A Compact Architecture for Montgomery Elliptic Curve Scalar Multiplication Processor

Yong Ki Lee[(1)] and Ingrid Verbauwhede[(1),(2)]

[(1)] University of California, Los Angeles, USA
[(2)] Katholieke Universiteit Leuven, Belgium
{jfirst,ingrid}@ee.ucla.edu

Abstract. We propose a compact architecture of a Montgomery elliptic curve scalar multiplier in a projective coordinate system over $GF(2^m)$. To minimize the gate area of the architecture, we use the common Z projective coordinate system where a common Z value is kept for two elliptic curve points during the calculations, which results in one register reduction. In addition, by reusing the registers we are able to reduce two more registers. Therefore, we reduce the number of registers required for elliptic curve processor from 9 to 6 (a 33%). Moreover, a unidirectional circular shift register file reduces the complexity of the register file, resulting in a further 17% reduction of total gate area in our design. As a result, the total gate area is 13.2k gates with 314k cycles which is the smallest compared to the previous works.

Key words: Compact Elliptic Curve Processor, Montgomery Scalar Multiplication

1 Introduction

Even though the technology of ASIC advances and its implementation cost decreases steadily, compact implementations of security engines are still a challenging issue. RFID (Radio Frequency IDentification) systems, smart card systems and sensor networks are good examples which need very compact security implementations. Public key cryptography algorithms seem especially taxing for such applications. However, for some security properties such as randomized authentications and digital signatures, the use of public key cryptography algorithms is often inevitable. Among public key cryptography algorithms, elliptic curve cryptography is a good candidate due to its efficient computation and relatively small key size.

In this paper, we propose an architecture for compact elliptic curve multiplication processors using the Montgomery algorithm [1]. The Montgomery algorithm is one of the most popular algorithms in elliptic curve scalar multiplication due to its resistance to side-channel attack. We use the projective coordinate system to avoid inverse operations.

In order to minimize the system size, we propose new formulae for the common projective coordinate system where all the Z-coordinate values are equal.

When we use López-Dahab's Montgomery scalar multiplication algorithm [2], two elliptic curve points must be kept where X and Z-coordinate values for each point. Therefore, by the use of the common Z projective coordinate property, one register for a Z-coordinate can be reduced. Considering that the register size is quite large, e.g. 163, reducing even one register is a very effective way to minimize the gate area. Moreover, efficient register management by reuse of the registers makes it possible to reduce two additional registers. Therefore, we reduce three registers out of nine in total compared to a conventional architecture. In addition, we design a unidirectional circular shift register file to reduce the complexity of the register file. While the multiplexer complexity of a register file increases as the square of the number of the registers, that of our register file is a small constant. Therefore, the proposed register file architecture effectively reduces the overall area. Though the register file is small (6 registers) an additional 17% of gate area is reduced using this technique. We also show the synthesis results for various digit sizes where the smallest area is 13.2k gates with the cycles of 314k.

The remainder of this paper is organized as follows. In Section 2, we review the background on which our work is based. In Section 3, the common Z projective coordinate system is introduced and its corresponding formulae are given. The proposed system architecture and the synthesis results are shown in Section 4 and Section 5 followed by the conclusion in Section 6.

2 Background

2.1 López-Dahab's Montgomery scalar multiplication

In this section we introduce López-Dahab's Montgomery scalar multiplication algorithm, which uses a projective coordinate system [2]. The algorithm is shown in Fig. 1. A non-supersingular elliptic curve E over $GF(2^m)$ is the set of coordinative points (x, y) satisfying $y^2 + xy = x^3 + ax^2 + b$ with the point at infinity O, where $a, b, x, y \in GF(2^m)$ and $b \neq 0$.

Input: A point $P = (x, y) \in E$ and a positive integer $k = 2^{l-1} + \Sigma_{i=0}^{l-2} k_i 2^i$
Output: $Q = kP$

1. if ($k = 0$ or $x = 0$) then output $(0, 0)$ and stop
2. $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
3. for $i = l - 2$ to 0 do
   if $k_i = 1$ then
   $(X_1, Z_1) \leftarrow \text{Madd}(X_1, Z_1, X_2, Z_2)$, $(X_2, Z_2) \leftarrow \text{Mdouble}(X_2, Z_2)$
   else $(X_2, Z_2) \leftarrow \text{Madd}(X_2, Z_2, X_1, Z_1)$, $(X_1, Z_1) \leftarrow \text{Mdouble}(X_1, Z_1)$
4. return $Q \leftarrow \text{Mxy}(X_1, Z_1, X_2, Z_2)$

Fig. 1. Montgomery scalar multiplication with López-Dahab algorithm

The adding formula of $(X_{Add}, Z_{Add}) \leftarrow \mathrm{Madd}(X_1, Z_1, X_2, Z_2)$ is defined in Eq. 1.

$$\begin{aligned} Z_{Add} &= (X_1 \times Z_2 + X_2 \times Z_1)^2 \\ X_{Add} &= x \times Z_{Add} + (X_1 \times Z_2) \times (X_2 \times Z_1) \end{aligned} \tag{1}$$

The doubling formula of $(X_{Double}, Z_{Double}) \leftarrow \mathrm{Mdouble}(X_2, Z_2)$ is defined in Eq. 2.

$$\begin{aligned} Z_{Double} &= (X_2 \times Z_2)^2 \\ X_{Double} &= X_2^4 + b \times Z_2^4 \end{aligned} \tag{2}$$

$Q \leftarrow \mathrm{Mxy}(X_1, Z_1, X_2, Z_2)$ is the conversion of projective coordinate to affine coordinate. López-Dahab's adding and doubling algorithms are described in Fig. 2 where $c^2 = b$.

The total number of registers in Fig. 2 is six, i.e. the registers for X_1, Z_1, X_2, Z_2, T_1 and T_2. The total field operations of Adding Algorithm are 4 multiplications, 1 square and 2 additions, and those of Doubling Algorithm are 2 multiplications, 4 squares and 1 addition. Note that it is not necessary to maintain Y-coordinate during the iterations since it can be derived at the end of the iterations.

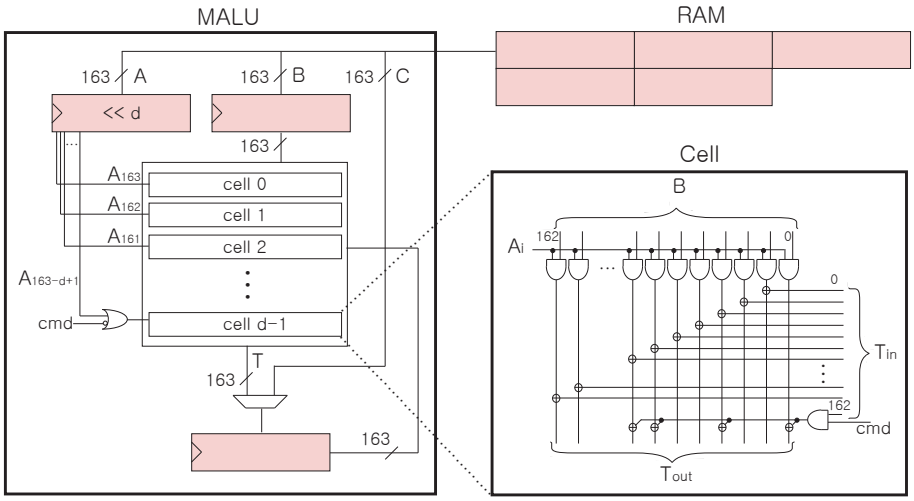
Adding Algorithm $(X_1, Z_1) \leftarrow \mathrm{Madd}(X_1, Z_1, X_2, Z_2)$	Doubling Algorithm $(X, Z) \leftarrow \mathrm{Mdouble}(X, Z)$
1. $T_1 \leftarrow x$	1. $T_1 \leftarrow c$
2. $X_1 \leftarrow X_1 \times Z_2$	2. $X \leftarrow X^2$
3. $Z_1 \leftarrow Z_1 \times X_2$	3. $Z \leftarrow Z^2$
4. $T_2 \leftarrow X_1 \times Z_1$	4. $T_1 \leftarrow Z \times T_1$
5. $Z_1 \leftarrow Z_1 + X_1$	5. $Z \leftarrow Z \times X$
6. $Z_1 \leftarrow Z_1^2$	6. $T_1 \leftarrow T_1^2$
7. $X_1 \leftarrow Z_1 \times T_1$	7. $X \leftarrow X^2$
8. $X_1 \leftarrow X_1 + T_2$	8. $X \leftarrow X + T_1$

Fig. 2. López-Dahab's Adding and Doubling Algorithms

2.2 Modular Arithmetic Logic Unit (MALU) and Elliptic Curve Processor Architecture

In order to perform the filed operations, i.e. the multiplications, squares and additions in Fig. 2, we need an Arithmetic Logic Unit (ALU). Fig. 3 shows the MALU architecture of K. Sakiyama et al [5]. This is a compact architecture which performs the arithmetic field operations as shown in Eq. 3.

$$\begin{aligned} C(x) &= A(x) * B(x) \bmod P(x) \quad \text{if } cmd = 1 \\ C(x) &= B(x) + C(x) \bmod P(x) \quad \text{if } cmd = 0 \end{aligned} \tag{3}$$



(a) MALU Architecture (b) Cell Architecture

Fig. 3. MALU Architecture

where $A(x) = \Sigma a_i x^i$, $B(x) = \Sigma b_i x^i$, $C(x) = \Sigma c_i x^i$ and $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

d is the digit size and the number of cells. The square operation uses the same logic as the multiplication by duplicating the operand. The arithmetic multiplication and addition take $\lceil \frac{163}{d} \rceil$ and one cycle respectively. The benefit of this architecture is that the multiplication, the square and the addition operations share the XOR array and by increasing the digit size, the MALU can be easily scaled. The architecture of our ALU starts from this MALU.

ECP (Elliptic Curve Processor) architecture based on MALU is shown in Fig. 4 [6]. Note that in Fig. 4, ALU is implemented with MALU and hence includes three registers, and RAM contains five words of 163 bit size.

2.3 Implementation Consideration

If López-Dahab's Montgomery scalar multiplication algorithm is implemented using Sakiyama's MALU in a conventional way, the total number of registers is 9, i.e. 3 registers for MALU plus 6 registers for the Montgomery scalar multiplication. In [6], 3 registers and 5 RAMs are used (8 memory elements in total). One register is reduced by modifying López-Dahab's algorithm and assuming that constants are loadable directly to the MALU without using a register. In our architecture, we are able to reduce the number of registers to 6 even without constraining ourselves to these assumptions. This was accomplished by observing the fact that the area of a scalar multiplier is dominated by register area. Note that the registers occupy more than 80% of the gate area in a conventional architecture. Therefore, reducing the number of the registers and the complexity of the register file is a very effective way to minimize the total gate area.

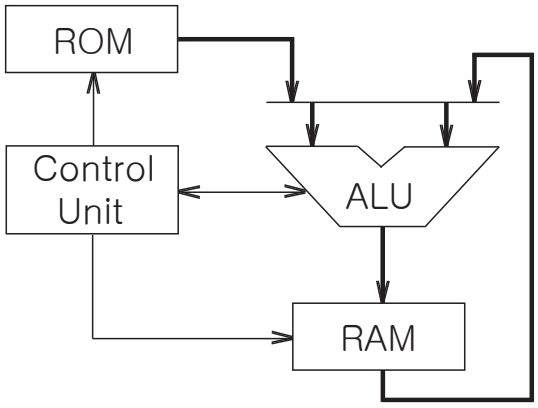


Fig. 4. MALU based Elliptic Curve Processor Architecture

Accordingly, our compact architecture is achieved in two folds: reducing the number of registers (one register reduction by using the common Z projective coordinate system and two register reduction by register reuse) and reducing the register file complexity by designing a unidirectional circular shift register file.

3 Common Z Projective Coordinate System

We propose new formulae for the common Z projective coordinate system where the Z values of two elliptic curve points in Montgomery scalar multiplication are kept to be the same during the process. New formulae for the common Z projective coordinate system have been proposed over prime fields in [3]. However, this work is still different from ours in that first, they made new formulae over prime filed while ours is over binary polynomial field and second, they made new formulae to reduce the computation amount in special addition chain while our formulae slightly increase the computation amount in order to reduce the number of the registers. Please note that reducing even one register decreases the total gate area considerably.

Since in López-Dahab's algorithm, two elliptic curve points must be maintained, the required number of registers for this is four (X_1, Z_1, X_2 and Z_2). Including two temporary registers (T_1 and T_2), the total number of registers is six. The idea of the common Z projective coordinate system is to make sure that $Z_1 = Z_2$ at each iteration of López-Dahab's algorithm. The condition at the beginning of the iterations is satisfied since the algorithm starts the iterations with the initialization of $Z_1 = Z_2 = 1$. Even if $Z_1 \neq Z_2$, we can make it satisfy this condition using three field multiplications as shown in Eq. 4 where the resulting coordinate set is (X_1, X_2, Z).

$$\begin{aligned} X_1 &\leftarrow X_1 \times Z_2 \\ X_2 &\leftarrow X_2 \times Z_1 \\ Z &\leftarrow Z_1 \times Z_2 \end{aligned} \tag{4}$$

Since we now assume $Z_1 = Z_2$, we can start the Adding Algorithm with the common Z projective coordinate system. With $Z = Z_1 = Z_2$, Eq. 1 is re-represented as shown in Eq. 5. Now Z_{Add} and X_{Add} have a common factor of Z^2.

$$\begin{aligned} Z_{Add} &= (X_1 \times Z_2 + X_2 \times Z_1)^2 = (X_1 + X_2)^2 \times Z^2 \\ X_{Add} &= x \times Z_{Add} + (X_1 \times Z_2) \times (X_2 \times Z_1) \\ &= x \times Z_{Add} + (X_1 \times X_2 \times Z^2) \end{aligned} \tag{5}$$

Due to the property of the projective coordinate system, we can divide Z_{Add} and X_{Add} by the common factor of Z^2. The comparison of the original equation and the modified equation is shown in Table 1. Note that the new formula of the Adding Algorithm is independent of the previous Z-coordinate value.

Table 1. The comparison between the original and the modified formulas

The original equation	The new equation assuming $Z = Z_1 = Z_2$
$Z_{Add} = (X_1 \times Z_2 + X_2 \times Z_1)^2$	$Z_{Add} = (X_1 + X_2)^2$
$X_{Add} = x \times Z_{Add} + (X_1 \times Z_2) \times (X_2 \times Z_1)$	$X_{Add} = x \times Z_{Add} + X_1 \times X_2$

In Doubling Algorithm, there is no such reduction since it deals with only one elliptic curve point. Nevertheless, we can simplify the Doubling Algorithm by noticing that $T_1^2 + X^2 \equiv (T_1 + X)^2$ at the steps of 6, 7 and 8 in Fig. 2. One field multiplication can be reduced using this mathematical equality. The Eq. 2 is re-represented in Eq. 6 where $c^2 = b$.

$$\begin{aligned} Z_{Double} &= (X_2 \times Z)^2 \\ X_{Double} &= (X_2^2 + c \times Z^2)^2 \end{aligned} \tag{6}$$

Note that the resulting Z-coordinate values are different between Adding and Doubling formulae. In order to maintain a common Z-coordinate value, some extra steps similar to Eq. 4 are required. These extra steps must follow every pair of Adding and Doubling Algorithm. The final mathematical expression and its algorithm are shown in Eq. 7 and Fig. 5 respectively.

$$\begin{aligned} X_1 &\leftarrow X_{Add}Z_{Double} = \left\{x(X_1 + X_2)^2 + X_1X_2\right\}(X_2Z)^2 \quad ; P1 \leftarrow P1 + P2 \\ X_2 &\leftarrow X_{Double}Z_{Add} = (X_2^2 + cZ^2)^2(X_1 + X_2)^2 \quad ; P2 \leftarrow 2 \times P2 \\ Z &\leftarrow Z_{Add}Z_{Double} = (X_1 + X_2)^2(X_2Z)^2 \quad ; \text{The new common Z-coordinate} \end{aligned} \tag{7}$$

Adding Algorithm	Doubling Algorithm	Extra Steps
1. $T_2 \leftarrow X_1 + X_2$	1. $X_2 \leftarrow X_2^2 \quad (T_1)$	1. $X_1 \leftarrow X_1 \times Z$
2. $T_2 \leftarrow T_2^2 \quad (T_1)$	2. $Z \leftarrow Z^2 \quad (T_1)$	2. $X_2 \leftarrow X_2 \times T_2$
3. $T_1 \leftarrow X_1 \times X_2$	3. $T_1 \leftarrow c$	3. $Z \leftarrow Z \times T_2$
4. $X_1 \leftarrow x$	4. $T_1 \leftarrow Z \times T_1$	
5. $X_1 \leftarrow T_2 \times X_1$	5. $Z \leftarrow Z \times X_2$	
6. $X_1 \leftarrow X_1 + T_1$	6. $X_2 \leftarrow X_2 + T_1$	
	7. $X_2 \leftarrow X_2^2 \quad (T_1)$	

Fig. 5. Proposing Adding and Doubling Algorithms

In Fig. 5 we mark with (T_1) at each square operation to indicate that the T_1 register is free to store some other value. The reason for this will be obvious in the next section. The comparison of the amount of field operations between López-Dahab's algorithm and our algorithm is shown in Table 2.

Table 2. Comparison of the computational workload

Field Operation	López-Dahab's algorithm	Our algorithm
Multiplication	6	7
Square	5	4
Addition	3	3

Noting that the multiplication and the square are equivalent in the MALU operation, the workload of our algorithm is the same as that of López-Dahab's algorithm and we still reduce one register.

4 Proposing System Architecture

4.1 Arithmetic Logic Unit (ALU) Architecture

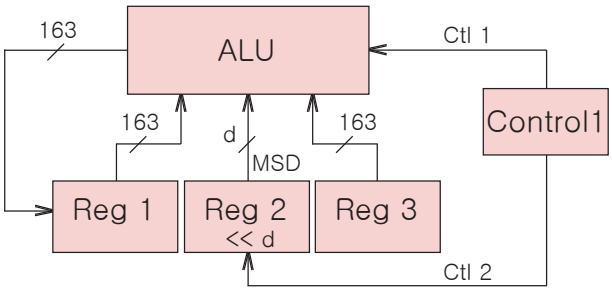


Fig. 6. ALU Architecture

The ALU architecture in Fig. 6 is similar to MALU in Fig. 3. The only difference is in the placement of the registers and the control outside the ALU block. Therefore, the ALU block is equivalent to an array of cells in Fig. 3. The reason we separate the registers from the ALU block is for the reuse of the registers. Note that at the completion of the multiplication or addition operation, only the register Reg1 is updated while the registers Reg2 and Reg3 are remained as the beginning of the operations. Therefore, Reg2 and Reg3 can be used not only to store field operands but also to store some values of the proposed Adding and Doubling algorithm where we need five registers for X_1, X_2, Z, T_1, and T_2 in Fig. 5.

A care should be taken at this point since the same value must be placed in the both of Reg2 and Reg3 for squaring. Therefore, during squaring, only one register can be reused. This fact would conflict with our purpose to reuse each of Reg2 and Reg3 as a storage of the adding and doubling algorithm. Fortunately, it is possible to free one of the registers to hold another value during squaring. As shown in Fig. 5, T_1 can be reused whenever a square operation is required.

In Fig. 6, the control line $Ctl1$ signals the command (multiplication or addition) and the last iteration of the multiplication. When ALU performs a multiplication, each digit of d bits of Reg2 must be entered into ALU in order. Instead of addressing each digit of the 163 bit word, the most significant digit (MSD) is entered and a circular shift of d bits is performed. The shift operation must be circular and the last shift must be the remainder of $163/d$ since the value must be kept as the initial value at the end of the operation. During performing the ALU operation, an intermediate result is stored in Reg1. Reg1, Reg2 and Reg3 are comparable with C, A and B in Fig. 3 respectively.

4.2 Circular Shift Register File Architecture

By reusing the registers, we reduce two of the registers in the previous sub-Section. This causes that all the registers should be organized in single register file. Therefore, the register file of our system consists of five registers. In our register file architecture, we use a circular shift register file with a minimum number of operations. The multiplexer complexity of a randomly accessible register file increases as the square of the number of registers. On the other hand, since the multiplexer complexity of a circular shift register file is a constant, this model effectively reduces the total gate area.

The operations defined in Fig. 7 are the minimum operations such that any replacement or reordering of the register values can be achieved. Since only Reg1 gets multiple inputs, only one multiplexer of fixed size is necessary.

Note that Reg1, Reg2 and Reg3 in Fig. 7 are the three registers which are connected to the ALU in Fig. 6. The assignment operation loads the constants of elliptic curve parameters into Reg1 which is the only register to be assigned a constant value. The shift operation shifts the register values in circular and the switch operation switches the values of Reg1 and Reg2. The copy operation replaces the value of Reg1 with Reg2. Note that the copy operation is required

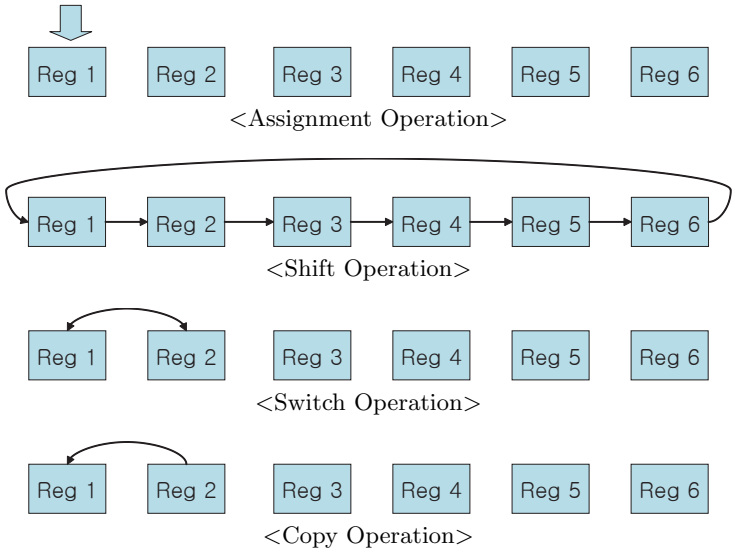


Fig. 7. Operations and Architecture of Register File

for the field square operation which is implemented as the field multiplication with two operands of the same value.

4.3 Overall System Architecture

The overall system architecture is shown in Fig. 8. Elliptic curve point add and doubler (EC Add&Doubler) consists of Control 1, ALU and the register file. Control 1 includes the hard-wired elliptic curve parameters and manages the register file. Control 2 detects the first bit of 1 in Key (or a scalar) and controls

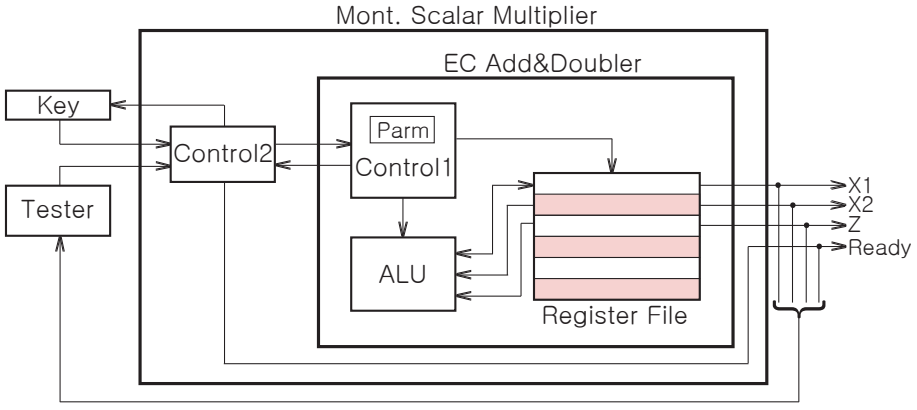


Fig. 8. Overall System Architecture

EC Add&Doubler depending on the Key values of the later bits according to the Montgomery algorithm in Fig. 1. Key and Tester are placed outside Montgomery scalar multiplier. We assume that Key can be addressable in single bit and the addressed bit is forwarded into Control 2. Control 2 also generates the Ready signal to indicate when the final outputs of $X1$, $X2$ and Z are ready. The outputs are compared with the pre-computed results in Tester.

In our system, we suppose that the coordinate conversion into affine coordinate system and calculation of Y-coordinate value are performed in the counterpart of this system if it is needed. If we assume that this system is implemented in RFID tags, the counter part can be a tag reader or back-end system.

4.4 Register File Management for Algorithm Implementation

For better understanding how the system works, the register file management of Adding Algorithm of Fig. 5 is shown in Fig. 9. Note that when the algorithm is actually implemented, some more detailed controls are required. In this example, only the register file rearrangement is shown. Remember that the field addition and multiplication are performed as Reg1 $\leftarrow$ Reg2 $\times$ Reg3 and Reg1 $\leftarrow$ Reg1 + Reg3 respectively. Initially, we assume that the arrangement of register values are as step (1) in Fig. 9, and that Reg4, Reg5 and Reg6 are not available (marked as –) since meaningful values are not stored yet. The commands of Assign, Shift, Switch and Copy work as described in Fig. 7 and the rearrangements of register values are shown in each step. Note that Shift$*$4 is the abbreviation of four times Shift operation and some register values are changed to be – when the old

Step	Field Operation	Command	Reg1	Reg2	Reg3	Reg4	Reg5	Reg6
(1)	Initial		X1	X2	Z	–	–	–
(2)		Shift	–	X1	X2	Z	–	–
(3)		Copy	X1	X1	X2	Z	–	–
(4)	1. $T_2 \leftarrow X_1 + X_2$	Add	T2	X1	X2	Z	–	–
(5)		Shift	–	T2	X1	X2	Z	–
(6)		Copy	T2	T2	X1	X2	Z	–
(7)		Shift	–	T2	T2	X1	X2	Z
(8)	2. $T_2 \leftarrow T_2^2$	Multiply	T2	–	–	X1	X2	Z
(9)		Shift$*$4	–	X1	X2	Z	T2	–
(10)	3. $T_1 \leftarrow X_1 \times X_2$	Multiply	T1	–	X2	Z	T2	–
(11)		Switch	–	T1	X2	Z	T2	–
(12)	4. $X_1 \leftarrow x$	Load x	X1	T1	X2	Z	T2	–
(13)		Shift$*$2	T2	–	X1	T1	X2	Z
(14)		Switch	–	T2	X1	T1	X2	Z
(15)	5. $X_1 \leftarrow T_2 \times X_1$	Multiply	X1	T2	–	T1	X2	Z
(16)		Switch	T2	X1	–	T1	X2	Z
(17)		Shift$*$5	X1	–	T1	X2	Z	T2
(18)	6. $X_1 \leftarrow X_1 + T_1$	Add	X1	–	T1	X2	Z	T2

Fig. 9. Register File Management for Adding Algorithm

values are not used any more. The rest of the Montgomery scalar multiplication algorithm can be also described similarly.

In fact, the use of this register file increases the number of cycles due to the control overhead. However, considering that a field multiplication takes a large number of cycles, the number of overhead cycles is relatively small. Note that a field multiplication requires 163 cycles for 163 bit words and the digit size of 1 (reference Fig. 3 for the digit size). We compare synthesis results for various cases in the following section.

5 Synthesis Results

In order to verify our algorithm and architecture, we synthesized the proposed architecture using TSMC $0.18\mu m$ standard cell library. Summarized results are shown in Table 3. While version 1 uses a randomly accessible register file, the other versions use the circular shift register file which is shown in the previous section. Comparing version 1 with version 2, we can see how changing the register file management strategy can effectively reduce the area (17% reduction of total gate area). The use of the circular shift register file requires more cycles. However, if we increase the digit size into 4 (version 5), a much smaller number of cycles can be achieved with even less gate area.

Table 3. Synthesis Results

Version	Digit size (d)	Register Type	Gate Area	Cycle
Ver1	1	Random Access	15,894	295,032
Ver2	1	Circular Shift	13,182	313,901
Ver3	2		14,188	168,911
Ver4	3		14,896	120,581
Ver5	4		15,538	95,521

A comparison with other works is shown in Table 4. Since the architecture of [4] uses an affine coordinate system, it requires only 6 registers but require a larger number of cycles due to field inverse operations. The ALU of [4] has separate logic modules for multiplication, square and addition where multiplication requires 163 cycles and square and addition require 1 cycle.

Except for [6], all the reported results include memory area. In [6], the reported gate area of 8,214 does not include the required RAM area. For a fairer comparison, we estimate the total gate area assuming that 1 bit memory is equivalent to 10 gate area. Note that 1 bit register require 6 gates and there should be some extra area for addressing. According to our experiment of synthesis for a 163 bit register in standard CMOS compilers, the number of gates per bit is above 10 gates. Even in this under-estimation, our results show much smaller gate number with a smaller cycle number. This result is obvious considering that our ALU is similar to [6] and the number of total memory units of our architecture is two less than [6]. In [8], among several implementations, we show the one

Table 4. Comparison with other works

	Technology	Key Size	Digit Size	Area (Gate)	Cycle	Memory Units
[7]	0.13	165	–	30,333	545,440	–
[8]	0.13	160	–	28,311	2,500,000	320∗8 bits
[4]	–	163	–	15,097	432,000	6∗163 bits
[6]	0.13	163	1	8,214 + 5 RAM ($\approx 16,364$)**	353,710	8∗163 bits
Our Work (Ver2)	0.18	163	1	13,182	313,901	6∗163 bits

**The gate area of 8,214 does not include RAM area. The gate area of 16,364 is an estimation including the RAM area.

having the smallest area, which is still much larger than our results and also has a much larger number of cycles. As a result, our implementation has not only the smallest area but also the smallest cycle.

6 Conclusion

We proposed a compact architecture for an elliptic curve scalar multiplier. This contribution has been achieved by reducing the number of the registers and the complexity of the register file.

The reduction of the number of the registers is done in two different approaches. By proposing new formulae for the common Z projective coordinate system, one register was reduced and by the reuse of the registers, two more registers were reduced. Accordingly, three registers were reduced in total. The reduction of the complexity of the register file is done by designing a circular shift register file.

As a result, for elliptic curve scalar multiplication, only 13.2k gates and 314k cycles are required. This result not only achieves the smallest area but also the smallest cycle number compared with fairly comparable architectures. Moreover, our processor architecture is secure against TA (Timing Analysis) and SPA (Simple Power Analysis) due to the property of Montgomery elliptic curve scalar multiplication.

Acknowledgments. This work is supported by NSF CCF-0541472, FWO and funds from Katholieke Universiteit Leuven.

References

1. P. Montgomery: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation, Vol. 48. (1987) 243–264

2. J. López and R. Dahab: Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, Vol. 1717. Springer-Verlag (1999) 316–327
3. N. Meloni: Fast and Secure elliptic Curve Scalar Multiplication Over Prime Fields Using Special Addition Chains. Cryptology ePrint Archive: listing for 2006 (2006/216) (2006)
4. C. Paar: Light-Weight Cryptography for Ubiquitous Computing. Invited talk at the University of California, Los Angeles (UCLA). Institute for Pure and Applied Mathematics (December 4, 2006)
5. K. Sakiyama, L. Batina, N. Mentens, B. Preneel and I. Verbauwhede: Small-footprint ALU for public-key processors for pervasive security. Workshop on RFID Security (2006) 12 pages
6. L. Batina, N. Mentens, K. Sakiyama, B. Preneel and I. Verbauwhede: Low-cost Elliptic Curve Cryptography for wireless sensor networks. Third European Workshop on Security and Pri-vacy in Ad hoc and Sensor Networks. Lecture Notes in Computer Science, Vol. 4357. Springer-Verlag (2006) 6–17
7. E. Ä OztÄurk, Berk Sunar and Erkay Savas: Low-power elliptic curve cryptography using scaled modular arithmetic. International Workshop on Cryptographic Hardware and Embedded Systems (CHES). Lecture Notes in Computer Science, Vol. 3156. Springer-Verlag (2004) 92–106
8. A. Satoh and K. Takano: A Scalable Dual-Field Elliptic Curve Cryptographic Processor. IEEE Transactions on Computers, Vol. 52. No. 4 (2003) 449–460