Hardware Implementation of Montgomery's Modular Multiplication Algorithm.

Stephen E. Eldridge & Colin D. Walter

Department of Computation
UMIST
PO Box 88
Manchester M60 1QD, UK
www.co.umist.ac.uk

Abstract. Hardware is described for implementing the fast modular multiplication algorithm of P. L. Montgomery. Comparison with previous techniques shows that his method is up to twice as fast as the best currently available, as well as being more suitable for alternative architectures. The gain in speed arises from the faster clock which results from simpler combinational logic.
Key Words and Phrases: Modular Multiplication, Fast Computer Arithmetic, Digital Arithmetic, Parallel Bit Operations, Redundant Number Systems, RSA Algorithm, Public-Key Cryptography.

0 Introduction.

The problem discussed here is that of providing hardware to perform modular multiplication using the algorithm described by P.L. Montgomery [9], and further developed by the first author [5] and Dussé and Kaliski [4]. Much more efficient use of hardware is made here than for the standard algorithm of Brickell [1] studied by the present authors in [16], although subsequent work by the second author [15] provides similar improvements for that also. We establish that the new method has a speed approximately twice as fast as previously existing ones. Initially this requires a review of current techniques. Then we show how to decrease the depth of combinational logic so that a faster clock speed can be used. Thus, although the number of clock cycles is essentially the same as before, overall time is roughly halved. The penalties of the simpler design are:

- the technique works only for moduli prime to the radix of the number representations (e.g. only odd moduli for binary numbers),
- an extra modular multiplication is required to adjust each input and output,
- a certain constant dependent only on the modulus needs to be pre-computed ($r^{2n}\,mod\,M$ in the notation used later), and
- area is increased by two extra registers.

Thus, using the algorithm makes sense only if substantial intervening modular arithmetic under a common modulus has to be done, as in the RSA algorithm [12] where the first of these is always satisfied. Then the intervening adjustments are not needed. However, the faster clock can be justified only by carefully comparing new circuit diagrams against those of existing methods. We spend some time on this, eventually deducing that the clock can have its speed doubled.

Our motivation for studying fast modular multiplication comes from cryptography, where the RSA algorithm [12], and another by Diffie and Hellman [3], make essential use of it. Both methods are already much used for the distribution of keys for less secure cryptographic methods. Improvements in technology now make the RSA system likely to become the most widespread method for guaranteeing security and for authentication purposes.

1 Existing Multipliers.

Existing state-of-the-art dedicated hardware for calculating $(A\times B)\ mod\ M$ makes use of several techniques for speeding up the calculation. Since it has not previously been done in the literature, we start by reviewing these. As in the paper and pencil method, but starting with the most significant digit of A first, the basic algorithm repeatedly adds to a running total R the product B^* of the next digit of the multiplicand A with the multiplier B and performs a shift up. At the end the result R is reduced by a multiple of the modulus M to yield the equivalent least positive residue, i.e. the remainder of $A \times B$ on division by M. Apart from further miniaturization to allow faster clocking of the hardware, this may be speeded up as follows:

I. Shift M up so that its most significant digit always has the same position in the hardware. This allows moduli with different numbers of bits to be used easily.
Appropriate adjusting shifts are made to A or B and the output. For convenience we assume in future that this has been done.

II. Interleave modular subtractions with the normal calculation of the product by repeated shift and add. Now numbers stay roughly the size of the modulus M rather than becoming as large as the product $A \times B$. This saves register space.

III. Look at only the most significant digits of M and the partial result R to decide which multiple of M to subtract. This may result in a slightly inaccurate result. However, with care it can be used in all intermediate calculations provided that a final correction is made if necessary.

IV. Use a redundant representation for calculations. This means that numbers may have digits from a range greater than that required, such as $\{0,1,2\}$ instead of only $\{0,1\}$ for binary representations. This avoids the unbounded propagation of carries illustrated by the decimal addition of 1 and 999...9. All the digit operations of an addition can now be done in parallel with carries influencing only the digit sums in the next one or two places.

V. Increase the base of the number representation. Grouping the bits of a binary representation in pairs gives a representation in base 4 which is often used. Although other bases are possible, usually a power of 2 is chosen to make conversion to and from binary easy. Increasing the base reduces the number of digits in the multiplicand A and so reduces the number of clock cycles in the algorithm. However, the depth of hardware that has to be driven in a single clock cycle is increased as well, so that a slower clock must be used.

VI. Shift up the multiplicand A and the modulus M by several places. Now addition of the digit multiple B^* of the multiplier does not affect the topmost bits used to decide the multiple M^* of M which must be subtracted. The number of iterations in the process must be increased by the number of shifts, but the circuitry for determining the multiple is much simplified so that the clock speed can be increased. Finally, the result is shifted back down. In brief, $((AS \times B)\, mod\, (SM))\, div\, S$ is calculated, where the factor S corresponds to the shift.

VII. Each iteration requires the addition or subtraction of digit multiples B^* and M^* of B and M respectively. The pre-calculation of some or all of the linear combinations $B^* \pm M^*$ may be an advantage. However, increased speed from reduced combinational logic, which selects instead of multiplies, must be weighed against the increased area of further registers.

VIII. If the modulus M has a known decomposition as a product of pairwise coprime numbers M_j then the arithmetic can be done independently modulo each factor M_j and the Chinese Remainder Theorem used to reconstruct the correct residue modulo M.

IX. Decide which multiple of M to subtract early enough for the adder not to be kept waiting for it. This may require partly calculating it in a previous clock cycle. But it can be achieved, for example, by scaling the modulus M so that the topmost bits of the modulus are known and the combinational logic for technique III is simplified. The penalty for scaling is that several final subtractions of the original modulus may be necessary to obtain the least non-negative residue.

X. If the multiplicand A is not already in non-redundant form, convert it to make the formation of B^* easier. In the basic algorithm above this must be done "on-the-fly" to produce the digits in the order they are consumed, i.e. most significant first.

The techniques I-IV and VI are all used by Brickell [1]; V is found in [14] and [7] and VII in, for example, [7] and [10]. The look-up table, which might be used in VII, is more usually associated with III, as in [13]. But, because of the potentially enormous size of such tables, this is appropriate only for software or hardware which does sequential digit operations with digits of, perhaps, 32 bits rather than the 512 or so parallel bit operations we are considering. IX is used in [15] whereas X is achieved as in [6] at the expense of a large worst case delay, but small average delay, between the generation of A and its use as an input. Brickell [2] provides a survey of currently available RSA chips. This, and our search of more recent work, leads us to believe that the techniques listed above are, at present, exhaustive. Thus Brickell's description [1] remains the basis of the best hardware to date, and so provides the standard against which to compare Montgomery's method.

For completeness we have included VIII, which is described in [11]. It shows how to combine rather than perform modular multiplications. We are concerned instead with what happens inside a single modular multiplication.

One of our main tasks is to find analogous techniques which apply to Montgomery's modular multiplication algorithm [9]. Several apply equally well and need no further comment, except that they can be safely ignored in order to simplify the comparison between the algorithms. Among these are V, VII and VIII. Moreover, using Brickell's adder provides I-IV and VI immediately. This leaves just IX and X. They have become explicit through the work here, leading the second author to apply the scaling described above (see [15]) to achieve IX for the standard algorithm [1] of Brickell.

2 Montgomery's Algorithm and Notation.

To describe the new algorithm clearly, we remind the reader of some notation used in modular arithmetic. The remainders on division by an integer m form what is called the ring of residues modulo m. This means we have addition and multiplication with similar properties to the integers. In particular, the remainders or residues of the integers 0 and 1 are additive and multiplicative identities respectively, i.e. $\alpha + 0 = \alpha = 0 + \alpha$ and $\alpha \times 1 = \alpha = 1 \times \alpha$ for all residues α. When α is non-zero, $m - \alpha$ is the additive inverse of α since $\alpha + (m - \alpha) = 0$ in this arithmetic. Also, if α has no factors in common with m, it has a multiplicative inverse, i.e. there is a remainder modulo m, say β, with the property $\alpha \times \beta = 1$. For example, 3 is its own multiplicative inverse modulo 8 because $3 \times 3 \equiv 1 \, mod \, 8$. We write $-\alpha$ and α^{-1} respectively for the additive and multiplicative inverses of α. In the algorithm below, the modulus under which multiplicative inversion is required is small, and so the process is simple (see [4]).

Suppose the base (or radix), r, for the representation of numbers is prime to M, i.e. r and M have no factors in common. This enables us to obtain a multiplicative inverse modulo r for $r - M_0$ in the code below, where M_0 is the lowest digit of M. Let the multiplicand A have n digits, denoted A_i $(0 \le i < n)$, so that $A = \sum_{i=0}^{n-1} A_i r^i$. We use similar notation for the other variables: the multiplier B, the modulus M, the residue R and the integer quotient Q. Then, using Pascal-like notation, the main part of Montgomery's algorithm [9] for computing the value R of $(A \times B) \, mod \, M$ is this:

$R := 0$;
For $i := 0$ **to** $n-1$ **do**
Begin
    $Q_i := \big((R_0 + A_i B_0)(r - M_0)^{-1} \big)$ **mod** r;
    $R := (R + A_i B + Q_i M)$ **div** r
End

where R_0 is the lowest digit of R, etc. Unfortunately this does not yet yield quite the right answer for $(A \times B)\, mod\, M$ – some post-processing is necessary. We know that $Q_i = (R + A_iB)(-M)^{-1}\, mod\, r$ because only the lowest digit of each argument contributes to the calculation modulo r. Substituting this value for Q_i shows that $R + A_iB + Q_iM$ is exactly divisible by r, and so no significant digits are lost in the division. Hence, as a result of the repeated divisions, the outputs R and Q actually satisfy $r^nR = AB + QM$. So a final extra modular multiplication by r^n is needed. In effect, the algorithm introduces an unwanted factor of $r^{-n}\, mod\, M$ into the product AB by calculating R satisfying $R \equiv ABr^{-n}\, (mod\, M)$. It is also easy to deduce from $r^nR = AB + QM$ that $R < M + B$ if A has digits bounded by $r - 1$, since then both A and Q are less than r^n. Indeed, this property is clearly a loop invariant. Further iterations, for which $A_i = 0$, would soon provide $R < 2M$, say (see [5]), although not necessarily the least non-negative residue. So an extra final subtraction of M may still be necessary.

The concluding multiplication by $r^n\, mod\, M$ could be done using the same algorithm again by taking the output R and $r^{2n}\, mod\, M$ as the new multiplicands, the latter having been calculated previously once and for all by some other means. However, when further modular arithmetic is involved, it is better to start by using the algorithm to pre-multiply all inputs, using $r^{2n}\, mod\, M$ as the other input. This yields, for example, $r^nA\ mod\, M$ and $r^nB\ mod\, M$ from A and B. Then all the required modular arithmetic (including additions and subtractions) can be performed without further intermediate scaling since every input and output contains an extra factor of $r^n\, mod\, M$. Lastly, a final such modular multiplication with 1 as the other argument will remove this extra factor from each output.

In the RSA algorithm, modular exponentiation is required. This is done by repeated multiplication under the same modulus. Invariably both encryption and decryption involve exponents with more than two bits, and at least one of the exponents must have many bits. Since the number of modular multiplications depends on the number of bits in the exponent, the two additional corrective modular multiplications, which the method entails, normally make little difference to the overall time. So doubling the clock speed would truly yield a faster algorithm – one that is effectively twice the speed for all exponents except those with just one or two bits. Thus the methods here, which enable this to be done, can certainly be used efficiently at one end at least of the encoding/decoding process, and usually at both.

In comparison with the usual algorithm described in Section 1, Montgomery's algorithm:

- reverses the order of treating the digits of the multiplicand A,
- performs a shift down instead of up on each iteration, and
- does an addition rather than a subtraction.

These changes allow several simplifications in the combinational logic, and have corresponding changes in the techniques listed in Section 1. However, recall that M needed to be prime to r, so that M is odd if r is a power of 2. The algorithm is thereby a little less general than Brickell's, but still equally applicable for RSA cryptography where the modulus, being a product of two large primes, is necessarily prime to the relatively small radix r.

In this paper we consider hardware for performing in one clock cycle, the iterative step in the code above. There are two operations to be performed: one is the addition of digit multiples combined with a shift, done by what we will call the adder; and the other is the determination of the multiple q' $(= Q_{i+1})$ of M needed in the next (the $i+1st$) iteration. For convenience, and to establish the notation used in the circuit diagrams, we write the repeated addition operation as

$$R' := (R + B^* + M^*)/r \ .$$

So the superscript $'$ will be used to distinguish inputs for the next iteration from those of the current one. The asterisk is used to denote digit multiples. Thus B^* and M^* are multiples of B and M respectively, the former by a digit of A, and the latter to make $R + B^* + M^*$ into a multiple of r. In fact, $M^* = qM$ for $q = Q_i$. The output, R', from the current cycle is fed into circuitry for computing in the same clock cycle the next digit multiple q' of M, ready for use in the next cycle. So the ends of clock cycles occur in the middle of the software loop described above.

Technique VI above is to modify B so that the choice of q is independent of it. Previously, A and M were shifted up relative to B. Here, the corresponding solution is to shift B up to make its lowest two digits zero. This is a relative shift in the opposite direction, achieved in the reorganised code below simply by writing r^2B in place of B in the previous code. In detail, the main body of the algorithm becomes:

> $R := 0 ;\ Q_0 := 0 ;$
> **For** $i := 0$ **to** $n+1$ **do**
> **Begin**
> $\qquad R \quad := (A_i r^2 B \ + \ R \ + \ Q_i M)$ **div** r ;
> $\qquad Q_{i+1} := \left(R_0 (r - M_0)^{-1}\right)$ **mod** r
> **End**

The output here satisfies $r^{n+2}R \equiv Ar^2B\ (mod\, M)$, which is as before. Generally R is now bounded by $M + r^2B$, but is reduced again to at most $M + B$ by the final two extra iterations for which the digits of A are zero. As can be seen from the previous code, shifting B up by at least one place makes the calculation of Q_{i+1} independent of the $A_{i+1}B$ term. However, shifting B up *two* places as here makes the new R_0 and hence Q_{i+1} independent also of the A_iB term during each loop iteration.

So now the choice of q' for the input $M^{*\prime}$ in the next iteration depends only on the lowest two digits of the current M^* and of the partial product R. The cost of this simplification is just 2 iterations more than the number of digits in the modulus compared with at least 5 in a straightforward application of Brickell's method (see [16]) or 3 if scaling the modulus is done (see [15]). Hence there is essentially no increase in the number of clock cycles for the new algorithm.

3 Circuits for Brickell's Multiplier.

We assume a particular hardware technology in the circuits illustrated in the figures, but the techniques and results obtained are applicable in general. The speed of the hardware clock is determined by the longest paths in the circuit driven during a single clock cycle. Our aim is to ensure that such paths lie in the adder. Since the adder forms almost all the hardware and it consists of many repeated units, if the aim were achieved then the hardware would be running at full capacity, with no further speed-up possible.

Even when the enhancements of Section 1 are included, previous designs based on Brickell's method have required a depth of circuitry roughly double that of the adder in order to calculate the multiple q' of the modulus to subtract next (see Figure 2). This forces the clock to have only half the speed at which the adder could operate. We show that with Montgomery's algorithm q' can be found sufficiently before the completion of the addition cycle to run the adder at full speed.

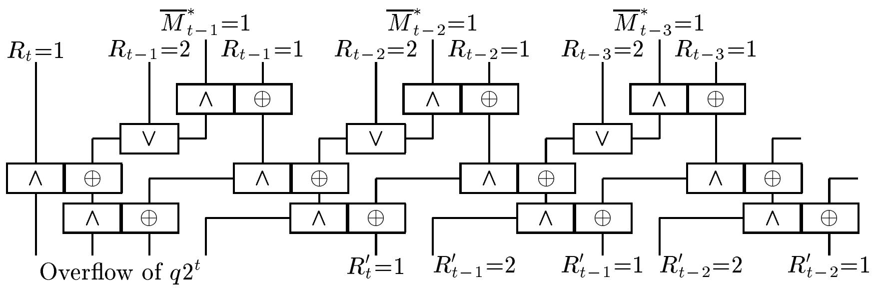

Fig. 1. Upper End of Existing Delay Carry Adder

Suppose the radix is 2, the digit range is $\{0, 1, 2\}$ for A, B, R and Q, and t and 0 are the indices of the most and least significant bits in any registers. Assume also that M has an irredundant form, i.e. digits 0 or 1. Figure 1 gives part of the circuit at the top end of the adder in chips which use Brickell's method, as corrected in [16]. Here t must be at least 5 more than the number of bits in B and M. Figure 2 extends this to give circuitry for the calculation of q'. For the main body of the adder, there are three inputs (to the top of these figures), namely M^* (or rather a shifted complement $\bar{M}^*$), R and B^*, and one main output R', which is the new value of R, (from the bottom of the figures). The most significant digits of B^* are zero because of the initial shift up of A and M. So there is no contribution from them over the digit range illustrated in these figures. The exclusive or, XOR, is denoted $\oplus$. Labels on inputs and outputs such as "$R_{t-1} = 2$" mean 1 when the condition is true, and 0 otherwise.

The top four bits of output marked as overflow are discarded. This is because the subtraction of the multiple M^* is performed by adding a shifted complement

$\bar{M}^*$. So we must discard an overflow of $q2^t$ if $M^* = qM$ is being subtracted. This overflow tells us what q is.

The topmost digits of the output R' are fed directly into circuitry to find the next quotient digit q'. This mainly comprises parallel copies, for each non-zero digit multiple of M, of that part of the hardware in Figure 1 which generates the overflow bits. These are combined to make the correct choice, having as inputs respectively the topmost digits of the possible values of $M^{*\prime}$. The result, given in Figure 2, has simplifications arising from particular properties of the input such as lack of lower (or higher) input bits.

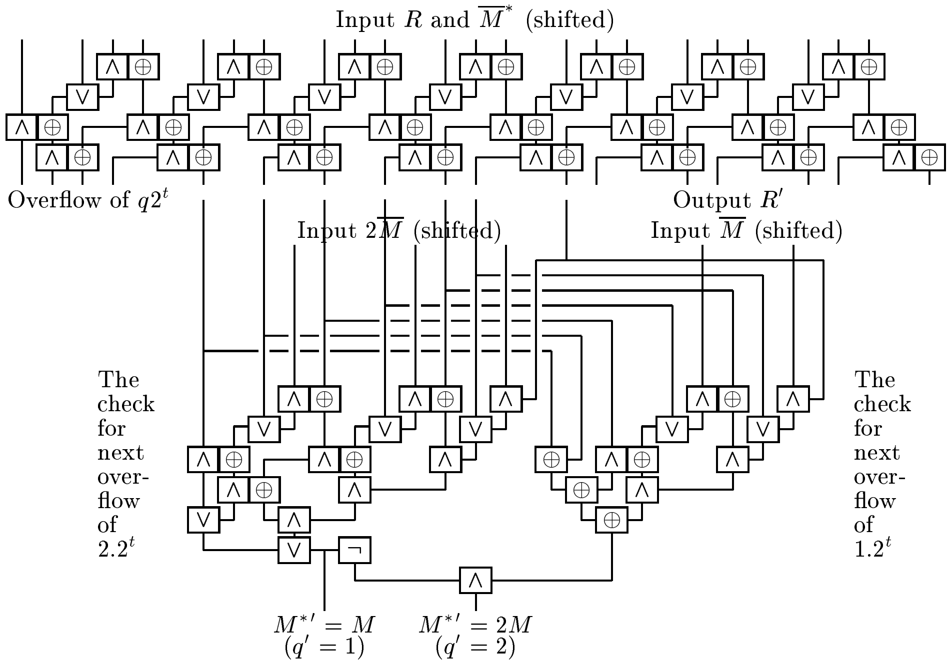

Fig. 2. Delay Carry Adder with Test for Next Multiple of M

These two parallel copies of the top of the adder add 7 more gates to the maximum depth of circuitry involved, giving 11 gates along the longest path from input to output. As most of these gates are of the same kind, namely XOR, we will not find it necessary to distinguish between the time delays caused by individual distinct gate types. These gate counts therefore closely approximate the time required for signals to propagate through the circuit. (Some hardware technologies allow the ANDing of several inputs together. Such gates would decrease the path length, making 11 gates a little pessimistic. However, a similar decrease would then be possible in the adder also.)

This total of 11 should be compared with a maximum adder depth of 6 gates evident from the circuit given by Brickell [1], which forms the basis of Figure 3. The figure of 11 seems to be optimal as the same depth was obtained by a hardware design tool at UMIST. The digit multiples cannot be computed earlier because they take more time to compute than the adder output, and so the adder is always waiting for them. Thus it is in general impossible to benefit from technique IX of Section 1.

4 Circuits for Montgomery's Multiplier and Comparative Speed.

Taking Montgomery's algorithm now, we use the same adder as before in order to make a fair comparison. So this is again the case of binary representations with inputs A and B having digits in $\{0,1,2\}$. For convenience, we assume that B has already been shifted up two places so that its lowest two digits are 0. Figure 3 gives the circuit analogous to Figure 2 for generating the quotient digit q', and makes use of $B_1 = B_0 = 0$.

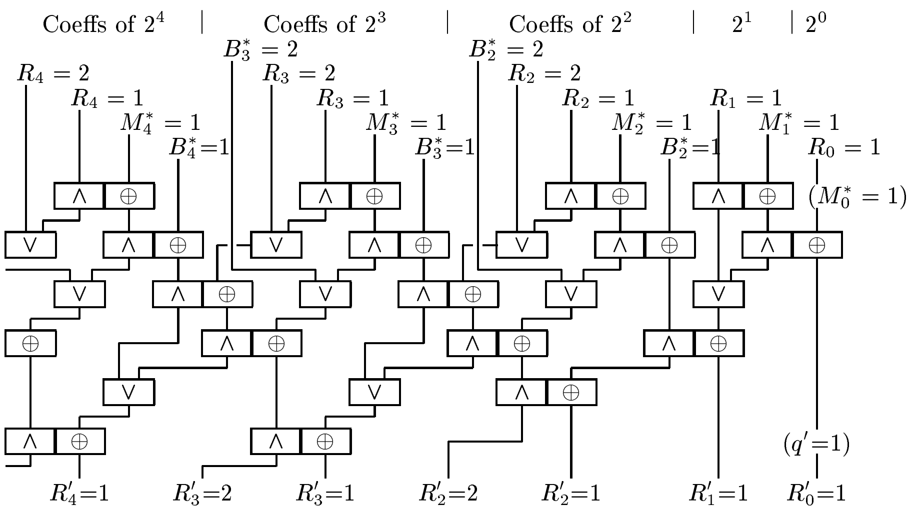

Fig. 3. Lower End of New Adder with B shifted 2 Places

Further simplifications arise in the diagram because, since $R + B^* + M^*$ and B are multiples of r, the input digits R_0 and M_0^* must be equal. (Indeed this entails $q = R_0 = M_0^*$, since $M_0 = 1$.) Hence inputs R_0 and M_0^* can be combined on one line as an input to the 2^1 bit slice, instead of appearing as two inputs to the 2^0 bit slice. As the lowest output comes from the 2^1 bit slice, the required shift down is possible. Lastly, neither R_0 nor R_1 is ever 2 because $R = 0$ initially, and the circuit shows that subsequently generated values of R retain this property.

The output q' $(= R'_0)$ determines the multiple of M to be input during the next cycle. It is obtained more quickly than the general digit of R', whose computation is therefore the limiting factor on the clock speed. For the usual adder, which is used for both methods, those general digits are obtained after the input has passed through an attainable maximum of 6 gates. Now the speed is roughly proportional to the depth of circuitry, Brickell's method led to an 11 gate critical path length, and 6 is the critical path length here. Hence this method apparently leads to a clock speed close to double that obtainable previously, namely an 11/6-fold increase. However, we shall have to modify this conclusion slightly after further analysis.

Before tackling this, we end the section with the observation that the direction of computation coincides with that of carry propagation. This is to our advantage in several ways. First, the multiplicand A can be converted into an irredundant form by keeping a carry bit which is updated as the digits of A are consumed, least significant first. Hence the digit multiples $B^* = A_iB$ need be selected from a set of only 2 (or r for the general radix). This is the technique X of Section 1. Secondly, by using the circuit of Figure 3, the lowest digit of the partial product is not redundant. Therefore, neither is the multiple q of M to be added at each iteration. However, under the Brickell scheme, q must be from a redundant digit range. In consequence, the adder here is simpler at least as far as some background control is concerned. This happens because non-redundant digits need 1 bit less in their representation, saving 1 gate of depth in the selections of both B^* and M^*.

5 Signal Broadcasting.

So far we have assumed that most of the hardware operating during a clock cycle is just what has already been studied. Unfortunately, this is not the case – in particular, signal broadcasting and amplification need to be performed.

The bits controlling the multiples q and A_i of M and B respectively need to be broadcast to each digit over the whole length of the adder in order for the next iteration to form or select the bits for M^* and B^*. Signal decay creates a need for about 1 extra amplification gate for every four outputs to which the signal needs distributing. So, with a 512 to 1024 bit modulus M, a tree structure with a depth of 5 gates is required to distribute these signals (see Figure 4). Moreover, clock signals need to be distributed in the same way to operate the registers.

In Brickell's scheme, the last bits to be formed are the two determining the multiple q of M. Hence, for a typical digit position in the adder, we need 5 gates to distribute these two bits to that position, 2 gates to use them to select the right digit in M^*, and, as noted before, up to 6 gates to perform the addition. This gives a total depth of about 13 gates for the adder proper – more than double that of our restricted view in the previous two sections.

However, specific outputs can be calculated more quickly by distributing signals preferentially to the associated inputs before they are fanned out elsewhere. In particular, we probably need just 2 rather than 5 gates to broadcast signals to the inputs which determine the next q. As before, 2 further gates are used to select the right input multiple of M, and, as in Figure 2, 11 more generate the required choice of q'. This gives a total depth of $11+2+2 = 15$ gates along a critical path in an implementation of the standard algorithm [1].

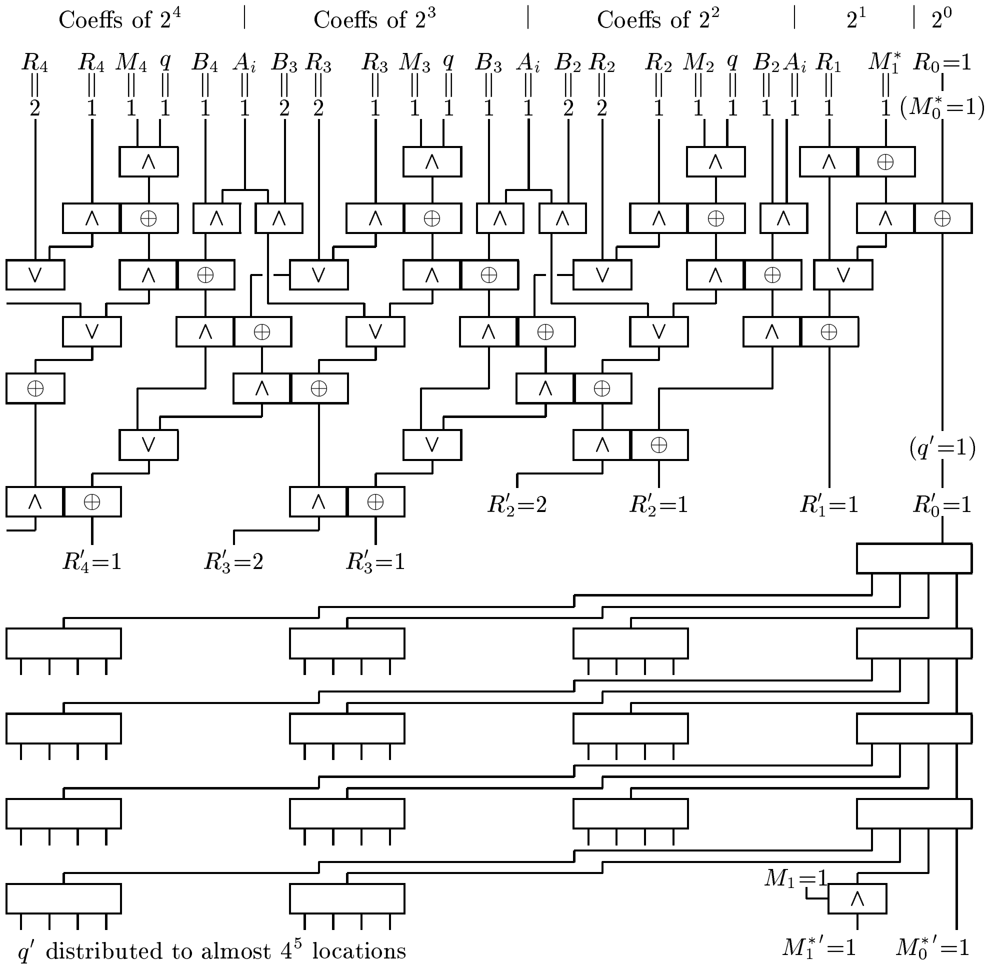

Fig. 4. New Adder with Distributed Signals

Under the new scheme, it is clear from Figure 3 that $q' = R'_0$ is formed 4 gates before the typical output bit of R'. Hence, using a tree of depth 4 rather than the 5 needed, that bit can be partly distributed over the length of the register in the current clock cycle while the adder settles. Then the complete distribution of the signal at the start of the next cycle requires just 1 more gate, and only 1 gate is needed to select the right input multiple of M. This gives a total adder depth of just $6 + 1 + 1 = 8$ gates.

Further improvement is possible. Consider Figure 4. This shows explicitly the initial gates needed to obtain M^* from M and q, and B^* from B and A_i. However, it assumes q and A_i already fully distributed, and M_1^* available from the previous clock cycle. The general output digit of R' is produced in a depth of only 7. So, once R'_0 is formed, there is a depth of 5 available for the full distribution of q' ($= R'_0$) before the adder settles, and $M_1^{*\prime}$ is easily computed during this. So the initial hypotheses about q and M_1^* can be met, and we obtain a circuit of depth 7.

The calculation of some values may be advanced so that $M^{*\prime}$ can be completely formed ready for the next iteration. This enables the gate depth to be reduced from 7 to the absolute lower bound of 6 determined by the main body of the adder. The details are provided in Figure 5, with the advanced inputs being marked as such. The speed of the combinational logic in the adder is the limiting factor there, which is what we wished to achieve.

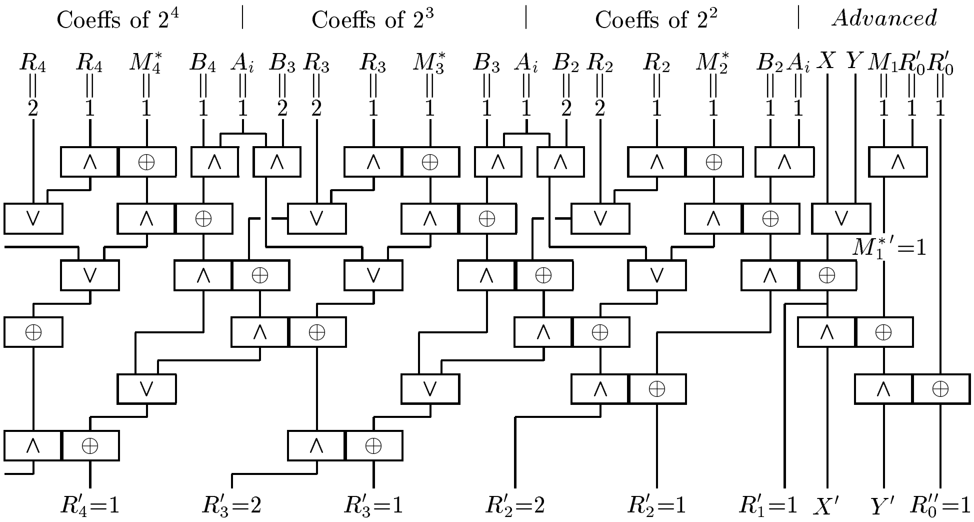

Fig. 5. Advanced Adder

Figure 5 assumes that R'_0 is available a full clock cycle ahead of the rest of R'. So R'_0 can be fully distributed to up to 4^5 places and ANDed with M to provide $M^{*\prime}$ in a register awaiting the start of the next cycle. This requires a depth of only 6 gates. Let R'' denote the value of the partial product one iteration after its value is R'. Then the output R''_0 is what we need to justify the assumption that R'_0 is available as an input. It is produced just in time.

To obtain this diagram, we have moved the circuitry for the coefficients of 2^1 and 2^0 back from the beginning of one clock cycle to the end of the previous one. The corresponding inputs and outputs represent bits further on in the calculation than those higher up the adder. In detail, the advanced inputs are defined by $X = R_1 \wedge M_1^*$, $Y = (R_1 \oplus M_1^*) \wedge R_0$ and $R'_0 = (R_1 \oplus M_1^*) \oplus R_0$. Thus, since initially $R = 0$ and $M^* = 0$, the first inputs for X, Y and R'_0 are also all zero.

For either circuit under the new algorithm, the extra cost over Brickell's scheme of making such efficient use of the combinatorial logic is two extra full length registers to store the partial or complete calculations of M^* and B^*. The prize is a reduction from 15 to just 6 gates along the maximum path length.

Next, a clock signal is required to operate the registers and other memory. The depth of the combinational logic needed to amplify the clock to each register bit is, as above, 5, which is less than the adder's logic. Hence the clocking should not add to the cycle time. However, we have to add on the set-up and hold and delay times for the operation of the latches on the registers. With typical CMOS technology, this amounts to about the delay of three 2-input XOR gates, which form the majority of the gates along the critical paths. So this reduces the speed-up ratio from an apparent 15/6 to a more realistic 18/9, i.e. the new design is about twice the old speed. Decreasing the number of latches to reduce this overhead can be achieved by increasing the radix, which we will not consider further here.

Finally, there is the problem of very long wires in our design. Through increased capacitance and resistance, this contributes to the power needed to drive the chip and cuts the clock speed. Although significant, we have mostly ignored this aspect in the belief that, by affecting equally all implementations using such chip architecture, it would not seriously upset the relative timing estimates. Given that the chip area is almost entirely covered by the adder, broadcasting information over the whole length of the adder will involve wires which are on average at least half the length of an edge of the chip.

One way of avoiding long wires is pipelining or, more generally, the use of systolic arrays. Given the large area of such chips, architectures with a fault-tolerant capacity might provide more cost-effective hardware. In both algorithms the chosen architecture needs to produce the multiple of M first. The new algorithm is particularly superior in this respect because, unlike Brickell's scheme, carries propagate away from this sub-calculation. So the natural way to treat this in a systolic array is in the same way as ordinary multiplication [8], treating digits together rather than single bits, and certainly working on only a single computation. This has been done by the second author in [17]. However, with Brickell's scheme, a systolic array could work only by interleaving independent modular multiplications. This would require a much larger chip or a time-consuming external interface.

6 Conclusions.

There is a vast world market for cryptography in which the over-riding concerns are security, authentication, and speed of encryption and decryption. If public keys are also required then the first two of these desires are currently obtainable to any extent only through the use of the RSA algorithm, and the last through dedicated hardware. We have been concerned with increasing the speed of hardware so that the RSA method becomes much more acceptable to the market. The first chips implementing RSA have already proved their worth with reasonable speeds but they have had high development costs.

We have shown above how to make much more efficient use of hardware for encrypting and decrypting messages using the RSA method by changing the method for modular multiplication. The hardware to achieve this is actually simpler than before, although at the cost of two extra full-length registers and some pre- and post- processing. Speed is improved two-fold when compared with the best existing hardware. It is clear that almost all the elements in the new design operate essentially continuously, with virtually no idle time during clock cycles as formerly. Thus it would now appear that no further significant speed-up is possible using this type of algorithm, assuming an optimal choice of number representation is made. Being simpler than before, the new method brings benefits at the design stage of the hardware, as well as during operation.

References

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

[2] E. F. Brickell, A Survey of Hardware Implementations of RSA, *Advances in Cryptology – CRYPTO '89*, ed. G. Brassard, Lecture Notes in Computer Science, vol. **435**, Springer-Verlag, 1990, pp. 368-370.

[3] W. Diffie and M. E. Hellman, New Directions in Cryptography, *IEEE Trans. Inform. Theory*, vol. **IT-22**, 1976, pp. 644-654.

[4] S. R. Dussé and B. S. Kaliski Jr., A Cryptographic Library for the Motorola DSP56000, in *Advances in Cryptology – EUROCRYPT '90*, ed. I. B. Damgård, Lecture Notes in Computer Science, vol. **473**, Springer-Verlag, 1991, pp. 230-244.

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

[6] M. D. Ercegovac and T. Lang, "On-the-Fly" Conversion of Redundant into Conventional Representations, *IEEE Trans. Comput.*, vol. **C-36**, 1987, pp. 895-897.

[7] M. Kameyama, S. Wei and T. Higuchi, Design of an RSA Encryption Processor Based on Signed-Digit Multivalued Arithmetic Circuits, *Systems & Computers in Japan*, vol. **21**, 1990, pp.21-31.

[8] 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.

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

[10] H. Orup, E. Svendsen and E. Andreasen, VICTOR An Efficient RSA Hardware Implementation, *Advances in Cryptology – EUROCRYPT '90*, ed. I. B. Damgård, Lecture Notes in Computer Science, vol. **473**, Springer-Verlag, 1991, pp. 245-252.

[11] J.-J. Quisquater and C. Couvreur, Fast Decipherment Algorithm for RSA Public-Key Cryptosystem, *Electronics Letters*, vol. **18**, 1982, pp. 905-907.

[12] 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.

[13] A. Selby and C. Mitchell, Algorithms for Software Implementations of RSA, *IEE Proceedings*, vol. **136E**, 1989, pp. 166-170.

[14] C. D. Walter, Fast Modular Multiplication using 2-Power Radix, *Intern. J. Computer Math.*, vol. **39**, 1991, pp. 21-28.

[15] C. D. Walter, Faster Modular Multiplication by Operand Scaling, *Advances in Cryptology – CRYPTO '91*, J. Feigenbaum, ed., Lecture Notes in Computer Science, vol. **576**, pp. 313-323, Springer-Verlag, 1992.

[16] C. D. Walter and S. E. Eldridge, A Verification of Brickell's Fast Modular Multiplication Algorithm, *Intern. J. Computer Math.*, vol. **33**, pp. 153-169, 1990.

[17] C. D. Walter, Systolic Modular Multiplication, *IEEE Trans. Comput.*, vol. **42**, pp. 376-378, 1993.