# Systolic Modular Multiplication.

Colin D. Walter

Computation Department, U.M.I.S.T.,
PO Box 88, Sackville Street, Manchester M60 1QD, U.K.
e-mail: cdw@sna.co.umist.ac.uk

13 March 1992

Key Words: *Modular Multiplication, Fast Computer Arithmetic, Digital Arithmetic Methods, RSA Algorithm, Cryptography, Systolic Array.*

## Abstract

A systolic array for modular multiplication is presented using the ideally suited algorithm of P.L. Montgomery. Throughput is one modular multiplication every clock cycle, with a latency of $2n + 2$ cycles for multiplicands having $n$ digits. Its main use would be where many consecutive multiplications are done, as in RSA cryptosystems.

## 1 Introduction.

Among other reasons, security for an ever-increasing number of electronic banking transactions has fuelled research into cryptographic algorithms and efficient implementations of them. Some of the main systems for encryption or key transfer, such as RSA [7], require fast modular multiplication of numbers containing 500 or more bits. Here we describe a systolic array for performing this. It provides another alternative for the rapid encryption of bulk data.

Two pieces of related work have appeared during revision of this paper. One, by Koç and Hung [4], is the only attempt so far at a systolic algorithm for modular multiplication. The other, by Shand, Bertin and Vuillemin [8], describes a pipeline similar to one row of the array presented here which the authors have programmed into their hardware array. The first of these suffers from excessive latency and a slow clock, the result of the unsuitability of the natural algorithm [1] which is based on Horner's nested multiplication method. It involves repeated additions of the multiplicand and repeated subtractions of the modulus. These are interleaved to keep register size down. For speed, just a few of the most significant digits of the partial product are used to decide the multiple required in the next modular subtraction. In a digit-level systolic array, this multiple must be pumped down the cells performing the subtraction from the cell for the most significant digit to that of the least. But, because carry propagation is in the opposite direction, a redundant number system has to be used to limit the influence of carries. However, a delay is still caused while the limited carries are accumulated at each cell. So, overall, the first digit of the output in [4] appears after about $13n/2$ clock cycles, where $n$ is the maximum number of digits in any input. Moreover, the clock cycle needs to be slow enough to allow for calculating the multiple of the modulus.

The clash in direction between the movement of the carries and that of the multiple of the modulus is resolved by using P. L. Montgomery's algorithm [6] which re-structures the operation so that the modular adjustment depends instead on the *least* significant digits. Like Shand *et al.* [8], we make use of this and, like them, we can expect similarly impressive computation speeds. The result of this choice is a truly systolic algorithm with much reduced latency and a faster clock. Indeed the most complex cell is the most common one, which performs two digit multiplications and three digit additions, and the delay till the first output digit appears is just $2n+2$ clock cycles. There is, unfortunately, a price to pay for the greater efficiency. Some post-processing of the output is required, but the cost of this is relatively slight if a number of modular multiplications with a common modulus are performed sequentially, as in the RSA cryptosystem.

## 2 The Basic Algorithm.

The ideas which are combined here are those of Montgomery [6] on modular arithmetic, and of McCanny and McWhirter [5] on systolic multiplication. Montgomery shows how to perform modular multiplication by a method which includes reversing the order of treating the digits of the multiplicand, shifting down instead of up, and adding rather than subtracting multiples of the modulus. A Pascal-like description of his algorithm for $(A \times B)$ *mod* $M$ is given below, and is followed by a short justification of how it works. Further detail is found also in [2]. First, however, we review the notation that is used.

Numbers $A$ are written with base $r$ and digits $A[i]$, so that $A = \sum_{i=0}^{n-1} A[i]r^i$ where $n$ is the number of digits in $A$. The choice of digit range is unimportant, but we interpret *mod* $r$ as yielding a digit value. Let $m$ and $n$ be the number of digits in the inputs $M$ and $A$ respectively. Normally, $B$ has at most the same number of digits as $M$ or, at worst, satisfies $B < 2M$. We assume the latter, so that $B$ has at most $m+1$ digits. The radix $r$ is chosen before implementation so that the operations *div* $r$ and *mod* $r$ are trivial. In particular, here they are done by shifting or inspecting the lowest digit respectively. The choice for $r$ will probably be a small, fixed power of 2 to make translation to and from binary easy.

A pre-condition for the algorithm to work is that there is no common factor between $r$ and $M$. In the definition of $Q[i]$ below, the multiplicative inverse $(r - M[0])^{-1}$ *mod* $r$ is required. This is the digit which, when multiplied by $r - M[0]$, gives a product with remainder 1 on division by $r$. The co-primeness of $M$ and $r$ ensures that such a number exists. Inspection of the assignment to $P$ shows that there is a multiple of $r$ to be divided by $r$. Hence no information is lost in the division, which is therefore exact.

```
$P := 0$;
For $i := 0$ to $n-1$ do
Begin
        $Q[i] := (\,(P[0] + A[i]\times B[0]) \times (r - M[0])^{-1}\,)\ mod\, r$;
        $P\ \ := (P + A[i]\times B + Q[i]\times M)\ div\ r$
End
```

To see what this code does, let $P_i$ be the value of the partial product $P$ at the start of the loop for which $i$ is the value of the control variable. So $P_0 = 0$. Let $A_i = \sum_{j=0}^{i-1} A[j]r^j$ and $Q_i = \sum_{j=0}^{i-1} Q[j]r^j$ be the lower parts of $A$ and $Q$. So $A_0 = Q_0 = 0$. Then, by induction, $r^i P_i = A_i \times B + Q_i \times M$. Thus the final value $P_n$ of $P$ satisfies $r^n P = A\times B + Q\times M$, so that $P \equiv (r^{-n}AB)\ mod\ M$. The systolic array described here produces output $P$ with this extra power of $r$, but it may also be used to remove that factor. This is done by an extra application of the operation with inputs $P$ and $r^{2n}\ mod\ M$, the latter being pre-computed once for all and stored with $M$. Further detail in the context of the RSA algorithm is provided in Section 4.

Suppose the standard set of digits $\{0, 1, ..., r-1\}$ is used for $A$ and $Q$. Then the maximum value $\kappa$ of $P$ satisfies $\kappa = (\kappa + (r-1)B + (r-1)M)\ div\ r$, and so $\kappa = M + B$. Once the digits of $A$ run out, subsequent values of $P$ decrease, being bounded above by $\kappa_i$ ($i = 0, 1, ...$) which satisfy $\kappa_0 = \kappa = M + B$ and $\kappa_{i+1} = (\kappa_i + (r-1)M)\ div\ r$, i.e. $\kappa_i = M + (B\ div\ r^i)$. These yield bounds on the number of digits needed to represent the intermediate and final values of $P$. If initially $B < 2M$ then $P < 3M$ always, so that $P$ requires up to two bits more than $M$. Furthermore, in the next iteration beyond the one using the last, most significant digit of $A$, the output satisfies $P < M + (2M\ div\ r) \le 2M$. This would be suitable for re-use as the $B$ input of a further modular multiplication.

## 3 The Systolic Array.

A key property of Montgomery's algorithm is that the choice of modulus multiple is based on the lowest digit $P[0]$ of the partial product. With this multiple $Q[i]$ determined, the digits of the $i+1^{st}$ partial product $P_{i+1}$ can be computed in order starting with the lowest. Also, with $Q[i]$ known, digits output from one addition cycle enable the corresponding digits for the next cycle to be found. This is done by the array illustrated in Figure 1 where each row performs an iteration of the loop and columns compute successive values for a single digit position. The typical cell performs a single digit calculation of the assignment $P := (P + A[i]\times B + Q[i]\times M)\ div\ r$, and generates a carry in the normal way. So it is specified by

$$P_{i+1}[j-1] \ + \ r\times Carry_{Out} \quad = \quad P_i[j] \ + \ A[i]\times B[j] \ + \ Q[i]\times M[j] \ + \ Carry_{In}$$

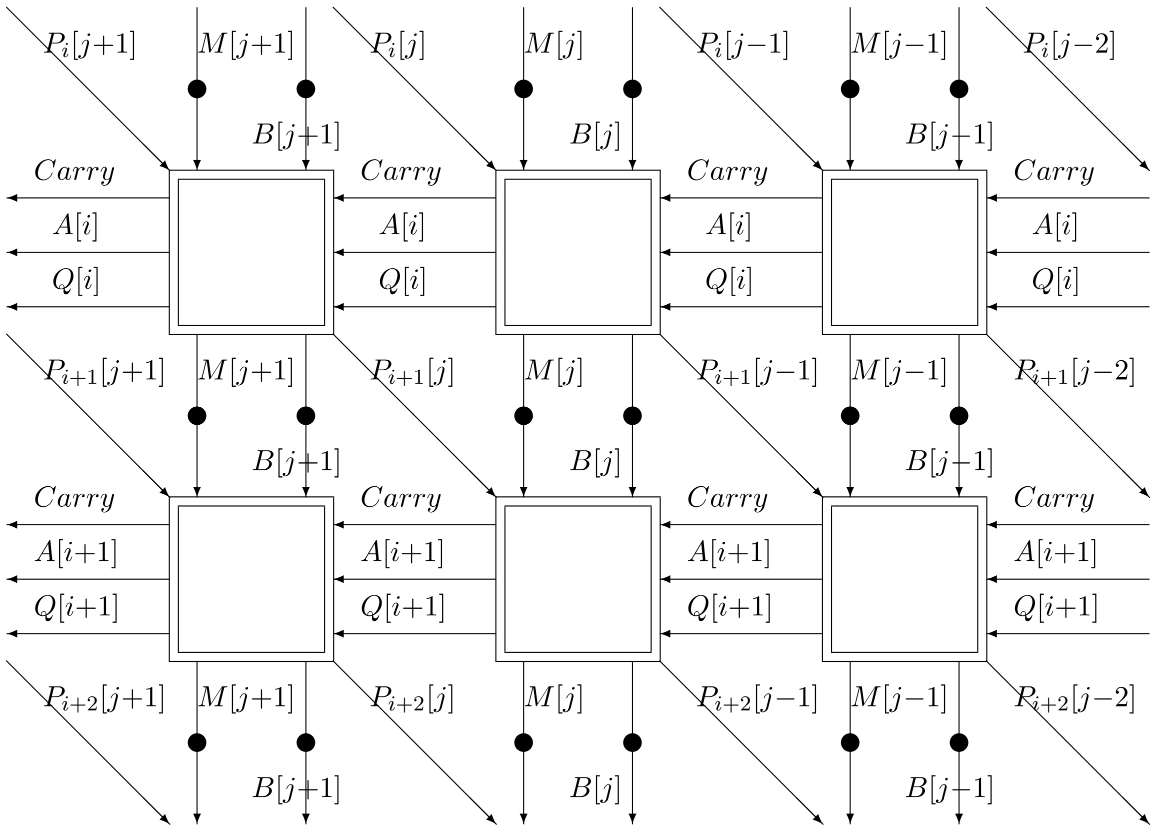

Figure 1: Arrangement of Modular Multiplication Cells.

The $i+1^{st}$ row down computes $Q[i]$ and $P_{i+1}$, whilst the $j+1^{st}$ column from the right finds the digit with index $j$ for each $P_i$. The input digits of $M$, $A$ and $B$, and also those of $Q$, are just pumped through cells without change until a different modular multiplication is commenced. In the cases of $M$ and $B$ the digits need delaying by an additional clock cycle (i.e. the operation time of a cell), so that they arrive when needed. This is achieved by the latches indicated between the cells.

The cells also forward up the rows the carries from the additions. If the digits of each number lie in the usual range $\{0, 1, ..., r-1\}$ then the above definition of the cell shows that carries are bounded by $2(r-1)$. Thus the carry needs one more bit than a digit for its representation.

Suppose the set-up described at the end of the previous section is used so that $B$ initially satisfies $B < 2M$. Then we wish to perform $n+1$ addition cycles to obtain output satisfying $P < 2M$. For this there must be $n+1$ rows in the array and $A$ must be padded at the top end with an extra zero digit, i.e. $A[n] = 0$. The output we desire is $P_{n+1}$ from the row with input $A[n]$. It will equal $(r^{-n-1}AB)$ *mod* $M$, or be $M$ more than this because it is bounded by $2M$. So, a further row might usefully be added to subtract the possible extra M, with the decision about whether to take $P_{n+1}$ or $P_{n+1} - M$ being made when the sign of the latter is established. This is further discussed below.

Intermediate values of $P$ are bounded by $3M$. This entails that there is no carry from the column in which each $P_i[m+1]$ is computed. So the array needs no column to the right of this and, since $P_i[m+2] = 0$ always, this value can be input at the left side of the array. Along the top edge the inputs $P_0[j]$ are all 0 because the initial value of $P$ is 0. Also on this edge, the digits of $M[j]$ and $B[j]$ are input $j$ clock cycles after $M[0]$ and $B[0]$ so that they match the progress of the carries.

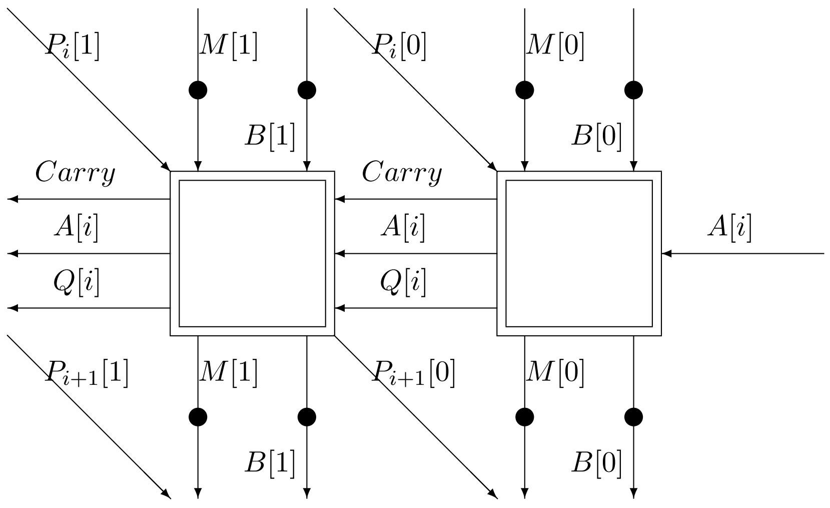

Figure 2: The Rightmost Cells.

The cells are all identical with the exception of the rightmost column, illustrated in Figure 2, which has the burden of computing the digits of $Q$. Carries entering the array from the right are, of course, zero. Also, the digit $A[i]$ is input $2i$ clock cycles after $A[0]$ since the shift down puts the calculation of each $P_i$ two cycles behind its predecessor. The digits of $Q$ are defined as in the Pascal code by

$$Q[i] \quad = \quad ((P_i[0] + A[i] \times B[0]) \ \times \ (r - M[0])^{-1}) \ \mathit{mod}\ r.$$

Although the rightmost cells do not generate output digits $P_{i+1}[-1]$ because they are deleted by the *div* $r$ operation, nevertheless a carry may be generated when evaluating the $0^{th}$ digit before the shift down:

$$r \times Carry_{Out} \quad = \quad P_i[0] + A[i]\times B[0] + Q[i]\times M[0]$$

In [3] Eldridge and Walter describe how to simplify the calculation of $Q[i]$ by shifting $B$ up to make $B[0] = 0$. This is possible here too, but at the cost of an extra row and an extra column to obtain an output less than $2M$. The advantage is that it reduces the complexity of the rightmost cells, and hence their operation time, to at most that of the standard cells. Such a simplification maximises the possible clock speed at very little cost. For this it is assumed that $(r - M[0])^{-1}$ *mod* $r$ is pre-computed once (a table look-up) and fed in like $M[0]$. There may also be a slight advantage in scaling $M$ to $M' = ((r - M[0])^{-1} mod\ r)\times M$ since then $M'[0] \equiv -1(mod\ r)$ leads to the simple definition $Q[i] = P_i[0]$. Computing would then be done modulo $M'$ giving a result bounded by $2M'$ rather than $2M$. So the penalty for that would be extra cleaning up afterwards.

The first output digit of a modular multiplication appears after $2n + 2$ clock cycles, and successive digits over the next $m$ clock cycles. Indeed, the output $P[j]$ appears $2n + 2$ cycles after $B[j]$ is input and $2n + 2 + j$ cycles after the very first input, namely $B[0]$. The latency is thus $2n + 2$. However, since $m + 1$ digits are output on each cycle, the throughput is equivalent to 1 modular multiplication per cycle.

Comparison with the array of Koç and Hung [4] is worthwhile. The logic of cells is simpler here because there is no need for a redundant number representation. Also the clock cycle time is shorter. In [4], this is bounded below by the time to compute the equivalents of the digits $Q[i]$ in their super cells $LY^5$ and $LU^5$. For radix 2 and a technology using only 2-input gates and no buffering to enable outputs to drive more than one load, this requires a circuit similar to that given in [3] Figure 2, with its critical path length of 13 XOR gates. Here a depth of 5 gates suffices for the general cell (see Figure 3), and less for the rightmost cells, when the suggested simplifications are made. Allowing for register set-up and hold times, the clock here should be about $(13 + 3)/(5 + 3) = 2$ times faster, with the same speed-up factor for the throughput. Since there are $2n + 2$ clock cycles per operation instead of about $13n/2$, the latency is reduced roughly 7-fold. Different technologies allow trade-offs here between power, area and speed.

## 4 RSA Cryptography.

In practice the systolic array is likely to be of most use for RSA cryptography where modular exponentiation is needed. This is normally performed by repeated modular multiplication. It has already been observed that the array calculates $(ABr^{-n-1})$ *mod* $M$ rather than $AB$ *mod* $M$, and an extra $M$ may be present. The extra $M$ makes no difference to subsequent modular arithmetic, and so can be ignored until the final result of decryption is obtained. The extra factor of $r^{-n-1}$ can be removed after each multiplication by using the array again to multiply $(ABr^{-n-1})$ *mod* $M$ and $r^{2n+2}$ *mod* $M$. However, in exponentiation this also can be left till the end of the calculation.

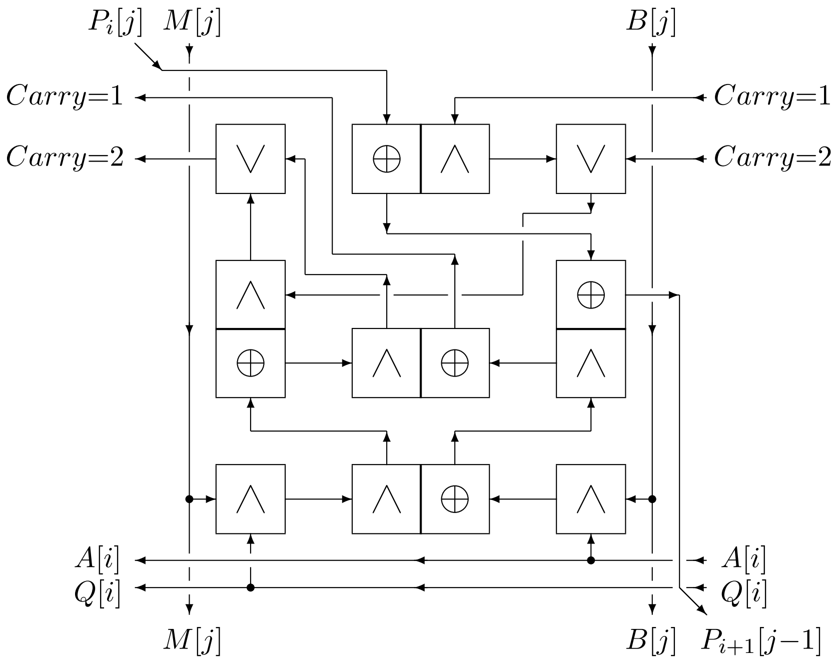

Figure 3: The Typical Cell for Radix $r = 2$.

Encryption and decryption is done using two keys, $E$ and $D$ respectively, with the property $A^{DE} \equiv A \ mod \ M$. Suppose the encryption of $A$ is done using the systolic array to successively square $A$ modulo $M$ and to multiply these 2-powers as necessary into a running total, retaining the unwanted powers of $r$. Then $(Ar^{-n-1})^E \ mod \ M$ is produced rather than $A^E \ mod \ M$. Hence, using the array for an extra modular multiplication by $r^{(n+1)(E+1)} \ mod \ M$ will provide $A^E \ mod \ M$. The cost of this extra multiplication is minor compared to the typical number of modular multiplications in the encryption process. Naturally, the owner of the key $E$ only needs to compute $r^{(n+1)(E+1)} \ mod \ M$ once and store it for use until the key is changed. Decryption is similar. In fact, only the decrypter needs to adjust for the power of $r$, and the property of $DE$ ensures that the scaling factor still depends only on his key. Another alternative strategy is described in [3].

The main overhead might be in the over-large residue. Although this problem is shared by all fast implementations of modular multiplication (see [3]), it seems worse here because the top output digits are generated last of all. Apparently, they need to be known in order to decide whether a final subtraction of $M$ is necessary or not. This is true in general but false under appropriate conditions! An easy solution is always to make the message for encryption into a multiple of $r$, thus ensuring that the lowest digit is 0. Then, at decryption, the output will have lowest digit $M[0]$ if, and only if, the extra multiple of $M$ is present. A bottom row of special cells can be added to the systolic array to remove $((P[0]\times M[0]^{-1}) mod\ r)\times M$ and produce the required answer. (The same formula works if up to $(r-1)M$ needs subtracting, and there is an obvious generalisation for higher multiples.)

For RSA applications, typical inputs have about $10^3/2$ bits. This means about $10^6/4$ bit-processing cells, or about $4\times10^6$ gates. As this may be beyond available technology for a single chip, suppose that as many rows as possible are put onto the chip. Theoretically, a number of chips could be used sequentially to construct the whole array, but in practice it would be impossible to transfer the required $10^3/2$ bits per cycle between the chips. Alternatively, it is clearly better to re-route the output back to the input within a single chip, repeatedly feeding it back in at the top digit by digit as it is calculated, until the required $n+1$ partial products have been computed. If just $n'$ rows of cells are built, then at full capacity the array is simultaneously processing $2n'$ different modular multiplications.

It takes $2n+2$ clock cycles from the input of one digit to the output of the corresponding digit after the modular multiplication. It is only after such a delay that further progress on an exponentiation can take place. (Strictly speaking, for a non-zero exponent bit, there is a modular multiplication into a running total and a modular squaring which can be done simultaneously.) Hence the modular multiplications which the array is performing simultaneously must come from different exponentiations. This indicates that for use in RSA the messages for encrypting or decrypting should generally be numerous. Of course, one interesting choice is to reduce the array to a pipeline by just implementing 1 row. This would give a very cheap, simple modular multiplying chip for performing single encryptions.

## 5 Conclusion.

A systolic array for modular multiplication has been presented. Although there is an extra unwanted factor present because of the choice of algorithm, this can be dealt with at virtually no cost when the algorithm is used in RSA cryptography. Since in the simplest case a single digit is used to determine the multiple of the modulus to add during iterations of the algorithm, the clock can be made very fast. Neat solutions have been included to avoid finishing with output containing an extra unwanted multiple of the modulus, and to achieve the best possible clock speeds.

## References

[1] E. F. Brickell, A Fast Modular Multiplication Algorithm with Application to Two-Key Cryptography, *Advances in Cryptology – Proc. of CRYPTO 82*, ed. Chaum et al., Plenum, 1983, pp. 51-60.

[2] S. E. Eldridge, A Faster Modular Multiplication Algorithm, *Intern. J. Computer Math.*, vol. **40**, 1991, pp. 63-68.

[3] S. E. Eldridge and C. D. Walter, Hardware Implementation of Montgomery's Modular Multiplication Algorithm, *IEEE Trans. Comp.*, vol. **42**, 1993, pp. 693-699.

[4] Ç. K. Koç and C. Y. Hung, Bit-Level Systolic Arrays for Modular Multiplication, *J. of VLSI Signal Processing*, vol. **3**, 1991, pp 215-223,

[5] J. V. McCanny and J. G. McWhirter, Implementation of Signal Processing Functions using 1-bit Systolic Arrays, *Electronics Letters*, vol. **18**, 1982, pp. 241-243.

[6] P. L. Montgomery, Modular Multiplication without Trial Division, *Mathematics of Computation*, vol. **44**, 1985, pp. 519 - 521.

[7] R. L. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Comm. ACM*, vol. **21**, 1978, pp. 120-126.

[8] M. Shand, P. Bertin and J. Vuillemin, Hardware Speedups in Long Integer Multiplication, *ACM Sigarch*, vol. **19**, 1991, pp. 106-113.