Modular Multiplication Using Special Prime Moduli

Mario Taschwer

University of Klagenfurt
Mario.Taschwer@uni-klu.ac.at

Abstract

Elliptic curve cryptosystems allow the use of prime fields with special moduli that speed up the finite field arithmetic considerably. Two algorithms for reduction with respect to special moduli have been implemented in software on both a 32-bit and a 64-bit platform and compared to well-known generic modular reduction methods. Timing results for multiplications in prime fields of size between 2^{191} and 2^{512} are presented and discussed.

1. Introduction

The growing use of public-key cryptosystems like RSA [17] or DSA [15] does not only provide a useful application of large finite fields, but also raises the need for efficient algorithms and implementations of finite field arithmetic. In particular, multi-precision modular reduction and modular exponentiation algorithms and implementation options have been investigated in the literature [6, 8]. As these contributions aim at supporting the arithmetic used in the RSA or DSA cryptosystems, they focus on the modular exponentiation operation for operand sizes of more than 1000 bits and on the asymptotic time complexity of the investigated algorithms.

Today, public-key cryptosystems based on elliptic curves [12, 5] have been integrated into current standards [10, 1] to provide an alternative to the classical systems mentioned above. The recommended elliptic curve cryptosystems operate on smaller finite fields of size close to 2^{200}, at the cost of a more complex arithmetic and a greater number of system parameters. The additional degree of freedom introduced by the curve parameters, however, allows the use of special finite fields yielding a very efficient arithmetic.

In this contribution, prime finite fields $\mathbb{F}_p$ of p elements are considered, where p is of some special form that allows an efficient modular reduction modulo p. One such special primes are *Pseudo-Mersenne* (PM) primes, the other ones are *Generalized Mersenne* (GM) primes, which are both generalizations of the well-known Mersenne numbers $2^k - 1$. We focus on modular multiplication modulo p, which represents the multiplicative operation in $\mathbb{F}_p$. Field exponentiations are not needed for arithmetic operations on elliptic curves, and field inversions can be avoided by using projective coordinates. According to the field sizes used in elliptic curve cryptosystems, we restrict our investigation to the range $2^{191} < p < 2^{512}$.

We present the results of a comparative software implementation of four modular reduction algorithms used for modular multiplication on both a 32-bit and a 64-bit platform. Two of these algorithms are well-known generic reduction methods, namely the classical one using divisions [11, Algorithm 4.3.1.D] and Montgomery's method [14], whereas the other ones are only applicable to PM or GM moduli, respectively.

We focus on prime finite fields, but for elliptic curve cryptography also fields of characteristic 2 and so-called Optimal Extension Fields (OEFs) [2] are used. The latter ones are small field extensions with characteristic close to the machine word base, which allow very efficient field operations, including field inversion. A recent work of Smart [18] indicates that OEFs have a clear performance advantage over both fields of characteristic 2 and prime fields for elliptic curve cryptosystems. However, the comparison was performed only for fields of size $\approx 2^{190}$ and only Montgomery and GM reduction were considered for prime field operations. De Win et al. [21] compare fields of characteristic 2 with prime fields, but only use Barrett reduction [3, 4] for the latter. The investigations in [6, 8] show that Barrett reduction is roughly as efficient as Montgomery reduction.

The paper is organized as follows. Section 2 introduces useful representations of prime finite fields and gives an overview over the modular multiplication operation. The following four sections describe each modular reduction algorithm in detail, and sections 7 and 8 present the implementation results and conclusion. The timing results are given in both graphical and tabular representation in the appendix.

2. Prime Finite Field Arithmetic

Besides the well-known standard representation of prime finite fields, a field representation utilizing Montgomery's reduction algorithm (see section 4) can be used, which has not yet been clearly stated in the literature. Field multiplications can be performed using the same methods of modular multiplication in both representations, as explained in section 2.3. Modular reduction with respect to special moduli, however, is only useful in standard representation.

2.1. Standard Representation

Let $p > 2^{160}$ be a prime. We can represent the elements of the finite field $\mathbb{F}_p$ by integers x in the range $0 \leq x < p$. We will call this the *standard representation* of prime field elements. In a computer with word base b, x is usually represented as a *multi-precision integer* $x = (x_{n-1}, \ldots, x_0)_b$ of non-negative *digits* $x_i < b$ such that $b^{n-1} \leq p < b^n$ and

$$x = \sum_{i=0}^{n-1} x_i b^i. \tag{2.1}$$

In the sequel we will always assume that x is stored in memory as an integer array $X[0 \ldots n-1]$ such that $X[i] = x_i$.

For an arbitrary integer value $b > 1$ we define $|x|_b$ to be the minimum number of digits x_i needed to represent x in the form of (2.1). That is, $|x|_b = n$ if and only if $b^{n-1} \leq x < b^n$. In particular, $|x|_2$ denotes the number of bits occurring in the binary representation of x.

2.2. Montgomery Representation

The set of elements of $\mathbb{F}_p$ can be represented by a certain permutation of the set $P = \{0, 1, \ldots, p-1\}$ such that an efficient use of the Montgomery reduction method (see section 4) is possible. Let $R > p$ be an integer coprime to p such that computations modulo R are easy to perform: $R = b^n$. Given an element $x \in \mathbb{F}_p$ in standard representation, we define the integer

$$\bar{x} = x \cdot R \,\mathrm{MOD}\, p \tag{2.2}$$

to be the *Montgomery representation*[1] of x. The mapping $\mu : P \to P$, $x \mapsto \bar{x}$ is bijective, since $\gcd(R, p) = 1$. Because $x = y$ if and only if $\mu(x) = \mu(y)$, testing equality of two field elements is equally easy for both standard and Montgomery representation. The inverse mapping is given by $\mu^{-1}(\bar{x}) = \bar{x} \cdot R^{-1} \,\mathrm{MOD}\, p$, and its extension to $\mathbb{Z}$ is called *Montgomery reduction*.

If we want to perform field operations in Montgomery representation, we will have to define addition $\oplus$ and multiplication $\odot$ for elements in Montgomery representation such that μ becomes a field homomorphism. That is, we must ensure that

$$\begin{aligned} \bar{x} \oplus \bar{y} &= \overline{x + y} \quad \text{and} \\ \bar{x} \odot \bar{y} &= \overline{x \cdot y} \end{aligned}$$

for all $x, y \in \mathbb{F}_p$ given in standard representation. For addition, we can just define $\bar{x} \oplus \bar{y} = \bar{x} + \bar{y} \,\mathrm{MOD}\, p$, since the map μ is linear. This means that field addition in Montgomery and standard representation can be performed using the same procedure. In particular, the additive inverse of $\bar{x}$ is given by $p - \bar{x}$. For multiplication, we define

$$\begin{aligned} \bar{x} \odot \bar{y} &:= \mu^{-1}(\bar{x} \cdot \bar{y}) = \bar{x} \cdot \bar{y} \cdot R^{-1} \,\mathrm{MOD}\, p \\ &= (x \cdot R \cdot y \cdot R) \cdot R^{-1} \,\mathrm{MOD}\, p \\ &= x \cdot y \cdot R \,\mathrm{MOD}\, p = \overline{x \cdot y}. \end{aligned} \tag{2.3}$$

Thus, a field multiplication in Montgomery representation is carried out by a multiplication of integers followed by a Montgomery reduction. Field additions and multiplications in Montgomery representation can hence be performed without any conversions to or from standard representation.

2.3. Modular Multiplication

As a field multiplication in both standard and Montgomery representation amounts to multiplying the operands and reducing them, modular multiplication is a basic building block of prime finite field arithmetic.

There are two possibilities to perform a modular multiplication of elements $x, y \in \mathbb{F}_p$. The straight-forward one is to multiply the n-digit integers x and y first, and then to reduce the $2n$-digit result $z = x \cdot y$ modulo p. This requires at least $2n$ digits of temporary storage

[1]This is called *p-residue* in the original paper [14].

space and the corresponding instructions for memory access. These costs can be reduced by *interleaving* multiplications and reductions according to the equation:

$$\begin{aligned} x \cdot y \,\mathrm{MOD}\, p &= \sum_{i=0}^{n-1} x_i \cdot y \cdot b^i \,\mathrm{MOD}\, p \\ &= (((\ldots(((x_{n-1} \cdot y \,\mathrm{MOD}\, p) \cdot b + x_{n-2} \cdot y) \,\mathrm{MOD}\, p) \cdot b + \ldots \\ &\quad + x_1 \cdot y) \,\mathrm{MOD}\, p) \cdot b + x_0 \cdot y) \,\mathrm{MOD}\, p \end{aligned} \tag{2.4}$$

However, the number of arithmetic operations needed to reduce a $2n$-digit integer may be smaller than the total number of arithmetic operations executed during n modular reductions of $(n+1)$-digit operands. This is even more likely to be true, if n is small, which is the case in current implementations of elliptic curve cryptosystems. In the sequel, we therefore focus on the non-interleaving method for modular multiplication.

The costs of a modular multiplication can be roughly estimated by counting only the number of single-precision multiplications, as these are the major and most expensive basic machine operations needed for a modular multiplication. Clearly, the non-interleaving method requires n^2 single-precision multiplications plus the costs for one modular reduction.

2.4. Modular Reduction

There are quite different algorithms available to perform the modular reduction $z \,\mathrm{MOD}\, p$. The classical one described by Knuth [11, Algorithm 4.3.1.D] first estimates the quotient $q = \lfloor z/p \rfloor$ and then computes $z \,\mathrm{MOD}\, p = z - q \cdot p$, with a subsequent possible correction of the error introduced by the quotient estimation.

As the computation of q involves single-precision divisions, which are rather expensive compared to single-precision multiplications, one may pre-compute a scaled value of $1/p$ to avoid divisions during reduction. This is what Barrett's algorithm [3, 4] does. As already mentioned, the description of this algorithm is not yet included in this paper.

An alternative approach to modular reduction is given in the original work of Montgomery [14], where an efficient method of computing the Montgomery reduction $z \cdot R^{-1} \,\mathrm{MOD}\, p$ without divisions is given (see sections 2.2 and 4).

Modular reductions can be performed without any integer division at all, if primes of a special form can be used. For instance, if p can be chosen as a *Mersenne prime*, that is as $p = 2^k - 1$, then an integer $z < p^2$ can be reduced modulo p by writing $z = u \cdot 2^k + v$, where u and v are k-bit integers. Now it follows from $2^k \equiv 1 \pmod{p}$ that

$$z \equiv u + v \pmod{p}.$$

Hence the modular reduction $z \,\mathrm{MOD}\, p$ can be performed by one modular addition of k-bit integers.

However, Mersenne primes are rare; there is none between the Mersenne primes $2^{127} - 1$ and $2^{521} - 1$. We might therefore look for primes of a more general form which allow us to apply the same idea explained above. Moreover, we want k to be a multiple of the word size w to avoid real bit shifts. Two such generalizations have been proposed so far. The first one is due to Crandall [7], who proposed the use of *Pseudo-Mersenne* (PM) primes $p = 2^n - c$ for some "small" positive integer c. The second generalization, due to Solinas [19], are primes generated by some polynomial expression of low coefficient norm, which are called *Generalized Mersenne* (GM) primes. We are going to describe each of these reduction methods in detail in the following sections.

3. Classical Reduction

The most obvious way of performing a modular reduction is that of adapting the ordinary pencil-and-paper method of division which has been formalized by Knuth [11, Algorithm 4.3.1.D]. The pseudocode of this algorithm is given in algorithm 1. Division of an $(n+m)$-digit number by an n-digit divisor is achieved by m divisions of $(n+1)$-digit numbers by the divisor. A quotient of the latter type, say $q = \lfloor x/p \rfloor$, is estimated by the expression

$$\hat{q} = \min(b-1, (x_n \cdot b + x_{n-1}) \,\mathrm{DIV}\, p_{n-1}).$$

It can be shown that $\hat{q}$ is never too small and, if $p_{n-1} \geq b/2$, at most two in error, i.e. $\hat{q} - 2 \leq q \leq \hat{q}$. To obtain the correct value of q, the multiplication $\hat{q} \cdot p$ can be reduced to $\hat{q} \cdot (p_{n-1} \cdot b + p_{n-2})$, at the cost of correcting a possible negative residue. However, negative residues occur only with probability $2/b$. Note that $x - q \cdot p < p$, which ensures that each time the **for** loop is entered in algorithm 1, the condition $z < p \cdot b^{i-n+1}$ is true. This implies that $z_i \leq p_{n-1}$.

The condition $p_{n-1} \geq b/2$ can always be satisfied by means of *normalization*, that is, by multiplying both z and p by an appropriate power of 2. This procedure might enlarge z by one more digit, but it does not if $z < p^2$.

The costs are dominated by the division in line 8, the (1×2)-multiplication in line 10, and the $(1 \times n)$-multiplication in line 15. As the **for** loop is executed m times, the algorithm performs m divisions and $m(n+2)$ single-precision multiplications. If $z < p^2$, we can set $m = n$.

4. Montgomery Reduction

As explained in section 2.2, performing a field multiplication in Montgomery representation makes use of an operation $x \mapsto x \cdot R^{-1} \,\mathrm{MOD}\, p$ $(x \in \mathbb{Z})$, which is called Montgomery reduction. The method uses a precomputed inverse of p modulo R and is based on the following theorem:

Theorem 4.1 ([14]). *Let $p, R > 1$ be coprime integers, and $p' = -p^{-1} \,\mathrm{MOD}\, R$. Then for any integer x, the number*

$$\hat{x} = (x + tp)/R, \ \textit{where } t = xp' \,\mathrm{MOD}\, R,$$

is an integer satisfying $\hat{x} \equiv xR^{-1} \pmod{p}$. Moreover, if $0 \leq x < pR$, then $0 \leq \hat{x} < 2p$.

Proof. Observe that $tp \equiv xpp' \equiv -x \pmod{R}$, so $\hat{x}$ is an integer. Since $\hat{x}R \equiv x \pmod{p}$, we have $\hat{x} \equiv xR^{-1} \pmod{p}$. If $0 \leq x < pR$, then $0 \leq x+tp < pR+Rp$, so $0 \leq \hat{x} < 2p$. □

Algorithm 1 Classical modular reduction.

```
Input: $(n+m)$-digit integer $z$, $n$-digit modulus $p$, $p_{n-1} \geq b/2$, $m \geq 1$, $n \geq 2$.
Output: $z \,\mathrm{MOD}\, p$.
 1: if $z \geq p \cdot b^m$ then
 2:     $z = z - p \cdot b^m$
 3: end if
 4: for $i = n+m-1$ downto $n$ do
 5:     if $z_i = p_{n-1}$ then
 6:         $q = b - 1$
 7:     else    // $z_i < p_{n-1}$
 8:         $q = (z_i \cdot b + z_{i-1}) \,\mathrm{DIV}\, p_{n-1}$
 9:     end if
10:     $y = q \cdot (p_{n-1} \cdot b + p_{n-2})$
11:     while $y > z_i \cdot b^2 + z_{i-1} \cdot b + z_{i-2}$ do
12:         $q = q - 1$
13:         $y = y - (p_{n-1} \cdot b + p_{n-2})$
14:     end while
15:     $z = z - q \cdot p \cdot b^{i-n}$
16:     if $z < 0$ then
17:         $z = z + p \cdot b^{i-n}$
18:     end if
19: end for
20: return $z$
```

Montgomery also proposed an algorithm to compute $\hat{x}$ efficiently for multi-precision operands [14]. Namely, $\hat{x}R$ is obtained by successively adding $p(x_i p' \,\mathrm{MOD}\, R)b^i$ to x for $i = 0, 1, \ldots, n-1$. However, this method can still be speeded up by observing [9] that the basic idea of this algorithm is to add multiples of p to x until the result becomes a multiple of R. This effect can also be achieved by computing $x_i p'_0 \,\mathrm{MOD}\, b$ instead of $x_i p' \,\mathrm{MOD}\, R$, where $p'_0 := -p_0^{-1} \,\mathrm{MOD}\, b$. This leads to algorithm 2 [6].

To show the correctness of this algorithm, let us denote the original value of x by $\bar{x}$ for now. Since $pp'_0 \equiv -1 \pmod{b}$, each time line 3 is executed the equation $pt = px_i p'_0 \equiv -x_i \pmod{b}$ holds. It follows by induction that the value assigned to x is a multiple of b^{i+1}. Moreover, $x \equiv \bar{x} \pmod{p}$ is a loop invariant. Hence, when execution has passed line 5, the equation $x \equiv \bar{x}(b^n)^{-1} \pmod{p}$ is true. Furthermore, when the **for** loop has been completed, the value of x satisfies

$$x = \bar{x} + p\Big(\sum_{i=0}^{n-1} t_{(i)}\, b^i\Big) < 2pb^n, \tag{4.1}$$

since each value $t_{(i)}$ of the variable t does not exceed $b-1$ and $\sum_{i=0}^{n-1}(b-1)\, b^i < b^n$. Thus, the final value of x computed by algorithm 2 is bounded by $0 \leq x < p$.

The number of single-precision multiplications performed by algorithm 2 is given by n^2+n, which is comparable to the costs of a multi-precision multiplication.

Using the GNU MP library, we could make use of the fast implementation of a multiply-accumulate routine for multi-precision integers by storing the carry word generated in

Algorithm 2 Montgomery reduction.

```
Input: $0 \le x < pb^n$, $p < b^n$, $p'_0 = -p_0^{-1} \,\mathrm{MOD}\, b$.
Output: $x(b^n)^{-1} \,\mathrm{MOD}\, p$.
1: for $i = 0$ to $n-1$ do
2:     $t = (x_i \cdot p'_0) \,\mathrm{MOD}\, b$
3:     $x = x + (p \cdot t)b^i$
4: end for
5: $x = x \,\mathrm{DIV}\, b^n$
6: if $x \ge p$ then
7:     $x = x - p$
8: end if
9: return $x$
```

Tab. 1: Values of n and c such that $p = 2^n - c$ is prime (with high probability)

n	$n/16$	$n/32$	$n/64$	c
160	10	5	–	47, 57, 75, 189, 285, 383, 465, 543, 659, 843
192	12	6	3	237, 333, 399, 489, 527, 663, 915, 945
224	14	7	–	63, 363, 573, 719, 773, 857
256	16	8	4	189, 357, 435, 587, 617, 923
320	20	10	5	197, 743, 825, 843, 873, 1007, 1017
384	24	12	6	317
448	28	14	7	203, 207, 825
512	32	16	8	569, 629, 875, 975

line 3 in the current digit x_i, which is not needed afterwards by the algorithm. This allows only n digits to be processed in line 3 and to add the carry words when the loop has been completed by computing

$$(x_{n-1}, \ldots, x_0) = (x_{2n-1}, \ldots, x_n) + (x_{n-1}, \ldots, x_0).$$

Note that if the latter addition generates a carry bit or if $x \,\mathrm{MOD}\, b^n \ge p$, it suffices to compute $x = x - p \,\mathrm{MOD}\, b^n$ to get the correct result for x.

5. Pseudo-Mersenne Primes

These are primes of the form $p = b^n - c$ where c is a “small” integer. Table 1 lists some primes of the desired form in the range of interest for elliptic curve cryptography. For each given choice of n, all values of c satisfying $0 < c < 1024$ such that $2^n - c$ is prime are listed.

Given an integer $z = z''\, b^n + z'$, we can write

$$z = z''\, b^n + z' \equiv z''\, c + z' \pmod{p}, \quad \text{since} \quad b^n \equiv c \pmod{p}. \tag{5.1}$$

By applying this method recursively on z'', $z \,\mathrm{MOD}\, p$ can be obtained using only additions and multiplications by c. The resulting algorithm 3 due to Crandall [7] and a proof of

Algorithm 3 Reduction by a Pseudo-Mersenne modulus $p = b^n - c$.

```
Input: a base $b$ integer $z$, $p = b^n - c$ where $0 < c < b^n$.
Output: $z \,\mathrm{MOD}\, p$.
 1: $q^{(0)} = z \,\mathrm{DIV}\, b^n$, $r = z \,\mathrm{MOD}\, b^n$, $i = 0$
 2: while $q^{(i)} > 0$ do
 3:     $r = r + (cq^{(i)} \,\mathrm{MOD}\, b^n)$
 4:     $q^{(i+1)} = cq^{(i)} \,\mathrm{DIV}\, b^n$
 5:     $i = i + 1$
 6: end while
 7: while $r \geq p$ do
 8:     $r = r - p$
 9: end while
10: return $r$
```

correctness for arbitrary values of c and z appear in [13, Ch. 14]. Note that each $q^{(i)}$ denotes a multi-precision integer here.

For use in finite field arithmetic, this algorithm can be optimized due to the fact that z is never greater than the product of two residues modulo p, i.e. $z \leq (p-1)^2 < b^{2n}$. Moreover, as $b \geq 2^{16}$, we are free to choose c less than b, which will accelerate the multiplications by c considerably. The latter assumption implies that, since $cq^{(i)} \leq b^n q^{(i+1)}$,

$$q^{(i+1)} \leq \frac{cq^{(i)}}{b^n} < \frac{q^{(i)}}{b^{n-1}}, \tag{5.2}$$

after each execution of line 4 of algorithm 3. Assuming that $z < b^{2n}$, we therefore obtain the following bounds for the values of the $q^{(i)}$'s:

$$q^{(0)} < b^n, \quad q^{(1)} < b, \quad q^{(2)} = 0 \text{ for } n \geq 2. \tag{5.3}$$

Hence the **while** loop is executed at most 2 times. The resulting optimized algorithm is shown as algorithm 4. Rq denotes a register variable and C and $C1$ are carry words. The number of single-precision multiplications is linear in n, namely $n+1$, such that the running time for a modular multiplication will be dominated by the multiplication step.

6. Generalized Mersenne Primes

A *Generalized Mersenne (GM) prime* is a prime p of the form

$$p = b^n + c_{n-1}\, b^{n-1} + \ldots + c_0\,, \tag{6.1}$$

where b is a power of 2 (the machine's word base), $c_i \in \mathbb{Z}$, and the norm $\sum_{i=0}^{n-1} |c_i|$ of the coefficient vector is "small". Of course, the condition that such an expression for p should be prime imposes a number of restrictions on the coefficients c_i. One immediate necessary condition is that c_0 is odd. However, the investigation of such necessary conditions is beyond the scope of this paper (see [19]). In what follows we assume that a prime of this form is already known, and we describe two classes of algorithms for performing modular reductions modulo p:

Algorithm 4 Reduction of $z < b^{2n}$ by a modulus $p = b^n - c$, optimized version.

```
Input: $z < b^{2n}$, $p = b^n - c$ where $0 < c < b$ and $n \geq 2$.
Output: $z \,\mathrm{MOD}\, p$.
 1: $q = z \,\mathrm{DIV}\, b^n,\ r = z \,\mathrm{MOD}\, b^n$
 2: $(Rq, q) = q \cdot c$
 3: $(C, r) = r + q$
 4: if $Rq > 0$ then
 5:     $(C1, r) = r + Rq \cdot c$
 6:     $C = C + C1$
 7: end if
 8: while $C > 0$ or $r \geq p$ do
 9:     $r = (r - p) \,\mathrm{MOD}\, b^n$
10:     $C = C - 1$
11: end while
12: return $r$
```

- A generic reduction algorithm based on Solinas' work [19] which works for every GM modulus p, but benefits from the special modulus structure.
- A number of hard-coded reduction algorithms, each of which belonging to only one special modulus p. These algorithms are just optimized versions of the generic method mentioned above.

6.1. Generic Generalized Mersenne Reduction

The idea of GM reduction is based on the fact that every modular reduction $z \,\mathrm{MOD}\, p$, where $z < p^2 < b^{2n}$, can be expressed as a sum of at most n multi-precision integers modulo p. In general, the computation of each digit of these integers requires n multiplications, yielding a total of n^3 single-precision multiplications. However, if the condition on the c_i's stated above holds, the products can be computed using only additions, and often the number of multi-precision additions can be kept small.

For the following description, we restrict the range of the coefficients to $|c_i| \leq 1$. This conforms to the generalized Mersenne primes recommended for the use in elliptic curve cryptosystems (e.g. by NIST [16]). Moreover, we want p to be smaller than b^n, i.e. for the largest $i < n$ satisfying $c_i \neq 0$ we require that $c_i < 0$.

To get to an efficient method for reduction modulo p, we first consider the polynomial

$$f(t) = t^n + \sum_{i=0}^{n-1} c_i \, t^i \tag{6.2}$$

(hence $f(b) = p$) and compute the polynomial residues of $t^n, t^{n+1}, \ldots, t^{2n-1}$ modulo $f(t)$. We arrange the coefficient vectors of these polynomials as the rows of an $(n \times n)$-matrix X. Explicitly, the matrix X, which we call the *reduction matrix of f*, is defined inductively by the equations

$$X[0,j] = -c_j \qquad \text{for } 0 \le j < n, \tag{6.3}$$

$$X[i,j] = \begin{cases} -c_0\, X[i-1,n-1] & \text{for } j = 0,\ i = 1,\ldots,n, \\ X[i-1,j-1] - c_j\, X[i-1,n-1] & \text{for } j > 0,\ i = 1,\ldots,n. \end{cases} \tag{6.4}$$

Note that $|X[i,j]| \le n$ for all $0 \le i,j < n$, because $|c_i| \le 1$. It is easy to show by induction that this definition implies that for all i in the range $0 \le i < n$,

$$t^{n+i} \equiv \sum_{j=0}^{n-1} X[i,j]\, t^j \pmod{f(t)}. \tag{6.5}$$

This equation allows us to perform reduction modulo $p = f(b)$ by evaluating a linear expression involving the matrix X. Given an integer

$$z = \sum_{i=0}^{2n-1} z_i\, b^i < p^2,$$

which is to be reduced modulo p, we can replace all powers $b^i \ge b^n$ by the expressions resulting from equation (6.5). This yields

$$z \operatorname{MOD} p = \sum_{j=0}^{n-1} y_j\, b^j \quad \text{where} \quad y_j = \Bigl(z_j + \sum_{i=0}^{n-1} z_{n+i}\, X[i,j]\Bigr) \operatorname{MOD} p\,. \tag{6.6a}$$

Using matrix notation, these equations can be denoted more compactly as

$$z \operatorname{MOD} p = (y_0\ \ldots\ y_{n-1}) = \bigl((z_0\ \ldots\ z_{n-1}) + (z_n\ \ldots\ z_{2n-1}) \cdot X\bigr) \operatorname{MOD} p\,. \tag{6.6b}$$

We want to evaluate this linear expression efficiently by taking advantage of two facts: first, the matrix product can be decomposed into a sum of row vectors which are regarded as n-digit integers. Second, as the entries of X are absolutely bounded by n, which is a small integer, we may replace the single-precision multiplications involved in the matrix product by successive additions and subtractions. These considerations lead to the basic algorithm 5 for a modular reduction.

The number of executions of the first **while** loop of algorithm 5 is equal to the maximal sum of positive entries along each column of the reduction matrix X. We therefore call this number the *modular addition weight* $w_a(f)$ of f. Similarly, the number of multi-precision modular subtractions only depends on the negative entries of X and is called the *modular subtraction weight* $w_s(f)$ of f. The total running time of the basic algorithm is determined by the value $w(f) = w_a(f) + w_s(f)$, which is called the *modular reduction weight* of f.

To indicate that the reduction matrix also depends on f, we write $X = X_f$ in the following. For the purpose of a simpler treatment of additions and subtractions, let us decompose X_f into a positive and a negative part:

$$X_f^+[i,j] = \begin{cases} X_f[i,j] & \text{if } X_f[i,j] > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6.7}$$

$$X_f^-[i,j] = \begin{cases} |X_f[i,j]| & \text{if } X_f[i,j] < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6.8}$$

Algorithm 5 Basic algorithm for reduction modulo a Generalized Mersenne prime.

Input: $z < p^2$, $p = f(b)$ a Generalized Mersenne prime, and the reduction matrix X of f.
Output: $z \,\mathrm{MOD}\, p$.

```
1:  $x = z \,\mathrm{MOD}\, p$
2:  **while** $X$ contains a positive entry **do**    // perform modular additions
3:     **for all** columns $j$ of $X$ **do**
4:        search for the least $i \geq 0$ such that $X[i,j] > 0$
5:        **if** found **then**
6:           $u_j = z_{n+i}$; decrement $X[i,j]$
7:        **else**
8:           $u_j = 0$
9:        **end if**
10:    **end for**
11:    $x = (x + u) \,\mathrm{MOD}\, p$    // multi-precision modular addition
12: **end while**
13: **while** $X$ contains a negative entry **do**    // perform modular subtractions
14:    **for all** columns $j$ of $X$ **do**
15:       search for the least $i \geq 0$ such that $X[i,j] < 0$
16:       **if** found **then**
17:          $u_j = z_{n+i}$; increment $X[i,j]$
18:       **else**
19:          $u_j = 0$
20:       **end if**
21:    **end for**
22:    $x = (x - u) \,\mathrm{MOD}\, p$    // multi-precision modular subtraction
23: **end while**
24: **return** $x$
```

Algorithm 6 Precomputation of the modular addition matrix of a generalized Mersenne prime $p = f(b)$.

Input: the positive part X_f^+ of the reduction matrix of f, and the modular addition weight $w_a(f)$.
Output: the modular addition matrix A_f.

```
1:  **for** $k = 0$ to $w_a(p) - 1$ **do**
2:     **for** $j = 0$ to $n - 1$ **do**
3:        $i = 0$
4:        **while** $i < n$ and $X_f^+[i,j] = 0$ **do**
5:           increment $i$
6:        **end while**
7:        **if** $i < n$ **then**    // positive entry found
8:           $A_f[k,j] = n + i$; decrement $X_f^+[i,j]$
9:        **else**
10:          $A_f[k,j] = 2n$
11:       **end if**
12:    **end for**
13: **end for**
14: **return** $A_f$
```

Clearly, we have $X_f = X_f^+ - X_f^-$, and the modular addition and subtraction weights are given by

$$w_a(f) = \max_{0 \le j < n} \Big\{ \sum_i X_f^+[i,j] \Big\}, \quad w_s(f) = \max_{0 \le j < n} \Big\{ \sum_i X_f^-[i,j] \Big\}. \tag{6.9}$$

We are now ready to speed up algorithm 5. Observe that the search for positive and negative entries in lines 4 and 15, respectively, only depends on the reduction matrix X_f. We can therefore precompute the positions (i,j) found by the basic algorithm and construct a matrix A_f of dimension $w_a(f) \times n$ such that $A_f[k,j]$ contains the index $n+i$ used in line 4 after the **while** loop has been executed k times $\big(0 \le k < w_a(f)\big)$. This means that row number k of A_f contains all indices of the digits of z which are to be processed in the $(k+1)$-th modular addition of the basic algorithm. We call A_f the *modular addition matrix* of f. The construction of A_f is shown in algorithm 6.

Another matrix S_f, which is called the *modular subtraction matrix* of f, is obtained using the same algorithm with input X_f^- and $w_s(f)$. Its dimension is $w_s(f) \times n$.

The basic algorithm 5 can now be improved using the matrices A_f and S_f, as shown in algorithm 7 in the appendix. Additionally, we could get rid of the temporary n-digit variable u by incorporating the use of A_f and S_f into the modular addition and subtraction operations, respectively. This optimization option, however, requires an extension of the low-level multi-precision integer library and has not been implemented.

To illustrate these algorithms, let us consider the prime

$$p = 2^{192} - 2^{64} - 1 = f(b) \quad \text{where} \quad f(t) = t^3 - t - 1,\ b = 2^{64}.$$

The reduced matrix is computed as

$$X_f = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \text{meaning} \qquad \begin{aligned} t^3 &\equiv 1 + t && \pmod{f(t)}, \\ t^4 &\equiv t + t^2 && \pmod{f(t)}, \\ t^5 &\equiv 1 + t + t^2 && \pmod{f(t)}. \end{aligned}$$

The modular addition weight is $w_a(f) = 3$, which is the maximum of the column sums in X_f. As X_f does not contain any negative values, the modular subtraction weight is $w_s(f) = 0$. A reduction modulo p can thus be performed using only 3 modular additions of 3-digit integers. The modular subtraction matrix is not defined, as its column dimension is 0. Applying algorithm 6 on $X_f^+ = X_f$ gives the modular addition matrix:

$$A_f = \begin{pmatrix} 3 & 3 & 4 \\ 5 & 4 & 5 \\ -1 & 5 & -1 \end{pmatrix}.$$

Given an integer $z = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 & z_4 & z_5 \end{pmatrix} < p^2$, algorithm 7 effectively computes

$$z \operatorname{MOD} p = \big(u^{(0)} + u^{(1)} + u^{(2)} + u^{(3)}\big) \operatorname{MOD} p, \text{ where}$$

$$\begin{aligned} u^{(0)} &= (\; z_0 \;\; z_1 \;\; z_2 \;), \\ u^{(1)} &= (\; z_3 \;\; z_3 \;\; z_4 \;), \\ u^{(2)} &= (\; z_5 \;\; z_4 \;\; z_5 \;), \\ u^{(3)} &= (\; 0 \;\; z_5 \;\; 0 \;). \end{aligned}$$

Tab. 2: Some Generalized Mersenne primes for use in elliptic curve cryptography

$f(t)$	b	$p = f(b)$	$w_a(f)$	$w_s(f)$
$t^3 - t - 1$	2^{64}	$2^{192} - 2^{64} - 1$	3	0
$t^7 - t^3 + 1$	2^{32}	$2^{224} - 2^{96} + 1$	2	2
$t^8 - t^7 + t^6 + t^3 - 1$	2^{32}	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	6	4
$t^{12} - t^4 - t^3 + t - 1$	2^{32}	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	7	3
$t^{14} - t^7 - 1$	2^{32}	$2^{448} - 2^{224} - 1$	3	0
$t^{16} - t - 1$	2^{32}	$2^{512} - 2^{32} - 1$	3	0

This result also occurs in [16, Appendix 1], where the entries along each column appear in a different order, but this does not alter the value of $z \operatorname{MOD} p$.

Finally, some Generalized Mersenne primes p in the range $2^{159} < p < 2^{512}$ for bases $b = 2^{32}$ and $b = 2^{64}$ are listed in table 2. Most of them are taken from [16]. The base b need not be the actual machine word base. The algorithm can also be applied to the modulus $p = 2^{192} - 2^{64} - 1$ on a 32-bit machine by using the polynomial $t^6 - t^2 - 1$. Conversely, the algorithm can be implemented for the other polynomials on a 64-bit architecture using half-word arithmetic. This may even be necessary, since suitable GM primes of low weight seem to be rare. Note that table 2 lists all GM primes in the given range which are generated by polynomials of the following forms (for $b = 2^{32}$ and $b = 2^{64}$):

$$\begin{aligned}
f(t) &= t - 1, \\
f(t) &= t - 3, \\
f(t) &= t^n - t^c - 1 && \text{where } 0 < 2c \leq n, \\
f(t) &= \frac{t^{n+1} + (-1)^n}{t + 1} \\
&= t^n - t^{n-1} + t^{n-2} - \cdots \pm 1 && \text{for } n > 1.
\end{aligned}$$

These are the polynomials of modular reduction weight 1 and 3 listed in [19].

6.2. Hard-Coded Generalized Mersenne Reduction

Implementors of the elliptic curve cryptosystems specified in the standards [10, 1] may choose to use only the recommended GM primes listed in table 2 for the underlying finite field arithmetic. In this case, the generic GM reduction algorithm can be optimized for different individual primes by hard-coding the multi-precision additions modulo p.

In the sequel we present the addition and subtraction matrices of the polynomials of weight ≤ 4 given in table 2. Sometimes, the implementation is facilitated by re-arranging the matrix elements within each column, which does not alter the result due to the commutativity of addition. The aim is to preserve the original order of the digits of the input value z as far as possible to minimize copy operations. We just present the transformed matrices.

$f(t) = t^3 - t - 1$ $(p = 2^{192} - 2^{64} - 1)$

$$A_f = \begin{pmatrix} 3 & 4 & 5 \\ 5 & 3 & 4 \\ -1 & 5 & -1 \end{pmatrix}$$

$f(t) = t^7 - t^3 + 1$ $(p = 2^{224} - 2^{96} + 1)$

$$A_f = \begin{pmatrix} -1 & -1 & -1 & 7 & 8 & 9 & 10 \\ -1 & -1 & -1 & 11 & 12 & 13 & -1 \end{pmatrix}$$

$$S_f = \begin{pmatrix} 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 11 & 12 & 13 & -1 & -1 & -1 & -1 \end{pmatrix}$$

$f(t) = t^{14} - t^7 - 1$ $(p = 2^{448} - 2^{224} - 1)$

$$A_f = \begin{pmatrix} 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 27 \\ 21 & 22 & 23 & 24 & 25 & 27 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \end{pmatrix}$$

$f(t) = t^{16} - t - 1$ $(p = 2^{512} - 2^{32} - 1)$

$$A_f = \begin{pmatrix} 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 27 & 28 & 29 & 30 & 31 & \\ 31 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 27 & 28 & 29 & 30 & \\ -1 & 31 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

7. Implementation Results

The modular reduction algorithms described above were implemented as an extension of the LiDIA Computational Number Theory library [20], using the GNU MP library as the underlying multi-precision arithmetic. The timing tests were run on both an Intel 32-bit platform, and an Alpha 64-bit architecture, both running Linux kernel version 2.2.

The 32-bit processor is a Pentium 200 Mhz with a 64-Bit data bus rated at 66 Mhz. The CPU possesses an 8 KB L1 instruction cache and an 8 KB write-back L1 data cache. The motherboard is equipped with a 512 KB pipelined burst SRAM L2 cache.

The 64-bit CPU is an Alpha 21164 600 Mhz on an AlphaPC 164UX Motherboard with a board-level 2 MB L3 synchronous SRAM cache and a 128-bit data path. The CPU possesses an 8 KB L1 instruction cache, an 8 KB write-through L1 data cache, and a 96 KB write-back L2 unified instruction and data cache.

The time measurements were done by reading the processor clock cycle counter before and after the relevant code section. To obtain reliable results, each modular multiplication was repeated 50 times with the same operands, and only the mean value and its expected error, denoted by $t \pm \Delta t$, were taken into account for further processing. The expected error has been computed from the individual timings τ_i as

$$\Delta t = \sqrt{\sum_{i=1}^{N} \frac{(\tau_i - t)^2}{N(N-1)}} \qquad (N = 50). \tag{7.1}$$

The overhead introduced by the time measuring procedure was eliminated by subtracting the mean timing of an "empty" code section from the measured timings. This overhead was ≈ 35 clock cycles on the Pentium and ≈ 15 clock cycles on the Alpha machine.

For each algorithm A and prime p, 100 integer pairs z_1, z_2 in the range $0 \leq z_1, z_2 < p$ were chosen randomly as input values for the modular multiplication timing procedure. Up to 4 different PM primes and one GM prime of the same bit length were tested and all individual timings $t_i \pm \Delta t_i$ were collected. Only the GM primes listed in table 2 on page 358 with modular reduction weight at most 4 were used. The classical and Montgomery methods were applied to both types of primes.

Tables 3 and 4 in the appendix list the statistical results for each algorithm and modulus length. The standard deviation σ is computed only from the t_i's. It describes the distribution of the timings as the modular multiplication operands vary. The error θ of the mean value is computed from the individual errors Δt_i as

$$\theta = \frac{\sqrt{\sum_{i=1}^{N} (\Delta t_i)^2}}{N}. \tag{7.2}$$

The total error $\Delta \bar{t}$ is just the sum of σ and θ. The values $\bar{t} \pm \Delta \bar{t}$ are shown graphically in figures 1 and 2. Because there is only one data point for the generic GM method on the Alpha platform, it is not displayed.

For comparison, the timings for the modular reduction operation are also included (see figures 3 and 4, and tables 5 and 6 in the appendix).

The peak in the curve for classical reduction on the Alpha at 224 bits is due to the normalization of the operands, which is only necessary if the modulus size is not a multiple of the word size. The timing for Montgomery reduction on the Alpha at the same bit length shows the effect of the precomputation on the modulus. Thus, the curve is really a step function, meaning that the running time is a function of $\lceil \log_b p \rceil$.

The results indicate that modular reductions by hard-coded GM primes are more efficient than the ones by Pseudo-Mersenne primes, provided that the GM prime is a polynomial function of the actual word base. If hard-coded GM reduction is implemented on a half-word base (see the 224-, 448-, and 512-bit GM primes on the Alpha), Pseudo-Mersenne reduction might be faster. Moreover, the efficiency of GM reductions strongly depends on the modular reduction weight of the generating polynomial. It is expected from the results in figure 3 that Pseudo-Mersenne reduction is faster than hard-coded GM reduction whenever the modular reduction weight exceeds 4. But also the structure of the modular addition and subtraction matrices determine the efficiency of a GM reduction implementation, as can be seen from the fact that the hard-coded GM reductions were faster for the 512-bit prime than for the 448-bit prime, on both platforms.

8. Conclusion

The implementation results show that the execution times of field multiplications can be remarkably improved with respect to Montgomery multiplication by using the modular reduction algorithms for special moduli. For field sizes near 2^{192}, the gain is about 35%

on the 32-bit platform, and 27% on the 64-bit architecture. The gap increases with the field size up to 51% and 41%, respectively, at sizes near 2^{512}.

The generic method based on Generalized Mersenne is up to 10% slower than Pseudo-Mersenne reduction on the 32-bit platform, and appears to be even slower than Montgomery multiplication on the 64-bit platform. However, when GM reduction is hard-coded for each individual modulus, it may have an advantage over Pseudo-Mersenne reduction of up to 9%.

From an implementor's point of view, the disadvantage of Generalized Mersenne reduction is its bad scalability with respect to the bit length of the modulus. This means that GM reduction must be implemented separately for each special modulus, and the performance strongly depends on the modular reduction weight of the generating polynomial, on the machine word base, and on the structure of the modular addition and subtraction matrices, as explained in the previous section. Moreover, GM primes of low modular reduction weight are rather rare compared to Pseudo-Mersenne primes.

An additional implication of the timing results given in this paper is that the efficiency of general prime finite field arithmetic can be increased considerably by using the Montgomery representation of field elements. The official LiDIA library does not yet support Montgomery representation, but uses classical reduction for all prime field computations. However, the measured improvements for field multiplications are 18-31% on the 32-bit platform and 26-42% on the 64-bit platform, in the investigated range of moduli.

References

[1] American National Standards Institute: Public key cryptography for the financial services industry – The elliptic curve digital signature algorithm, January 1999, ANSI X9.62-1998.

[2] D.V. Bailey, C. Paar: Optimal extension fields for fast arithmetic in public-key algorithms, Advances in Cryptology – CRYPTO'98, LNCS 1462, 1998, pp. 472–485.

[3] P. Barrett: Communications authentication and security using public key encryption – a design for implementation, Master's thesis, Oxford University, September 1984.

[4] P. Barrett: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor, Advances in Cryptology – CRYPTO'86, LNCS, vol. 263, Springer, 1987, pp. 311–323.

[5] I. Blake, G. Seroussi, N. Smart: Elliptic curves in cryptography, Cambridge University Press, 1999.

[6] A. Bosselaers, R. Govaerts, J. Vandewalle: Comparison of three modular reduction functions, Advances in Cryptology – CRYPTO'93, LNCS, no. 773, Springer, 1994, pp. 175–186.

[7] R. Crandall: Method and apparatus for public key exchange in a cryptographic system, U.S. Patent Number 5159632, 1992.

[8] J.-F. Dhem: Design of an efficient public-key cryptographic library for RISC-based smart cards, Ph.D. thesis, Laboratoire de microélectronique de l'Université catholique de Louvain, Belgium, May 1998.
http://www.dice.ucl.ac.be/crypto/dhem/these/

[9] S.R. Dussé, B.S. Kaliski, Jr.:A cryptographic library for the motorola dsp56000, Advances in Cryptology – EUROCRYPT'90, LNCS, no. 473, Springer, 1991, pp. 230–244.

[10] IEEE P1363 Working Group: Standard specifications for public key cryptography, Draft Version 13, November 1999. http://grouper.ieee.org/groups/1363/

[11] D.E. Knuth: The art of computer programming, third ed., vol. 2 – Seminumerical Algorithms, Addison-Wesley, 1998.

[12] N. Koblitz: Elliptic curve cryptosystems, Mathematics of Computation 48(1987), 203–209.

[13] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone: Handbook of applied cryptography, CRC Press, 1997.

[14] P.L. Montgomery: Modular multiplication without trial division, Mathematics of Computation 44(1985), 519–521.

[15] National Institute of Standard and Technology: Digital signature standard, May 1994, NIST FIPS PUB 186.

[16] National Institute of Standard and Technology: Recommended elliptic curves for federal government use, July 1999. http://csrc.nist.gov/encryption

[17] R.L. Rivest, A. Shamir, L.M. Adleman: A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 2(1978), no. 21, 120–126.

[18] N.P. Smart: A comparison of different finite fields for use in elliptic curve cryptosystems, Tech. Report CSTR-00-007, University of Bristol, June 2000, to appear in Computers and Mathematics with Applications.

[19] J.A. Solinas: Generalized mersenne numbers: Tech. Report CORR 99-39, Centre for Applied Cryptographic Research, Waterloo, Canada, 1999. http://www.cacr.math.uwaterloo.ca/

[20] TU Darmstadt: LiDIA Computational Number Theory Library. http://www.informatik.tu-darmstadt.de/TI/LiDIA/

[21] E. De Win, S. Mister, B. Preneel, M. Wiener: On the performance of signature schemes based on elliptic curves, in: J. P. Buhler (ed.): Algorithmic Number Theory – ANTS-III, LNCS, no. 1423, Springer, 1998, pp. 252–266.

A. Generic Generalized Mersenne Reduction

Algorithm 7 Efficient reduction modulo a Generalized Mersenne prime $p = f(b)$.

Input: $z < p^2$, the modular addition matrix A_f, the modular subtraction matrix S_f, and the corresponding column dimensions $w_a(f)$ and $w_s(f)$.

Output: $z \,\mathrm{MOD}\, p$.

```
1: $i = 2n$
2: while $b^i > z$ do
3:    $z_i = 0$
4:    $i = i - 1$
5: end while
6: $x = z \,\mathrm{MOD}\, p,\ C = 0$
7: if $w_a(f) > 0$ then
8:    for $k = 0$ to $w_a(p) - 1$ do     // perform modular additions
9:       for $j = 0$ to $n - 1$ do
10:         $u_j = z_{A_f}[k, j]$
11:      end for
12:      $(C1, x) = x + u$
13:      $C = C + C1$
14:   end for
15:   while $C > 0$ or $x \geq p$ do
16:      $x = (x - p) \,\mathrm{MOD}\, b^n$
17:      $C = C - 1$
18:   end while
19: end if
20: if $w_s(f) > 0$ then
21:   $C = 0$
22:   for $k = 0$ to $w_s(p) - 1$ do     // perform modular subtractions
23:      for $j = 0$ to $n - 1$ do
24:         $u_j = z_{S_f}[k, j]$
25:      end for
26:      $(C1, x) = x - u$
27:      $C = C + C1$
28:   end for
29:   while $C > 0$ do
30:      $x = (x + p) \,\mathrm{MOD}\, b^n$
31:      $C = C - 1$
32:   end while
33: end if
34: return $x$
```

B. Timing Diagrams

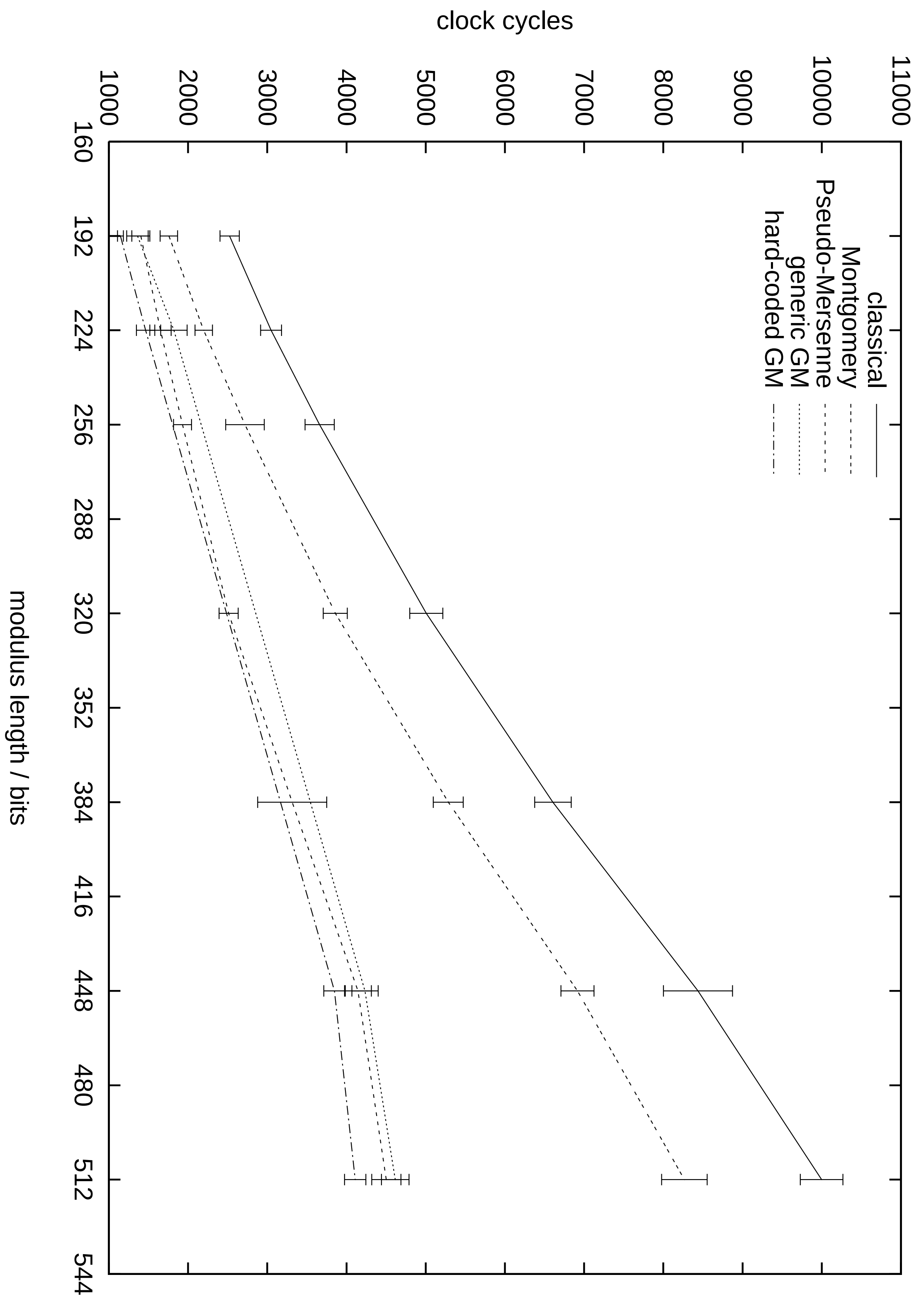

Fig. 1: Modular multiplication timings for special prime moduli on a Pentium-II based PC.

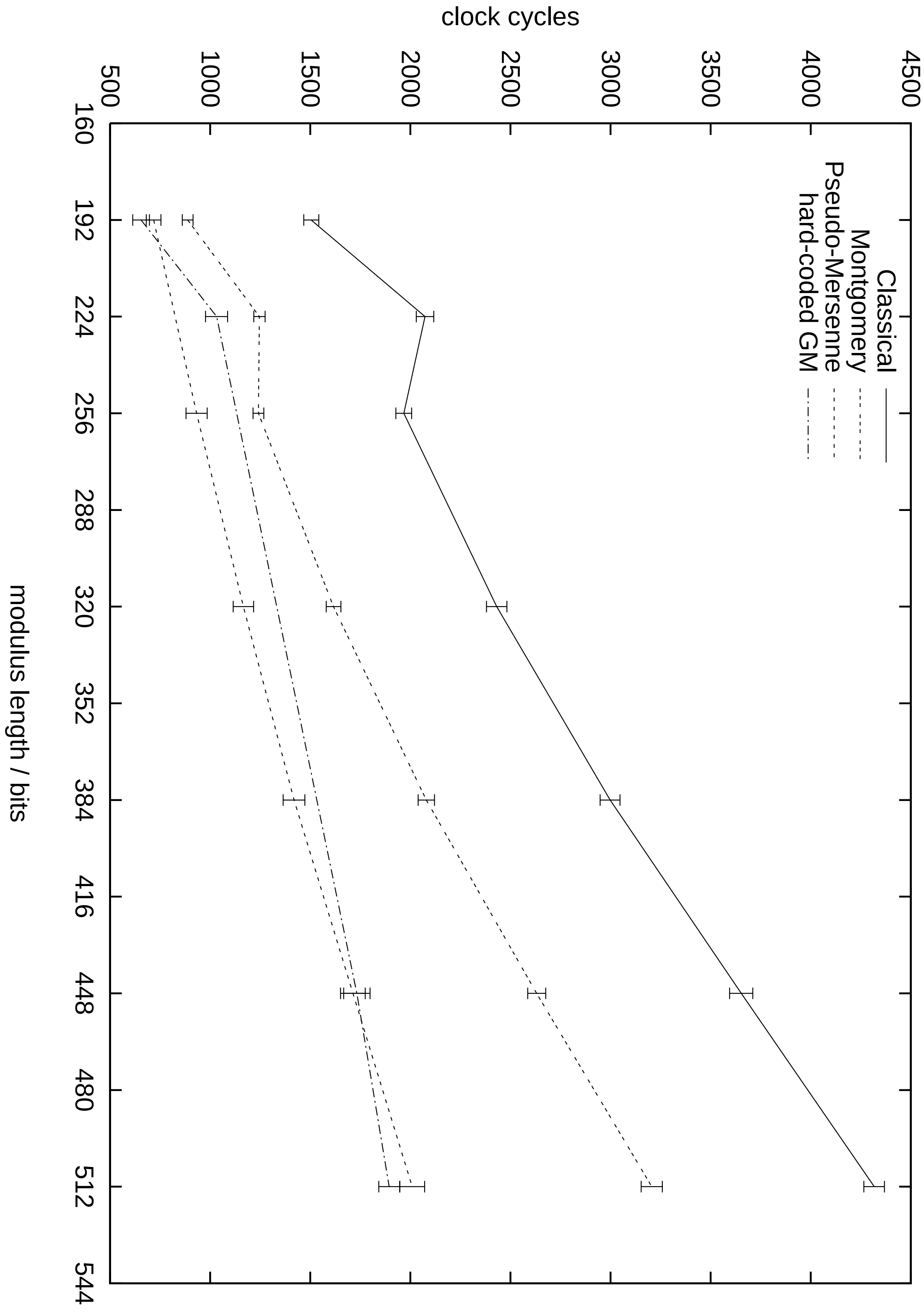

Fig. 2: Modular multiplication timings for special prime moduli on an Alpha 21164.

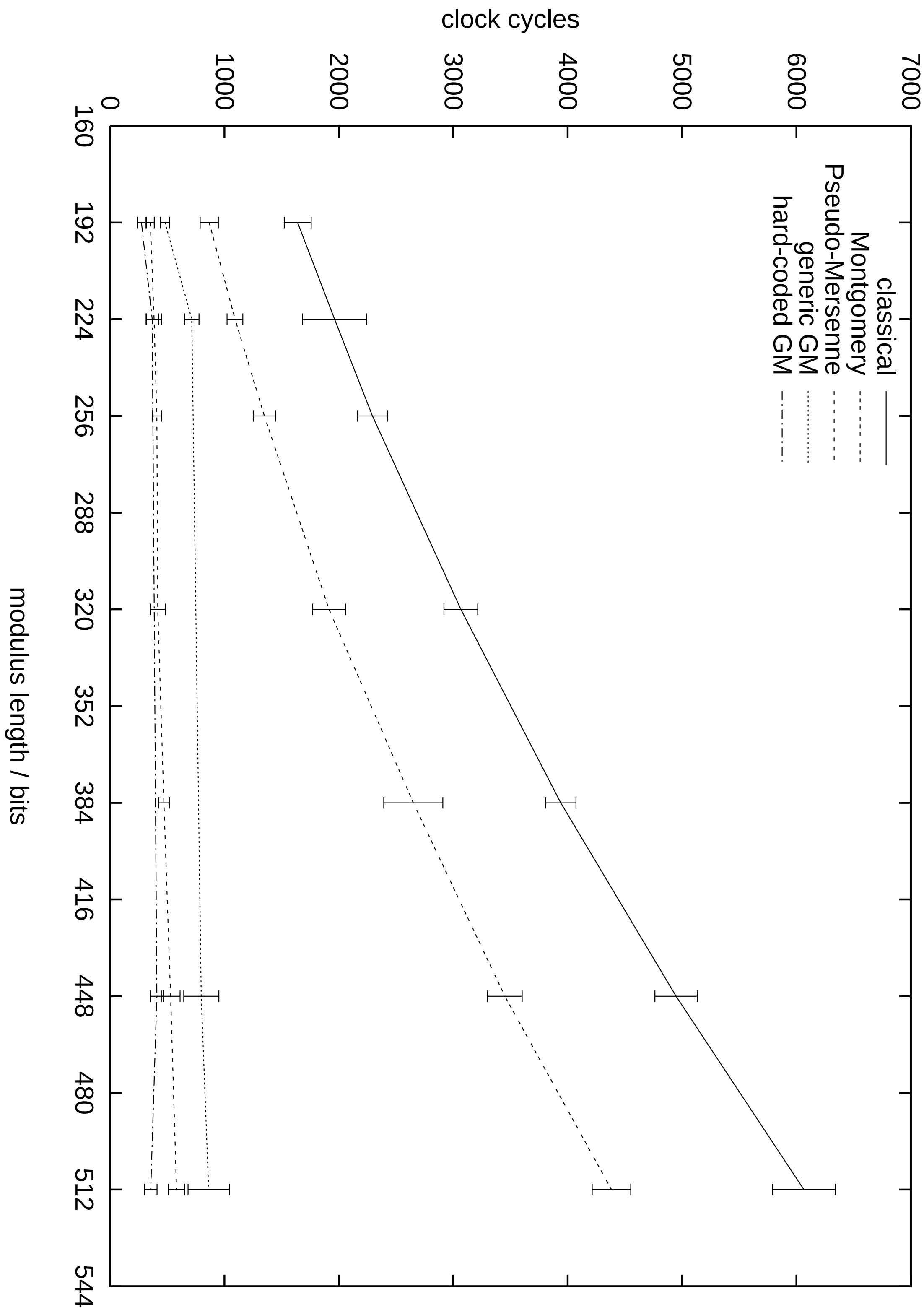

Fig. 3: Modular reduction timings for special prime moduli on a Pentium-II based PC.

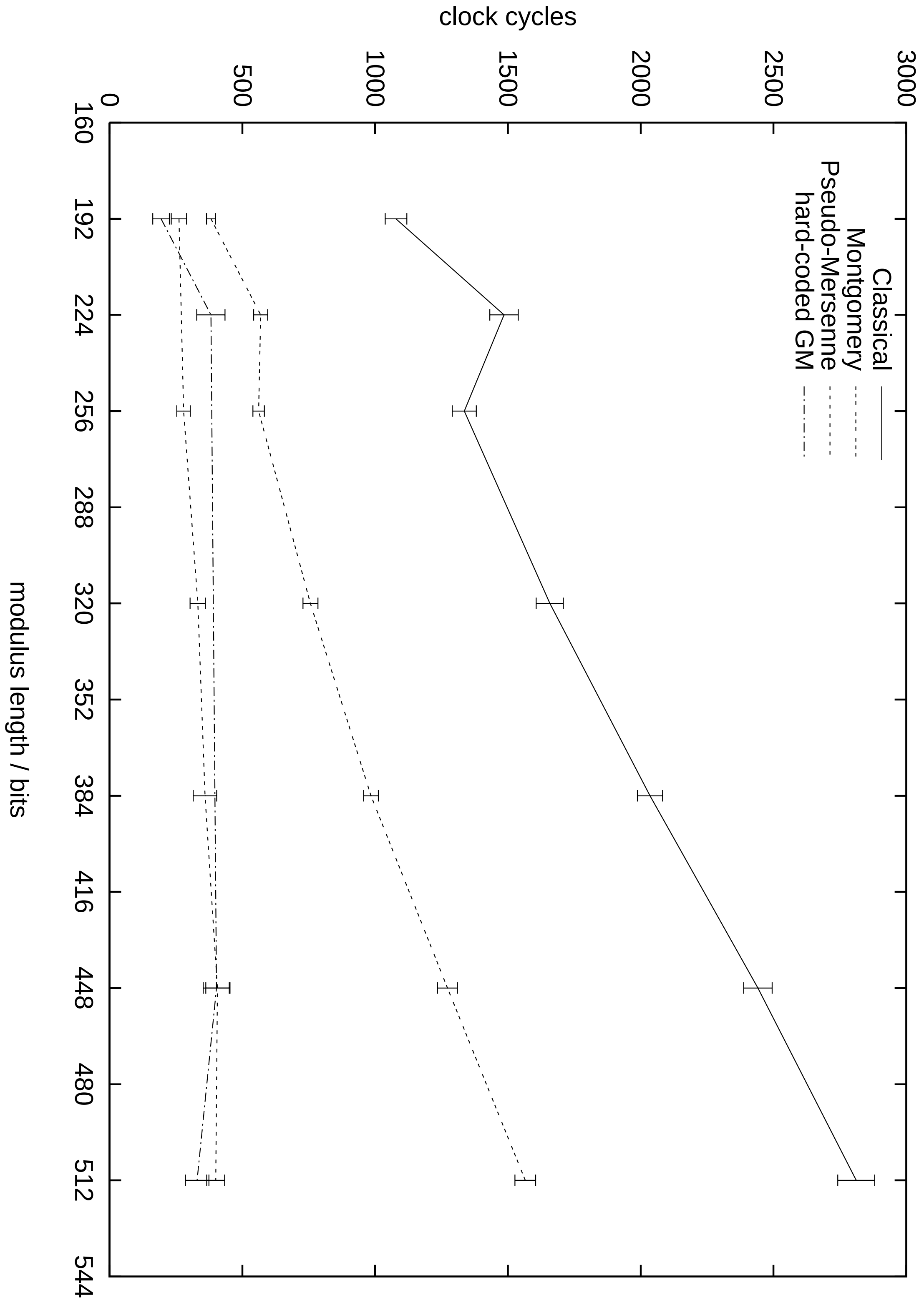

Fig. 4: Modular reduction timings for special prime moduli on an Alpha 21164.

C. Timing Data

Tab. 3: Timing statistics for modular multiplication algorithms on a Pentium-II based PC. Timing values are in clock cycles.

modulus length/bits $\lvert p\rvert_2$	mean timing $\bar{t}$	total error $\sigma + \theta$	deviation σ	error of mean θ	count N
classical reduction					
192	2524.0	122.0	116.9	5.1	500
224	3048.6	131.9	126.4	5.5	500
256	3660.9	185.2	176.5	8.6	400
320	5007.9	208.4	198.6	9.8	400
384	6606.2	230.0	209.8	20.2	100
448	8436.9	436.0	415.1	20.9	400
512	9998.4	268.2	257.1	11.1	500
Pseudo-Mersenne reduction					
192	1403.1	113.7	108.6	5.0	400
224	1651.1	134.6	128.4	6.1	400
256	1930.4	114.0	109.0	5.0	400
320	2511.7	120.0	114.5	5.5	400
384	3314.1	436.2	397.1	39.1	100
448	4145.2	169.2	160.3	8.9	300
512	4503.0	185.0	176.2	8.8	400
generic GM reduction					
192	1360.8	135.4	123.7	11.7	100
224	1820.5	166.9	151.8	15.1	100
448	4233.8	166.3	151.9	14.4	100
512	4615.6	174.8	160.1	14.7	100
hard-coded GM reduction					
192	1146.1	36.9	35.5	1.4	100
224	1463.4	116.3	106.4	9.8	100
448	3848.4	135.8	125.3	10.6	100
512	4109.7	134.6	124.0	10.6	100
Montgomery reduction					
192	1757.6	110.6	105.9	4.8	500
224	2197.2	109.8	105.1	4.7	500
256	2718.8	243.8	232.2	11.6	400
320	3858.5	151.6	144.3	7.3	400
384	5285.0	189.3	172.0	17.4	100
448	6916.6	208.9	198.8	10.1	400
512	8265.5	287.8	275.4	12.4	500

Tab. 4: Timing statistics for modular multiplication algorithms on an Alpha 21164 based workstation. Timing values are in clock cycles.

modulus length/bits $\|p\|_2$	mean timing $\bar{t}$	total error $\sigma + \theta$	deviation σ	error of mean θ	count N
classical reduction					
192	1505.1	37.5	35.8	1.7	500
224	2073.3	43.9	41.8	2.1	500
256	1967.1	39.1	37.0	2.1	400
320	2431.5	50.8	48.4	2.5	400
384	2997.8	49.5	44.9	4.6	100
448	3652.6	58.2	55.3	2.9	400
512	4317.0	51.3	49.0	2.3	500
Pseudo-Mersenne reduction					
192	717.8	36.8	35.9	0.9	400
256	932.1	53.0	51.0	2.0	400
320	1166.1	51.4	49.9	1.6	400
384	1418.9	54.3	51.5	2.8	100
448	1713.1	61.7	59.9	1.8	300
512	2009.1	62.1	59.9	2.2	400
generic GM reduction					
192	911.1	43.7	41.3	2.4	100
hard-coded GM reduction					
192	654.7	41.8	39.3	2.5	100
224	1032.2	55.0	52.2	2.8	100
448	1732.7	66.0	62.9	3.1	100
512	1894.6	52.5	50.3	2.3	100
Montgomery reduction					
192	887.8	27.3	26.1	1.2	500
224	1246.1	28.1	26.8	1.2	500
256	1241.2	26.5	25.5	1.1	400
320	1616.4	36.9	35.1	1.8	400
384	2079.6	41.1	37.4	3.7	100
448	2631.1	45.2	43.2	2.0	400
512	3205.6	52.9	50.6	2.2	500

Tab. 5: Timing statistics for modular reduction algorithms on a Pentium-II based PC. Timing values are in clock cycles.

modulus length/bits $\lvert p\rvert_2$	mean timing $\bar{t}$	total error $\sigma + \theta$	deviation σ	error of mean θ	count N
classical reduction					
192	1640.4	118.1	113.1	4.9	500
224	1963.3	280.0	268.1	11.9	500
256	2293.5	132.3	126.5	5.8	400
320	3066.7	147.1	140.5	6.6	400
384	3940.5	131.9	121.4	10.5	100
448	4948.0	185.5	177.1	8.4	400
512	6064.9	276.2	265.1	11.2	500
Pseudo-Mersenne reduction					
192	352.3	34.2	33.7	0.5	400
224	385.0	65.7	63.2	2.5	400
256	408.6	40.9	40.4	0.5	400
320	417.2	67.3	64.4	2.9	400
384	471.4	45.8	45.0	0.8	100
448	530.0	81.3	77.5	3.8	300
512	580.8	70.7	68.2	2.5	400
generic GM reduction					
192	480.5	39.5	37.7	1.8	100
224	715.0	63.3	61.7	1.7	100
448	797.5	153.6	139.9	13.7	100
512	862.5	180.7	165.3	15.4	100
hard-coded GM reduction					
192	274.9	34.2	33.7	0.5	100
224	368.9	54.5	53.5	1.1	100
448	408.4	55.4	54.2	1.2	100
512	355.9	55.4	53.9	1.6	100
Montgomery reduction					
192	867.0	79.3	76.0	3.4	500
224	1091.9	68.7	65.9	2.8	500
256	1348.4	97.5	92.9	4.6	400
320	1914.6	143.9	137.1	6.8	400
384	2650.5	258.0	238.2	19.8	100
448	3451.1	151.7	144.5	7.2	400
512	4382.5	168.7	161.5	7.2	500

Tab. 6: Timing statistics for modular reduction algorithms on an Alpha 21164 based workstation. Timing values are in clock cycles.

modulus length/bits $\lvert p\rvert_2$	mean timing $\bar{t}$	total error $\sigma+\theta$	deviation σ	error of mean θ	count N
classical reduction					
192	1078.7	40.6	38.7	2.0	500
224	1485.3	53.5	51.2	2.3	500
256	1335.8	45.4	43.2	2.2	400
320	1657.7	51.1	48.7	2.5	400
384	2034.8	47.3	42.6	4.7	100
448	2441.3	53.5	50.8	2.7	400
512	2811.4	69.3	66.3	3.0	500
Pseudo-Mersenne reduction					
192	261.3	28.6	27.9	0.7	400
256	278.7	25.5	25.0	0.5	400
320	332.3	28.9	28.2	0.7	400
384	359.5	44.2	42.0	2.2	100
448	406.5	44.3	43.0	1.2	300
512	399.7	33.8	33.1	0.8	400
generic GM reduction					
192	379.5	33.7	31.3	2.4	100
hard-coded GM reduction					
192	194.1	31.1	30.1	1.0	100
224	381.7	53.4	51.1	2.2	100
448	402.9	50.4	48.4	2.0	100
512	329.8	44.2	42.2	2.0	100
Montgomery reduction					
192	382.3	17.1	16.4	0.7	500
224	569.2	26.4	25.3	1.2	500
256	561.5	21.8	20.8	1.1	400
320	756.5	28.4	27.1	1.3	400
384	984.3	27.8	25.3	2.4	100
448	1272.2	37.6	35.9	1.6	400
512	1565.3	39.1	37.4	1.7	500