Chapter 1

Introduction.

1.1 Introduction

With the rapid inflation of internet and the growing popularity of electronic commerce, secure electronic transactions are becoming a major concern. It has become important to develop new way to guarantee their security. Two techniques are available: the Secret Key Cryptosystem such as DES (Data Encryption Standard), and the Public Key Cryptosystem such as RSA (Rivest, Shamir, and Adleman)[3].

RSA cryptosystem was invented in 1977; it is the best known and most widely used public-key cryptosystem today. The basic operation of RSA is modular exponentiation operations on extremely long bit streams, which take an immense amount of computation processing. During the recent years, Montgomery's modular multiplication was the most efficient method for faster implementations to the RSA. The quotient is dependent on the least significant digit of operands, no comparison used in the computing procedure, however which is the critical operation in traditional sequential division. Although this method requires pre-processing and post-processing to get the final correct result, but the number of modular multiplication operations are dependent on the number of bit in the exponent.

Several algorithms and hardware implementations of the Montgomery multiplication for a limited precision of the operands were proposed. In order to get improved performance, high-radix algorithms have also been proposed. However, these high-radix algorithms usually are more complex and consume significant amounts of chip area, and it is not so evident whether the complex circuits derived from them provide the desired speed increase. For this reason, low-radix designs are usually more attractive for hardware implementation.

In this thesis, we proposed a method based on Scalable Montgomery's algorithm to calculate modular multiplication, which allows us to investigate different areas of the design space, and thus, analyze the design tradeoffs for the best performance in a limited chip area. We then propose a modified word-based algorithm,and show the parallel evaluation of its steps in detail. By this analysis, we then show architecture for the modular multiplier and present the design of the module. Using TSMC 0.25μm process technology and Synopsys Design Analyzer, we simulate our new design and compared the performance with the original design.

We will describe the RSA algorithm in Chapter 2. Chapter 3 is the Scalable Montgomery Multiplication algorithm and our new hardware architecture. Chapter 4 shows the simulation result. Chapter 5 is the conclusion and future work.

1.2 Background

1.2.1Cryptography

Cryptography is the art of secret writing. The basic service provided by cryptography is the ability to send information between participants in a way that prevents others from reading it. Besides, cryptography can also provide other service such as:

- **Integrity checking**---reassuring the recipient of a message that the message has not been altered since it was generated by a legitimate source.
- **Authentication**---verifying someone's (or something's) identity.

- **Nonrepudiation**: It should be impossible for the sender of message to deny that he send it.

A message in its original form is known as **plaintext** and the mangled information is known as **ciphertext**. The process for producing ciphertext from plaintext is known as **encryption**. The reverse of encryption is called **decryption**.

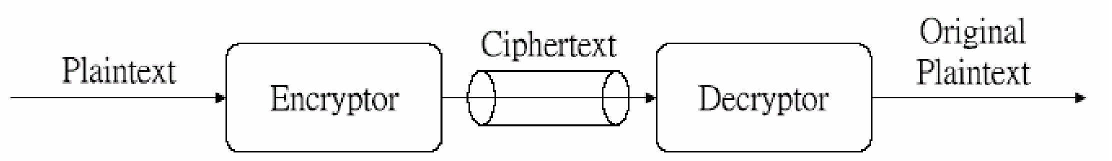


Figure 1.1 Cryptosystem process

Some encryption algorithms use a key, denoted by $\boldsymbol{k}$ so that the ciphertext message depends on both the original plaintext message and the key value, denoted $E_k(P) = C$. Essentially, E is a set of encryption algorithms, and the key $\boldsymbol{k}$ selects one specific algorithm. Sometimes the encryption and decryption keys are the same, so that $D_k(E_k(P)) = P$. This style of encryption is called **symmetric** encryption because D and E are mirror-image processes. These algorithm, require the sender and receiver to agree on a key before they pass messages back and forth. This key must be kept secret. The security of a symmetric algorithm resets in the key; divulging the key means that anybody could encrypt and decrypt messages in this cryptosystem. At other times encryption and decryption keys come in pairs. Then a decryption key, k_D , inverts the encryption of key k_E , so that $D_{k_D}(E_{k_E}(P)) = P$. Encryption algorithms of this form are called **asymmetric**, because converting C back to P is not just reversing the steps of E. A cryptosystem uses a symmetric key algorithm is called a **secret key cryptosystem** and used an asymmetric key algorithm is called a **public key cryptosystem** .

1.2.2 Secret Key Cryptosystem

Secret key cryptography involves the use of a single key. Given a message(called plaintext) and the key, encryption produces unintelligible data(called ciphertext), which is about the same length as the plaintext was. Decryption is the reverse of encryption, and uses the same key as encryption. The most well known example of a secret key cryptosystem is DES (Data Encryption Standard).[2] A secret key cryptosystem is shown as Fig. 1.2.

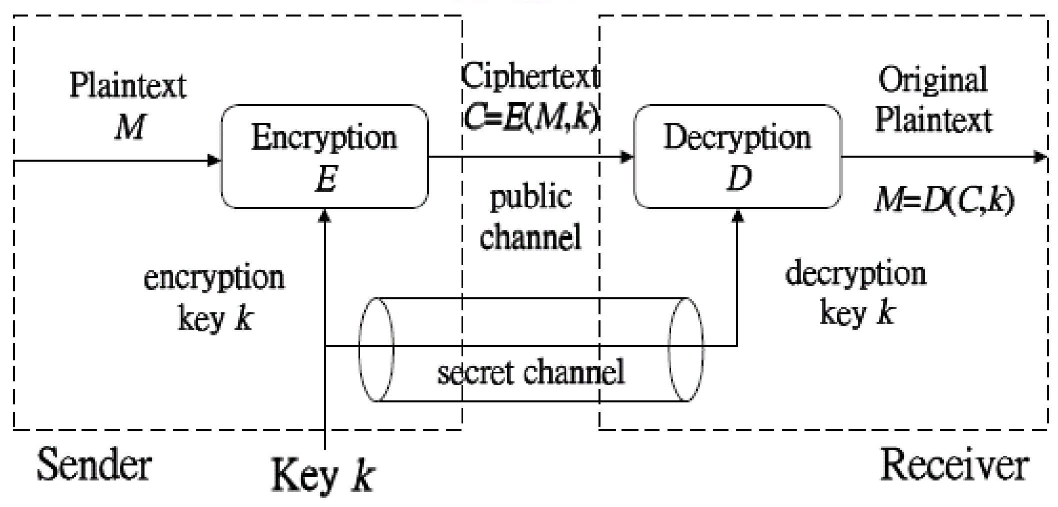


Figure 1.2 Block diagram of secret-key cryptosystem

A secret key cryptosystem have several disadvantages described as below:

(1) How to get the encryption and decryption keys between the sender and receiver? This problem is called the key distribution problem. If the sender and receiver who have never meet before, then this problem become more seriously. Furthermore, a secure channel is hard to achieve in a real world.

(2) If there are n subscribers in a network, then everyone have to hold n-1 keys to communicate with others, in other words, then there will be n(n-1)/2 key pairs in a

network. If n=1000, then there are 499500 key pairs in a network. It is a big problem to manage such many keys.

(3) Digital signature can not be achieved by using the secret key cryptosystem since the sender and receiver both have the same key for encryption and decryption. Thus the sender may repudiate afterwards to the information that has sent before because you cannot tell who is responsible for a signature generated with a shared key.

1.2.3 Public Key Cryptosystem

In traditional cryptography, the sender and receiver of a message know and use the same secret key; the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. The main challenge is getting the sender and receiver to agree on the secret key without anyone else finding out. If they are in separate physical locations, they must trust a courier, a phone system, or some other transmission medium to prevent the disclosure of the secret key. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all messages encrypted or authenticated using that key. The generation, transmission and storage of keys are called **key management**; all cryptosystems must deal with key management issues. Because all keys in a secret key cryptosystem must remain secret, secret-key cryptography often has difficulty providing secure key management, especially in open systems with a large number of users.

In order to solve the key management problem, Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptography in 1976. Public-key cryptosystems have two primary uses, encryption and digital signatures. In their system, each person gets a pair of keys, one called the public key and the other called the private key. The public key is published, while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. In this system, it is no longer necessary to trust the security of some means of communications. The only requirement is that public keys be associated with their users in a trusted (authenticated) manner (for instance, in a trusted directory). Anyone can send a confidential message by just using public information, but the message can only be decrypted with a private key, which is in the sole possession of the intended recipient. Furthermore, public-key cryptography can be used not only for privacy (encryption), but also for authentication (digital signatures) and other various techniques.

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. Typically, the defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For instance, some public-key cryptosystems are designed such that deriving the private key from the public key requires the attacker to factor a large prime number. In this case, it is computationally infeasible to perform the derivation. This is the idea behind the RSA public-key cryptosystem. A secure communication using a public-key cryptosystem is shownin Fig. 1.3.

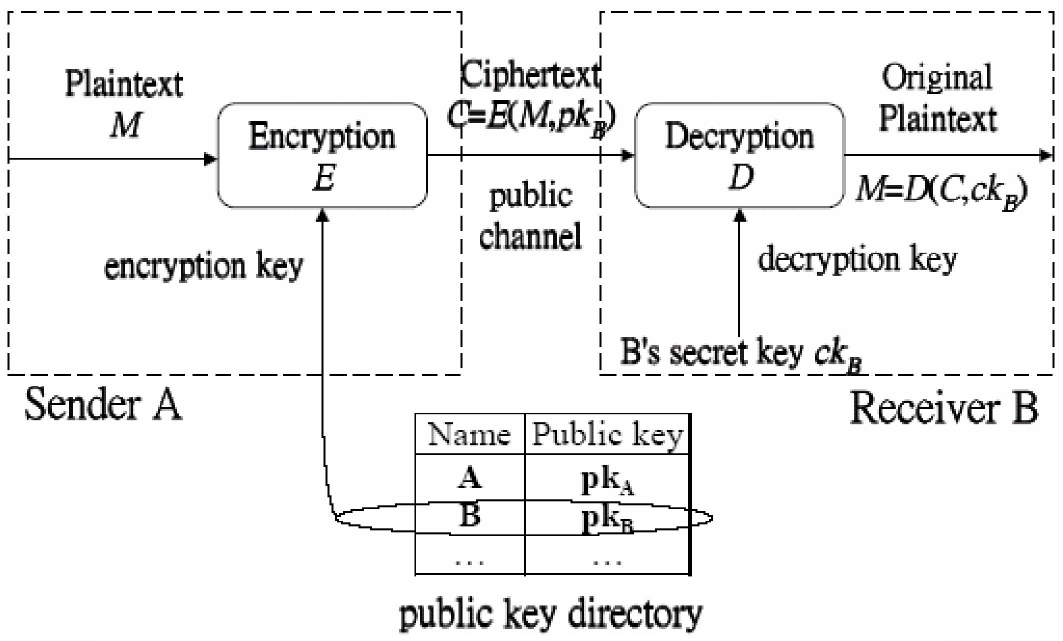


Figure 1.3 Block diagram of public-key cryptosystem

Chapter 2

The RSA Cryptosystem

2.1 The RSA Algorithm

2.1.1 Number Theory

Before describing RSA cryptosystem, something about number theory should be introduced as background knowledge.

1. Modular arithmetic

Given integers a, b, and n. a is congruent to b modulo n means $a-b=k\times n$ for some integer k. The integer b is called a residue of a modulo n, and it is written by

$$a \equiv b \bmod n$$

Conversely, a is called a residue of b modulo n.

2. Multiplicative inverse

Given integers a and n in the range [0, n-1], and gcd (a, n) = 1. There exists an integer x such that $ax \bmod n = 1$, i.e.,

$$ax \equiv 1 \bmod n$$

The integers a and x are multiplicative inverses.

3. Euler totient function

The Euler totient function $f(n)$ is the number of positive integers less than n that are relatively prime to n. For $n = p\times q$ and p, q primes,

$f(n) = f(p) \times f(q) = (p-1)(q-1)$

4. Euler theorem

For integers a and n with gcd $(a, n) = 1$,

$a^{f(n)} \equiv 1 (\bmod n)$

2.1.2 The RSA Scheme

The RSA algorithm was invented by Rivest, Shamir, and Adleman [3]. Let p and q be two distinct large random primes. The modulus n is the product of these two primes: $n = pq$.

Euler's totient function of n is given by

$$f(n) = (p-1)(q-1) \quad .$$

Now, select a number $1 < e < f(n)$ such that

$$\gcd(e, f(n)) = 1 \quad ,$$

and compute d with

$$d = e^{-1} \bmod f(n)$$

using the extended Euclidean algorithm. Here, e is the public exponent and d is the private exponent. Usually one selects a small public exponent, e.g., e = 216 + 1. The modulus n and the public exponent e are published. The value of d and the prime numbers p and q are kept secret. Encryption is performed by computing

$$C = M^e (\bmod n) \quad ,$$

where M is the plaintext such that $0 \le M < n$. The number C is the ciphertext from which the plaintext M can be computed using

$$M = C^d (\bmod n)$$

The correctness of the RSA algorithm follows from Euler's theorem: Let n and a be positive, relatively prime integers. Then

$$a^{f(n)} = 1(\bmod n) \quad .$$

Since we have $ed = 1 \bmod f(n)$, and $ed = 1 + kf(n)$ for some integer K, we can write

$$\begin{aligned} C^d &= (M^e)^d \bmod n \\ &= M^{ed} \bmod n \\ &= M^{1+kf(n)} \bmod n \\ &= M \cdot (M^{f(n)})^k \bmod n \\ &= M \cdot 1 \bmod n \end{aligned}$$

As an example, we construct a simple RSA cryptosystem as follows: Pick $p = 7$ and $q = 17$, and compute

$$n = p \cdot q = 7 \cdot 17 = 119,$$
$$f(n) = (p-1)(q-1) = 96.$$

The public exponent e is selected such that $1 < e < f(n)$ and

$$\gcd(e, f(n)) = \gcd(e, 96) = 1 \quad .$$

For example, $e = 5$ would satisfy this constraint. The private exponent d is computed by

$$\begin{aligned} d &= e^{-1}(\bmod n) \\ &= 5^{-1}(\bmod 96) \\ &= 77 \end{aligned}$$

which is computed using the extended Euclidean algorithm, or any other algorithm for computing the modular inverse. Thus, the user publishes the public exponent and the modulus: $(e, n) = (5, 119)$, and keeps the following private: $d = 77$, $p = 7$, $q = 17$.

A typical encryption/decryption process is executed as follows:

Plaintext: M = 19

Encryption: $C = M^e(\bmod n)$

$C = 2476099$ (mod 119)

$C = 66$

Ciphertext: $C = 66$

Decryption: $M = C^d \pmod{n}$

$M = 1.27\ldots\times 10^{140} \pmod{119}$

$M = 19$

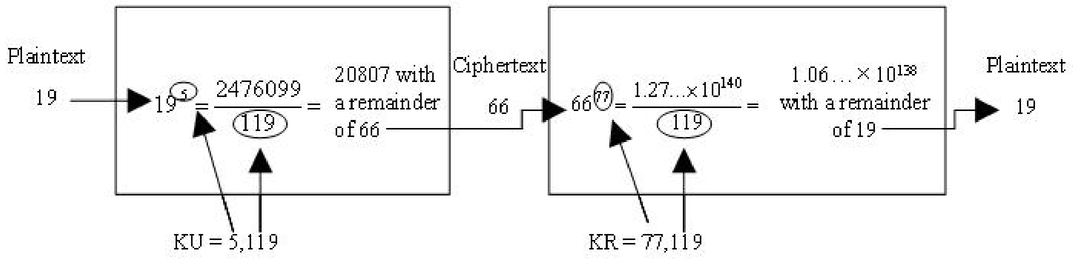


Figure 2.1 **An example of RSA encryption/decryption**

2.2 Digital Signature

The RSA algorithm provides a procedure for signing a digital document, and verifying whether the signature is indeed authentic. The signing of a digital document is somewhat different from signing a paper document, where the same signature is being produced for all paper documents. A digital signature cannot be a constant; it is a function of the digital document for which it was produced. After the signature (which is just another piece of digital data) of a digital document is obtained, it is attached to the document for anyone wishing the verify the authenticity of the document and the signature. Here we will brie fly illustrate the process of signing using the RSA cryptosystem. Suppose Alice, who has public key (e_a , n_a) and private key (d_a , n_a), wants to sign a message, and Bob would like to obtain a proof that this message is indeed signed by Alice.

First, Alice executes the following steps:

1. Alice takes the message M and computes $S = M^{d_a} \pmod{n_a}$.

2. Alice makes her message M and the signature S available to any party wishing to verify the signature.

Bob executes the following steps in order to verify Alice's signature S on the document M:

1. Bob obtains M and S, and obtains Alice's public key (e_a , n_a).

2. Bob computes $M' = S^{e_a} \pmod{n_a}$.

3. If $M' = M$ then the signature is verified. Otherwise, either the original message M or the signature S is modified. Thus, the signature is not valid.

We note that the protocol examples given here for illustration purposes only, they are simple 'textbook' protocols; in practice, the protocols are somewhat more complicated. For example, secret key cryptographic techniques may also be used for sending private messages. Also, signing is applied to messages of arbitrary length. The signature is often computed by first computing a hash value of the long message and then signing this hash value.

2.3 The security of RSA

Three possible approaches to attacking the RSA algorithm are as follows:

- **Brute force:** This involves trying all possible private keys.
- **Mathematical attacks:** There are several approaches, all equivalent in effect to factoring the product of two primes.
- **Timing attacks:** These depend on the running time of the decryption algorithm.

There defense against the brute-force approach is the same for RSA as for other cryptosystems—namely, use a large key space. Thus, the larger the number of bits in e

and *d*, the better. However, because the calculations involved, both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run. In this section, we provide an overview of mathematical and timing attacks.

We can identify three approaches to attacking TSA mathematically:

- Factor n into its two prime factors. This enables calculation of $f(n) = (p-1)(q-1)$, which enables determination of $d = e^{-1} \bmod f(n)$.
- Determine $f(n)$ directly, without first determining *p* and *q*. again, this enables determination of $d = e^{-1} \bmod f(n)$.
- Determine *d* directly, without first determining $f(n)$.

Most discussions of the cryptanalysis of RSA have focused on the task of factoring *n* into its two prime factors. With presently known algorithms, determining *d* given *e* and *n*, appears to be at least as time-consuming as the factoring problem. Hence, we can use factoring performance as a benchmark against which to evaluate the security of RSA.

For a large *n* with large prime factors, factoring s a hard problem, but not as hard as it used to be. Table 2.1 shows the results to date that return the RSA ciphertext to plaintext. The level of effort is measured in MIPS-years: a million-instruction-per-second processor running for one year, which is about 3×10^{13} instructions executed. A 200-MHz Pentium is about a 50-MIPS machine.

The threat to larger key sizes is two fold: the continuing increase in computing power, and the continuing refinement of factoring algorithm. We have seen that the move to a different algorithm resulted in a tremendous speedup. We can expect further refinements in the GNFS, and the use of an even better algorithm is also a possility. In fact, a related algorithm, the special number field sieve (SNFS), can factor numbers with a specialized form considerably faster than the generalized number field sieve.

Thus, we need to be careful in choosing a key size for RSA. For the near future, a key size in the range of 1024 to 2048 bits seems reasonable.

In addition to specifying the size of *n*, a number of other constraints have been suggested by researchers. To avoid values of *n* that may be factored more easily, the algorithm's inventors suggest the following constraints on *p* and *q*:

1. *p* and *q* should differ in length by only a few digits. Thus, both *p* and *q* should be on the order of 10^{75} to 10^{100}.
2. Both $(p - 1)$ and $(q - 1)$ should contain a large prime factor.
3. $\gcd(p - 1, q - 1)$ should be small.

In addition, it has been demonstrated that if $e < n$ and $d < n^{1/4}$, then *d* can be easily determined.

Number of Decimal Digits	Approximate Number of Bits	Data Achieved	MIPS-Years	Algorithm
100	332	April 1991	7	Quadratic sieve
110	365	April 1992	75	Quadratic sieve
120	398	June 1993	830	Quadratic sieve
129	428	April 1994	5000	Quadratic sieve
130	431	April 1996	500	Generalized number field sieve

Table 2.1 Progress in factorization

Chapter 3
Scalable Montgomery Multiplication Architecture

3.1 Modular Exponentiation Operation

Once an RSA cryptosystem is setup, the modulus and the private and public exponents are determined and the public components have been published, the senders as well as the recipients perform a single operation for signing, verification, encryption, and decryption. The RSA algorithm in this respect is one of the simplest cryptosystems. The operation required is the computation of $M = C^d \pmod{n}$, which is the modular exponentiation. In the following sections we will discuss how to solve modular exponentiation problem and what is the most popular hardware structures for performing the modular multiplication and exponentiations.

3.1.1 Modular Exponentiation Algorithm

There are two algorithms to convert the modular exponentiation of $C = M^E \pmod{N}$ into a square of modular multiplications. (Note that from this chapter, we change the notation e to E and n to N.) Each is determined by the order in which it process exponent e, from low bits to high bits in algorithm LSB-First, and conversely in MSB-First. [4] We consider the binary representation of the exponent $E = (e_{k-1}e_{k-2}...e_2e_1e_0)$ with $e_{k-1} = 1$, where E is a k-bit integer.

```
Algorithm MSB-First ( $M$, $E$, $N$)
Int $R$ = 1;
for $i$ = $k$-1 downto 0
    $R = R \times R$ mod $N$;
    if $e_i$ = 1 then
        $R = R \times M$ mod $N$;
return $R$;
end
```

```
Algorithm LSB-First ( $M$, $E$, $N$)
int $R$ = 1, $P$ = $M$;
For $i$ = 0 to $k$-1
if $e_i$ = 1 then
    $R = R \times P$ mod $N$;
$P = P \times P$ mod $N$;
Return $R$;
end
```

Both of the two algorithms above need to do a lot of modular multiplication operations. Therefore, how to improve the speed of modular multiplication is a critical issue in RSA implementation.

3.1.2 Montgomery Modular Multiplication Algorithm

The Montgomery multiplication algorithm [5] is an efficient method for modular multiplication with an arbitrary modulus, particularly suitable for implementation on general-purpose computers (signal processors or microprocessors). The character of the Montgomery Algorithm is that it performs modular reduction during the multiplication process, and replaces division by *M* operation with division by a power of 2. No division operation is needed at any point in the process. This operation is easily accomplished on a computer since the numbers are typically represented in binary form.

Let *X*, *Y*, and *M* be the multiplicand and multiplier and the modulus respectively and let *n* be the number of digits in their binary representation. So we can denote *X*, *Y*, and *M* as follows:

$$X = \sum_{i=0}^{n-1} x_i \times 2^i, \quad Y = \sum_{i=0}^{n-1} y_i \times 2^i, \quad M = \sum_{i=0}^{n-1} m_i \times 2^i$$

The pre-condition of the Montgomery algorithm are as follows:

- The modulus *M* needs to be relatively prime to the *radix*, i.e. there exists no common divisor for *M* and the radix;
- The multiplicand(*Y*) and the multiplier(*X*) need to be smaller than *M*.

As we use the binary representation of the operands, then the modulus *M* needs to be odd to satisfy the first precondition. The Montgomery algorithm uses the least significant digit of the accumulating *modular partial product* to determine the multiple of *M* to subtract. The usual multiplication order is reversed by choosing multiplier digits from least to most significant and shifting down. If *S* is the current modular partial product, then *q* is chosen so that *S* + *qM* is a multiple of the radix *r*, and this is right shifted by *r* positions, i.e. divided by *r* for use in the next iteration. So, after n iterations, the result obtained is $S = X \times Y \times r^{-n} \bmod M = MM(X, Y)$ .The Montgomery Multiplication Algorithm is shown below:

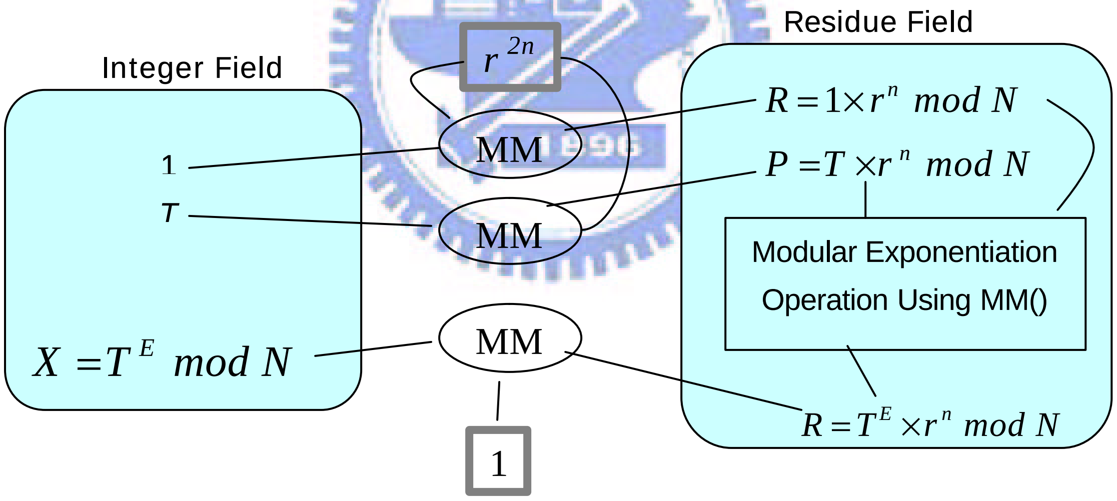
```
Montgomery Algorithm ($X$, $Y$, $M$)
Step 1:   $S := 0$
Step 2:     for $i = 0$ to $n - 1$
Step 2a:          $S := S + X_i \cdot Y$
Step 2b:          if $S$ is odd, $S := S + M$
Step 2c:          $S := S / 2$
Step 3:   If $S \geq N$, $S = S - M$
Step 4:   Return $S$
```

3.1.3 Appling Montgomery Multiplication Algorithm to Modular Exponentiation Operation

We can employ the Montgomery Multiplication Algorithm to do modular exponentiation operation. To achieve this goal, the Montgomery's technique requires some pre-processing and post-processing steps, which are needed to convert the numbers to and from the residue based representation. However, the cost of these steps is negligible when many consecutive modular multiplications are to be executed, as in the case of RSA. This is the reason why the Montgomery' s method is considered the most efficient algorithm for implementing RSA operations. The applied Montgomery algorithm is shown as following:

```
ModExp Algorithm ($T$, $E$, $N$)
Step 1. $R$ = MM(1, $r^{2n}$) = 1 $\times r^n$ mod $N$;
Step 2. $P$ = MM($T$, $r^{2n}$) = $T$ $\times r^n$ mod $N$;
Step 3. For $i$ = 0 to $k$-1;
Step 4.     if $e_i$ = 1 then
Step 5.         $R = R \times P$ mod $N$;
Step 6.     $P = P \times P$ mod $N$;
Step 7. $X$ = MM($R$, 1)
Step 8. Return $X$;
```

Form this algorithm, we can get: $X = T^E \bmod N$. We also show an illustration of the algorithm above as follows:



Integer Field
Residue Field
r^{2n}
MM
MM
MM
1
T
$X = T^E \ mod\ N$
$R = 1 \times r^n \ mod\ N$
$P = T \times r^n \ mod\ N$
Modular Exponentiation Operation Using MM()
$R = T^E \times r^n \ mod\ N$
1

Figure 3.1 Montgomery Modular Exponentiation Algorithm

3.2 Scalable Architecture for Montgomery Multiplication

There are many algorithms and hardware implementations of the Montgomery multiplication for a limited precision of the operands were proposed [7, 8]. In order to get improved performance, high-radix algorithms have also been proposed [9, 10]. . However, these high-radix algorithms usually are more complex and consume significant amounts of chip area, and it is not so evident whether the complex circuits derived from them provide the desired speed increase. A theoretical investigation of the design tradeoffs for high-radix modular multipliers is given in [11]. Low-radix designs are usually more attractive for hardware implementation [12].

3.2.1 Scalability

We consider an arithmetic unit as scalable if:

the unit can be reused or replicated in order to generate long-precision results independently of the data path precision for which the unit was originally designed.

For example, a multiplier designed for 768 bits cannot be immediately used in a system which needs 1,024 bits. The functions performed by such designs are not consistent with the design which required in the larger precision system, and the multiplier must be redesigned. In order to make the hardware scalable, the usual solution is to use software and standard digit multipliers. In the following, Koc[13] propose a hardware algorithm and design approach for the Montgomery multiplication that are attractive in terms of performance and scalability.

3.2.2 A Word-Based Radix-2 Montgomery Multiplication Algorithm

In 2001, Koc proposed a word-based radix-2 Montgomery Multiplication Algorithm [13]. Given two operands Y (multiplicand) and X (multiplier) and the modulus M, the algorithm presented in this section executes a series of operations to generate $X \times Y \times r^{-n} \bmod M$, scanning Y and M word-by-word and scanning X bit-by-bit. This characteristic can derive a hardware implementation that is very regular and based on simple operations. This algorithm is shown as follows:

<u>MWR2MM Algorithm</u>

```
1.  $S = 0$
2.  for $i = 0$ to $m-1$
3.      $(C, S^{(0)}) = x_i Y^{(0)} + S^{(0)}$
4.      if $S_0^{(0)} = 1$ then
5.      $(C, S^{(0)}) = (C, S^{(0)}) + M^{(0)}$
6.          for $j = 1$ to $e-1$
7.              $(C, S^{(j)}) = C + S^{(j)} + x_i Y^{(j)} + M^{(j)}$
8.              $S^{(j-1)} = (S_0^{(j)}, S_{w-1,\dots,1}^{(j-1)})$
9.          $S^{(e-1)} = (C, S_{w-1,\dots,1}^{(e-1)})$
10.     else
11.         for $j = 1$ to $e-1$
12.             $(C, S^{(j)}) = C + S^{(j)} + x_i Y^{(j)}$
13.             $S^{(j-1)} = (S_0^{(j)}, S_{w-1,\dots,1}^{(j-1)})$
14.         $S^{(e-1)} = (C, S_{w-1,\dots,1}^{(e-1)})$
15. If $S > M$ then $S = S - M$
```

The MWR2MM algorithm computes a partial sum S for each bit of X, scanning the words of Y and M. Once the precision is exhausted, another bit of X is taken, and the scan is repeated. Thus, the algorithm imposes no constraints to the precision of operands. The arithmetic operations are performed in precision w bits, and they are independent of the precision of operands. The carry variable C must be in the set $\{0, 1, 2\}$. This condition is imposed by the addition of the three vectors S, M, and x_iY.

In this algorithm, the n-bits operands are split into w-bits words. For now, suppose that e words are used. Word and bit vectors are represented as: $M = (0, M^{(e-1)}, ..., M^{(1)}, M^{(0)})$, $Y = (0, Y^{(e-1)}, ..., Y^{(1)}, Y^{(0)})$, $S = (0, S^{(e-1)}, ..., S^{(1)}, S^{(0)})$, and $X = (0, X^{(e-1)}, ..., X^{(1)}, X^{(0)})$, where the words are marked with superscripts and the bits are marked with subscripts. M, Y, and S are extended to $e + 1$ words by a most significant zero word. Inserting an extra most-significant word with value 0 allows the computation of $S^{(e-1)}$ once the loop is completed. The concatenation of two vectors A and B is represented as (A, B). A particular range of bits in a vector A from position i to position j, $j > i$, is represented as $A_{j...i}$. The bit position i of the kth word of an operand A is represented as $A_i^{(k)}$.

3.3 Mapping the MWR2MM Algorithm to Hardware

3.3.1 Parallel Computation of the MWR2MM

The dependency between operations within the j loop restricts their parallel execution due to the dependency on the carry bits. However, instructions in different i loops may be executed in parallel. The dependency graph for the algorithm is shown in Fig. 3.2. An atomic task is represented by a circle and it is labeled according to the type of action it performs. Tasks A and B executes basically the following steps:

1. Add one word from each one of the vectors S, x_iY, and M (the addition of M depends on a test), and
2. One bit right shift of an S word. For this operation, the generation of the shifted $S^{(j-1)}$is possible only after computing the least significant bit of $S^{(j)}$.

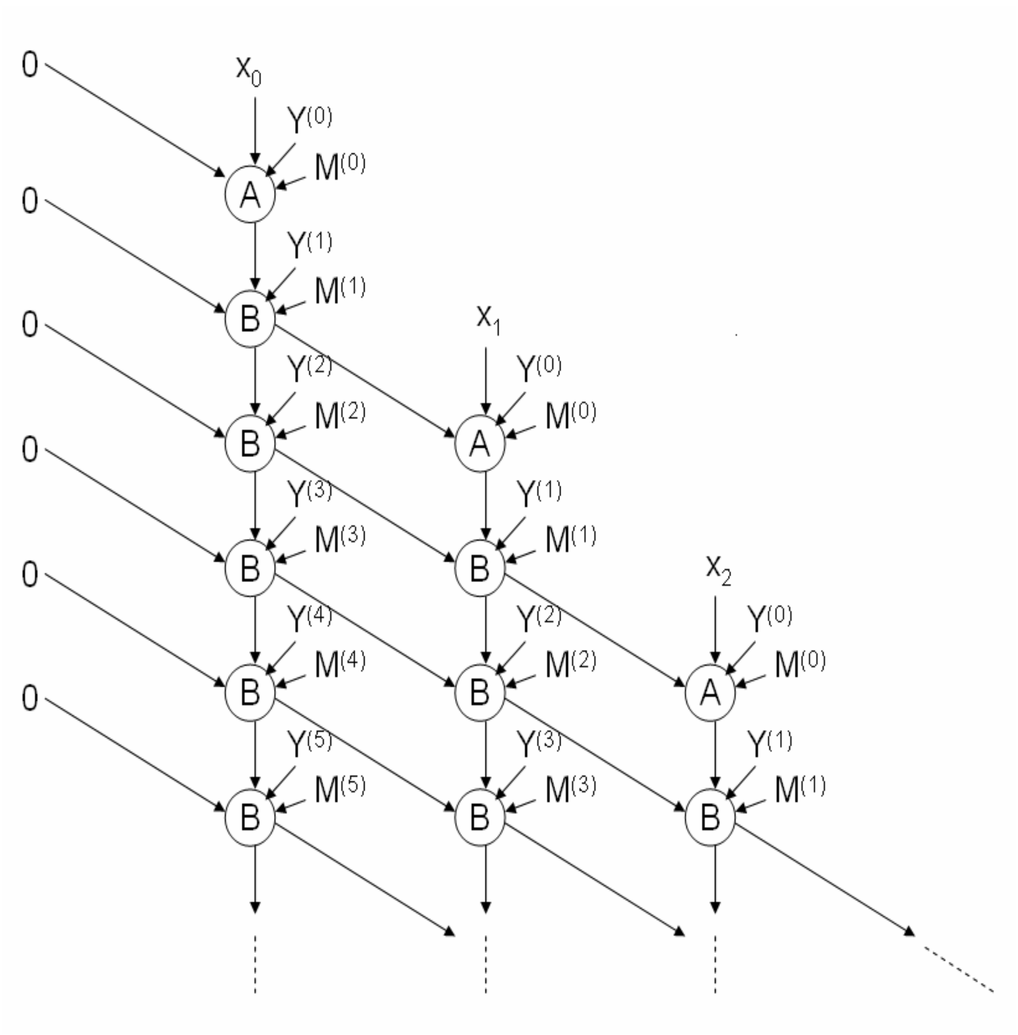


Figure 3.2 The dependency graph for the MWR2MM algorithm

Task A differs from task B because, additionally to these two steps, it also needs to test the even condition for the result of the addition $x_iY^{(0)} + S^{(0)}$, and store this test result for the other tasks dealing with the next words of the same operands (same column in the graph). The dependency graph has $e + 1$ task per column. Each column may be computed by a separate processing element (PE) and the data generated by one PE may be passed to another PE in a pipelined fashion. Each task is computed in one clock cycle. The partial data pass through the computation process is in Carry-Save form. A number (S) is represented by two bit vectors: carry vector (SC) and partial-sum vector (SS) such that the form of S is obtained computing $SC + SS$. After $e + 1$ clock cycles, the PE finishes its portion of work, and becomes available for further computation. In case there is no available PE and there is work to do, the pipeline must stall and wait for the working PEs to finish their jobs. Since the PE at the end of the pipeline has no way of communicating its result to another PE, we need to provide extra buffers for them. In the worst case, which happens when there is only one PE, there must be $2e$ extra buffers of w length to hold these partial sum. An example of the computation for 7-bits operands is shown in Figure 3.3 for the word size $w = 1$ provided that there are sufficient number of PEs preventing the pipeline to stall. Note that there is a delay of 2 clock cycles between the stage for x_i and the stage for x_{i+1}. The total execution time for the computation takes 20 clock cycles in this example.

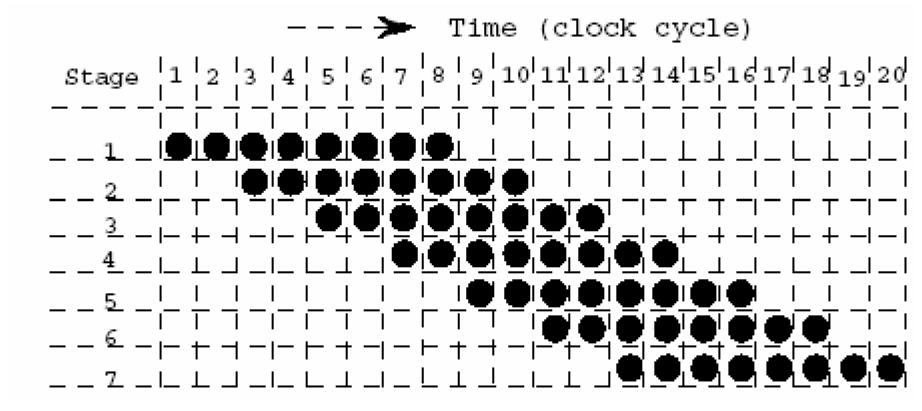


Figure 3.3 An example of pipeline computation for 7-bits operands, where $w = 1$.

If there are at least $\lceil (e+1)/2 \rceil$ PEs in the pipeline organization, the pipeline stalls do not take place. For the example in Figure 4.3, less than $\lceil 8/2 \rceil = 4$ PEs cause the pipeline to stall. Figure 3.4 shows what happens if there are only three PEs available for the same example.

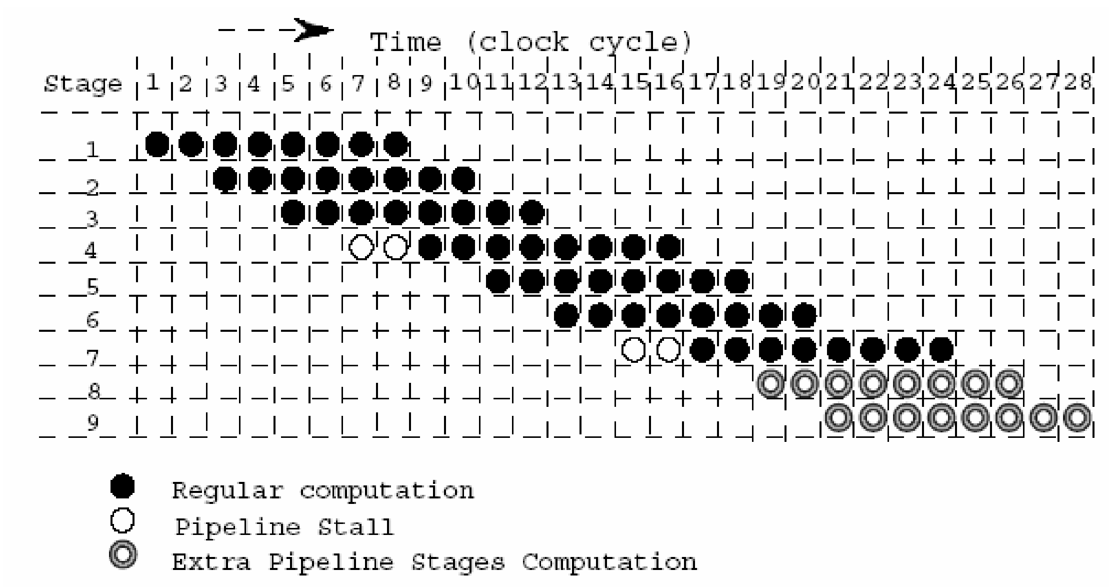


Figure 3.4 An example of pipeline computation for 7-bit operands, where *w* = 1.

3.3.2 Scalable Architecture for Montgomery Multiplication

A pipelined organization for the system is shown in Fig. 4.4. Each processing element in the pipeline relays the received words to the next downstream unit. All paths are *w*-bits wide, except for the x_i inputs (only 1 bit). If more precision is required, it is only necessary to feed more words. The final and intermediate results are stored in the queue. Gray boxes indicate registers.

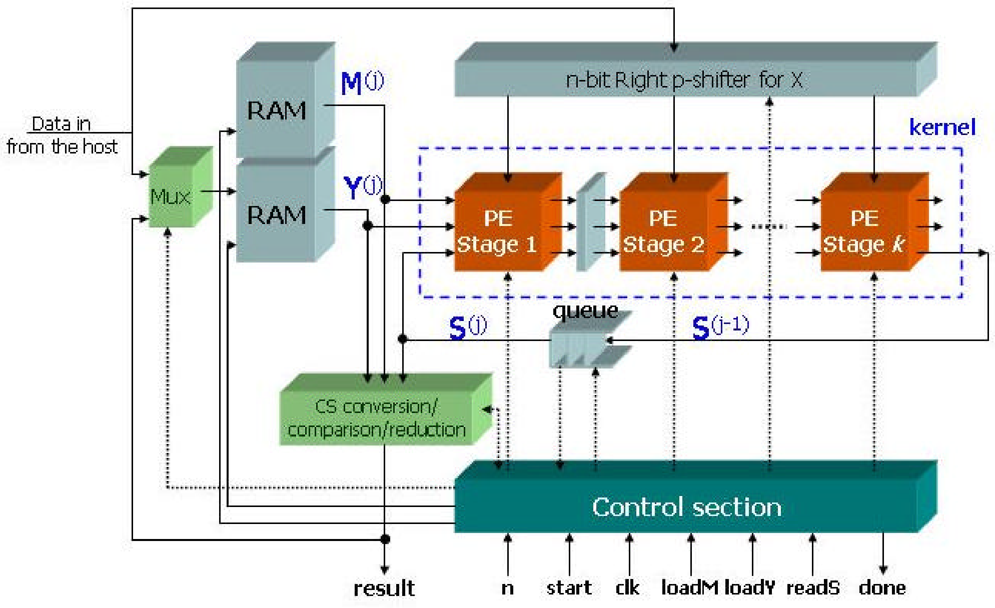


Figure 3.5 Pipelined organization for the multiplier.

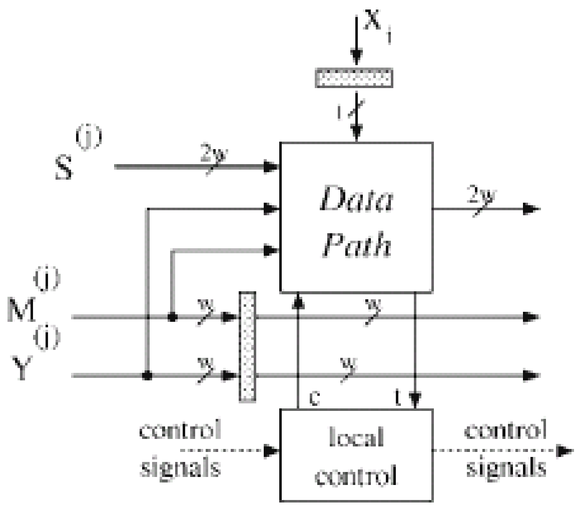


Figure 3.6 The block diagram of the processing element (PE)

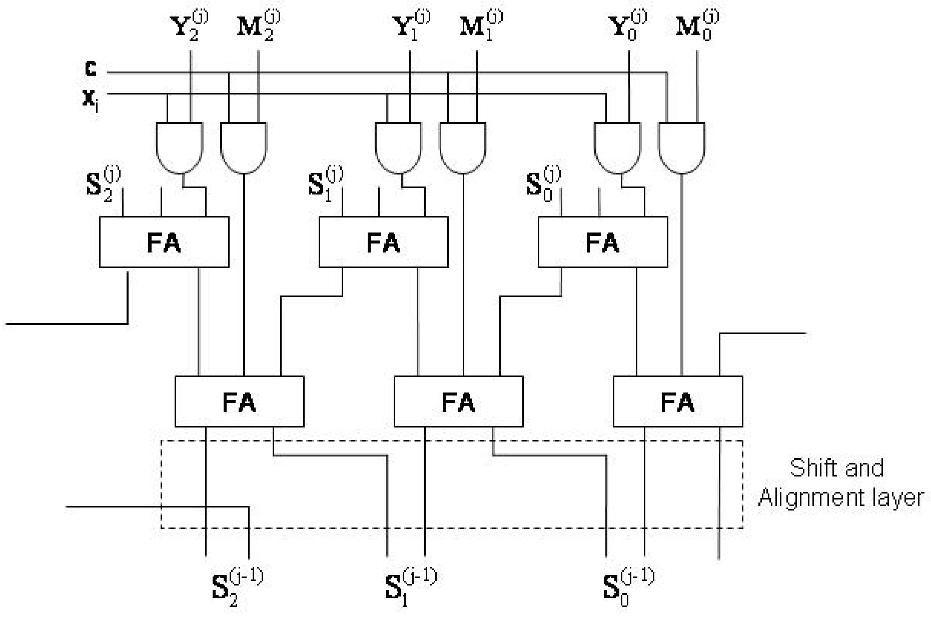


Figure 3.7 PE's data path for *w* = 3 bits.

The block diagram of the PE is shown in Figure 3.6, and the PE's data path is shown in Figure 3.7. The data path receives words $M^{(j)}$, $Y^{(j)}$, and $S^{(j)}$ from the previous stage in the pipeline and computes the new value of $S^{(j-1)}$. Delaying inputs M and Y, the module provides as outputs the words $M^{(j-1)}$, $Y^{(j-1)}$, and $S^{(j-1)}$. In fact, since the signals provided by the PE pass by an interstage register, these signals will reach the next PE one clock cycle later. That means, when one PE is working on word j, the next PE is working on word j - 2.

The PE's data path needs to make the information on the least significant bit (t) of the computation $S^{(0)} + x_i Y^{(0)}$ available to the local control. This bit is used to generate the control signal c (controls addition of M). The local control is responsible for generating and keeping c during a pipeline cycle, and also relay some control signals to downstream PEs. The basic operations executed in the data path are: (1) generation of the product $x_i Y^{(j)}$, (2) generation of the product $cM^{(j)}$, and (3) addition of three words ($S^{(j)}$, $x_i Y^{(j)}$, and $cM^{(j)}$) and a carry digit in the set {0, 1, 2}. The shaded box represents a register. The data path has an alignment section to generate the output words. When computing bits of word j (step j), the circuit generates w - 1 bits of $S^{(j)}$ and the new most significant bit of $S^{(j-1)}$. Bits of $S^{(j-1)}$ computed at step j - 1 must be delayed and concatenated with the most significant bit generated at step j.

The total computation time T in clock cycles (assuming that each task consumes one clock cycle) when p stages are used in the pipeline to compute the MM with n bits of precision is

$$T = \begin{cases} Lkp + e - 1 & \text{if } (e+1) \le Lp \\ k(e+1) + L(p-1) & \text{otherwise.} \end{cases} \qquad (1)$$

The first case shown in the equation represents the situation when the first PE in the pipeline cannot start its computation with another bit of X because the least significant word of S didn't show up at the pipeline output yet. The second case models the condition when the number of words in the operands is large enough to keep all the PEs working all the time. As the word size increases, there is a reduction in the total execution time up to a lower bound that can be obtained from the equation above. The best parameters in terms of number of PEs and word size for a given operand precision and chip area depends on the clock cycle time of the final implementation and the total number of clock ticks.

There are only two important values of operand precision are presented. Observe that, when the operand precision is small, the number of PEs may be small and, when the precision is high, the number of PEs should be as high as possible. Thus, the final decision on the actual configuration depends on the precision for which the hardware will be used the most and the available area. In order to make the design efficient for a large range of operand precision, the optimal solution for the highest expected precision is the best choice. Figure 3.8 and figure 3.9 show different configurations. The areas of those configurations are approximately when (w, p) = (8, 40), (16, 20), and (32, 10). It is easy to see that going from *p* = 40 to *p* = 30 will not affect the computation time for *n* = 256, but will impact the time to compute the multiplication for n = 1024. Thus, the concept of optimal design in this case is relative to the precision of the operands and the available area.

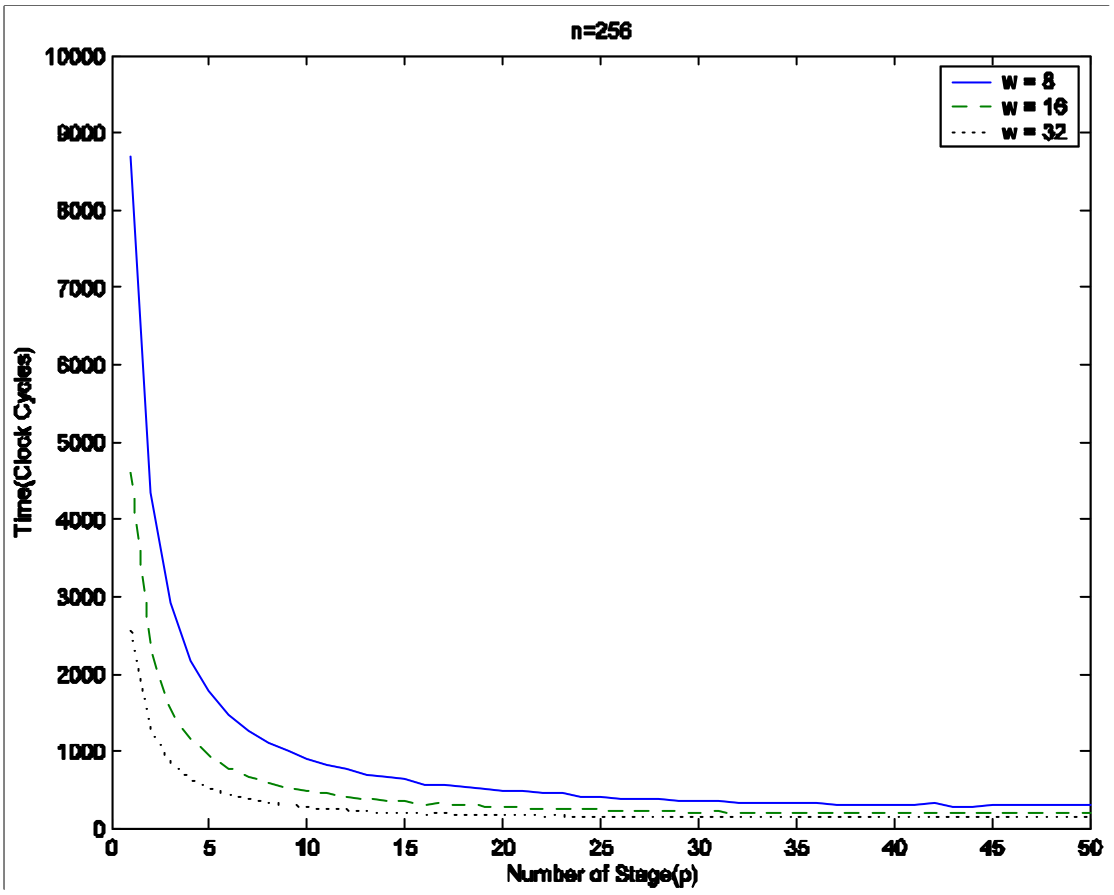


Figure 3.8 The cycle time of different configurations which n = 256

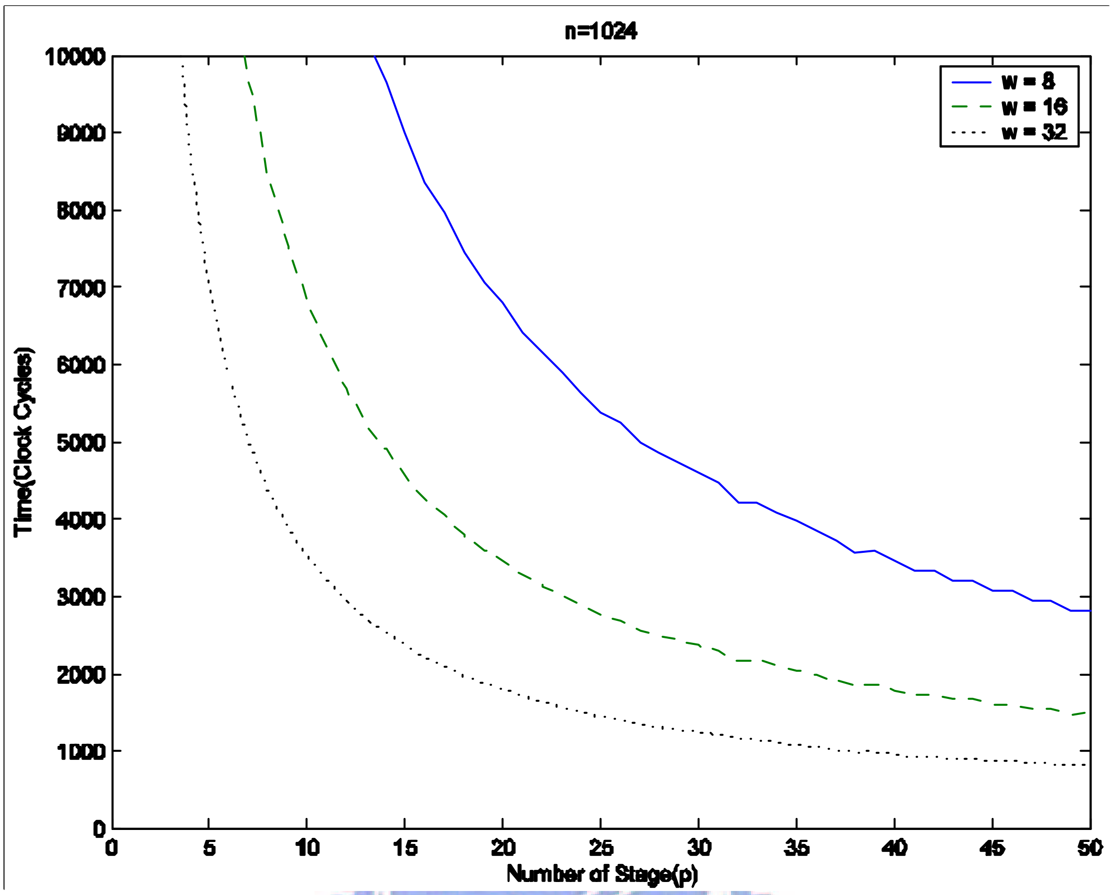


Figure 3.9 The cycle time of different configurations which n = 1024

3.3.3 The Final Word Adder and Subtraction

The proposed Montgomery multiplier generates results in the redundant Carry-Save form; hence we need to perform an extra addition operation at the end of the calculation to obtain the nonredundant form of the result. A full-precision adder would increase the critical path delay and the area, and would also be hard to scale. A word adder of the type given in Figure 3.10 would be suitable for our implementation since the multiplier generates only one word at each clock cycle in the last stage of pipeline, thus we need to perform one word addition at a time.

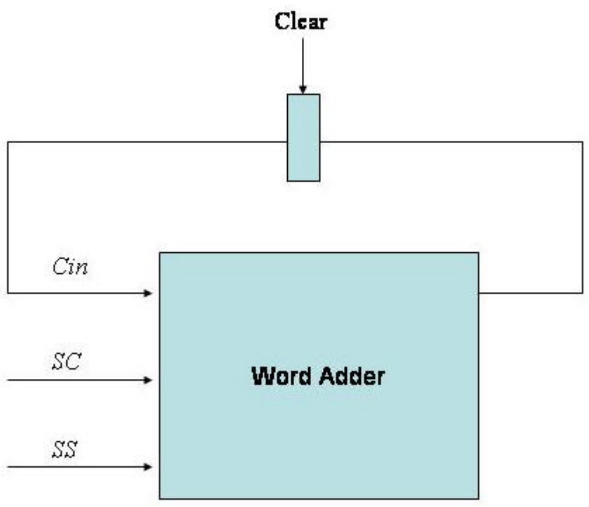


Figure 3.10 The Word Adder for Final Addition.

The adder propagates the carry bit to the next word additions. Thus, the carry from a word addition operation is delayed using a latch and fed back into the *Cin* input of the adder for the next word addition at the next clock cycle. It needs to add subtraction functionality to the word adder because the result might be larger than the modulus, and hence one final subtraction operation is necessary. The final subtraction operation takes place only if the result is larger than the modulus.

Figure 3.11 illustrates what happens in last stage of the pipeline. A pair of redundant words ($SC^{(j)}$, $SS^{(j)}$) are generated each cycle for e clock cycles. The word adder can be used to add these pairs in order to obtain the result words $FS^{(j)}$(Final Sum). Note that only one extra cycle is needed to convert the result from the Carry-Save form to the nonredundant form.

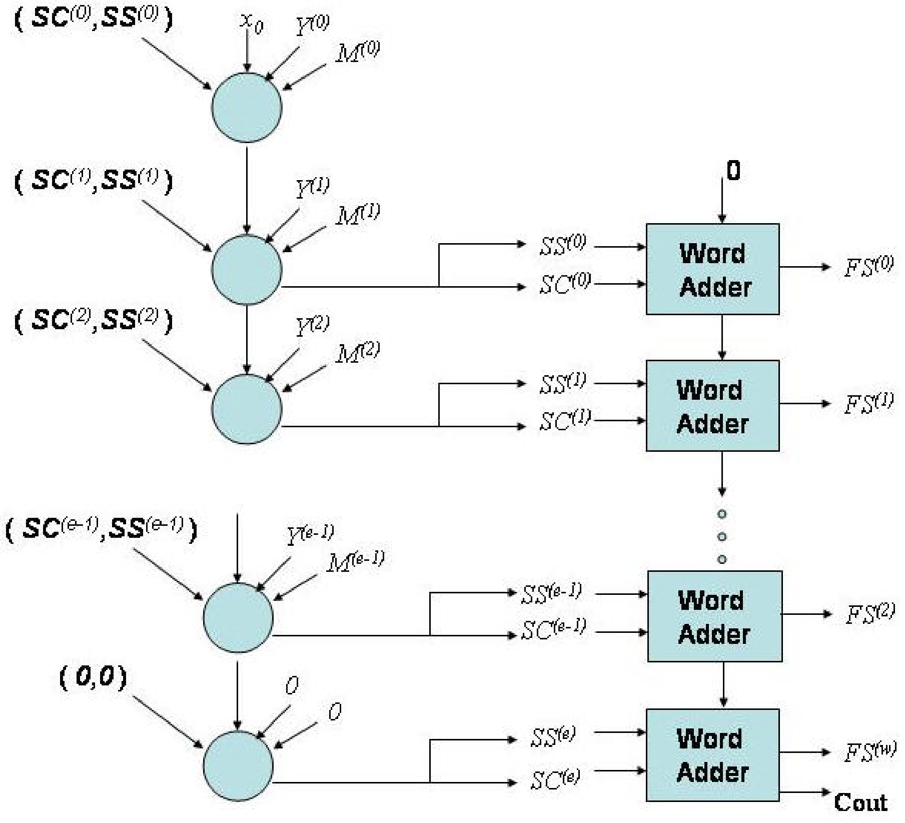


Figure 3.11 Converting the result to final form in the last stage of the pipeline.

3.4 An Improved Scalable Montgomery Multiplication Architecture.

We proposed an improved scalable Montgomery Multiplication Algorithm, and the new architecture can achieve higher clock rate. The main idea is that we substitute the multiplexer for the full adder in the PE's data path. In addition to that idea, we combined the idea from Colin. D. Water [14]. The advantages of the Water's idea are:

(1) We can make the pipeline form of the architecture being continuous without being interrupted by the comparison and subtraction operation.

(2) Since we change the original architecture in order to achieve higher clock rate, additional data must be generated faster. By using the Water's idea, we can solve this problem. We will discuss this problem in next section.

The improved MWR2MM algorithm is shown as follows:

```
Improved MWR2MM Algorithm
pre-condition:
$X < 2M; Y < 2M; mb = M + Y$
$M$ is $m$ bits; $n = m + 2$;
1. $S = 0, C = 0$
2. for $i = 0$ to $n-1$
3.    $q_i = S_0^{(0)} \oplus x_i \cdot Y_0^{(0)}$
4.       for $j = 0$ to $e-1$
5.          case $(x_i, q_i)$
6.          $(1, 1)$: $T^{(j)} = mb^{(j)}$;
7.          $(1, 0)$: $T^{(j)} = Y^{(j)}$;
8.          $(0, 1)$: $T^{(j)} = m^{(j)}$;
9.          $(0, 0)$: $T^{(j)} = 0$;
10.          $(C, S^{(j)}) = C + S^{(j)} + T^{(j)}$
11.             $S^{(j-1)} = (S_0^{(j)}, S_{w-1,\ldots,1}^{(j-1)})$
12.       $S^{(e-1)} = (C, S_{w-1,\ldots,1}^{(e-1)})$
```

Note that the improved MWR2MM algorithm is a little different from the original MWR2MM. The difference between the two algorithms is shown in table 3.1.

	Original	Improved
Difference	1. M is m bits, and computes m iterations.	1. M is m bits, and computes $n = m + 2$ iterations.
	2. The multiplicand Y and multiplier X are smaller then the modulus M. ($X < M, Y < M$)	2. The multiplicand Y and multiplier X are smaller then $2M$. ($X < 2M, Y < 2M$)
	3. Need the final subtraction step to make the result to the correct range.	3. There is no need for final subtraction if the exponentiation is set up in the right way. [14]

Table 3.1 The difference between the two MWR2MM algorithm.

In this architecture, we first substitute the 4 to 2 multiplexer for the full adder in the PE's data path. This can reduce the data arrive time when the data pass through the PEs. Beside, the area of the multiplexer is slightly small than the full adder. Therefore, we can achieve better performance without increase the area. Table 3.2 shows the area and data arrive time between multiplexer and full adder.

	Area(gate count)	data arrive time(ns)
Mux	10.58	0.21
Full Adder	14.1	1.01

Table 3.2 The area and data arrive time between Mux and Full Adder.

Although we can achieve higher speed by the way above, but it still has some problem: how to make the data $mb(mb=M+Y)$ arrive in time without delay the all process of the data path? We find a solution to solve it. We changed the pre-condition so that we can use the idea of Water in order to reduce the final comparison and subtraction. By this method, we can execute the multiplication of the exponential computation continuously. We also solve the *mb* problem. It is known that in modular exponentiation algorithm, the exponential computation is composed of many continuous modular multiplications. The result of the current multiplication is the input of next multiplication. Thus, we insert an extra word adder between two continuous multiplication of exponentiation to compute the result of ($mb = M + Y$) before next multiplication. It only needs an extra cycle time which is not obvious. The new PE's data path of 3 bits word is shown in Figure 3.12, and the diagram between two multiplications of exponential computation is shown in Figure 3.13.

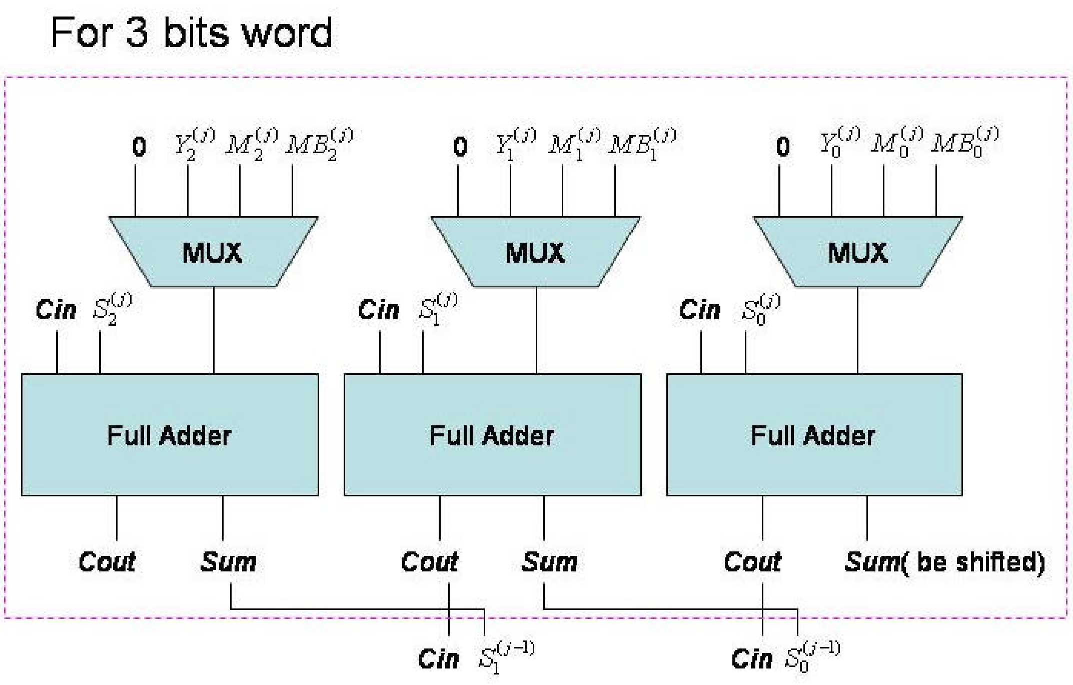


Figure 3.12 An example of improved architecture's data path.

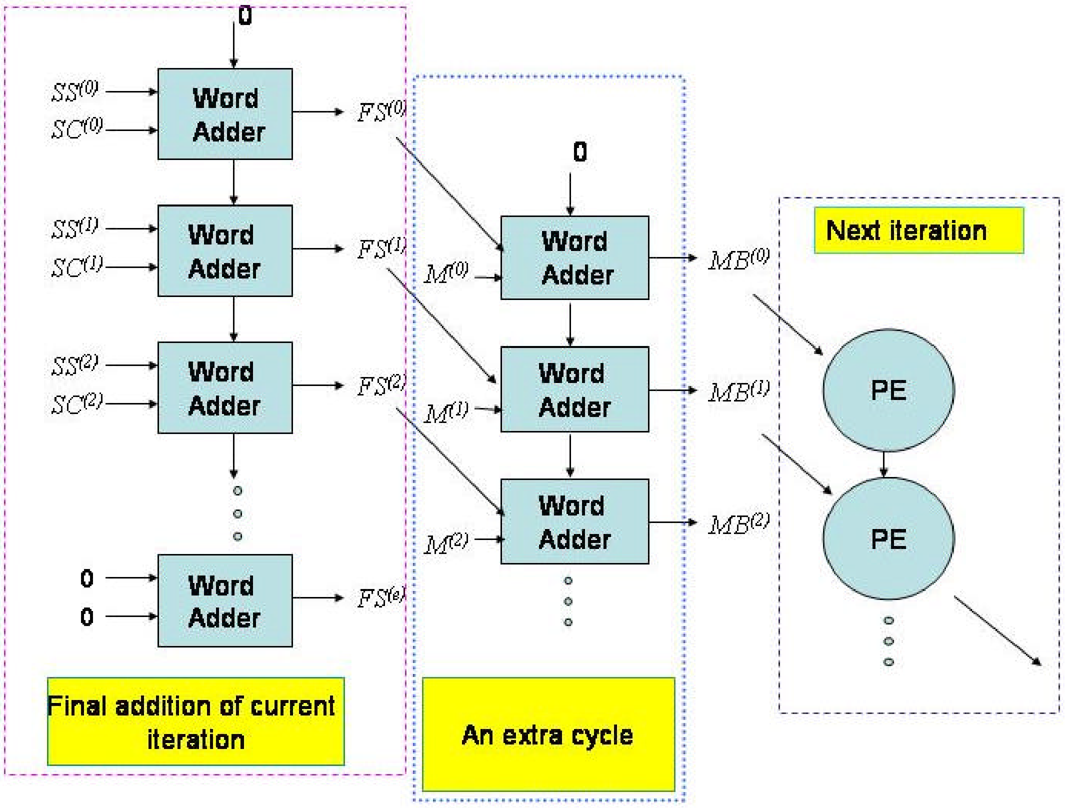


Figure 3.13 The diagram between two multiplications of exponential computation.

Chapter 4
Simulation Result

Our goal is to design a high performance Montgomery Scalable Montgomery Modular Multiplication chip, and compare the performance with the original architecture.

When the circuit architecture is completed, the realization of such a highly integrated chip requires a consequent top-down design flow to ensure the first time functionality and an acceptable total design time. Figure 4.1 shows the typical top-down design flow chart.

At the beginning in our design, we should define the specification to fit related protocol and standard. According to the defined specification, we start to design the chip architecture, then we use the hardware description language called *Verilog* to describe our circuit and simulate. From the simulation results, we can check the architecture fit our specification or not. If it doesn't fit the specification, then we should restart to design a new architecture. After we complete the correct behavior model circuit, the behavior-level circuit can be mapped to the gate-level representation by using the logic synthesis tool(*Synopsys*). Then the gate-level simulation must be done. If the simulation result is not correct or do not meet the requirements, rather in the behavior-level design or the gate-level design, the design must be redone or revised. After that, we can see the performance of our design.

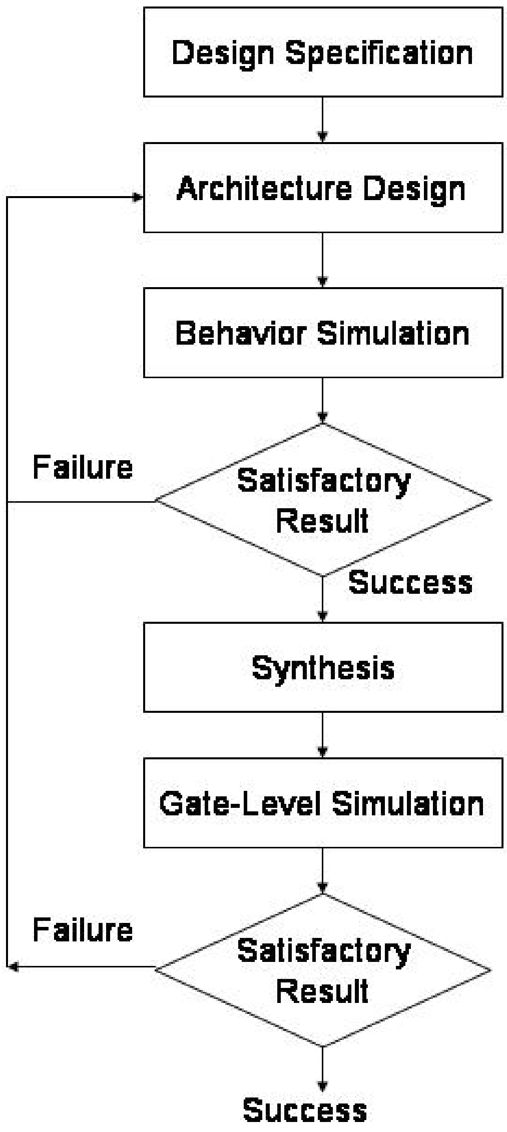


Figure 4.1 The design flow of the our Architecture

4.1 Design Evaluation

In this section, we discuss the area/time tradeoffs that arise for different values of operand precision (*n*), word size (*w*), and number of stages in the pipeline (*p*). The proposed architecture is highly flexible and allows the investigation of several design tradeoffs in word size (*w*) and number of PEs (*p*). Note that in order to simplify the problem, we assume the number of PEs (*p*) is more enough to execute the pipeline data path without extra buffer. There are many kinds of combinations of word size (*w*) and number of PEs (*p*) can implement the same operand precision (*n*). For example, if we want to implement a 256 bits RSA system, we may choose (*w*, *p*) = (8, 17) or (*w*, *p*) = (16, 9) or (*w*, *p*) = (32, 5). However, we do not suggest the word size larger than 16 bits. Because we execute the high clock rate computation, the larger word of final adder may be the bottleneck of clock rate in entire system, and the area of the adder may be very large. Table 4.1 shows the adder' s clock rate and area in TSMC 0.25 μ m process technology. Note that the increase in word size is the only parameter that effects the clock period since the architecture is very modular. Increasing the number of stages (*p*) shouldn't impact the clock period after placement and routing if neighboring modules in the pipeline are kept close to each other.

	Clock Rate(in MHz)	Area(gate count)
8 bits Adder	625	779.95
16 bits Adder	555	1731.53
32 bits Adder	500	3657.60

Table 4.1 The clock rate and gate count of different adders.

4.2 Experimental Result

The performance evaluation presented in this section is based on area and time estimates obtained with TSMC 0.25 μ m process technology and Synopsys Design Analyzer to synthesize our RTL code.

4.2.1 A Single Processing Element

The main architecture of our improved algorithm was discussed in chapter 3. Table 4.2 shows a single 8 bits and 16 bits PE's area of Koc's architecture and our architecture.

	Koc's Design (8 bits)	**Koc's Design (16 bits)**	**Our Design (8 bits)**	**Our Design (16 bits)**
Area (Gate Count)	912.7	1882.55	784.14	1542.35

Table 4.2 A single 8 bits and 16 bits PE's area of Koc's architecture and our architecture

Figure 4.2 shows the symbol view of the single 8-bits PE. The *M*, *Y*, *MB*, *S*, and Cin are inputs of the processing element, and the c1 and c2 represent the two control signal x_i and q_i .

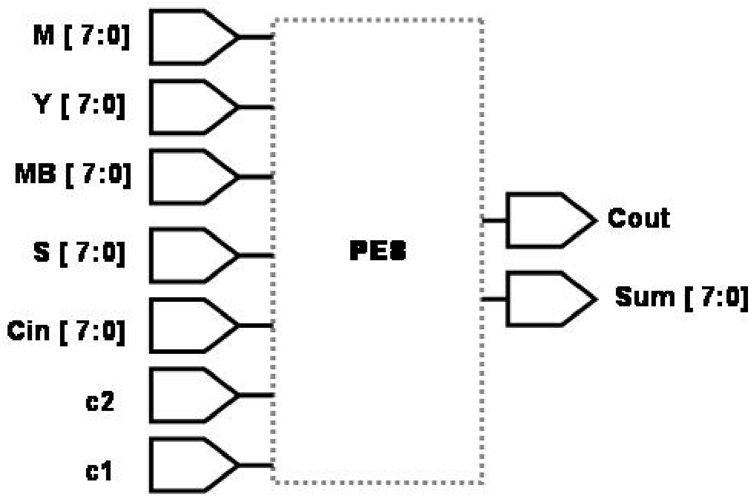


Figure 4.2 The symbol view of 8 bits PE.

4.2.2 Synthesis result

In this section, we will show the speed (MHz) and the area of the original and improved architecture. Table 4.3 and 4.4 shows the area and speed original and improved design with different number of PEs. The results also show the increasing of processing elements will not affect the clock rate. Therefore, we can assume that our architecture can achieve the same clock rate within any number of PE. Table 4.5 and 4.6 shows the speed and area of the original and improved design with different precision of PEs. The results show that the increasing the precision of PEs will increase the clock period time. We then show our implementation and compared the result with the Koc's scalable architecture. Note that the area used by registers for the input operand and modulus was not included in the area calculations.

According to the simulation results, in 8 bits precision, our improved design is 47% faster than the original design with the area is slightly small then original design; in 16 bits precision, our improved design is 38% faster than the original design with the area is slightly small then original design.

Original design (8 bits)	**Speed (MHz)**	**Area (Gate count)**
2 Processing Elements	400	3130.92
4 Processing Elements	400	5385.51
8 Processing Elements	400	10007.65

Table 4.3 The speed and area of the original design with different PEs

Improved design (8 bits)	**Speed (MHz)**	**Area (Gate count)**
2 Processing Elements	588.23	2946.63
4 Processing Elements	588.23	5161.84
8 Processing Elements	588.23	9285.92

Table 4.4 The speed and area of the improved design with different PEs

Original design (8 PEs)	Speed (MHz)	Area (Gate count)
8 bits	400	10007.65
16 bits	312.5	20309.29

Table 4.5 The speed and area of the original design with different precision of PEs

Improved design (8 PEs)	Speed (MHz)	Area (Gate count)
8 bits	588.23	9285.92
16 bits	434.78	18675.31

Table 4.6 The speed and area of the improved design with different precision of PEs

Figure 4.3 shows the symbol view of the design above. The design includes the intermediate registers and the final adder.

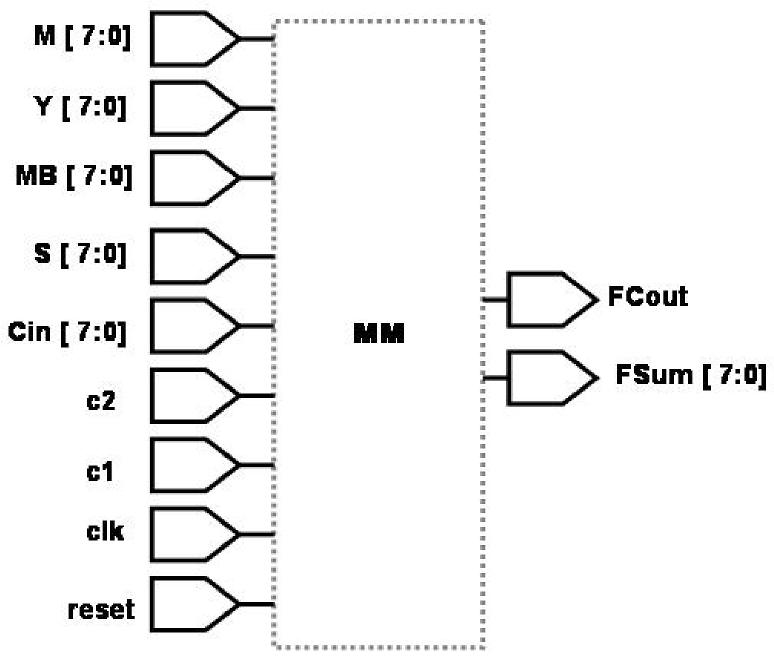


Figure 4.3 The symbol view of the final design.

4.3 Comparison with Other Approaches

The comparison with other hardware implementations of the Montgomery multiplication algorithm is not straightforward since there is no other hardware design that presents the same scalability features. Systolic implementations of the Montgomery multiplier such as the one in [8] are done for full precision of the operands. A systolic multiplier for n = 512 bits consumes about 50K gates and performs the operation in approximately $2n$ = 1024 clock cycles. Our design with a configuration of w = 8 and p = 40 uses an area of 46K and computes the multiplication in 1,103 clock cycles. The design is 8 percent slower using only slightly more than half the area. Besides, the systolic design couldn’ t directly compute with more than 512 bits, while our design could.

Chapter 5
Conclusion

In this thesis, we have proposed an improved architecture of the Scalable Montgomery Multiplication algorithm. We used some methods to enhance the speed (MHz) of the architecture. The new architecture has a 47% speed more than the original architecture, and its area is slightly less than the original architecture. Using TSMC 0.25μm CMOS process technology and specified the design with (*w*, *p*) = (8, 4), the speed is 588.23 MHz and gate counts of a single element is 197k. Because increasing the number of PE does not affect the clock rate, our architecture can achieve the same clock rate within any number of PE. The total time to compute the Montgomery multiplication for a given precision of the operands depends on the kernel configuration. The upper limit on the operands' precision is imposed only by the memory available to store the operands and internal results. In the future, the encryption and decryption system plays an important role in the content protection. We believe that higher speed and scalability must be helpful to content protection technology, and we will continuously dedicate to those goals.

Reference

[1] C¸ .K. Koc¸, T. Acar, and B.S. Kaliski, “Analyzing and Comparing Montgomery Multiplication Algorithms,” IEEE Micro, vol. 16, no. 3, pp. 26-33, June 1996.

[2] E. Savas, A.F. Tenca, and C¸ .K. Koc¸, “A Scalable and Unified Multiplier Architecture for Finite Fields GFepT and GFe2mT,” Proc. Second Int’l Workshop Cryptographic Hardware and Embedded Systems

[3] 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.

[4] M. Shand, J. Vuillemin, “Fast implementations of RSA cryptography,” *Proceedings of 11th IEEE Symposium on Computer Arithmetic*, pp.252-259, 1993.

[5] P. L. Montgomery, “Modular multiplication without trial division,” *Math. Comput.*, vol. 44, pp.519-521, April 1985.

[6] A. Bernal and A. Guyot. Design of a modular multiplier based on Montgomery's algorithm. In 13th Conference on Design of Circuits and Integrated Systems, pages 680-685, Madrid, Spain, November 17-20 1998.

[7] 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.

[8] C.-Y. Su, S.-A. Hwang, P.-S. Chen, and C.-W. Wu, “An Improved Montgomery’s Algorithm for High-Speed RSA Public-Key Cryptosystem,” IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol. 7, no. 2, pp. 280- 284, June 1999.

[9] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In S. Knowles and W. H. McAllister, editors, *Proceedings, 12th Symposium on Computer Arithmetic*, pages 193{199, Bath, England, July 19{21 1995. Los Alamitos, CA: IEEE Computer Society Press.

[10] P. Kornerup. High-radix modular multiplication for cryptosystems. In E. Swartz-lander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedings, 11th Symposium on Computer Arithmetic*, pages 277{283, Windsor, Ontario, June 29 { July 2 1993. Los Alamitos, CA: IEEE Computer Society Press.

[11] C. D. Walter. Space/Time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2):139-141, February 1997.

[12] A. Royo, J. Moran, and J.C. Lopez, "Design and Implementation of a Coprocessor for Cryptography Applications," Proc. European Design and Test Conf., pp. 213-217, Mar. 1997.

[13] A.F. Tenca and C¸ .K. Koc¸, "A Scalable Architecture for Montgomery Multiplication," Proc. First Int'l Workshop Cryptographic Hardware and Embedded Systems—CHES '99, C¸ .K. Koc¸ and C. Paar, eds., pp. 94-108, Aug. 1999.

[14] Colin D. Walter. Montgomery Exponentiation Needs no Final Subtractions. Electronics Letters, 35(21):1831-1832, October 1999.

[15] C. Walter, "Systolic Modular Multiplication," IEEE Trans. Computers, vol. 42, no. 3, pp. 376-378, Mar. 1993.

[16] A.F. Tenca, G. Todorov, and C¸ .K. Koc¸, "High-Radix Design of a Scalable Modular Multiplier," Proc. Workshop Cryptographic Hardware and Embedded Systems, C¸ .K. Koc¸, D. Naccache, and C. Paar, eds., pp. 185-201, 2001.

[17] William Stallings, Cryptography and Network Security Principles and Practice second edition. Prentice hall, 1999.