Excel Journal of Engineering Technology and Management Science
(An International Multidisciplinary Journal)
Vol. I No. 2 January - June 2012 (Online) ISSN 2277-3339

PERFORMANCE ANALYSIS OF MONTGOMERY MODULAR MULTIPLICATION ON EMBEDDED SYSTEMS, IMPLEMENTED USING A FLEXIBLE DATA STRUCTURE IN OBJECT ORIENTED PLATFORM OF C++

* **Rahat Afreen,** *Millennium Institute of Management, Dr. Rafiq Zakaria Campus, Rouza Bagh, Aurangabad, India.*

INTRODUCTION

Security of cryptographic algorithms mostly depends on large data sizes. Storing, retrieving and manipulating such large bit sized numbers mathematically is challenging task in almost all cryptographic systems based on either hardware or software platforms Especially for public key algorithms which use computationally irreversible techniques, that are time consuming. Modular Multiplication is one of such method which is very important in various public key cryptographic schemes like RSA and Elliptic Curve Cryptography. Security of public key encryption scheme depends on the length of encryption/decryption keys. Larger the key, size more secure the system will be. Moor's Law [1] should play an important role while deciding key sizes. According to Moore's law computing power gets doubled every 18 months and hence the power of an attack on a secure system. It means key sizes have to be increased every 18 months, or a cryptographic scheme has to be designed to suffice this requirement for a considerably suitable time. Cryptographic applications should be flexible enough to change the key sizes as and when needed.

Most common public key algorithms like RSA and the emerging elliptic curve cryptography (ECC) – depend on modular arithmetic, that is,

"The multiplication in a finite field like GF(P) or GF(2^m), where the operations are executed modulo a prime number P or a prime polynomial of bit length m, respectively."

Further, Modular Multiplication (MM) is considered as the most critical operation. It is defined as:

*"Given a word length of n bits, an n-bit integer M called the modulus, and two n-bit operands X and Y the problem is the computation of X*Y mod M. the result should be at most n bits wide."*

The typical problem size *n* is rather large (e.g. *n*=160 to 4096 bits). Also, long exponentiation based cryptographic operations are performed infrequently in secure client devices like smart cards. It is not economical to include a large piece of hardware dedicated only to that task Therefore, there is a need for algorithms that are on the one hand fast and on the other hand area and power efficient when implemented.

Modular Multiplication is considered to be slow compared to other operations performed for a cryptographic system. Thus, improving the efficiency of modular multiplication will significantly enhance the efficiency of many cryptographic algorithms. Various implementations of modular multiplication have been proposed over several years to enhance its efficiency. In 1985 Montgomery introduced a new method for modular multiplication. The approach of Montgomery avoids the time consuming trial division that is a common bottleneck of other algorithms. It

enhances the speed of modular multiplication up to five times [2]. His method is proved to be very efficient and it is the basis of many implementations of modular multiplication, both in software and hardware.

Security requirements change vastly depending on various factors. As cryptographic algorithms run very slow, we have to consider the data processing and storage capabilities of the platform. We also need to consider the sensitivity of information that we are encrypting. It is also important how economical a particular scheme is, both from implementation and from manufacturer's point of view. For example cable T.V. set up boxes incorporate some level of encryption which is much lower than the specified standards. The reason may be that currently cable T. V providers don't find themselves at the risk of any serious attack and low level encryption methods are cost effective. But with the growing misuse of networking facilities it is quite possible that this mass mode of communication may fall prey at wrong hands. At that time we cannot upgrade encryption parameters of thousands of set up boxes.

Therefore it is necessary to develop a unified architecture that can perform cryptographic operations at varying level of security requirements. It should provide following key features: algorithm agility, resource utilization and compatibility.

Algorithm Agility

Algorithm Agility allows the user to switch cryptographic algorithms during runtime. Therefore, if current cryptosystems become obsolete due to new security needs, several back-up cryptosystems can be used as a replacement without requiring redesign. Most of the times just switching to a larger key size can help effectively.

This is an important requirement because, The security of most cryptographic algorithms is not proven, but merely presumed to be intractable with currently available computing power. Also, one can never be entirely sure that better methods for cryptanalysis, than those currently known, do not exist. Quite recently a theoretical attack on RSA, based on an improved scheme for factoring integers, has been proposed which, if practical, could render RSA keys of less than 1500 bits insecure.

Resource Utilization

It means to use the same hardware to perform the majority of the arithmetic used for the different supported algorithms. Therefore, it reduces power and area consumption. This is mainly important for resource constrained environment of embedded systems. An efficient software implementation can also help to run the same cryptographic procedures on different platforms.

Compatibility

Compatibility allows interoperability of various applications through a multitude of cryptographic algorithms. The problem stems from the fact that high end platforms may utilize cryptosystems such as RSA, whereas, for smart cards or other embedded applications ECC may be more reasonable choice. It is also required as the common procedure to design a cryptographic system today is to combine public and private encryption schemes together to ensure better security.

A number of different public key algorithms are in use today. To ensure compatibility with the rest of the world, cryptographic applications have to support a large portion of those algorithms. While software implementations are often easy to upgrade and to adapt to new algorithms or larger key sizes, the same is not necessarily true for hardware implementations.

In the software implementation of public key cryptography, we need a method to store, retrieve and perform mathematical operations on very long integer numbers. These numbers are called as multi precision integers [11]. They are represented as consecutive memory words which are brought word by word inside the processor, various operations are performed on them and results are stored back. Performing mathematical operations on them means following problems should be addressed;

- We need to consider propagation of carry and borrow from one word to another.
- Some of the mathematical operations are performed from least significant bit (LSB) to most significant bit (MSB) and some in the reverse direction like division operation; this is a challenge as for ALU arithmetic operations proceed from LSB to MSB. This is the reason why division is time consuming even as machine level instruction and not included into instruction set of most of the RISC processors.
- As same type of operation has to be performed on every data word one by one, we need to use loops and therefore, we have to establish loop boundaries. These boundaries will depend on actual data size, more specifically, the key size when we are generating keys. This will be one of the factors restricting flexible key size during runtime.

To address the first problem we use binary polynomials of the form GF (2^m). For these polynomials, addition and subtraction are equivalent to bit-wise XOR operation [11], hence no carry or borrow will propagate. Montgomery's method helps to solve the second problem as it proceeds from LSB to MSB.

Third problem, at first sight does not look like a considerable problem as we can define some global constant matching our key size and use it inside the entire software. When we need to change or increase the key size, we only have to modify that constant. But actually it adds inflexibility to the system as we need to modify it regularly to support required key size, and according to the processing and storage capabilities of different platforms.

Sensitivity of the information varies from platform to platform and also it varies in different situations for same user. A data may be needed to be kept secret only for a couple of days or for very long time by the same user on same platform. So if it is said that keys should be sufficiently long to protect the data, it also must not be too long to utilize extra computing resources for a less sensitive piece of data.

This paper presents a scheme which is able to perform Montgomery Modular Multiplication independent of the data size, making it more platform-independent. The scheme can be extended to implement various cryptography related operations. The scheme is implemented using class-object concept of C++ on Atmel 8, Atmel 16 and Atmel 128.

METHODOLOGY

Before describing the actual scheme presented in this paper let us first present the Montgomery Modular Multiplication algorithm.

The basic idea behind Montgomery multiplication is the fact that one can add a multiple of the modulus M to the product $A \cdot B$ to yield a result that is at most $2n+1$ bits wide. Adding, instead of subtracting, a multiple of the modulus M does not affect the computation, since the result will be congruent to $A \cdot B$ modulo M. Two numbers are said to be congruent if their remainder when divided by the modulus is the same. Thus, $A \cdot B$, $A \cdot B + M$, $A \cdot B + 2M$ ... $A \cdot B + kM$ are all congruent modulo M. This implies: $A \cdot B \equiv A \cdot B + M \equiv A \cdot B + 2M \equiv \ldots (A \cdot B + kM) \bmod M$. In the Montgomery algorithm, the multiple of the modulus M that is added to $A \cdot B$ is chosen in such a way that the lower n-bits of the $2n+1$-bit result are all zeroes [3]. The least significant half of the $2n+1$-bit result that are all zeroes is then discarded. This way, the result would have been reduced to at most $n+1$ bit in width. A single subtraction of the modulus M can then be performed to further reduce the result to at most n-bits and make it less than M if required. It has been shown by Walter [4] that the extra subtraction may not be necessary under certain conditions. Specially it will not be needed when we are using binary polynomial of the form $GF(2^m)$.[7]

Algorithm : Montgomery multiplication

Inputs: A, B, M with $0 \le A, B < M$, Output: R = Montgomery Product $(A \cdot B 2^{-n}) \bmod M$

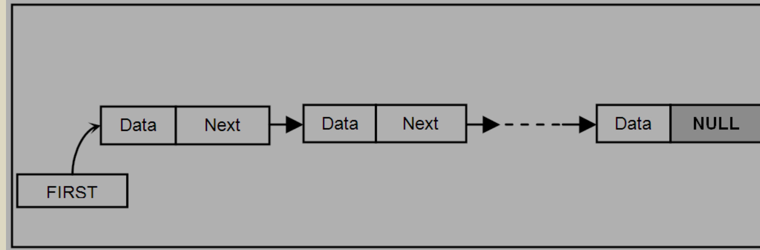
```
R = 0;
For i = 0 to n-1 do
Begin
R = R + a_i·B;
R = R + r_0·M;
R = R div 2;
End
if R ≥ M then R = R - M;
```

Montgomery multiplication is easier in binary [5]. The key simplification is that the adding of the multiple of the modulus becomes much easier. The rule is this: if the number you are looking at is odd, add R before you halve it; if it's even, just halve it by shifting it one bit to the right. In each iteration of the loop, the least significant bit of the intermediate result is inspected. If it is '1', i.e. the intermediate result is odd; we add the modulus M to make it even. This is possible since M is guaranteed to be odd in the cryptographic applications of interest. Thus, at each step the intermediate result is made to be even. This even number can be divided by 2 without any remainder. This division by 2 reduces the intermediate result to $(n+1)$ bits again. After n steps these divisions add up to one division by 2^n, or discarding the least significant n-bits that are zeroes.

System Design

This system is designed to perform Montgomery Modular Multiplication on linked lists containing eight bytes per node as data and a pointer to next node.

This is a one way linked list which can be represented graphically as below



One Way Linked List of 8 bytes per node

The end of list is specified by a NULL value to the pointer next. The list will also need one more pointer that will point to the beginning of the list. It is defined as a class data member. Using the structure explained above a C++ class longint is defined as below

```
Class longint {
Protected:
byt * first;
uint16_t size;
public:
longint () {first = NULL; size=0;}
// list of other functions.
};
```

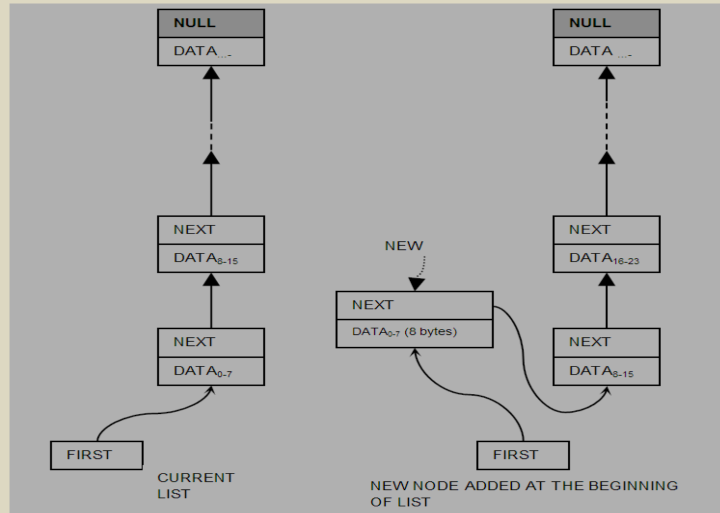
A pointer 'byt' is defined as first which points to the beginning of the list. It is initialized as NULL (ASCII equivalent 00H) to represent list is empty. The 16 bit long variable size will hold the count for number of bytes in the presented structure. Thus we can accommodate upto $2^{16}*8$ = 5,24,288 bits of integers using this scheme. This is sufficiently large than current key size of 2048 bits for RSA or160 bits for ECC. As size is a class variable, varying byte sized data objects can be defined using same class type and each object will have its own byte length as its property.

Class Member Functions:

Functions help to aid a conceptual organization of a program. It is one of the major principals of structured programming. They also help in reducing program size [6].

Class member functions are always called to act on a specific object, not a class in general. This means, to perform an operation on a class, there should be an occurrence of at least one object of that class hence accidental misuse is not possible. The member functions defined for 'longint' class are finalized in such a way that they are highly task specific. For example, from algorithm 1 we can observe that in first two steps where addition is performed, one of the operands in RHS is equal to LHS. Thus while implementing addition on prime field, only one operand is passed as function parameter assuming Result R as object on which addition is performed. As result is stored back in R, no return object is specified. This was possible only because C++ is used for implementation.

Following graphical representation shows how a new linked list is created;

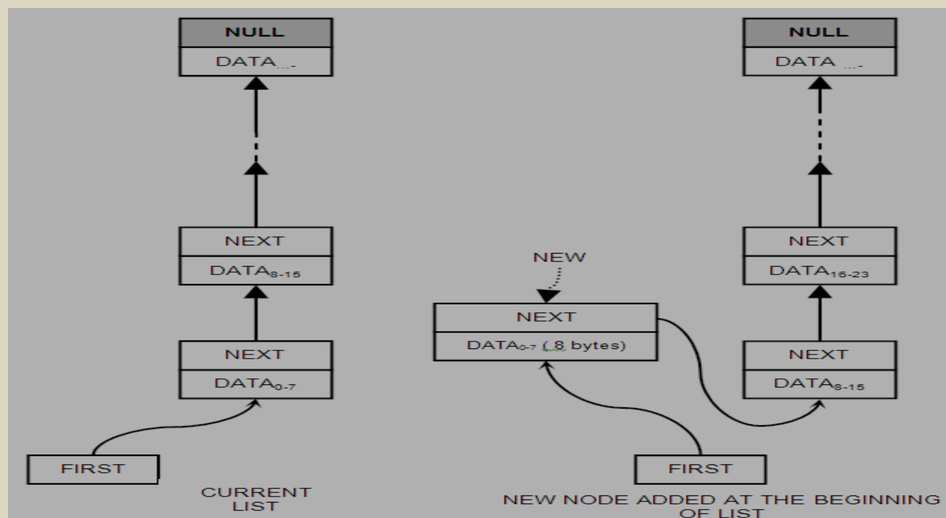


creating an object of type 'longint'

Adding a new set of data is always performed at the beginning of list. That is data is added from MSB to LSB ensuring that first byte accessible containing least significant bit is available at the start of the list. Considering MSB first while storing the data and considering LSB first while performing basic operations is similar to day to day arithmetic operations.

For example, consider a six digit decimal number 635298 it is natural to start writing from MSD to LSD either in numerical format or in words (six lacs thirty five thousand two hundred ninety eight). But when we want to perform some arithmetic operation say 'addition' on numbers we start from LSD eg: 286 + 154 we add 6 & 4 first, and then 8 & 5 with carry and so on.

But while performing arithmetic's, result of least significant byte is generated first and so on. When they will be stored back, a reverse list will be generated. However it will not be the case with our implementation as we have implemented addition as per the Montgomery Algorithm. It will not generate any new list, rather, it will store the result in the object on which addition function is called. However a function reverse() is also defined to reverse the list. Its working is shown below.



Working of function reverse()

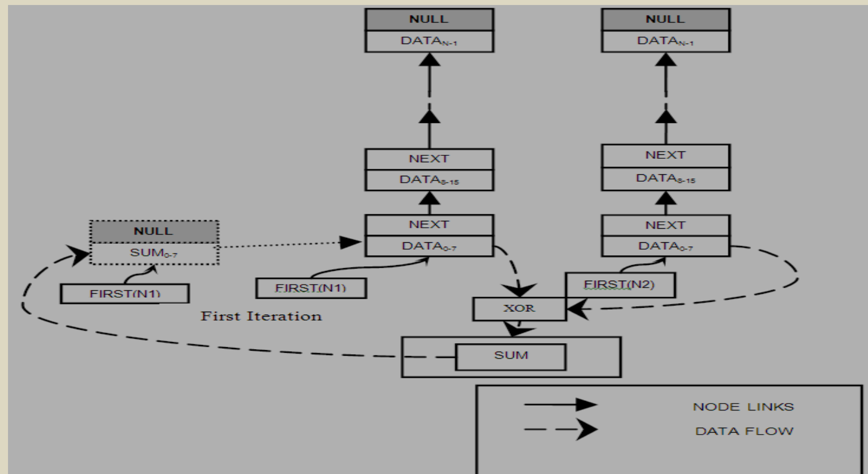
Reversing a unidirectional linked list is very simple. You have to just take out nodes one by one and allocate it to a new pointer to data structure 'byt'. First byte taken out will be LSB. When it is added into new list it becomes MSB and list is reversed. This method has a drawback that at the end of each arithmetic operation, we have one extra function call and one extra list is created every time. Another alternative approach can be used by defining a temporary pointer of type byt during the arithmetic operation. This pointer can hold the address of previous node of the result generated. So, after performing any arithmetic operation on a set of eight bytes, the result can be appended at the end of previously generated result.

Arithmetic operations for the field GF(2^m):

As this implementation is considering binary polynomial of the form GF(2^m). For this representation, addition and subtraction are equivalent to bit-wise XOR operation. To differentiate that arithmetic operations are performed modulo a binary prime field, a new class prime_fld is derived from the base class longint as followes;

```
class prime_fld :public longint
{
protected:
uint16_t DEGREE;
public:
add(prime_fld);
prime_fld mlt(prime_fld);
};
```

The unsigned integer data type DEGREE represents the degree of binary polynomial. That is, total number of bits in the binary number of the form GF(2^m).Addition operation is implemented as per the following figure;



Bitwise xor of two varying sized linked lists.

Modular multiplication is implemented using the algorithm described previously. This function takes two parameters of type prime_fld, which are considered as A & B of modular multiplication. The reduction polynomial M is defined globally and modular multiplication is called upon any one of the operands A while another operand is passed as parameter. The function returns R as result of multiplication.

R = A. mod (B); // M being defined globally.

All the steps of Montgomery Modular Multiplication are implemented as follows

Step1:$R = R + a_i.B$

On each iteration i^{th} bit of parameter A is checked, if it is 1 then only the function add is called to add R & B

R .add (B)

Step2: $R = R + r_0 M$

0th bit of R resulting from step1 is checked. If it is 1 then function add( ) is called as

R.add (M) ;

Step3: R=R div 2

function shift () is called as shifting right by one bit is equal to divide by zero

R.shift( );

RESULTS AND DISCUSSIONS:

Following table shows initial storage requirement on various platforms, calculated independent of data length.

STORAGE REQUIREMENT IN BYTES

Device	**Data Memory-(Bytes)**		**Code Memory-(Bytes)**	
	For initial conversion to class object	***For Modular Multiplication***	***For initial conversion to class object***	***For Modular Multiplication***
Atmega 8	34	16	1396	1910
Atmega 128	34	16	1524	2058
Atmega 16	34	16	1464	1998

Object oriented platforms are still rarely considered for embedded systems specially while implementing cryptography related operations because of their restricted storage and processing capabilities, but this implementation shows that Atmel chips can support object oriented platform of C++ effectively, even for complex cryptographic operations. The main concern during this implementation was to check whether proposed scheme is suitable to perform arithmetic operations on multiprecision integers of varying sizes. The scheme was implemented successfully and results were checked of basic arithmetic operations of addition, binary shift left and right, reversing a number and for multiplication using Montgomery's method.

Generally we perform arithmetic operations on the data with same word length. In this proposed scheme, we can multiply numbers of different word length. If the size of result is greater than word size of operands, a new node will be defined to store extra bytes of result.

The same scheme was proposed earlier [8], with one byte per node for Pentium based processors. This scheme was so flexible that we can multiply two byte sized integers and get a word sized result without any truncation of MSB bits. Regular addition and subtraction was also implemented in that scheme considering carry and borrow. Inline assembly was used for that purpose. Using inline assembly is also the most preferred option when programming on embedded platforms. It optimises the code size and data size of resulting program.

This implementation does not consider speed optimisation but since Montgomery's method is very famous, a variety of schemes are available to optimise the performance like [7] [9] [10] etc, for software as well as hardware implementations. The software optimisation methods can be implemented taking the advantage of class –object platform used here.

However following table shows time (in Microseconds) of various operations for three standard binary polynomials B163, B233 and B283 respectively. When calculated at the frequency of 8 Mhz.

COMPARISION OF SPEED FOR VARIOUS OPERATIONS

Operation	**Initial parsing(µs)**			**Convert to longint (µs)**		
Data size	***B163***	***B233***	***B283***	***B163***	***B233***	***B283***
Atmega 8	1156.75	2065	2415	236.25	314.5	390.75
Atmega 128	1008.00	1687	1996.5	238.25	316.5	393.25
Atmega 16	589.5	1687	1996.5	200.53	316.5	393.25
Operation	**Add/Subtract in GF(2^m) (µs)**			**Shift Right (µs)**		
Data size	***B163***	***B233***	***B283***	***B163***	***B233***	***B283***
Atmega 8	480.63	619	765.37	76.5	106.88	128.28
Atmega 128	477.62	625.5	773.38	76.5	106.88	128.25
Atmega 16	485.62	625.5	773.38	76.5	106.88	128.25

The speed of modular multiplication will depend on number of 1's in the operand A and at the LSB of intermediate result. Every time we get a 1 at the ith bit of A, in first step and at 0th bit of intermediate result addition function will be called. The table shows that there is a mismatch of time for initial parsing, which is implemented as a group of independent functions apart of the proposed class structure.

CONCLUSION:

A linked list data structure is presented in this paper that can accommodate any integer independent of its length. This scheme is suitable for multi precision integers that are used in cryptography. We can easily perform various arithmetic operations on multi precision integers as proved by implementing Montgomery Modular Multiplication for binary polynomials of the form GF (2^m). This scheme is also efficient if operands of any arithmetic operation are of varying length. The scheme is implemented on the embedded platform using Atmega AVR based microcontrollers and results are analyzed. It can be observed from the table 2 that time taken for various operations remains almost constant on selected platforms but it depends on length of data.

REFERENCES

1. A.S. Tanenbaum, " Structured Computer Organization", Fifth Edition
2. P. Montgomery, Modular Multiplication without Trial Division. Mathematics of Computation, vol. 44,no. 170, April 1985, pp. 519-521.
3. Moboluwaji O. Sanu, Earl E. Swartzlander, Jr. and Craig M. Chase "Parallel Montgomery Multipliers", Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04) pp1-10 1063-6862/04.

4. C. D. Walter, “Montgomery Exponentiation Needs No Final Subtractions”, *Electronics Letters*, vol. 35, no. 21, pp. 1831-1832, 1999.
5. http://www.nugae.com/encryption/fap4/montgomery.html,“Montgomery multiplication: a surreal technique”
6. Robert Lafore, “Object Oriented Programming in C++”, Third Edition
7. C¸ETIN K. KOC¸ AND TOLGA ACAR “Montgomery Multiplication in GF(2^k)” in Designs, Codes and Cryptography, 14(1), 57–69 (April 1998)
8. Khan Rahat Afren “A linked List implementation of Montgomery Modular Multiplication” Dissertation submitted for the Degree of Master of Engineering in Computer Science and Engineering, 2007-08, Dept. of Computer Science and Engineering, Govt. College of Engg. Aurangabad, M.S.
9. C. D. Walter, “Montgomery Exponentiation Needs No Final Subtractions”, *Electronics Letters*, vol. 35, no. 21, pp. 1831-1832, 1999.
10. Jean-Claude Bajard, Laurent Imbert, and Graham A. Jullien “Parallel Montgomery Multiplication in GF(2^k) using Trinomial Residue Arithmetic”, University of Calgary, Canada
11. Darrel Hankerson, Alfred Menezes, Scott Vanston “Guide to Elliptic curve cryptography”, Pp 47-50, Springer