efficient modular inversion module using resource sharing," *Multimedia Information Technology and Applications, 2015, MITA 2015, 11th KMMS International Conference on*, pp.298-299, Jun., 2015.

[2] P. Choi, J.-T. Kong, and D. K. Kim, "Analysis of hardware modular inversion modules for elliptic curve cryptography," *International SoC Design Conference, 2015, ISOCC 2015*, pp.313-314, Nov., 2015.

[3] E. Barker and A. Roginsky, "NIST Special Publication 800-131A Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths," ed: NIST, 2011.

[4] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," *Cryptographic Hardware and Embedded Systems, 2008, CHES 2008*, ed: Springer, pp.62-78, 2008.

[5] B. MuthuKumar and S. Jeevananthan, "High speed hardware implementation of an elliptic curve cryptography (ECC) co-processor," *Trendz in Information Sciences & Computing, 2010, TISC 2010*, pp.176-180, 2010.

[6] G. Chen, G. Bai, and H. Chen, "A dual-field elliptic curve cryptographic processor based on a systolic arithmetic unit," *Circuits and Systems, 2008, ISCAS 2008, IEEE International Symposium on*, pp.3298-3301, 2008.

[7] J.-Y. Lai and C.-T. Huang, "Elixir: High-throughput cost-effective dual-field processors and the design framework for elliptic curve cryptography," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol.16, pp.1567-1580, 2008.

[8] J.-Y. Lai and C.-T. Huang, "A highly efficient cipher processor for dual-field elliptic curve cryptography," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, Vol.56, pp. 394-398, 2009.

[9] J.-Y. Lai and C.-T. Huang, "Energy-adaptive dual-field processor for high-performance elliptic curve cryptographic applications," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol.19, pp.1512-1517, 2011.

[10] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*: Springer Science & Business Media, 2006.

[11] P. Gallagher and C. Kerry, "Fips pub 186-4: Digital signature standard, dss," ed: NIST, 2013.

[12] N. Takagi, "A modular inversion hardware algorithm with a redundant binary representation," *Information and Systems, IEICE Transactions on*, Vol.76, pp.863-869, 1993.

[13] X. Yan and S. Li, "Modified modular inversion algorithm for VLSI implementation," *7th International Conference on ASIC*, pp.90-93, 2007.

[14] C. Chen and Z. Qin, "Fast algorithm and hardware architecture for modular inversion in GF (p)," *Intelligent Networks and Intelligent Systems, 2009, ICINIS 2009, 2nd International Conference on*, pp.43-45, 2009.

[15] S. Ma, Y. Hao, Z. Pan, and H. Chen, "Fast implementation for modular inversion and scalar multiplication in the elliptic curve cryptography," *Intelligent Information Technology Application, 2008, IITA 2008, 2nd International Symposium on*, pp.488-492, 2008.

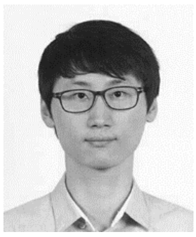
[16] R. Lórencz, "New algorithm for classical modular inverse," in *Cryptographic Hardware and Embedded Systems, 2002, CHES 2002*, ed: Springer, pp. 57-70, 2002.

[17] J. Hlaváč and R. Lórencz, "Arithmetic unit for computations in GF (p) with the left-shifting multiplicative inverse algorithm," *Architecture of Computing Systems, 2013, ARCS 2013*, ed: Springer, pp.268-279, 2013.

[18] L. Hars, "Modular inverse algorithms without

multiplications for cryptographic applications," *Embedded Systems, EURASIP Journal on*, Vol.2006, pp.1-13, 2006.

[19] D. Galbi and A. K. Chan, "Four-to-two adder cell for parallel multiplication," ed: Google Patents, 1990.



Piljoo Choi received the B.S. and M.S. degrees in Electronic Engineering from Hanyang University in 2010 and 2012, respectively. He is currently a Ph.D. candidate in the Department of Electronic Engineering at Hanyang University, Korea. His research interests are in the areas of security SoC (System on Chip), crypto-coprocessors, and information security.

Mun-Kyu Lee received the B.S. and M.S. degrees in Computer Engineering from Seoul National University in 1996 and 1998, respectively, and the Ph.D. degree in Electrical Engineering and Computer Science from Seoul National University in 2003. From 2003 to 2005, he was a senior engineer at Electronics and Telecommunications Research Institute, Korea. He is currently a professor in the Department of Computer and Information Engineering at Inha University, Korea. His research interests are in the areas of cryptographic algorithms, information security and theory of computation.

Jeong-Taek Kong received the B.S. degree in Electronics Engineering from Hanyang University, Korea, in 1981, the M.S. degree in Electronics Engineering from Yonsei University, Korea, in 1983, and the Ph.D. degree in Electrical Engineering from Duke University, Durham, NC, in 1994. From 1983 to 1990, he was with Samsung Electronics Co., Ltd., as a VLSI CAD manager. From 1990 to 1994, he was at Duke University granted by a Fellowship from Samsung Electronics Co., Ltd. He was with Semiconductor Business, Samsung Electronics Co., as VP of CAE Team, Senior VP of Intellectual Property Team, and Vice Chancellor of Samsung Institute of Technology. In 2014, he became Professor of Dept. of Electronic Engineering at Hanyang University, Korea. He is currently Professor of Software College at Sungkyunkwan University, Korea. He has authored and coauthored more than 130 technical papers and a book titled Digital Timing Macromodeling for VLSI Design Verification (Norwell, MA: Kluwer, 1995). His research interests focus on various VLSI CAD tools and design methodologies. He has served on the program committees of a number of Conferences such as IEEE Electron Devices Meeting (IEDM) and IEEE International Symposium on Quality of Electronic Design (ISEQD) where he received distinguished Fellow Award. He served as a Distinguished Lecturer for the IEEE Circuits and Systems Society. He was also an Associate Editor of IEEE Transactions on Circuits and Systems-II and IEEE Transactions on Very Large Scale Integrated (VLSI) Systems.

Dong Kyue Kim received the B.S., M.S. and Ph.D. degrees in Computer Engineering from Seoul National University in 1992, 1994, and 1999, respectively. From 1999 to 2005, he was an assistant professor in the Division of Computer Science and Engineering at Pusan National University. He is currently a full professor in the Department of Electronic Engineering at Hanyang University, Korea. His research interests are in the areas of security SoC (System on Chip), crypto-coprocessors, and information security.