Chapter 3

The PKCryptography Implementation: Background and Requirements

To implement a specific algorithm in public-key cryptography such as the RSA algorithm three levels of arithmetic functions needed to be constructed: (i) modular exponentiation; (ii) modular multiplication and modular reduction to perform modular exponentiation; (iii) addition to perform the modular multiplication operation. This chapter examines these functions and also discusses the Montgomery method in detail. the Montgomery method is employed here for implementing modular multiplication because of its efficiency, as will be discussed later in this chapter. This chapter first reviews previous works in hardware implementations of the RSA algorithm.

3.1 Previous RSA Implementations

Appendix I lists the implementations of RSA to date introduced in literature [Riv84], [Bri89], [BD89], [BFS91], [IWI92] [OPT93] [Sch96] or in their original papers with some of their important characteristics. They are either implementations or new method proposals for speeding up the RSA algorithm using dedicated hardware. In order to review the list in this section, they are grouped based on some important features and properties. The names of some works, which are just found in papers without any more information, are included in the table for completeness. Some information is just the published estimated performance or an estimation of the required hardware.

3.1.1 The First Efforts

Rivest and colleagues, as inventors of the RSA algorithm, began implementing the RSA scheme after it was introduced. Sandia started building the hardware at the same time [Dif92]. In 1979 a board was implemented by Sandia [Dif92] which could carry out this computation. It was followed by two chip implementations described in [RSW82]. Rivest first produced a board [Dif92] [Riv84] which could carry out one hundred digits modulus calculations in about a twentieth of a second. It was just a "proof of concept" and could not be commercial implementation because it was very expensive. The next Rivest implementation incorporated an NMOS chip using a 512-bit general-purpose ALU for computation of the RSA algorithm [Riv80]. Using conventional arithmetic the throughput of this design was expected to be around 1 kbps baudrate. However, this chip did not work completely.

The Brickel [Bri82] and Miyaguchi [Miy82] implementations are two other instances of early approaches, but they introduced new methods which were different from the conventional arithmetic methods (These methods are discussed in 3.4.1 and 3.4.2). As the result, they improved the modular exponentiation performance significantly in comparison with the other early research items. The Brickel method was used in AT&T implementations in 1987 [Dif92] and 1989 [Bri89] and they achieved to 20k bps baudrate using 172k gates. Walter and Eldridge in [WE90] modified this method and Cochrane implemented this modification in [Coc89] which is the base for the Plessey Semiconductor implementation. The implementation based on the Miyaguchi method was expected to have a 50 kbps performance using 100k gates [Miy82]. Madhavan and Peppard [MP89] exploited this method using GaAs technology and achieved a 180 kbps baudrate in a multi-chip implementation.

3.1.2 Other Approaches

Using DSP

Kochanski was the first to implement the RSA algorithm using digital signal processors (DSP) [Koc85]. Weiner [Bri89] used the DSP56000 series and computed a 512-bit modulus at a 4 kbps bit rate. Barret [Bar86] implemented his quotient estimation method, which is described in section 3.4.1, by using DSPs. Another instance was carried out by Dusse and Kaliski also using DSP56000s [DK90]. They achieved 11.6 kbps with the Chinese Remainder Theorem (CRT) for a 512-bit exponentiation.

Systolic Arrays

Even [Eve90] has suggested simple systolic arrays for modular reduction. There are some other proposals using systolic arrays to process modular exponentiation in parallel [Sau92] [KH91] [IWI92] [IWI92a] [Wal93] [Kor95] [GC96] [YB97] [Tio98]. Montgomery's algorithm has been the basis for many of these systolic array implementations.

Diminished Radix

Mohan and Adiga [MA85] suggested using a diminished radix form to provide faster reduction. But Meister [Mei90] has shown this suggestion decreases security and proposed a modification. Orton et al [OPT93] developed the diminished radix form for modular multiplication. However their algorithm has some problems with its security [OPT93].

Higher Radix

Baker [Bak87] and Morita [Mor89] considered higher radix than 2 as a generalised binary reduction method. Morita used a radix 4 division to perform modular reduction. The Orup et al proposals, [Oru95a] [OK91] and [OSA90], Dusse and Kaliski [DK90] and Miyaguchi [Miy82] implementations can be classified as using higher radix also. However, some of them use a multiprecision approach and not a generalisation of the binary reduction method as is described later in section 3.4.

Using Montgomery's Algorithm

Recent research has tended to use Montgomery's algorithm as a reduction method more than other technique [EW93] [Oru95a] [WTS97] [BG98] [Mar98] [BP99]. The Shand and Vuillemin work in [SV93] is a high performance implementation used Montgomery's algorithm. Section 3.5 describes the development of this method in detail.

Using Other Techniques

The work by Hoornaert et al [HDVG88] and the CrypTech chip [Sed87] [SG86] used an estimated quotient and a look up table for modular reduction. The CrypTech chip achieves a baudrate of 17 kbps by using this method. Vandemeulebroecke et al [VVDJ89] tried to speed up the scheme by using a redundant-signed-digit system. Orton et al [ORS86] used an asynchronous pulse-timed adder.

Commercial Chips

Cylink, AT&T, RSA Security CrypTech and Phillips have all produced commercial chips for implementing the RSA scheme [Sch96] [BFS92]. Pijnenburg [Pij98], EMI Thorn, Calmos Sys. Inc and AT&T with 50, 29, 28 and 20 kbps baudrates respectively have high baudrates. British Telecom and CNET have also produced commercial products [Sch96]. NTT products are based on a series of RSA implementations reported in [IOY94] [IOY96] [IOY97] [IOY98] [IOT98] [IOT99].

The Highest Clock Rate

The highest clock rate is claimed by the research conducted by Ivey et al [IWDS92]. They used the simple algorithm, which is called simultaneous multiply/division algorithm to perform the RSA scheme [ICHO89]. This algorithm is carried out by the add, shift, subtract and shift procedure. As a result of using a simple algorithm (so a simple circuit) and a fast technology (SOI) [WE93], their design has a target clock speed of 150 MHz.

3.1.3 The Highest Speed

To review the speed of implementations to date it is better to categorise fast RSA implementations into three categories as follows:

Board implementations: The implementation described in [SV93] is instance of this category as it uses more than one chip. They reported achieving 200 kbps bit rate for modular exponentiation.

Single chip implementations: The second category is the single chip implementations such as RSA200 described in [Poc98] with a 200k bps baudrate. This is in a series of RSA implementations described in [PP90] [LPP92] [Sch97a]. The other notable implementation in this category is the Orup chip [Oru95a] which achieved more than 100 kbps. All of these chips used multiple-bit scan technique which requires a considerable number of gate and transistors. A complete full custom design using dynamic logic [Sch97] provides these requirements for RSA200. However the highest speed instance of a single bit scan implementation is [IWSD92] with 60k bps encryption/decryption rate.

Commercial chips: The fastest commercial chip is the Pijnenburg production device [Pij98] which achieves encryption and decryption at a baudrate of 50 kbps for 512 bit keys using a 25 MHz clock. It has 400k transistors and uses 1 μm CMOS technology.

3.2 Modular Exponentiation

In order to compute the modular exponentiation M^e **mod** N for cryptography applications, the exponentiation M^e is not computed first, rather a part of the exponentiation operation, usually a partial multiplication, is performed. Then the partial result is reduced modulo N at each step. This is simply because the exponentiation result of large numbers is very large. As an instance, the result of M^e when M and e are 256-bit integers, requires about 2^{256} bits which is larger than the amount of total articles in the universe. Similar to the exponentiation procedure which performs a successive multiplications, modular exponentiation uses the same procedure as exponentiation, but it performs modular multiplication. Therefore the exponentiation methods can be used for modular exponentiation without loss of generality. The next section briefly describes the exponentiation algorithms, further details of which can be found in the appropriate references.

3.2.1 Binary Methods

The two commonly used algorithms which are for hardware implementation of exponentiation are the left-to-right and right-to-left algorithms. These algorithms, which are also called the square and multiply method, perform exponentiation by repeated squaring and multiplication.

Left-to-Right algorithm: For the integers M, N, C and $e = \{e_{n-1}e_{n-2}...e_0\}$, $e_i \in \{0,1\}$, the following algorithm computes $C = M^e$ **mod** N:

The Left-to-Right Exponentiation Algorithm

```
C := 1;
for i = n-1 downto 0 loop
    C := $C^2$ mod N;
    if $e_i$ = 1    then   C := M.C  mod N;
return C;
```

where $C = M^e$ **mod** N. According to Knuth [Knu98], this method was known as early as 200 B.C. by Indian mathematicians. However, a clear discussion of how to compute 2^n efficiently for arbitrary n was given by al-Uqlidisi, an Islamic mathematician, in 952 A.D. Beginning initially with C = 1, this algorithm sets C to C^2 and then, if $e_i = 1$, sets C to C.M. The exponent bits are scanned from left to right and the zeros before the MSB are ignored. The following binary representation of the exponent e can be used for validity of this algorithm as:

$$e = \sum_{i=0}^{n-1} e_i.2^i = ((...((e_{n-1}).2 + e_{n-2}).2 + ...).2 + e_1).2 + e_0$$

Therefore the exponentiation can be expressed as:

$$M^e = M^{((\ldots((e_{n-1}).2 + e_{n-2}).2\ldots.) .2 + e_1).2 + e_0)} = ((\ldots((M^{e}{}_{n-1})^2)^2.M^{e}{}_{n-2})^2.M^{e}{}_1)^2.M^{e}{}_0$$

For example:

$$M^{23} = M^{(10111)b} = (((((1)^2.M)^2)^2.M\ )^2.M)^2.M$$

The left-to-right algorithm involves $\lfloor \log_2{}^e \rfloor$ squarings and $\nu(n) - 1$ multiplications (or $\lfloor \log_2{}^e \rfloor + \nu(n) - 1$ multiplications if both operands use the multiplier) when both operations on the initial 1 are ignored. $\nu(n)$ is the number of non-zero bits in the binary representation of e. In a hardware implementation, this algorithm requires one storage register to keep the intermediate result.

The right-to-Left algorithm: for i from 0 up to k-1, this algorithm first sets C to C.M if $e_i = 1$ and then sets M to M^2.

The algorithm can be also written for modular exponentiation as follows :

The Right-to-Left Exponentiation Algorithm

```
C   := 1;
for i = 0 to n-1  loop
      IF $e_i$ = 1  THEN   C := M.C   mod N;
      M :=  $M^2$  mod N;
return   C;
```

where $C = M^e$ **mod** N. According to Knuth [Knu98], al-Kashi an Iranian mathematician stated this algorithm about 1414 A.D. The method is closely related to the multiplication procedure used by Egyptian mathematicians as early as 1800 B.C and widely used in Russia in the nineteenth century. It is often called "Russian peasant method" of multiplication.

This method is easily justified by a consideration of the sequence of exponents in the calculation. Since:

$$e = \sum_{i=0}^{n-1} e_i.2^i = 2^{n-1}.e_{n-1} + 2^{n-1}.e_{n-1} + \ldots + 2^1.e_1 + 2^0.e_0$$

The exponentiation can be written as:

$$M^e = M^{((\ldots((e_0 + 2^1 . e_1 + \ldots + 2^{n-2} . e_{n-2} + 2^{n-1} . e_{n-1})} = (M^{e_0}).(M^{2.e_1})\ldots(M^{2^{n-2} . e_{n-2}}).(M^{2^{n-1} . e_{n-1}})$$

$$M^e = (M)^{e_0}.(M^2)^{.e_1}\ldots(M^{2^{n-2}})^{e_{n-2}}.(M^{2^{n-1}})^{e_{n-1}}$$

and it corresponds to the algorithm procedure.

For example $M^{23} = M^{(10111)b} = (((((1 . M) . M^2) . M^4)) .M^{16})$.

Ignoring multiplication by the initial 1 and also ignoring the last squaring which will not be used, the number of multiplications required is $\lfloor \log_2 e \rfloor + \nu(n) - 1$. Squaring and multiplication can be performed in parallel in this method [Riv84] [Kor95] [Poc98]. Two extra registers are needed to keep the partial result and the squaring result.

3.2.2 Addition Chains

In order to minimise the number of multiplications required for an exponentiation, addition chain techniques can be used. An addition-chain for an integer n is a sequence of integers e_i satisfying

1. $e_0 = 1, e_r = n$
2. $e_i = e_j + e_k$, , where $0 \leq j \leq i \leq r$.

The given number n occurs in the chain. So each element can be expressed as a sum of two elements previously computed in the chain. An addition chain for a given exponent e is an algorithm for computing M^e. In this method, exponentiation is performed by a series of multiplications such that each multiply takes as input the values of two previous multiplies, starting with the initial values M and 1. The exponentiation process starts with M^1, then computes M^{e_k} using the two previously computed values M^{e_i} and M^{e_j} as:

$$M^{e_k} = M^{e_i}.M^{e_j}$$

The number of multiplications required is the length of the addition chain. As an instance, M^{23} can be computed using the addition chain 1, 2, 3, 5, 10, 20, 23 as:

$M^{23} = ((((1.M).M).\ M).\ M^2).\ M^5).M^{10}).M^3$

So 6 multiplications are required using the addition chain method versus 7 multiplications in the binary method.

The length of the minimum length addition chain for n is denoted by $l(n)$. The problem of determining $l(n)$ for an arbitrary integer n is the main concern in addition chains. This problem was apparently first raised by H. Dellac in 1894 [Knu98]. Clearly the minimum length addition chain for an exponent e minimises the number of total squarings and multiplications required for the exponentiation M^e.

Some interesting results and observations concerning the minimum length $l(n)$ in addition chains can be listed as:

1. The minimum length addition chain is not always unique.
2. The upper bound can be obtained by constructing an addition chain from its binary representation. So

   $l(n) \leq \lfloor \log_2{}^e \rfloor + \nu(n) - 1$
3. The lower bound of $l(n)$ as expressed in [Sch73] is:

   $l(n) \geq \log_2{}^n + \log_2{}^{\nu(n)} - 2.13$
4. Determining the shortest addition chain for a positive integer was shown to be an NP-hard problem [DLS81].

Star Chains: A star chain is an addition chain for which each element a_i is always the sum of the previous element a_{i-1} and some earlier element a_j, $j \leq i-1$. This special class of addition chain can be more efficiently implemented in hardware as the previous result is one of the operands [SV93] [Knu98].

Addition sequence: An addition sequence is an addition chain containing the given numbers which occur in the sequence.

In general, the addition chains approach for exponentiation requires a considerable amount of memory to keep the intermediate power results. More details in using addition chains for exponentiation can be found in Knuth [Knu98].

3.2.3 The r-ary Methods

The binary method is generalised to a higher radix or r-ary method by Knuth [Knu98]. Knuth denoted this method as m-ary because m was the radix in his description. This method is referred to as r-ary here, because r is the radix. The r-ary method is based on the representation of the exponent in the radix $r = 2^k$ as:

$$e = \sum_{i=0}^{n'-1} e_i.r^i = ((...((e_{n'-1}).r + e_{n'-2}).r + ...).r + e_1).r + e_0$$

The digits of the exponent e are scanned and subsequent multiplications are performed accordingly. In this method, first the values of M^p are precomputed and stored where $p \in \{0,1,...,k-1\}$. Then the k-bit digits of e are scanned at a time from the most significant to the least significant. k is called the window size. It has been shown the window size k = 5 for 512-bit and k = 6 for 1024-bit exponent size are optimal choices [EK94] [Koc94] [PB96].

Thurber's Modification: Thurber [Thu73] [Knu98] observed that only odd powers are needed by rewriting all non-zero digits as $e_i = 2^{h_i}.e'_i$ where e'_i is an odd integer. So a half size pre-computed table is required and this results in a reduced computing time and registers for the table. Therefore the modified left-to-right r-ary method is as follows:

The Modified Left-to-Right r-ary Exponentiation Method

```
C := 1;
Precomputation:
    $M^2$ :=M.M;
    for i = 1 to $2^{k-1}-1$ loop
        $M^{2i+1}$ := $M^{2i-1}.M^2$;
```

```
Precomputation End;
for i = n-1 downto 0 loop
if $e_i$ = 0 then $e_i'$ = 0; $h_i$ = 0;
            else  $e_i'$ = $2^{-h_i}.e_i$ such that $e_i = 2^{h_i}.e_i'$ ($e_i'$ is odd)
       C := $C^{2^{k-hi}}$ . $M^{e'i}$ mod N;
       C := $C^{2^{hi}}$ . mod N;
return   C;
```

where C = M^e **mod** N. As an example, the following expansion shows how the exponent e = 1011011000010 can be encoded with odd digits in the 4-bit window:

$$\mathbf{e} = \underset{\mathbf{e'}_3=9,\ h_3=0}{\underline{1\ 0\ 0\ 1}}\quad \underset{\mathbf{e'}_2=3,\ h_2=2}{\underline{1\ 1\ 0\ 0}}\quad \underset{\mathbf{e'}_1=1,\ h_1=2}{\underline{0\ 1\ 0\ 0}}\quad \underset{\mathbf{e'}_0=5,\ h_0=1}{\underline{1\ 0\ 1\ 0}}$$

Then the modified left-to-right r-ary method can be used for the exponentiation of M^e.

The encoding can be performed either right-to-left or left-to-right. It is independent from scanning the exponent in the exponentiation process which can be either right-to-left or left-to-right as expressed in [Oru95a]. In the right-to-left r-ary exponentiation with precomputing odd powers the final squaring is not required so it needs h_{n-1} squarings less than the left-to-right method.

The r-ary method reduces the number of multiplications from ν(n) for the binary case to the number of non-zero e'_is.

This approach is also called constant length non-zero windows or CLNW [Koc94]. This can be compared with the variable length non-zero window developed by Bos and Coster [BC89]. In their approach first e is rewritten in non-zero and zero words. The length of the non-zero words can be a variable and k-bit maximum size. All odd powers ($M^{3 \cdots} M^{k-1}$) are not pre-calculated in this approach, rather, as Bos and Coster proposed, an addition sequence can be found in order to pre-compute only the required powers based on this sequence. More details of four techniques which generate a good addition sequence can be found in [BC89].

Based on the analysis carried out by Koc in [Koc95a], the r-ary method, CLNW and VLNW (the Bos and Coster algorithm) require 636, 607 and 595 multiplications respectively on average for a random 512-bit exponent and k=5. These numbers for an 1024-bit exponent are 1247, 1195 and 1176 respectively (k=6). This represents about a 6% improvement in performance over using the VLNW algorithm. Those multiplications required to establish the pre-computed table are not included.

There are other variants of the CLNW method in which the rewriting of non-zero and zero words starts from the most significant bit [HL94] or from least significant bit [Yac90]. These approaches with odd powers and separating nonzero strings are also called the sliding window method [Koc94] [MOV97].

Yacobi in [Yac90] presented the algorithm in which the binary representation of the exponent is rewritten as odd digits from right-to-left (LSB first). Yacobi proposed to precompute only subexponents that would be used in the exponentiation process.

In the Hui and Lam algorithm [HL94] [LH94] [Yen95], the exponent is rewritten as odd digits from left-to-right (MSB first). They proved that the average number of multiplications in exponentiations tends to n/(r+1) for large n.

In [PB96] a simulation of 5000 random exponents in exponentiation with 1024-bit length shows the slightly better performance for the Yacobi algorithm by 1161 multiplications versus 1170 on average for the Hui and Lam algorithm. Although the latter algorithm requires fewer multiplications and squarings for precomputation (31 multiplications and 1 squaring versus 68 multiplications and 8.5 squaring for the former algorithm on average).

3.2.4 Recoding Methods

Another alternative to reduce the number of multiplications in the exponentiation process is the recoding method. In this method the binary representation of the exponent is replaced by representations which have fewer non-zero terms. The property of this class is that it requires the inverse of M modulo N (or M^{-1}) in order to efficiently compute M^e **mod** N. After recoding, the exponent is expressed in the redundant signed-digit representation using the digits $\{\bar{1},0,1\}$. Then the binary method (or r-ary methods) can be employed to compute M^e **mod** N provided that M^{-1} is supplied along with M. (In here the recoding class is limited to transforming only signed-digit representation so it does not include the r-ary methods.)

In general the signed-digit representation for an integer is not unique. Booth's recoding and its modifications are instances of the algorithms which can be used for this recoding.

Booth's Recoding: The original recoding technique expressed by Booth [Boo59] uses the following property:

$$2^{i+j-1} + 2^{i+j-2} + \ldots + 2^i = 2^{i+j} - 2^i$$

to replace a block of 1s by a single 1 and a single $\bar{1}$ where $\bar{1} = -1$. In Booth's algorithm [Boo59], the binary number e is scanned from right to left and is replaced by the recoded number D using the following table:

e_i	e_{i-1}	d_i
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

where $e_{-1} = 0$. For example $47 = (01111)_2$ can be recoded as:

$$01111 = 2^3 + 2^2 + 2^1 + 2^0$$

$$1000\bar{1} = 2^4 - 2^1$$

So the exponentiation of M^{15} can be performed using

$$M^{15} = (((M)^2)^2)^2.M^{-1}$$

Booth's algorithm does not always produce fewer non-zero terms than the binary representation. In addition its result does not guarantee the minimum number of non-zero terms. Several modifications can be found in the literature such as [WF82] [Hwa79] [Kor93] [Omo94] to solve these problems for use in exponentiation.

Sparse Signed-digit Representations: If no two non-zero entries are adjacent in a signed-digit representation of an integer e, it is said to be sparse [MOV97]. It is also called a canonical signed-digit vector [Koc94]. Every integer e has a unique sparse signed digit representation which has the smallest number of non-zero terms. The canonical recoding algorithm produces a sparse signed-digit representation for e, if the binary expansion of e is viewed as padded with an initial zero. This algorithm can be expressed as follows:

The Canonical Recoding Algorithm

```
$c_0$ := 0;
for i = 0 to n-1 loop
    $c_{i+1}$ := $\lfloor (e_i + e_{i+1} + c_i)/2 \rfloor$;
    $d_i$ := $e_i + c_i - 2c_{i+1}$;
return D = $(d_{n-1} d_{n-2} \ldots d_0)$;
```

This algorithm is due to Reitwienser [Rei60] [MOV97] and is a modification of Booth's recoding. As an example, this algorithm recodes e = $(1101110111)_2$ to $(100\bar{1}000\bar{1}00\bar{1})_{SD}$.

Therefore, the sparse signed digit representation of an exponent can guarantee the smallest number of multiplications, but only for the binary exponentiation algorithm. This result is not correct if r-ary methods are used. Egecioglu and Koc in [EK94] showed that although the canonically recoded r-ary method for constant r requires fewer multiplications than the standard r-ary method, however choosing an optimal r for each method for a given n, the average number of multiplications required by the r-ary (standard) method is fewer than those required in the recoded version. More details about the comparisons between different combination of r-ary algorithms and recoding algorithms can be found in [Koc94] [Koc95a] and [PB96].

3.2.5 The Exponent Decomposition Methods

There are exponentiation methods in which the exponent is considered to be divided into two or more parts. They are classified here as the exponent decomposition methods.

Factor method: The factor method is due to Knuth [Knu98][1] [Koc94] and based on factorisation of an exponent as: $e = d.f$ where d is the smallest prime factor of e. The exponentiation is then computed as $(M^d)^f$. Since the exponent can be a large number and factorisation of a large number is not easy, this technique is not appropriate for cryptography applications.

BGMW method: This method is due to Brickel et al [BGMW92]. In this method, the exponent e is represented using a basic digit set for base b. The basic digit set is a set of integers D if any integer can be represented in base b using digits from the set D. Therefore, the exponent can be rewritten as $e = \sum_{i=0}^{n'-1} d_i.b^i$ where $d_i = m_i.k_i$. Then M^e can be computed by

$$M^e = \prod_{i=0}^{n'-1} M^{d_i.b^i} = \prod_{i=0}^{n'-1} M^{m_i.k_i.b^i} = \prod_{i=0}^{n'-1} (M^{m_i.b^i})^{k_i}$$

So $P_i = M^{m_i.b^i}$ can be precomputed to be used later in exponentiation. This method is only useful when the base M in exponentiation is fixed such as in the El-Gamal algorithm. However, this method requires considerable storage and precomputations.

Lim and Lam method: Lim and Lam [LL94] observed that if an exponent is divided into two equal blocks as $e = e_h.2^{n/2} + e_l$ and if $M_1 = M^{(2^{n/2})}$ is precomputed, then M^e can be evaluated by $M_1^{e_h}.M^{e_l}$

[1] The first edition published in 1969

in half of the time required by the binary method . This method is very useful when the base M is fixed.

Division chains: Walter [Wal98] proposed the iterative application of a decomposition $e = d.e'+r$ where r is usually the least non-negative residue of e modulo d. He showed that this method requires 5n/4 multiplications.

CRT (Chinese remainder theorem) method: When the modulus N satisfies $N = p.q$ where p and q are the prime integers, the Chinese remainder theorem can be used for modular exponentiation. Quisquater and Couveur [QC82] show how this method can speed up the modular exponentiation for the RSA algorithm decryption (not for encryption) where the knowledge of the N factorisation is available. This method does not perform the exponentiation C^d for the decryption, rather it first computes the two following modular exponentiations as:

$$C_p = C^{d}{}_{p} \textbf{ mod } p \qquad \text{where } d_p = d \textbf{ mod } p\text{-}1 \qquad (3\text{-}1)$$

$$C_q = C^{d}{}_{q} \textbf{ mod } q \qquad \text{where } d_q = d \textbf{ mod } q\text{-}1 \qquad (3\text{-}2)$$

It then completes the computation by the following non-exponentiation operations as:

$$M = (((C_p - C_q).q^{-1}) \textbf{ mod } p).q + C_q$$

The exponents in (3-1) and (3-2) are shorter than d (they can be assumed as a half of the d size), therefore this method reduces the number of bit operations for performing the decryption exponentiation.

This method is a special case of a general solution for the Chinese remainder theorem. It is due to Garner [Gar59] [MOV97] and called as the mixed-radix conversion (MRC) algorithm [Koc94]. Quisquater and Couveur proposed using d_p and d_q instead of d after applying the CRT in order to divide the computation of C^{dp} **mod** N into C^{dp} **mod** p and C^{d} **mod** q. There is another general solution referred to as the single-radix conversion (SRC) algorithm [Koc94] which is due to Gauss [MOV97].

The SRC algorithm requires more operations compared with the MRC algorithm. A detailed discussion of these algorithms is given by Knuth [Knu98] and Cohen [Coh93].

3.2.6 Some Considerations

Table 3.2 shows the performance analysis results of the different methods described so far from [EK94] [Koc94] [Koc95] [Koc95a] [PB96] [Wal98].

Table 3-1: The resource requirements and the performance of the exponentiation methods

Exp. Alg. Name	Exponent Preprocessing	Pre-computations	Extra Memory	No. of operations
Binary	-	-	Very Low	768
r-ary (fixed window)	Low	Low	Medium	636
CLNW	Medium	High	High	607
VLNW	High	High	High	595
Yacobi	High	High	High	< 597
Hui & Lim	High	High	High	597
BGMW	Very High*	Very High	Very High	< 512
Lam & Lim	Very High*	Very High	Very High	< 512
Division chains	Very High	High	Low	640
The canonical recoding	Low	Medium	Low	683
recoding & r-ary	Medium	High	Medium	636
recoding & adaptive r-ary	High	High	High	< 597

* Base Fixed Exponentiation

The performance is expressed only in terms of the average number of the required multiplications (including squarings) to perform an exponentiation on 512-bit and 1024-bit exponents. This comparison results in the following conclusions:

1. Those methods which have a better performance on average (Bos & Coster, Yacobi, Hui& Yam) are quite comparable.
2. Except the binary method and the r-ary method, others require preprocessing on the exponents.
3. Except the binary and the division chains others require precomputations.
4. Except the binary and the division chains others require considerable storage to keep precomputation values.
5. Compared with the binary method, the maximum improvement is 25%.

The non-binary methods need extra hardware to provide extra memory and extra computations for exponent processing and precomputations. Moreover the extra computations add the time required for processing each block of data. These requirements result with at most a 25% increase in speed are not very attractive in hardware implementations. For this reason the binary algorithms are usually used for exponentiation in hardware. In here the left-to-right algorithm was chosen for exponentiation. However, the increase in speed using non-binary algorithm can be compared with other improvements that will be introduced later to evaluate them.

Since the memory which can be developed on a chip with a reasonable area is increasing every year, these techniques may become more attractive in increasing the speed of hardware implementation of exponentiation in the not too distant future.

3.3 Modular Multiplication

Modular exponentiation can be performed by implementing a modular multiplication routine which can accomplish a modular squaring as well. In general a modular multiplication such as S = A.B **mod** N consists of two operations: multiplication and modular reduction. So there are two approaches to compute a modular multiplication as follows:

1. First multiplication is completed and then modular reduction are performed.
2. Interleaving of multiplication and modular reduction steps.

There are only a few instances of using the first approach in cryptography applications because of the great increase in the operands' size. The interleaving approach is commonly used and is more suitable as it avoids increasing the size. The operations in this approach may be performed in either binary (low radix level) or in high radix level.

Multiplication: The conventional algorithm for multiplication is “the add-shift algorithm” which is the same method as the hand-paper method. In this algorithm, the multiplier bits are scanned one digit per step, and then this digit is multiplied by the multiplicand followed by adding into the partial product. To form the final result, the partial product is shifted each step. The multiplication procedure usually starts by scanning the MSB (or MSDigit) first for modular reduction of Euclid’s division type and the LSB (or LSDigit) first for modular reduction of Hensel’s division type as described later in this chapter.

Modular reduction (MR) and division: As has been explained, the modular reduction on large numbers is one of the basic operations in public-key cryptography. Generally this operation can be perfomed using division. Although there are conventional algorithms for the hardware implementation of division (such as the restoring and the non-restoring algorithms [Omo94]) other techniques have been developed to deal with large operands in cryptography applications. These techniques result in a high performance implementation of cryptosystems.

However, a simple description for the modular multiplication A.B **mod** M for A = $\{a_{n-1}.r^{n-1}+a_{n-2}.r^{n-2}+...+a_0\}$ can be given by the following algorithm:

A General Algorithm for Modular Multiplication:

```
      S(0) := 0;
            for i = n-1 downto 0 loop
L1:               T := r.S(i) + $a_i$.B;
L2:               S(i+1) := T mod M;
      return S(n);
```

where S(n) = A.B **mod** M. In each iteration, a partial multiplication is performed in L1 followed by a modular reduction in L2.

For a better description of the techniques developed for modular reduction, two classifications are introduced: (i) Euclid's division versus Hensel's division; (ii) multiprecision techniques versus binary techniques. These classifications are only to facilitate providing a clear description of the reduction methods and they are just used here in the context of this chapter.

Division types: Euclid's division and Hensel's division

The conventional methods for division are classified by Shand and Vuillemin in [SV93] as Euclid's division versus Hensel's division.

Euclid's division: To divide $P = \{p_{(n+l)-1}p_{(n+l)-2}\dots p_0\}$ by N, this class accomplishes division based on performing the following:

$$P = Q.N + S(n)$$

where Q is quotient and $S(n) < N$. In this class the following recursive procedure is executed:

$$S(i + 1) = (r.S(i) + p_{l-i}) - q_i .N$$

i is the recursion index, q is quotient digit, r is radix, $S(0) = \{p_{(n+l)-1}p_{(n+l)-2}\dots p_{l+1}\}$ is the high-order part of the dividend P (initial partial remainder) and $S(n)$ is the final remainder. This class scans the dividend digits left-to-right (MSDigits first). Moreover the partial result is shifted to the left each iteration.

Hensel's division: Hensel introduced a division class in which division is based on performing the following [SV93]:

$$R.P = Q.N + S(n)$$

where Q is quotient, R is r^n and $S(n) < 2N$. The procedure which is recursively executed in this class is as:

$$S(i + 1) = ((S(i) + s_{n+i}) + q_i .N) \textbf{ div } r$$

where $S(0) = \{p_{n-1}p_{n-2}... p_0\}$ is the low-order part of the dividend (initial partial remainder) and S(n) is the final remainder. The Montgomery reduction method is the important member of this class for cryptographic applications. The dividend digits are scanned in this class right-to-left (LSDigits first). Moreover, the partial result is shifted to the right each iteration. A description of the binary algorithms for these two classes is given in [SV93].

Binary techniques versus Multiprecision techniques

Multiprecision techniques: There are techniques developed for modular reduction which are based on multiprecision division techniques. They are originally more suitable for software implementation, but they have been modified to fit hardware implementation as well.

Binary techniques: In this chapter, the binary class is used to refer to those techniques in which bit level operations perform modular reduction. This class is more suitable for hardware implementation than the “opposite” class, multiprecision techniques. A high radix version of the binary techniques in this class is taken into account as a generalisation of the binary technique. So they are still classified as binary techniques in this thesis.

3.4 MR Methods Based on Euclid's Division

This section and section 3.5 reviews modular reduction (MR) methods using the classifications described in 3.3. Since this review is aimed at describing different modular multiplication methods, this operation is also described wherever it is required.

The reduction methods described here have originally been developed for modular multiplication (except the classical method which is the conventional method for multiprecision division). A general description for interleaving multiplication and modular reduction steps can be given by the following algorithm whose reduction method (MR) is based on Euclid's division:

Modular Multiplication Algorithm based on Euclid's division:

```
S(0) := 0;
for i = n-1 downto 0 loop
      q_i := (r.S(i) + a_i.B) div M;
      S(i+1) := (r.S(i) + a_i.B - q_i.M);
return S(n);
```

where S(n) = A.B **mod** M. In each iteration, first the quotient q_i is formed as the division result of the preliminary partial result by the modulus. Then the partial result can be formed by reducing a q_i multiple of modulus. The quotient calculation is one of the major bottlenecks in this class when the multiplier operand is not a simple bit. Multiprecision methods mainly deal with this problem while binary methods focused on the other problems in this class.

3.4.1 Multiprecision Methods

The classical method: This method is the conventional division method (multi-precision division) and is based on the estimate-and-correct technique [Knu98] in which the quotient is estimated by dividing two MSDigits of the partial dividend P' by the MSDigit of divisor M as: $q_{es} = \{p'_{n-1}p'_{n-2}\} / m_{n-1}$ where $q_i \leq q_{es} < q_i+2$. After correcting the estimated quotient, the partial dividend is subtracted by the q_i multiple of the modulus as: $S(i+1) = T - q_i.M$. This method is commonly used for (software) multiprecision division and is described in more detail in chapter 4 (sub-section 4.3.5).

Miyaguchi's method: Miyaguchi [Miy82] introduced an algorithm based on replacing division by multiplication in modular multiplication. In this method, the modulus' reciprocal is first precomputed as: $v = 2^u$ **div** (N **div** 2^p) where N **div** 2^p is the high-order part of modulus and 2^u provides enough accuracy for this computation. Then the high-order digits of the partial dividend T' is used to calculate quotient as: $q_i = T'.v.2^{-u} + w$ where w is 0 or 1. Finally $P(i+1) = T' - q_i.N$. Using this technique no division is performed, rather a multiplication by the modulus' reciprocal is accomplished. Modular multiplication is performed on base r and it is interleaved by modular reduction. The Miyaguchi method is a serial-parallel method and was one of the first quotient estimation methods proposed for implementing the RSA algorithm.

Barret's Method: Barret [Bar86] [BGV93] used the similar approach as Miyaguchi used in the precomputation of the modulus reciprocal. This is performed by: $\mu = r^{2n}$ **div** M. This precomputation in Barret's method (and also in Miyaguchi's method) is advantageous if many modular reductions are performed with a single modulus such as in the RSA algorithm. Then the quotient is estimated as q_i = ((T **div** r^{n-1}). μ) **div** r^{n+1} where T **div** r^{n-1} is the MSDigits of partial dividend. This quotient estimation at most requires to be reduced by 2. Finally, the remainder S is computed as S = T **mod** r^{n+1} - q.M **mod** r^{n+1}. This algorithm is based on the observation that $q_i = \lfloor T / M \rfloor$ and it can be also expressed as:

$$q_i = \lfloor (T / r^{n-1})(r^{2n} / M)(1 / r^{n+1}) \rfloor$$

The relaxed residuum method: This method was developed by G. Puche and T. A. Puche in [PP90] and they denoted it as the relaxed residuum method. This method calculates the estimated quotient by $q_i = T. \overline{M}$ where $\overline{M}$ is a modulus' reciprocal precomputed as $\overline{M} = 2^{2n+4}/M$ where $(T/M)-1-\varepsilon < q_i \leq (T/M)$. The maximum error ε is ignored, and the relaxed residuum is computed as $P = T - q_i.M$ where $0 \leq q_i < M.\varepsilon$ [PP90]. The complete modular reduction is performed only after finishing the modular exponentiation process. This method gives a solution for correcting the estimated quotient at a cost of extra length of operands. The high speed single-chip implementation reported in [Poc98] is based on this method.

3.4.2 Binary Methods

Brickel's method: Brickel developed this method in 1982 to speed up modular multiplication [Bri82]. The modular reduction in Brickel's method is started 11 iterations after starting the multiplication process, so the accumulator has grown to n+11 bits. In Brickel's method, first the 2's complement of the modulus, $\overline{M}'$, is precalculated, then the partial result is subtracted by $2^{11}.\overline{M}'$ or $2^{10}.\overline{M}'$ based on the comparison result of high-order 4 bits of the partial result and high-order 4 or 3 bits of $2^{11}.\overline{M}'$ or $2^{10}.\overline{M}'$ respectively. Brickel used carry delay adders (described in section 3.6) for the partial result. The final result of modular multiplication is saved in the multiplier in carry delay adder format for the next iteration. So carry propagation is not required between successive modular multiplications in the exponentiation process. In this method two required subtractions (as will be discussed later) are reduced to a single addition. The comparison is performed on a shorter sized number and uses only a 4-bit comparator. Brickel's method using these significant advantages achieves a modular multiplication in n plus some overhead iterations (in here n+11 iterations totally). More details are given by Walter and Eldridge [WE90] about this method.

Blakley's method: In each iteration of modular multiplication, the previous partial result is shifted to the left a single bit and added to the partial product $a_i.B$ to perform the multiplication A.B. So the current partial result satisfies the followings before performing modular reduction:

$$T = 2P(i) + a_i.B \leq 2(M-1) + (M-1) = 3M-3$$

where A, B and P(i) are less than M. This means at most two subtractions by M are required to perform the modular reduction T **mod** M. Blackley's method [Bla83] performs operations required in each iteration (adding $a_i.B$ followed by a modular reduction) as:

```
if M>P(i) then T := P(i)+B else T := P(i)-(M-B);
if M>T    then P(i+1) := 2T else P(i+1) := 2(T-M);
```

Therefore, only an addition or at most two subtractions are required for each iteration in this method. Tomlinson [Tom93] implemented a modification of this method and achieved a 10 kbps data rate.

Omura's method: Omura [Omu90] described a method in which the carry-out of the n-bit partial result is considered. When a carry-out is detected, it is ignored and a correction is then performed. This means the partial result is allowed to grow larger than M, but it is kept less than 2^n. The correction factor is $\overline{M}' = 2^k - M$ which is added into the partial result. Since ignoring the carry-out means subtracting by 2^k, then the correctness of this method can be shown as follows:

$$P = T - 2^k + \overline{M}' = T - 2^k - (2^k - M) = T - M$$

3.5 MR Methods Based on Hensel's Division

Since Montgomery's method is used for implementing the RSA algorithm in this thesis, modifications to this method are examined in more detail in the following sub-sections.

3.5.1 The Montgomery Reduction Algorithm

In 1985 Montgomery presented a method for modular reduction without division by modulus M [Mon85]. In fact the algorithm performs the function $T.R^{-1}$ **mod** M (instead of T **mod** M) where R is a radix (preferably a power of 2) and M and R are relatively prime integers. The following algorithm can perform the modular reduction $T.R^{-1}$ **mod** M where $R.R^{-1} - M.M' = 1$.

The Montgomery Reduction Method

```
M' := $-M^{-1}$ mod R
Begin
    Q := (T mod R).M' mod R;
    S := (T + Q.M)/R;
    if  S $\geq$ M  then  return  S - M  else return  S;
End
```

The basic idea in Montgomery's algorithm is to make T a multiple of R by adding multiples of M. To validate this method, notice that

Q.M $\equiv$ (T.M').M **mod** R

$\equiv -T$ **mod** R

So $(T + Q.M)$ is divisible by R because $T + Q.M = T + (-T$ **mod** $R)$ and S is an integer. Then:

$S.R \equiv (T + Q.M)$ **mod** M

$\equiv T$ **mod** M

So: $S \equiv T.R^{-1}$ **mod** M

S also satisfies in $0 < S < 2M$ because $0 \leq T + Q.M < R.M + R.M$ and this results in $0 < t = (T + Q.M)/R < 2M)$.

This description of the Montgomery's method can be used for software implementation and a detailed discussion about its development are given in [KAK96]. However, the Montgomery method has the disadvantages of pre- and post-processing requirements. The term $-M^{-1}$ needs to be precomputed and the final result is required to be converted from TR^{-1} **mod** M to T **mod** M.

A description about the binary method was given by Montgomery for hardware implementation of the modular multiplication A.B **mod** M in [Mon85] and it can be written as the following algorithm:

The Binary version of Montgomery's Modular Multiplication

```
S(0) := 0;
Begin
    for i = 0 to n-1 loop
        $q_i$ := (S(i) + $a_i$.B) mod 2;
        S(i+1) := (S(i) + $a_i$.B + $q_i$.M) / 2;
    if S(n) ≥ M then
        return S(n) - M else return S;
End
```

This method has been developed in many research efforts as discussed next.

3.5.2 The Dusse and Kaliski Method

Dusse and Kaliski [DK90] presented a series of improvements on Montgomery's algorithm to make it suitable for implementing on DSProcessors. In their method the quotient Q is not computed in one operation, rather one digit of Q as q_i is computed at a time. It is followed by adding $q_i.M.r^i$ to T. This procedure is repeated for n iterations where n is the number of modulus digits. The result (T) may not be same as in the original algorithm, but it keeps the idea of adding multiples of M to make T a multiple of R. In this approach the term M^{-1} **mod** r needs to be computed instead of M'. So only the least significant digit of M^{-1} is required. This method for modular multiplication can be rewritten as the following algorithm:

A, B and M are three integers, each presented by n digits in radix $r = 2^k$. The modulus M is relatively prime to r. This algorithm will return the result of $A.B.R^{-1}$ **mod** M:

The Dusse and Kaliski Method for Modular Multiplication

```
Begin
      $m_0' := -m_0^{-1}$ mod r;
      $S(0) := 0$;
      for i:= 0   to   n-1 loop
            $S := S(i) + a_i.B.r^i$;
            $q_i := s_i.m_0'$  mod r;
            $S(i+1) := S + q_i.M.r^i$;
      $Sn := S(n') $ div R;
      if    $Sn \geq M$     then return  $Sn - M$
                          else return  Sn;
End
```

Other implementations of Montgomery's algorithm use this idea of Dusse and Kaliski.

3.5.3 The Shand and Vuillemin Method

The implementation of Shand and Vuillemin [SV93] used a generalisation of Montgomery's algorithm for the high radix $r = 2^p$ (or p-bit scan) [SV 93]. The higher radix can be used in the algorithm as follows:

The Shand and Vuillemin Method for MR:

```
$m_0' := -m_0^{-1}$  mod r;
$S(0) := 0$;
```

```
Begin
for i := 0   to   n-1  loop
      $q_i$ := $m_0$'($a_i$.$b_0$ + $s_0$(i)) mod r;
      S(i+1) := ($a_i$.B + S(i) + $q_i$.M) div r;
return S(n);
End
```

Finally $S.r^n = A.B + M.Q$ where $Q = [q_{n-1} \ldots q_0]$ and S = S(n). The conditional term at the end of the original algorithm returns S which is less than M ($0 < S < M$), but Shand and Vuillemin moved this conditional term to the end of modular exponentiation and proved the modular multiplication result would be less than 2M by adding two extra bits. Iwamura *et al* in [IWI92a] used two extra iterations to keep the result bounded to 2M and to remove conditional subtraction.

In this algorithm the quotient is obtained from the least significant digits of A and S(i) which can further simplify the algorithm. Shand and Vuillemin also showed that it was possible to pipeline the quotient calculation in each step.

3.5.4 The Eldridge and Walter Method

Eldridge and Walter [EW93] introduced a modification of Montgomery's algorithm at the same time as Shand and Vuillemin. They simplified one operation which is required in the computation of the quotient. They used $r^2.B$ in place of B (i.e. they shifted B two digits to the left) so that q is independent of it. In their implementation they considered performing each iteration in one clock cycle. So $q_i.M$ should be ready in this cycle to calculate ($a_i.B + S(i) + q_i.M$) in the next. Because q_i needs the least significant digit of S(i) which is computed in the previous iteration, they changed the order of computation for these two terms in the algorithm as follows:

The Eldridge and Walter Method

```
$m_0' = (r - m_0)^{-1}$;
```

```
S(0) := 0; $q_i$ := 0;
Begin
     for i := 0  to  n+1  loop
          S(i+1) := ($a_i$.$r^2$.B + S(i) + $q_i$.M)  div r;
          $q_{i+1}$ := [($s_0$(i+1).$m_0'$]  mod r;
     if  S(n) ≥ M then
          return  S(n) - M  else return S(n);
End
```

The output here satisfies: $r^{n+2}.S(n) = A.r^2.B$ **mod** M. Since the least significant digit of r^2B is zero, computing q_i is independent of B.

Eldridge and Walter proposed using the delay adder circuit [Bri82] with some modifications. Walter has also suggested a method for implementing Montgomery's algorithm based on systolic arrays, but this solution needs a large amount combinatorial logic [Wal93].

3.5.5 Orup’s Method

Regarding these developments for performing modular multiplication, Orup [Oru95] noticed that another operation can be simplified in the quotient calculation using a technique which was previously introduced by Walter in [Wal91]. In his method, the modulus scaled as: M" = (-M^{-1} **mod** r).M and is precomputed once. His algorithm is as follows:

The Orup Method

```
     S(0) := 0;
     Begin
     for i := 0  to  n-1  loop
L1:       $q_i$ := $s_0$(i);
L2:       S(i+1) := (S(i) + $q_i$.M") div r + $a_i$.B;
     return S(n);
     End
```

where the quotient q_i is the LSDigit of the partial result S(i). In comparison with other algorithm, the addition that is needed in the quotient computation is transferred to L2 (in the computation of S(i+1)). He also suggested an implementation which includes two types of pipelining: a 3 level pipelined adder and a pipelined method for quotient computation.

3.5.6 Walter's Method

Walter in [Wal95] proposed two modifications in order to simplify the quotient calculation. Firstly, the multiplicand B is replaced by $2^k.B$. Secondly, the modulus M is scaled as $M'' = m_k^{-1}.M$ where $m_k^{-1}.M \equiv 1$ **mod** 2^k. Alternatively, the modulus can be chosen to satisfy $M \equiv 1$ **mod** 2^k. In the iteration i partial result computation is as:

$S(i+1) := (S(i) + q_i.M'' + a_i.B')$ **div** r;

Then the low-order k bits of S(i+1) are determined by shifting down the low-order k+1 bits of S(i). Thus the digit q_i is produced k-1 iterations before the iteration in which it is used as the quotient. This gives the implementation k-1 iterations time to complete the $(S(i)+q_i.M'' + a_i.B)$ calculation.

3.6 The Major Block For Implementation

Adders are the basic block for implementing modular multiplication and have a major role in speed and area of the implementation. The next section briefly reviews some of the adders used in the modular multiplication implementations of cryptographic applications.

Carry propagation adders: A carry propagation adder consists of n parallel full adders. n is the number of operand bits, and the carry out of one full adder is connected to the carry input of the next stage full adder. The speed of the adder is determined by the time which it takes to propagate the carry from the least significant bit to the most significant bit. The delay time will increase when the number of bits increases.

Carry look-ahead adder: In this method, the carries of a parallel adder are generated simultaneously by additional logic circuitry. This results in a constant addition time independent of the length of the mid-size adder. By denoting $P_i = A_i \oplus B_i$ as the propagation term and $G_i = A_i.B_i$ as the generation term ($\oplus$ refer to XOR function), the addition results S_i and C_i (sum and carry) can be expandeded in terms of P and G as:

$$C_i = G_{i-1} + G_{i-2}.P_{i-2} + ... + G_0.P_0....P_{i-1}$$

A carry look-ahead adder directly computes all C_is in advance using generation and propagation terms based on the above description of C_i. Then S_i can be obtained as:

$$S_i = P_i \oplus C_i$$

Carry save adder: The carry save adder was developed by Robertson, Rohatsch and Wallace [Hwa79] [OPT93] and is commonly used for implementing multiplication to avoid carry propagation during partial product generation.

This adder has 3 inputs and two outputs, sum and carry, as follows:

$$S_i := A_i \oplus B_i \oplus C_i$$

$$C_{i+1} := A_i.B_i \vee A_i.C_i \vee B_i.C_i$$

These are full adder functions to produce sum and carry. The outputs of carry save adder, sum and carry, are separately saved for the next iteration. Therefore, the carry propagation time can be removed during the multiplication process. Only at the end of this process the carry needs to be propagated using a carry propagation adder or a carry look-ahead adder.

Carry delay adder: The carry delay adder was developed by Norris and Simmons in 1981 [NS81] [Bri82]. It consists of a carry save adder followed by a half adder stage. The CSAdder outputs, S and C, are two inputs for the half adder stage. So the carry delay outputs in general can be described in bit level as:

$u_i \quad := s_i \oplus c_i$

$v_{i+1} := s_i.c_i$

where $v_{i+1}.u_i = 0$ as the main feature of the carry delay adder ($v_0 = 0$). When an accumulator is saved using this format as $a_i = u_i + 2v_{i+1}$, its partial product can be described as: $a_i.B = u_i.B + v_{i+1}.(2B)$. Since one of u_i or v_{i+1} is zero, $a_i.B$ is either B or 2B respectively. Therefore, the modular multiplication result in a modular exponentiation is not required to propagate the carry, rather the carry delay adder is used and the result in this format is saved for the next modular multiplication iteration.

3.7 Some Considerations

Montgomery's method and its derivatives have many benefits over conventional algorithms (classified in this thesis as Euclid's division) such as:

1. It does not need to divide by the modulus.
2. No comparison is required.
3. No subtraction by the modulus is needed during the iterations.
4. The order of treating the digits in multipliers is from low order (LSB) to high order (rather than high to low in conventional methods).

In the conventional modular multiplication methods, besides the quotient estimation, comparison with the modulus and subtraction from it are necessary. Montgomery's methods and its variants do not need these operations. This facilitates the computations and increases the implementation performance considerably. The cost is two extra modular multiplications for pre- and post-computation.

As a result of these advantages, the Montgomery method was chosen for implementing modular multiplication in the RSA algorithm.

6.2.1 Modular Exponentiation Unit

In order to compute the modular exponentiation M^e **mod** N, the left-to-right square-multiply algorithm is modified to use the Montgomery modular multiplication method. The result is suitable for hardware implementation and can also handle arbitrary sizes of the modulus N and the exponent E up to n bit. The base M is less than the modulus N. This algorithm (after modification) is as follows and referred to here as the LME algorithm (the Left-to-right Modular Exponentiation algorithm):

The LME algorithm

```
    input: M, E, N, R2
    output: C = M^E mod N
    C := 1;
    k := n;
    while e_{k-1} = 0 do k := k-1;
    if k = 0 then goto L3 else
L1:     M' := M4(R2,M,N);
        C := M';
        if k < 2 then goto L2 else
            for i = k-2 downto 0 loop
                C := M4(C,C,N);
                if e_i = 1   then   C := M4(C,M',N);
L2:     C := M4(C,1,N);
        if C ≥ N then C := C - N;
L3: return C;
```

where the exponent is E = {$e_{n-1}e_{n-2}...e_1e_0$} and R2 is the precalculated number equal to R^2 **mod** N. The radix R satisfies: $R = 2^n$. The internal routine of modular exponentiation (the **for** loop in the algorithm) was explained in the previous chapters so only the other parts of the LME algorithm are discussed here.

The LME algorithm first scans the exponent bits left to right to find the most significant bit (MSB) of the exponent E. The while statement in this algorithm is for implementing this function and returns the variable k as the exponent size. If there is no non-zero bit, the LME algorithm returns 1 which corresponds to M^0. Otherwise this algorithm performs the left-to-right exponentiation routine. When the first 1 (the MSB of the exponent) is detected, the algorithm ignores the first square and multiplication as these correspond to squaring one and multiplication of M by one. So the number of iterations in the exponentiation routine is reduced to k-1 for a k-bit exponent. Moreover, if the exponent has only a single bit, k = 1, the algorithm does not perform the internal modular multiplication loop. This is summarised in table 6-1 for different exponent values: 0, 1 and more in the LME algorithm:

Table 6-1: The exponent size and the loop index in the LME algorithm for different values of the exponent E

The exponent size k	e(0)	The Exponent E	The No of iterations in the Exp loop	The loop index i	M^E mod N
0	0	0	0	0	1
1	1	1	0	0	M
≥ 2	X	≥ 2	k-1	k-2	≥ 0, N $\geq$

Since modular multiplication in the LME algorithm is performed using the M4 algorithm as: $M4(X,Y,N) = X.Y.R^{-1}$ **mod** N and it is based on Montgomery's method, it requires pre- and post-computations. The M4 algorithm has been implemented in the modular multiplication unit as discussed later. The pre- and post- computations are accomplished in the lines labelled L1 and L2 in the algorithm respectively. The pre-calculation converts M to M.R using M4(R2,M,N). The LME algorithm then initialises the intermediate result C by this new value. When C^2 is calculated, the result still includes the R coefficient as:

$$M4(C,C,N) = C.C.R^{-1} \textbf{ mod } N = (M.R).(M.R).R^{-1} \textbf{ mod } N = M^2.R \textbf{ mod } N$$

By replacing M by M’ = M.R, the result includes the R coefficient in the rest of exponentiation routine.

The post-calculation in L2 removes the R coefficient using a multiplication by 1. Before exiting the routine the final comparison and subtraction is performed to keep the result less than the modulus as described in section 5.3.

procedure Montgomery_product (x, y, m: in bit_vector
(0..n 2 1); z: out bit_vector (0..n 2 1));
and that the value of
exp 2n ¼ 22:n mod m (8:6)
has been previously computed. Then z ¼ x.y mod m can be computed as follows:
z ¼ (x:y:2
_n):22:n:2
_n mod m ¼ (x:y:2
_n):(exp 2n):2
_n mod m:

Algorithm 8.11 Modular Product Based on the Montgomery Product
Montgomery_product (x, y, m, z1);
Montgomery_product (z1, exp_2n, m, z);

Example 8.3 n ¼ 8, m ¼ 239, x ¼ 217, y ¼ 189; in base 2, x ¼ 11011001;
exp_2n ¼ 216 mod 239 ¼ 50.
First compute the Montgomery product of x and y:
r(0) ¼ 0,
a ¼ r(0) þ x(0):y ¼ 189; r(1) ¼ (189 þ 239)=2 ¼ 214;
a ¼ r(1) þ x(1):y ¼ 214; r(2) ¼ 214=2 ¼ 107;
a ¼ r(2) þ x(2):y ¼ 107; r(3) ¼ (107 þ 239)=2 ¼ 173;
a ¼ r(3) þ x(3):y ¼ 173 þ 189 ¼ 362; r(4) ¼ 362=2 ¼ 181;
a ¼ r(4) þ x(4):y ¼ 181 þ 189 ¼ 370; r(5) ¼ 370=2 ¼ 185;
a ¼ r(5) þ x(5):y ¼ 185; r(6) ¼ (185 þ 239)=2 ¼ 212;
a ¼ r(6) þ x(6):y ¼ 212 þ 189 ¼ 401; r(7) ¼ (401 þ 239)=2 ¼ 320;
218 FINITE FIELD OPERATIONS
a ¼ r(7) þ x(7):y ¼ 320 þ 189 ¼ 509; r(8) ¼ (509 þ 239)=2 ¼ 374;
z1 ¼ 374 _ 239 ¼ 135;
in base 2 z1 ¼ 10000111;
then compute the Montgomery product of z1 and exp_2n:
r(0) ¼ 0,
a ¼ r(0) þ x(0):y ¼ 50; r(1) ¼ 50=2 ¼ 25;
a ¼ r(1) þ x(1):y ¼ 25 þ 50 ¼ 75; r(2) ¼ (75 þ 239)=2 ¼ 157;

a ¼ r(2) þ x(2):y ¼ 157 þ 50 ¼ 207; r(3) ¼ (207 þ 239)=2 ¼ 223;
a ¼ r(3) þ x(3):y ¼ 223; r(4) ¼ (223 þ 239)=2 ¼ 231;
a ¼ r(4) þ x(4):y ¼ 231; r(5) ¼ (231 þ 239)=2 ¼ 235;
a ¼ r(5) þ x(5):y ¼ 235; r(6) ¼ (235 þ 239)=2 ¼ 237;
a ¼ r(6) þ x(6):y ¼ 237; r(7) ¼ (237 þ 239)=2 ¼ 238;
a ¼ r(7) þ x(7):y ¼ 238 þ 50 ¼ 288; r(8) ¼ 288=2 ¼ 144;
z ¼ 144;
conclusion: 217 _ 189 mod 239 ¼ 144.

4.3.5 BitVector Division

Division is generally considered the most difficult of the 4 basic arithmetic operations. The BitVector division in the MATHIC package is based on the estimate-and-correct method. This method is described in the division algorithm in [Knu98] which it has been modified for use in the MATHIC package.

The general idea of the method is the estimation of quotient and correcting it later during the division procedure. The BitVector division function is 'divide' whose outputs are the quotient and remainder. Two other functions use the divide function to return quotient and remainder separately. The operator ‘/’ returns the quotient and the operator ‘%’ returns the remainder. The divide function includes the following steps: some initial tests, normalisation, main division loop, denormalisation. The main division loop consists of quotient digit estimation and first correction, the partial division loop, remainder test and final correction. The output of each iteration of the main loop is one quotient digit and one partial remainder. The single-precision division is only performed in the quotient estimation.

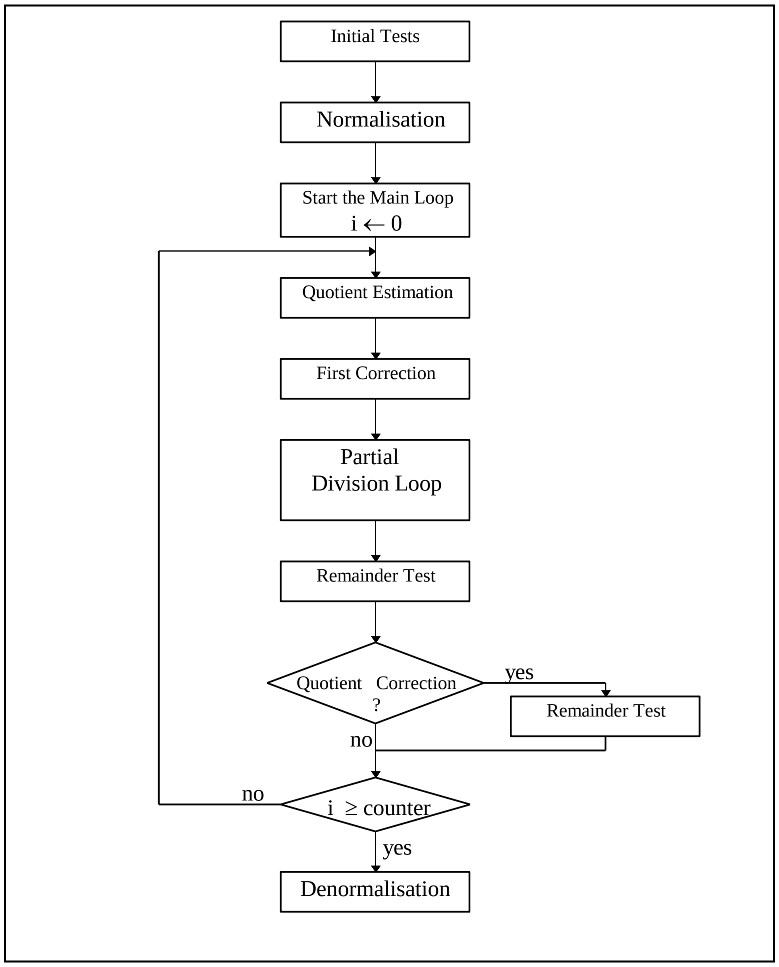

Figure 4-6: The BitVector division algorithm

1. Initial tests: This function first tests whether the divisor is nonzero. Then it examines if the divisor is less than the dividend. Finally, it checks whether the dividend and the divisor are 32-bit lengths.

2. Normalisation: The dividend and the divisor are left-shifted until the most significant digit of the divisor will be more than R/2 where R is the base of representation, in here $R = 2^{16}$. This guarantees the maximum error for the estimated quotient $q^{'}$ cannot be more than two ($q' - 2 \leq q \leq q'$; q: the precise quotient).

3. The main loop: The new dividend and the new divisor after normalisation are $D = \{d_{n+k-1}d_{n+k-2}.....d_0\}$ and $M = \{m_{n-1}m_{n-2}....m_0\}$ respectively. In each iteration, the algorithm uses $A = \{a_na_{n-}$

$_1$.....a_0} as the partial dividend made by the dividend digits and the partial remainder as described next. The following procedures are repeated in each iteration:

3.1. The quotient estimation and first correction procedure: If the MSDigit of the partial dividend and divisor are equal, then q' = r - 1. Otherwise, the quotient digit is estimated by single-precision dividing the two MSDigits of partial dividend $a_n a_{n-1}$ by the MSDigit of the divisor m_{n-1}. So $q' = \lfloor a_n.a_{n-1} / m_{n-1} \rfloor$. Each digit has a 16-bit length which is the same as the multiplication function. The estimated remainder r' is calculated by $r' = (a_n a_{n-1} - m_{n-1}.q')$ which is used only for correcting q'.

Figure 4-7 shows the quotient estimation and the first correction procedure. The first correction tests if $m_{n-1}.q' > r'. r + a_{n-2}$; if so, q' is decreased by one and the test is repeated. This test covers all cases in which q = q' - 1. It also covers most cases which q = q' - 2. Other cases are left for the remainder test and the final correction.

3.2. The partial division loop: The BitVector division or multi-precision division is done step by step using the estimated quotient in this loop. In each iteration, first q' is multiplied by the divisor digit, then the result is subtracted from the partial dividend. All operations are single-precision and all operands are 16-bit digits. The loop implements a multiplication of the divisor BitVector by a quotient digit and subtracts the product from the partial dividend BitVector. The output is a partial remainder whose digits are computed in each iteration.

3.3. Remainder test and final correction: If the partial remainder is negative, the quotient is decreased by one and the partial remainder modified.

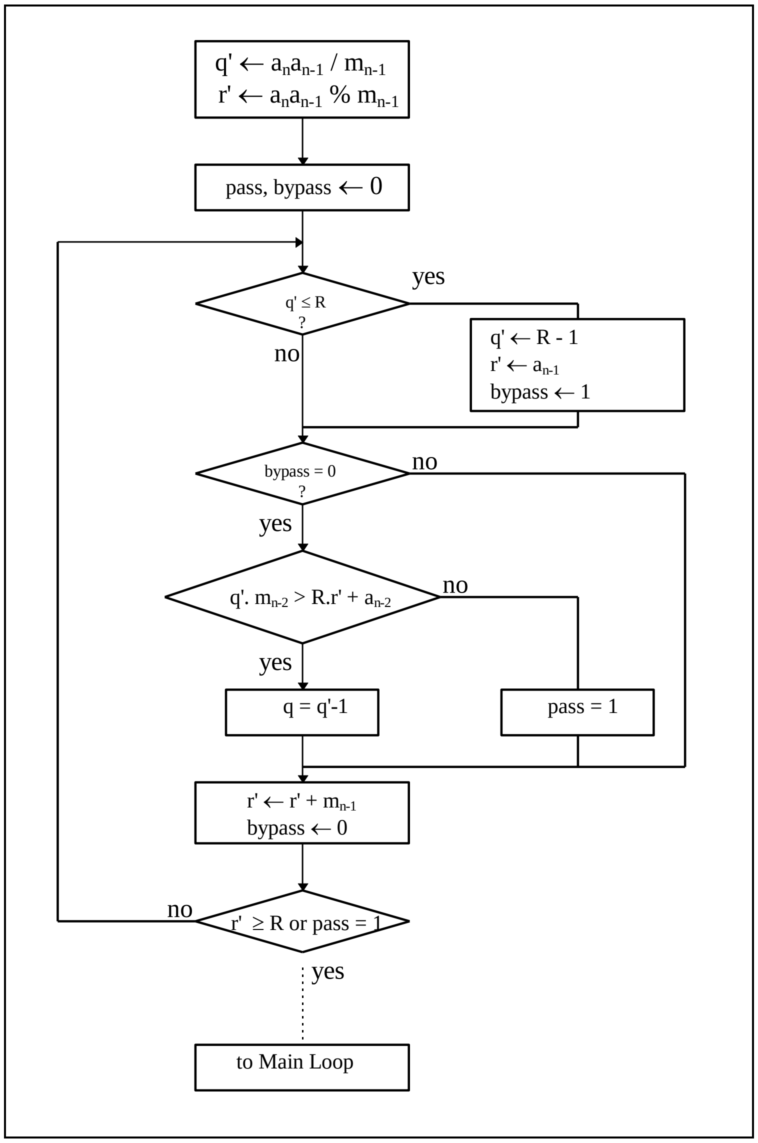

Figure 4-7: The quotient estimation and first correction algorithm

The outer loop moves the index variable through the dividend digits from left to right because the division operates on the most significant digits first. It shifts the partial remainder to the left and adds a new digit from the dividend to make a new partial dividend for the next iteration. In the first iteration, the partial dividend is the n most significant digits of the dividend where n is the number of digits in the divisor. This is extended with a leading zero. (So the partial dividend has n + 1 digits.) This leading 0 digit guarantees that the estimated quotient for the first iteration is less than r which is required for correctness of the quotient estimation [Knu98]. For other iterations, this condition is met automatically. Using the result of the inner loop iterations, the division of BitVector by BitVector is implemented in the main loop.

4. Denormalisation: The remainder needs to be shifted to the right as many bits as the operands have been shifted to the left in normalisation.

There are some modifications in the main loop steps as Figure 4-6 and 4-7 show. Referring to the divide function in Appendix II, first q’ and r’ are set, then the MSDigits are tested for equality. Some flags (pass, bypass, warning) are checked as conditional terms to finish the test. In the second modification to avoid negative results, first the product is compared with the partial dividend. If the product is less than the partial dividend the product is modified before the subtraction is carried out.