Time Complexities of Multiple-precision Modular Operations and Related Ratios

Shenghui Su [1, 3] and Ping Luo [2]

[1] College of Computers, Nanjing University of Aero. and Astro., Nanjing 211106, PRC
[2] Software School, Tsinghua University, Beijing 100084, PRC
[3] Laboratory of Computational Complexity, BFID Corporation, Fuzhou 350207, PRC

Abstract: Modular arithmetic used for cryptography includes modular adding, modular subtracting, modular multiplying, modular inverting, modular exponentiating etc. In this paper, the authors well analyze the bit complexity of a bitwise modular operation and the time complexity of a non-bitwise modular operation. Besides discuss the clock cycles for one bytewise modular operation utilizing directives from the ATmel 8-bit AVR instruction set. Last, reveal that the ratio of derivate numbers of clock cycles for two modular operations under different modulus lengths is almost a constant.

Keywords: Multiple-precision modular arithmetic, Montgomery multiplication algorithm, Time complexity, Ratio of derivate numbers of clock cycles, Bytewise, 8-bit AVR instruction set

1 Introduction

The time complexity of an algorithm is also called the running time or time cost. It is the number of required arithmetical operations for the algorithm to complete a specified task in the worst case [1][2]. Sometimes, an average case will be considered if the worst case is obviously irrational. As a basic unit of measurement, the time complexity in bits (shortly bit complexity) is usually adopted [3][4]. The bit complexity is namely the number of required bit operations to complete the task.

A non-bitwise (namely bytewise or wordwise) modular arithmetic operation is just an algorithm. Its time complexity is essentially independent of assembly programming languages or processor clock cycles, but it is usually measured with the number of assembly directives or clock cycles[5][6].

In cryptographic systems, multiple-precision (or super-long) modular arithmetic involving modular adding, modular subtracting, modular multiplying, modular inverting, modular exponentiating etc is widely employed [7][8]. The running time (in bits, directives, or cycles) of a modular operation has to be concerned. Occasionally the ratio of fit numbers of clock cycles for two operations is also concerned.

Throughout the paper, unless otherwise specified, there lies $\underset{\sim}{b}$ = 256 or 65536, $\lg P$ (= $\lfloor \log_2 P \rfloor + 1$) denotes the bit-length of a prime modulus P with being divisible by 8 or 16, n does the byte-length or word-length of P, gcd(X, Y) signifies the greatest common divisor of two integers X and Y, $X \equiv Y$ (mod P) means that X equals Y modulo P, and $|X|$ represents the absolute value of a number X.

2 Running Time of a Multiple-precision Modular Operation

Let X ($< P$) and Y ($< P$) be two arbitrary nonnegative integers, and modular arithmetic be performed over the prime field $\mathbb{F}_P$. Assume that P, X, and Y in radix 2 or $\underset{\sim}{b}$ representation are $(p_{(\lg P)-1}\ldots p_0)_2$, $(x_{(\lg P)-1}\ldots x_0)_2$, and $(y_{(\lg P)-1}\ldots y_0)_2$, or $(\dot{p}_{n-1}\ldots \dot{p}_0)_{\underset{\sim}{b}}$, $(\dot{x}_{n-1}\ldots \dot{x}_0)_{\underset{\sim}{b}}$, and $(\dot{y}_{n-1}\ldots \dot{y}_0)_{\underset{\sim}{b}}$, where $n = \lg P / 8$ (or / 16).

Notice that P, X, and Y in radix 2 representation may be converted into ones in $\underset{\sim}{b}$ representation, and the left of the most significant bit (or digit) of representation of X or Y is filled with 0-bits (or -digits).

2.1 Running Time of a Modular Addition

2.1.1 Bit Complexity of a Bitwise Modular Addition

Assume that bitwise operators on adding are available. Let $S = (s_{(\lg P)}\ldots s_0)_2$ be a mid variable.

Apparently, our computing $(s_{(\lg P)}\ldots s_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2 + (y_{(\lg P)-1}\ldots y_0)_2$ requires $\lg P$ bit operations (with carry). If $(X+Y) > P$, we need further seeking the difference $(s_{(\lg P)}\ldots s_0)_2 - (p_{(\lg P)-1}\ldots p_0)_2$, which also requires $\lg P$ bit operations (likewise with carry).

Observe an example of $(X + Y) \bmod P$.

Let $\lg P = 8$.

Set 11111011 (= 251) to $(p_7 \ldots p_0)_2$, 11110101 to $(x_7 \ldots x_0)_2$, and 10111101 to $(y_7 \ldots y_0)_2$. Then,

$X + Y \equiv (X + Y) - P \equiv ((x_7 \ldots x_0)_2 + (y_7 \ldots y_0)_2) - (p_7 \ldots p_0)_2$

$\equiv (11110101 + 10111101) - 11111011$

$\equiv 110110010 - 11111011 \equiv 10110111 \pmod{11111011}$.

Therefore, the bit complexity of a bitwise modular addition is $2\lg P$ (= $O(\lg P)$) [4], where big-O is the asymptotic notation, and it popularly represents an asymptotic upper bound disregarding factors, coefficients, or lower order terms [9][10].

2.1.2 Time Complexity of a Non-bitwise Modular Addition

Assume that 8- or 16-bit single-precision adding (or subtracting on occasion) directives are available. Let $S = (\dot{s}_n \ldots \dot{s}_0)_{\underline{b}}$ be a mid variable.

Algo 14.7 in Reference 4 manifests that computing $(\dot{s}_n \ldots \dot{s}_0)_{\underline{b}} \leftarrow (\dot{x}_{n-1} \ldots \dot{x}_0)_{\underline{b}} + (\dot{y}_{n-1} \ldots \dot{y}_0)_{\underline{b}}$ requires n single-precision adding directives (with carry) [4]. If $(X + Y) > P$, we will need searching the difference $(\dot{s}_n \ldots \dot{s}_0)_{\underline{b}} - (\dot{p}_{n-1} \ldots \dot{p}_0)_{\underline{b}}$, which requires n single-precision subtracting directives (likewise with carry) according to Algo 14.9 [4].

Analogous to a bitwise modular addition, the time complexity of a non-bitwise modular addition is $2n$ (= $O(n)$) single-precision adding (or subtracting) directives [4].

Notice that during the time complexity evaluation, a subtraction may be regarded as an addition because a subtracting directive consumes the same number of clock cycles as an adding directive [11].

2.2 Running Time of a Modular Subtraction

2.2.1 Bit Complexity of a Bitwise Modular Subtraction

Assume that bitwise operators on subtracting are available. Let $S = (s_{(\lg P)} \ldots s_0)_2$ be a mid variable.

Clearly, we first need computing $(s_{(\lg P)} \ldots s_0)_2 \leftarrow (x_{(\lg P)-1} \ldots x_0)_2 + (p_{(\lg P)-1} \ldots p_0)_2$ if $X < Y$, which requires $\lg P$ bit operations (with carry), and further, seeking the difference $(s_{(\lg P)} \ldots s_0)_2 - (y_{(\lg P)-1} \ldots y_0)_2$ also requires $\lg P$ bit operations (likewise with carry).

Observe an example of $(X - Y) \bmod P$.

Still let $\lg P = 8$.

Set 11111011 (= 251) to $(p_7 \ldots p_0)_2$, 01010101 to $(x_7 \ldots x_0)_2$, and 10101010 to $(y_7 \ldots y_0)_2$ (notice that there is $X < Y$). Then,

$X - Y \equiv (X + P) - Y \equiv ((x_7 \ldots x_0)_2 + (p_7 \ldots p_0)_2) - (y_7 \ldots y_0)_2$

$\equiv (01010101 + 11111011) - 10101010$

$\equiv 101010000 - 10101010 \equiv 10100110 \pmod{11111011}$.

Therefore, the bit complexity of a bitwise modular subtraction is $2\lg P$ (= $O(\lg P)$) [4].

2.2.2 Time Complexity of a Non-bitwise Modular Subtraction

Assume that 8- or 16-bit single-precision subtracting (or adding on occasion) directives are available. Let $S = (\dot{s}_n \ldots \dot{s}_0)_{\underline{b}}$ be a mid variable.

It is well understood that we first need computing $(\dot{s}_n \ldots \dot{s}_0)_{\underline{b}} \leftarrow (\dot{x}_{n-1} \ldots \dot{x}_0)_{\underline{b}} + (\dot{p}_{n-1} \ldots \dot{p}_0)_{\underline{b}}$ if $X < Y$, which requires n single-precision adding directives (with carry) according to Algo 14.7 [4], and further, searching the difference $(\dot{s}_n \ldots \dot{s}_0)_{\underline{b}} - (\dot{y}_{n-1} \ldots \dot{y}_0)_{\underline{b}}$ requires n single-precision subtracting directives (likewise with carry) according to Algo 14.9 [4].

Analogous to a bitwise modular subtraction, the time complexity of a non-bitwise modular subtraction is $2n$ (= $O(n)$) single-precision subtracting (or adding) directives [4].

2.3 Running Time of a Modular Multiplication

2.3.1 Bit Complexity of a Bitwise Modular Multiplication

Assume that a bitwise addition and subtraction are available. Let $A = (a_{2(\lg P)-1}\ldots a_0)_2$ be a mid variable.

Owing to $y_i = 0$ or 1, the bitwise multiplication $(a_{2(\lg P)-1}\ldots a_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2 \times (y_{(\lg P)-1}\ldots y_0)_2$ may be fulfilled through $\lg P$ iterations of the bitwise addition (with shift), which requires $(\lg P)^2$ bit operations. Further, the bitwise reduction $(a_{2(\lg P)-1}\ldots a_0)_2 \bmod (p_{(\lg P)-1}\ldots p_0)_2$ may be fulfilled through $\lg P$ iterations of the bitwise subtraction (with shift), which also requires $(\lg P)^2$ bit operations.

Observe an example of $(X \times Y) \bmod P$.

Still let $\lg P = 8$.

Set 11111011 (= 251) to $(p_7\ldots p_0)_2$, 11001100 to $(x_7\ldots x_0)_2$, and 00110011 to $(y_7\ldots y_0)_2$. Then,

$$X \times Y \equiv (x_7\ldots x_0)_2 \times (y_7\ldots y_0)_2$$
$$\equiv 11001100 \times 00110011 \equiv 0010100010100100 \equiv 01110001 \pmod{11111011}.$$

Hence, the bit complexity of a bitwise modular multiplication is $2(\lg P)^2$ (= $O((\lg P)^2)$) [4]. Notice that trivial judging and shifting operations are customarily neglected during the time cost evaluation [4].

2.3.2 Time Complexity of a Non-bitwise Modular Multiplication

Assume that 8- or 16-bit single-precision multiplying, adding, and subtracting directives are available. Let $A = (\dot{a}_{2n-1}\ldots\dot{a}_n\ldots\dot{a}_0)_{\underline{b}}$ be a resultant variable.

Ordinarily, first we need to get $(\dot{a}_{2n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \times (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$, which requires $n^2 (= O(n^2))$ single-precision multiplying directives and $3n^2$ (= $O(n^2)$) single-precision adding directives according to Algo 14.12 in which the value of a carry digit is at most $(\underline{b} - 1)$ but not 1 [4], and then we need to reduce $(\dot{a}_{2n-1}\ldots\dot{a}_0)_{\underline{b}} \bmod (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$, which demands single-precision dividing directives.

However, there is no single-precision dividing directive in some CPU assembly instruction sets (for example the ATmel 8-bit AVR instruction set [11]); thus the Montgomery multiplication algorithm without explicit dividing is designed for the modular product of two multiple-precision integers [4].

Let a mid variable U in radix $\underline{b}$ representation be $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}}$.

Similar to Algo 14.36 [4], the Montgomery multiplication algorithm in non-bitwise (shortly Mont(X, $Y \bmod P$) or Mont(X, Y)) is described as follows:

```
INPUT:   $X = (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, $Y = (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$, $P = (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$ with $0 \le X, Y < P$;
         $\acute{R} = (\check{r}_{n-1}\ldots\check{r}_0)_{\underline{b}} = \underline{b}^n \bmod P$ with $\gcd(P, \underline{b}) = 1$, and $\acute{P} = -P^{-1} \bmod \underline{b}$.
OUTPUT: $XY\acute{R}^{-1} \bmod P$.
S1: Set $(\dot{a}_n\ldots\dot{a}_0)_{\underline{b}} \leftarrow 0$. (Note that $A = (\dot{a}_n\ldots\dot{a}_0)_{\underline{b}}$.)
S2: For $i$ from 0 to $(n-1)$ do:
    S2.1: Get $\ddot{u}_i \leftarrow (\dot{a}_0 + \dot{x}_i\dot{y}_0)\acute{P} \bmod \underline{b}$.
    S2.2: Get $(\dot{a}_n\ldots\dot{a}_0)_{\underline{b}} \leftarrow ((\dot{a}_n\ldots\dot{a}_0)_{\underline{b}} + \dot{x}_i(\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}} + \ddot{u}_i(\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}) / \underline{b}$.
S3: If $(\dot{a}_n\ldots\dot{a}_0)_{\underline{b}} \ge (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$ then $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\dot{a}_n\ldots\dot{a}_0)_{\underline{b}} - (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$.
S4: Return ($A$).
```

Through two iterations of the Montgomery multiplication algorithm (namely Mont(X, $Y \bmod P$) and Mont($XY\acute{R}^{-1}$, $\acute{R}^2 \bmod P$), where $\acute{R}$ (= $\underline{b}^n \bmod P$) and $\acute{R}^2 \bmod P$ may be calculated in advance), we can obtain the product $A = XY \bmod P$ (namely $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} = (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \times (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}} \bmod (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$), which indicates the time complexity of the $XY \bmod P$ operation is $4n(n+1)$ (= $O(n^2)$) single-precision multiplying directives and $4n(n+2)$ (= $O(n^2)$) single-precision adding (or subtracting) directives (concretely including $2n$ subtracting directives) [4].

2.3.3 Time Complexity of a Non-bitwise Modular Squaring Operation

Under the circumstances of there being no single-precision dividing directive, computing $X^2 \bmod P$ demands three steps. Let $A = (\dot{a}_{2n-1}\ldots\dot{a}_n\ldots\dot{a}_0)_{\underline{b}}$ be a resultant variable.

First, to compute $(\dot{a}_{2n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \times (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, which requires $n(n+1)/2\,(= O(n^2))$ single-precision multiplying directives and $2n+4n(n-1)/2 = 2n^2\,(= O(n^2))$ single-precision adding directives according to Algo 14.16 [4].

Next, to reduce A (= $(\dot{a}_{2n-1}\ldots\dot{a}_0)_{\underline{b}}$) $2n$-digits in length to $A\acute{R}^{-1}$ mod P n-digits in length (where $\acute{R} = \underline{b}^n$ mod P) through the Montgomery reduction algorithm without explicit dividing, which requires $n(n+1)$ $(=O(n^2))$ single-precision multiplying directives and $n(n+2)\,(=O(n^2))$ single-precision adding (or subtracting) directives (concretely including n subtracting directives) according to Algo 14.32 [4].

Last, to extract A (= $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}$) from $A\acute{R}^{-1}$ through the Montgomery multiplication (Mont($A\acute{R}^{-1}$, $\acute{R}^2$ mod P)), which requires $2n(n + 1)$ (= $O(n^2)$) single-precision multiplying directives and $2n(n + 2)$ (= $O(n^2)$) single-precision adding (or subtracting) directives according to Algo 14.36.

Hence, the time complexity of a non-bitwise modular squaring operation is $3.5n(n+1)(=O(n^2))$ single-precision multiplying directives and $n(5n+6)(=O(n^2))$ single-precision adding (or subtracting) directives.

2.4 Running Time of a GCD(X, Y) Operation with $Y \le X \le P$

2.4.1 Bit Complexity of the Euclidean Algorithm in Bitwise

Let $R = X - YQ$, where R (= $(r_{(\lg P)-1}\ldots r_0)_2$) is a remainder, and Q (= $(q_{(\lg P)-1}\ldots q_0)_2$) is a quotient.

We can gain Q and R through multiple iterations of a bitwise subtraction instead of a bitwise division.

Similar to Algo 2.104 [4], the Euclidean algorithm in bitwise (shortly EAB) is described as follows:

```
INPUT:    Two positive integers $X$ and $Y$ with $Y \le X \le P$.
OUTPUT: The greatest common divisor of $X$ and $Y$.
S1: While $(y_{(\lg P)-1}\ldots y_0)_2 \ne 0$ do:
    S1.1: Get $(r_{(\lg P)-1}\ldots r_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2$ mod $(y_{(\lg P)-1}\ldots y_0)_2$;
    S1.2: Let $(x_{(\lg P)-1}\ldots x_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2$;
    S1.3: Let $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (r_{(\lg P)-1}\ldots r_0)_2$.
S2: Return $((x_{(\lg P)-1}\ldots x_0)_2)$.
```

Notice that the most significant bits of X, Y, and R will gradually move rightward.

Now, we analyze the running time $Ŧ$ of the Euclidean algorithm in bitwise.

Let k be the number of iterations of S1.1-S1.3, then k is less than or equal to $5(0.301\lg X) \approx 1.5\lg X$[12].

Due to (running time of X mod Y) = (valid bit-length of Y) $\times$ (valid bit-length of Q), we have

$$\begin{aligned}
Ŧ &= \textstyle\sum_{i=1}^{k} \lg Y_i\,(\lg X_i - \lg Y_i + 1)\\
&\le \textstyle\sum_{i=1}^{k} \lg X\,(\lg X_i - \lg Y_i + 1)\\
&= \lg X\,(\,\lg X_1 - \lg Y_k + k)\\
&\le \lg X\,(\,\lg X + k)\\
&\le \lg X\,(\,\lg X + 1.5\lg X)\\
&= 2.5(\lg X)^2 \le 2.5(\lg P)^2,
\end{aligned}$$

where $X_1 = X$, $Y_1 = Y$, $X_2 = Y_1$, $X_3 = Y_2$, …, and $X_k = Y_{k-1}$ [13].

Hence, the bit complexity of the Euclidean algorithm in bitwise is $2.5(\lg P)^2$ (= $O((\lg P)^2)$) [4].

Note that the Euclidean algorithm in bytewise demands single-precision dividing directives; thus it will be inapplicable to 8-bit AVR assembly source programs.

2.4.2 Time Complexity of the Binary GCD Algorithm in Non-bitwise

Let mid variables G and T in radix 2 or $\underline{b}$ representation be separately $(g_{(\lg P)-1}\ldots g_0)_2$ and $(t_{(\lg P)-1}\ldots t_0)_2$, or $(\dot{g}_{n-1}\ldots\dot{g}_0)_{\underline{b}}$ and $(\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}}$.

Due to $\lg P$ being divisible by 8 or 16, transforming $(x_{(\lg P)-1}\ldots x_0)_2$ into $(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$ or $(y_{(\lg P)-1}\ldots y_0)_2$ into $(\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$ will be very convenient.

Similar to Algo 14.54 without explicit dividing [4], the binary GCD algorithm in non-bitwise (shortly BGANB) is described as follows:

```
INPUT:    Two positive integers $X$ and $Y$ with $Y \le X \le P$.
OUTPUT: The greatest common divisor of $X$ and $Y$.
S1: Set $(\dot{g}_{n-1}\ldots\dot{g}_0)_{\underline{b}} \leftarrow 1$.
S2: While both $(x_{(\lg P)-1}\ldots x_0)_2$ and $(y_{(\lg P)-1}\ldots y_0)_2$ are even do:
    S2.1: Let $(x_{(\lg P)-1}\ldots x_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2 / 2$;
    S2.2: Let $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2 / 2$;
    S2.3: Let $(g_{(\lg P)-1}\ldots g_0)_2 \leftarrow (g_{(\lg P)-1}\ldots g_0)_2 \times 2$.
S3: While $(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \ne 0$ do:
    S3.1: While $(x_{(\lg P)-1}\ldots x_0)_2$ is even do: $(x_{(\lg P)-1}\ldots x_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2 / 2$;
    S3.2: While $(y_{(\lg P)-1}\ldots y_0)_2$ is even do: $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2 / 2$;
    S3.3: Get $(\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}} \leftarrow |(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} - (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}|$,
          and let $(t_{(\lg P)-1}\ldots t_0)_2 \leftarrow (t_{(\lg P)-1}\ldots t_0)_2 / 2$;
    S3.4: If $(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \ge (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$ then $(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}} \leftarrow (\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}}$;
          else $(\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}} \leftarrow (\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}}$.
S4: Get $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2 \times (g_{(\lg P)-1}\ldots g_0)_2$, and return $(Y)$.
```

Notice that an 8- or 16-bit single-precision subtraction is employed at S3.3, and the most significant bits of X, Y, and T will gradually move rightward along with increase of iterations.

It is not difficult to comprehend that the number of valid bits of X or Y will decrease by (at least) 1 after at most two iterations of S3.1-S3.4, and thus BGANB takes at most $2\lg P$ such iterations, which tells us that BGANB requires $2n\lg P$ ($= O(n\lg P)$) 8- or 16-bit subtracting directives [4].

2.5 Running Time of a Modular Inversion

2.5.1 Bit Complexity of the Extended Euclidean Algorithm in Bitwise

By the greatest common divisor theorem [12], for the coprime P and Y ($< P$), it holds that $PS + YT = V$ ($= \gcd(P, Y)) = 1$, where S and T are two integers, and T is called the multiplicative inverse of Y.

Coherently, let S, T, and V in radix 2 representation be $(s_{(\lg P)-1}\ldots s_0)_2$, $(t_{(\lg P)-1}\ldots t_0)_2$, and $(v_{(\lg P)-1}\ldots v_0)_2$.

Again let mid variables A, B, C, D, Q, and R (a remainder) in radix 2 representation be $(a_{(\lg P)-1}\ldots a_0)_2$, $(b_{(\lg P)-1}\ldots b_0)_2$, $(c_{(\lg P)-1}\ldots c_0)_2$, $(d_{(\lg P)-1}\ldots d_0)_2$, $(q_{(\lg P)-1}\ldots q_0)_2$, and $(r_{(\lg P)-1}\ldots r_0)_2$.

Referring to Algo 2.107 [4], we characterize the extended Euclidean algorithm in bitwise (shortly EEAB) for seeking the gcd of X and Y or the multiplicative inverse of Y in $\mathbb{F}_P$ as follows:

```
INPUT:    Two nonnegative integers $X$ (or $P$) and $Y$ with $Y \le X \le P$.
OUTPUT: $V = \gcd(X, Y)$ and integers $S$, $T$ (or an inverse) satisfying $XS + YT = V$.
S1: If $(y_{(\lg P)-1}\ldots y_0)_2 = 0$ then let $(v_{(\lg P)-1}\ldots v_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2$,
        set $(s_{(\lg P)-1}\ldots s_0)_2 \leftarrow 1$, $(t_{(\lg P)-1}\ldots t_0)_2 \leftarrow 0$, and return $(V, S, T)$.
S2: Set $(b_{(\lg P)-1}\ldots b_0)_2 \leftarrow 1$, $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow 0$, $(d_{(\lg P)-1}\ldots d_0)_2 \leftarrow 0$, $(c_{(\lg P)-1}\ldots c_0)_2 \leftarrow 1$.
S3: While $(y_{(\lg P)-1}\ldots y_0)_2 > 0$ do:
    S3.1: Get $(q_{(\lg P)-1}\ldots q_0)_2$ and $(r_{(\lg P)-1}\ldots r_0)_2$ by $(x_{(\lg P)-1}\ldots x_0)_2 / (y_{(\lg P)-1}\ldots y_0)_2$,
          $(s_{(\lg P)-1}\ldots s_0)_2 \leftarrow (b_{(\lg P)-1}\ldots b_0)_2 - (q_{(\lg P)-1}\ldots q_0)_2 \times (a_{(\lg P)-1}\ldots a_0)_2$,
          $(t_{(\lg P)-1}\ldots t_0)_2 \leftarrow (d_{(\lg P)-1}\ldots d_0)_2 - (q_{(\lg P)-1}\ldots q_0)_2 \times (c_{(\lg P)-1}\ldots c_0)_2$.
    S3.2: Let $(x_{(\lg P)-1}\ldots x_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2$, $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (r_{(\lg P)-1}\ldots r_0)_2$,
          $(b_{(\lg P)-1}\ldots b_0)_2 \leftarrow (a_{(\lg P)-1}\ldots a_0)_2$, $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow (s_{(\lg P)-1}\ldots s_0)_2$,
          $(d_{(\lg P)-1}\ldots d_0)_2 \leftarrow (c_{(\lg P)-1}\ldots c_0)_2$, $(c_{(\lg P)-1}\ldots c_0)_2 \leftarrow (t_{(\lg P)-1}\ldots t_0)_2$.
S4: Let $(v_{(\lg P)-1}\ldots v_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2$, $(s_{(\lg P)-1}\ldots s_0)_2 \leftarrow (b_{(\lg P)-1}\ldots b_0)_2$,
    $(t_{(\lg P)-1}\ldots t_0)_2 \leftarrow (d_{(\lg P)-1}\ldots d_0)_2$, and return $(V, S, T)$.
```

We see that at S3.1 of EEAB, one bitwise division for a quotient with a remainder and two bitwise multiplications are required, which can be implemented separately through iterations of the bitwise subtraction (with shift) and iterations of the bitwise addition (with shift).

Paralleling EAB, EEAB will expend $3(2.5(\lg P)^2) = 7.5(\lg P)^2$ ($= O((\lg P)^2)$) bit operations after at most $1.5\lg P$ iterations of S3.1-S3.2 [4].

Hence, the bit complexity of EEAB is $7.5(\lg P)^2$ which is three times as large as EAB. However, they both have the same magnitude of $O((\lg P)^2)$ [4].

Note that the extended Euclidean algorithm in bytewise demands single-precision dividing directives; thus it will be inapplicable to 8-bit AVR assembly source programs.

2.5.2 Time Complexity of the Binary Extended GCD Algorithm in Non-bitwise

Algo 14.61 is called the binary extended GCD algorithm utilizing 8- or 16-bit adding and subtracting directives for searching the integers S, T and V such that $XS + YT = V$ ($= \gcd(X, Y)$) with $Y \le X \le P$ [4]. Especially, when $X = P$ (namely $V = 1$), T is the multiplicative inverse of Y in $\mathbb{F}_P$.

Let the variables S, T and V in radix 2 or $\underline{b}$ representation be respectively expressed as $(s_{(\lg P)-1}\ldots s_0)_2$, $(t_{(\lg P)-1}\ldots t_0)_2$, and $(v_{(\lg P)-1}\ldots v_0)_2$, or $(\dot{s}_{n-1}\ldots\dot{s}_0)_{\underline{b}}$, $(\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}}$, and $(\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}}$, where $n = \lg P / 8$ (or / 16).

Again let mid variables A, B, C, D, G, and U in radix 2 or $\underline{b}$ representation be separately $(a_{(\lg P)-1}\ldots a_0)_2$, $(b_{(\lg P)-1}\ldots b_0)_2$, $(c_{(\lg P)-1}\ldots c_0)_2$, $(d_{(\lg P)-1}\ldots d_0)_2$, $(g_{(\lg P)-1}\ldots g_0)_2$, and $(u_{(\lg P)-1}\ldots u_0)_2$, or $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}$, $(\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}}$, $(\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}}$, $(\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}}$, $(\dot{g}_{n-1}\ldots\dot{g}_0)_{\underline{b}}$, and $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}}$.

Similar to Algo 14.61 without explicit dividing and multiplying [4], the binary extended GCD algorithm in non-bitwise (shortly BEGANB) for searching the gcd or inverse is described as follows:

```
INPUT:   Two positive integers $X$ (or $P$) and $Y$ with $Y \le X \le P$.
OUTPUT: Integers $S$, $T$ (or an inverse), and $V$ such that $XS + YT = V$, where $V = \gcd(X, Y)$.
S1: Set $(\dot{g}_{n-1}\ldots\dot{g}_0)_{\underline{b}} \leftarrow 1$.
S2: While $(x_{(\lg P)-1}\ldots x_0)_2$ and $(y_{(\lg P)-1}\ldots y_0)_2$ are both even do:
    S2.1: Let $(x_{(\lg P)-1}\ldots x_0)_2 \leftarrow (x_{(\lg P)-1}\ldots x_0)_2 / 2$;
    S2.2: Let $(y_{(\lg P)-1}\ldots y_0)_2 \leftarrow (y_{(\lg P)-1}\ldots y_0)_2 / 2$;
    S2.3: Let $(g_{(\lg P)-1}\ldots g_0)_2 \leftarrow (g_{(\lg P)-1}\ldots g_0)_2 \times 2$.
S3: Let $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}} \leftarrow (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, $(\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}} \leftarrow (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$;
    Set $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow 1$, $(\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}} \leftarrow 0$, $(\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}} \leftarrow 0$, $(\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}} \leftarrow 1$.
S4: While $(u_{(\lg P)-1}\ldots u_0)_2$ is even do:
    S4.1: Let $(u_{(\lg P)-1}\ldots u_0)_2 \leftarrow (u_{(\lg P)-1}\ldots u_0)_2 / 2$;
    S4.2: If $(a_{(\lg P)-1}\ldots a_0)_2 \equiv (b_{(\lg P)-1}\ldots b_0)_2 \equiv 0 \pmod 2$
            then $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow (a_{(\lg P)-1}\ldots a_0)_2 / 2$, $(b_{(\lg P)-1}\ldots b_0)_2 \leftarrow (b_{(\lg P)-1}\ldots b_0)_2 / 2$;
            else $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} + (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$, $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow (a_{(\lg P)-1}\ldots a_0)_2 / 2$,
                $(\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}} \leftarrow (\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}} - (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, $(b_{(\lg P)-1}\ldots b_0)_2 \leftarrow (b_{(\lg P)-1}\ldots b_0)_2 / 2$.
S5: While $(v_{(\lg P)-1}\ldots v_0)_2$ is even do:
    S5.1: Let $(v_{(\lg P)-1}\ldots v_0)_2 \leftarrow (v_{(\lg P)-1}\ldots v_0)_2 / 2$;
    S5.2: If $(c_{(\lg P)-1}\ldots c_0)_2 \equiv (d_{(\lg P)-1}\ldots d_0)_2 \equiv 0 \pmod 2$
            then $(c_{(\lg P)-1}\ldots c_0)_2 \leftarrow (c_{(\lg P)-1}\ldots c_0)_2 / 2$, $(d_{(\lg P)-1}\ldots d_0)_2 \leftarrow (d_{(\lg P)-1}\ldots d_0)_2 / 2$;
            else $(\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}} \leftarrow (\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}} + (\dot{y}_{n-1}\ldots\dot{y}_0)_{\underline{b}}$, $(c_{(\lg P)-1}\ldots c_0)_2 \leftarrow (c_{(\lg P)-1}\ldots c_0)_2 / 2$,
                $(\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}} \leftarrow (\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}} - (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, $(d_{(\lg P)-1}\ldots d_0)_2 \leftarrow (d_{(\lg P)-1}\ldots d_0)_2 / 2$.
S6: If $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}} \ge (\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}}$ then $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}} \leftarrow (\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}} - (\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}}$,
        $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} - (\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}}$, $(\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}} \leftarrow (\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}} - (\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}}$;
    else $(\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}} \leftarrow (\ddot{v}_{n-1}\ldots\ddot{v}_0)_{\underline{b}} - (\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}}$,
        $(\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}} \leftarrow (\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}} - (\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}$, $(\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}} \leftarrow (\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}} - (\dot{b}_{n-1}\ldots\dot{b}_0)_{\underline{b}}$.
S7: If $(\ddot{u}_{n-1}\ldots\ddot{u}_0)_{\underline{b}} = 0$ then $(\dot{s}_{n-1}\ldots\dot{s}_0)_{\underline{b}} \leftarrow (\dot{c}_{n-1}\ldots\dot{c}_0)_{\underline{b}}$, $(\dot{t}_{n-1}\ldots\dot{t}_0)_{\underline{b}} \leftarrow (\dot{d}_{n-1}\ldots\dot{d}_0)_{\underline{b}}$,
        $(v_{(\lg P)-1}\ldots v_0)_2 \leftarrow (v_{(\lg P)-1}\ldots v_0)_2 \times (g_{(\lg P)-1}\ldots g_0)_2$, and return $(S, T, V)$;
    else go to S4.
```

We observe S4 and S5 of BEGANB.

When U is even, the commands $A \leftarrow A + Y$ and $B \leftarrow B - X$ are always executed in the worst case. When V is even, the commands $C \leftarrow C + Y$ and $D \leftarrow D - X$ are also always executed in the worst case.

Notice that it is impossibly that U at S4 and V at S5 are simultaneously even.

Next observe S6. There always lie 3 executed subtracting commands whether U is bigger than V or not.

Hence, a single iteration of S4-S7 requires 5 multiple-precision adding (or subtracting) operations. Analogous to BGANB, BEGANB takes at most $2\lg P$ iterations. Further, $2\lg P$ iterations require $10\lg P$ adding (or subtracting) operations, which indicates that BEGANB requires $10n\lg P$ (= $O(n\lg P)$) 8- or 16-bit single-precision adding (or subtracting) directives [4].

2.6 Running Time of a Modular Exponentiation

2.6.1 Bit Complexity of the Left-to-right Modular Exponentiation Algorithm in Bitwise

Suppose that the resultant variable A in radix 2 representation is $(a_{(\lg P)-1}\ldots a_0)_2$.

Similar to Algo 14.79 [4], the left-to-right modular exponentiation algorithm in bitwise (shortly LRMEAB) is described as follows:

```
INPUT:   Two positive integers $X$ and $Y$ with $X < P$ and $Y < P$.
OUTPUT: The power $A = X^Y \bmod P$.
S1: Set $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow 1$.
S2: For $i$ from $(\lg P)-1$ down to 0 do:
    S2.1: Get $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow (a_{(\lg P)-1}\ldots a_0)_2 \times (a_{(\lg P)-1}\ldots a_0)_2 \bmod (p_{(\lg P)-1}\ldots p_0)_2$;
    S2.2: If $y_i = 1$ then $(a_{(\lg P)-1}\ldots a_0)_2 \leftarrow (a_{(\lg P)-1}\ldots a_0)_2 \times (x_{(\lg P)-1}\ldots x_0)_2 \bmod (p_{(\lg P)-1}\ldots p_0)_2$.
S3: Return $((a_{(\lg P)-1}\ldots a_0)_2)$.
```

We see that S2.1 is a bitwise modular multiplication, which requires $2(\lg P)^2$ bit operations.

Likewise, we see that S2.2 is a bitwise modular multiplication, which requires $2(\lg P)^2$ bit operations in the worst case.

Consequently, the bit complexity of LRMEAB is $4(\lg P)^3$ (= $O((\lg P)^3)$) [4].

2.6.2 Time Complexity of the Montgomery Exponentiation Algorithm in Non-bitwise

Algo 14.94 is called the Montgomery exponentiation algorithm without explicit dividing operation, and suitable for 8-bit AVR assembly programming [4].

As they come, X and P in radix $\underline{b}$ (= 256 or 65536) representation are respectively $(\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$ and $(\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$, and Y in radix 2 representation is $(y_{(\lg P)-1}\ldots y_0)_2$.

Let $\ddot{X} = (\bar{x}_{n-1}\ldots\bar{x}_0)_{\underline{b}}$ be a mid variable, and $A = (\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}$ be a resultant variable. Similar to Algo 14.94 [4], the Montgomery exponentiation algorithm in non-bitwise is described as follows:

```
INPUT:    $X = (\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}$, $P = (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$, and $Y = (y_{(\lg P)-1}\ldots y_0)_2$ with $y_{(\lg P)-1} = 1$;
          $\acute{R} = (\check{r}_{n-1}\ldots\check{r}_0)_{\underline{b}} = \underline{b}^n \bmod P$, $\ddot{R} = (\tilde{r}_{n-1}\ldots\tilde{r}_0)_{\underline{b}} = \acute{R}^2 \bmod P$, and $\acute{P} = -P^{-1} \bmod \underline{b}$.
OUTPUT: $X^Y \bmod P$.
S1: Let $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow (\check{r}_{n-1}\ldots\check{r}_0)_{\underline{b}} \bmod (\dot{p}_{n-1}\ldots\dot{p}_0)_{\underline{b}}$,
    get $(\bar{x}_{n-1}\ldots\bar{x}_0)_{\underline{b}} \leftarrow \mathrm{Mont}((\dot{x}_{n-1}\ldots\dot{x}_0)_{\underline{b}}, (\tilde{r}_{n-1}\ldots\tilde{r}_0)_{\underline{b}})$.
S2: For $i$ from $(\lg P)-1$ down to 0 do:
    S2.1: Get $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow \mathrm{Mont}((\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}, (\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}})$.
    S2.2: If $y_i = 1$ then $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow \mathrm{Mont}((\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}, (\bar{x}_{n-1}\ldots\bar{x}_0)_{\underline{b}})$.
S3: Get $(\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}} \leftarrow \mathrm{Mont}((\dot{a}_{n-1}\ldots\dot{a}_0)_{\underline{b}}, 1)$.
S4: Return $(A)$.
```

We see that at S1, one Montgomery multiplication is required, at S2, $1.5(\lg P)$ Montgomery multiplications are required (notice that the probability of S2.2 being executed is obviously $1/2$), and at S3, one special Montgomery multiplication is required.

Then, at S1, single-precision $2n(n + 1)$ multiplying directives and $2n(n + 2)$ adding (or subtracting) directives are required, at S2, single-precision $3n(n + 1)(\lg P)$ multiplying directives and $3n(n + 2)(\lg P)$ adding (or subtracting) directives are required, and at S3, single-precision $n(n + 1)$ multiplying directives and rational $n(n + 2)$ adding (or subtracting) directives are required.

Consequently, the time complexity of the Montgomery exponentiation algorithm is $3n(n+1)(1+\lg P)$ $(= O(n^2 \lg P))$ single-precision multiplying directives and $3n(n+2)(1+\lg P)(=O(n^2 \lg P))$ single-precision adding (or subtracting) directives.

3 Clock Cycles for a Bytewise Multiple-precision Modular Operation

Assume that multiple-precision modular arithmetic is performed on the ATmel 8-bit AVR processor, which means that bytewise multiple-precision modular arithmetic is operated. Notice that there is no single-precision dividing directive in the ATmel 8-bit AVR instruction set.

Again assume that the bit-length of a cryptographic multiple-precision modulus is 160, 192, 224, or 256.

We will only consider clock cycles taken by adding (or subtracting) and multiplying directives, and ignore clock cycles taken by trivial judging, shifting, moving, loading, storing etc directives. The demand for every trivial directive of an 8-bit AVR assembly source program may be formulated.

The ATmel 8-bit AVR instruction set shows that one adding directive (with carry) consumes 1 clock cycle, one subtracting directive (with carry) consumes 1 clock cycle, and one unsigned multiplying directive consumes 2 clock cycles (note that in cryptosystems with moduli all integers are nonnegative).

When $\underline{b} = 256$ and $\lg P = 160, 192, 224$, or 256, we have $n = 20, 24, 28$, or 32.

3.1 Clock Cycles for a Bytewise Modular Addition

It is known from Sect 2.1.2 that the running time of a bytewise modular addition is $2n$ single-precision adding (or subtracting) directives.

	Add (or Sub)		**Mul**		**Total Clock Cycles**
	Directives	Clock Cycles	Directives	Clock Cycles	
$\lg P = 160$ ($n = 20$)	40	40	0	0	40
$\lg P = 192$ ($n = 24$)	48	48	0	0	48
$\lg P = 224$ ($n = 28$)	56	56	0	0	56
$\lg P = 256$ ($n = 32$)	64	64	0	0	64

Table 1: Number of Clock Cycles for a Bytewise Modular Addition

Note that the practical number of clock cycles for a bytewise modular addition will be bigger than the table number when trivial directives are considered.

3.2 Clock Cycles for a Bytewise Modular Subtraction

It is known from Sect 2.2.2 that the running time of a bytewise modular subtraction is $2n$ single-precision adding (or subtracting) directives.

	Add (or Sub)		**Mul**		**Total Clock Cycles**
	Directives	Clock Cycles	Directives	Clock Cycles	
$\lg P = 160$ ($n = 20$)	40	40	0	0	40
$\lg P = 192$ ($n = 24$)	48	48	0	0	48
$\lg P = 224$ ($n = 28$)	56	56	0	0	56
$\lg P = 256$ ($n = 32$)	64	64	0	0	64

Table 2: Number of Clock Cycles for a Bytewise Modular Subtraction

Note that the practical number of clock cycles for a bytewise modular subtraction will be bigger than the table number when trivial directives are considered.

3.3 Clock Cycles for a Bytewise Modular Multiplication

It is known from Sect 2.3.2 that the running time of a bytewise modular multiplication utilizing the Montgomery multiplication algorithm without explicit dividing is $4n(n + 2)$ single-precision adding (or subtracting) directives and $4n(n + 1)$ single-precision multiplying directives.

	Add (or Sub)		**Mul**		**Total Clock Cycles**
	Directives	Clock Cycles	Directives	Clock Cycles	
$\lg P = 160$ ($n = 20$)	1760	1760	1680	3360	5120
$\lg P = 192$ ($n = 24$)	2496	2496	2400	4800	7296
$\lg P = 224$ ($n = 28$)	3360	3360	3248	6496	9856
$\lg P = 256$ ($n = 32$)	4352	4352	4224	8448	12800

Table 3: Number of Clock Cycles for a Bytewise Modular Multiplication

Note that the practical number of clock cycles for a bytewise modular multiplication will be bigger than the table number when trivial directives are considered.

3.4 Clock Cycles for a Bytewise Modular Squaring Operation

It is known from Sect 2.3.3 that the running time of a bytewise modular squaring operation which utilizes respectively the Montgomery reduction algorithm and the homonymous multiplication algorithm without explicit dividing is $n(5n + 6)$ single-precision adding (or subtracting) directives and $3.5n(n+1)$ single-precision multiplying directives.

	Add (or Sub)		**Mul**		**Total Clock Cycles**
	Directives	Clock Cycles	Directives	Clock Cycles	
$\lg P = 160$ ($n = 20$)	2120	2120	1470	2940	5060
$\lg P = 192$ ($n = 24$)	3024	3024	2100	4200	7224
$\lg P = 224$ ($n = 28$)	4088	4088	2842	5684	9772
$\lg P = 256$ ($n = 32$)	5312	5312	3696	7392	12704

Table 4: Number of Clock Cycles for a Bytewise Modular Squaring Operation

Note that the practical number of clock cycles for a bytewise modular squaring operation will be bigger than the table number when trivial directives are considered.

3.5 Clock Cycles for the Binary GCD Algorithm in Bytewise

It is known from Sect 2.4.2 that the running time of the binary GCD algorithm in bytewise without explicit dividing is $2n \lg P$ single-precision adding (or subtracting) directives.

	Add (or Sub)		**Mul**		**Total Clock Cycles**
	Directives	Clock Cycles	Directives	Clock Cycles	
$\lg P = 160$ ($n = 20$)	6400	6400	0	0	6400
$\lg P = 192$ ($n = 24$)	9216	9216	0	0	9216
$\lg P = 224$ ($n = 28$)	12544	12544	0	0	12544
$\lg P = 256$ ($n = 32$)	16384	16384	0	0	16384

Table 5: Number of Clock Cycles for the Binary GCD Algorithm in Bytewise

Note that the practical number of clock cycles for the binary GCD algorithm in bytewise will be bigger than the table number when trivial directives are considered.

3.6 Clock Cycles for the Binary Extended GCD Algorithm in Bytewise

It is known from Sect 2.5.2 that the running time of the binary extended GCD algorithm in bytewise (for searching one gcd or multiplicative inverse) without explicit dividing and multiplying is $10n \lg P$ single-precision adding (or subtracting) directives.

	Add (or Sub)		Mul		Total Clock Cycles
	Directives	Clock Cycles	Directives	Clock Cycles	
lg$P$ = 160 (n = 20)	32000	32000	0	0	32000
lg$P$ = 192 (n = 24)	46080	46080	0	0	46080
lg$P$ = 224 (n = 28)	62720	62720	0	0	62720
lg$P$ = 256 (n = 32)	81920	81920	0	0	81920

Table 6: Number of Clock Cycles for the Binary Extended GCD Algorithm in Bytewise

Note that the practical number of clock cycles for the binary extended GCD algorithm in bytewise will be bigger than the table number when trivial directives are considered.

3.7 Clock Cycles for the Montgomery Exponentiation Algorithm in Bytewise

It is known from Sect 2.6.2 that the time complexity of the Montgomery exponentiation algorithm in bytewise without explicit dividing is $3n(n + 2)(1 + \lg P)$ single-precision adding (or subtracting) directives and $3n(n + 1)(1 + \lg P)$ single-precision multiplying directives.

	Add (or Sub)		Mul		Total Clock Cycles
	Directives	Clock Cycles	Directives	Clock Cycles	
lg$P$ = 160 (n = 20)	212520	212520	202860	405720	618240
lg$P$ = 192 (n = 24)	361296	361296	347400	694800	1056096
lg$P$ = 224 (n = 28)	567000	567000	548100	1096200	1663200
lg$P$ = 256 (n = 32)	838848	838848	814176	1628352	2467200

Table 7: Number of Clock Cycles for the Montgomery Exponentiation Algorithm in Bytewise

Note that the practical number of clock cycles for the Montgomery exponentiation algorithm in bytewise will be bigger than the table number when trivial directives are considered.

4 Ratio of Derivate Numbers of Clock Cycles for Two Modular Operations

Still assume that multiple-precision modular arithmetic is done on the ATmel 8-bit AVR processor.

Still again assume that the bit-length of a multiple-precision modulus is 160, 192, 224, or 256.

Since the demand for the single-precision adding, subtracting, and multiplying directives in a byte-wise modular operation may be formulated, the number of clock cycles for the modular operation may be formulated, and further, the ratio of derivate numbers of clock cycles for two modular operations under different modulus lengths may be computed.

Let MADD symbolize a modular addition, MMUL do a modular multiplication, MINV do a modular inversion (namely the binary extended GCD algorithm), and MEXP do a modular exponentiation.

4.1 Ratio of Square of Number of Cycles for MADD to Number of Cycles for MMUL

Sect 3.1 tells us that the number of clock cycles for MADD (Ca) is $2n$.

Sect 3.3 tells us that the number of clock cycles for MMUL (Cm) is $4n(n+2)+4n(n+1)2 = 12n^2+8n$.

Then, $R_{(C\mathrm{a})^2:C\mathrm{m}} = (C\mathrm{a})^2/C\mathrm{m} = 4n^2/(12n^2+8n) = 1/(3+2/n)$.

When lg$P$ = 160 (viz. n = 20), $R_{(C\mathrm{a})^2:C\mathrm{m}}$ = 1 / (3 + 2 / 20) = 1 / 3.100000 = 0.322581.

When lg$P$ = 192 (viz. n = 24), $R_{(C\mathrm{a})^2:C\mathrm{m}}$ = 1 / (3 + 2 / 24) = 1 / 3.083333 = 0.324324.

When lg$P$ = 224 (viz. n = 28), $R_{(C\mathrm{a})^2:C\mathrm{m}}$ = 1 / (3 + 2 / 28) = 1 / 3.071429 = 0.325581.

When lg$P$ = 256 (viz. n = 32), $R_{(C\mathrm{a})^2:C\mathrm{m}}$ = 1 / (3 + 2 / 32) = 1 / 3.062500 = 0.326531.

The average of the ratios = (0.322581 + 0.324324 + 0.325581 + 0.326531) / 4 = 1.299017 / 4 = 0.324754.

The average of |the errors| = (0.002173 + 0.000430 + 0.000827 + 0.001777) / 4 = 0.005207 / 4 = 0.001302.

The weight of average of |the errors| = 0.001302 / 0.324754 = 0.004009 = 0.4009 % < 1 %.

Thus, the ratio of the square of number of clock cycles for MADD to the number of clock cycles for MMUL is almost one constant, which hints that the ratio is determined by two the highest order terms.

4.2 Ratio of Number of Cycles for MINV to Number of Cycles for MMUL

Sect 3.3 tells us that the number of clock cycles for MMUL (Cm) is $4n(n+2)+4n(n+1)2 = 12n^2+8n$.

Sect 3.6 tells us that the number of clock cycles for MINV (Ci) is $10n\lg P = 10n(8n) = 80n^2$.

Then, $R_{C\mathrm{i}:C\mathrm{m}} = C\mathrm{i}/C\mathrm{m} = 80n^2/(12n^2+8n) = 20/(3+2/n)$.

When $\lg P$ = 160 (viz. n = 20), $R_{C\mathrm{i}:C\mathrm{m}}$ = 20 / (3 + 2 / 20) = 20 / 3.100000 = 6.451613.

When $\lg P$ = 192 (viz. n = 24), $R_{C\mathrm{i}:C\mathrm{m}}$ = 20 / (3 + 2 / 24) = 20 / 3.083333 = 6.486487.

When $\lg P$ = 224 (viz. n = 28), $R_{C\mathrm{i}:C\mathrm{m}}$ = 20 / (3 + 2 / 28) = 20 / 3.071429 = 6.511627.

When $\lg P$ = 256 (viz. n = 32), $R_{C\mathrm{i}:C\mathrm{m}}$ = 20 / (3 + 2 / 32) = 20 / 3.062500 = 6.530612.

The average of the ratios = (6.451613 + 6.486487 + 6.511627 + 6.530612) / 4 = 25.980339 / 4 = 6.495085.

The average of |the errors| = (0.043472 + 0.008598 + 0.016542 + 0.035527) / 4 = 0.104139 / 4 = 0.026035.

The weight of average of |the errors| = 0.026035 / 6.495085 = 0.004008 = 0.4008 % < 1 %.

Thus, the ratio of the number of clock cycles for MINV to the number of clock cycles for MMUL is almost one constant, which implies that the ratio is determined by two the highest order terms.

4.3 Ratio of Cube of Number of Cycles for MADD to Number of Cycles for MEXP

Sect 3.1 shows that the number of clock cycles for MADD (Ca) is $2n$.

Sect 3.7 shows that the number of clock cycles for MEXP (Ce) is $3n(n+2)(1+\lg P)+3n(n+1)(1+\lg P)2$.

Then, $R_{(C\mathrm{a})^3:C\mathrm{e}} = (C\mathrm{a})^3/C\mathrm{e} = 8n^3/(72n^3+105n^2+12n) = 8/(72+105/n+12/n^2)$.

When $\lg P$ = 160 (viz. n = 20), $R_{(C\mathrm{a})^3:C\mathrm{e}}$ = 8 / (72 + 105 / 20 + 12 / 400) = 8 / 77.280000 = 0.103519.

When $\lg P$ = 192 (viz. n = 24), $R_{(C\mathrm{a})^3:C\mathrm{e}}$ = 8 / (72 + 105 / 24 + 12 / 576) = 8 / 76.395833 = 0.104718.

When $\lg P$ = 224 (viz. n = 28), $R_{(C\mathrm{a})^3:C\mathrm{e}}$ = 8 / (72 + 105 / 28 + 12 / 784) = 8 / 75.765306 = 0.105589.

When $\lg P$ = 256 (viz. n = 32), $R_{(C\mathrm{a})^3:C\mathrm{e}}$ = 8 / (72 + 105 / 32 + 12/1024) = 8 / 75.292969 = 0.106252.

The average of the ratios = (0.103519 + 0.104718 + 0.105589 + 0.106252) / 4 = 0.420078 / 4 = 0.105019.

The average of |the errors| = (0.001500 + 0.000301 + 0.000570 + 0.001233) / 4 = 0.003604 / 4 = 0.000901.

The weight of average of |the errors| = 0.000901 / 0.105019 = 0.008579 = 0.8579 % < 1 %.

Thus, the ratio of the cube of number of clock cycles for MADD to the number of clock cycles for MEXP is almost one constant, which indicates that the ratio is decided by two the highest order terms.

4.4 Ratio of (3/2)-th Root of Number of Cycles for MINV to Number of Cycles for MEXP

Sect 3.6 shows that the number of clock cycles for MINV (Ci) is $10n\lg P = 10n(8n) = 80n^2$.

Sect 3.7 shows that the number of clock cycles for MEXP (Ce) is $3n(n+2)(1+\lg P)+3n(n+1)(1+\lg P)2$ $= 72n^3 + 105n^2 + 12n$.

Then, $R_{(C\mathrm{i})^{3/2}:C\mathrm{e}} = (C\mathrm{i})^{3/2}/C\mathrm{e} = (80n^2)^{3/2}/(72n^3+105n^2+12n) = 715.54/(72+105/n+12/n^2)$.

As $\lg P$ = 160 (viz. n = 20), $R_{(C\mathrm{i})^{3/2}:C\mathrm{e}}$ = 715.54 / (72 + 105 / 20 + 12 / 400) = 715.54 / 77.280000 = 9.259058.

As $\lg P$ = 192 (viz. n = 24), $R_{(C\mathrm{i})^{3/2}:C\mathrm{e}}$ = 715.54 / (72 + 105 / 24 + 12 / 576) = 715.54 / 76.395833 = 9.366218.

As $\lg P$ = 224 (viz. n = 28), $R_{(C\mathrm{i})^{3/2}:C\mathrm{e}}$ = 715.54 / (72 + 105 / 28 + 12 / 784) = 715.54 / 75.765306 = 9.444164.

As $\lg P$ = 256 (viz. n = 32), $R_{(C\mathrm{i})^{3/2}:C\mathrm{e}}$ = 715.54 / (72 + 105 / 32 + 12/1024) = 715.54 / 75.292969 = 9.503411.

The average of the ratios = (9.259058 + 9.366218 + 9.444164 + 9.503411) / 4 = 37.572851 / 4 = 9.393213.

The average of |the errors| = (0.134155 + 0.026995 + 0.050951 + 0.110198) / 4 = 0.322299 / 4 = 0.080575.

The weight of average of |the errors| = 0.080575 / 9.393213 = 0.008578 = 0.8578 % < 1 %.

Thus, the ratio of the (3/2)-th root of number of clock cycles for MINV to the number of clock cycles for MEXP is almost one constant, which hints that the ratio is decided by two the highest order terms.

4.5 Ratio of Number of Cycles for MADD to 2nd Root of Number of Cycles for MMUL

Sect 3.1 says that the number of clock cycles for MADD (Ca) is $2n$.

Sect 3.3 says that the number of clock cycles for MMUL (Cm) is $4n(n+2)+4n(n+1)2 = 12n^2+8n$.

Then, $R_{C\mathrm{a}:(C\mathrm{m})^{1/2}} = C\mathrm{a}/(C\mathrm{m})^{1/2} = 2n/(12n^2+8n)^{1/2} = 1/(3+2/n)^{1/2}$.

When $\lg P$ = 160 (viz. n = 20), $R_{C\mathrm{a}:(C\mathrm{m})^{1/2}}$ = 1 / (3 + 2 / 20)$^{1/2}$ = 1 / 3.100000$^{1/2}$ = 0.567962.

When $\lg P$ = 192 (viz. n = 24), $R_{C\mathrm{a}:(C\mathrm{m})^{1/2}}$ = 1 / (3 + 2 / 24)$^{1/2}$ = 1 / 3.083333$^{1/2}$ = 0.569495.

When $\lg P$ = 224 (viz. n = 28), $R_{C\mathrm{a}:(C\mathrm{m})^{1/2}}$ = 1 / (3 + 2 / 28)$^{1/2}$ = 1 / 3.071429$^{1/2}$ = 0.570597.

When $\lg P$ = 256 (viz. n = 32), $R_{C\mathrm{a}:(C\mathrm{m})^{1/2}}$ = 1 / (3 + 2 / 32)$^{1/2}$ = 1 / 3.062500$^{1/2}$ = 0.571429.

The average of the ratios $= (0.567962 + 0.569495 + 0.570597 + 0.571429) / 4 = 2.279483 / 4 = 0.569871$.

The average of |the errors| $= (0.001909 + 0.000376 + 0.000726 + 0.001558) / 4 = 0.004569 / 4 = 0.001142$.

The weight of average of |the errors| = 0.001142 / 0.569871 = 0.002004 = 0.2004 % < 1 %.

Thus, the ratio of the number of clock cycles for MADD to the 2nd root of number of clock cycles for MMUL is almost one constant, which means that the ratio is decided by two the highest order terms.

Extendedly, other similar ratios will also be individually smaller than 1%, and almost one constant.

Sect 4.1-4.5 illustrate that when $\lg P \geq 160$ (viz. $n \geq 20$), the ratio of derivate numbers of clock cycles for two bytewise modular operations is decided by the highest order terms of the two time polynomials, and is almost a constant under different modulus lengths, and moreover, logically the ratio is still a constant even if the numbers of clock cycles taken by relevant trivial directives are considered.

5 Conclusion

In the paper, we analyze detailedly the bit complexity of every bitwise modular operation and the time complexity by single-precision directives of every non-bitwise modular operation in multiple-precision modular arithmetic, and give the number of clock cycles for every bytewise modular operation which utilizes directives from the ATmel 8-bit AVR instruction set.

At last, we acquire a interesting discovery that the ratio of derivate numbers of clock cycles for two bytewise modular operations is decided by the highest order terms of the two time polynomials, and is almost a constant under different modulus lengths when $\lg P \geq 160$ (or $n \geq 20$), and furthermore, there is a logical extension that the ratio is still a constant even if the numbers of clock cycles consumed by trivial directives of a modular operation shaped as an assembly source program are considered, where the demand amount for every trivial directive will be formulated during the running time evaluation.

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, etc. Introduction to Algorithms (3rd Edition). MIT Press, Cambridge, 2009.
2. E. N. Zalta, U. Nodelman, C. Allen, etc. Stanford Encyclopedia of Philosophy: Computational Complexity Theory. Stanford University, Stanford, 2016.
3. D. Z. Du and K. Ko, Theory of Computational Complexity, John Wiley & Sons, New York, 2000.
4. A. J. Menezes, P. van Oorschot, and S. Vanstone. Handbook of Applied Cryptography (Chapter 2 and 14). CRC Press, UK: London, 1997.
5. M. Scott and P. Szczechowiak. Optimizing Multiprecision Multiplication for Public Key Cryptography. Cryptology ePrint Archive, Report 2007/299, 2007.
6. E. Çelebi, M. Gözütok, and L. Ertaul. Implementations of Montgomery Multiplication Algorithms in Machine Languages. The 2008 Int'l Conference on Security & Management, LasVegas, 2008.
7. H. Davenport. The Higher Arithmetic (7th Edition). Cambridge University Press, UK: Cambridge, 1999.
8. T. W. Hungerford. Algebra. Springer-Verlag, New York, 1998.
9. M. Sipser. Introduction to the Theory of Computation. PWS Publishing, Boston, 1997.
10. S. Arora and B. Barak. Computational Complexity: A Modern Approach. Cambridge University Press, New York, 2009.
11. Atmel Corporation. 8-bit AVR Instruction Set. http://www.atmel.com, 2002.
12. K. H. Rosen. Elementary Number Theory and Its Applications (5th Edition). Addison-Wesley, Boston, 2005.
13. P. Q. Nguên and D. Stehlé. Floating-Point LLL Revisited. Advances in Cryptology – EUROCRYPT 2005, May 2005.