Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor

Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee

Dept. of Electronic and Electrical Eng., POSTECH
{jwchung,sim}@oberon.postech.ac.kr
pjl@postech.ac.kr

Abstract. In this paper, we propose fast finite field and elliptic curve (EC) algorithms useful for embedding cryptographic functions on high performance device such that most instructions take just one cycle. In such case, the integer multiplications and additions have the same computational cost so that the computational cost analyses that were previously done in traditional manner may be invalid and in some cases the new algorithms should be introduced for fast computation. In our implementation, column major method for field multiplication and BP inversion algorithm are used for fast field arithmetic, and mixed coordinates method is used for efficient EC exponentiation. We give here analyses on various algorithms that are useful for implementing EC exponentiation on CalmRISC microcontroller with MAC2424 coprocessor, as well as new exact analyses on BP (Bailey-Paar) inversion algorithm and EC exponentiation. Using techniques shown in this paper, we implemented EC exponentiation for various coordinate systems and the best result took 122ms, assuming 50ns clock cycle.

1 Introduction

Since Koblitz[8] and Miller[11] first introduced elliptic curve cryptography (ECC), many works[7,10,2] have shown that ECC can be very efficiently embedded into restricted hardware such as smart cards. During the past few years, most people believed that elliptic curve defined over $GF(2^m)$ was the only useful one for hardware implementation since it can be implemented with only simple bit operations. $GF(p)$ and $GF(p^m)$ were popular in computer software implementation but they were not in hardware implementation because a math coprocessor is required for its implementation in smart cards and it significantly increases the cost.

However ECC is not restricted to smart cards. There can be many hardware applications that already have a fast microcontroller with math coprocessor. One such application is a portable MP3 player, it needs a high performance microcontroller which supports fast integer multiplication and division to decode MP3 data and it also needs cryptographic services to prevent unauthorized copy of

Ç.K. Koç and C. Paar (Eds.): CHES 2000, LNCS 1965, pp. 57–70, 2000.

MP3 files. In this case, $GF(p)$ or $GF(p^m)$ is more likely the better than $GF(2^m)$, since they can utilize the fast multiplication and division instructions. $GF(p)$ and $GF(p^m)$ both are good choices for that kind of application, but $GF(p^m)$ seems to be a better choice because there is no need to implement complex multiple precision routines and it utilizes the full capability of the microcontroller. Not only is it easy to implement but also more efficient, since there are no carry propagation and the inversion in $GF(p^m)$ is far more efficient than that in $GF(p)$. Many works [10,2,1] have shown that $GF(p^m)$ is very suitable choice for computer software implementation.

We have implemented EC over $GF(p^{10})$ in CalmRISC microcontroller with MAC2424 coprocessor. CalmRISC is a very fast 8-bit RISC microcontroller and MAC2424 is a high performance math coprocessor that can compute 24-bit signed multiplication just in one cycle and that provides efficient division step instruction. Since integer multiplication and division are the critical operations in $GF(p^m)$, such devices provide the best platform for implementing EC defined over $GF(p^m)$.

This paper focuses on implementing EC defined over $GF(p^m)$ in CalmRISC microcontroller with MAC2424 math coprocessor. In particular, we have used $GF(p^{10})$, satisfying OEF (Optimal Extension Field [2]) conditions, where $p = 2^{16} - 165 =$ `0xff5b` and the irreducible polynomial being $f(x) = x^{10} - 2$.

2 Processor Features

2.1 CalmRISC Microcontroller

CalmRISC is Samsung's 8-bit low power RISC microcontroller that follows Harvard style. Both instruction and data can be fetched simultaneously without causing a stall using separate paths for memory access. CalmRISC has a 3-stage pipeline:

1. Instruction Fetch (IF)
2. Instruction Decode/Data Memory Access (ID/MEM)
3. Execution/Writeback (EXE/WB)

The first stage (or cycle) is IF, where the instruction pointed to by the program counter is read into the instruction register (IR). The second stage is ID/MEM, where the fetched instruction (stored in IR) is decoded and the data memory access is performed, if necessary. The final stage is Execution and Writeback stage (EXE/WB), where the required ALU operation is executed and the result is written back into the destination registers. Since CalmRISC instructions are pipelined, the next instruction fetch is not postponed until the current instruction is completely finished, but it is performed immediately after the current instruction fetch is done.

Most of CalmRISC instructions are 1-word instruction, while branch instructions such as long "call" and "jump" instructions are a 2-word instruction. Thus the number of clocks per instruction (CPI) is 1 except for long branches, which take 2 clock cycles per instruction.

2.2 MAC2424 Math Coprocessor

MAC2424 is a 24-bit high performance fixed-point DSP coprocessor for CalmRISC microcontroller. Main datapaths are constructed to 24-bit width, but it can also perform 16-bit data processing efficiently in 16-bit operation mode.

There are two modes of operation in MAC2424: 24-bit mode operation and 16-bit mode operation.

24-Bit Mode Operation.

- Signed fractional/integer 24 x 24-bit multiplication in single cycle
- 24 x 24-bit multiplication and 52-bit accumulation in single cycle
- 24-bit arithmetic operation
- Two 48-bit multiplier accumulator with 4-bit guard
- Two 32K x 24-bit data memory spaces

16-Bit Mode Operation.

- Four-Quadrant fractional/integer 16 x 16-bit multiplication in single cycle
- 16 x 16-bit multiplication and 40-bit accumulation in single cycle
- 16-bit arithmetic operation with 8-bit guard
- Two 32-bit multiplier accumulator with 8-bit guard
- Two 32K x 16-bit data memory spaces

2.3 Programming Environment

CalmSHINE is a C compiler for CalmRISC and MAC2424. It also supports assembly language. Thus architecture specific low-level instructions (such as 24-bit by 24-bit multiplication and accumulation) can be utilized via assembly language. Non-architecture specific functions may be written in C language.

3 Finite Field Arithmetic

Optimization of finite field arithmetic is very critical to the overall performance of EC operations. In this section, we describe algorithms for implementing efficient finite field arithmetic. In our implementation, we use $GF(p^m)$ where $p =$`0xff5b` (16 bits), $m = 10$ and $f(x) = x^{10} - 2$ as an irreducible polynomial. Although CalmRISC supports 24-bit by 24-bit multiplication, it is signed multiplication and memory access is very inefficient in 24-bit mode due to the memory alignment. This is why we use 16-bit p.

3.1 Modular Reduction

Modular reduction is used very frequently and it is the bottleneck of the performance of finite field arithmetic. In most computer software implementations, the modular reduction using simple bit shifts and additions is a popular choice and provides a very good performance when p is a pseudo-Mersenne number. This is due to the fact that the division instruction is very slow for most hardware. However this is not the case for MAC2424, since every operation is simple and it takes only one cycle except long-branch operations. Thus modular reduction using division step instruction is desirable for MAC2424. Moreover it has an advantage that the intermediate values do not need to move around between the registers. During the division steps in MAC2424, the dividend and divisor keep their position until the division ends. In our implementation, the modular reduction by repeated division step instruction takes 39 cycles, while the modular reduction by bit shifts and additions takes 90 cycles.

3.2 Field Multiplication and Squaring

We considered three different algorithms for finite field multiplication, Karatsuba-Offman algorithm (KOA), column major method and row major method. First we consider KOA. KOA works by reducing the number of multiplications while increasing the number of cheap additions/subtractions by the recursively. In general, it gives about $10 \sim 20\%$ performance enhancement for most architecture. However the computational cost for multiplication and addition/subtraction is exactly the same for MAC2424, so reducing the number of multiplication with sacrificing the number of addition/subtraction does not help.

Row major method is just a schoolbook method, so we skip the description here. Column major method is described as follows. This is not a general method but it is for our specific case where $f(x)$ is binomial ($f(x) = x^m - \alpha$), and with this algorithm the polynomial reduction and multiplication can be done simultaneously.

Algorithm 1 (Column Major Multiplication).

Input: $A(x)$ *and* $B(x) \in GF(p^m)$
Output: $C(x) = A(x)B(x)$ *mod* $f(x)$ $(f(x) = x^m - \alpha)$
for $k = 0$ *to* $m - 1$ *do followings*

1. $z \leftarrow 0$, $i \leftarrow m - 1$, $j \leftarrow k + 1$
2. *while* $i > k$, $z \leftarrow z + a_i b_j$, $i \leftarrow i - 1$, $j \leftarrow j + 1$
3. $z \leftarrow z \cdot \alpha$, $j \leftarrow 0$
4. *while* $i \geq 0$, $z \leftarrow z + a_i b_j$, $i \leftarrow i - 1$, $j \leftarrow j + 1$
5. $c_k \leftarrow z$ *mod* p

Row major method and column major method both may be good choices because the required number of operation is both equal, but the row major

method has the disadvantage of storing full one intermediate row in a temporary memory. Moreover column major is preferable since MAC2424 can multiply-and-accumulate simultaneously in one cycle. So the column major method of field multiplication is definitely a better choice for MAC2424. Algorithm 1 uses m^2+m-1 multiplication instructions and m modular reductions with modulus p. Algorithm 1 can be similarly applied to field squaring, so $\frac{m(m+1)}{2}+m-1$ multiplication instructions and m modular reductions with modulus p are needed for field squaring. Modular reduction is performed only m times because product of two subfield elements can be safely accumulated multiple times in MAC2424's accumulator and α is small enough ($\alpha = 2$) that z in Algorithm 1 never overflows. This means we don't need to reduce the intermediate values, instead we need to reduce just the final values. In our implementation, field multiplication takes 723 cycles and field squaring takes 717 cycles. The ratio of field squaring to field multiplication is almost close to 0.9 in our case. This is due to the fact that the most of the time is taken in modular reduction and that MAC2424 can multiply very fast.

3.3 Field Inversion

There have been many research efforts on finite field inversion algorithm. Well-known algorithms are extended Euclidean algorithm, almost inversion algorithm, and their variants. The efficiency of a finite field inversion algorithm can be roughly measured by counting the subfield inversion it uses since the subfield inversion is the most time consuming job among the subfield arithmetic. Even for MAC2424, subfield inversion could not be done fast. It takes 670 cycles in our implementation. Among the various finite field inversion algorithms we consider IM (Inversion with Multiplication) [10] and BP [1] algorithms since only they require just one subfield inversion. Here we review the IM algorithm and the BP algorithm.

Algorithm 2 (IM Inversion Algorithm). *Initialize* $B \leftarrow 0$, $C \leftarrow 1$, $F \leftarrow f(x)$, $G \leftarrow A(x)$

1. *If* $deg(F) = 0$ *then* $B \leftarrow B \cdot (F_0^{-1} \bmod p)$, *return* B.
2. *If* $deg(F) < deg(G)$ *then exchange* F, B *with* G, C.
3. $j = deg(F) - deg(G)$
   (a) *If* $j \neq 0$ *do the followings.*
   $\alpha \leftarrow G_{deg(G)}^2 \bmod p$,
   $\beta \leftarrow F_{deg(F)} G_{deg(G)} \bmod p$,
   $\gamma \leftarrow G_{deg(G)} F_{deg(F)-1} - F_{deg(F)} G_{deg(G)-1} \bmod p$,
   $\{F,\ B\} \leftarrow \alpha\{F,\ B\} - (\beta x^j + \gamma x^{j-1})\{G,\ C\}$.
   (b) *If* $j = 0$ *do the followings.*
   $\{F,\ B\} \leftarrow G_{deg(F)}\{F,\ B\} - F_{deg(F)}\{G,\ C\}$
4. *Goto step 2.*

In Algorithm 2, capitalized variables represent the polynomial representation with indeterminate x, $deg(F)$ denotes the degree of polynomial F and F_i is the coefficient of x_i in F.

The BP algorithm is exponentiation-based inversion algorithm using the fact that A^p where $A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_0 \in GF(p^m)$ can be computed very efficiently only if the irreducible polynomial for $GF(p^m)$ is a binomial of the form $f(x) = x^m - \alpha$. The following equation shows that only $m-1$ subfield multiplications are required for p-th power of a field element. Note that $\alpha^{\lfloor pi/m \rfloor} \bmod p$ and $pi \bmod m$ $(i = 1, \ldots, m-1)$ should be pre-computed beforehand.

$$\begin{aligned} A(x)^p &= a_{m-1}x^{p(m-1)} + a_{m-2}x^{p(m-2)} + \cdots + a_0 \\ &= \sum_{i=0}^{m-1} a_i \alpha^{\lfloor pi/m \rfloor} x^{pi} \bmod m \end{aligned} \tag{1}$$

One might have noticed that the p-th power repeated by i times can be collapsed to one p^i-th power which have the same computational cost with p-th power. Now we have all apparatus for BP algorithm:

Algorithm 3 (BP Inversion Algorithm). *Computes* $A(x)^{-1}$ *as follows.* $A(x)^{-1} = (A(x)^r)^{-1}A(x)^{r-1}$ *mod* $f(x)$ *where* $r = \frac{p^m-1}{p-1} = 1+p+p^2+\cdots+p^{m-1}$

In Algorithm 3, $(A(x)^r)^{-1}$ is always a subfield inversion. In this algorithm, computing $A(x)^{r-1}$ is very critical to the performance of finite field inversion. Since $r-1$ is in a special form we can compute $A(x)^{r-1}$ efficiently by addition chain and p^i-th power. The efficient method was already shown in [1], but we want to show that the analysis shown in [1] and [10] is incorrect.

Table 1. Example of Computing A^{r-1} for $r = p + p^2 + \cdots + p^5$, $m = 6$

Computing A^{r-1} where $r = p + p^2 + \cdots + p^5$	
Our method	Bailey & Paar's method
$B \leftarrow A^p = A^{(10)}$	$B \leftarrow A^p = A^{(10)}$
$T_1 \leftarrow AB = A^{(11)}$	$T_1 \leftarrow BA = A^{(11)}$
$T_2 \leftarrow T_1^{p^2} = A^{(1100)}$	$B \leftarrow T_1^{p^2} = A^{(1100)}$
$T_2 \leftarrow T_1T_2 = A^{(1111)}$	$B \leftarrow BT_1 = A^{(1111)}$
$T_2 \leftarrow T_2^{p^2} = A^{(111100)}$	$B \leftarrow B^p = A^{(11110)}$
$B \leftarrow T_2B = A^{(111110)}$	$B \leftarrow BA = A^{(11111)}$
	$B \leftarrow B^p = A^{(111110)}$

The required number of p^i-th power in BP algorithm is not always $\lfloor log_2(m-1) \rfloor + H_W(m-1)$ where $H_W(\cdot)$ is Hamming weight. Instead, the number of p^i-th power is at least one less than this when m is even except for $m = 2$. Table 1 shows that one p^i-th power can be reduced for $m = 6$ considering this fact. In general, if m is even, the exact number of p^i-th power is:

$$\#p^i\text{-th power} = \begin{cases} \lfloor log_2(m-1) \rfloor + 1 & \text{if } m \text{ is 2's power} \\ \lfloor log_2(m-1) \rfloor + H_W(m) - 1 & \text{if } 2|m \text{ , not 2's power} \end{cases} \quad (2)$$

Table 2. Computational Cost Analysis for BP and IM Inversion Algorithm

Algorithm		#Multiplication	#Reduction	#Inv
IM ($f(x) = x^m - w$)		$3m^2 - m - 7$	$m^2 + 4m - 8$	1
BP	Original analysis	$t_1(m) \cdot (m^2 + 2m - 2) + 3m - 1$	$t_1(m) \cdot (3m - 2) + 2m$	1
	New analysis	$t_1(m) \cdot (m^2 + m - 1)$ $+t_2(m) \cdot (m-1) + 2m + 1$	$t_1(m) \cdot (2m - 1)$ $+t_2(m) \cdot (m-1) + m + 2$	1
	special case	„	$t_1(m) \cdot m$ $+t_2(m) \cdot (m-1) + m + 1$	1
	$t_1(m) = \lfloor log_2(m-1) \rfloor + H_W(m-1) - 1$ $t_2(m) = \begin{cases} \text{Eqn.(2)} & \text{if } m \text{ is even} \\ t_1(m) + 1 & \text{if } m \text{ is odd} \end{cases}$			

Table 2 shows the new exact analyses of BP algorithm and it is compared with IM algorithm. Note that we corrected minor counting errors that were shown in previous works[1,10]. We also show analyses for the 'special case' when the product of two subfield elements can be accumulated without overflow, and when α is small, that is our case. In that case, we can save $m - 1$ modular reduction in field multiplication and 1 modular reduction in final field multiplication that only computes the constant term of $A(x)^r$ from $A(x)^{r-1}$ and $A(x)$.

In Table 2, the term "multiplication" means general multiplication performed by the processor, not the field or subfield multiplication. According to Table 2, the complexity of BP looks greater than that of IM. However if m is not too large, BP can provide better performance. When $m = 10$, that is the specific case for our implementation, IM requires 283 multiplication and 132 modular reduction, and BP requires 493 multiplication and 87 modular reduction (note that one less number of p^i-th power is required for $m = 10$ than the analysis shown in Table 2). Recall that, for MAC2424 the number of modular reduction is more significant (costs much more) than that of multiplication. Hence we conclude that BP algorithm will perform much better in our case ($283 + 132 \times 39 = 5431$(IM) $> 493 + 87 \times 39 = 3886$(BP)). In addition, not only does the BP algorithm take fewer cycles but also it is simple to implement as looping or branching is not needed. More improvement is possible for BP algorithm when m has a factor of 2. In our specific case, p^i-th power is computed as follows.

Algorithm 4 (p-th Power Algorithm for Our Specific Case). $B(x) = A(x)^p$

1. *Array X is pre-computed as*
   $X = \{$`1, 0x5AC3, 0x7B13, 0x9EEF, 0x7E9E, 0xFF5A(= -1), 0xA498, 0x8448, 0x606C, 0x80BD`$\}$
2. *For $i = 0$ to 9*
   $B_i = B_{(ip \bmod m)} = A_i \cdot X_i \bmod$ `0xff5b`

Algorithm 4 requires 8 multiplications, 8 modular reductions and 1 subtraction. Note that multiplying -1 is equal to subtracting from p. The following is the pre-computed value X for p^2-th power and p^4-th power algorithm for our specific case, respectively. To compute p^2-th or p^4-th power we only need to substitute the X in Algorithm 4 with the following Xs.

For p^2-th power: $X = \{$`1, 0x7B13, 0x7E9E, 0xA498, 0x606C, 1, 0x7B13, 0x7E9E, 0xA498,0x606C`$\}$
For p^4-th power: $X = \{$`1, 0x7e9e, 0x606c, 0x7b13, 0xa498, 1, 0x7e9e, 0x606c, 0x7b98, 0xa498`$\}$

The above each pre-computed array X has two 1s, so only 8 multiplications and 8 modular reductions are needed to compute p^2-th or p^4-th power. And even more, $X_{[1-4]}$ is identical to $X_{[6-9]}$, thus the memory can be saved.

Now we are ready to construct the most efficient method to compute $A(x)^{r-1}$ for our specific case, which leads to the least number of field multiplications and fully utilizes the above facts. The following algorithm efficiently computes $A(x)^{r-1}$ and we used this in our actual implementation.

$$
\begin{array}{lll}
T_1 \leftarrow A^p & (T_1 = A^p) & \text{(exp)} \\
T_2 \leftarrow A^p \cdot A & (T_2 = A^{1+p}) & \text{(mul)} \\
T_3 \leftarrow T_2^{p^2} & (T_3 = A^{p^2+p^3}) & \text{(exp)} \\
T_3 \leftarrow T_3 \cdot T_2 & (T_3 = A^{1+p+p^2+p^3}) & \text{(mul)} \\
T_4 \leftarrow T_3^{p^4} & (T_4 = A^{p^4+p^5+p^6+p^7}) & \text{(exp)} \\
T_3 \leftarrow T_3 \cdot T_4 & (T_3 = A^{1+p+p^2+p^3+p^4+p^5+p^6+p^7}) & \text{(mul)} \\
T_3 \leftarrow T_3^{p^2} & (T_3 = A^{p^2+p^3+p^4+p^5+p^6+p^7+p^8+p^9}) & \text{(exp)} \\
T_2 \leftarrow T_3 \cdot T_1 & (T_2 = A^{p+p^2+p^3+p^4+p^5+p^6+p^7+p^8+p^9}) & \text{(mul)}
\end{array}
$$

As shown above $A(x)^{r-1}$ is done by 4 field multiplications and 4 p^i-th powers. As it can be seen, since m is even, the number of field multiplication is equal to that of p^i-th power.

3.4 Performance of Field Arithmetic

Table 3 shows our finite field $GF(p^{10})$ implementation results. The cycles for each functions were measured using Samsung's CalmSHINE compiler and all

finite field functions were written in assembly. All cycles include the functional overhead such as parameter loading and data movements when entering and leaving the functions. In Table 3, 'I/M' is the ratio of field inversion to the field multiplication and 'S/M' is the ratio of field squaring to the field multiplication.

Table 3. Finite Field Implementation Result

Operation	Required Cycles
Add	187
Sub	141
Mult	723
Square	667
Sub_Inv	670
Mod	39
Inversion	5378
I/M	7.4
S/M	0.9

4 Elliptic Curve Arithmetic

In this section we discuss the method of optimizing EC exponentiation using mixed coordinate system. We optimize the EC exponentiation by combining mixed coordinate system from [5] and the Lim-Hwang's method [10].

4.1 Signed Window Algorithm for EC Exponentiation

Signed window method is known to be the most efficient method for computing EC exponentiation (EC scalar multiplication) excluding the fixed base exponentiation algorithms. Let k be a positive integer, and suppose we want to compute kP where P is an arbitrary point on an elliptic curve. Then k can be expressed as follows.

$$k = 2^{k_0}(2^{k_1}(\cdots 2^{k_{v-1}}(W[v] + W[v-1]) + W[v-2]\cdots) + W[0]) \tag{3}$$

where $W[i]$ is odd, $-2^w + 1 \leq W[i] \leq 2^w - 1$ and $w \leq k_i$. To compute EC exponentiation with signed window method, first pre-compute $P_i = iP$ ($i = \pm 1, \pm 3, \cdots, \pm(2^w - 1)$) and then evaluate the following equation using the pre-computed values.

$$kP = 2^{k_0}(2^{k_1}(\cdots 2^{k_{v-1}}(2^{k_v} P_{W[v]} + P_{W[v-1]}) + P_{W[v-2]}\cdots) + P_{W[0]}) \tag{4}$$

However pre-computation is not needed for $-2^w + 1 \leq W[i] \leq -1$ since negating EC point can be done easily, that is computing $P_{W[i]} = -P_{-W[i]}$ takes negligible time. It is also possible to implement an EC subtraction function to get rid of the redundant EC negating time using a small portion of additional program memory.

The first k_v doublings, in case $W[v] < 2^{w-1}$, can be more efficiently computed [5]. There have been an analysis on this in [5], however it is incorrect and we want to correct it here.

First we need to consider the probability that the bit size of $W[v]$ equals j. This can be easily computed and the following equation shows it.

$$Pr(|W[v]| = j) = \begin{cases} \frac{1}{2^{w-1}} & \text{if } j = 1 \\ \frac{1}{2^{w-j+1}} & \text{if } 2 \leq j \leq w \end{cases} \tag{5}$$

Use the fact that the above modification reduces $w + 1 - j$ doublings and increases 1 addition, and that we do not apply the above modification when $j = w$. Then can be easily verified that the average number of doublings reduced is $\frac{3}{2} - \frac{1}{2^{w-1}}$ and the average number of addition increased is $\frac{1}{2}$. Note that the average value of k_v is $w + 2$, so $w + 2$ doublings are needed in an average case if we don't use the above modification.

4.2 Mixed Coordinates System

We use mixed coordinates system to speed up computation of EC exponentiations. For a given rational integer k and an elliptic curve point P, we can evaluate the EC exponentiation kP by the following steps.

$$\begin{aligned} &T_0 = P_{W[v]} \\ &T_{i+1} = 2^{k_{v-i}} T_i + P_{W[v-i-1]} \text{ for } i = 0, 1, \ldots, v-1 \\ &kP = 2^{k_0} T_v \end{aligned} \tag{6}$$

The EC exponentiation kP is computed by repeating basic step $T_{i+1} = 2^{k_{v-i}} T_i + P_{W[v-i-1]}$, which is equal to $T_{i+1} = 2T' + P_{W[v-i-1]}$ where $T' = 2^{k_{v-i}-1} T_i$. If we represent the elliptic curve points $(T_i, 2T', P_{W[v-i-1]})$ as coordinates (C_1, C_2, C_3), the computational cost for a basic step is

$$(k_{v-i} - 1) \cdot t(2C_1) + t(2C_1 = C_2) + t(C_2 + C_3 = C_1).$$

In this paper, we denote affine coordinate as A [6,12], projective coordinate P [9,6], Jacobian coordinate J [3], modified Jacobian coordinate J^m [10] and Chudnovsky-Jacobian coordinate J^C [3]. Note that we use a different Modified Jacobian coordinate system. The Modified Jacobian coordinate shown in [10] is better because it reduces one field addition/subtraction in EC addition.

Let us now discuss suitable coordinate systems for C_1, C_2 and C_3. Since doublings in C_1 are repeated most frequently, we should choose C_1 such that $t(2C_1)$ is the smallest, thus we select J^m as C_1.

Since pre-computation $P_{W[i]}$ is done during online time in signed window method, i.e. the resulting values are not saved in auxiliary memory for another exponentiation, the pre-computation time is included in total elapsed time. Thus we should select coordinate C_3 suitable to compute the values $P_{w[i]}$. Cohen *et al.*[5] proposed to use either affine coordinate or Chudnovsky-Jacobian coordinate as C_3 and to select one by comparing $t(J^C + J^C)$ and $t(A + A)$. However, since the ratio I/M is relatively small, we chose $C_3 = A$. Then there are two pre-computation methods. First, we can compute $Pi = iP(i = 1, 3, 5, \ldots, 2^w - 1)$ in affine coordinate by simple method by repeating $P_{i+2} = P_i + P'$ for $i = 1, 3, 5, \ldots, 2^w - 3$ where $P_1 = P$ and $P' = P + P$. Here, the total computational cost is $t(2A) + (2^{w-1} - 1) \cdot t(A + A) = 2^{w-1}\mathrm{I} + 2^w\mathrm{M} + (2^{w-1} + 1)\mathrm{S}$. To reduce the number of inversion in $F(p^m)$, we can apply 'Montgomery trick of simultaneous inversion [4]' with sacrificing the number of multiplications and squares. The total cost in that case is $w\mathrm{I} + (5 \cdot 2^{w-1} + 2w - 10)\mathrm{M} + (2^{w-1} + 2w - 3)\mathrm{S}$. Table 4 shows the expected computational cost for these two methods. In Table 4, the computational costs for pre-computation in case of $C_3 = J^c$ were also shown for comparison.

Table 4. Computational Cost for Various Pre-computation Methods

Method	Computational cost
Affine(simple)	8I + 16M+9S = 83.3M
Affine(Mont. trick)	4I + 38M + 13S = 79.3M
Chudnovsky-Jacobian 1	77M + 26S = 100.4M
Chudnovsky-Jacobian 2	I + 55M + 23S = 83.1M

In Table 4, Montgomery's trick is shown to be the best choice. However we didn't use the Montgomery trick, since online pre-computation time is just a very small part of EC exponentiation, and it does not significantly improves the EC exponentiation time. In addition, the Montgomery's method requires much more program memory than the simple method without giving much improvement in performance. We chose to use simple method in online pre-computation.

Let us discuss suitable coordinate for C_2. Since we selected modified Jacobian coordinate and affine coordinate for C_1 and C_3 respectively, coordinate for C_2 should minimize $(k_{v-i} - 1) \cdot t(2J^m) + t(2J^m = C_2) + t(C_2 + A = J^m)$, that is, it should minimize $t(2J^m = C_2) + t(C_2 + A = J^m)$. Although there are 5 candidates for C_2, Table 5 shows computational amounts to compute a basic step (Eqn. 6) using 3 candidates of least cost. In Table 5, we assumed window size $w = 4$.

In Table 5, the 1-bit gap between the two neighboring diminished windows is considered to be the worst case (i.e. $k_i = w + 1$ for $i = 1, 2, \ldots, v$), and the 2-bit

Table 5. Candidates for Best Mixed Coordinates System and their Analyses

Coordinate	Cost	Worst case ($k_{v-1}=5$)	Average case ($k_{v-1}=6$)
(J^m, J, A)	$(7.6k_{v-1}+12.5)$M	50.5M	58.1M
(J^m, J^c, A)	$(7.6k_{v-1}+12.5)$M	50.5M	58.1M
(J^m, J^m, A)	$(7.6k_{v-1}+13.5)$M	51.5M	59.1M

gap is considered to be the average case (i.e. $k_i = w+2$ for $i = 1, 2, \ldots, v$). According to Table 5, we can select either Jacobian coordinate(J) or Chudnovsky-Jacobian coordinate(J^c) for C_2. Since Chudnovsky-Jacobian coordinate uses 2 more finite field $F(p^m)$ elements than Jacobian, it is inefficient in storage. Thus we select Jacobian coordinate for C_2. Consequently, for $(C_1, C_2, C_3) = (J^m, J, A)$, $w = 4$ and $|k| = 160$, we can compute an EC exponentiation kP with following computational cost in average.

$$\begin{aligned} t(2A) + (2^{w-1}-1)\cdot t(A+A) + \frac{1}{2}\cdot t(A+A=J^m) \\ + (w + \frac{1}{2^{w-1}} - \frac{1}{2})\cdot t(2J^m) + t(J+A=J^m) \\ + \{\frac{|k|-w+1-(\frac{1}{2})^{w-1})}{w+2} - 1\}\cdot \left\{ \begin{array}{l} (w+1)\cdot t(2J^m) \\ +t(2J^m=J) + t(J+A=J^m) \end{array} \right\} \\ \approx 8I + 849.7M + 763.7S \approx \ 1596\text{M} \quad (7) \end{aligned}$$

In worst case, we can compute kP with the following cost.

$$\begin{aligned} t(2A) + (2^{w-1}-1)\cdot t(A+A) + t(A+A=J^m) + t(2J^m=J) + t(J+A=J^m) \\ + \{\frac{|k|-1}{w+1} - 1\}\cdot \{w\cdot t(2J^m) + t(2J^m=J) + t(J+A=J^m)\} \\ \approx 8I + 895.4M + 792S \approx \ 1667\text{M} \quad (8) \end{aligned}$$

5 Implementation Results

We implemented elliptic curve exponentiation in CalmRISC with MAC2424 coprocessor using all algorithms shown in previous sections. All finite field functions were written in assembly language since time critical low-level instructions cannot be programmed in high-level language, and all elliptic curve functions were written in C language on top of the finite field functions. Table 6 shows our implementation of elliptic curve exponentiation in various coordinate systems. Note that the result shown in Table 6 was measured using CalmSHINE C compiler. CalmSHINE compiler measures the clock cycle for each function exactly, however it can be done only in 'debug build mode' and CalmSHINE compiler

does very poor code optimization in 'debug build mode'. This is why the result shown in Table 6 is slower than what is expected. In real implementation with optimized codes, it will perform much better. Referring to Table 6, mixed coordinates is the best with almost 10% of improvement over fastest single coordinate system (Modified Jacobian).

Table 6. Implementation Result of Elliptic Curve Exponentiation

EC Exponentiation Result		
Coordinate	Cycles	Time
(J^m, J, A)	2448265	122ms
(A, A, A)	3632657	182ms
(J^m, J^m, J^m)	2711543	135ms

6 Conclusions

In this paper, we proposed optimized algorithms for implementing EC in CalmRISC with MAC2424 math coprocessor, in which all instructions take just one clock cycle, and we showed implementation results and full analyses on their performances. We also gave new exact alalyses on BP inversion algorithm and EC exponentiation. In our implementation, we used column major method for field multiplication and slightly improved BP algorithm for field inversion. Mixed coordinates using Lim-Hwang's Modified Jacobian coordinate was applied for for efficient EC exponentiation. Our implementation of EC exponentiation took about 122ms (assuming one cycle takes 50ns), which is about 10% of improvement over single coordinate system. This result can be much better in real implementation with CalmSHINE's optimized compile mode. Although the algorithms shown in this paper is focused on our specific case, it can be easily applied to other environments where all basic arithmetic instructions have the same computational cost.

Acknowledgement

This project was supported mainly by Samsung Electronics Co. Ltd., and partially by Brain Korea 21 and Com^{2}MaC-KOSEF.

References

1. Bailey, D., Paar, C.: Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography, To appear in *Journal of Cryptology* (Available at `http://ece.wpi.edu/People/faculty/cxp.html`)
2. Bailey, D. V. and Paar, C.: Optimal extension field for fast arithmetic in public key algorithms, *Advances in Cryptology-Crypto'*98, Lecture Notes in Computer Science, Vol 1462. Springer-Verlag, (1998), 472-485.
3. Chudnovsky, D. V. and Chudnovsky, G. V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests, *Advances in Applied Math.*, Vol. 7. (1986), 385-434.
4. Cohen, H.: *A course in computational algebraic number theory*, Graduate Texts in Math., Vol. 138. Springer-Verlag, (1993).
5. Cohen, H., Miyaji, A. and Ono, T.: Efficient Elliptic Curve Exponentiation Using Mixed Coordinates, *Advances in Cryptology-Asiacrypt'98*, Lecture Notes in Computer Science, Vol. 1514. Springer-Verlag, (1998), 50-65.
6. IEEE P1363: Standard Specifications for Public Key Cryptography, Working Draft 12, Nov. (1999).
7. Itoh, K., Takenaka, M., Torll, N., Temma, S. and Kurihara, Y.: Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201, *Cryptographic Hardware and Embedded Systems*'99, Lecture Notes in Computer Science, Vol. 1717. Springer Verlag, (1999), 61-72.
8. Koblitz, N.:, Elliptic Curve Cryptosystems, *Math. Comp.*, Vol. 48. pp. (1987), 203-209.
9. Koyama, K. and Tsuruoka, Y.: Speeding up elliptic cryptosystems by using a signed binary window method, *Advances in Cryptology-Proceedings of Crypto*'92, Lecture Notes in Computer Science, Vol. 740. Springer-Verlag, (1993), 345-357.
10. Lim, C. H. and Hwang, H. S.: Fast Implementation of Elliptic Curve Arithmetic in $GF(p^n)$, *Public Key Cryptography*, Lecture Notes in Computer Science, Vol. 1751. Springer-Verlag, (2000), 405-421.
11. Miller, V. S.: Use of Elliptic Curves in Cryptography, *Advances in Cryptology-Proceedings of Crypto* '85, Lecture Notes in Computer Science, Vol. 218. Springer-Verlag, (1986), 417-426.
12. Silverman, J. H.: *The Arithmetic of Elliptic Curves*, GTM 106. Springer-Verlag, New York (1986).