AN ABSTRACT OF THE THESIS OF

Tolga Acar for the degree of Doctor of Philosophy in Electrical & Computer Engineering presented on December 4, 1997. Title: High-Speed Algorithms & Architectures For Number-Theoretic Cryptosystems.

Redacted for Privacy

Abstract approved: ____________________

Çetin K. Koç

Computer and network security systems rely on the privacy and authenticity of information, which requires implementation of cryptographic functions. Software implementations of these functions are often desired because of their flexibility and cost effectiveness. In this study, we concentrate on developing high-speed and area-efficient modular multiplication and exponentiation algorithms for number-theoretic cryptosystems.

The RSA algorithm, the Diffie-Hellman key exchange scheme and Digital Signature Standard require the computation of modular exponentiation, which is broken into a series of modular multiplications. One of the most interesting advances in modular exponentiation has been the introduction of Montgomery multiplication. We are interested in two aspects of modular multiplication algorithms: development of fast and convenient methods on a given hardware platform, and hardware requirements to achieve high-performance algorithms.

Arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is large, such as elliptic curve cryptosystems.

High-Speed Algorithms & Architectures For Number-Theoretic Cryptosystems

by

Tolga Acar

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed December 4, 1997
Commencement June 1998

Doctor of Philosophy thesis of Tolga Acar presented on December 4, 1997

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

Chair of the Department of Electrical & Computer Engineering

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Tolga Acar, Author

ACKNOWLEDGMENT

I would like to thank Çetin K. Koç for his encouragement and help, starting his role as my thesis advisor. It is also my priviledge to acknowledge the support of my colleagues and friends. Last, but not least, Barçın, gave me lots of emotional support, without which I could not have succeeded.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Dedicated to my parents,

Ayla and M. Teoman Acar.

HIGH-SPEED ALGORITHMS & ARCHITECTURES FOR NUMBER-THEORETIC CRYPTOSYSTEMS

1. INTRODUCTION

Computer and network security systems rely on the privacy and authenticity of information, which requires implementation of cryptographic functions. Basic functions of information security include secret-key and public-key cryptosystems, message-digest algorithms, and digital signature functions. Software implementations of these algorithms are often desired because of their flexibility and cost effectiveness. However, performance has always been an issue, requiring the algorithm engineer to invent and develop new methods for high-speed implementations. The computational cost of software cryptography is a function of the underlying algorithm and the quality of the implementation of the algorithm [1]. In this study, we concentrate on developing high-speed and area-efficient algorithms for number-theoretic cryptosystems. In this framework, we study modular multiplication and exponentiation in finite fields.

We first investigate fast algorithms for modular multiplication which is a popular operation used in number-theoretic cryptosystems. The RSA algorithm [2], the Diffie-Hellman key exchange scheme [3] and Digital signature standard [4] require the computation of modular exponentiation, which is broken into a series of modular multiplications by the application of the binary or m-ary methods [5]. These algorithms can be implemented in hardware and software [6, 7]. One of the most interesting advances in modular exponentiation has been the introduction of Montgomery multiplication [8]. It became *de facto* algorithm in any cryptographic method utilizing modular multiplication, especially the RSA cryptosystem [9, 10]. We are interested in two aspects of modular multiplication algorithms: development of fast and convenient methods on a given hardware platform, and hardware requirements to achieve high-performance algorithms.

Arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is very large. Examples of the cryptographic applications are the Diffie-Hellman key exchange algorithm [3] based on the discrete exponentiation and elliptic curve cryptosystems [11, 12, 13] over the field GF(2^k). The Diffie-Hellman algorithm requires implementation of the exponentiation g^e, where g is a fixed primitive element of the field and e an integer. In elliptic curves, the exponentiation is used to compute inverse of an element $a \in$ GF(2^k), based on Fermat's identity $a^{-1} = a^{2^k-2}$ [14, 15, 16]. The exponentiation operation can be implemented using a series of squaring and multiplication operations in GF(2^k) using the binary method [5].

In Chapter 2., several Montgomery multiplication algorithms are discussed [8]. We focus on developing time and space efficient algorithms for the Montgomery multiplication. We first study two previously introduced algorithms [17, 18]. We then introduce three new time and space efficient algorithms, and analyze their space and time requirements in detail. These algorithms are implemented in C and in Intel 486 assembler for comparing their time requirements.

In Chapter 3., we show that the operation $c = a \cdot b \cdot r^{-1}$ in the field GF(2^k) can be implemented significantly faster in software than the standard multiplication, where r is a special element of the field [19]. This operation is a finite field analogue of the Montgomery multiplication for modular multiplication of integers [8]. The Montgomery multiplication can be used to achieve fast discrete exponentiation operation in GF(2^k), and is particularly suitable for cryptographic applications where k is large.

In Chapter 4., we examine the implementation issues of number-theoretic cryptographic algorithms (e.g., RSA and Diffie-Hellman), and propose a design methodology for obtaining high-speed implementations. We show that between the full assembler and standard C implementations, there is a design option in which only a small number of code segments are written in assembler. These small code segments are the *kernel* of arithmetic operations for number-theoretic cryptographic algorithms. We propose eight kernel operations. We are currently working on the specification and implementation of this cryptographic kernel.

We have also analyzed the instruction set and related architectural features of the Intel Pentium processor and MMX technology for high-speed implementations of number-theoretic cryptographic algorithms in Chapter 5.. After carefully examining the kernel operations for the number-theoretic cryptographic algorithms, we propose new instructions for the Intel MMX technology.

2. NEW ALGORITHMS FOR MONTGOMERY MULTIPLICATION

2.1. Introduction

This chapter discusses several Montgomery multiplication algorithms, two of which have been proposed before. We describe three additional algorithms, and analyze in detail the space and time requirements of all five methods [20]. These algorithms are implemented in C and in assembler. The analyses and actual performance results indicate that the Coarsely Integrated Operand Scanning (CIOS) method, detailed in this chapter, is the most efficient of all five algorithms, at least for the general class of processor we considered. The Montgomery multiplication methods constitute the core of the modular exponentiation operation which is the most popular method used in public-key cryptography for encrypting and signing digital data.

The motivation for studying high-speed and space-efficient algorithms for modular multiplication comes from their applications in public-key cryptography. The RSA algorithm [2] and the Diffie-Hellman key exchange scheme [3] require the computation of modular exponentiation, which is broken into a series of modular multiplications by the application of the binary or m-ary methods [5]. Various hardware algorithms for modular multiplication have been proposed [21, 22, 23]. Modular exponentiation algorithms using division chains [24], a double-base number system [25], and complex arithmetic [26] are applicable to software implementations. However, these methods concentrate on fast modular exponentiation, not on the particular modular multiplication method employed.

Certainly one of the most interesting and useful advances has been the introduction of the so-called Montgomery multiplication algorithm due to Peter L. Mongtomery [8] (for some of the recent applications see the discussion by Naccache *et al.* [27], Koç *et al.* [20] and Bajard *et al.* [28]). Various hardware implementations of the Montgomery multiplication have been proposed and some of them have been used in commercially available chips [29, 9, 30]. The Montgomery multiplication algorithm is used to speed up the modular multiplications and squarings required during the exponentiation process. The Montgomery algorithm computes

$$\text{MonPro}(a, b) = a \cdot b \cdot r^{-1} \bmod n \tag{2.1}$$

given $a, b < n$ and r such that $\gcd(n, r) = 1$. Even though the algorithm works for any r which is relatively prime to n, it is more useful when r is taken to be a power of 2. In this case, the Montgomery algorithm performs divisions by a power of 2, which is an intrinsically fast operation on general-purpose computers, e.g., signal processors and microprocessors; this leads to a simpler implementation than ordinary modular multiplication, which is typically faster as well [27].

In this chapter, we study the operations involved in the computing the Montgomery product, describe several high-speed and space-efficient algorithms for computing MonPro(a, b), and analyze their time and space requirements. Our focus is to collect together several alternatives for Montgomery multiplication, three of which are new.

2.2. Montgomery Multiplication

Let the modulus n be a k-bit integer, i.e., $2^{k-1} \leq n < 2^k$, and let r be 2^k. The Montgomery multiplication algorithm requires that r and n be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if n is odd. A modified Montgomery multiplication has also been introduced for an even modulus [31]. In order to describe the Mongtomery multiplication algorithm, we first define the n-residue of an integer $a < n$ as $\bar{a} = a \cdot r \pmod{n}$. It is straightforward to show that the set

$$\{ \, a \cdot r \bmod n \mid 0 \leq a \leq n - 1 \, \}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{a}$ and $\bar{b}$, the Montgomery product is defined as the n-residue

$$\bar{c} = \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \ , \tag{2.2}$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property $r^{-1} \cdot r = 1 \pmod{n}$. The resulting number c in (2.2) is the n-residue of the product $c = a \cdot b \pmod{n}$, since

$$\begin{aligned} \bar{c} &= \bar{a} \cdot \bar{b} \cdot r^{-1} \pmod{n} \\ &= a \cdot r \cdot b \cdot r \cdot r^{-1} \pmod{n} \\ &= c \cdot r \pmod{n} \ . \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property $r \cdot r^{-1} - n \cdot n' = 1$. The integers r^{-1} and n' can both be computed by the extended Euclidean algorithm [5, 32]. The computation of MonPro$(\bar{a}, \bar{b})$ is achieved as follows:

```
function MonPro($\bar{a}, \bar{b}$)
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $u := (t + (t \cdot n' \bmod r) \cdot n)/r$
Step 3. if $u \geq n$ then return $u - n$ else return $u$
```

Multiplication modulo r and division by r are both intrinsically fast operations, since r is a power of 2. Thus the Montgomery product algorithm is potentially faster and simpler than ordinary computation of $a \cdot b \bmod n$, which involves division by n. However, since conversion from an ordinary residue to an n-residue, computation of n', and conversion back to an ordinary residue are time-consuming, it is not a good idea to use the Montgomery product computation algorithm when a single modular multiplication is to be performed. It is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute modular exponentiation. Using the binary method for computing the powers [5], we replace the exponentiation by a series of square and multiplication operations modulo n. Let j be the number of bits in the exponent e. The following exponentiation algorithm is one way to compute $x := a^e \bmod n$ with $O(j)$ calls to the Montgomery multiplication algorithm. Step 4 of the modular exponentiation algorithm computes x using $\bar{x}$ via the property of the Montgomery algorithm: $\text{MonPro}(\bar{x}, 1) = \bar{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \bmod n$.

```
function ModExp($a, e, n$)
Step 1.  $\bar{a} := a \cdot r \bmod n$
Step 2.  $\bar{x} := 1 \cdot r \bmod n$
Step 3.  for $i = j - 1$ downto 0
             $\bar{x}$ :=MonPro($\bar{x}, \bar{x}$)
             if $e_i = 1$  then $\bar{x}$ :=MonPro($\bar{x}, \bar{a}$)
Step 4.  ret $x$ :=MonPro($\bar{x}, 1$)
```

In typical implementations, large numbers are stored by breaking the numbers into words. If w is the wordsize of the computer, then a number can be thought of as a sequence of integers each represented in radix $W = 2^w$. If these "multi-precision" numbers require s words in the radix W representation, then we take r as $r = 2^{sw}$.

In the following sections, we give several algorithms for doing the Montgomery multiplication MonPro(a, b), and analyze their time and space requirements. We counted the total number of multiplications, additions (subtractions), and memory read and write operations in terms of the input size parameter s for the time analysis. For example, the following operation

```
(C,S) := t[i+j] + a[j]*b[i] + C
```

is assumed to require three memory reads, two additions, and one multiplication since most microprocessors multiply two one-word numbers, leaving the two-word result in one or two registers.[1]

Multi-precision integers are assumed to reside in memory throughout the computations. Therefore, the assignment operations performed within a routine correspond to the read or write operations between a register and memory. They are counted to calculate the proportion of the memory access time in the total running time of the Montgomery multiplication algorithm. In our analysis, loop establishment and index computations are not taken into account. The only registers we assume are available are those to hold the carry C and the sum S as above (or equivalently, borrow and difference for subtraction). Obviously, in many microprocessors there will be more registers, but this gives a first-order approximation to the running time, sufficient for a general comparison of the approaches. Actual implementation on particular processors gives a more detailed comparison.

The space analysis is performed by counting the total number of words used as the temporary space. However, the space required to keep the input and output values a, b, n, n'_0, and u is not taken into account.

2.3. Algorithms

There are a variety of ways to perform the Montgomery multiplication, just as there are many ways to multiply. Our purpose in this chapter is to give fairly broad coverage of the alternatives.

Roughly speaking, we may organize the algorithms based on two factors [20]. The first factor is whether multiplication and reduction are *separated* or *integrated*. In the

[1] We note that in some processors the additions may actually involve two instructions each, since the value `+a[j]*b[i]` is double-precision; we ignore this distinction in our timing estimates.

separated approach, we first multiply a and b, then perform a Montgomery reduction. In the integrated approach, we alternate between multiplication and reduction. This integration can be either *coarse-grained* or *fine-grained*, depending on how often we switch between multiplication and reduction (i.e., after processing an array of words, or just one word); there are implementation tradeoffs between alternatives.

The second factor is the general form of the multiplication and reduction steps. One form is the *operand scanning*, where an outer loop moves through words of one of the operands; another form is *product scanning*, where the loop moves through words of the product itself [17]. This factor is independent of the first; moreover, it is also possible for multiplication to have one form and reduction to have the other form, even in the integrated approach.

In all the cases we consider, the algorithms are described as operations on multi-precision numbers. Thus it is straightforward to rewrite the algorithms in an arbitrary radix, e.g., in binary or radix-4 form for hardware.

Clearly, the foregoing discussion suggests that quite a few algorithms are possible, but in this chapter we focus on five as representative of the whole set, and which for the most part have good implementation characteristics. The five algorithms we discuss include the following:

- Separated Operand Scanning (SOS) (Section 2.3.1.)
- Coarsely Integrated Operand Scanning (CIOS) (Section 2.3.2.)
- Finely Integrated Operand Scanning (FIOS) (Section 2.3.3.)
- Finely Integrated Product Scanning (FIPS) (Section 2.3.4.)
- Coarsely Integrated Hybrid Scanning (CIHS) (Section 2.3.5.)

Other possibilities are variants of one or more of these five; we encourage the interested reader to construct and evaluate some of them. Two of these methods have been described previously, SOS (as Improvement 1 in [18]) and FIPS (in [17]). The other three, while suggested by previous work, have been first described in detail or analyzed in comparison with the others in [20].

2.3.1. The Separated Operand Scanning (SOS) Method

The first method to be analyzed for computing MonPro(a, b) is what we call the Separated Operand Scanning method (see Improvement 1 in [18]). In this method we first compute the product $a \cdot b$ using

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    t[i+s] := C
```

where t is initially assumed to be zero. The final value obtained is the $2s$-word integer t residing in words

```
t[0], t[1], ... , t[2s-1]
```

Then we compute u using the formula $u := (t + m \cdot n)/r$, where $m := t \cdot n' \bmod r$. In order to compute u, we first take $u = t$, and then add $m \cdot n$ to it using the standard multiplication routine, and finally divide it by $r = 2^{sw}$ which is accomplished by ignoring the lower s words of u. Since $m = t \cdot n' \bmod r$ and the reduction process proceeds word by word, we can use $n'_0 = n' \bmod 2^w$ instead of n'. This observation was first made in [18], and applies to all five methods presented in this chapter. Thus, after t is computed by multiplying a and b using the above code, we proceed with the following code which updates t in order to compute $t + m \cdot n$.

```
for i=0 to s-1
    C := 0
```

```
        m := t[i]*n'[0] mod W
        for j=0 to s-1
            (C,S) := t[i+j] + m*n[j] + C
            t[i+j] := S
        ADD (t[i+s],C)
```

The ADD function shown above performs a carry propagation adding C to the input array given by the first argument, starting from the first element (t[i+s]), and propagates it until no further carry is generated. The ADD function is needed for carry propagation up to the last word of t, which increases the size of t to $2s$ words and a single bit. However, this bit is saved in a single word, increasing the size of t to $2s+1$ words.[2] The computed value of t is then divided by r which is realized by simply ignoring the lower s words of t. These steps are given below:

```
    for j=0 to s
        u[j] := t[j+s]
```

Finally we obtain the number u in $s+1$ words. The multi-precision subtraction in Step 3 of MonPro is then performed to reduce u if necessary. Step 3 can be performed using the following code:

```
    B := 0
    for i=0 to s-1
        (B,D) := u[i] - n[i] - B
        t[i] := D
    (B,D) := u[s] - B
    t[s] := D
```

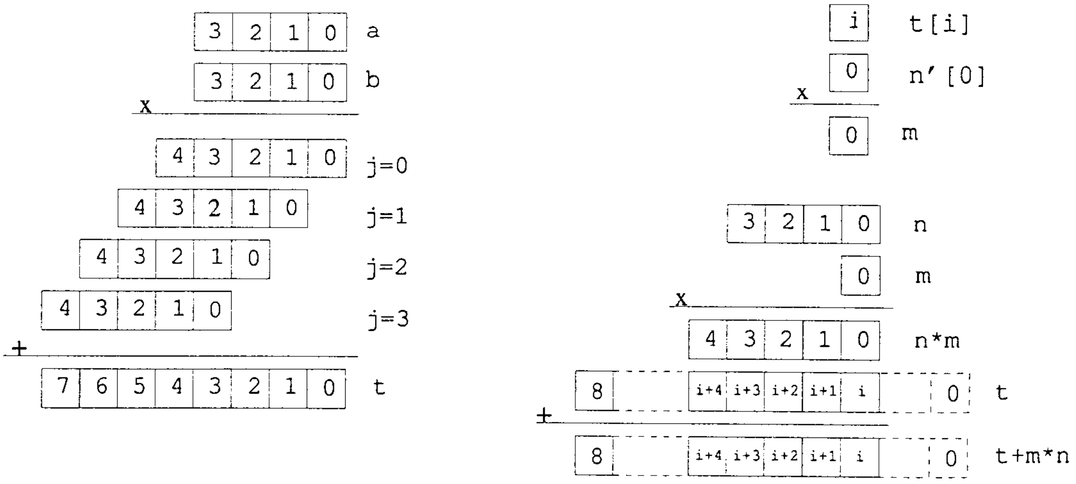
[2] This extra bit, and hence an extra word, is required in all the methods described. One way to avoid the extra word in most cases is to define s as the length in words of $2n$, rather than the modulus n itself. This s will be the same as in the current definition, except when the length of n is a multiple of the word size, and in that case only one larger than currently.

```
    if B=0 then return t[0], t[1], ... , t[s-1]
           else return u[0], u[1], ... , u[s-1]
```

Step 3 is performed in the same way for all algorithms described in this chapter, and thus, we do not repeat this step in the description of the algorithms. However, its time and space requirements will be taken into account. The operations above contain $2(s+1)$ additions, $2(s+1)$ reads, and $s+1$ writes.

A brief inspection of the SOS method, based on our techniques for counting the number of operations, shows that it requires $2s^2+s$ multiplications, $4s^2+4s+2$ additions, $6s^2+7s+3$ reads, and $2s^2+6s+2$ writes. (See Section 2.4. for discussion of how to count the number of operations required by the ADD function.) Furthermore, the SOS method requires a total of $2s+2$ words for temporary results, which are used to store the $(2s+1)$-word array t and the one-word variable m. The SOS method is illustrated in Figure 2.1 for $s=4$.

FIGURE 2.1: The Separated Operand Scanning (SOS) method for $s=4$. The multiplication operation $t=a\times b$ is illustrated on the left. Then, n'_0 is multiplied by each word of t to find m. The final result is obtained by adding the shifted $n\times m$ to t, as shown on the right.



The value n_0', which is defined as the inverse of the least significant word of n modulo 2^w, i.e., $n_0' = -n_0^{-1} \pmod{2^w}$, can be computed using a very simple algorithm given in [18]. Furthermore, the reason for separating the product computation $a \cdot b$ from the rest of the steps for computing u is that when $a = b$, we can optimize the Montgomery multiplication algorithm for squaring. The optimization of squaring is achieved because almost half of the single-precision multiplications can be skipped since $a_i \cdot a_j = a_j \cdot a_i$. The following simple code replaces the first part of the Mongtomery multiplication algorithm in order to perform the optimized Montgomery squaring:

```
for i=0 to s-1
    (C,S) := t[i+i] + a[i]*a[i]
    for j=i+1 to s-1
        (C,S) := t[i+j] + 2*a[j]*a[i] + C
        t[i+j] := S
    t[i+s] := C
```

(One tricky part here is that the value `2*a[j]*a[i]` requires more than two words to store; if the C value does not have an extra bit, then one way to deal with this is to rewrite the loop so that the `a[j]*a[i]` terms are added first, without the multiplication by 2; the result is then doubled and the `a[i]*a[i]` terms are added in.) In this chapter, we analyze only the Montgomery multiplication algorithms. The analysis of Montgomery squaring can be performed similarly.

2.3.2. The Coarsely Integrated Operand Scanning (CIOS) Method

The next method, the Coarsely Integrated Operand Scanning method, improves on the first one by integrating the multiplication and reduction steps. Specifically, instead of computing the entire product $a \cdot b$, then reducing, we alternate between iterations of the outer loops for multiplication and reduction. We can do this since the value of m in the ith iteration of the outer loop for reduction depends only on the value `t[i]`, which is completely computed by the *i*th iteration of the outer loop for the multiplication. This leads to the following algorithm:

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[j] + a[j]*b[i] + C
        t[j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := C
    C := 0
    m := t[0]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[j] + m*n[j] + C
        t[j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := t[s+1] + C
    for j=0 to s
        t[j] := t[j+1]
```

Note that the array t is assumed to be set to 0 initially. The last j-loop is used to shift the result one word to the right (i.e., division by 2^w), hence the references to `t[j]` and `t[0]` instead of `t[i+j]` and `t[i]`. A slight improvement is to integrate the shifting into the reduction as follows:

```
    m := t[0]*n'[0] mod W
    (C,S) := t[0] + m*n[0]
    for j=1 to s-1
```

```
        (C,S) := t[j] + m*n[j] + C
        t[j-1] := S
    (C,S) := t[s] + C
    t[s-1] := S
    t[s] := t[s+1] + C
```

The auxiliary array t uses only $s + 2$ words. This is due to fact that the shifting is performed one word at a time, rather than s words at once, saving $s - 1$ words. The final result is in the first $s + 1$ words of array t. A related method, without the shifting of the array (and hence with a larger memory requirement), is described as Improvement 2 in [18].

The CIOS method (with the slight improvement above) requires $2s^2 + s$ multiplications, $4s^2 + 4s + 2$ additions, $6s^2 + 7s + 2$ reads, and $2s^2 + 5s + 1$ writes, including the final multi-precision subtraction, and uses $s + 3$ words of memory space. The memory reduction is a significant improvement over the SOS method.

We say that the integration in this method is "coarse" because it alternates between iterations of the outer loop. In the next method, we will alternate between iterations of the inner loop.

2.3.3. The Finely Integrated Operand Scanning (FIOS) Method

This method integrates the two inner loops of the CIOS method into one by computing the multiplications and additions in the same loop. The multiplications $a_j \cdot b_i$ and $m \cdot n_j$ are computed in the same loop, and then added to form the final t. In this case, t_0 must be computed before entering into the loop since m depends on this value which corresponds to unrolling the first iteration of the loop for $j = 0$.

```
for i=0 to s-1
    (C,S) := t[0] + a[0]*b[i]
    ADD(t[1],C)
    m := S*n'[0] mod W
```

```
(C,S) := S + m*n[0]
```

The partial products of $a \cdot b$ are computed one by one for each value of i, then $m \cdot n$ is added to the partial product. This sum is then shifted right one word, making t ready for the next i-iteration.

```
for j=1 to s-1
    (C,S) := t[j] + a[j]*b[i] + C
    ADD(t[j+1],C)
    (C,S) := S + m*n[j]
    t[j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1] + C
t[s+1] := 0
```

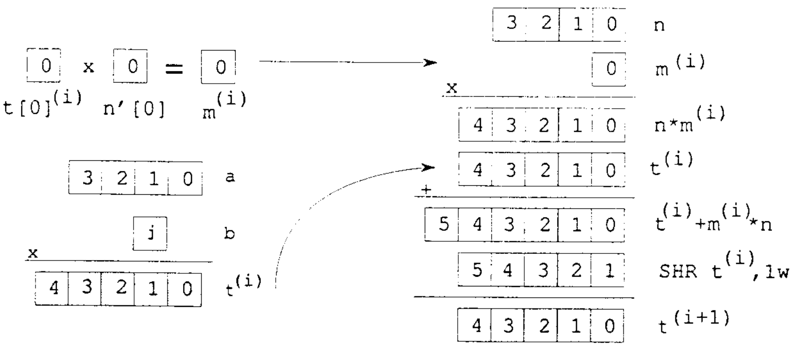
The difference between the CIOS method and this method is that the FIOS method has only one inner loop. We illustrate the algorithm in Figure 2.2 for $s = 4$. The use of the ADD function is required in the inner j-loop since there are two distinct carries, one arising from the multiplication of $a_j \cdot b_i$ and the other from the multiplication of $m \cdot n_j$. (Thus the benefit of having only one loop is counterbalanced by the requirement of the ADD function.) The array t is assumed to be set to 0 initially.

The FIOS method requires $2s^2+s$ multiplications, $5s^2+3s+2$ additions, $7s^2+5s+2$ reads, and $3s^2 + 4s + 1$ writes, including the final multi-precision subtraction. This is about s^2 more additions, writes, and reads than for the CIOS method. The total amount of temporary space required is $s + 3$ words.

2.3.4. The Finely Integrated Product Scanning (FIPS) Method

Like the previous one, this method interleaves the computations $a \cdot b$ and $m \cdot n$, but here both computations are in the product-scanning form. The method keeps the values

FIGURE 2.2: An iteration of the Finely Integrated Operand Scanning (FIOS) method. The computation of partial product $t^{(i)} = a \times b_i$, illustrated on the left, enables the computation of $m^{(i)}$ in that iteration. Then an intermediate result $t^{(i+1)}$ is found by adding $n \times m^{(i)}$ to this partial product, as shown on the right.



of m and u in the same s-word array m. This method was described in [17] and is related to Improvement 3 in [18]. The first loop given below computes one part of the product $a \cdot b$ and then adds $m \cdot n$ to it. The three-word array t, i.e.,

```
t[0], t[1], t[2],
```

is used as the partial product accumulator for the products $a \cdot b$ and $m \cdot n$.[3]

```
for i=0 to s-1
    for j=0 to i-1
        (C,S) := t[0] + a[j]*b[i-j]
        ADD(t[1],C)
        (C,S) := S + m[j]*n[i-j]
        t[0] := S
        ADD(t[1],C)
```

[3] The use of a three-word array assumes that $s < W$; in general, we need $\log_W(sW(W-1)) \approx 2+\log_W s$ words. The algorithm is easily modified to handle a larger accumulator.

```
        (C,S) := t[0] + a[i]*b[0]
        ADD(t[1],C)
        m[i] := S*n'[0] mod W
        (C,S) := S + m[i]*n[0]
        ADD(t[1],C)
        t[0] := t[1]
        t[1] := t[2]
        t[2] := 0
```

In this loop, the ith word of m is computed using n'_0, and then the least significant word of $m \cdot n$ is added to t. Since the least significant word of t always becomes zero, the shifting can be carried out one word at a time in each iteration. The array t is assumed to be set to 0 initially.

The second i-loop, given below, completes the computation by forming the final result u word by word in the memory space of m.

```
    for i=s to 2s-1
        for j=i-s+1 to s-1
            (C,S) := t[0] + a[j]*b[i-j]
            ADD(t[1],C)
            (C,S) := S + m[j]*n[i-j]
            t[0] := S
            ADD(t[1],C)
        m[i-s] := t[0]
m[i-s+1] := t[1]
        t[0] := t[1]
        t[1] := t[2]
        t[2] := 0
```

An inspection of indices in the second i-loop shows that the least significant s words of the result u are located in the variable m. The most significant bit is in `t[0]`. (The values `t[1]` and `t[2]` are zero at the end.)

The FIPS method requires $2s^2+s$ multiplications, $6s^2+2s+2$ additions, $9s^2+8s+2$ reads, and $5s^2+8s+1$ writes. The number of additions, reads and writes is somewhat more than for the previous methods, but the number of multiplications is the same. The method nevertheless has considerable benefits on digital signal processors, as discussed in Section 2.4.. (Note that many of the reads and writes are for the accumulator words, which may be in registers.) The space required is $s+3$ words.

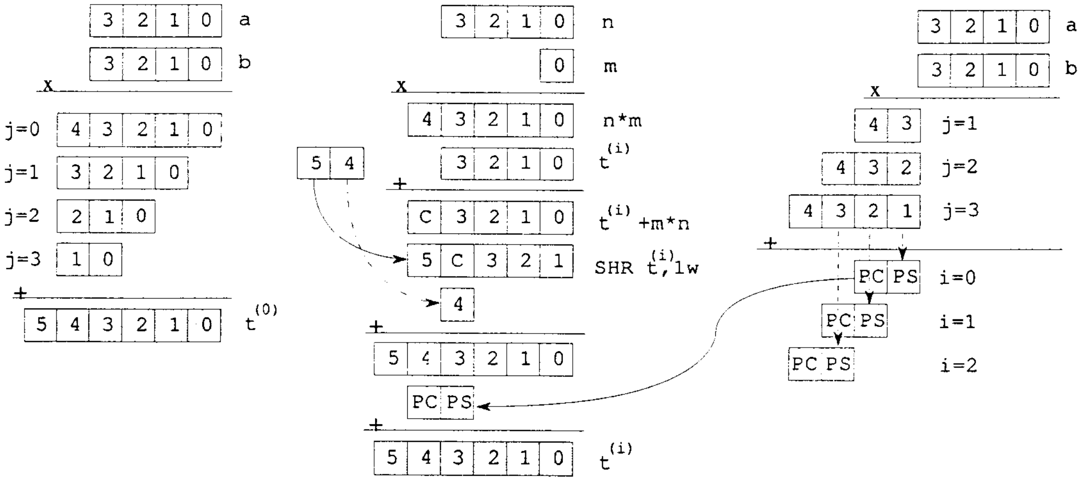
2.3.5. The Coarsely Integrated Hybrid Scanning (CIHS) Method

This method is a modification of the SOS method, illustrating yet another approach to Montgomery multiplication. As was shown, the SOS method requires $2s+2$ words to store the temporary variables t and m. Here we show that it is possible to use only $s+3$ words of temporary space, without changing the general flow of the algorithm. We call it a "hybrid scanning" method because it mixes the product-scanning and operand-scanning forms of multiplication. (Reduction is just in the operand-scanning form.) First, we split the computation of $a \cdot b$ into two loops. The second loop shifts the intermediate result one word at a time at the end of each iteration.

The splitting of multiplication is possible because m is computed by multiplying the ith word of t by n'_0. Thus, the multiplication $a \cdot b$ can be simplified by postponing the word multiplications required for the most significant half of t to the second i-loop. The multiplication loop can be integrated into the second main i-loop, computing one partial product in each iteration and reducing the space for the t array to $s+2$ words from $2s+1$ words. In the first stage, $(n-j)$ words of the jth partial product of $a \cdot b$ are computed and added to t. In Figure 2.3, the computed parts of the partial products are shown by straight lines, and the added result is shown by shaded blocks. This computation can be performed using the following code:

```
for i=0 to s-1
    C := 0
    for j=0 to s-i-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := C
```

FIGURE 2.3: An iteration of the the Coarsely Integrated Hybrid Scanning (CIHS) method for $s = 4$. The left-hand side figure shows the accumulation of the right half of the partial products of $a \times b$ which is performed in the first i-loop. The second i-loop is depicted in two parts in the middle and the right. The addition of $n \times m$ to t and the shifting of $t + m \times n$ are illustrated in the middle, which are performed in the first j-loop of the second i-loop. The computation of the remaining words of the partial products of $a \times b$ is illustrated on the right-hand side. Each (PC,PS) pair is the sum of the columns connected with lines. As illustrated in the bottom of the middle part, the (PC,PS) pair is added to $t^{(i)}$, which is performed in the last j-loop.



The multiplication of $m \cdot n$ is then interleaved with the addition $a \cdot b + m \cdot n$. The division by r is performed by shifting one word at a time within the i-loop. Since m is one word long and the product $m \cdot n + C$ is two words long, the total sum $t + m \cdot n$ needs at most $s + 2$ words. Also note that the carry propagation into the sth word is performed into the $(s - 1)$st word after the shifting. The array t is assumed to be set to 0 initially.

```
for i=0 to s-1
    m := t[0]*n'[0] mod W
    (C,S) := t[0] + m*n[0]
    for j=1 to s-1
        (C,S) := t[j] + m*n[j] + C
        t[j-1] := S
    (C,S) := t[s] + C
    t[s-1] := S
    (C,S) := t[s+1] + C
    t[s] := S
    t[s+1] := C
```

The computation of m requires the use of t_0 instead of t_i, as in the original SOS algorithm. This is due to the shifting of t in each iteration. The two excess words computed in the first loop are used in the following j-loop which computes the $(s + i)$th word of $a \cdot b$.

```
    for j=i+1 to s-1
        (C,S) := t[s-1] + b[j]*a[s-j+i]
        t[s-1] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := t[s+1] + C
```

We note that the above four lines compute the most significant three words of t, i.e., the $(s-1)$st, sth, and $(s+1)$st words of t. The above code completes Step 1 of MonPro(a, b). After this, n is subtracted from t if $t \geq n$. We illustrate the algorithm in Figure 3 for

Montgomery multiplication of two four-word numbers. Here, the symbols PC and PS denote the two extra words required to obtain the correct $(s+i)$th word. Each PC, PS pair is the sum of their respective words connected by vertical dashed lines in Figure 3. The number of multiplications required in this method is also equal to $2s^2 + s$. However, the number of additions decreases to $4s^2 + 4s + 2$. The number of reads is $6.5s^2 + 6.5s + 2$ and the number of writes is $3s^2 + 5s + 1$. As was mentioned earlier, this algorithm requires $s + 3$ words of temporary space.

2.4. Results and Conclusions

The algorithms presented in this chapter require the same number of single-precision multiplications, however, the number of additions, reads and writes are slightly different. There seems to be a lower bound of $4s^2 + 4s + 2$ for addition operations. The SOS and CIOS methods reach this lower bound. The number of operations and the amount of temporary space required by the methods are summarized in Table 2.1. The total number of operations is calculated by counting each operation within a loop, and multiplying this number by the iteration count. As an example we illustrate the calculation for the CIOS method in Table 2.2.

We note that the `ADD(x[i],C)` function, which implements the operation `x[i] := x[i] + C` including the carry propagation, requires one memory read (`x[i]`), one addition (`x[i]+C`) and one memory write (`x[i]:=`) operation during the first step. Considering the carry propagation from this addition, on average one additional memory read, one addition, and one memory write is performed (in addition to the branching and loop instructions). Thus, the `ADD` function is counted as two memory reads, two additions, and two memory writes in our analysis.

Clearly, our counting is only a first-order approximation; we are not taking into account the full use of registers to store intermediate values, cache size in the data and

TABLE 2.1: The time and space requirements of the methods.

Method	Multiplications	Additions	Reads	Writes	Space
SOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 3$	$2s^2 + 6s + 2$	$2s + 2$
CIOS	$2s^2 + s$	$4s^2 + 4s + 2$	$6s^2 + 7s + 2$	$2s^2 + 5s + 1$	$s + 3$
FIOS	$2s^2 + s$	$5s^2 + 3s + 2$	$7s^2 + 5s + 2$	$3s^2 + 4s + 1$	$s + 3$
FIPS	$2s^2 + s$	$6s^2 + 2s + 2$	$9s^2 + 8s + 2$	$5s^2 + 8s + 1$	$s + 3$
CIHS	$2s^2 + s$	$4s^2 + 4s + 2$	$6.5s^2 + 6.5s + 2$	$3s^2 + 5s + 1$	$s + 3$

instruction misses, and the special instructions such as multiply and accumulate. We also did not count loop overhead, pointer arithmetic, and the like, which will undoubtedly affect performance.

In order to measure the actual performance of these algorithms, we implemented them in C and in Intel 386-family assembler on an Intel Pentium-60 Linux system. Table 2.3 summarizes the timings of these methods for $s = 16, 32, 48$, and 64. These correspond to 512, 1024, 1536, and 2048 bits since $w = 32$. The timing values given in Table 2.3 are in milliseconds, and are the average values over several thousand executions. The timing values given in Table 2.3 are in milliseconds, and are the average values over one thousand executions including the overhead of the loop that calls the MonPro function. The table also contains the compiled object code sizes of each algorithm which is important when the principles of locality and instruction cache size are considered.

In the C version of the functions, the single-precision (32-bit) multiplications are realized by dividing them into two 16-bit words. The C version of the function has more overhead compared to the assembler version, in which 32-bit multiplication operations are carried out using a single assembler instruction. The assembler version of the `ADD`

TABLE 2.2: Calculating the operations of the CIOS method.

STATEMENT	Operation				Iterations
	Mult	Add	Read	Write	
`for i=0 to s-1`	-	-	-	-	-
`C := 0`	0	0	0	0	s
`for j=0 to s-1`	-	-	-	-	-
`(C,S) := t[j] + b[j]*a[i] + C`	1	2	3	0	s^2
`t[j] := S`	0	0	0	1	s^2
`(C,S) := t[s] + C`	0	1	1	0	s
`t[s] := S`	0	0	0	1	s
`t[s+1] := C`	0	0	0	1	s
`m := t[0]*n'[0] mod W`	1	0	2	1	s
`(C,S) := t[0] + m*n[0]`	1	1	3	0	s
`for j=1 to s-1`	-	-	-	-	-
`(C,S) := t[j] + m*n[j] + C`	1	2	3	0	$s(s-1)$
`t[j-1] := S`	0	0	0	1	$s(s-1)$
`(C,S) := t[s] + C`	0	1	1	0	s
`t[s-1] := S`	0	0	0	1	s
`t[s] := t[s+1] + C`	0	1	1	1	s
Final Subtraction	0	$2(s+1)$	$2(s+1)$	$s+1$	1
	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	

function is optimized to use one 32-bit register for addition and a 32-bit register for address computation. The propagation of the carry is performed using the carry flag.

The CIOS and FIOS methods are similar to one another in their use of embedded shifting and interleaving the products $a_i \cdot b$ and $m \cdot n_j$. The only difference is that CIOS method computes the partial product $a_i \cdot b$ by using a separate j-loop. Then, the accumulation of $m \cdot n_j$ to this partial product is performed in the succeeding j-loop. The FIOS method combines the computation of partial product $a_i \cdot b$ and accumulation of $a_i \cdot b$ and $m \cdot n_j$ in one single j-loop, thereby obligating the use of the ADD function for propagation of two separate carries.

The CIOS algorithm operates faster on the selected processor compared to the

TABLE 2.3: The timing values of MonPro in milliseconds on a Pentium-60 Linux system. The assembly code is for the Intel 386 family; further improvements may be possible by exploiting particular features of the Pentium.

	512 bits		1024 bits		1536 bits		2048 bits		Code size (bytes)	
Method	C	ASM	C	ASM	C	ASM	C	ASM	C	ASM
SOS	1.376	0.153	5.814	0.869	13.243	2.217	23.567	3.968	1084	1144
CIOS	1.249	0.122	5.706	0.799	12.898	1.883	23.079	3.304	1512	1164
FIOS	1.492	0.135	6.520	0.860	14.550	2.146	26.234	3.965	1876	1148
FIPS	1.587	0.149	6.886	0.977	15.780	2.393	27.716	4.310	2832	1236
CIHS	1.662	0.151	7.268	1.037	16.328	2.396	29.284	4.481	1948	1164

other Montgomery multiplication algorithms, especially when implemented in assembly language. However, on other classes of processor, a different algorithm may be preferable. For instance, on a digital signal processor, we have often found the FIPS method to be better because it exploits the "multiply-accumulate" architecture typical with such processors, where a set of products are added together. On such architectures, the three words $t[0]$, $t[1]$ and $t[2]$ are stored in a single hardware accumulator, and the product `a[j]*b[i-j]` in the FIPS j-loop can be added directly to the accumulator, which makes the j-loop very fast.

Dedicated hardware designs will have additional tradeoffs, based on the extent to which the methods can be parallelized; we do not make any recommendations here, but refer the reader to Even's description of a systolic array as one example of such a design [33]. On a general-purpose processor, the CIOS algorithm is probably the best, as it is the simplest of all five methods, and it requires fewer additions and fewer assignments than the other four methods. The CIOS method requires only $s+3$ words of temporary space, which is just slightly more than half the space required by the SOS algorithm.

3. MONTGOMERY MULTIPLICATION IN GF(2^k)

3.1. Introduction

Arithmetic operations in the Galois field GF(2^k) have several applications in coding theory, computer algebra, and cryptography. We are especially interested in cryptographic applications where k is very large. Examples of the cryptographic applications are the Diffie-Hellman key exchange algorithm [3] based on the discrete exponentiation and elliptic curve cryptosystems [34, 11, 12, 35] over the field GF(2^k). The Diffie-Hellman algorithm requires computation of g^e, where g is a fixed primitive element of the field and e is an integer. In elliptic curves, the exponentiation operation is used to compute inverse of an element in GF(2^k), based on Fermat's identity $a^{-1} = a^{2^k-2}$ [14, 15, 16, 36].

Cryptographic applications require fast hardware and software implementations of the arithmetic operations in GF(2^k) for large values of k. The most important advance in this field has been the Massey-Omura algorithm [37] which is based on the normal bases. Subsequently, the optimal normal bases were introduced [38], and their hardware [39, 40] and software [41, 42] implementations were given. While hardware implementations are compact and fast, they are also inflexible and expensive. The change of the underlying field in a hardware implementation requires a complete redesign. Software implementations, on the other hand, are perhaps slower, but they are cost-effective and flexible, i.e., the algorithms and the field parameters can easily be modified without requiring redesign. Recently, there has been a growing interest to develop software methods for implementing GF(2^k) arithmetic operations for cryptographic and coding applications [42, 43, 44, 45, 10].

We show that the multiplication operation $c = a \cdot b \cdot r^{-1}$ in the field GF(2^k) can be implemented significantly faster in software than the standard multiplication, where r is a special fixed element of the field [19, 46]. This operation is the finite field analogue of the Montgomery multiplication for modular multiplication of integers. We give the bit-level and word-level algorithms for computing the product, perform a thorough performance analysis, and compare the algorithm to the standard multiplication algorithm in GF(2^k).

We also present a new algorithm for computing a^e where $a \in$ GF(2^k) and e is an integer. The proposed algorithm is more suitable for implementation in software, and relies on the Montgomery multiplication in GF(2^k). The speed of the exponentiation algorithm largely depends on the availability of a fast method for multiplying two polynomials of length w defined over GF(2). The theoretical analysis and our experiments indicate that the proposed exponentiation method is about 6 times faster than the exponentiation method using the standard multiplication when $w = 8$. Furthermore, the availability of a 32-bit GF(2) polynomial multiplication instruction on the underlying processor would make the new exponentiation algorithm up to 37 times faster.

3.2. Polynomial Representation

The elements of the field GF(2^k) can be represented in several different ways [47, 48, 49]. We find the polynomial representation useful and suitable for software implementation. The algorithm for the Montgomery multiplication described in this study is based on the polynomial representation. According to this representation an element a of GF(2^k) is a polynomial of length k, i.e., of degree less than or equal to $k-1$, written as

$$a(x) = \sum_{i=0}^{k-1} a_i x^i = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0 \ ,$$

where the coefficients $a_i \in$ GF(2). These coefficients are also referred as the bits of a, and the element a is represented as $a = (a_{k-1}a_{k-2}\cdots a_1 a_0)$. In the word-level description of the algorithms, we partition these bits into blocks of equal length. Let w be the wordsize of the computer. We assume that $k = sw$, and write a as an sw-bit number consisting of s blocks, where each block is of length w. If k is less than sw (and more than $(s-1)w$), then we pad the high-order bits of the most significant block with zero and take k as sw.

Thus, we write a as $a = (A_{s-1}A_{s-2} \dots A_1 A_0)$, where each A_i is of length w such that

$$A_i = (a_{iw+w-1}a_{iw+w-2} \cdots a_{iw+1}a_{iw}) \ .$$

In the polynomial case, this is equivalent to

$$a(x) = \sum_{i=0}^{s-1} A_i(x)x^{iw} = A_{s-1}(x)x^{(s-1)w} + A_{s-2}(x)x^{(s-2)w} + \cdots + A_1(x)x^w + A_0(x) \ ,$$

where $A_i(x)$ is a polynomial of length w such that

$$A_i(x) = \sum_{j=0}^{w-1} a_{iw+j}x^j = a_{iw+w-1}x^{w-1} + a_{iw+w-2}x^{w-2} + \cdots + a_{iw+1}x + a_{iw} \ .$$

The addition of two elements a and b in GF(2^k) are performed by adding the polynomials $a(x)$ and $b(x)$, where the coefficients are added in the field GF(2). This is equivalent to bit-wise XOR operation on the vectors a and b. In order to multiply two elements a and b in GF(2^k), we need an irreducible polynomial of degree k. Let $n(x)$ be an irreducible polynomial of degree k over the field GF(2). The product $c = a \cdot b$ in GF(2^k) is obtained by computing

$$c(x) = a(x)b(x) \bmod n(x) \ ,$$

where $c(x)$ is a polynomial of length k, representing the element $c \in$ GF(2^k). Thus, the multiplication operation in the field GF(2^k) is accomplished by multiplying the corresponding polynomials modulo the irreducible polynomial $n(x)$.

3.3. Montgomery Multiplication in GF(2^k)

Instead of computing $a \cdot b$ in GF(2^k), we propose to compute $a \cdot b \cdot r^{-1}$ in GF(2^k), where r is a special fixed element of GF(2^k). A similar idea was proposed by Montgomery in [8] for modular multiplication of integers. We show that Montgomery's technique is applicable to the field GF(2^k) as well. The selection of $r(x) = x^k$ turns out to be very useful in obtaining fast software implementations. Thus, r is the element of the field, represented by the polynomial $r(x) \bmod n(x)$. The Montgomery multiplication method requires that $r(x)$ and $n(x)$ are relatively prime, i.e., $\gcd(r(x), n(x)) = 1$. For this assumption to hold, it suffices that $n(x)$ be not divisible by x. Since $n(x)$ is an irreducible polynomial over the field GF(2), this will always be case. Since $r(x)$ and $n(x)$ are relatively prime, there exist two polynomials $r^{-1}(x)$ and $n'(x)$ with the property that

$$r(x)r^{-1}(x) + n(x)n'(x) = 1 , \tag{3.1}$$

where $r^{-1}(x)$ is the inverse of $r(x)$ modulo $n(x)$. The polynomials $r^{-1}(x)$ and $n'(x)$ can be computed using the extended Euclidean algorithm [49, 47]. The Montgomery multiplication of a and b is defined as the product

$$c(x) = a(x)b(x)r^{-1}(x) \bmod n(x) , \tag{3.2}$$

which can be computed using the following algorithm:

```
Algorithm for Montgomery Multiplication
Input:    $a(x), b(x), r(x), n'(x)$
Output:   $c(x) = a(x)b(x)r^{-1}(x) \bmod n(x)$
Step 1.   $t(x) := a(x)b(x)$
Step 2.   $u(x) := t(x)n'(x) \bmod r(x)$
Step 3.   $c(x) := [t(x) + u(x)n(x)]/r(x)$
```

In order to prove the correctness of the above algorithm, note that $u(x) = t(x)n'(x) \bmod r(x)$ implies that there is a polynomial $K(x)$ over GF(2) with the property

$$u(x) = t(x)n'(x) + K(x)r(x) . \tag{3.3}$$

We write the expression for $c(x)$ in Step 3, and then substitute $u(x)$ with the expression (3.3) as

$$\begin{aligned} c(x) &= \frac{1}{r(x)} \, [t(x) + u(x)n(x)] \\ &= \frac{1}{r(x)} \, [t(x) + t(x)n'(x)n(x) + K(x)r(x)n(x)] \end{aligned}$$

Furthermore, we have $n'(x)n(x) = 1 + r(x)r^{-1}(x)$ according to Equation (3.1). Thus, $c(x)$ is computed as follows:

$$\begin{aligned}
c(x) &= \frac{1}{r(x)}\,[t(x) + t(x)[1 + r(x)r^{-1}(x)] + K(x)r(x)n(x)] \\
&= \frac{1}{r(x)}\,[t(x)r(x)r^{-1}(x) + K(x)r(x)n(x)] \\
&= t(x)r^{-1}(x) + K(x)n(x) \\
&= t(x)r^{-1}(x) \bmod n(x) \\
&= a(x)b(x)r^{-1}(x) \bmod n(x)\ ,
\end{aligned}$$

as required. The above algorithm is similar to the algorithm given for the Montgomery multiplication of integers. The only difference is that the final subtraction step required in the integer case is not necessary in the polynomial case. This is proved by showing that the degree of the polynomial $c(x)$ computed by this algorithm is less than or equal to $k-1$. Since the degrees of $a(x)$ and $b(x)$ are both less than or equal to $k-1$, the degree of $t(x) = a(x)b(x)$ will be less than or equal to $2(k-1)$. Also note that the degrees of $n(x)$ and $r(x)$ are both equal to k. The degree of $u(x)$ computed in Step 2 will be strictly less than k since the operation is performed modulo $r(x)$. Thus, the degree of $c(x)$ as computed in Step 3 of the algorithm is found as

$$\begin{aligned}
\deg\{c(x)\} &\le \max[\deg\{t(x)\}\ ,\ \deg\{u(x)\} + \deg\{n(x)\}] - \deg\{r(x)\} \\
&\le \max[2k-2\ ,\ k-1+k] - k \\
&\le k-1
\end{aligned}$$

Thus, the polynomial $c(x)$ is already reduced.

3.4. Computation of Montgomery Multiplication

The computation of $c(x)$ involves standard multiplications, a modulo $r(x)$ multiplication, and a division by $r(x)$. The modular multiplication and division operations in

Steps 2 and 3 are intrinsically fast operations since $r(x) = x^k$. The remainder operation in modular multiplication using the modulus x^k is accomplished by simply ignoring the terms which have powers of x larger than or equal to k. Similarly, division of an arbitrary polynomial by x^k is accomplished by shifting the polynomial to the right by k places. A drawback in computing $c(x)$ is the precomputation of $n'(x)$ required in Step 2. However, it turns out the computation of $n'(x)$ can be completely avoided if the coefficients of $a(x)$ are scanned one bit at a time. Furthermore, the word-level algorithm requires the computation of only the least significant word $N_0'(x)$ instead of the whole $n'(x)$. In order to explain this, we note that the Montgomery product can be written as

$$c(x) = x^{-k}a(x)b(x) = x^{-k} \sum_{i=0}^{k-1} a_i x^i b(x) \pmod{n(x)}.$$

The product

$$t(x) = (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0)b(x)$$

can be computed by starting from the most significant digit, and then proceeding to the least significant as

```
$t(x) := 0$
for $i = k-1$ to 0
   $t(x) := t(x) + a_i b(x)$
   $t(x) := x t(x)$
```

The shift factor x^{-k} in $x^{-k}a(x)b(x)$ reverses the direction of summation. Since

$$x^{-k}(a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0) =$$
$$a_{k-1}x^{-1} + a_{k-2}x^{-2} + \cdots + a_1 x^{-k+1} + a_0 x^{-k} \ ,$$

we start processing the coefficients of $a(x)$ from the least significant, and obtain the following bit-level algorithm in order to compute $t(x) = a(x)b(x)x^{-k}$.

```
$t(x) := 0$
```

```
for $i = 0$ to $k - 1$
    $t(x) := t(x) + a_i b(x)$
    $t(x) := t(x)/x$
```

This algorithm computes the product $t(x) = x^{-k}a(x)b(x)$, however, we are interested in computing $c(x) = x^{-k}a(x)b(x) \bmod n(x)$. Following the analogy to the integer algorithm, we achieve this computation by adding $n(x)$ to $c(x)$ if c_0 is 1, making the new $c(x)$ divisible by x since $n_0 = 1$. If c_0 is already 0 after the addition step, we do not add $n(x)$ to it. Therefore, we are computing $c(x) := c(x) + c_0 n(x)$ after the addition step. After this computation, $c(x)$ will always be divisible by x. We can compute $c(x) := c(x)x^{-1} \bmod n(x)$ by dividing $c(x)$ by x since $c(x) = xu(x)$ implies $c(x) = xu(x)x^{-1} = u(x) \bmod n(x)$. The bit-level algorithm is given below:

```
Bit-Level Algorithm for Montgomery Multiplication
Step 1.    $c(x) := 0$
Step 2.    for $i = 0$ to $k - 1$ do
Step 3.        $c(x) := c(x) + a_i b(x)$
Step 4.        $c(x) := c(x) + c_0 n(x)$
Step 5.        $c(x) := c(x)/x$
```

The bit-level algorithm for the Montgomery multiplication given above is generalized to the word-level algorithm by proceeding word by word where the wordsize is $w \geq 2$ and $k = sw$. Recall that $A_i(x)$ represents the ith word of the polynomial $a(x)$. The addition step is performed by multiplying $A_i(x)$ by $b(x)$ at steps $i = 0 \ldots (s-1)$. We then need to multiply the partial product $c(x)$ by x^{-w} modulo $n(x)$. In this step, we add a multiple of $n(x)$ to $c(x)$ so that the least significant w coefficients of $c(x)$ will be zero, i.e., $c(x)$ will be divisible by x^w. Thus, if $c(x) \neq 0 \bmod x^w$, then we find $M(x)$ (which is a polynomial of length w) such that $c(x) + M(x)n(x) = 0 \pmod{x^w}$. Let $C_0(x)$ and $N_0(x)$ be the least significant words of $c(x)$ and $n(x)$, respectively. We calculate $M(x)$ as

$$M(x) = C_0(x)N_0^{-1}(x) \bmod x^w \ .$$

We note that $N_0^{-1}(x) \bmod x^w$ is equal to $N_0'(x)$ since the property (3.1) implies that

$$\begin{aligned} x^{sw}x^{-sw} + n(x)n'(x) &= 1 \pmod{x^w} \\ N_0(x)N_0'(x) &= 1 \pmod{x^w} \end{aligned}$$

The word-level algorithm for the Montgomery multiplication is obtained as

```
Word-Level Algorithm for Montgomery Multiplication
Step 1.  $c(x) := 0$
Step 2.  for $i = 0$ to $s - 1$ do
Step 3.      $c(x) := c(x) + A_i(x)b(x)$
Step 4.      $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 5.      $c(x) := c(x) + M(x)n(x)$
Step 6.      $c(x) := c(x)/x^w$
```

3.5. Montgomery Squaring

The computation of the Montgomery squaring can be optimized due to the fact that cross terms disappear because they come in pairs and the ground field is GF(2). It is easy to show that

$$\begin{aligned} c(x) &= a^2(x) \\ &= a_{k-1}x^{2(k-1)} + a_{k-2}x^{2(k-2)} + \cdots + a_1x^2 + a_0 \\ &= (a_{k-1}\mathbf{0}a_{k-2}\mathbf{0}\cdots\mathbf{0}a_1\mathbf{0}a_0) \ , \end{aligned}$$

and thus, the multiplication steps can be skipped. The Montgomery squaring algorithm starts with the degree $2(k-1)$ polynomial $c(x) = a^2(x)$, and then reduces $c(x)$ by computing $c(x) := c(x)x^{-k} \bmod n(x)$. The steps of the word-level algorithm are given below:

Word-Level Algorithm for Montgomery Squaring

```
Step 1.  $c(x) := \sum_{i=0}^{k-1} a_i x^{2i}$
Step 2.  for $i = 0$ to $s - 1$ do
Step 3.      $M(x) := C_0(x)N_0'(x) \pmod{x^w}$
Step 4.      $c(x) := c(x) + M(x)n(x)$
Step 5.      $c(x) := c(x)/x^w$
```

3.6. Computation of the Inverse

The word-level algorithm requires the computation of the w-length polynomial $N_0'(x)$ instead of the entire polynomial $n'(x)$ which is of length $k = sw$. It turns out that the algorithm introduced in [18] for computing n_0' in the integer case can also be generalized to the polynomial case. The inversion algorithm is based on the observation that the polynomial $N_0(x)$ and its inverse satisfy

$$N_0(x)N_0^{-1}(x) = 1 \pmod{x^i} \tag{3.4}$$

for $i = 1, 2, \ldots, w$. In order to compute $N_0'(x)$, we start with $N_0'(x) = 1$, and proceed as

Algorithm for Inversion

```
Step 1.  $N_0'(x) := 1$
Step 2.  for $i = 2$ to $w$
Step 3.      $t(x) := N_0(x)N_0'(x) \bmod x^i$
Step 4.      if $t(x) \neq 1$ then $N_0'(x) := N_0'(x) + x^{i-1}$
```

3.7. Montgomery Exponentiation in GF(2^k)

The proposed exponentiation algorithm is based on the Montgomery multiplication and squaring operations [50, 51]. Let e be an m-bit integer, where $e_i \in \{0, 1\}$ is the ith bit of e for $i = 0, 1, \ldots, m-1$. In order to compute $c = a^e$ for a given $a \in \mathrm{GF}(2^k)$, we first compute the Montgomery images of 1 and a using standard multiplications. The exponentiation algorithm based on the binary method then proceeds to compute c using only the Montgomery squaring and multiplication operations.

```
Algorithm for Montgomery Exponentiation
Step 1.   $\bar{c} := 1 \cdot r$
Step 2.   $\bar{a} := a \cdot r$
Step 3.   for $i = m-1$ downto 0 do
Step 4.       $\bar{c} := \bar{c} \times \bar{c}$
Step 5.       if $e_i = 1$ then $\bar{c} := \bar{c} \times \bar{a}$
Step 6.   $c := \bar{c} \times 1$
```

The difference of the above algorithm from the binary method using the standard squaring and multiplication operations is that in Steps 4 and 5, respectively, we perform the Montgomery squaring and multiplication operations. Initially, we have $\bar{c} = 1 \cdot r$. When a Montgomery squaring or a Montgomery multiplication is performed, the multiplicative factor r remains in place since

$$\bar{c} \times \bar{c} = (c \cdot r) \cdot (c \cdot r) \cdot r^{-1} = (c \cdot c) \cdot r \ , \tag{3.5}$$

$$\bar{c} \times \bar{a} = (c \cdot r) \cdot (a \cdot r) \cdot r^{-1} = (c \cdot a) \cdot r \ . \tag{3.6}$$

We remove this multiplicative factor on $\bar{c}$ in Step 6 by performing a Montgomery multiplication, and obtain

$$\bar{c} \times 1 = (c \cdot r) \cdot 1 \cdot r^{-1} = c \ . \tag{3.7}$$

In order to perform the Montgomery squaring and multiplication operations, we use the algorithm introduced in [19]. This method is based on the polynomial representation of the elements of $\mathrm{GF}(2^k)$, and is particularly suitable for software implementation due to the fact that it proceeds in a word-level fashion.

3.8. Analysis

In this section, we give a rigorous analysis of the Montgomery exponentiation algorithm in GF(2^k) by calculating the number of word-level operations. The word-level GF(2) polynomial addition is simply the bitwise XOR operation which is a readily available instruction on most general purpose microprocessors and signal processors. The word-level GF(2) polynomial multiplication operation receives two 1-word (w-bit) polynomials $a(x)$ and $b(x)$ defined over the field GF(2), and computes the 2-word ($2w$-bit) polynomial $c(x) = a(x)b(x)$. For example, given $a = (1101)$ and $b = (1010)$, this operation computes c as

$$\begin{aligned} a(x)b(x) &= (x^3 + x^2 + 1)(x^3 + x) \\ &= x^6 + x^5 + x^4 + x \\ &= (0111\ 0010)\ . \end{aligned}$$

The implementation of this operation, which we call MULGF2, can be performed in 3 distinctly different ways:

- An instruction on the underlying processor.
- Table lookup approach.
- Emulation using SHIFT and XOR operations.

In the first approach, the underlying processor has a special MULGF2 instruction as defined above. The availability of an instruction to perform this operation would definitely be the fastest method. However, none of the general purpose processors contains an instruction to perform this operation.

A simple method for implementing the table lookup approach is to use two tables, one for computing the higher (H) and the other for computing the lower (L) bits of the product. The tables are addressed using the bits of the operands, and thus, each table is of size $2^w \times 2^w \times w$ bits. We obtain the values H and L in two table reads. However, we note that these tables are different from the tables in [41, 43], which are used to implement GF(2^w) multiplications. Here we are using the tables to multiply two $(w-1)$-degree polynomials over GF(2) to obtain the polynomial of degree $2(w-1)$.

In the emulation approach, two w-bit polynomials A and B are multiplied using shift, rotate, and xor operations. The 2-word product is accumulated in two words H and L as follows:

```
H := 0
L := 0
for j=w-1 downto 0 do
    L := SHL(L,1)
    H := RCL(H,1)
    if BIT(B,j)=1 then L := L XOR A
```

Here SHL shifts its first operand to the left by the number of bits given in the second operand. RCL is a rotate (circular shift) instruction shifting the first operand to the left circularly by the number of bits given in the second operand.

This algorithm computes the 2-word result using a total of $2w$ SHIFT/ROTATE and w XOR operations. The emulation approach is usually slower than the table lookup approach, particularly for $w \geq 8$.

Our analysis is a first-order approximation. We do not consider certain processor features such as specialized bit-level instructions (test jth bit), conditional executions (delay slots in conditional branches), and conditional data movement instructions. We also do not count loop overhead, pointer arithmetic, and the like, which undoubtly affects the performance.

In order to compare the exponentiation algorithms using the standard and the Montgomery multiplications, we count the number of word-level operations required by these algorithms [51]. We perform this analysis by fixing the exponentiation method as the binary method, and taking m as the number of bits in the exponent e. The standard and Montgomery exponentiation algorithms are given in Figure 3.1 below.

FIGURE 3.1: Exponentiation Algorithms.

```
Montgomery Exponentiation
    Step 1.  $\bar{c} := 1 \cdot r$
    Step 2.  $\bar{a} := a \cdot r$
    Step 3.  for $i = m - 1$ downto 0 do
    Step 4.      $\bar{c} := \bar{c} \cdot \bar{c} \cdot r^{-1}$
    Step 5.      if $e_i = 1$ then $\bar{c} := \bar{c} \cdot \bar{a} \cdot r^{-1}$
    Step 6.  $c = \bar{c} \cdot 1 \cdot r^{-1}$

Standard Exponentiation
    Step 1.  $c := 1$
    Step 2.  for $i = m - 1$ downto 0 do
    Step 3.      $c := c \cdot c$
    Step 4.      if $e_i = 1$ then $c := c \cdot a$
```

The Montgomery exponentiation algorithm relies on the subroutines for computing the inverse, the Montgomery squaring and multiplications (in Steps 4 and 5), and a single standard multiplication (in Step 2). We do not need to perform a multiplication in Step 1. The standard exponentiation algorithm, on the other hand, requires only the standard squaring and multiplication subroutines.

The detailed analyses of the word-level Montgomery and standard multiplication algorithms are given in [19]. Similar analyses can also be given for the word-level Montgomery and standard squaring algorithms. The number of word-level operations required by these algorithms are summarized in Table 3.1.

TABLE 3.1: Operation counts for the multiplication and squaring algorithms.

Subroutine	MULGF2	XOR/AND/OR	SHIFT
Montgomery Multiplication	$2s^2 + s$	$4s^2$	-
Montgomery Squaring	$s^2 + s$	$2s^2 + (2w + 1)s$	$(2w + 1)s$
Standard Multiplication	s^2	$3(\frac{w}{2} + 1)s^2 + \frac{w}{2}s$	$2(w + 1)s^2 + (w + 1)s$
Standard Squaring	-	$\frac{9w}{4}s^2 + (2w + \frac{3}{2})s$	$3ws^2 + (3w + 1)s$

On the other hand, the inversion algorithm given in Section 5 requires $(w - 1)$ MULGF2, $(w - 1)$ AND, and $(w - 1)$ SHIFT operations in Step 3. Assuming the least significant coefficient of $t(x)$ is equal 0 with probability 1/2, we obtain the number of XOR and SHIFT operations in Step 4 as $(w - 1)/2$ and $(w - 1)/2$, respectively. Using these values and Table 3.1, we summarize the operation counts of the exponentiation algorithms in Table 3.2.

In Table 3.3, we summarize the total number of operations required by the Montgomery and standard exponentiation algorithms for $w = 8, 16, 32$. Table 3.4 tabulates the maximum speedup of the proposed exponentiation method assuming the word-level operations XOR/AND/OR and SHIFT take nearly the same amount of time. The emulation cost of MULGF2 is $2w$ SHIFT and w XOR operations in the emulation case. The cost of MULGF2

TABLE 3.2: Operation counts for the Montgomery and the standard exponentiation.

Montgomery Exponentiation			
	MULGF2	XOR/AND/OR	SHIFT
Inversion	$w-1$	$\frac{3}{2}(w-1)$	$\frac{3}{2}(w-1)$
Step 2	s^2	$3(\frac{w}{2}+1)s^2+\frac{w}{2}s$	$2(w+1)s^2+(w+1)s$
Step 4	ms^2+ms	$2ms^2+(2w+1)ms$	$(2w+1)ms$
Step 5	$ms^2+\frac{m}{2}s$	$2ms^2$	-
Step 6	$2s^2+s$	$4s^2$	-
Standard Exponentiation			
Step 3	-	$\frac{9w}{4}ms^2+(2w+\frac{3}{2})ms$	$3wms^2+(3w+1)ms$
Step 4	$\frac{m}{2}s^2$	$(\frac{3w}{4}+\frac{3}{2}ms^2+\frac{w}{4}ms$	$(w+1)ms^2+(\frac{w}{2}+\frac{1}{2})ms$

instruction is assumed equal to those of SHIFT and XOR operations in the instruction case.

3.9. Implementation Results and Conclusions

We implemented the Montgomery and standard exponentiation algorithms in C, and obtained timings on a 100-MHz Intel 486DX4 processor running the NextStep 3.3 operating system. We executed the exponentiation programs several hundred times and obtained the average timings for each k. The modulus polynomial $n(x)$ is generated randomly for k = 64, 128, 256, 512, 1024, 1536, 2048. The exponent is an m-bit integer with equal number of 0 and 1 bits.

The multiplication operation MULGF2 was implemented using three approaches. In

TABLE 3.3: Comparing the Montgomery and standard exponentiation algorithms for the number of MULGF2 operations.

MULGF2	w	Standard	Montgomery
Emulation	8	$70.5ms^2 + 49ms$	$(52m + 109)s^2 + (70m + 37)s + 189$
"	16	$138.5ms^2 + 95ms$	$(100m + 209)s^2 + (138m + 73)s + 765$
"	32	$274.5ms^2 + 187ms$	$(196m + 409)s^2 + (274m + 145)s + 3069$
Instruction	8	$59ms^2 + 49ms$	$(6m + 40)s^2 + (35.5m + 14)s + 28$
"	16	$115ms^2 + 95ms$	$(6m + 68)s^2 + (67.5m + 26)s + 60$
"	32	$227ms^2 + 187ms$	$(6m + 124)s^2 + (131.5m + 50)s + 124$

the first approach, we used the emulation algorithm given in the previous section.

In the second approach, a lookup table is used for $w = 8$, as described before. For $w = 8$, each of the tables is of size 64 Kilobytes, which is reasonable. However, for $w = 16$, the table size increases to $2^{16} \times 2^{16} \times 16$ bits, which gives 8 Gigabytes. Therefore, we implemented the table lookup MULGF2 operation only for $w = 8$.

For $w = 16$ and $w = 32$, we implement the MULGF2 operation using a *hybrid ap-*

TABLE 3.4: Montgomery exponentiation speedup for different MULGF2 implementations.

MULGF2→	Emulation			Instruction		
w	8	16	32	8	16	32
Speedup	1.36	1.39	1.40	9.83	19.17	37.83

proach: 8-bit tables coupled with emulation to obtain the 16-bit or 32-bit result. For example, 16-bit multiplication using two 8-bit tables is computed as shown below.

```
a1 := SHR(a,8)
a0 := a AND 0xff
b1 := SHR(b,8)
b0 := b AND 0xff
L := TableL[a0][b0] XOR SHR(TableH[a0][b0]
     XOR TableL[a1][b0] XOR TableL[a0][b1],8)
H := TableH[a1][b0] XOR TableH[a0][b1]
     XOR TableL[a1][b1] XOR SHR(TableH[a1][b1],8)
```

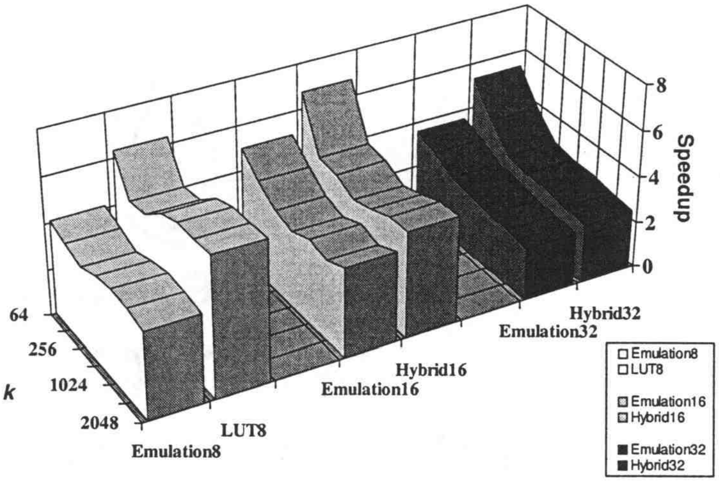
where TableL and TableH are the 8-bit tables giving the low and high order 8-bits of an 8-by-8 bit GF(2) polynomial multiplication. The 32-bit hybrid multiplication algorithm also uses these 8-bit tables.

TABLE 3.5: Experimental speedup values of Montgomery exponentiation for $m = 128$.

$w \rightarrow$	8		16		32	
k	Tab8	Emu	Hyb8	Emu	Hyb8	Emu
64	6.32	4.10	6.75	5.00	5.33	3.61
128	4.85	3.79	4.51	4.20	4.00	3.25
256	4.95	3.49	4.40	3.60	3.03	2.79
512	5.66	3.83	3.96	3.35	2.88	2.66
1024	5.97	4.04	4.22	3.70	2.83	2.44
1536	6.00	3.95	4.58	3.51	2.69	2.44
2048	6.05	3.76	4.62	3.89	2.56	2.30

The experimental speedup values are given in Table 3.5. These speedup values are obtained by dividing the time elapsed for standard exponentiation by the time elapsed for Montgomery exponentiation. Montgomery exponentiation time includes computation of $N_0'(x)$, precomputation of $\bar{a}$ and $\bar{c}$, and final computation by 1 to obtain c.

FIGURE 3.2: Comparative illustration of Montgomery exponentiation speedup.



3.10. Conclusions

As the theoretical results summarized in Tables 3.3 and 3.4 the experimental data in Table 3.5 indicate, the Montgomery exponentiation algorithm is about 6 times faster than the standard exponentiation for $w = 8$. The table lookup approach for $w \geq 16$ seems unrealistic due to the size of the tables. An efficient way to implement the `MULGF2` operation is to add an instruction to the processor to perform this multiplication. The availability of such an instruction would yield more speedup than the table lookup approach, because memory accesses would be eliminated which are required in the table lookup approach. For example, the availability of a 32-bit MULGF2 instruction would make the Montgomery exponentiation about 37 times faster than the standard exponentiation, as seen in Table 3.4. Table 3.2 illustrates a comparative graph of the speedup values for Montgomery exponentiation for aforementioned methods.

The crucial part of the proposed exponentiation algorithm is the Montgomery multiplication in GF(2^k) introduced in [19]. The computation of the Montgomery multiplication in GF(2^k) is similar to the one for modular arithmetic. A review of the Montgomery multiplication algorithms for modular arithmetic is given in [20]. We are currently analyzing these algorithms, and comparing their time and space requirements for performing the Montgomery multiplication operation in GF(2^k). Another possible avenue of research is to compare the proposed exponentiation method to the one which uses trinomials and the normal basis. The squaring operation in the normal basis is trivial, however, the software implementation of the multiplication is more complicated.

4. A METHODOLOGY FOR HIGH-SPEED SOFTWARE IMPLEMENTATIONS OF NUMBER-THEORETIC CRYPTOSYSTEMS

4.1. Introduction

Because of their flexibility and cost effectiveness, software implementations of number-theoretic cryptographic algorithms (e.g., RSA and Diffie-Hellman) are often desired. In order to obtain the required level of performance (speed) on a selected platform, the developers turn to algorithm-level optimizations and assembly language programming. In this chapter, we examine these implementation issues and propose a design methodology for obtaining high-speed implementations. We show that between the full assembler implementation and the standard C implementation, there is a design option in which only a small number of code segments (kernel operations) are written in assembler in order to obtain a significant portion of the speed increase gained by the full assembler implementation. We propose a small set of kernel operations which are as simple as $a \cdot b + c$, where the numbers a, b, c are 1-word integers. The results of our experiments on several processors are also summarized.

The privacy and authenticity of information (whether it is stored on a single computer or shared on a network of computers) requires implementation of cryptographic functions [52]. The basic functions of information security services are very few, and almost invariant. These include public-key cryptosystems, digital signatures, message digest functions, and secret-key cryptographic algorithms [53, 54]. The design and evaluation of these cryptographic functions is a special topic on its own, requiring advanced knowledge of combinatorial mathematics, number theory, abstract algebra, and theoretical computer science [55]. There is also the subject of *algorithm engineering*, which refers to high-speed and cost effective hardware and software implementation of cryptographic algorithms [56].

Most public-key cryptographic functions require operations with elements of a large finite group, and need to be optimized on the chosen platform for high-speed implementation. As an example, the RSA cryptosystem [2] uses modular arithmetic operations (addition, multiplication, and exponentiation) with large integers, usually in the range of 512 to 1024 bits. Arithmetic with such large integers is time consuming, considering the fact that currently available processor have arithmetic logic units with wordsize up to 32 bits. The current fast implementations of the RSA signature algorithm with a 512-bit key size require on the order of 50 ms on a signal processor using advanced algorithmic techniques and assembly language programming [18]. However, the security requirements are already pushing the key size to 1024 bits, at which a signature operation takes nearly half a second. This is not an acceptable speed for most networking applications. Other cryptographic algorithms, for example, the Diffie-Hellman key exchange method [3], the ElGamal public-key cryptosystem and digital signature algorithm [57], the Digital Signature Standard [58] also require implementation of modular arithmetic operations with large integers.

Software implementations of the number-theoretic cryptographic algorithms are often desired because of their flexibility and cost effectiveness [1]. Furthermore, certain applications are suitable for software implementations because of their very nature. However, the performance is always an issue, requiring the designer to optimize these cryptographic algorithms on the selected processor. In order to exploit the architectural and arithmetic-logic properties of the processor, the designer needs to analyze and reformulate the underlying algorithms. Almost inevitably, the programming is performed in assembly language in order to take advantage of the specific architectural properties of the processor, and thus, to obtain the desired performance [18, 6, 17, 20, 59].

In this chapter, we examine these implementation issues in order to determine the actual contribution of assembly level programming to the speed of the cryptographic algorithms. We show that between the full assembler implementation and the standard C implementation, there is a design option in which only a small number of code segments are written in assembler in order to obtain a significant portion of the speed increase gained by the full assembler implementation [60]. These small code segments are the *kernel* of arithmetic operations for number-theoretic cryptographic algorithms, and have been obtained by analyzing several different implementations of these number-theoretic cryptosystems. We propose a small set (only 8) of such code segments, implementing certain arithmetic operations which are as simple as the computation of $a \cdot b + c$, where the numbers a, b, c are 1-word integers.

Our experimental results on the Pentium PC show that by developing efficient assembly language implementation of these 8 operations as 'in-line' assembly code segments in the RSA cryptosystem, we can obtain a speedup of 2.33 over the standard C implementation. This speedup is about 64 % of the speedup obtained by a full assembler implementation.

4.2. Implementation Methods

The usefulness of a C implementation is due to its portability, i.e., the fact that the program can easily be compiled and executed on another machine. However, the C program may not execute as fast as an assembler program accomplishing the same computation since specific architectural properties of the new machine are not taken into account. On the other hand, efficient assembler software development requires full understanding of the sophisticated microprocessor architecture. The assembly language programmer needs to know the properties of the assembler instructions, the operation of multiple functioning units, the rules of instruction issuing, pipeline structure, and alignment rules, and also certain specific information about the cache and the memory structure. The development of assembler programs is a tedious, lengthy, and expensive task. It can be argued that a smart compiler will be aware of the detailed architectural issues, and thus, can produce more efficient code than a straightforward assembler implementation in many instances [61]. However, the developers of cryptographic systems often have to turn to assembly language programming in order to obtain the required speed. This way the programmer gets the chance to reformulate the algorithm to be implemented by taking into account the architectural properties of the processor.

In this study, we consider the design options between the standard C and the assembly language for implementing the number-theoretic cryptographic algorithms. The properties of these two extreme design options are:

- Standard C: Portable, inexpensive, short development time, slow execution.
- Assembler: Not portable, expensive, long development time, fast execution.

There are several design options between these two ends. A particular design option involves the use of *non-standard* C data types such as `int64` or `long long`. We name this approach *C with Extended Types*. It turns out some amount of speed increase can be gained using such data types for number-theoretic cryptography. We are however limited to those platforms which support these data types and their particular definitions and uses. We gain a certain amount of speed by renouncing a small amount of portability.

As soon as the use of assembly language programming enters the picture, we loose portability. Once the portability is no longer an issue, the development cost of assembly language programming needs to be taken into account. One approach is the development of the entire code or the most crucial subroutines (e.g., the Montgomery multiplication and squaring) in the assembly language. This involves a great amount of assembly language programming, and we argue that it is not necessary in many instances. We propose a design approach in which only a specific kernel of operations need to be developed in assembler. In this chapter, we evaluate and compare the following four approaches in terms of their resulting performance.

- Standard C code
- C with extended types

- Complete assembler
- C with kernel in assembler

4.2.1. Standard C Code

In the C language, operands of an arithmetic expression are converted to a common type before the computation [62, 63], which is referred to as *converted type.* The value of a variable may be *truncated* to a less significant type, or it can be *promoted* to a more significant type. The high order bits are ignored in case of truncation. The promotion is performed using zero padding or sign extension. The result of an operation is also of the converted type. The truncation inhibits availability of high order bits of certain arithmetic operations in C, enforcing an emulation approach for precise calculations. For an addition of n operands using w-bit scalar type, maximum value of result is $n(2^w - 1)$. Assuming $n \leq 2^w$, the exact result can be stored in two w-bit words. Exact addition of such variables can be accomplished by computing lower and higher bits separately. The code segment given below adds two w-bit words to obtain the $(w + 1)$-bit result. Low w bits are stored in S, and high order 1-bit (the carry) is stored in C. Multi-operand addition can be carried out similarly.

```
#define WSIZE      (8*sizeof(word))
#define MSBMASK    ((word)1 << (WSIZE-1))
  S=(a & ~MSBMASK) + (b & ~MSBMASK);
  C=(a >> (WSIZE-1)) + (b >> (WSIZE-1)) + (S >> (WSIZE-1));
  S = a + b;
  C >>= 1;
```

A multiplication expression in C language stores only the low order word of the two-word product. Let the multiplication be $c := a \cdot b$ where a and b are word type variables. The type of the result is also word. Thus, the actual product is truncated if $a \cdot b \geq 2^{8*sizeof(\texttt{word})}$. In order to compute the complete product, the w-bit input operands are split into two $w/2$-bit numbers. The following C code segment can be used to obtain the full result of c in the word pair (C,S).

```
#define WSIZE          (8*sizeof(word))
#define LOWBITS(x)     ((x) & (~((word)0) >> (WSIZE/2)))
#define HIGHBITS(x)    ((x) >> (WSIZE/2))
  albl = LOWBITS (a) * LOWBITS (b);
  ahbl = HIGHBITS(a) * LOWBITS (b);
  albh = LOWBITS (a) * HIGHBITS(b);
  ahbh = HIGHBITS(a) * HIGHBITS(b);
  sum = LOWBITS(albh) + LOWBITS(ahbl) + HIGHBITS(albl);
  S = (sum << (WSIZE/2)) + LOWBITS(albl);
  C = ahbh + HIGHBITS(albh) + HIGHBITS(ahbl) + HIGHBITS(sum);
```

4.2.2. C with Extended Types

This approach relies on a non-standard type of the C programming language. The code is still portable, maintainable, and testable, however, it is restricted to the platform on which the the non-standard language extensions are supported. This method depends on the fact that a variable of twice the size of a general purpose register contains all result bits for the operation. Let the name of this extended type be dword. We can then implement the addition and multiplication of two words as follows:

Addition

```
#define WSIZE     (8*sizeof(word))
  CS = (dword)a + (dword)b;
  S = (word)CS;
  C = (word)(CS >> WSIZE);
```

Multiplication

```
#define WSIZE     (8*sizeof(word))
  CS = (dword)a * (dword)b;
  S = (word)CS;
  C = (word)(CS >> WSIZE);
```

Here C and S are word type variables, and CS is a dword type variable. The extended type can be used for other operations, e.g., shift and division. The C compiler must convert the right shifting by *wordsize* bits to a single register access to achieve better peformance.

This approach does not require the assembler level implementation. However, the compiler must support the extended type, and also it should be capable of generating efficient code for C blocks involving the extended type. Currently, most C compilers support double register-size variables. For example, Microsoft Visual C++ has "`__int64`" while most of UNIX C compilers have "`long long`" type for variables of twice the register-size of the platform processor.

4.2.3. Complete Assembler

An efficient assembler implementation of a cryptographic algorithm requires a detailed study of the architecture of the underlying processor. Issues related to the instruction set architecture, register space, multiple functional units, and memory hierarchy need to be well understood. An assembler implementation produces smaller and faster code by sacrificing portability. However, assembler implementations are preferred if development costs are relatively less than final benefits.

The Appendix gives assembler programming example codes in Intel Pentium and Sparc V9 assembly languages for performing several different operations with 32-bit numbers. For example, the operations `ADD(C,S,a,b)` and `MUL(C,S,a,b)` respectively denote $(C, S) := a + b$ and $(C, S) := a \cdot b$, where C, S, a, and b are 32-bit unsigned integers.

4.2.4. C with Kernel in Assembler

The speedup obtained with the extended types is not as high as possible due to inefficient utilization of the processor architecture. The performance is limited by the optimization capabilities of the C compiler. We propose an alternative hybrid approach which benefits from flexibility of C and high performance of assembly languages. We minimize the development cost of assembly language programming by proposing a small set of arithmetic operations which need to be coded in the assembler. The remainder of the code is produced in the standard C. The proposed set of kernel operations is given on Table 4.1.

TABLE 4.1: Kernel operations.

Operation	Description
`ADD(C,S,a,b)`	$(C,S) := a+b$
`ADD2(C,S,a,b,c)`	$(C,S) := a+b+c$
`MUL(C,S,a,b)`	$(C,S) := a \cdot b$
`MULADD(C,S,a,b,c)`	$(C,S) := a \cdot b + c$
`MULADD2(C,S,a,b,c,d)`	$(C,S) := a \cdot b + c + d$
`MUL2ADD2(CC,C,S,a,b,c,d)`	$(CC,C,S) := 2 \cdot a \cdot b + c$
`SQU(C,S,a)`	$(C,S) := a^2$
`SQUADD(C,S,a,b)`	$(C,S) := a^2 + b$

The operations in Table 4.1 need to be coded in the assembly language as macros or in-line assembly code segments. They can be written as functions, but this creates considerable overhead. The best situation will be the one in which these operations are supported by the hardware either as instructions or macro instructions.

The amount of assembly language is indeed minimal: each one of these operations can be coded using about 4 to 8 lines of assembler instructions. Therefore, the entire set requires about 60 lines of assembly code. The resulting standard C plus assembler code, if carefully constructed, can be ported on another machine quite easily: only the assembly code segments need to be developed for the new machine, replacing the existing segments.

4.2.5. Determination of Kernel

The arithmetic operations in the kernel are obtained by analyzing the algorithms and the actual implementations of number-theoretic cryptosystems. The proposed kernel is quite minimal in the sense adding other similar operations does not provide any

FIGURE 4.1: Addition of two multi-precision integers: $s = a + b$ where $a = (a_3a_2a_1a_0)$ and $b = (b_3b_2b_1b_0)$.

$$\begin{array}{cccccc} & & \boxed{a_3} & \boxed{a_2} & \boxed{a_1} & \boxed{a_0} \\ + & & \boxed{b_3} & \boxed{b_2} & \boxed{b_1} & \boxed{b_0} \\ \hline & \boxed{s_4} & \boxed{s_3} & \boxed{s_2} & \boxed{s_1} & \boxed{s_0} \end{array}$$

$$\begin{aligned} (c_0, s_0) &= a_0 + b_0 \\ (c_1, s_1) &= a_1 + b_1 + c_0 \\ (c_2, s_2) &= a_2 + b_2 + c_1 \\ (s_4, s_3) &= a_3 + b_3 + c_2 \end{aligned}$$

more considerable speedup gain. Since our objective is to perform as little assembly programming as possible, we carefully selected these operations among many candidates. These experiments were run on the Intel 486 DX4, Intel Pentium, and Sun UltraSparc-II V8+ machines by examining the algorithms and codes for the RSA and Diffie-Hellman algorithms. The implementation results are summarized in the next section.

Determination of a minimal cryptographic kernel plays a key role in the efficiency introduced by the primitive operations. Our focus has been on describing the simple primitives corresponding to a sequence of lengthy C statements observed in pseudocodes of the aforementioned algorithms, e.g., RSA and Diffie-Hellman. These algorithms require modular arithmetic using large integers. Multi-precision modular exponentiation composed of modular multiplications is accomplished by the Montgomery's multiplication method [5, 8, 20]. Multi-precision addition is used in the RSA private key operation based on Chinese Remainder Theorem [64, 32]. Addition and multiplication of two quad-precision unsigned integers are illustrated in Figures 4.1 and 4.2. Pseudocodes for these operations utilizing the kernel primitives are given in Figure 4.3. In the pseudocodes, variables a, b and c are arrays of type `word`, k is the number of words to process. `C` and `S` are `word` variables.

In Figure 4.1, each word of the 5-word sum $(s_4, s_3, s_2, s_1, s_0)$ is formed by the ADD2 kernel operation. In Figure 4.2, $x_{i,j}$ denotes the jth word of the partial product $a \cdot b_i$. The multiplication of two quad-precision w-bit unsigned integers is accomplished by accumulating the partial products in the 8-word array s. The accumulation requires addition of the previous product word s_i to the partial product word $x_{i,j}$. Words of these partial products are computed by $(c_{i,j}, x_{i,j}) = a_j \cdot b_i + c_{i,j-1}$, where $c_{i,-1} = 0$ and $i, j = 0 \ldots k-1$. Addition of the carry-word $c_{i,j-1}$ is implicitly present in $\sum_{j=0}^{k-1} a_j \cdot b_i$. This operation can be accomplished using the MULADD primitive. We used MULADD2 kernel primitive to compute $(c_{i,j}, s_{i+j}) = a_j \cdot b_i + c_{i,j-1} + s_{i+j}$. The sample pseudocode in Figure 4.3 forms the product $a \cdot b$ in the k-word array s by employing the MULADD2 kernel operation. Other primitive operations (e.g., MUL2ADD2 and SQUADD are used in squaring, i.e., when $a = b$.

The plain C code fragments are obtained by replacing the addition and multiplication statement sequences given in the previous sections. It is clear that the use of primitive operations has drastically reduced the code size. Moreover, the performance gained by the kernel operations exceeds the benefits of the extended types in both speed and code size.

4.3. Implementation Results

The proposed kernel of operations were implemented in the assembly languages of the Intel 486, Intel Pentium, and Sparc V9 machines. The Intel 486 DX4 processor has the speed of 100 MHz, and runs the NextStep operating system v3.3. We used the C compiler of the NextStep. Microsoft Visual C++ v4.2 and Intel VTune v2.0 are used in the development and analysis for the Intel Pentium processor on a Windows NT 3.51 system. The SPARCompiler SC4.0 is used on a UltraSparc-II V8+ system. The C compilers are configured to obtain speed-optimized code.

We implemented the 512-bit and 1024-bit bit modular exponentiation operations which are common in the RSA and Diffie-Hellman algorithms. The exponentiation algorithm is the binary method [5] using the Montgomery multiplication [8, 20]. The size of the modulus is 512 bits and 1024 bits. The exponent is selected as 1-word (32-bit) and full-size (the size of the modulus). Tables 4.2 and 4.3 give the timings in milliseconds for these operations.

TABLE 4.2: Modular exponentiation timings for 32-bit exponent in milliseconds.

	Modulus		C with	C with	
Processor and OS	Size (bits)	C	E.Types	Kernel	Asm.
i486DX4 100 MHz	512	29	28	26	10
NextStep v3.3	1024	110	103	95	39
UltraSparc-II V8+	512	8	5	6	3
Solaris 5.5.1	1024	31	21	23	12
Pentium 120 MHz	512	15	11	8	5
NT v3.51	1024	57	43	28	18

The fastest implementation is obtained using assembly language programming. For example, the assembler implementation of full-size modular exponentiation with 1024 bits is about 3.63 times faster than the standard C implementation on the Pentium machine. The standard C coupled with kernel operations produces a code which is 2.33 times faster than the standard C code, which is about 64 % of the speed increase gained by the assembler implementation. Table 4.4 illustrates the speedup of the other three approaches to the standard C implementation for performing modular exponentiation where the exponent is the full size (i.e., it is equal to the modulus size).

On the UNIX machines (NextStep and Solaris), we implemented the kernel operations using *functions*, since in-line assembly coding is not flexible due to inability to access

TABLE 4.3: Modular exponentiation timings for full-word exponent in milliseconds.

Processor and OS	Modulus Size (bits)	C	C with E.Types	C with Kernel	Asm.
i486DX4 100 MHz	512	488	405	363	205
NextStep v3.3	1024	3,775	3,195	2,800	1,559
UltraSparc-II V8+	512	150	103	106	55
Solaris 5.5.1	1024	1,144	790	795	414
Pentium 120 MHz	512	206	151	91	59
NT v3.51	1024	1,618	1,166	694	446

the C variables in the inline assembly code. In this case, the speed increase gained by the use of kernel operations is given away due to the function calling overhead. For example, the C with kernel operations case is slightly slower than the C with extended types case on the Sparc machine running Solaris. It seems that the extended types are more efficiently utilized by the C compiler on the Next machine. Thus, we observe a small amount of speedup comparing the C with kernel operations to the C with extended types.

4.4. Conclusions

We have proposed a design methodology and a small set of kernel operations for obtaining high-speed implementations of the number-theoretic cryptographic algorithms. It is shown that up to 64 % of speed increase gained by the use of full assembler implementation can be obtained by coding only this small set of kernel operations in the assembly language of the underlying processor. It is preferred that the development system provide in-line assembly coding in order to avoid the overhead of function calling in implementing the kernel operations.

TABLE 4.4: Speedup with respect to the standard C implementation.

	Modulus	C with	C with		Kernel
Processor and OS	Size (bits)	E.Types	Kernel	Asm	vs Asm
i486DX4	512	1.20	1.34	2.38	56 %
NextStep v3.3	1024	1.18	1.34	2.42	55 %
UltraSparc-II V8+	512	1.46	1.42	2.73	52 %
Solaris v5.5.1	1024	1.45	1.45	2.76	53 %
Pentium	512	1.36	2.26	3.49	65 %
NT v3.51	1024	1.39	2.33	3.63	64 %

This approach allows the programmer to drastically reduce assembly language programming while gaining a significant speedup. Since the assembler portion is quite minimal (a total of 60 lines at most), the maintainability and testability of the code are retained. The code can easily be ported to a different platform (processor) by implementing only the suggested set of kernel operations. Furthermore, the kernel operations proposed in this chapter are easy to implement in hardware. If they are available as instructions (or macros) on microprocessors or signal processors, high-speed implementations of number-theoretic cryptographic algorithms can easily be obtained.

4.5. Implementation of Kernel Operations

The Pentium and Sparc V9 assembler implementations of the proposed kernel are given in the following two sections.

4.5.1. Intel Pentium

```
ADD(C,S,a,b)
mov     eax,dword ptr [a]
mov     ebx,dword ptr [b]
mov     dword ptr [C],0
add     eax,ebx
mov     dword ptr [S],eax
setc    byte ptr [C]
```

```
ADD2(C,S,a,b,c)
mov     eax,dword ptr [a]
mov     ebx,dword ptr [b]
xor     edx,edx
mov     ecx,dword ptr [c]
mov     dword ptr [C],edx
add     eax,ebx
setc    byte ptr [C]
add     eax,ecx
mov     dword ptr [S],eax
adc     dword ptr [C],0
```

```
MUL(C,S,a,b)
mov     eax,dword ptr [a]
mul     dword ptr [b]
mov     dword ptr [C],edx
mov     dword ptr [S],eax
```

```
MULADD(C,S,a,b,c)
mov     eax,dword ptr [a]
mov     ebx,dword ptr [c]
mul     dword ptr [b]
add     eax,ebx
adc     edx,0
mov     dword ptr [S],eax
mov     dword ptr [C],edx
```

```
MULADD2(C,S,a,b,c,d)
mov     eax,dword ptr [a]
mov     ebx,dword ptr [c]
mul     dword ptr [b]
add     eax,ebx
mov     ebx,dword ptr [d]
adc     edx,0
add     eax,ebx
adc     edx,0
mov     dword ptr [S],eax
mov     dword ptr [C],edx
```

```
MUL2ADD2(CC,C,S,a,b,c)
mov     eax,dword ptr [a]
mul     dword ptr [b]
add     eax,eax
mov     dword ptr [CC],0
adc     edx,edx
mov     ebx,dword ptr [c]
adc     byte ptr [CC],0
add     eax,ebx
adc     edx,0
mov     dword ptr [S],eax
mov     dword ptr [C],edx
adc     byte ptr [CC],0
```

```
SQU(C,S,a)
mov     eax,dword ptr [a]
mul     eax
mov     dword ptr [S],eax
mov     dword ptr [C],edx
```

```
SQUADD(C,S,a,b)
mov     eax,dword ptr [a]
mov     ebx,dword ptr [b]
mul     eax
add     eax,ebx
adc     edx,0
mov     dword ptr [S],eax
mov     dword ptr [C],edx
```

4.5.2. Sparc V9

```
ADD(C,S,a,b)
clruw   %o2
clruw   %o3
add     %o2,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
ADD2(C,S,a,b,c)
clruw   %o2
clruw   %o3
clruw   %o4
add     %o2,%o3,%g1
add     %g1,%o4,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
MUL(C,S,a,b)
clruw   %o2
clruw   %o3
mulx    %o2,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
MULADD(C,S,a,b,c)
clruw   %o2
clruw   %o3
clruw   %o4
mulx    %o3,%o2,%g1
add     %g1,%o4,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
MULADD2(C,S,a,b,c,d)
clruw   %o2
clruw   %o3
clruw   %o4
clruw   %o5
mulx    %o2,%o3,%g1
add     %o4,%o5,%g2
add     %g1,%g2,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
MUL2ADD2(CC,C,S,a,b,c)
clruw   %o3
clruw   %o4
clruw   %o5
mulx    %o4,%o3,%g1
mov     %g0,%g2
addcc   %g1,%g1,%g1
movcs   %xcc,1,%g2
addcc   %g1,%o5,%g1
movcs   %xcc,1,%g2
stuw    %g1,[%o2]
srlx    %g1,32,%g1
stuw    %g2,[%o0]
retl
stuw    %g1,[%o1]
```

```
SQU(C,S,a)
clruw   %o2
mulx    %o2,%o2,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

```
SQUADD(C,S,a,b)
clruw   %o2
clruw   %o3
mulx    %o2,%o2,%g1
add     %g1,%o3,%g1
srlx    %g1,32,%g2
stuw    %g1,[%o1]
retl
stuw    %g2,[%o0]
```

FIGURE 4.2: Multiplication of two multi-precision integers: $s = a \cdot b$ where $a = (a_3a_2a_1a_0)$ and $b = (b_3b_2b_1b_0)$. Here, $x_{i,j}$ denotes a partial product.

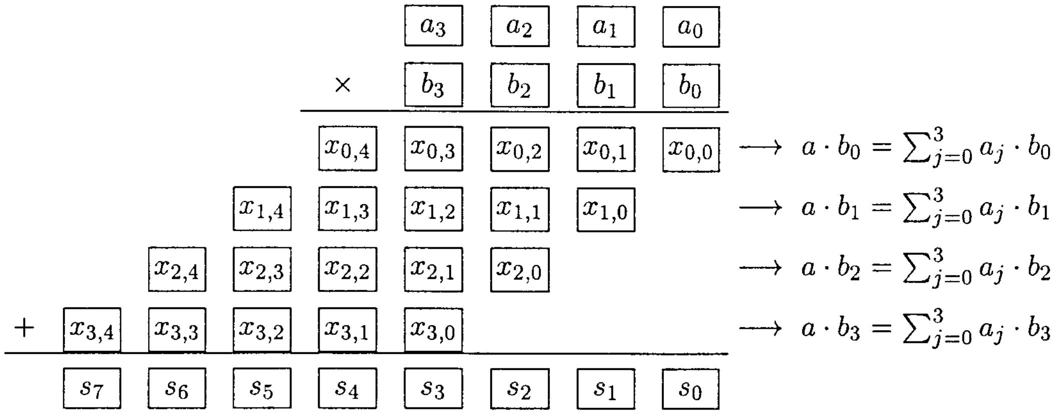

a_3 a_2 a_1 a_0
$\times$ b_3 b_2 b_1 b_0
$x_{0,4}$ $x_{0,3}$ $x_{0,2}$ $x_{0,1}$ $x_{0,0}$ $\longrightarrow a \cdot b_0 = \sum_{j=0}^{3} a_j \cdot b_0$
$x_{1,4}$ $x_{1,3}$ $x_{1,2}$ $x_{1,1}$ $x_{1,0}$ $\longrightarrow a \cdot b_1 = \sum_{j=0}^{3} a_j \cdot b_1$
$x_{2,4}$ $x_{2,3}$ $x_{2,2}$ $x_{2,1}$ $x_{2,0}$ $\longrightarrow a \cdot b_2 = \sum_{j=0}^{3} a_j \cdot b_2$
$+$ $x_{3,4}$ $x_{3,3}$ $x_{3,2}$ $x_{3,1}$ $x_{3,0}$ $\longrightarrow a \cdot b_3 = \sum_{j=0}^{3} a_j \cdot b_3$
s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0

FIGURE 4.3: Pseudocodes for multi-precision addition and multiplication.

Addition

```
C := 0
for i=0 to k-1 do
   ADD2(C,S,a[i],b[i],C)
   s[i] := S
s[k] := C
```

Multiplication

```
for i=0 to 2*k-1 do   s[i] := 0
for i=0 to k-1 do
  C := 0
  for j=0 to k-1 do
    MULADD2(C,S,a[i],b[j],s[i+j],C)
    s[i+j] := S
  s[i+k] := C
```

5. AN INSTRUCTION SET ARCHITECTURE FOR CRYPTOGRAPHY

5.1. Introduction

The execution time of a certain task is computed by accumulating the time consumed for each instruction executed in the life time of that task. The execution time is a function of the instruction count and cycles a processor consumes for each instruction. Thus, a software runs faster if the frequently used instructions are executed in fewer clock cycles. In order to obtain the instruction histogram of a number-theoretic cryptosystem software, we modified our programs to compute the number of instructions actually executed on Intel Pentium platforms. Then, we deduced which instructions should be optimized for a better performance. In addition to that, we also proposed a set of new instructions which would speed up numerous cryptographic algorithms including RSA, DSA, RC5 and DES.

In the foregoing sections, we explore individual instructions of the Intel and SPARC architectures and then propose new instructions to obtain high-speed number-theoretic cryptographic applications. Because of the particular challanges of fast software implementation of cryptographic algorithms on Intel 486/Pentium/Pentium Pro, and because of the pervasiveness of this processor family, we concentrated on Intel Pentium architecture. By doing well on these difficult-to-optimize-for vehicles we expect to do well on any modern, 32 or 64-bit processor.

We restrain ourselves from introducing a completely new instruction set architecture since it would hardly find actual application and introduce export restriction disadvantage.

5.2. Instruction Distribution of Cryptographic Software

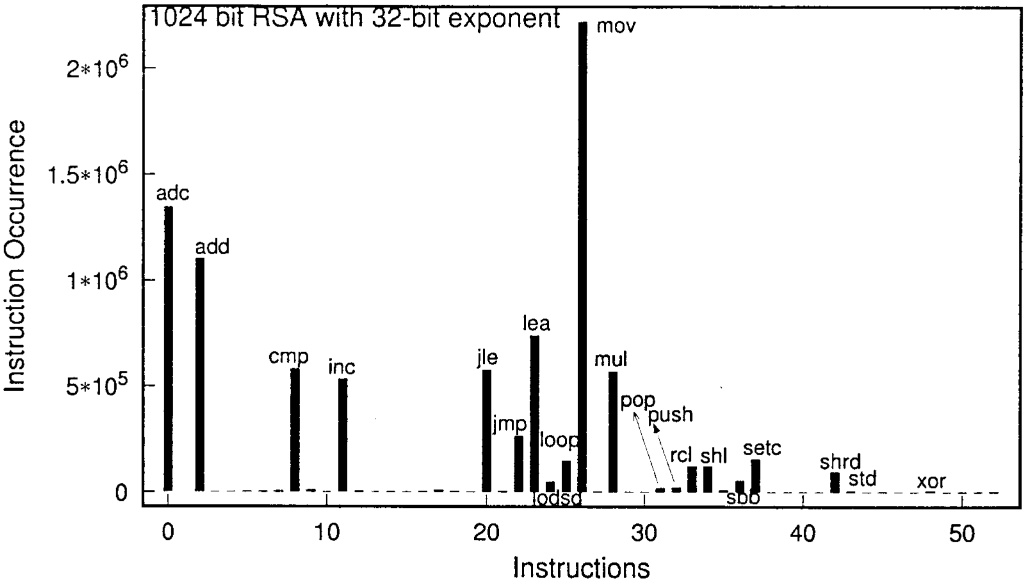
Our RSA implementation on the Intel Pentium processor yields the instruction distribution depicted on Tables 5.1 and 5.2. These figures denote the instruction distribution of Intel Pentium assembler programs. Table 5.1 denotes the number of instructions executed to perform a modular exponentiation with 32-bit exponent and 1024-bit modulus. This corresponds to 1024-bit RSA encryption.

Table 5.2 denotes the number of instructions executed to perform a modular exponentiation with 1024-bit exponent and 1024-bit modulus. However, 1024-bit exponent is splited into two 512-bit exponents, and 1024-bit modulus is splited into two 512-bit moduli. Then, the final operation is accomplished by combining two 512-bit exponentiations using the Chinese Remainder Theorem [64]. This corresponds to 1024-bit RSA decryption.

Both figures indicate that the `mov` instruction is the most used instruction. The Intel Pentium processor executes `mov` instruction in single cycle and this instruction can be paired in either U or V pipes. Thus, frequent use of the `mov` instruction does not introduce a significant bottleneck. Similar observation is also made for `add` and `adc` instructions. However, the compiler or the assembler programmer must be aware that `adc` instruction is not pairable in the V pipe. On the other hand, the `mul` instruction, with its 10-cycle execution time, is not pairable in either pipe. Therefore, one `mul` instructions consumes roughly $10 \div 0.5 = 20$ times more CPU cycles compared to `mov` instruction, assuming 100% pairing for `mov` instruction. That introduces a significant drawback in the implementations of number-theoretic cryptosystems.

One interesting instruction is the `lea`, load effective address, with its relatively high occurence. `lea` is also used to add a register to another register scaled by 2, 4, or 8, and an 8-bit immediate constant, resulting in a three additions in one cycle. This instruction adds & scales registers in base address calculations, finds a common usage in regular additions

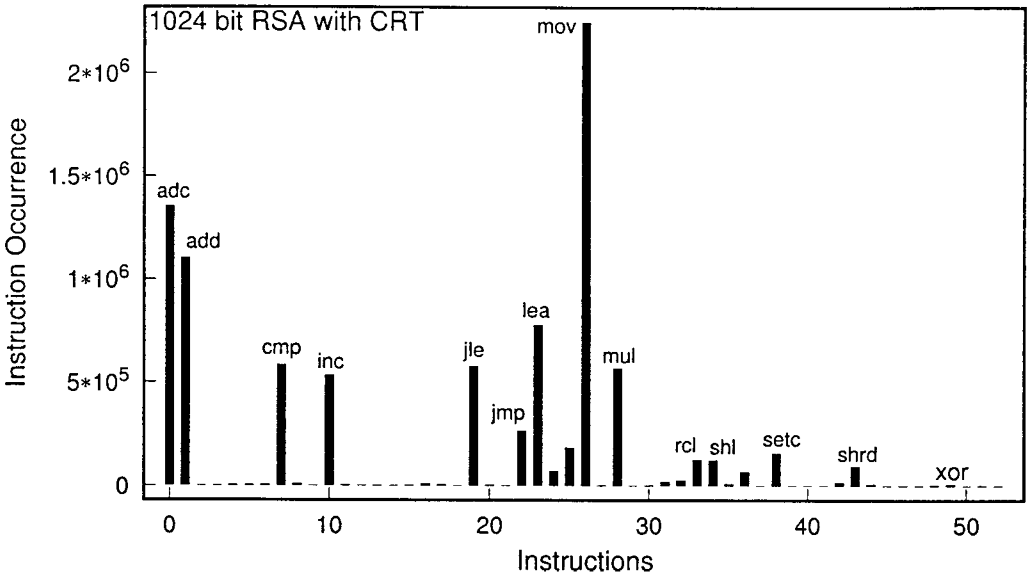
FIGURE 5.1: Intel Pentium instruction counts for 1024-bit modular exponentiation with 32-bit exponent.



and address increments in loops, which increased its occurrence count.

Instructions rcl and shl are heavily used in Montgomery squaring to compute $2 \cdot a \cdot b$, where a and b are two 32-bit quantities. rcl and shl instructions are preferred over shld, because the latter one is not pairable in either pipe and comsumes a relatively higher number of cycles. setc is used to eliminate a considerable number of conditional jump statements, thereby preventing BTB (branch target buffer) misses. cmp, jle and inc instructions are primarily used to set up loops and manage loop counters. Similar counts for these instructions on Table 5.1 confirms this observation.

FIGURE 5.2: Intel Pentium instruction counts for 1024-bit modular exponentiation with CRT using 1024-bit exponent.



5.3. Case 1: Intel Pentium and PentiumPro Processors

Many features of the Intel Pentium and Pentium Pro are inherited from the Intel x86 architecture which dates back to more than 20 years ago. The relevant limitations of the 486/Pentium/Pentium Pro processor family are: a small register set and a two operand architecture [65, 66, 67, 68]. In more detail, these chips are 32-bit CISC processors with current ones running at upto 233 MHz. They have eight general purpose registers, and six segment registers. The instructions generally work on two operands ($A \leftarrow A$ op B) instead of three ($A \leftarrow B$ op C). Pentium and Pentium Pro have separate on-chip instruction and data caches and pipelined instruction decoding. Pentium has two execution pipelines, U and V, where a limited set of instructions are paired. On the other hand, Pentium Pro has three parallel instruction decoders, out-of-order issue and out-of-order execution features.

We propose additional instructions on Table 5.1 to the current Pentium instruction set architecture. We used the cryptographic kernel operations introduced in the previous chapter to derive these instructions.

Operands of the instructions are shown as registers, but might also be memory operands. It might not be easy to encode the three-operand instructions on the intel Pentium architecture. However, the operands may be implicit operands, i.e., the destination might be hardcoded as `edx,eax` register pair as in the existing `mul` instruction.

TABLE 5.1: Proposed instructions to the Intel Pentium processor.

Instruction	Operands	Operation	Description
`muladd`	r_1, r_2, r_3	$(r_1, r_2) = r_1 \cdot r_2 + r_3$	multiply-add
`mul2add`	r_1, r_2, r_3	$(C, r_1, r_2) = 2 \cdot r_1 \cdot r_2 + c_3$	multiply by 2, then add
`mulgf2`	r_1, r_2	$(r_1, r_2) = r_1 \cdot r_1$	GF(2) multiply

The result of the first operation always fits into two registers, because the maximum value is less than 2^{2w}, where w is the wordsize of the operands: $(2^w - 1)(2^w - 1) + (2^w - 1) = 2^{2w} - 2^w < 2^{2w}$.

The `mul2add` instruction first multiplies two registers specified by its operands. Then, it shifts the product to the left by one bit position and adds the third operand to this shifted product. Then, the result fits in 65 bits for 32 bit operands, hence the carry bit C is used as part of the result.

The last instruction, `mulgf2`, multiplies two polynomials of length $w = 32$ bits over GF(2), and generates $2w = 64$ bit product in the destination register pair. Hardware requirements of this instruction is fairly less: inhibition of carry generation in the integer multiplication hardware yields the desired operation. A comparison of GF(2) multiplication and exponentiation operations using `mulgf2` instruction is elaborated in [19, 50].

5.4. Case 2: Intel MMX Technology

We have analyzed the instruction set and related architectural features of the MMX technology for obtaining high-speed implementations of certain cryptographic algorithms. In this section, we mainly concentrated on the RSA and DSS algorithms, DES, IDEA and RC5 block ciphers. We implemented the RSA and DSS algorithms on a MMX platform, and obtained timings on a 233 MHz Pentium Pro/MMX processor. These timings indicate that the number-theoretic cryptographic algorithms implemented using the MMX instructions are significantly *slower* than those implemented using Pentium instructions. This turns out to be mainly due to the fact that the MMX architecture performs signed arithmetic and lacks certain instructions such as 16-bit and 32-bit unsigned multiply and multiply-add. After carefully examining the kernel of operations for the considered cryptographic algorithms, we propose a set of new instructions for the next generation MMX technology.

5.4.1. RSA and DSS

The current MMX instruction set has three multiplication instructions. Two of them multiply two signed 16 bit words and compute the low (`PMULL`) or high (`PMULH`) 16 bits of the product. Another instruction multiplies two signed 16 bit words and adds two such products (`PMADD`). Each instruction executes four of such multiplications in parallel However, number-theoretic cryptographic applications require unsigned multiplications. Therefore, we propose the set of instructions on Table 5.2 to the current MMX instruction set.

On Table 5.2, each register r_i is a 64-bit MMX register. The second operands are also represented by a registers, but they might be 64-bit memory locations similar to other

TABLE 5.2: Proposed instruction additions to the Intel MMX technology.

Instruction	Operation
paddq r_1, r_2	$r_1 = r_1 + r_2$
padcq r_1, r_2	$r_1 = r_1 + r_2 + C$
pmuluwd r_1, r_2	$r_1(63-32) = r_1(47-32) \cdot r_2(47-32)$
	$r_1(31-0) = r_1(15-0) \cdot r_2(15-0)$
pmuludq r_1, r_2	$r_1 = r_1(31-0) \cdot r_2(31-0)$
pmuluq r_1, r_2	$(r_1, r_2) = r_1 \cdot r_2$
pmaddudq r_1, r_2	$r_1 = r_1(63-32) \cdot r_2(63-32) \ + \ r_1(31-0) \cdot r_2(31-0)$

MMX instructions. Indices in the paranthesis denote the bit numbers in the 64-bit MMX registers.

5.4.2. RC5

RC5 algorithm has three primitive operations [69] as listed below.

1. Two's complement addision and subtraction.

   These are readily available on MMX. However, 64-bit RC5 addition would benefit 64-bit addition and subtraction (`paddq, psubq`) instructions.

2. Bit-wise exclusive-OR, XOR.

3. Data dependent left and right rotate (spin), ROR, ROL.

   There is no rotate instruction in the MMX instruction set. Data dependent rotate (spin) instructions constitute the security core of the RC5 algorithm. These operations are emulated using shift and OR instructions. Thus, it is anticipated that data dependent left/right rotate instructions (`pror, prol`) would improve RC5 performance on MMX.

Thus, for a fast RC5 implementation exploiting the Intel MMX features, we find the instructions on Table 5.3 beneficial. The second operands are also represented by registers, but they might be 64-bit memory locations similar to other MMX instructions. On the table, $\lll$ and $\ggg$ denote left and right rotate (spin) operations.

TABLE 5.3: Proposed instruction additions for to the Intel MMX technology for RC5.

Instruction	Operation
`pror` $r_1, r_2/imm8$	$r_1 = r_1 \lll r_2/imm8$
`prol` $r_1, r_2/imm8$	$r_1 = r_1 \ggg r_2/imm8$
paddq r_1, r_2	$r_1 = r_1 + r_2$
psubq r_1, r_2	$r_1 = r_1 - r_2$

5.4.3. DES

DES has the following primitive operations [70]:

1. BIT-wise exclusive-OR, XOR.
2. Shift by immediate count.
3. Rotate by immediate count.

However, 64-bit registers and 64-bit shift/XOR MMX instructions eliminate the need for rotate instruction. Thus, we did not find a general instruction which would improve DES throughput on the MMX technology.

5.4.4. IDEA

The primitive operations of IDEA are listed below [71, 72]:

1. Bit-wise exclusive-OR, XOR.

2. Two's complement 16-bit addition modulo 2^{16}, ADD, no carry.

3. Unsigned 16-bit multiplication modulo $2^{16} + 1$.

The first two operations are readily available in the MMX instruction set. The third operation is a special multiplication modulo $2^{16} + 1$ where inputs are treated as unsigned integers, but 0 (zero) corresponds to -1, i.e. $2^{16} = -1$. This multiplication does not produce a 0 product provided that operands are not zero, because $2^{16} + 1$ is a prime. Thus, this operations is essentially a 16-bit unsigned multiplication where 0 is treated separately. The 32-bit signed multiplication instruction `pmull` can be used to multiply two unsigned 16-bit operands.

5.5. Case 3: SPARC V9

Sparc V9 is a 64-bit architecture downward compatible with the earlier Sparc family of processors [73, 74]. It contains 64-bit register and a 64-bit ALU which are important for development of number-theoretic cryptographic algorithms. It can perform arithmetic operations on two 64-bit operands such as addition, subtraction, multiplication and division. However, lack of certain 64-bit operations limits the maximum performance this architecture would yield.

The first problem appears in the 64-bit addition-with-carry. The carry-in bit is always the 32-bit carry bit. The 64-bit add-with-carry instruction adds two 64-bit operands and the 32-bit carry bit. Then, addition of a very long integer using multiple 64-bit addition instructions is not easily streamlined, requiring a manipulation with the carry bits, or using 32-bit arithmetic. The second problem is with the 64-bit multiplication instruction. The result of a 64 by 64-bit multiplication is 64-bit: the high order 64-bits are truncated, causing the developer to feed two 32-bit operands to obtain a 64-bit product.

We propose three additional instructions to the V9 architecture. These are listed on Table 5.4.

TABLE 5.4: Proposed instructions to Sparc V9 architecture.

Instruction	Operation
addcc $i\_or\_x\_cc, reg_{rs1}, reg\_or\_imm, reg_{rd}$	Add with carry $i\_or\_x\_cc$
addccc $i\_or\_x\_cc, reg_{rs1}, reg\_or\_imm, reg_{rd}$	Add with carry $i\_or\_x\_cc$ and modify cc's
umulx $reg_{rs1}, reg\_or\_imm, reg_{rd}$	Unsigned multiply, result in reg_{rd}, reg_{rs1} pair

Addition instructions addcc and addccc accepts either 32-bit carry (i_cc) or 64-bit carry (x_cc) as carry-out and carry-in. Both instructions compute $reg_{rd} = reg_{rs1} + reg\_or\_imm + i\_or\_x\_cc$. addccc also modifies the 32-bit or 64-bit carry flag ($i\_or\_x\_cc$) according to the result of the addition.

umulx is an unsigned extended multiplication instruction computing $(reg_{rd}, reg_{rs1}) = reg\_or\_imm \times reg_{rs1}$. If the operands are less than 32-bits, then the product fits in one 64-bit register. In this case, the result is in reg_{rd}, and reg_{rs1} is zeroized. However, if the product is larger than $2^{64} - 1$, then the high-order 64-bits of the product are stored in reg_{rs1}, and low-order 64-bits are stored in reg_{rd}. In either case, the initial contents of reg_{rs1} are overwritten.

6. CASE STUDIES

6.1. Introduction

We developed a number of cryptographic functions in C and in various assembly languages. In spite of many cryptographic algorithms available, only a subset of them found applications in real life. In the developed software routines, we experienced that the following algorithms are often practised:

a) RSA encryption and decryption [53, 75]

b) Diffie-Hellman key exchange [53, 75]

c) Data Encryption Standard (DES) [70, 76, 71, 77]

d) Secure Hash Algorithm (SHA-1) [78, 79]

e) MD2, Message Digest Algorithm [80]

f) MD4, Message Digest Algorithm [71]

g) MD5, Message Digest Algorithm [81]

h) Digital Signature Algorithm (DSA, DSS) [4]

i) Password based encryption and decryption (PBE) [53]

j) International Data Encryption Algorithm (IDEA) [71, 72]

k) RC2, encryption [71]

l) RC5, variable key-size encryption [69, 71]

m) Random number generation [82, 83]

We refrain from giving a complete list, because we believe that there is not an actual complete list of cryptographic algorithms. Some of the items on the presented list may be obsolete, and others may (and will) be added. In the following sections, our implementations on various platforms are introduced.

6.2. A Cryptographic Multi-Precision Library in C

A Cryptographic Multi-Precision (CMP) library is designed and developed for RSA Data Security. The library contains core functions of the RSA data security system including modular exponentiation. Programs are written in C and in i486 assembler. Montgomery multiplication and squaring routines are optimized for the Intel 486 processor.

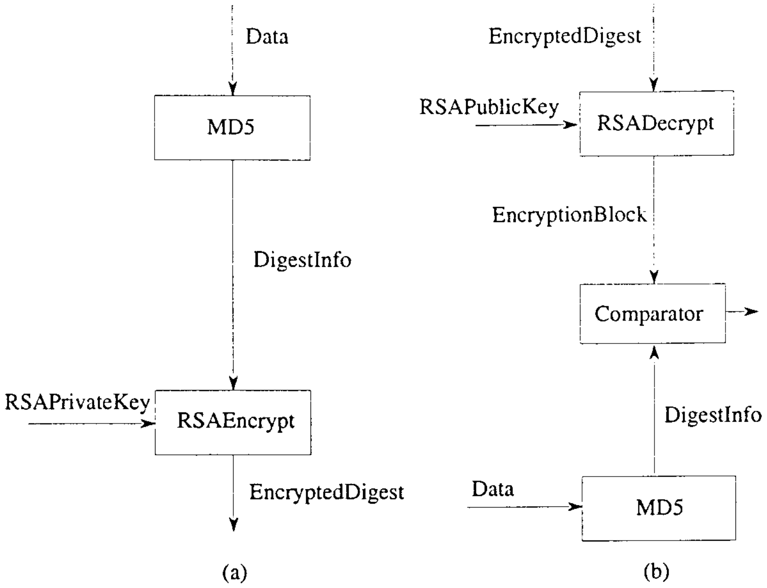
The CMP library contains arithmetic functions which can be used as building blocks of number-theoretic cryptosystems. These functions perform modular addition, subtraction, multiplication and exponentiation on multi-precision unsigned integers. The functions in this library are listed on Table 6.1. Each argument is of type `CMPInt`. This is a structure composed of three members: `value` an unsigned integer array where the words of a multi-precision integer are stored, `length` the number of words in the array, `space` the number of words reserved in memory.

6.3. RSA and MD5 Implementations on TMS320C16

A cryptographic library is developed for RSA Data Security. The library contains PKCS compliant 512-bit RSA crypto-system and MD5 message digest algorithm [53, 81, 84] for signature operations. Programs are written in TMS320C16 assembly language using a TMS320C16 add-on PC card and development package. The function calling sequences of signing and verification are depicted in Figure 6.1. In the figure, boxes represent function blocks. Function input and output arguments are BSafe compatible type, which enables the developed software package to interface with BSafe using ASN.1 data structures [85, 86].

This signal processor does not have an unsigned multiplication instruction. Thus, 16 by 16-bit unsigned multiplication is accomplished using 15-bit multiplication and corrective additions. If the product $t := a \times b$ is computed, the multiplication operation is

FIGURE 6.1: Signing and verification on TMS320C16 implementation.



written as:

$$\begin{aligned}
t &= a \times b \\
&= \sum_{i=0}^{15} a_i 2^i \sum_{j=0}^{15} b_j 2^j \\
&= \left(a_{15} \cdot 2^{15} + \sum_{i=0}^{14} a_i 2_i\right) \times \left(b_{15} \cdot 2^{15} + \sum_{j=0}^{14} b_j 2_j\right) \\
&= \underbrace{2^{30} \cdot a_{15} \cdot b_{15}}_{A} + \underbrace{2^{15} \cdot a_{15} \sum_{j=0}^{14} b_j 2^j}_{B} + \underbrace{2^{15} \cdot b_{15} \sum_{i=0}^{14} a_i 2^i}_{C} + \underbrace{\sum_{i=0}^{14} a_i 2^i \times \sum_{j=0}^{14} b_j 2^j}_{D}
\end{aligned}$$

where a, b and t are 16-bit quantities and subscripts are used as bit indices. Product in part D is computed by a multiplication instruction. Operands to this instruction are low order 15 bits of a and b. Low order 15 bits of b are left-shifted 15 bits and added to this partial product if $a_{15} = 1$ in part B. Similarly, low order 15 bits of a are left-shifted 15 bits and added if $b_{15} = 1$ in part C. Finally, in part A, a constant number 2^{30} is added if both $a_{15} = 1$ and $b_{15} = 1$.

A 512-bit RSA decryption takes about 1.5 second on TMS320C16. The major reason of this slow execution time is the lack of unsigned multiplication instruction as expressed in the previous paragraph.

MD5 algorithm heavily uses logical operations such as shift and rotate. However, shift instructions on this processor are also signed and there is no rotate instruction. Thus, an unsigned shift and rotate are also accomplished through emulation, introducing a drawback in performance.

As a conclusion, double-size accumulator and single-cycle multiplication are the only beneficial features of TMS320C16 for number-theoretic cryptographic algorithms. Conversely, lack of unsigned operations, i.e., multiplication and shift/rotate, restricted register set, and indirect program memory access supersedes the advantages.

6.4. Intel Cryptographic Library on The Pentium Processor

A cryptographic library is designed and developed for Intel Corporation. This project is currently extended to include elliptic curve cryptosystems. We named this software Intel Cryptographic Library (ICL). Programs are written in C and Pentium assembly language, and optimized for the Pentium processor. Up to our knowledge, the fastest RSA, MD5, MD2, DES, SHA, DSS, RC5 implementations on Intel processors are contained within this library.

RSA cryptosystem, password-based encryption, and Digital Signature Algorithm and a random number generator are written by myself. A 1024-bit RSA public key operation and a private-key operations takes 5 and 110 milliseconds, respectively, on an Intel Pentium 120 MHz system running Windows NT v4.0. Programs are developed using Microsoft Visual C++ v4.x.

6.5. RSA Implementation on Sparc V9

RSA crypto-system has been developed for Naval Research Laboratories, Internet Technologies Division. We designed and developed the library on Sun UltraSparc-II (V8+) based on a SPARC V9 processor system running Solaris 5.1 [73, 74]. Most of the functions are written in Sparc V9 assembly language, utilizing the 64-bit architecture of the processor.

We have developed the software as a replacement to RSA part of the RSA Data Security's BSAFE v3.0 cryptographic toolkit [85]. Software has a BSAFE-like interface featuring full PKCS compatibility with ASN.1, BER and DER [53, 86]. Programs are developed using SparCompiler v4.0. Up to our knowledge, the fastest RSA encryption and decryption functions on Sun SPARC V9 processor is contained within our library.

A 1024-bit input is encrypted by RSA algorithm in less than 10 milliseconds using the assembler version of the developed software running on a UltraSparc-II, a V9 powered V8+ Sun architecture. 1024-bit RSA decryption takes 94 milliseconds. RSA encryption and decryption benchmarks of the C version are 21 and 171 milliseconds, respectively. In the C versions, extended type `long long` is used. These figures are significant improvements over RSA BSafe benchmarks which takes 230 milliseconds for 1024-bit RSA decryption.

TABLE 6.1: List of CMP fuctions.

Function	Description
CMPMove (a, t)	$t := a$
CMPCompare (a, b)	$a <=> b$
CMPInc (a, x)	$a := a + x$, x one word
CMPDec (a, x)	$a := a - x$; x one word
CMPAdd (a, b, t)	$t := a + b$
CMPMul (a, b, t)	$t := a \cdot b$
CMPDiv (a, b, q)	$t := a \text{ div } b$
CMPRem (a, b, r)	$t := a \mod b$
CMPModAdd (a, b, n, t)	$t := a + b \mod n$
CMPModSub (a, b, n, t)	$t := a - b \mod n$
CMPModMul (a, b, n, t)	$t := a \cdot b \mod n$
CMPGCD (a, b, g)	$g := \text{gcd}(a, b)$
CMPModInv (a, n, t)	$t := a^{-1} \mod n$
CMPModExp (a, e, n, t)	$t := a^e \mod n$
CMPModExpCRT $(a, dp, dq, n, p, q, c, t)$	$t := ((M_1 - M_2) \cdot c \text{ mod } p) \cdot q + M_2$
	$M_1 := (C \text{ mod } p)^{dp} \mod p$
	$M_2 := (C \text{ mod } 1)^{dq} \mod q$

BIBLIOGRAPHY

1. P. Rogaway and D. Coppersmith. A software-optimised encryption algorithm. In R. Anderson, editor, *Proceedings of Fast Software Encryption*, pages 56–61, Cambridge, UK, December 9-11 1993. Springer-Verlag.

2. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

3. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.

4. Federal Information Processing Standards Publication, U.S. Department of Commerce/ NIST. *Digital signature standard (DSS)*, May 1994.

5. D. .E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, MA, second edition, 1981.

6. D. Laurichesse and L. Blain. Optimized implementation of RSA cryptosystem. *Computers & Security*, 10(3):263-267, May 1991.

7. C. N. Zhang. An improved binary algorithm for RSA. *Computers and Mathematics with Applications*, 25(6):15–24, March 1993.

8. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

9. M. Abdelguerfi, B. S. Kaliski Jr, and W. Patterson. Public-key security systems. *IEEE Micro*, 16(3):10–13, June 1996.

10. I. E. Bocharova and B. D. Kudryashov. Fast exponentation in cryptography. In *Proceedings of the 11th International Symposium, AAECC-11*, pages 146–157, Paris, France, July 17-22 1995. Lecture Notes in Computer Science n948, Springer-Verlag.

11. N. Koblitz. *A Course in Number Theory and Cryptography.* New York, NY:Springer-Verlag, New York, NY, second edition, 1994.

12. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.

13. V. Miller. Uses of elliptic curves in cryptology. In *Advances in Cryptology - CRYPTO 85, Proceedings, Lecture Notes in Computer Science, No.218*, pages 417–426, New York, NY, 1985. Springer-Verlag.

14. T. Itoh, O. Teachai, and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^t) using normal bases. *J.Soc.Electron.Communications (Japan)*, 44:31–36, 1986.

15. G. B.Agnew, T. Beth, R. C. Mullin, and S. A. Vanstone. Arithmetic operations in GF(2^m). *Journal of Cryptology*, 6(1):3–13, 1993.

16. S. T. J. Fenn, M. Benaissa, and D. Taylor. Finite field inversion over the dual basis. *IEEE Transactions on VLSI Systems*, 4(1):134–137, March 1996.

17. B. S. Kaliski. The Z80180 and big-number arithmetic. In *Dr.Dobb's Journal*, pages 50–58, September 1993.

18. S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology - EUROCRYPT 90, Lecture Notes in Computer Science, No. 473*, pages 230–244, New York, NY, 1990. Springer-Verlag.

19. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). In *Third Annual Workshop on Selected Areas in Cryptography*, pages 95–106, Queen's University, Kingston, Canada, 15-16 August 1996. IACR.

20. Ç. K. Koç, T. Acar, and B. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

21. E. F. Brickell. A survey of hardware implementations of RSA. In G. Brassard, editor, *Advances in Cryptology, CRYPTO'89 Proceedings*, Lecture Notes in Computer Science, vol.435, pages 368–370. Springer-Verlag, 1990.

22. N. Takagi. A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 35–42. IEEE Computer Society, 1991.

23. H. Orup and P. Kornerup. A high-radix hardware algorithm for calculating the exponential m^e modulo n. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 51–56. IEEE Computer Society, 1991.

24. C. Walter. Exponentiation using division chains. In *In Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 92–98, Asilomar, CA, July 1997. IEEE CS Press.

25. V. Dimitrov, G. Julliend, and W. Miller. Theory and applications for a double-base number system. In *In Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 44–51, Asilomar, CA, July 1997. IEEE CS Press.

26. V. Dimitrov, G. Jullien, and W. Miller. Algorithms for multi-exponentiation based on complex arithmetic. In *In Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 208–215, Asilomar, CA, July 1997. IEEE CS Press.

27. D. Naccache, D. M'Raihi, and D. Raphaeli. Can Montgomery parasites be avoided? a design methodology based on key and cryptosystem modifications. *Designs, Codes and Cryptography*, 5(1):73–80, January 1995.

28. J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. In *In Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 234–239, Asilomar, CA, July 1997. IEEE CS Press.

29. S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, June 1993.

30. D. Naccache and D. M'Raihi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, June 1996.

31. Ç. K. Koç. Montgomery reduction with even modulus. *IEE Proceedings - Computers and Digital Techniques*, 141(5):314–316, September 1994.

32. K. H. Rosen. *Elementary Number Theory and Its Applications.* Addison-Wesley Publishing Company, third edition, 1993.

33. A. J. Menezes and S. A. Vanstone. Systolic modular multiplication. In *Advances in Cryptology - CRYPTO'90 Proceedings Lecture Notes in Computer Science, No:537*, pages 619–624, New York, NY, 1991. Springer-Verlag.

34. Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, IT-24(1):106–110, January 1978.

35. M. J. B. Robshaw and Yiqun Lisa Yin. Elliptic curve cryptosystems. Technical report, RSA LaboratoriesTechnical Note, April 30 1997.

36. H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in GF(2^m). *IEEE Transactions on Computers*, 42(8):1010–1015, August 1993.

37. J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. Technical report, U.S. Patent Number 4,587,627, May 1986.

38. R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal normal bases in GF(p^n). *Discrete Applied Mathematics*, 22:149–161, 1988.

39. G. B. Agnew, R. C. Mullin, I. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3(2):63–79, 1991.

40. G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

41. G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R.A. Rueppel, editor, *Advances in Cryptology - EUROCRYPT 92, Lecture Notes in Computer Science, No. 658*, pages 163–173, New York, NY, 1992. Springer-Verlag.

42. R. Schroeppel, S. O'Malley, H. Orman, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology* - CRYPTO 95, *Lecture Notes in Computer Science, No. 973*, pages 43–56, New York, NY, 1995. Springer-Verlag.

43. E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). In *Advances in Cryptology - ASIACRYPT 96*, 1996.

44. A. Pincin. A new algorithm for multiplication in finite fields. *IEEE Transactions on Computers*, 38(7):1045–1049, July 1989.

45. D. C. FeldMeier. Fast software implementation of error detection codes. *IEEE/ACM Transactions on Networking*, 3(6):640651, December 1995.

46. Ç. K. Koç and T. Acar. Montgomery multiplication in GF(2^k). *Design, Codes, and Cryptography*, 14(1):59–67, January 1998.

47. R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

48. A. J. Menezes, editor. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.

49. R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, Cambridge, UK, revised edition, 1994.

50. Ç. K. Koç and T. Acar. Fast software exponentiation in GF(2^k). In *In Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, pages 225–231, Asilomar, CA, July 1997. IEEE CS Press.

51. Ç. K. Koç and T. Acar. Fast exponentiation in GF(2^k). *Submitted to IEEE Transactions on Computers*, 1998.

52. D. P. Maher. Trust in the new information age. *AT & T Technical Journal*, pages 9–16, September/October 1994.

53. RSA Laboraties. *The Public-Key Cryptography Standards*, November 1993.

54. IEEE. *IEEE P1363 Working Draft*, February 1997.

55. Paul F. Syverson. Limitations on design principles for public key protocols. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 62–73, Oakland, CA, 1996. IEEE, IEEE CS Press.

56. T. Beth and D. Gollmann. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications*, 7(4):458–466, May 1989.

57. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

58. National Institute for Standards and Technology, Federal Register. *Digital signature standard (DSS)*, 56:169 edition, August 1991.

59. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

60. Ç. K. Koç and T. Acar. A methodology for high-speed software implementations of number-theoretic cryptosystems. *Submitted to Computers & Security.*

61. M. Atkins and R. Subramaniam. PC software performance tuning. *IEEE Computer*, 29(8):47–54, Aug 1996.

62. B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

63. G. Satir and D. Brown. *C++: The Core Language.* O'Reilly & Associates, first edition, October 1995.

64. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.

65. D. Alpert and D. Avnon. architecture of the Pentium microprocessor. *IEEE Micro*, pages 11–21, June 1993.

66. D. Anderson and T. Shanley. *Pentium Processor System Architecture.* Mindshare, Inc., second edition, October 1995.

67. H-P. Messmer. *The Indispensible Pentium Book.* Addison-Wesley, 1995.

68. T. Coe, T. Mathisen, C. Mohler, and V. Pratt. Computational aspects of the Pentium affair. *IEEE Computational Science & Engineering*, pages 18–31, Spring 1995.

69. R. L. Rivest. *RC5 Encryption Algorithm.* RSA Data Security, April 1995.

70. Federal Information Processing Standards Publication, U.S. Department of Commerce/ NIST. *Data Encryption Standard (DES)*, December 1993.

71. B. Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., second edition, 1996.

72. W. Stallings. *Network and Internetwork Security: Principles and Practice.* Prentice Hall, IEEE Press, Englewood Cliffs, NJ, 1995.

73. D. L. Weaver and Editors T. Germond. *The SPARC Architecture Manual - Version 9.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

74. B. J. Catanzaro, editor. *The SPARC Technical Papers.* SUN Technical Reference Library. Springer-Verlag, 1991.

75. A. Salomaa. *Public-Key Cryptography.* Springer-Verlag, 1990. Editors: W.Braver and G.Rozenberg and A.Salomaa, EATCS Monographs on Theoretical Computer Science Vol.23.

76. E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, New York, 1993.

77. E. Biham. On modes of operation. In R. Anderson, editor, *Proceedings of Fast Software Encryption*, pages 116–120, Cambridge, UK, December 9-11 1993. Springer-Verlag.

78. Federal Information Processing Standards Publication, U.S. Department of Commerce/ NIST. *Secure Hash Standard (SHA)*, May 1993.

79. Federal Information Processing Standards Publication, U.S. Department of Commerce/ NIST. *Secure Hash Standard (SHA)*, April 1995.

80. B. Kaliski Jr. *The MD2 Message-Digest Algorithm.* RSA Data Security, April 1992.

81. R. Rivest. *The MD5 Message-Digest Algorithm.* RSA Data Security, April 1992.

82. T. Matthews. Suggestions for random number generation in software. Technical Report 1, RSA Laboratories's Bulletin, January 22 1996.

83. R. W. Baldwin. Proper initialization for the BSAFE random number generator. Technical Report 3, RSA Laboratories's Bulletin, January 25 1996.

84. B. Kaliski Jr. and M. Robshaw. *Message Authentication with MD5.* RSA Data Security, 1995.

85. RSA Data Security, Inc., Redwood City, CA. *BSAFE: A Cryptographic Toolkit, Version 2.1*, 1994.

86. B. S. Kaliski Jr. *A Layman's guide to a Subset of ASN.1, BER, and DER.* RSA Data Security, RSA Laboraties, first edition, November 1993.