An Improved Linear Systolic Array for Fast Modular Exponentiation

Colin D. Walter

Computation Department, UMIST,
PO Box 88, Sackville Street, Manchester M60 1QD, U.K.
e-mail: C.Walter@umist.ac.uk
www.co.umist.ac.uk

Abstract. Because of the large word lengths involved, communication and buffering are potentially the major problems in implementing the modular arithmetic used in several cryptosystems. It is shown here how a single, linear systolic array eliminates much of the associated overheads, thereby improving through-put and the ratio of speed to area for modular exponentiation. Alternative forms produce simpler processing elements and make fuller use of the hardware, making it more easily implemented in current technology. Such designs are regarded as much safer for use in smartcards and embedded systems as they offer greater protection against attacks using differential power analysis. A 1024-bit array can be built in an area comparable to a 64-bit multiplier.

Index Terms: Computer arithmetic, cryptography, RSA cryptosystem, Diffie-Hellman, linear systolic array, Montgomery modular multiplication, exponentiation, differential power analysis, DPA.

1 Introduction

Efficient modular exponentiation is essential to fast encryption using, for example, the RSA [19], ElGamal [5], Diffie-Hellman [3] or DSA [17] schemes. Brickell [2] describes the earliest hardware speedups for modular multiplication. He employs digit slices working in parallel and redundant number representations in order to perform modular multiplication by repeated addition. However, the simultaneous broadcast of multiplier digits to all digit positions is required, with a consequent burden on hardware area, capacitance and clock period. To avoid this, digits in the adder must be processed serially using a systolic array. In [22], the author showed how this could be done by using Montgomery's modular multiplication algorithm [16] which reverses the order of processing multiplier digits when compared with the standard multiplication algorithm. This was the first design where, apart from clock and power, all the communication was entirely local, between nearest neighbour processing elements. Without the need to await the completion of carry propagation, the clock cycle is much shorter, so that throughput is increased and hardware cost is reduced. Indeed, there is improved *latency* over the most recent implementations of traditional methods [24].

For convenience, [22] initially describes a full, rectangular, systolic array with one row for each addition step and as many rows as are necessary to complete a modular multiplication. The final paragraphs describe the possibilities of fewer rows and, in particular, mention its minimal linear form as a single row which can perform two multiplications in parallel by feeding the output directly back in. This indeed provides a modular equivalent of the pipeline multiplier of Jackson *et al.* [7]. Here some quantification and amplification of the benefits of this special case are given.

In particular, we first show how to adapt the array to modular exponentiation without having to buffer results or idle between successive multiplications. The square and multiply algorithm for exponentiation leads to only 75% use of the resulting hardware on average. Moreover, the silicon area is still substantial since each processing element (PE) contains two multipliers. The second part of this article therefore reduces the PEs to contain a single multiplier and makes them work on every cycle instead of only on alternate cycles to compute one modular multiplication. This cuts the total area almost in half, making it more amenable to current technology and allows 100% use of the array rather than 75%.

Alternative linear arrays have been given by Kornerup [14] and Jeong and Burleson [8]. By comparison, Jeong and Burleson [8], Fig. 5, require 50% more clock cycles per multiplication than here and perform only one multiplication at once rather than two as here. Their array uses the same number of PEs and they are also more complex than here. Hence throughput for an exponentiation is at least 3 times slower and the hardware has the additional overhead of several extra registers (7 for the binary case). The PEs there only operate on every third cycle. The most competitive hardware is that of Kornerup, who groups pairs of product terms [14], Fig. 2, in order to utilise the array fully. Consequently, he has half the number of cells as here but they are twice as complex, yielding the same overall hardware area. By running the two halves of his PEs in parallel, his cell speed is only marginally slower than that here but he uses half the number of clock cycles. Thus, his array has superior latency by a factor of almost 2, but similar throughput since two modular multiplications cannot be computed in parallel. Two copies of his array are needed for efficient exponentiation. This provides almost twice the speed as here but at a cost of just over twice the hardware and slightly more buffering is required. The $area{\times}time$ product is thus similar. With pressure on chip area, the more minimal design here is of importance. Our second design uses only a quarter of the area of Kornerup and reduces $area{\times}time$ for exponentiation by 25% on average.

The exposition closes with a look at the more general context of the hardware, particularly considering its potential strength against differential power analysis [12], [13].

2 The Systolic Array for Modular Multiplication

Suppose $(A{\times}B)$ mod M is to be calculated, where the radix r of the representations is prime to the modulus M and the non-negative integers A, B are bounded above by, say, $2M$. Assume A, B and M all have at most n digits in base r, with, for example, $A = \sum_{i=0}^{n-1} a_i r^i$ where $0 \le a_i < r$. In practice, for the systolic arrays here, r will probably be a power of 2 between 2^2 and 2^{64}, giving digits of between 2 and 64 bits. Historically ([2],[4],[23]), redundant representations have been used to limit carry propagation and enable parallel digit operations. This is unnecessary here because numbers are processed digit serially, least significant digit first. This allows in-line conversion back to a standard, non-redundant representation if necessary.

Montgomery's modular multiplication algorithm [16] is essential here. It computes $R = (A{\times}B)r^{-n}$ mod M where n is the number of addition cycles in the process. The extra power of r is ignored at this point, but will be treated when exponentiation is reached. It arises because the ith iteration of the algorithm computes the partial modular product

$$R_{i+1} = r^{-1}(R_i + a_iB + q_iM)$$

where $R_0 = 0$ and q_i is a digit chosen to make $R_i + a_iB + q_iM$ exactly divisible by r. The $j{-}1$st digit of R_{i+1} is obtained using the recurrence relation

$$r_{i+1,j-1} + r{\times}c_{i,j+1} \quad = \quad r_{i,j} + a_ib_j + q_im_j + c_{i,j} \qquad (*1)$$

where $c_{i,j+1} \in [0..2r{-}2]$ is the carry for propagation up the adder, initialised with $c_{i,0} = 0$. A model (i,j)th *processing element* (PE) for calculating this in the rectangular array [22] is illustrated in Figure 1. The ith row computes R_{i+1} from R_i. For simplicity, suppose each PE operates in a single clock cycle. Then the (i,j)th PE processes the digits of (*1) at clock cycle time $2i{+}j$, buffering or storing them as indicated. PE $(i,0)$ is slightly different as it determines q_i as well, and uses 0 for its input $c_{i,0}$.

To ensure the calculation of (*1) is possible, the inputs must be generated in time. This can be checked using a data dependency graph, but we need exact times for later. According to the formula, $r_{i,j}$ is generated by cell $(i{-}1,j{+}1)$ at time $2i{+}j{-}1$, which is just in time for its use by cell (i,j). Also, q_i must be generated by cell $(i,0)$ at time $2i$ ready for use by cell $(i,1)$ at time $2i{+}1$. Its definition is $q_i \equiv -m_0^{-1}(r_{i0} + a_ib_0)$ mod r where r_{i0} is generated by cell $(i{-}1,1)$ at time $2i{-}1$, also just in time. If cell $(i,0)$ has difficulty in performing its calculation within the same time as the other PEs then various simplifications are possible (*see* [22]): in particular, M might be scaled so that $-m_0^{-1} \equiv 1$ mod r and/or B might be shifted up so that $b_0 = 0$. Section 6 presents more detail for the latter.

For inputs of up to n digits, the topmost ordinary PE has index $j = n{-}1$. For greater j, the inputs b_j and m_j are 0 so that no multipliers are necessary in the

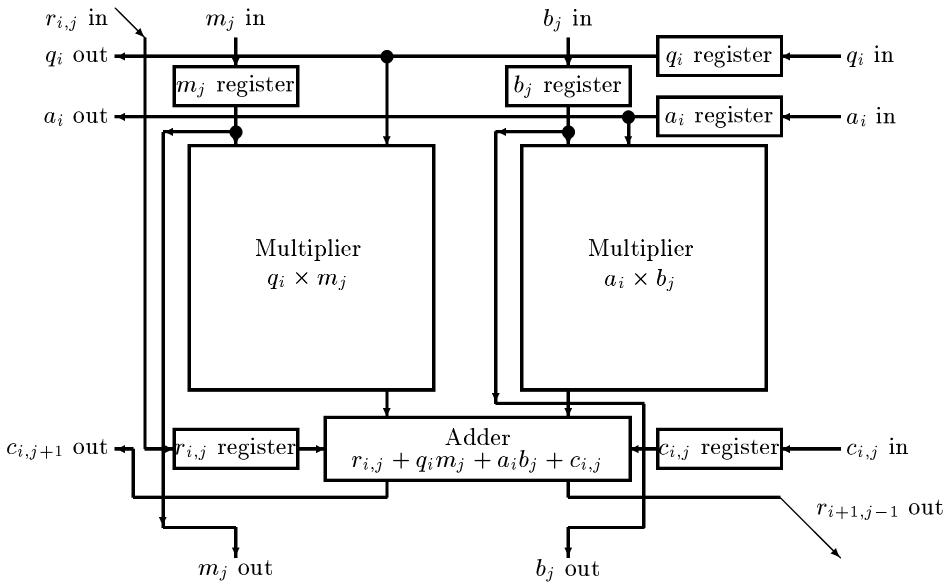

Fig. 1. The $(i, j)^{th}$ Processing Element for Array Modular Multiplier

PE. Since, by assumption, $A < 2M$ and $B < 2M$, we can prove inductively that $R_i < 3M$ from $R_{i+1} = r^{-1}(R_i + a_iB + q_iM) < r^{-1}(3M + 2(r{-}1)M + (r{-}1)M)$ $= 3M$. So the (i, n)th PE will generate digit $r_{i+1,n-1}$ with a carry $c_{i,n+1}$ of at most 2. Hence, (assuming a radix of at least 4) we just need a very simple top $(i, n{+}1)$th PE which calculates and stores $r_{i+1,n} = c_{i,n+1}$ (the other inputs to (*1) are zero) and feeds it back and down to the $(i{+}1, n)$th PE at the right time. At the row with $i = n$ we have $a_i = 0$ so that $R_{n+1} = r^{-1}(R_n + q_iM)$ $< r^{-1}(3M + (r{-}1)M) \le 2M$. Thus row n generates output which satisfies the input condition of being less than $2M$. This is then suitable for re-input during exponentiation. Note that extra PEs at the top end won't affect the calculations as long as the appropriate most significant digits are initialised to 0.

To obtain a linear, pipeline modular multiplier, only one row of PEs is taken (the ith, for example). For this, the jth PE behaves like cell (i, j) of the rectangular array, computing (*1) at times $2i{+}j$ for $i = 0, 1, ..., n$. Data which was previously passed between rows must be re-directed back into the single line of PEs. So the output $r_{i+1,j-1}$ from the jth PE is fed into the $j{-}1$st PE where it arrives at exactly the right time for executing (*1). Of course, some control must be added to initialise and clock the registers correctly and to catch the final output $r_{n+1,j-1}$ at the right time. This is shown in Figure 2.

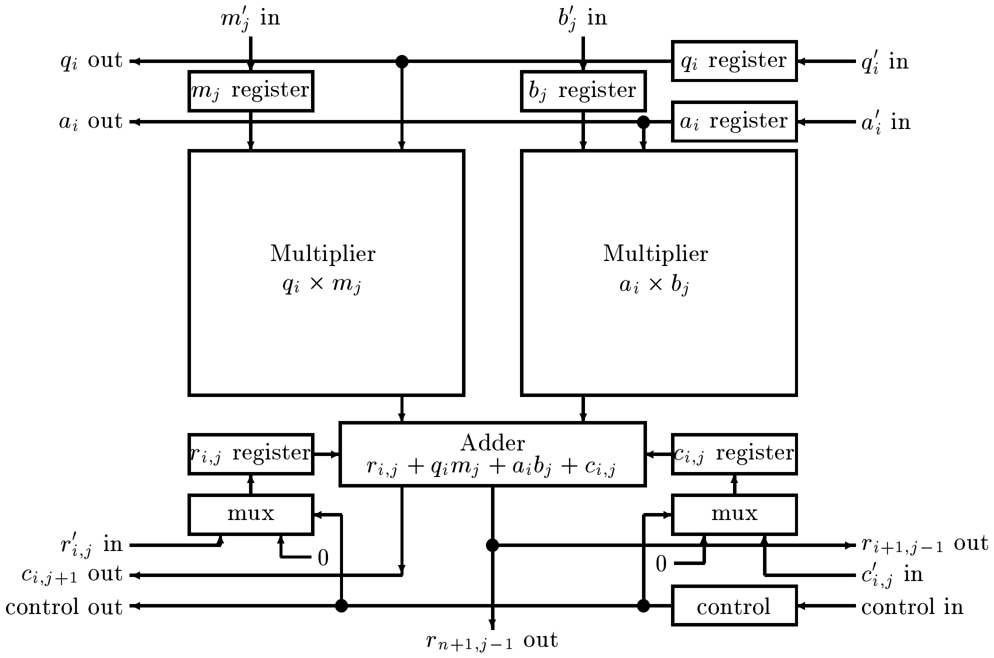

Fig. 2. I/O at Time $2i+j$ for the j^{th} PE of the Pipeline Multiplier

Each PE is only used on alternate clock cycles, the jth one operating on cycles j, $j+2$, $j+4$, etc. So another modular multiplication can be interleaved in the gaps, say $(A' \times B')r^{-n'}$ mod M', running one cycle, or any odd number of cycles, behind the first multiplication. Some communication wires and registers may need to be duplicated for this and appropriate control added, but the multipliers and adder are reused. This is also shown in Figure 2, which illustrates the arrival of updating input for the other, interleaved multiplication.

The linear array may be modified to have all I/O through cell 0. Such a design would allow the circuit to be incorporated easily into awkward spaces on a chip. Thus, in Figure 3, the interleaved output digits $r_{n+1,j}$ and $r'_{n+1,j}$ are piped back to cell 0. Digit $r_{n+1,j}$ is created at time $2n+j+1$ and so, progressing one PE at a time, reaches cell 0 at time $2n+2j+1$, showing that it does not overwrite the other output digits on the same pipeline. A more complex arrangement may be needed for routing the digits of M and B from cell 0. The modulus digit m_j is needed for first use in the jth PE at time j. By sending digits m_j out from cell 0 at the rate of two digits per cycle and at the speed of two cells per cycle, m_j can leave at time j div 2 and still arrive just in time. This explains why two digits of M have to pass through each PE and only one is clocked through a register. However, if the PE is slower than this pipeline, then a simpler arrangement is possible, with all the digits of M in a single pipeline advancing by one PE every half of a cell cycle. A control bit piped along the array to reach cell j at time $j-1$ enables the correct modulus digit to be captured and loaded into the PE register and for any other cell initialisation, such as the resetting $r_{0j} = 0$, to be signalled. The digits of B are treated similarly, as are the digits of M' and B', which are not shown in the figure.

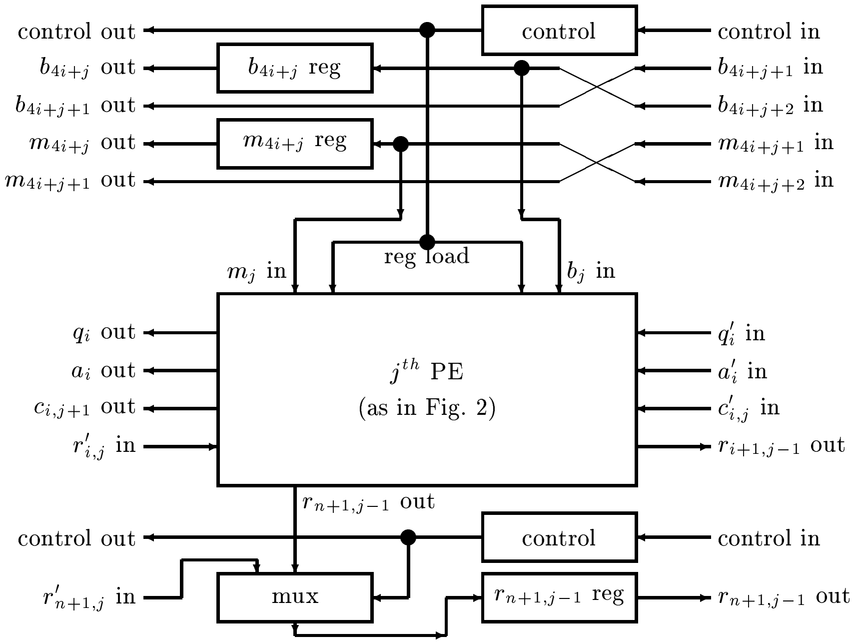

Fig. 3. Communication for I/O directed through cell 0

3 Exponentiation

The cryptographic applications of this array require exponentiating plaintext or ciphertext T to some power $\Delta = \sum_{i=0}^{d-1} \delta_i 2^i$ of d bits, say. This can be done by computing successive squares $S_i = T^{2^i}$ mod M and cumulative products $P_i = \prod_{j=0}^{i-1} T^{\delta_j 2^j}$ mod M using the parallel modular multiplications:

$$\begin{array}{rcl} S_{i+1} & \leftarrow & S_i \times S_i \bmod M \\ P_{i+1} & \leftarrow & P_i \times S'_i \bmod M \\ \textit{where} & S'_i = 1 & \textit{for}\ \ \delta_i = 0 \\ \textit{and} & S'_i = S_i & \textit{for}\ \ \delta_i = 1 \end{array}$$

If Montgomery's algorithm is used then each multiplication picks up an extra factor r^{-n} where n is the number of addition cycles per multiplication. This is compensated for by pre-Montgomery-multiplying T by r^{2n} mod M so that Tr^n mod M is fed into the exponentiator. Each multiplication or squaring maintains the factor r^n so that $T^{\Delta}r^n$ mod M is output. The only necessary post-processing is then another Montgomery multiplication, this time by 1, which removes the unwanted r^n.

Both the square and multiply operations can be calculated in parallel by the array and successive pairs pipelined with virtually no intervening breaks or buffering. Thus the array should be very well suited to exponentiation, as each cell can then be almost fully utilised, performing its task on almost every cycle. The only reservation for the scheme here is that the multiplications by 1 are included here but generally omitted. (Multiplication by 1 here would actually be done via Montgomery multiplication by r^n.) Consequently, on average the array is utilised at only 75% of its full capacity. However, the exponentiation will still normally have to perform d squarings and therefore must take at least the time used here (*but see* [6]).

A PE of the exponentiating array is illustrated in Figure 4. In terms of the previous notation, the interleaved multiplication sequences have

$$A = B = S_i \qquad R = S_{i+1} \qquad A' = S'_i \ (= r^n \text{ or } S_i) \qquad B' = P_i \qquad R' = P_{i+1}$$

where the choice for A' is determined by the exponent bit δ_i. The array is initialised for the first two simultaneous multiplications by supplying $M' = M$ and $A = B = Tr^n$ mod M $(= S_0)$, setting $B' = r^n$ $(= P_0)$ and computing A' from A, as defined. Most of this and the subsequent operation is just as in Figure 2. The main difference is that the digits of B and B' are also needed for immediate consumption by the neighbouring cell.

For the interface between successive multiplications, suppose the products $R = S_i$ and $R' = P_i$ have just been computed and are about to be used as the inputs B and B' for the next pair of multiplications. Since the ith multiplication is offset by $2in$ clock cycles, the jth output digits $r_{n,j}$ and $r'_{n,j}$ are generated by cell $j{+}1$ at times $2in{+}j{-}1$ and $2in{+}j$ respectively. These digits become inputs b_j and b'_j for the following multiplications. They must be ready for use by cell j at times $2in{+}j$ and $2in{+}j{+}1$ respectively. This is just possible with no gap between successive multiplication cycles. So there is no need for buffering the data further nor is there any idle time in the hardware.

The other inputs A and A' must also be checked for timeliness. A is initialised digit serially from cell 0. Thereafter, the output $R = S_i$ and S'_i provide the inputs A and A'. The digit $r_{n,j}$ is computed at time $2in{+}j{-}1$ and reaches cell 0 at exactly the right time $2in{+}2j$ for its first use there as a_j. This leaves one clock cycle for cell 0 to compute a'_j from a_j using $A' = \delta_i A + \overline{\delta_i} r^n$. The final output of the array, T^{Δ} mod M, is a value of R', and so can be collected serially from cell 0 after a multiplication by 1 to remove the extra power of r.

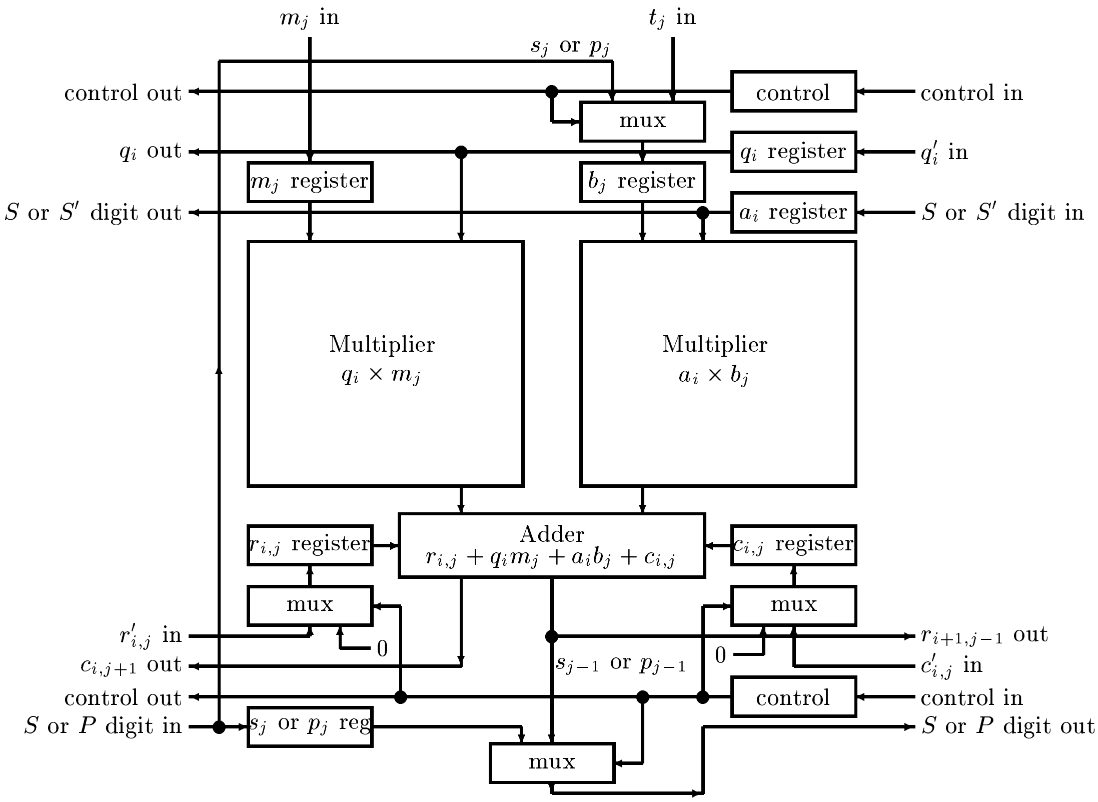

Fig. 4. j^{th} PE for Exponentiation

As in Figure 2, the main control bit is set by cell 0 on every $2n$th cycle and reaches the jth PE at time $2in+j-1$ ($i = 0, 1, 2, ...$). It indicates that the cell should update its current b_j to the incoming $r_{n,j}$, and make a similar change to b'_j on the next clock tick. A similar control bit indicates when the outputs $r_{n,j}$ and $r'_{n,j}$ should be captured for routing to cell 0. If cell 0 is now slower than the normal PE, extra cycles can be inserted between successive multiplications to allow it to complete without slowing down the clock. Specifically, each multiplication is just allowed to run on as necessary before starting the next one (perhaps with some updating for the next multiplication) but the result is still extracted at the same point. This is a minimal extra cost as n is relatively large.

4 External Connections and Comparative Study

Hardware for modular multiplication will probably either reside on a dedicated chip or be part of a larger digital signal processing chip. In both cases the I/O bandwidth is likely to be limited. Hence the external communication problems will be similar. With little if any loss of generality, we therefore phrase the discussion in terms of a dedicated chip. It also suffices to consider only exponentiations since data rates for multiplication sequences will be about twice those for exponentiations to the power 1 (because the exponentiation may perform a useless final squaring).

The connections of the array to the external world are fairly simple: the inputs are M, T, Δ and perhaps r^{2n} mod M, and the output is $R' = T^{\Delta}$ mod M after post-processing. Each of these is required, respectively generated, digit serially. Corresponding input and output digits are separated by time $2dnt$. At worst, three digits are input and one digit output from the array over the operation time t of a cell. In practice, chips have relatively few pins for I/O and communication with the rest of the world may be at a slower clock speed. Hence external communication of large integers is in effect limited to the sequential I/O of a few bits over a number of clock cycles. Providing the radix r is not too large, the internal clock not too fast, and also the number of pins not too small, the chip I/O will be fast enough not to hold up the exponentiation and also fast enough to avoid the need for more than minimal buffering of the I/O. Current internal chip and communication technologies are sufficiently in line with each other that these conditions are easily satisfied.

A desirable choice for r might involve digits with the same range as normal machine arithmetic, say 32 or even 64 bits of input, so that PEs can use a standard off-the-shelf multiplier design containing years of optimisation (e.g. [18]) and the I/O requirements might match internal bus speeds. However, a large RSA word length of perhaps 2^{10} bits leads to huge usage of chip area from, for example, 64 32×32-bit multipliers. Then, assuming a multiplication takes 4 cycles, the throughput for a full length exponent is 2^{10} bits (=text length) every 2^{18} clock cycles ($= 2dnt = 2{\times}2^{10}{\times}2^5{\times}2^2$ cycles). On a 400MHz clock, this would lead to a data decryption rate of about 1.5Mbits/sec. After scaling for technology differences, this is essentially the same as the programmable active memory (FPGA) technology of Shand *et al.* [21] which ran at 40MHz and gave a throughput of 165Kbits/sec for 2^{10}-bit inputs and keys. Although they used a pipelined version of Montgomery's algorithm, they also made use of a fourfold speedup from using the factorisation of the modulus and the Chinese Remainder Theorem to divide the operation into two parts running in parallel. This is balanced by using about four times the silicon area here (about 400K gates as opposed to 100K for their equivalent ASIC) to obtain similar throughput per clock cycle. Of course, such large multipliers, which take several cycles to operate, can be pipelined. With minor overhead, we could build each PE with a single, pipelined multiplier. Then both the multiplications could be computed in 5 cycles rather than 4, thereby reducing the $Area{\times}Time$ cost by almost half.

A 56×64 bit integer multiplier uses about 3% of a large chip in today's technology [20]. An array with $r = 2^{56}$ would require about $n = 18$ PEs and hence 36 such multipliers (18 if pipelined) plus a little more area for the communications. This puts it on the limit of current technology. Again, assuming it is a 4-stage multiplier, we obtain only an average exponentiation throughput of 1 bit per 2^7 cycles. If the chip also has a frequency multiplier of 4, then RSA decryption is still a mere 1 bit per 2^5 internal bus cycles – much below the bus capacity. However, the I/O comes in bursts, $56{\times}2^5$ bits plaintext input per bus cycle for these parameters, plus the same again for the modulus if that is also needed.

At the opposite extreme, the smallest choice of $r = 2$ probably suffers from too high an overhead in terms of storage area and time spent clocking latches. As the total area required is only a very small constant multiple of the area required to hold the input data, the choice is close to the minimal possible area needed to perform exponentiation. Slightly larger r, say 4 or 8, are of more interest. The former would require a PE with two 2-bit multipliers and containing about 2^6 gates. There would be 2^9 PEs for 2^{10}-bit inputs, yielding an array with similar area to only a couple of 64-bit multipliers (about 2^{12} full adders each). The array would perform decryptions in $2dnt = 2^{20}$ of its cycles, whereas the two 64-bit multipliers could complete a modular multiplication in 2^8 of their cycles and so a decryption in an average of $1.5{\times}2^{18}$ of its much longer cycles. The computing powers are therefore roughly similar for comparable areas. Thus the array here appears to be at least as cost effective. However, with only local connections and regular nature, layout is much more straightforward and it can be adjusted more easily to fit unusually shaped areas on the chip. Its parameter r can be chosen to make it utilise fully all available free space and, as we shall see, it is more resistant to differential power analysis attacks.

Throughput for the array here is essentially the same as for the parallel digit implementation of modular multiplication described in [24] if it uses the same basis r and identical circuitry for computing digit sums and products. The main advantages of the exponentiation process here are that a considerable digit distribution area is saved, and each modular multiplication avoids the small number of extra iterations which are required there to enable the timely arrival of the distributed digits. Thus, this array should have slightly better throughput and should have noticeably smaller area. Moreover, in a design like that of [24], all the input and output digits are consumed, respectively generated, simultaneously. Since the number of I/O bits in an RSA cryptosystem exceeds the number of pins available by an order of magnitude, the I/O data must be supplied serially and buffered on the chip. This decreases the performance of [24] as far as latency is concerned, at best to what the array here produces, and makes the area discrepancy even greater. In conclusion, the design here seems to out-perform the best modular exponentiation circuits which employ parallel digit processing and multiplication via repeated addition.

5 Elliptic Crypto-Systems

The standard integer RSA algorithm has a corresponding version for elliptic curves [10], [15] over a finite field $\mathbf{F}$. The above hardware design is particularly well suited to the case when this field has odd prime order p: elements are represented by integers modulo p rather than modulo M, but otherwise the arithmetic is identical.

Integer RSA uses the multiplicative group of residues prime to the modulus M, whilst elliptic RSA uses a group of points on the elliptic curve which is generally written additively. Encryption and decryption are performed by multiplying the message point (x, y) on the chosen elliptic curve $y^2 = x^3{+}ax{+}b$ by a key d. This is the analogue of exponentiation to the power d and is achieved by the equivalent of the square and multiply algorithm in Section 3. Doubling (x, y), which corresponds to squaring, gives the point (x', y') defined, in the case of characteristic greater than 3, by (*see* [11]) :

$$x' = \left(\frac{3x^2 + a}{2y}\right)^2 - 2x \qquad y' = \left(\frac{3x^2 + a}{2y}\right)(x{-}x') - y$$

and the sum $(x', y') = (x_1, y_1) + (x_2, y_2)$ is given by

$$x' = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \qquad y' = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1{-}x') - y_1$$

By far the most expensive operations here are inverting y and $x_2{-}x_1$. Agnew, Mullin & Vanstone [1] recommend swapping the affine co-ordinates (x, y) for projective ones (x, y, z) so that no inversion is required until encryption is completed and affine co-ordinates must be recovered. For odd prime fields, elliptic point doubling using projective co-ordinates requires 10 field multiplications and elliptic point addition 16 field multiplications (*see* [25], A.10.4 and A.10.5). The miscellaneous other operations (field additions, subtractions and scalar multiplications) are trivial to perform in one cycle each.

The extra arithmetic in doubling and adding points means use of the array here is more complex: there are a few more registers and so extra control. However, the schedule for doubling and adding elliptic points is simpler than that for squaring and multiplying in the integer case. Two additions from one or other can always be done simultaneously. So doubling and adding of points can be arranged sequentially instead of interleaved, yet still using the array fully. Consequently, additions need only done when necessary. The final inversion to recover affine co-ordinates can be done by exponentiation, using the array in the same way as before, without change and with the same efficiency. Overall, with inversion using only a fraction of the total time, the array is almost fully utilised.

6 Re-Structuring for 100% Use

For larger radices, the hardware cost is considerable. We suggest two ways in which the number of multipliers might be halved. First, the equation (*1) may be split into two almost identical sequential multiply-accumulate operations with one multiplication each:

$$\begin{aligned} t_{i,j} \quad + r\times d_{i,j+1} &= r_{i,j} + a_i\times b_j \; + c_{i,j} && (*2a) \\ r_{i+1,j-1} + r\times c_{i,j+1} &= t_{i,j} + q_i\times m_j + r\times d_{i,j+1} && (*2b) \end{aligned}$$

where $t_{i,j}$ is an intermediate digit value with carry $d_{i,j+1}$. A PE with two multipliers would perform the multiplications of (*1) in parallel. The cheaper solution of one multiplier used twice to perform (*2) will take twice the time, but, with just half the area, the product $area\times time$ is unchanged. In that context a larger cell 0 with two multipliers working in parallel could dispatch the required digits in time without any problems. If the single multiplier is large enough, it might be pipelined to perform the two multiplications in less than double the time, thereby reducing the $area\times time$ product.

Secondly, exponentiation is slightly unsatisfactory for a couple of reasons. Only the standard square and multiply method is well suited to the array here, and, on average, it still leaves the PEs unused for 25% of the time. To solve this, the array can be restructured to make each PE work on the same multiplication for every cycle. The original PE functionality is split in two for using a single multiplier. Equation (*1) is re-written as

$$\begin{aligned} t_{i,j} &= a_i\times b_j && (*3a) \\ r_{i+1,j-1} + r\times c_{i,j+1} &= r_{i,j} + t_{i,j} + q_i\times m_j + c_{i,j} && (*3b) \end{aligned}$$

by introducing intermediate values $t_{i,j}$. Both (*3a) and (*3b) are executed by the jth PE, the first on cycle $2i+j-1$ and the second on cycle $2i+j$. Outputs $r_{i,j}$ and $c_{i,j}$ are produced and used on the same cycles as before. An extra control bit selects between the two functions of the PE causing a negligible increase in execution time. Once r increases above 4, the area gained from halving the number of multipliers quickly outweighs the small area for extra control. This soon gives a hardware saving of over a third. However, exponentiation takes only 50% longer than before because the multiplications and squares are run sequentially, but no useless multiplications by 1 are done. So, for larger r, this may be a cost effective solution as both $area$ and $area\times time$ are improved.

Digits a_i and b_j need to be available one clock cycle earlier than before. Multiplications and squarings alternate so that P_i is only required as an input on every other multiplication. Its value must be buffered with an extra register in each PE. It will certainly be available in time for use as the B input. S_i is used in every multiplication as the A input and for squarings also as the B input. Its digits $r_{n,j}$ are calculated at time $2in+j-1$ and reach cell 1 for use as a_j at time $2in+2j$. It doesn't reach cell 0 in time, but this is not important if $b_0 = 0$. The condition $b_0 = 0$ was needed before to simplify cell 0 to help it operate in time. The same solution is employed here, namely to shift B up. B is input with a hardwired shift up to make $b_0 = 0$. The cost is cheap: at most one extra iteration per multiplication, which is required to ensure the output is under $2M$, and a different power of r being introduced into Montgomery products. As part of the shifted input $B = rR_n$, $r_{n,j}$ is calculated at time $2in+j-1$ and now first required as b_{j+1} in a_0b_{j+1} at time $2in+j$. As before, the digit $q_i \equiv -m_0^{-1}r_{i,0} \bmod r$ is computable by cell 0 at time $2in$, the cycle after $r_{i,0}$ is generated.

Other exponentiation schemes now become possible. In particular, the m-ary method [9] requires a pre-computed list of $m-1$ products. These can be stored in cyclic shift registers at cell 0 and loaded through that cell in the same way as S_i or P_i under the square and multiply scheme.

7 Differential Power Analysis

Differential power analysis (DPA) [13] is a method for attacking cryptographic hardware through variations in current arising from data dependent calculations. Typically an attack might be made against a smartcard or embedded cryptographic device containing a private key. The presence of a high noise-to-signal ratio in the current means that measurements are best made by averaging over a large number of cases. When the square and multiply method is used, the attacker would hope to average over a large enough set of exponentiations to be able to separate squares from multiplies and thereby read off the secret exponent. Similarly, where the same argument A, perhaps related to a chosen plaintext, is used repeatedly in calculating products $A\times B$ it might be possible to average over the many values of B to identify some property of A.

In standard implementations which use a single, large multiplier the digits of an operand A are very clearly separated out into the discrete time intervals when they are used. Also, with such hardware, squarings and multiplications have clearly defined beginnings and endings. With the systolic array described here, all multiplications merge into each other. First, successive multiplications overlap by about $2n$ cycles. Also, at any instant, all the PEs which are computing a digit product of one multiplication $A\times B$ are using different digits of A, not the same one. Hence the power variation is very much in line with the average digit of A rather than a succession of ups and downs representing the sequential use of distinctive individual digits of A. This makes the recognition of a particular operand A much more difficult in DPA, and the distinction between squares and multiplies irrelevant.

8 Conclusion

We have constructed three array modular multipliers and exponentiators parametrised by the radix r of the input representation, namely those implementing equation (*1) directly, and those implementing (*1) via equations (*2) or (*3). Each is well suited to the modular arithmetic of many encryption schemes, and also RSA over elliptic curves of non-zero characteristic. For larger r, the last design is not much more than half the size of the first, but uses only 50% more time. However, in all cases a choice of r can be made to make maximal use of the chip area available for the array, and thereby minimise latency. The processing elements consist of standard components, the main one being one or two multipliers for base r digits, which might be picked off the shelf. As the designs are linear with I/O through one end, the circuits are easy to lay out and can even be bent around corners to fit into any odd-shaped space. The time per exponentiation is $O(ndt)$ where the exponent has d bits, the modulus and input text have n digits base r and t is the execution time for the digit multiplier. There is no overhead for short inputs: they are proportionally faster and use the time expected from supplying smaller values of n and d.

Unlike parallel digit implementations using the classical modular multiplication algorithm, the array can be implemented using non-redundant representations, and yet there are no delays for carry propagation. This both increases clock speed by reducing critical path lengths and reduces area by eliminating redundancy and much of the communication wiring. Only local communication is required. There is no costly simultaneous digit distribution across the whole circuit. Furthermore, another efficiency gain comes through the sequential digit I/O, which helps to reduce the necessity of buffering data. The design now makes real time RSA decryption with 1024 bit keys closer on a single chip. Finally, the arrays will be of particular use in set top boxes and smartcards since they seem to give much greater protection against attacks using differential power analysis.

References

[1] G. B. Agnew, R. C. Mullin & S. A. Vanstone, "On the Development of a Fast Elliptic Curve Cryptosystem", *Proc. Eurocrypt '92*, R. A. Rueppel (ed.), Lecture Notes in Comp. Sci. vol. **658**, 1993, pp. 482-487, Springer-Verlag.

[2] 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, pp. 51-60.

[3] W. Diffie & M.E. Hellman, "New Directions in Cryptography", *IEEE Trans. Info. Theory*, vol. **IT-22**, no. 6, Nov 1976, pp. 644-654.

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

[5] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *IEEE Trans. Info. Theory*, vol. **IT-31**, no. 4, 1985, pp. 469-472.

[6] T. Hamano, N. Takagi, S. Yajima & F. Preparata, "O(n)-Depth Modular Exponentiation Circuit Algorithm", *IEEE Trans. Comp.*, vol. **46**, 1997, pp. 701-704.

[7] L. B. Jackson, J. F. Kaiser & H. S. McDonald, "An Approach to the Implementation of Digital Filters", *IEEE Trans. Audio & Electroacoustics*, vol. **AU-16**, 1968, pp. 413-421.

[8] Y. J. Jeong & W. P. Burleson, "VLSI Array Algorithms and Architectures for RSA Modular Multiplication", *IEEE Trans. VLSI Systems*, 1997, vol. **5**, no. 2, pp. 211-217.

[9] D. E. Knuth, *The Art of Computer Programming*, vol. **2**, "Seminumerical Algorithms", 2nd Edition, Addison-Wesley, 1981, pp. 441-466.

[10] N. Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, vol. **48**, 1987, pp. 203-209.

[11] N. Koblitz, *A Course in Number Theory and Cryptography*, Graduate Texts in Mathematics, vol. **114**, Springer-Verlag, 1987.

[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, vol. **1109**, N. Koblitz editor, Springer-Verlag, 1996, pp 104-113.

[13] P. Kocher, J. Jaffe & B. Jun, "Introduction to Differential Power Analysis and Related Attacks", *at* www.cryptography.com/dpa, 1998.

[14] P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms", *IEEE Trans. Comp.*, 1994, vol. **43**, no. 8, pp. 892-898.

[15] V. Miller, "Use of Elliptic Curves in Cryptography", *Proc. CRYPTO '85*, H. C. Williams (ed.), Lecture Notes in Comp. Sci., vol. **218**, 1986, pp. 417-426, Springer-Verlag.

[16] P. L. Montgomery, "Modular Multiplication without Trial Division", *Math. Computation*, vol. **44**, 1985, pp. 519-521.

[17] National Institute of Standards and Technology, "Digital Signature Standard", *NIST FIPS PUB* **186**, May 1994.

[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, pp. 156-165.

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

[20] E. M. Schartz, R. M. Averill III & L. J. Segal, "A Radix-8 CMOS S/390 Multiplier", *Proc. 13th IEEE Symp. on Computer Arith.*, Asilomar, CA, USA, 6-9 July 1997, IEEE Press, 1997, pp 2-9.

[21] M. Shand & J. Vuillemin, "Fast Implementations of RSA Cryptography", *Proc 11th IEEE Symp. on Computer Arith.*, Windsor, ONT, Canada , July 1993, pp. 252-259.

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

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

[24] C. D. Walter, "Space/Time Trade-offs for Higher Radix Modular Multiplication using Repeated Addition", *IEEE Trans. Comp.*, vol. **46**, 1997, pp. 139-141.

[25] "Standard Specifications for Public Key Cryptography", *IEEE Standard P1363*, Draft Version **7**, IEEE, New York, 1997, 1998.