Large Number Multiplication in Cryptographic Systems

By: Abdulla Fouad Bubshait
May 2001

Prof. Ruby Lee

Introduction

Large number multiplication is used heavily in number theoretic cryptographic applications. In particular, it is heavily used in public key cryptography. With the growth of the Internet the demand for cryptosystems has risen. They provide secure communication over insecure channels such as the Internet. Some protocols that depend on large number multiplication are the following:

- RSA public key encryption
- El-Gamal public key encryption
- Diffie-Hellman Key Exchange
- DSA public key signatures

Multiplier Choice

The selection of the multiplier unit to be used for big number multiplication depends on several factors. First the choice of symmetry is discussed, then the analysis of size selection for the multiplier are brought up to detail which multiplier configuration would be optimal for large number multiplication.

Symmetric vs. Non-Symmetric.

The choice of multiplier configuration is an important one in large number multiplications. Since it is the building block used to perform the multiplications. The choice of using a symmetric multiplier vs. a non-symmetric one is analyzed here.

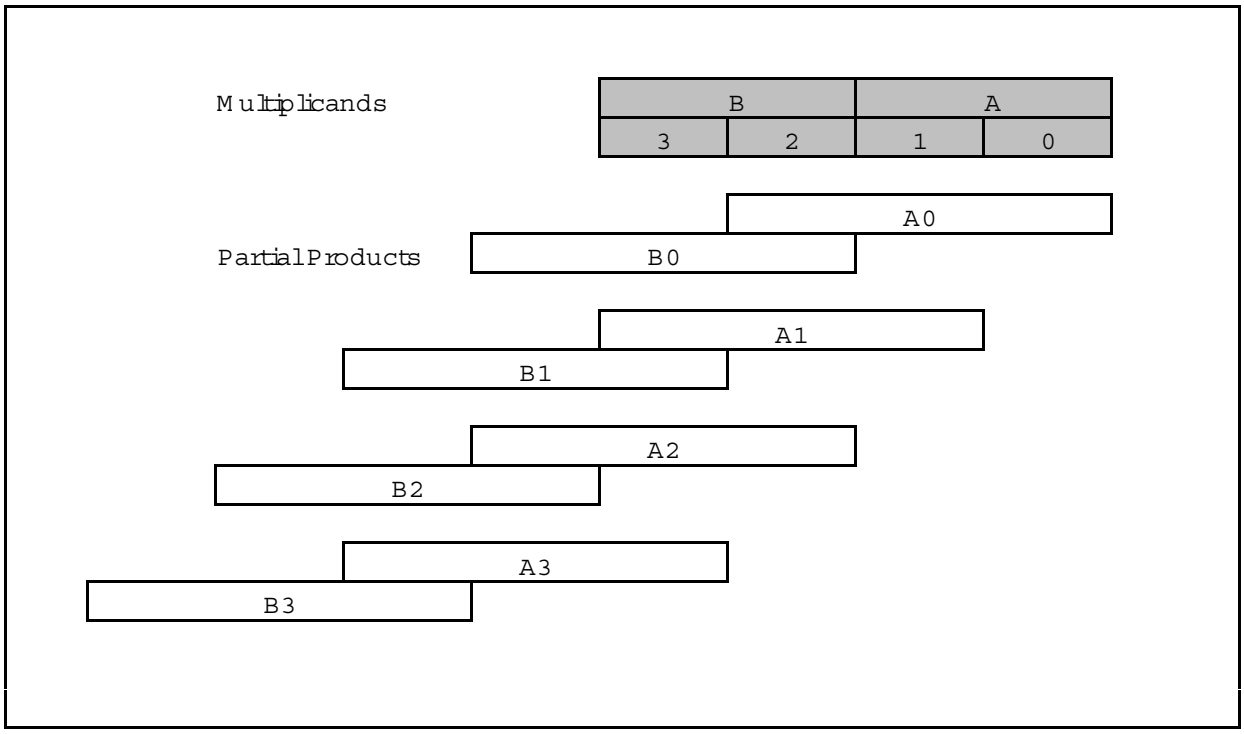


Figure 1 - Non-Symmetric Multpliers

The non-symmetric multiplies has 3 intrinsic problems that make it unfavorable for performing large number multiplications.

1. The partial product configuration makes row carry propagate difficult.

   One of the schemes that are used to sum the partial products is to propagate carries across the rows of partial products. In order to facilitate this method, the partial products must be organized in a specific manner. The organizational scheme is shown graphically in the following figure.

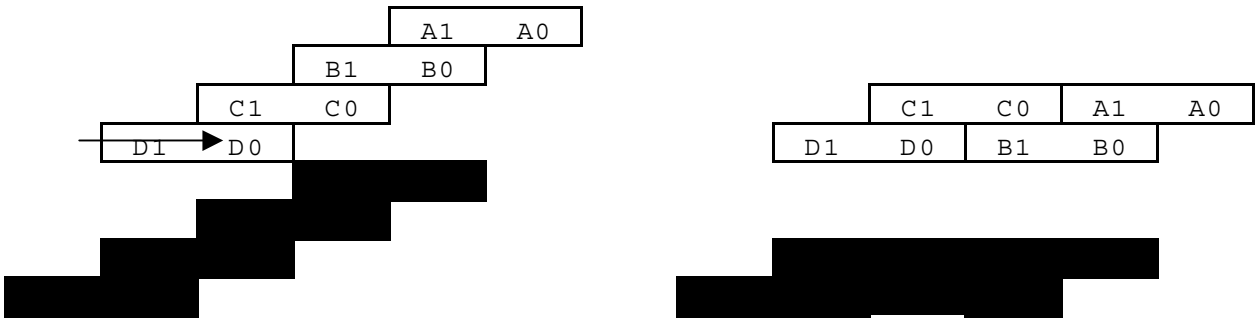


Figure 2 - Symmetric Partial Product Organization for row carry propagate

2. The partial product configuration makes column accumulate difficult.

   The other method of summing the partial products is the column accumulate method. This method splits the partial product into two halves, a sum and a carry. The sum half becomes the output of the column while the carry half is propogated to the next column. The reason for this split is because it lies on the column boundaries. In Figure-1 it is clear that the partial product cannot be easily split, since its halfway point does not lie on the column boundary.

   This means that the efficient techniques for summing partial products of symmetric multipliers cannot be used to sum the partial product of non-symmetric multipliers.

3. The size of the partial products is irregular.

   The product of a non-symmetric multiplier is difficult to manipulate and organize. The use of subword permutations provides significant performance benefits, especially in ordering and packing of partial products to perform summations. Subword permutations act on elements of 8,16 or 32 bits. The irregular size of the partial product means that they do not lend themselves to be manipulated by subwords permutes, and so requires more calculations in order to be reorganized.

For these reasons the Non-Symmetric multipliers are deemed inappropriate for large number multiplication.

Size Analysis.

Other than the symmetry, the decision of the size of multiplier used is also important. There are several factors that play a role in the choice of multiplier, these factors include speed, size and power consumption. In order to give a better understanding on how these factors depend on the size of the multiplier, the following tables provide information on multipliers based on several different designs.

Table 1

Speed [MHz] / *Multiplier type*	*8×8*	*12×12*	*16×16*	*24×24*
Basic Array Multiplier	23.680	13.752	9.637	4.299
Array M. with CLA Addition	24.420	13.265	10.549	5.124
Array M. with Carry Logic Add.	29.772	16.713	12.582	6.448
Booth algorithm	22.158	15.427	12.990	5.368
Log Tree Multiplier	30.662	22.666	22.732	13.650
Synthesised Mult	26.843	19.571	13.133	5.619
Single Pipeline Mult. With Carry Logic Add.	42.143	32.350	20.933	10.017
Double Pipeline with Carry Logic Add	61.679	39.029	21.675	10.510

Maximum operating freq. for the different multiplier algorithms tested

The previous table provides the speed comparison for the multipliers. To perform on 16x16 multiplication with an 8x8 multiplier requires 4 sepereate multiplications extra addition and arranging instructions to achieve the result. So while smaller multipliers are faster, for larger numbers, the smaller multipliers will require a longer time to perform the total multiplication.

Table 2

Size [number of CLBs] / *Multiplier type*	*8×8*	*12×12*	*16×16*	*24×24*
Basic Array Multiplier	60	139	243	554
Array M. with CLA Addition	58	138	243	590
Array M. with Carry Logic Add.	56	134	237	544
Booth algorithm	80	177	303	858
Log Tree Multiplier	59	133	217	478
Synthesised Mult	57	130	230	534
Single Pipeline Mult. With Carry Logic Add.	58	135	239	550
Double Pipeline with Carry Logic Add	61	139	243	555

FPGA Area occupied by the different multiplier algorithms tested

The multiplier size increases exponentially as the input size increases.

Table 3

Dynamic Power Cons. [mW] / *Multiplier type*	*8×8*	*12×12*	*16×16*
Basic Array Multiplier	198.75	695.79	695.60
Array M. with CLA Addition	196.77	668.86	652.22
Array M. with Carry Logic Add.	197.52	639.57	673.29
Booth algorithm	288.95	990.31	975.60
Log Tree Multiplier	162.76	500.87	478.82
Synthesised Mult	230.90	851.31	856.77
Single Pipeline Mult. With Carry Logic Add.	164.48	432.44	485.89
Double Pipeline with Carry Logic Add	155.42	417.33	444.10
P_{other}	368.57	366.73	358.33

Dynamic Power consumption for the tested multiplication algorithms

While it is apparent that larger multipliers perform large number multiplication more efficiently, it is not feasible to arbitrarily increase the multiplier size, this is clearly apparent from the power consumption and size tables. So, from among the feasible choices the largest symmetric multiplier will be the most advantageous to use in large number multiplication.

To perform multiplications of number as large as 1024 bits requires a lot of computation. While it could ideally be performed in one step using a 1024x1024 bit multiplier, we have shown that such an approach is not practical. Feasible multipliers are of sizes 64x64 or 32x32. So it is imperative to be able to perform large multiplications using these small multipliers. Methods of performing such multiplications are discussed in the next section.

Large Multiplication

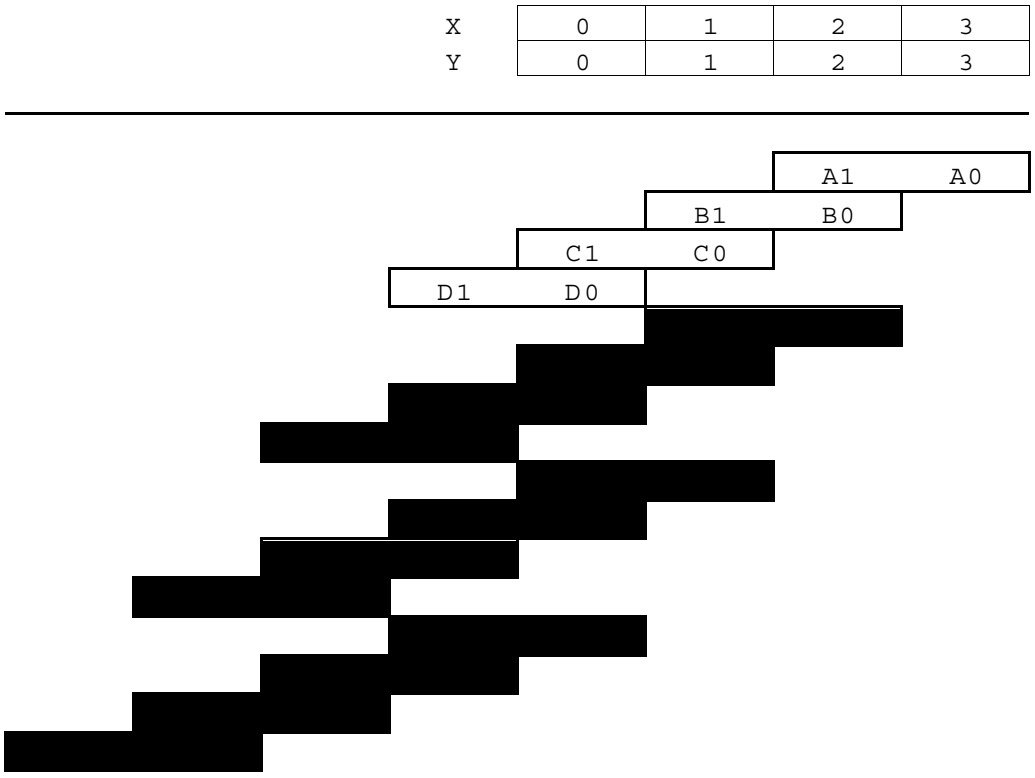
Assuming we have an NxN multiplier, in order to multiply two numbers of lengths NM would require M^2 multiplications. Each multiplication outputs a result of length 2N, this result is the partial product of the multiplication. These partial products have to be summed in an efficient manner to produce the final output.

Methods to sum partial products

Once the partial products have been generated using the NxN multiplier, the next step is to sum the partial products in an efficient manner. As the numbers being multiplied increase in size on the order ‘B’, the number of partial products increases by B^2, and so to does the difficulty of summing these partial products. Described below are two different methods of adding the partial products in large number multiplication. Their performance is analyzed and finally both methods are compared against each other.

Carry Follow

One of the difficulties of summing the partial products arises from following the carry that is produced when summing the partial product. The following implementation is an in-order execution that propagates the carry across the rows of partial product. The execution is in order because the technique uses a carry flag, so when the sum produces a carry out, a carry in will be added to the next addition instruction performed. The row propagation algorithm will be explained with the following example.



Assuming an NxN multiplication unit, and a 2N addition unit, we have the partial products produced by the multiplication stage shown in the figure above. Each partial product is of length 2N. The row carry algorithm is explained in the first row of partial products. The 2N addition unit is aligned with the start of the row, and the two registers that are added are the following:

R1	A1	A0
R2	B0	

In order to correctly propogate the carry, the next addition should be on elements that were to the direct left of the values in the previous addition. This makes the next two values to be summed the following:

R1	D0	B1
R2	C1	C0

The summation moves again to the left by 2N, and the registers added for this step are:

R1		D1
R2		

The last addition will have either one or three elements, but the result will never have a carry out. This completes the summation of the row with proper carry propagation. These steps are repeated for the remaining rows.

There is some overhead involved using this scheme, which stems from arranging the data for the addition. This requires shifting and packing of data in the registers in order to achieve the proper arrangement for the addition.

Once all the rows have been summed, the summation is performed on each pair of rows. This summation is done in a similar manner –in order- and repeated until the result is generated.

The performance of this method is measured in the following table. It details the number of calculations needed to process each row, and all the calculations needed to combine the rows and get the final result. It also indicates the number of extra registers needed to perform calculations of size 256, 512 and 1024 bits. The calculations were made based on 32x32 multiplications to achieve the partial products. The register size is 64 bits.

Table 4 - Performance of Carry Follow Method

Input Size		256	512	1024
Mux	Rows	32	128	512
	Combine	96	512	2560
	Total	128	640	3072
Addition	Rows	32	128	512
	Combine	96	512	2560
	Total	128	640	3072
Registers	Total	8	16	32

Overflow Prevention

Another method for addition that requires less overhead than the previously described carry propagation is a column based accumulation scheme that prevents overflow. Again, assuming NxN multiplication, we final output is split into segments of size N, each segment is associated with a column. Each partial product is placed into its corresponding column and split into two parts, the lower N bits are the sum, while the upper N bits are the carry. Each is stored in a register of size 2N. The scheme is detailed in the following example.

Multiplicands		D	C	B	A
	Y	3	2	1	0

Partial products							
			D0	C0	B0	A0	
		D1	C1	B1	A1		
	D2	C2	B2	A2			
D3	C3	B3	A3				

Output	7	6	5	4	3	2	1	0

For column 2, each partial product (of length 2N), is split into a sum and a carry, each of length N. This is done for partial products, C0, B1 and A2. The sum and carry portions of all the partial products in the column are added together. Then the columns are traversed in order from 0 to 7. For each column, the upper N bits of the sum are added to the carry, the carries of the column are then summed and propagated to the next column.

The strength of this method comes from the fact that N bit numbers are added using 2N bit addition. This method insures that no overflow ever occurs in the registers. So, while the previous technique requires that the additions be preformed in order, so as to follow the carry bit, this method accumulates the carries generated in the upper part of the 2N bit holder. The benefit of this is that it saves on the overhead needed to arrange the data, the

downside however, is that a larger number of registers are needed to represent the partial products, since each partial product is stored in 4N bits.

The following table details the number of calculations required to perform this method of partial product summation. The number of calculations was measured for 256, 512 and 1024 bits, assuming that 32x32 multiplication was used and each register holds 64 bits. The number of extra registers over those needed to store the final result and partial sums were also calculated.

Table 5 – Performance of Overflow Prevention method

Input Size		256	512	1024
Unpack	Partial Products	64	256	1024
	Carry	30	62	126
	Total	94	318	1150
Addition	Partial Products	128	512	2048
	Carry	30	62	126
	Total	158	574	2174
Registers	Total	30	62	126

Comparison

The advantage of the carry follow method is that it doesn't require a lot of additional resources, namely extra registers. The additions are performed in order, while some sub word permutations are required to properly align the partial products for additions. It can be clearly seen that this method does not provide the highest benefits in term of number of calculations.

The advantage of the overflow prevention method is that it does not require following the overflow bit. This means that the additions do not need to be performed in order. A much higher degree of instruction level parallelism can be achieved since additions can be performed in order. Another advantage is that this method provides improved performance overt the carry follow method. This comes at the price of added resources, namely this method requires a larger number of registers.

The following table summarizes the number or permutation instructions, number of additions, and number of registers required by each method. Assuming and NxN multiplier, and registers are able to hold 2N bits of data. The numbers being multiplied are of length MN.

Method	Carry Follow	Overflow Prevent
Number of Additions	$(Log_2(m) + 1) * m^2/2$	$2*(m^2) + (2*m - 1)$
Number of Mux/Unpack	$(Log_2(m) + 1) * m^2/2$	$m^2 + (2*m - 1)$
Number of Registers	m	$4m - 2$

Modular Multiplication

Encryption methods such as RSA use modular arithmetic, specifically modular exponentiation. Modular exponentiation requires performing large number multiplication as well as division to be able to represent numbers in modular form. The division, however, is extremely computationally intensive, much more so than multiplication. For this reason, a method used for efficient RSA encryption and decrytpion requires a technique to efficiently handle the division required in modular arithmetic.

Montgomery's Algorithm

A method of calculating modular multiplication without trial division was published by Peter Montgomery on April 1995. This method circumvents the computationally intensive division step in modular exponentiation, and so is of great benefit. When dealing with multiplication mod Q the algorithm defines an alternate representation of the integers (0,1,2,...Q-1). We translate normal integers to Montgomery representation, do our multiplications with this new representation, then translate back to the normal representation. In the case of modular exponentiation we perform all the successive multiplication steps in the Montgomery representation before converting the answer back to the normal representation. The following is a detailed description of the algorithm.

To form the Montgomery representation, we choose R, which is greater than Q and relatively prime to Q. Since division by R has to be inexpensive, and since the crypto modulus Q usually has not small factors we choose R to be some power of 2 a little bigger than Q. Division by any power of two is easy, since we just chop off low order bits. We also define R' and Q' as the following:

$$R' = R^{-1} \bmod Q$$
$$Q' = -Q^{-1} \bmod R$$

Now to represent the integers in 0..Q-1, we call the Montgomery representation of x as:

$$M(x) = xR \bmod Q$$

And the inverse representation is just:

$$M'(x) = xR' \bmod Q$$

Montgomery defined the procedure REDC(x) as the following:

function REDC(x)
$m = (x \bmod R) * Q' \bmod R$
$t = (x + m*Q) / R$

if t < Q
return t
else return Q - t

Note that the division in this procedure will always be possible since,

$$MQ = ((x \bmod R)Q' \bmod R)\ Q = x\ QQ' \bmod R = -x \bmod R$$
$$x + Qm = x + -x \bmod R = 0 \bmod R$$

So to perform a multiplication of two numbers in Montgomery representation (x,y) to get a result also in Montgomery's representation (z) we use:

$$Z = REDC(X*Y)$$

Proposed Algorithm

Mongomery's Algorithm provides a way to perform modular multiplication in a method that requires three multiplication instead of a multiplication and a division. There are other benefits that could be extracted from the fact that the multiplication is now done mod R where R is a power of 2. The algorithm will be explained in an example. Suppose X and Y are two large numbers in Montgomery's representation, to perform their multiplication we follow these steps.

*m = (XY mod R) * Q' mod R*
*t = (XY + m*Q) / R*
if t < Q
return t
else return Q - t

There are three multiplications and the optimizations for each will be detailed.

1- First Multiplication: XY
This multiplication is performed completely, and the lower order bits are taken to get a result that is mod R.
2- Second Multiplication: *(XY mod R) * Q'*
Not that this entire multiplication is mod R, so the full multiplication is not needed. Suppose the number eing multiplied are slightly smaller than R. That would mean the number of partial products required would be (N*(N+1)/2) instead of N^2, so roughly half the multiplications can be ignored.
3- Third Multiplication: *m*Q*
This multiplication is devided by R, which means the result will be right shifted. Here ignoring the lower order multiplications can perform the optimization. The choice of which multiplications can be ignored in this

case is slightly more complicated, since some multiplications provide carries that propagate to the upper half of the result. So the upper V column of the lower half has to be calculated, where V depends on the size of the multiplier and the size of the multiplicands. The carry produced by the addition instruction only exists if there is a 1 in the lower half of XY, since the addition is sure to generate a number divisible by R.

Conclusion

Large number multiplication is a difficult computational process. The source of the difficulty one can practically only use small multipliers (around 64x64) to perform multiplication of large number. This brings up the task of efficiently summing the partial products produced. Two methods have been discussed in this paper and the overflow prevent scheme seems most efficient for fast multiplication.

In modular exponentiation Montgomery's algorithm becomes very useful as it converts difficult division into two multiplication steps. This algorithm also has potential to reduce the number of multiplications required for large number multiplication.

References

1. **Dan Zuras, "More on Squaring and Multiplying Large Integers"**
2. **Alvaro Bernal, Alain Guyot, "Hardware for Computing Modular Multiplication Algorithm"**
3. **David Pearson, "A Parallel Implementation of RSA"**
4. **DSP Labs: http://www.ucc.ie/ucc/depts/elec/postgrad/Fabio/dsplab_res.htm**