Faster Modular Multiplication by Operand Scaling

Colin D. Walter

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

Abstract. There are a number of techniques known for speeding up modular multiplication, which is the main arithmetic operation in RSA cryptography. This note shows how to gain speed by scaling the modulus. Resulting hardware is limited only by the speed of addition[1]. Detailed analysis of fan out shows that over existing methods the speedup is potentially as much as two-fold. This is because the addition and fan out can now be done in parallel. Of course, in RSA the modulus can be chosen to need no scaling, so that most of the minor extra costs are eliminated.

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

1 Introduction

One of the motivations for studying fast modular multiplication is its use in cryptography, including the RSA algorithm [5]. That algorithm provides potentially the most widely useful system as it appears to be arbitrarily secure. However, its arithmetic intensity requires dedicated hardware if it is to be used in a real-time system working with bulk data.

A number of techniques are already known for improving the speed of hardware for modular multiplication of integers. These are surveyed in, for example, Eldridge & Walter [3]. Most can be combined without difficulty with the modification suggested here, and so our contribution is presented in terms of the essential, basic techniques described by Brickell [2]. Fundamental there is the use of a truncated partial product and truncated modulus to determine, with sufficient accuracy, the correct multiple of the modulus to subtract during each of the repeated addition cycles that perform the multiplication. This had been used for some time in the case of real number division, being reported on by, for example, Atkins [1] and Taylor [6] in cases of number representations with radix

[1] J.-J. Quisquater informed me at the conference that he had spoken on a similar technique for software in the rump session at EUROCRYPT '90, but nothing appears in the Proceedings.

greater than 2. In a recent paper [4], Ercegovac and Lang show how improve this technique for division by scaling both numerator and divisor by the same amount in order to obtain known, fixed, most significant digits for the divisor. With these digits known, the hardware logic for deciding the multiple of the divisor to subtract is much simpler. So the clock may be run faster and the quotient obtained more quickly. A similar procedure works for modular multiplication by scaling the modulus, and we present the details for this here.

As in the case of division, the speed-up consequent from this technique derives from the reduced complexity of the hardware logic for deciding the multiple of the divisor to subtract from the dividend. Analysis by Eldridge and Walter in [3] of logic for the usual modular multiplication algorithm shows that, as with division (see [6]), this complexity normally determines the critical path length in the hardware, and so the clock speed and overall time.

The overheads entailed by employing the technique here are minimal. Initially, scaling of the modulus would probably be done by software. In the case of the RSA algorithm the same modulus is used over and over again, so that scaling done once for all is very cheap. During computation the registers need to be a digit or so larger, which affects the chip area only marginally. Also one or two more iterations of the main loop need to be done. This hardly affects the time at all. After the loop, the result may be too large and a few extra subtractions of the original modulus may be necessary. This is potentially the most expensive overhead as the original modulus may need to be reloaded or kept in a further register. Overall, the hardware is almost the same as before, with slightly adjusted parameters, except for the improved logic mentioned above. The gain in efficiency against the minor overheads is worked out in detail in this paper, with encouraging results.

2 Overview of the Algorithm

We begin by noting that fast modular multiplication is usually done by repeated cycles involving shifting and addition, as in ordinary multiplication, together with a simultaneous modular subtraction. Thus, each cycle also needs to predict the multiple of the modulus to subtract in the next cycle.

Suppose we represent numbers with radix r. If we wish to calculate the residue R of $(A{\times}B)\ mod\ M$, or indeed the integer quotient $Q = (A{\times}B)\ div\ M$, then, with some detail yet to be explained, the basis of the algorithm in [2] is the following:

```
Type  Index       =  0..MaxIndex ;
      Register    =  Array[Index] of Digit ;

Procedure ModMult( A,B,M : Register ; Var R,Q : Register ) ;
{ Pre-Conditions : M_min ≤ M ≤ M_max  and A,B ≥ 0 }
{ Post-Conditions: A×B = Q×M + R  and Top(R) ≤ L   }

Var   J  : Index ;

      Function Quotient(ToprR,TopM : Int) : Digit ;
      { Post-Condition : Quotient ≈ (ToprR) div TopM  }
      Begin ... End ;

Begin { ModMult }
      R := 0 ;  Q := 0 ;
      For J := MaxIndex DownTo 0 do
      Begin { Loop Invariant: Top(R) ≤ L  and R ≥ 0  }
            Q[J] := Quotient(Top(r*R),Top(M)) ;
            R    := r*R + A[J]*B - Q[J]*M ;
      End ;
End ; { ModMult }
```

It is fairly straightforward to see that the output satisfies $A{\times}B = Q{\times}M + R$. For speed, the quotient digits are generated by only considering the topmost digits of the partial product R and the modulus M. These are extracted by the function Top, which truncates a fixed number (usually most) of the lowest digits. By allowing the digits of Q to lie in a sufficiently wide range, the accumulating partial product can be kept fairly small, being bounded through some fixed L. Appropriate choices make R less than $2M$, but not necessarily less than M. So the final output may fail to be the least non-negative residue of $A{\times}B$ modulo M, but it is easy to subtract an extra M to obtain $(A{\times}B)\ mod\ M$, if necessary. The precise conditions required for undefined constants such as L are given in [7]. Such detail is not needed here, although we look at L in section 4.

3 Scaling the Modulus

Let q be the number of the most significant digits of the modulus M which the function *Quotient* needs and suppose M has a standard, non-redundant representation, i.e. digits in the range $0..r{-}1$. Assume inputs are shifted as necessary to give the modulus exactly m digits, so that the hardware function Top just truncates the $m-q$ least significant digits of both M and the partial product R. We want to scale the modulus M by a factor f such that fM has its q most significant digits fixed, say, to M_{fix}.

The revised algorithm uses fM, with an appropriate shift, in place of M. One benefit of this is that *Quotient* is easier to calculate because it no longer depends on any digits of $Top(M)$, as they are fixed. This saves minimal hardware area, but, more importantly, shortens the cycle time of each iteration. In particular, if *Quotient* is performed by doing $div\ (Top(M){+}1)$ and $M_{fix} = r^q{-}1$ then the implied integer division is by r^q and can be done just by shifting. The penalties of the technique include the pre-calculation of f and fM (which may be needed in non-redundant form), an increase in register lengths by the number of digits in f, and, if necessary, up to $2f$ final subtractions of M from the output, which may otherwise be nearly as large as $2fM$.

Let us now show how to calculate f. Suppose fM has p non-redundant digits. Then, for the q most significant digits of fM to be M_{fix}, f must satisfy

$$M_{fix} \quad \le \quad r^{q-p} fM \quad < \quad M_{fix}{+}1$$

This is equivalent to demanding that f lie in the real interval

$$[\ r^{p-q} M_{fix}/M \quad , \quad r^{p-q}(M_{fix}{+}1)/M\ [$$

This must be of length at least 1 in order that it always contain an integer which can be chosen as the value of f. The condition for this is $M \le r^{p-q}$, in other words, fM has at least q more digits than M. Ideally, p should be picked minimally. Thus, f could be calculated by brute force using

$$f \quad = \quad (\ r^m \times (M_{fix}{+}1) - 1\ )\ \ div\ M \qquad \text{with} \quad p = m + q \qquad (*1)$$

A more efficient approach may be desirable, one which derives f from a truncated value of M. Suppose q' digits of M are needed to find such an f. Let Top' be the function that provides these. From the property $r^{m-q'}Top'(M) \le M < r^{m-q'}(Top'(M){+}1)$ we can approximate the ends of the interval above to obtain the strict sub-interval

$$[\ r^{p-q}M_{fix}\ /\ r^{m-q'}Top'(M) \quad , \quad r^{p-q}(M_{fix}{+}1)\ /\ r^{m-q'}(Top'(M){+}1)\ ]$$

This has length at least 1 precisely when

$$Top'(M) \times (\ Top'(M){+}1\ ) \quad \le \quad r^{p-m+q'-q}(\ Top'(M) - M_{fix}\ )$$

By viewing this as a quadratic in $Top'(M)$, it is most difficult to satisfy at the extreme of the range $[r^{q'-1}, r^{q'}-1]$ which is furthest from the turning point $(r^{p-m+q'-q}-1)/2$. Unfortunately, for $p = m+q$ it does not hold when $Top'(M) = r^{q'}-1$, and so we need to increase p to $p = m+q+1$. Then the lower limit $r^{q'-1}$ is the harder to satisfy and we require

$$r^{q'-1} \times (\ r^{q'-1} + 1\ ) \quad \le \quad r^{q'+1} \times (\ r^{q'-1} - M_{fix}\ )$$

or, equivalently,

$$M_{fix} \quad < \quad r^{q'-1} \; - \; r^{q'-3} \; .$$

Since $M_{fix} < r^q$, this will always hold by taking $q' = q+2$, (and usually when $q' = q+1$). Thus a solution which yields f from only the topmost digits of M is given by

$$f \;\; = \;\; (r^{q'+1}(M_{fix}{+}1)) \; div \; (Top'(M){+}1) \quad \text{with } p = m{+}q{+}1 \text{ and } q' = q{+}2 \quad (*2)$$

To summarise the results of this section, we begin by choosing a suitable M_{fix} with q digits which would make computation of *Quotient* easy if M_{fix} were given by Top applied to the modulus. Next we replace the modulus M by fM where f is as defined in either $(*1)$ or $(*2)$. Lastly, when running the modular multiplication algorithm with the new modulus, perform extra subtractions of the original modulus as necessary after the main loop to obtain the least non-negative residue. This can be done with about $q+2$ shifts and subtractions of digit multiples of the original modulus because $(*1)$ and $(*2)$ yield $f \le r^{q+1}$ and $f \le r^{q+2}$ respectively in the worst cases.

4 Solutions for Radix 2

Now let us look at the saving in computational time by seeing how the hardware is affected in the case of radix 2. Assume that M has already been replaced by fM and shifted, so that $Top(M) = M_{fix}$. Suppose also that M is in usual non-redundant binary form. However, let R have digits from 0..2. Speed is obtained mainly by using this redundant representation in order to curtail carry propagation to only one or two places during the addition. This enables digit operations for addition to be carried out in parallel. One choice for *Quotient* which is discussed in [7] is

$$Quotient(ToprR, TopM) \quad = \quad ToprR \; div \; (TopM{+}1)$$

A sensible choice for M_{fix} is therefore $2^q - 1$ so that div can be performed simply by shifting. A value for q which makes the algorithm work now has to be determined.

The loop invariant for the addition cycle must be preserved. So $Top(R) \le L$ must imply the condition

$$Top(\; 2R + A[J]{\times}B - (Top(2R) \; div \; 2^q){\times}M \;) \quad \le \quad L \qquad\qquad (*3)$$

since the left side is the value of $Top(R)$ for the next iteration. A scheme following the lines of Brickell [2] computes $(A{\times}B) \; mod \; M$ as $((SA{\times}B) \; mod \; SM) \; div \; S$ for some shift factor S, with Top truncating appropriately more digits. Here S

is chosen sufficiently large for the input $A[J]\times B$ not to affect any of the thus-redefined top digits in the value of R to which it contributes. So, that term may be ignored. Now, looking at top digits only,

$$\begin{aligned}
&2R \;-\; (Top(2R)\ div\ 2^q)\times M \\
&\quad <\; (\ Top(2R)+1\ )2^{m-q} \;-\; (\ Top(2R)\ div\ 2^q\ )\times(2^m - 2^{m-q}) \\
&\quad =\; \{Top(2R) - 2^q(Top(2R)\ div\ 2^q)\}\times 2^{m-q} + 2^{m-q} + (Top(2R)\ div\ 2^q)\times 2^{m-q} \\
&\quad =\; (\ Top(2R)\ mod\ 2^q\ )\times 2^{m-q} \;+\; 2^{m-q} \;+\; (Top(2R)\ div\ 2^q)\times 2^{m-q}
\end{aligned}$$

Applying Top to this, and noting the strictness of the inequality, ensures the condition $(*3)$ is met if

$$Top(2R)\ mod\ 2^q \quad + \quad Top(2R)\ div\ 2^q \quad \leq \quad L \qquad (*4)$$

Here R may equal M. Thus $Top(R)$ may be at least as great as $Top(M) = 2^q - 1$. Hence $Top(2R) = 2^{q+1} - 1$ is possible, and for this the inequality requires $L \geq 2^q$. We will show that $(*4)$ is satisfied by taking $L = 2^q$ and $q \geq 2$. So suppose this is the value of L and that $Top(R) \leq L$. As digits of R are at most 2, we have $Top(2R) \leq 2Top(R) + 2$ when the multiplication is done by shifting. Thus $Top(2R) \leq 2L + 2 = 2^{q+1} + 2$. The left side of $(*4)$ is a saw-tooth function of $Top(2R)$, with increasing maxima before each multiple of 2^q. So $(*4)$ is satisfied if it holds at the last value, when $Top(2R) = 2^{q+1} + 2$, and at the previous maximum, when $Top(2R) = 2^{q+1} - 1$. Both are easily seen to satisfy the inequality if $q \geq 2$, confirming the validity of the choice for L. The output conveniently satisfies $R < 2M$ because $Top(R) \leq L < 2L{-}2 = 2Top(M) \leq Top(2M)$ and similarly $Q[J] \leq 2$ because $Top(2R)\ div\ (Top(M){+}1) \leq (2L{+}2)\ div\ L = 2$.

There are no solutions at all for $q \leq 1$. Larger values of q progressively simplify the hardware, but each increase by 1 costs an extra digit position in registers, an extra addition cycle, and another final subtraction of a shifted digit multiple of the original modulus.

5 Improved Circuits for Radix 2

Now recall that hardware clock speed is limited by the longest path in the circuit from input to output. The length of the shortest possible clock cycle is approximately the sum of delay times associated with the gates on such a path. This in turn is roughly proportional to the number of such gates. In [3], Figure 2, there is a circuit for implementing the software addition cycle with a delay carry adder. This uses the same redundant number system assumed at the start of the previous section to allow parallel digit operations. Generating the new value for R as well as the *Quotient* digit for the next iteration results in a critical path length of 11 XOR gates compared with the 6 needed for calculating a typical output digit. However, like the clock signals, the *Quotient* digit needs to be broadcast subsequently to each place in the adder. We will assume the technology requires a tree 5 gates deep to do this for a 512 to 1024 bit modulus. Then the correct multiple of M can be selected ready for the next iteration using two more gates. If we preferentially broadcast to the topmost inputs first (2 gates) then the path length at the top end is actually 11+2+2 = 15 gates, whilst that for a typical output digit is 6+5+2 = 13 gates.

If the *Quotient* digit were to be computed earlier, then the fanning out of this information could be overlapped with the current addition to reduce the critical path length closer to the theoretical minimum of 6, which is the number of gates for finding a typical output digit in the adder. We now show how scaling the modulus makes this possible.

Suppose we take $q \geq 2$. Then the most significant digit of R has index at most m because $R < 2M$. Indeed, if a suffix $_i$ denotes the digit coefficient of 2^i in a number representation, then $R_m = 0$ or 1. This bound on R determines the size of registers as needing $m{+}1$ digits. So the subtraction of $Q_J{\times}M$ might be implemented here by adding Q_J times the complement $(2^{m+1}{-}1) - M$ together with an initial carry Q_J at the bottom end and ignoring an overflow of $Q_J{\times}2^{m+1}$. Call this input M^*, and assume that the inherent non-zero digit multiples are obtained by shifting so that its digits are bits. Then, because $M_{fix} = 2^q - 1$ gives $M_m = 0$ and $M_i = 1$ for $m{-}1 \geq i \geq m{-}q$, the topmost digits of M^* satisfy $M^*_m = 0$ or 1, and $M^*_i = 0$ for $m{-}1 \geq i > m{-}q$. The initial carry does not propagate up the adder more than a couple of places, and so it does not affect the top digits. Finally, Q_J has the simple formula $2R_m + R_{m-1}$.

Now take $q = 4$. The top end of the delay carry adder simplifies to the typical bit slice illustrated in Figure 1 because the most significant digits of input $A_J{\times}B$ are 0. Using the various bit values just described, consequent simplifications to the bit slice yield most of the top end of Figure 1. However, once R is computed, part of the next addition cycle can be performed on its topmost digits to convert them nearer to non-redundant form. This is illustrated in that part of the figure below the dotted line which marks the end of one iteration of the software algorithm proved above. Enough has been done there to remove the possibility that $R_{m-4} = 2$, i.e. $(2R)_{m-3} = 2$, which explains the other simplification to the input.

The advantage of starting part of the next iteration is that the quotient digit can be calculated earlier in the cycle. Here it appears after a maximum of 3 gates, rather than the 11 noted above: a substantial reduction. Since it is actually computed earlier than a typical digit output from lower down the adder, it is clearly possible after further modifications to fan out this information in parallel with the addition rather than sequentially after it, thereby reducing the critical path length to that of the adder (6 here). This requires a lot of the top digit calculations to be considerably advanced, but it would enable chips using scaling to run at about double the speed of others with the only significant cost being an extra register to hold M^*. To build such a circuit is just tedious development and we omit the details.

Finally, we consider how to add surrounding detail to Figure 1 without trying to advance the quotient digit computations still further. If all the topmost input bits needed for calculating the quotient digit are already in position at the start of a clock cycle, then generating the same inputs for the next cycle requires 3 gates for the quotient digit, 2 for fanning it out and 2 for selecting digits for M^*: a total of 7. Let us use 5 gates to completely disseminate the new quotient digit to all digit positions. So this makes a total depth of 3+5 = 8 gates at the top end. The main part of a 512- to 1024-bit adder then just needs a depth of 2 gates for selecting M^* as well as the 6 gates of the adder itself. This makes the critical path length just 8, compared to 15 without modulus scaling. Counting 2 gates as the equivalent of the set-up and hold times for registers, the hardware presented here should be able to be run at about (15+2)/(8+2) times the speed of comparable hardware without a scaled modulus, i.e. 70% faster.

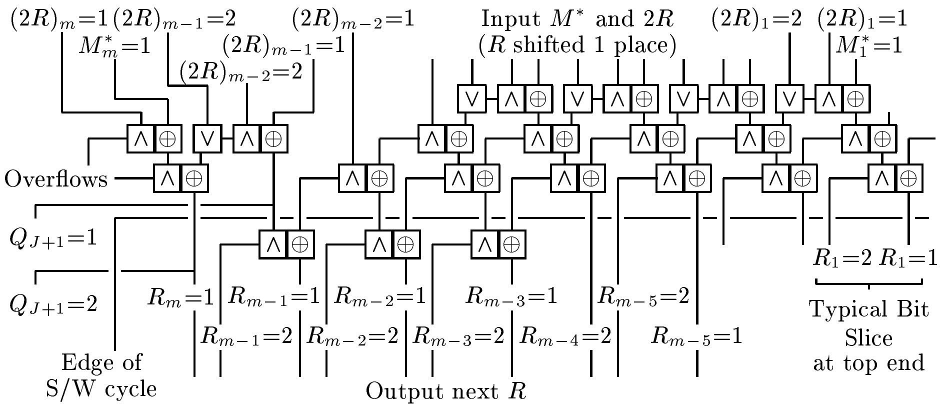

Fig. 1. Adder for radix 2 when $q = 4$ and $M_{fix} = 15$

6 Final Detail and Conclusions

We have shown how to scale the modulus for modular multiplication to potentially double the speed of hardware, giving sufficient detail to achieve a speedup factor of 70%. The cost for radix 2 involved 4 extra bit positions in registers and consequently 4 extra clock cycles – less than 1% in space or time for typical RSA applications. For the full doubling of speed an extra register holding M^* is required. Further penalties are slight. They concern pre- and post- processing. Initial scaling is cheap when the modulus is much used as it is done once for all. The output is bounded by $2fM$, where f is the scaling factor. Hence M needs to be loaded and subtracted as necessary. However, in RSA cryptography this does not need to be done until decryption, and it can be avoided entirely by choosing a modulus which needs no scaling.

References

[1] D. E. Atkins, *Higher Radix Division using Estimates of the Divisor and Partial Remainders*, IEEE Trans. Comp., vol. **C-17**, 1968, pp. 925-934.

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

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

[4] M. D. Ercegovac and T. Lang, *Simple Radix-4 Division with Operands Scaling*, IEEE Trans. Comp., vol. **39**, 1990, pp. 1204-8.

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

[6] G. S. Taylor, *Radix 16 SRT Dividers with overlapped Quotient Selection Stages*, Proc. IEEE 7th Symp. Comp. Arith., 1985, pp. 64-73.

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