A New Method of Serial Modular Multiplication

Martin Kochanski

ADDRESS: Cardbox Software Limited, Scriventon House, Speldhurst, Kent, TN3 0TU, England. `mjk@cardbox.com`

ABSTRACT: A novel serial algorithm is described for modular multiplication of large integers and a proof is given of its correct behaviour. Comparisons are made with published designs by Brickell [1] and Montgomery [2]. The new algorithm is advantageous where modular multiplications need to be performed individually and not chained for exponentiation.

KEYWORDS: Fast multiplication, modular multiplication, carry-save arithmetic, public-key encryption.

Introduction

Many modern cryptographic algorithms rely on modular arithmetic where the modulus is large (of the order of 512 to 1024 bits): notably multiplication and exponentiation. Doing this arithmetic in software is slow and so it is important to find efficient hardware implementations.

The fundamental problem with division and modulo reduction is that they involve conditional operations that depend on the result of the immediately preceding operation. For example, one step in the natural implementation of kindergarten long division in hardware is to subtract the divisor from an accumulator and to store the result of the subtraction if it is non-negative. To check for non-negativeness effectively means to check the value of the most significant bit of the result, and this in turn means waiting until a possible carry has had time to propagate from the least significant to the most significant bit. Even with the available techniques for predicting when carries will occur, this leads to a tenfold slowing of the subtraction and thus of the multiplication as a whole.

For this reason some alternative algorithms have been proposed, of which the most often cited are Brickell's [1] and Montgomery's [2]. Montgomery's algorithm requires translation from conventional integers into a different number representation at the start and both algorithms require translation back into conventional integers at the end of each operation.

The present algorithm uses a simpler arithmetic unit than Brickell's (requiring approximately half the silicon) and unlike Montgomery's algorithm it delivers its result directly as a binary integer without requiring further conversion. In circumstances where modular multiplication rather than exponentiation is required, these are significant advantages. In addition, the algorithm's structure is simple and the proof of its correctness is straightforward (compare the complexity of the proofs in [5] and [6]).

The core of the present design is an arithmetic unit (AU) which consists of an array of simple cells implementing a carry-save adder, plus a control unit (CU) which examines the result of the calculation so far and sends appropriate control signals to the AU.

Carry-save integers

When two binary digits are added, their sum can be 0, 1, or 2. In normal binary addition 2 is written as 0 and a *carry* of 1 is then added to the next digit of the sum (reading from right to left). If this digit is itself 1, it becomes 0 and the carry is added to the next digit on the left. In certain cases this *ripple carry* process can go through every digit of the sum, and having to allow for the possibility of a ripple carry greatly slows down the speed at which addition can be performed.

In a carry-save design, the carry is stored and not passed on to the next digit. It is thus necessary to have two bits of storage per digit instead of one but the time penalty of ripple carry is avoided.

We shall follow the general convention of numbering long binary integers from the least significant digit so that, for instance, X_0 is the least significant digit of X, X_1 is the second least significant, and so on.

AU design

The AU contains an accumulator, denoted by ACC, which contains a number that is represented by a pair of registers S and C. We adopt a convention where the weights of correspondingly numbered bits of S and C are equal: thus S_4 and C_4 both have a weight of $2^4=16$. This identification makes it easy to find the number represented by ACC: simply evaluate S + C, treating both S and C as ordinary binary integers.

In its normal carry-save mode, the AU implements the following function:

$$ACC' = 2 \times ACC + jX + kR$$

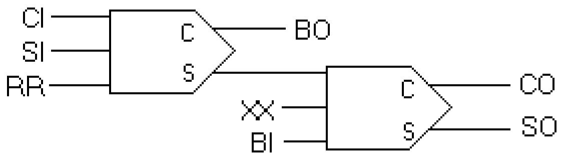
where $j \in \{-1,0,1\}$ and $k \in \{-2,-1,0,1,2\}$ are determined by the CU, and ACC is a carry-save integer. The CU determines the j values by reference to the bits of the multiplier Y, and it determines the k values by inspecting the top few bits of the value of ACC in order to ensure that the result does not overflow the accumulator.

In the ripple carry mode the function implemented is

$$ACC' = ACC + kR$$

where ACC' is a standard binary integer. Note that there is no automatic doubling in this case.

The logic at the core of each cell of the AU is shown here.



The layout has been drawn with two full adders but in fact any design capable of adding five bits and giving a three-bit result can be used. The only constraint is that one of the outputs (BO in the diagram) should be independent of one of the inputs (BI): this is to prevent a ripple-carry path from propagating through the entire array.

We define n as the length of R and m as the length of Y (in most applications $m <= n$, but this is not a requirement). We shall denote the individual bits of registers by subscripts, with X_0 being the least significant bit of X.

To simplify the CU design, the most significant bit of R should be 1. If R is too small then it can be shifted to the left and X can be shifted to match.

Using the notation in the diagram above, cell i has its input signals connected as follows:

CI = latched copy of C_{i-1} from the previous clock cycle.
SI = latched copy of S_{i-1} from the previous clock cycle.
RR = $\sim R_{i-1}$ if k=-2, $\sim R_i$ if k=-1, 0 if k=0, R_i if k=1, R_{i-1} if k=2.
XX = $\sim X_i$ if j=-1, 0 if j=0, X_i if j=1.
BI = BO from the previous cell.

The outputs are connected as follows:

CO goes to the input of the C_{i+1} latch.
SO goes to the input of the S_i latch.
BO goes to BI of the next cell.

Apart from the logic shown, some additional logic is used to convert the AU into a ripple-carry adder at the end of a multiplication. During ripple-carry operation cell i has its input signals connected as follows:

CI = latched copy of C_i from the previous clock cycle.
SI = latched copy of S_i from the previous clock cycle.
RR = $\sim R_{i-1}$ if k=-2, $\sim R_i$ if k=-1, 0 if k=0, R_i if k=1, R_{i-1} if k=2.
XX = CO from the previous cell.
BI = BO from the previous cell.

During ripple-carry operation the outputs are connected as follows:

CO goes to the XX of the next cell.
SO goes to the input of the S_i latch.
The input of the C_{i-1} latch is zero.
BO goes to BI of the next cell.

The storage required by this cell design is 5 bits per stage: X, Y, R, C, and S. Y does not need to be stored as part of a cell because only the CU uses it, so it will usually be better to implement it as a shift register or in RAM. Implementation of the control logic is made simpler if (as is usual) X and R are stored in latches that output both a value and its complement.

CU operation

The function of the CU is to decide what multiples of X and R to add or subtract.

The multiplicand X

A conventional multiplier design conditionally adds X in each clock cycle, depending on the value of the appropriate bit Y_t of Y (with t counting down from m-1 to 0). The present algorithm cannot tolerate the repeated adding of X in consecutive cycles, so an alternative method is used, with the t now counting down from m-1 to –1. Y_m and Y_{-1} are both interpreted as 0. NX is a state flag that will be relevant in the computation of R: its initial value is "+".

Y_{t+1}:Y_t	*j*	Action	NX
0:0	0	Do nothing	No change
0:1	1	Add X	"-"
1:0	-1	Subtract X	"+"
1:1	0	Do nothing	No change

It should be noted that implementing subtraction does not add greatly to the complexity of the design. Given that both X and ~X are available (because of the nature of D-type latches), an addition-only design requires one NAND gate per cell and an addition/subtraction design only requires three. In a full-custom design, the difference is even less: only one transistor more is needed to implement subtraction.

The modulus R

This is the crucial aspect of the present design. A conventional ripple-carry modulo reduction device would subtract R from ACC if and only if ACC >= R, so that in effect the only values of k needed would be 0 or –1. Carry-save designs do not have the luxury of knowing the value of ACC exactly and they have to guess on the basis of inadequate evidence. Because a guess is sometimes wrong, the computed value of ACC may differ from the correct value by a multiple or R. The art of designing a controller is to ensure that this error is detected before it has exceeded the capacity of the AU to correct it.

We add two extra cells (n and n+1) are added at the most significant end of the array. R and X are assumed to be zero for $i >= n$. At the start of each clock cycle a simple four-bit adder computes the following function:

$D = 8 (S_{n+1} + C_{n+1}) + 4 (S_n + C_n) + 2 (S_{n-1} + C_{n-1}) + (S_{n-2} + C_{n-2}) + (S_{n-3} \bullet C_{n-3})$

— where the Si and Ci values are the ones latched from the previous clock cycle.

Note that $S_{n-3} \bullet C_{n-3}$ is just the integer part of ½ $(S_{n-3} + C_{n-3})$.

D is interpreted as a signed 4-bit integer (with any carry discarded) and the value of *k* is then computed as follows:

D	*k*
-9 to –4	+2
-3	+2 if NX= “-”, otherwise +1
-2	+1
-1	0
0	-1
+1	-2 if NX= “+”, otherwise –1
+2 to +7	-2

Note: there is an ambiguity between D=-9 and D=+7 because they have the same binary representation. Analysis of the overall algorithm shows that D can be -9 only when NX= “+” and can be +7 only when NX= “-”, so NX can be used to discriminate between these two cases.

Overall sequence of operations

With the AU in its normal carry-save mode, *t* counts down from *m*-1 to -1. At each stage the function computed is

$$\text{ACC}' = 2 \times \text{ACC} + j\text{X} + k\text{R}$$

A ripple-carry cycle (“RC1”) is performed, with NX = “0”:

$$\text{ACC}' = \text{ACC} + k\text{R}$$

If ACC' is non-negative, the calculation is complete.

If it is negative, an additional ripple-carry cycle (“RC2”) is performed:

$$\text{ACC}' = \text{ACC} + \text{R}$$

Thus the overall work required is *m*+1 carry-save cycles plus one or two ripple-carry cycles.

Why the algorithm works

To make this description more readable, we will assume that R is 8 bits long, and we will write it in hexadecimal. Thus (given that the top bit of R has to be 1) the possible values of R lie in the half-open interval [80,100).

We will denote the value stored in the accumulator ACC by *z*.

Note that *z* cannot be less than –2R, because then 2*z*+2R would be even more negative and the algorithm would diverge; similarly, *z* cannot be more than +2R because then 2*z*-2R > 2R and once again the algorithm would diverge..

Interval arithmetic

In this algorithm, all decisions about the multiple of R to use have to be taken on the basis of the calculated value D. What does D actually mean? Consider D = +1. If we calculate ACC from C + S by adding corresponding digits together (so that the weights are still binary but the digit “2” is allowed) we see that the smallest number that can give D = +1 is 00 0020 00000 and the largest is 00 0112 2222: thus D = +1 implies that $z \in [40,\text{A0})$. Similarly D = 0 implies $z \in [00,60)$: in fact, the width of the interval is always 60.

A simple case

Let us start by assuming that R has its smallest possible value: that is, that R = 80.

Suppose that $|z| < R$. Then it is possible to look at each value of D and assign a value of k to it so that $|2z+2kR| < R$ also. For example: D = -1 means that $z \in$ [-40,20). k=-1 maps this onto [-100,-40) and k=+1 maps it onto [0,C0), so the only allowable value of k in this case is 0, which maps z onto [-80,40).

It is also possible to extend this assignment of k-values so that if $R < |z| < 2R$ then $|z|$ will get smaller in each iteration.

Now consider what happens if j=1 so that X has to be added in this cycle. In that case, the result of the calculation is $2z+X+2kR$. Since X $\in$ [0,R), it follows that the result may have been "kicked" upwards by an amount just less than R. Since we have set up the values of k so that $2z+2kR < R$, the result after the "kick" is less than 2R, and we are safe.

However, we are not safe if we immediately get kicked in the same direction again while $z > R$, and this is the reason for the unusual approach to multiplication that the present design uses, with its alternate addition and subtraction of X. After a kick upwards, it is always safe to be kicked downwards; and vice versa.

The general case

Now let us assume that R has its largest possible value, which is 2^n-1: we shall write this as R = 100 to emphasize its closeness to 2^n in the general case.

Everything that we did for R = 80 translates into the case of R = 100, but there is a snag: for D=+1, the only usable value of k is -2 if R = 80 but -1 if R = 100.

Here is the problem. Recall that D=+1 implies $z \in$ [40,A0). The values of $2z + kR$ are:

	R = 80	R = 100
k = -2	[-80,40)	[**-180**,-C0)
k = -1	[0,**C0**)	[-80,40)

The numbers shown in boldface are outside the range [-R,+R) and will therefore cause trouble if we get a kick within this cycle: C0 (=1½R) could get kicked up to 2½R and –180 (=-1½R) could get kicked down to –2½R: in both cases, the values of z would then diverge and the algorithm would fail.

Rescue comes from the fact that we always know where the next kick is coming from (though not necessarily when) because the multiplication algorithm is always a sequence of alternate additions and subtractions. We can use this knowledge to derive k as follows: if you know that you will not be kicked upwards, choose k=-2; if you know that you cannot be kicked downwards, choose k=-1.

Exactly the same problem occurs when D=-3, and the solution is the same.

The last few cycles

At t=-1 (recall that t counts downwards) NX="+" always, since the number of additions and subtractions of X in a multiplication is always even.

Referring to the D-to-k table and allowing for a possible downwards kick from X, we find that $z \in$ [-100,80) for R = 80 and $z \in$ [-100,C0) for R = 100.

Now perform one further cycle, the one denoted by "RC1" in the original description. Referring to the D-to-k table again and remembering that NX="0", this cycle ensures that $z \in$ [-R,R) for all valid values of R. If $z >= 0$, z is the result that we want; if not, we perform the extra ripple-carry cycle RC2 to compute z+R.

RC1 has to be a ripple-carry cycle because at the end of it we need to know for certain whether $z >= 0$. If $z < 0$, then RC2, the cycle that adds R to it, has to be a ripple-carry cycle also, since we want the final result to be an integer.

Comparison with other designs

Brickell [1]

	Present design	Brickell
Size of core arithmetic unit	2 full-adders	5 half-adders + 3 OR gates
using Austria Microsystems C35 [3]	546 μm², 2.26 μW/MHz	1006 μm², 5.72 μW/MHz
using NEC CMOS-8L [4]	18 cells	36 cells
Storage required (X, Y, R, ACC)	5*n* bits	7*n* bits
Multiples of X needed	-1, 0, +1	0, +1
Multiples of R needed	-2, -1, 0, +1, +2	-2, -1, 0
Additional time per multiplication	1 or 2 ripple-carry cycles	10 normal cycles
Need to convert result to binary	No	Yes

- Sizes have been taken from the manufacturers' catalogues without attempting any optimisation. Both designs require some additional space for multiplexers etc. Both designs require some additional (partly degenerate) cells at the top of an array, but this overhead is not significant in the context of 512-cell or 1024-cell designs.
- Ripple-carry cycles are slower than normal ones but can be accelerated by look-ahead carry circuits and typically last between 4 and 6 normal cycles.

The present design uses considerably less silicon than Brickell's. It also does not require a final conversion of the result into binary, which is an advantage if the operation required is modular multiplication (when doing modular exponentiation, the overhead of one the conversion would be negligible).

Montgomery [2]

Montgomery's algorithm is fast and it uses little silicon. It is very good for modular exponentiation but prohibitively slow for modular multiplication.

The reason for this slowness is that Montgomery's algorithm represents numbers differently: denoting an application of the algorithm by MA(x, y), the algorithm computes MA(x, y) = $2^{-n}xy$, or, equivalently, MA(2^nX, 2^nY) = 2^nXY.

Any computation that uses Montgomery's algorithm must therefore

1. Compute 2^{2n} mod R. This computation cannot be done using Montgomery hardware, so it will be slow.
2. Convert each operand into the required form: for example, MA(2^{2n}, X) = 2^nX.
3. Perform the desired computation. This requires one use of MA per modular multiplication.
4. Convert the result into a conventional number: for example MA(1, 2^nZ) = Z.

We leave aside the cost of step 1, since the result depends only on the modulus and in most applications a modulus is used for very many calculations.

If step 3 is a modular exponentiation involving a thousand multiplications, the additional overhead of steps 2 and 4 is negligible; but if one wants to use a Montgomery design to perform a single modular multiplication, then this requires a total of four applications of MA, which makes Montgomery's hardware almost four times slower than more conventional approaches.

In contrast, the design described in this paper needs no pre- or post-computation at all, and given conventional binary inputs it produces a conventional binary output.

Conclusions

The algorithm described here is simple: its implementation in silicon is economical and the proof of its correctness is straightforward. Because it delivers its results as binary integers, it is as efficient when used for modular multiplication as it is for modular exponentiation.

[1] Ernest F. Brickell, "A Fast Modular Multiplication Algorithm with Applications to Two Key Cryptography", in *Advances in Cryptology: Proceedings of CRYPTO '82*, Plenum, New York (1983), pp. 51-60.

[2] P.L. Montgomery, "Modular multiplication without trial division", *Math. Computation*, (1985), 44:519-521.

[3] Austrian Microsystems: http://asic.austriamicrosystems.com/databooks/c35/databook_c35_33/

[4] NEC: http://www.necelam.com/ASIC/gateArray.cfm

[5] J.K. Gibson, "A Generalisation of Brickell's Algorithm for Fast Modular Multiplication", *BIT* 28(4): 755-764 (1988).

[6] C.D. Walter & S.E. Eldridge, "A Verification of Brickell's Fast Modular Multiplication Algorithm", *International Journal of Computer Mathematics*, vol. 33 (1990), pp. 153-169.