Montgomery's Multiplication Technique: How to Make it Smaller and Faster

Colin D. Walter

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

Abstract. Montgomery's modular multiplication algorithm has enabled considerable progress to be made in the speeding up of RSA cryptosystems. Perhaps the systolic array implementation stands out most in the history of its success. This article gives a brief history of its implementation in hardware, taking a broad view of the many aspects which need to be considered in chip design. Among these are trade-offs between area and time, higher radix methods, communications both within the circuitry and with the rest of the world, and, as the technology shrinks, testing, fault tolerance, checker functions and error correction. We conclude that a linear, pipelined implementation of the algorithm may be part of best policy in thwarting differential power attacks against RSA.

Key Words: *Computer arithmetic, cryptography, RSA, Montgomery modular multiplication, higher radix methods, systolic arrays, testing, error correction, fault tolerance, checker function, differential power analysis, DPA.*

1 Introduction

An interesting fact is that the faster the hardware the more secure the RSA cryptosystem becomes. The effort of cracking the RSA code via factorization of the modulus M doubles for every 15 or so bits at key lengths of around 2^{10} bits [10]. However, adding the 15 bits only increases the work involved in decryption by $((1024+15)/1024)^2$ per multiplication and so by $((1024+15)/1024)^3$ per exponentiation, i.e. 5% extra! Thus speeding up the hardware by just 5% enables the cryptosystem to become about twice as strong without needing any other extra resources. Speed, therefore, seems to be everything. Indeed it is essential not just for cryptographic strength but also to enable real time decryption of the large quantities of data typically required in, for example, the use of compressed video.

On the other side of the Atlantic, the first electronic computer is generally recognised to be the Colossus, designed by Tommy Flowers, and built in 1943-4 at Bletchley Park, England. It was a dedicated processor to crack the Enigma code rather than a general purpose machine like the ENIAC, constructed slightly later by John Eckert and John Mauchly in Philadelphia. With the former view of history, cryptography has a fair claim to have started the (electronic) computer age. Breaking Enigma depended on a number of factors, particularly human weakness in adhering to strict protocols, but also on inherent implementation weaknesses.

Timing analysis and differential power analysis techniques [12] show that RSA cryptosystems mainly suffer not from lack of algorithmic strength but also from implementation weaknesses. No doubt governments worked on such techniques for many years before they appeared in the public domain and have developed sufficiently powerful techniques to crack any system through side channel leakage. Is it a coincidence that the US became less paranoid about the use of large keys when such techniques were first published? Now that there seems to be no significant gain to be made from further improvement of algorithms, the top priority must be to prevent such attacks by reducing or eliminating variations in timing, power and radiation from the hardware.

This survey provides a description of the main ideas in the hardware implementation of the RSA encryption process with an emphasis on the importance of Montgomery's modular multiplication algorithm [17]. It indicates the main publications where the significant contributions are to be found, but does not attempt to be exhaustive. The paper discusses the major problems associated with space- and time- efficient implementation and reviews their solution. Among the issues of concern are carry propagation, digit distribution, buffering, communication and use of available area. Finally, there are a few remarks on the reliability and cryptographic strength of such implementations.

2 Notation

An RSA cryptosystem [20] consists of a modulus M, usually of around 1024 bits, and two keys d and e with the property that $A^{de} \equiv A \bmod M$. Message blocks A satisfying $0 \le A < M$ are encrypted to $C = A^e \bmod M$ and decrypted uniquely by $A = C^d \bmod M$ using the same algorithm for both processes. $M = PQ$ is a product of two large primes and e is often chosen small with few non-zero bits (e.g. a Fermat prime, such as 3 or 17) so that encryption is relatively fast. d is picked to satisfy $de \equiv 1 \bmod \phi(M)$ where ϕ is Euler's *totient* function, which counts the number of residue classes prime to its argument. Here $\phi(M) = (P-1)(Q-1)$ so that d usually has length comparable to M. The owner of the cryptosystem publishes M and e but keeps secret the factorization of M and the key d. Breaking the system is equivalent to discovering P and Q, which is computationally infeasible for the size of primes used.

The computation of $A^e \bmod M$ is characterised by two main processes: modular multiplication and exponentiation. Here we really only consider computing $(A \times B) \bmod M$. Exponentiation is covered in detail elsewhere, e.g. [9],[28]. To avoid potentially expensive full length comparisons with M, it is convenient to be able to work with numbers A and B which may be larger than the modulus. Assume numbers are represented with base (or radix) r which is a power of 2, say $r = 2^k$, and let n be the maximum number of digits needed for any number encountered. Here r is determined by the multiplier used in the implementation: an $r \times r$ multiplier will be used to multiply two digits together. The hardware also determines n if we are considering dedicated co-processors with a maximum register size of n digits. Generally, M must have fewer bits than the largest representable number; how much smaller will be determined by the algorithm used.

Except for the exponent e, each number X will have a representation of the form $X = \sum_{i=0}^{n-1} x_i r^i$. Here the ith digit x_i often satisfies $0 \leq x_i < r$, yielding a *non-redundant* representation, i.e. one for which each representation is unique. The modulus M will always have such a representation. However, in order to limit the scope of interactions between neighbouring digits, a wider range of digits is very useful. Typically this is given by an extra (carry) bit so that digits lie in the range $0 \,..\, 2r-1$. For example, the output from a carry-save adder (with $r = 2$) provides two bits for each digit and so, in effect, is a *redundant* representation where digits lie in the range $0 \,..\, 3$ rather than the usual $0 \,..\, 1$. Here the k-bit architecture means that our adder will probably propagate carries up the length of a single digit, providing a "save" part in the range $0 \,..\, r-1$ and a "carry" part representing a small multiple of r. A digit x split in this way is written $x = x_s + rx_c$. In fact, the addition cycle in our algorithms involves digit products, so that a double length result is obtained. Hence, with some notable exceptions, the carry component regularly consists of another digit and a further one or two more bits. In a calculation $X \leftarrow X+Y$, the digit slices can operate in parallel, with the jth digit slice computing

$$x_{j,s} + rx_{j,c} \leftarrow x_{j,s} + x_{j-1,c} + y_j$$

The extra range of x_j given through the carry component keeps x_j from having to generate a separate carry which would need propagating. Since only old values appear on the right, not new ones, carry propagation does not force sequential digit processing. The digit calculations can therefore be performed in parallel when there is sufficient redundancy.

3 Digit Multipliers and their Complexity

Early modular multiplication designs treated radix 2, 4 or even 8 separately at a gate level. With rapidly advancing technology, these have had to be replaced by the generic radix r viewpoint which is now essential for a better understanding of the general principles as well as for a modular approach to design and for selecting from parametrised designs to make best use of available chip area. Today's embedded cryptosystems are already using off-the-shelf 32-bit multipliers [21] where reduction of the critical path length by one gate makes virtually no difference to the speed − and would probably cost too much in terms of additional silicon area. These $r \times r$ multipliers form the core of an RSA processor, forming the digit-by-digit products.

In the absence of radical new algorithms we need to be aware of complexity results for multiplication but prepared to use pre-built state-of-the-art multipliers which contain years of experience in optimisation and which come with a guarantee of correctness. Practical planar designs are known for multipliers which are optimal with respect to some measure of time and area [3], [4], [15], [16], [19], [23]. A reasonable assumption which held until recently is that wires take area but do not contribute noticeably to time. Under such a model, $Area{\times}Time^2$ complexity for a k-bit multiplication is bounded below by k^2 [3] and this bound can be achieved for any time in the range $\log k$ to $\sqrt{k}$ [16]. Such designs tend to use the Discrete Fourier Transform and consequently involve large constants in their measures of time and area. There are more useful designs which are asymptotically poorer but perform better if k is not too large. The cross-over point is greater than the size of the digits here [15]. So classical multiplication methods are preferable. Indeed, for a chip area of around 10^7 transistors devoted entirely to RSA and containing hardware for a full length digit multiplication $a_i{\times}B$, $k = 32$ or 64 is the maximum practical since there must be space for registers and for other operations such as the modular reduction. In the latest technology wires have significant capacitance and resistance and there is a requirement from applications as diverse as cell phones and deep space exploration for low power circuitry. This requires a different model which is more sensitive to wire length and for which results are only just emerging [18].

Speed is most easily obtained by using at least n multipliers to perform a full length multiplication $a_i{\times}B$ (or equivalent) in one clock cycle. If we were not worried about modular reduction, the carry propagation problem could be taken care of by pipelining this row of multipliers (Fig. 1): $a_i b_j$ is then computed by the jth multiplier during the $i{+}j$th clock cycle, generating a carry which is fed into the next multiplier, which computes $a_i b_{j+1}$ in the next cycle.

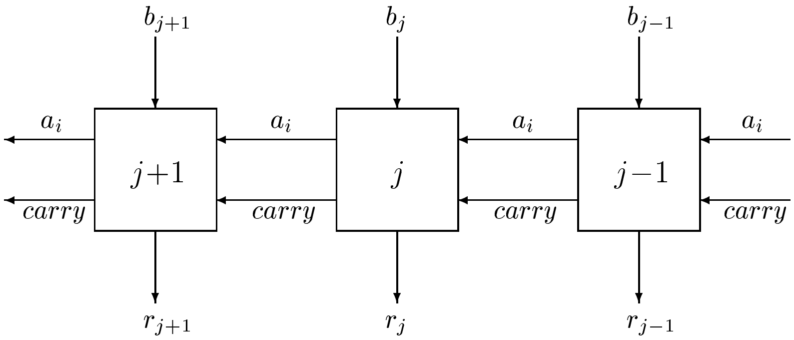

Figure 1. A Pipeline of Multipliers for $R \leftarrow a_i{\times}B$.

Is this set-up fast enough for real-time processing? A realistic measure of the speed required for real-time decryption is provided by an assumption that the internal bus speed is in the order of one k-bit digit per clock cycle. If the k-bit multiplier operates in one cycle with no internal pipelining then computing $A{\times}B$ takes n cycles using n multipliers in parallel in order to compute $a_i{\times}B$ in one cycle. The throughput is therefore one digit per cycle for a multiplication.

Unfortunately, since RSA decryption requires $O(nk)$ multiplications, we may actually need a two dimensional array of multipliers rather than just a row of them to perform real-time decryption.

Of course, there is an immediate trade-off between time and area. Doubling the number of digit multipliers in an RSA co-processor allows the parallel processing of twice as many digits and so halves the time taken. This does not contradict the $Area \times Time^2$ measure being constant for non-pipelined multipliers, although it appears to require less area than expected for the speed-up achieved. Having two rows of digit multipliers with one row feeding into the other creates a pipeline (now with respect to digits of A) which doubles the throughput that the complexity rule expects. This indicates that choosing the largest r possible for the given silicon area may not be the best policy; a pipelined multiplier or several rows of smaller multipliers may yield better throughput for a given area.

Finally, despite a wish to use well-established multipliers, differential power analysis (DPA) attacks on cryptographic products [13] suggest that special purpose multipliers need to be designed for some RSA applications which contain the secret keys, such as smart cards. Briefly, switching a gate consumes more power than not doing so. Inputs for which Hamming weights are markedly more or less than average could therefore have a power consumption with measurable deviation from average and reveal useful information to an attacker. This is true of today's optimised multipliers.

4 Modular Reduction & the Classical Algorithm

The reduction of $A \times B$ to $(A \times B) \bmod M$ can be carried out in several ways [11]. Normally it is done through interleaving the addition of $a_i B$ with modular reductions instead of computing the complete product first. This makes some savings in hardware. In particular, it enables the partial product to be kept inside an n-digit register without overflow into a second such register. Each modular reduction involves choosing a suitable digit q and subtracting qM from the current result. The successive choices of digit q can be pieced together to form the integer quotient $Q = \lfloor (A \times B)/M \rfloor$ or a closely related quantity:

Classical Modular Multiplication Algorithm:

{ Pre-condition: $0 \le A < r^n$ }

```
R := 0 ;
For i := n-1 downto 0 do
Begin
    R  := r×R + a_i×B ;
    q_i := R div M ;
    R  := R - q_i×M ;
End
```

{ Post-condition: $R = A \times B - Q \times M$
and, consequently, $R \equiv (A \times B) \bmod M$ }

If we define $A_i = \sum_{j=i}^{n-1} a_j r^{j-i}$ so that $A_i = rA_{i+1}+a_i$ and use similar notation for Q_i then it is easy to prove by induction that at the end of the ith iteration $R = A_i{\times}B - Q_i{\times}M$. Hence the post-condition holds.

Brickell [5] observes that R div M need only be approximated using the top bits of R and M in order to keep R bounded above. In particular [24], suppose that $\underline{M}$ is the approximation to M given by setting all but the k+3 most significant bits of M to zero, and that $\underline{M}'$ is obtained from $\underline{M}$ by incrementing its k+3rd bit by 1. Then $\underline{M} \leq M < \underline{M}' \leq (1{+}2^{-2}r^{-1})\underline{M}$. Assume $\underline{R}$ is given similarly by setting the same less significant bits to zero and that the redundancy in the representation of R is small enough for $R < \underline{R}{+}M$ to hold. The approximation to q_i which is used is defined by the integer quotient $\underline{q_i} = \lfloor \underline{R}/\underline{M}' \rfloor$. Then

$$\begin{aligned}\underline{q_i} = \lfloor \underline{R}/\underline{M}' \rfloor \ &\leq \lfloor R/M \rfloor = q_i \\ &\leq 1 + \underline{R}/M \leq 1 + (1{+}2^{-2}r^{-1})\underline{R}/\underline{M}' \leq 1 + (1{+}2^{-2}r^{-1})(\underline{q_i} + 1)\end{aligned}$$

so that $q_i{-}\underline{q_i} \leq 1 + (1{+}2^{-2}r^{-1})^{-1} + q_i(1{+}4r)^{-1}$ from which, at the end of the loop, $R < (1{+}q_i{-}\underline{q_i})M < 2M + (1{+}4r)^{-1}q_iM$. Assume the (possibly redundant) digits a_i are bounded above by a. We will establish inductively that

$$R < 3M + a(1{+}3r)^{-1}B$$

at each end of every iteration of the loop. Using the bound on the initial value of R in the loop yields $q_iM \leq rR{+}aB < 3rM{+}a(1{+}r(1{+}3r)^{-1})B$ so that, from the above, the value for R at the end of the loop satisfies $R < 2M{+}(1{+}4r)^{-1}(3rM + a(1{+}r(1{+}3r)^{-1})B) < 3M + a(1{+}3r)^{-1}B$, as desired. Naturally, more bits for a better approximation to q_i will yield a lower bound on R, whilst fewer will yield a worse bound or even divergence.

The output from such a multiplication can be fed back in as an argument to a further such multiplication without any further adjustment by a multiple of M providing the bound on R does not grow too much. Assuming, reasonably, that redundancy in A is bounded by $a \leq 2r$ we obtain $R < 3M + \frac{2}{3}B$ from which i) if $B \geq 9M$ then B is an upper bound for successive modular multiplications and indeed the bound decreases towards a limit of $9M$, and ii) if $9M$ is an upper bound for the input it is also an upper bound for the output. Hence only a small number of extra subtractions at the end of the exponentiation yields M as a final upper bound.

There are no communication or timing problems when there is just one multiplier. So, for the rest of this article, we will assume that the hardware for the algorithm consists of an array of cells, each one for computing a digit of R, and so each containing two multipliers. The main difficulty is that $\underline{q_i}$ needs to be computed before any progress can be made on the partial product R. Scaling M to make its leading two digits 10_r makes the computation easier [25],[27], as does shifting B downwards to remove the dependency on B. However, $\underline{q_i}$ still has to be broadcast simultaneously to every digit position (Fig. 2) and redundancy has to be employed in R so that digit operations can be performed in parallel [5], [27]. These severe drawbacks make Montgomery's algorithm for modular multiplication appear more attractive.

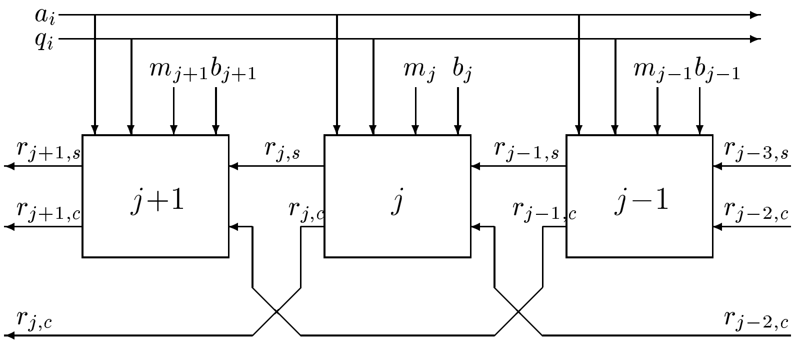

Figure 2. Classical Algorithm with Shift Up, Carries and Digit Broadcasting

5 Montgomery's Algorithm

Peter Montgomery [17] has shown how to reverse the above algorithm so that carry propagation is away from the crucial bits, redundancy is avoided and simultaneous broadcasting of digits is no longer required. This noticeably reduces silicon area and shortens the critical path. Although the complexities of the two algorithms seem to be identical in time and space, the constants for Montgomery's version are better in practice. Montgomery uses the *least* significant digit of an accumulating product R to determine a multiple of M to *add* rather than subtract. He chooses multiplier digits in the opposite order, from *least* to *most* significant and shifts *down* instead of up on each iteration:

Montgomery's Modular Multiplication Algorithm:

{ Pre-condition: $0 \le A < r^n$ }

```
R := 0 ;
For i := 0 to n-1 do
Begin
    R  := R + a_i×B ;
    q_i := (-r_0 m_0^-1) mod r ;
    R  := (R + q_i×M) div r ;
    { Invariant: 0 ≤ R < M+B }
End
{ Post-condition: Rr^n = A×B + Q×M
   and, consequently, R ≡ (A×B×r^-n) mod M }
```

Here ${m_0}^{-1}$ is a residue $\bmod r$ satisfying $m_0 \times {m_0}^{-1} \equiv 1 \bmod r$. Since r is a power of 2 and M is odd (because it is a product of two large primes) r and M are co-prime, which is enough to guarantee the existence of ${m_0}^{-1}$. The digit q_i is chosen so that the expression $R+q_i\times M$ is exactly divisible by r. Its lowest digit is clearly 0. If we define $A_i = \sum_{j=0}^{i} a_j r^j$ and Q_i analogously then $A_i = A_{i-1}+a_i r^i$ and

$A_n = A$. The value of R at the end of the iteration whose control variable has value i is easily shown by induction to satisfy $Rr^{i+1} = A_i{\times}B + Q_i{\times}M$ because the division is exact. Hence the post-condition holds. The digits of A are required in ascending order. Thus, they can be converted on-line into a non-redundant form and so we may assume $a_i \le r{-}1$. This enables the loop invariant bounds to be established by induction.

The extra power of r factor in the output R is easily cleared up by minor pre- and post-processing [6]. The easiest way to explain this is to associate with every number its *Montgomery class* mod M, namely

$$\overline{A} \equiv Ar^n \bmod M$$

and to use $\overline{\times}$ to denote the Montgomery modular multiplication. The *Montgomery product* of $\overline{A}$ and $\overline{B}$ is $\overline{A}\,\overline{\times}\,\overline{B} \equiv \overline{A}\,\overline{B}\,r^{-n} \equiv ABr^n \equiv \overline{AB} \bmod M$. So applying Montgomery multiplication to $\overline{A}$ in an exponentiation algorithm is going to produce $\overline{A^e}$ rather than $(\overline{A})^e$. Introduction of the initial power of r to obtain $\overline{A}$ is performed using the precomputed value

$$R_2 = \overline{r^n} \equiv r^{2n} \bmod M$$

and a Montgomery multiplication thus [7]: $A\overline{\times}R_2 \equiv Ar^n \equiv \overline{A} \bmod M$. Removal of the final extra power of r is also performed by a Montgomery multiplication: $\overline{A^e}\,\overline{\times}\,1 \equiv A^e \bmod M$.

Throughout the exponentiation, an output from one multiplication is used as an input to a subsequent multiplication. Without care the outputs will slowly increase in size. However, suppose $a_{n-1} = 0$. Then the bound $R < M{+}B$ at the end of the second last loop iteration is reduced to $M{+}r^{-1}B$ on the final round, which prevents unbounded growth when outputs are used as inputs. In particular, if the second argument satisfies $B < 2M$ then the output also satisfies $R < 2M$. Thus, suppose $2rM < r^n$. It is reasonable to assume that A and R_2 are less than $2M$, even less than M, so that their topmost digits are both zero. Then the scaling of A to $\overline{A}$ by Montgomery multiplication yields $\overline{A} < 2M$ and this bound is maintained as far as the final output A^e. So only a single extra subtraction of M may be necessary at the very end to obtain a least non-negative residues.

However, when all the I/O is bounded by $2M$, an interesting and useful observation about the output R of the final multiplication $\overline{A^e}\,\overline{\times}\,1 \equiv A^e \bmod M$ can be derived from the post-condition of the modular multiplication, namely $Rr^n = \overline{A^e} + QM$. Q has a maximum value of $r^n{-}1$. Hence, $\overline{A^e} < 2M$ would lead to the output satisfying $Rr^n < (r^n{+}1)M$ and so to $R \le M$, whilst a sub-maximal value for Q immediately yields $R < M$ in the same way. Hence a final subtraction is *only* necessary when $R = M$, i.e. when $\overline{A^e} \equiv 0 \bmod M$, that is, for $A \equiv 0 \bmod M$. It is entirely reasonable to assume that this never occurs in the use of the RSA cryptosystem as it would require starting with $A = M$, whereas invariably the initial value should be *less* than M. Moreover, each modular multiplication in the exponentiation would also have to return M rather than 0 to prevent all subsequent operands becoming 0. Hence the final subtraction need never occur after the final re-scaling of an exponentiation.

Computation of q_i is a potential bottleneck here. Simplification may be achieved in the same two ways that applied to the classical algorithm above: this time scale M to make $m_0 = -1$ and shift B up so that $b_0 = 0$. These would make $q_i = r_0$, avoiding any computation at all. We consider the shift first, supposing initially that $B < 2M$. If the shift $B \leftarrow rB$ is added to the start of the multiplication code, then the loop invariant becomes $R < M{+}rB$. Hence we require the top *two* digits of A to be zero in order for the bound on R to be reduced first to $M{+}B$ and then to $M{+}r^{-1}B$, so that $R < 2M$ is output. If we always have $A < 2M$ then this is achieved if $2r^2M$ fits into the hardware registers of n digits. The cost of this shift is hidden by the definition of n. The shift can be hardwired and counted as free, as can a balancing adjustment of the output through a higher power of r in R_2. However, there is an extra iteration of the loop: before we had one more iteration than digits in M, now we have two more. Fortunately, the cost of one more iteration per multiplication is low compared with the delay on every iteration which computing q_i may cause. Apart from the extra storage cost of the scaled number, scaling M has a similar cost, namely one more iteration per multiplication: M is replaced by $(r{-}{m_0}^{-1})M$, which increases its number of digits by 1 [26]. However, at the end of the exponentiation, the original M also needs to be loaded into the hardware and some extra subtractions of M may be necessary to reduce the output from a bound of $2rM$.

6 Digit-Parallel and Digit-Serial Implementations

To overcome the problems of carry propagation in the classical algorithm, redundancy and extra hardware for digit broadcasting were required. Here, too, the same methods enable parallel digit processing. Indeed, if the shift direction were reversed, the diagram of Fig. 2 would cover Montgomery's algorithm also.

For both algorithms, define the ith value of R, written $R^{(i)} = \sum_{j=0}^{n-1} r_j^{(i)} r^j$, to be the value immediately before the shift is performed. This is calculated at time $2i$ in the parallel digit implementations, with q_i computed and broadcast at time $2i{+}1$, say. The jth cell operates on the jth digits, transforming the $i{-}1$st value of R into the ith value. A common view of this process is:

$$r_j^{(i)} \leftarrow r_{j\pm1,s}^{(i-1)} + r_{j\pm1-1,c}^{(i-1)} + a_i b_j \pm q_i m_j$$

where the choice of signs is $-$ for the classical algorithm and $+$ for Montgomery's. (The input values of R on the right are partitioned into save and carry/borrow parts.)

A restriction to only nearest neighbour communication is desirable because of the delays and wiring associated with global movement of data. For Montgomery's algorithm, a systolic array makes this possible [26]. In this, the cells are transformed into a pipeline in which the jth cell computes $r_j^{(i)}$ at time $2i{+}j$ (Fig. 3). The input $r_{j+1}^{(i-1)}$ is calculated on the preceding cycle by cell $j{+}1$ and a carry $c_{j-1}^{(i)}$ from $r_{j-1}^{(i)}$ is computed in cell $j{-}1$, also in that cycle. This means that carries can be propagated and so the cell function can become:

$$r_j^{(i)} + rc_j^{(i)} \leftarrow r_{j+1}^{(i-1)} + c_{j-1}^{(i)} + a_i b_j + q_i m_j$$

where the digits $r_j^{(i)}$ are now in the standard, non-redundant range $0 \mathinner{..} r-1$. (The different notation for the carries recognises that they do not form part of the value of $R^{(i)}$, unlike in the carry-save view.) If the digit q_i is produced at time $2i$ in cell 0, it can be pipelined and received from cell $j-1$ at time $2i+j-1$ ready for the calculation. This pipeline can be extended to part or all of a 2-dimensional array with n rows which computes iterations of the loop in successive rows.

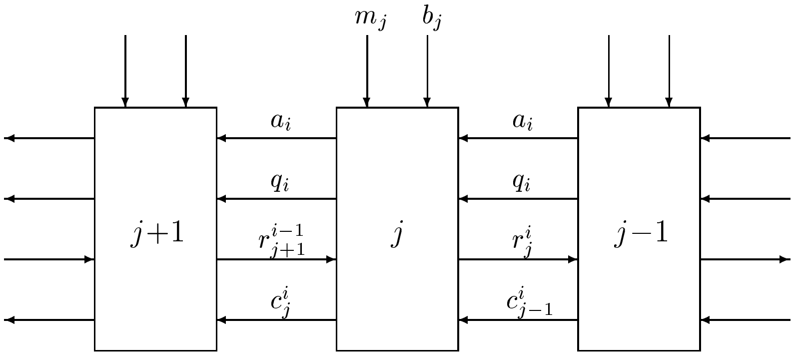

Figure 3. Pipelined Montgomery Multiplication: I/O for cell j at time $2i+j$.

The lowest cell, cell 0, computes only the quotient digit q_i. Digits of index 0 are always discarded by the shift down and so do not need computing; the lowest digit of the final output is shifted down from index 1. We have $q_i = -(r_1^{(i-1)} + a_i \times b_0) {m_0}^{-1} \bmod r$. This can indeed be calculated at time $2i$ because its only timed input $r_1^{(i-1)}$ is computed at time $2i-1$. Observe that pre-computation of $b_0 \times {m_0}^{-1}$ reduces the computation of q_i to a single digit multiplication and an addition, giving lower complexity for cell 0 than for the other cells. Hence, computing q_i no longer holds up the multiplication. Instead, the critical path lies in the repeated, standard cell.

The communication infrastructure is less here than for the parallel digit operations illustrated in Fig. 2. Although the number of bits transmitted is almost the same in both cases and is independent of n, the parallel digit set-up requires an additional $\mathrm{O}(\log n)$ depth network of multiplexers to distribute the digit q_i. Here the inputs and output are consumed, resp. generated, at a rate of one digit every other cycle for A and R, and one digit every cycle in the case of B. Unlike the parallel digit model, this is very convenient for external communication over the bus, reducing the need for buffering or increased bandwidth.

When one multiplication has completed, another one can start without any pause. However, the opposing directions of carry propagation and shift mean that each cell is idle on alternate cycles. Thus, full use of the hardware requires two modular multiplications to be processed in parallel. The normal square and multiply algorithm for exponentiation can be programmed to compute squares nose-to-tail starting loop iterations on the even cycles and interleave any necessary multiplications to start on the odd cycles. This enables an average 75% take-up of the processing power, but has some overhead in storage and switching between the two concurrent multiplications. Overall, with this added complexity, the classical, parallel digit, linear array might be faster for small n, but for larger n and/or smaller r the broadcasting problem for q_i means that a pipelined implementation of Montgomery's algorithm should be faster.

In [14] Peter Kornerup modified this arrangement, pairing or grouping digits in order to reduce the waiting gap. In effect he alters the angle of the timing front in the data dependency graph and, in the case he illustrates, he uses half the number of cells with twice the computing power. This can be advantageous in some circumstances.

An idea of the current speed of such array implementations is given in Blum and Paar [1] and amongst those actually constructed is one by Vuillemin *et al.* [22]

7 Data Integrity

Correct functioning is important not only for critical applications but as a protection against, for example, attacks on RSA signature schemes through single fault analysis [2]. Moreover, it is difficult and expensive to check all gate combinations for faults at fabrication time because of the time need to load sufficiently many different moduli [29] and, when smart cards are involved, a low unit price may only be possible by using tests which occasionally allow sub-standard products through to the market.

However, run-time checker functions are possible. These can operate in a similar way to those for multiplication in current chips. For example, results there are checked mod 3 and mod 5 in one case [8]. Here the cost of a similar check is minimal compared to that of the total hardware. The key observation is that the output from the modular multiplication algorithm satisfies an arithmetic equation:

$$R = A\times B - Q\times M \qquad \text{or} \qquad Rr^n = A\times B + Q\times M$$

These are easily checked $\bmod m$ for some suitable m by accumulating partial results for both sides on a digit-by-digit basis as the digits become available. A particularly good choice for m is a prime just above the maximum cell output value, but smaller m prime to r are also reasonable. The hardware complexity for this is then equivalent to about one cell in the linear array and so the cost is close to that of increasing n by 1.

If a discrepancy is found by the checker function, the computation can be aborted or re-computed by a different route. For example, to avoid the problem, M might be replaced by dM for a digit d prime to r and combined as necessary with some extra subtractions of the original M at the end.

8 Timing and Power Attacks

The literature contains descriptions of a number of attacks on the RSA cryptosystem which use timing or power information from hardware implementations which contain secret keys [12], [13]. Experience of implementing both the classical and Montgomery algorithms for modular multiplication suggests that most optimisation techniques which work with one also apply to the other. This suggests that attacks which succeed on well-designed implementations of the classical algorithm will have equivalents which apply to implementations of Montgomery's algorithm.

However, an important difference arises when the pipelined linear array is used since, judging from the data dependency graph, there seems to be no equivalent for the classical algorithm. With parallel digit processing of the multiplication $A{\times}B \bmod M$, the same digits of A and Q are used in every digit slice, opening up the possibility of extracting information about both by averaging the power consumption over all cells. However, during the related Montgomery multiplication in a pipelined array, many digits of A and Q are being used simultaneously for forming digit products. This should make identification of the individual digits much more difficult, and certainly increases the difficulty of any analysis.

9 Conclusion

We have reviewed and compared the main bottlenecks which may arise in hardware for implementing the RSA cryptosystem using both the classical algorithm for modular multiplication and Montgomery's version, and shown how these are solved. The hardware still suffers from broadcasting problems with the classical algorithm and scheduling complications with Montgomery's. However, as far as implementation attacks using power analysis are concerned, the pipelined array for the latter seems to offer considerable advantages over any other implementations.

References

1. T. Blum & C. Paar, "Montgomery Modular Exponentiation on Reconfigurable Hardware", *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press (1999) 70-77
2. D. Boneh, R. DeMillo & R. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", *Eurocrypt '97*, Lecture Notes in Computer Science, vol. 1233, Springer-Verlag (1997) 37-51
3. R. P. Brent & H. T. Kung, "The Area-Time Complexity of Binary Multiplication", *J. ACM* **28** (1981) 521-534
4. R. P. Brent & H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Trans. Comp.* **C-31** no. 3 (March 1982) 260-264

5. E. F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography", *Advances in Cryptology - CRYPTO '82*, Chaum et al. (eds.), New York, Plenum (1983) 51-60
6. S. E. Eldridge, "A Faster Modular Multiplication Algorithm", *Intern. J. Computer Math.* **40** (1991) 63-68
7. S. E. Eldridge & C. D. Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm", *IEEE Trans. Comp.* **42** (1993) 693-699
8. G. Gerwig & M. Kroener, "Floating Point Unit in Standard Cell Design with 116 bit Wide Dataflow", *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press (1999) 266-273
9. D. E. Knuth, *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms*, 2nd Edition, Addison-Wesley (1981) 441-466
10. N. Koblitz, *A Course in Number Theory and Cryptography*, Graduate Texts in Mathematics **114**, Springer-Verlag (1987)
11. . K. Ko, T. Acar & B. S. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms", *IEEE Micro* **16** no. 3 (June 1996) 26-33
12. P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", *Advances in Cryptology, Proc Crypto 96*, Lecture Notes in Computer Science **1109**, N. Koblitz editor, Springer-Verlag (1996) 104-113
13. P. Kocher, J. Jaffe & B. Jun, *Introduction to Differential Power Analysis and Related Attacks* at `www.cryptography.com/dpa`
14. P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms", *IEEE Trans. Comp.* **43** no. 8 (1994) 892-898
15. W. K. Luk & J. E. Vuillemin, "Recursive Implementation of Optimal Time VLSI Integer Multipliers", *VLSI '83*, F. Anceau & E.J. Aas (eds.), Elsevier Science (1983) 155-168
16. K. Mehlhorn & F. P. Preparata, "Area-Time Optimal VLSI Integer Multiplier with Minimum Computation Time", *Information & Control* **58** (1983) 137-156
17. P. L. Montgomery, "Modular Multiplication without Trial Division", *Math. Computation* **44** (1985) 519-521
18. S. F. Obermann, H. Al-Twaijry & M. J. Flynn, "The SNAP Project: Design of Floating Point Arithmetic Units", *Proc. 13th IEEE Symp. on Computer Arith.*, Asilomar, CA, USA, 6-9 July 1997, IEEE Press (1997) 156-165
19. F. P. Preparata & J. Vuillemin, "Area-Time Optimal VLSI Networks for computing Integer Multiplication and Discrete Fourier Transform", *Proc. ICALP*, Haifa, Israel, 1981, 29-40
20. R. L. Rivest, A. Shamir & L. Adleman, "A Method for obtaining Digital Signatures and Public-Key Cryptosystems", *Comm. ACM* **21** (1978) 120-126
21. A. van Someren & C. Attack, *The ARM RISC Chip: a programmer's guide*, Addison-Wesley (1993)
22. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati & P. Boucard, "Programmable active memories: Reconfigurable systems come of age", *IEEE Trans. on VLSI Systems* **5** no. 2 (June 1997) 211-217
23. C. S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Trans. Electronic Computers* **EC-13** no. 2 (Feb. 1964) 14-17
24. C. D. Walter, "Fast Modular Multiplication using 2-Power Radix", *Intern. J. Computer Maths.* **39** (1991) 21-28
25. C. D. Walter, "Faster Modular Multiplication by Operand Scaling", *Advances in Cryptology - CRYPTO '91*, J. Feigenbaum (ed.), Lecture Notes in Computer Science **576**, Springer-Verlag (1992) 313-323

26. C. D. Walter, "Systolic Modular Multiplication", *IEEE Trans. Comp.* **42** (1993) 376-378
27. C. D. Walter, "Space/Time Trade-offs for Higher Radix Modular Multiplication using Repeated Addition", *IEEE Trans. Comp.* **46** (1997) 139-141
28. C. D. Walter, "Exponentiation using Division Chains", *IEEE Trans. Comp.* **47** no. 7 (July 1998) 757-765
29. C. D. Walter, "Moduli for Testing Implementations of the RSA Cryptosystem", *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press (1999) 78-85