Modular Exponentiation Using Recursive Sums of Residues

P.A.Findlay and B.A.Johnson (Hatfield Polytechnic, UK)

1. Summary

This paper describes a method for computing a modular exponentiation, useful in performing the RSA Public Key algorithm, suitable for software or hardware implementation. The method uses conventional multiplication, followed by partial modular reduction based on sums of residues. We show that for a simple recursive system where the output of partial modular reduction is the input for the next multiplication, overflow presents few problems.

2. Rivest, Shamir, and Adleman public key cryptosystem

The Rivest, Shamir, and Adleman (RSA) [6] public key cryptosystem uses exponentiations of the form:

$$y = x^k \bmod m.$$

where y is either ciphertext, or deciphered plaintext, and (k,m) form the enciphering/deciphering key. Note that k is different for enciphering and deciphering. In an encryption system offering a practical level of security, x,k, and m need to be 256 bits or more in length. Exponentiation is performed by repeated squaring operations, along with conditional multiplications by the original x, e.g.

$$x^5 \bmod m = ((x^2 \bmod m)^2 \bmod m).x \bmod m$$

Note that in each case, the previous modulo reduced result is fed back to be multiplied by itself, or by x. Modulo reduction

G. Brassard (Ed.): Advances in Cryptology - CRYPTO '89, LNCS 435, pp. 371-386, 1990.
© Springer-Verlag Berlin Heidelberg 1990

is associative, so can be carried out at each stage to prevent the intermediate results from growing too large.
Several algorithms for performing this exponentiation already exist . These can be divided into algorithms that are suitable for hardware implementation [1,3,5], and those that are suitable for software implementation [2,7].

3. The Method as implemented in Hardware

The core operation of exponentiation is modulo multiplication, and this can be performed in two ways:

a) Multiplication and reduction can be combined into a single operation. As multiplication partial products are formed, a decision is taken whether or not to perform a reduction on these partial products.

b) Multiplication and reduction are separate tasks, with the output of the multiplier feeding the input of the reduction unit.

For the purposes of this paper, case b) is being considered. Also, for the purposes of the hardware method, multiplication is best performed in a bit-serial form using a multiplier as described in [4,8]. These multipliers take the two arguments in bit-serial form, least significant bit first, and produce the product in bit-serial form, least significant bit first. They have the advantage that they are simple, are of cellular construction, and are easily expandable to larger bit-widths.

3.1 Sums-of-Residues Reduction

Modulo reduction can be performed by division, which is slow, or by trial subtractions incorporated into the multiplication that modify the partial products formed in the multiplication process [1,3]. It is suggested that neither of these is used

as a reduction method. If the number, P, to be reduced by the modulus, m, is expressed as a binary vector:

$$P = [p_1, p_2, ..p_n],\ p_i = \{0,1\} \text{ for } i = 1 \text{ to } 2n.$$

then the modulo reduction could be expressed as:

$$P \bmod m = \left(\sum_{i=1}^{i=2n} p_i . 2^{i-1}\right) \bmod m.$$

The modulus operation is associative, so the above could be expressed as:

$$P \bmod m = \left(\sum_{i=1}^{i=2n} p_i . (2^{i-1} \bmod m)\ \right) \bmod m.$$

Now the reduction is simply a conditional sum of powers of 2 reduced modulo m: "residues", hence the name of sums-of-residues (SOR) reduction. This reduction is simple to calculate, given a table of residue values [5].

The final modulus operation in the above equation is necessary because of the possibility of getting incomplete reductions. For example, 15 mod 13 using the above method is still 15. This may lead to the conclusion that a conventional reduction using division or other techniques is still required. In order to perform an exponentiation, these incomplete reductions could lead to the multiplications in the next exponentiation step overflowing. A limited overflow does occur, but one which is bounded above, and can be taken into account by using extra hardware. A simple derivation of this upper bound is now given.

3.2 Sums-of-Residues Overflow Bound

Let P be the result of a squaring or multiplication step, i, in the exponentiation, i.e.

$$P_i = X_i^2 \quad \text{or} \quad P_i = A.X_i;\ \text{where } A < m < 2^n,$$

and X_i is an intermediate result of bit-length L_i.

Clearly, the product will give a maximum bit-length following a squaring, rather than a multiplication, and hence the maximum bit-length of P_i is $2L_i$.

$$\text{Let } P = \sum_{j=1}^{j=2L} p_j.2^{j-1} \text{ define } p_j \text{, and}$$

$$\text{let } r_k = 2^{k-1} \bmod m, \text{ where } m < 2^n.$$

Then the next value of X will be $X_{i+1} = \sum_{j=1}^{j=2L_i} p_j r_j$

Simplifying things slightly, this new X value can be considered to be the sum of at most $2L_i$ n-bit numbers. Therefore,

$X_{i+1} < 2L_i(2^n-1)$, hence L_{i+1}, the max. bit-length of X_{i+1} is

$$L_{i+1} < \lceil \log_2(2L_i) \rceil + n.$$

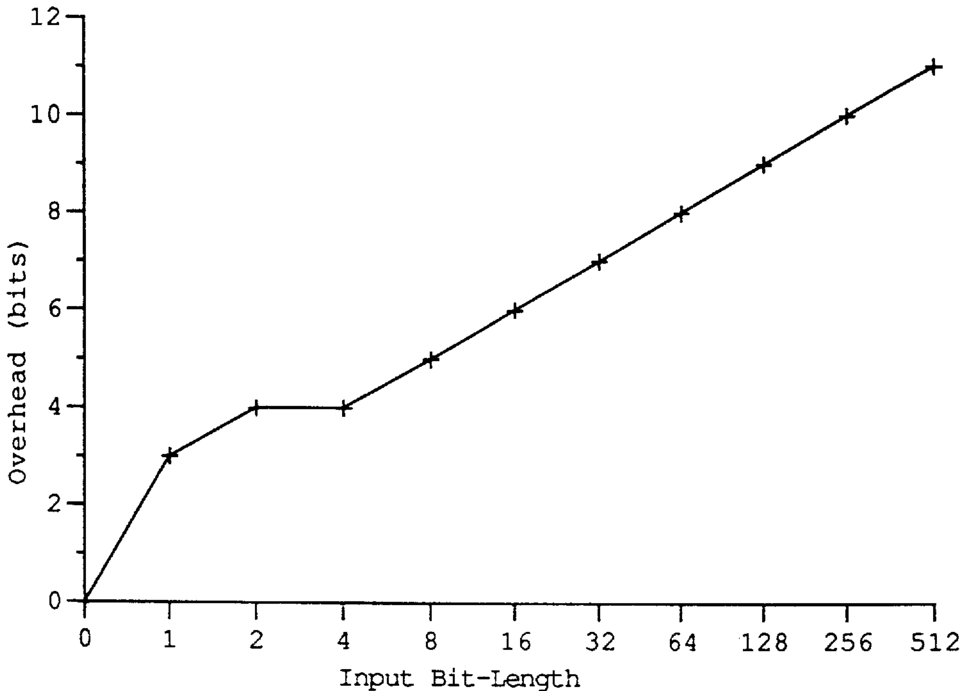

Figure 1: Bit-length overhead versus input bit-length

To find the maximum for L, the above equation is relaxed, starting with an initial L = n. A graph of the bit-length overhead, h (= L-n), is shown in figure 1.

The results show that for a typical input bit-length of 256 bits, the actual hardware is required to handle 266 bit numbers. This represents an overhead of only 4%.

Any final reduction then becomes necessary only at the very output of the exponentiation, and not at the output of each modulo multiplication.

3.3 Residue Calculation

The residues can be stored in a look-up table [5], and used when needed. This uses a modest amount of storage, typically 2n by n-bits. However, the data paths for this storage are very wide, and this is undesirable in silicon. It is easier to calculate the residues, r, as they are needed, and this can be done with a simple recursive formula:

$$r_i = \begin{Bmatrix} 2.r_{i-1} & \text{iff } (2.r_{i-1} - m) < 0 \\ 2.r_{i-1} - m & \text{iff } (2.r_{i-1} - m) \geq 0 \end{Bmatrix}, \quad i = 2..2n, \; r_1 = 1.$$

Figure 2 shows a possible architecture for sum-of-residues calculation corresponding to the serial product P, assuming that P appears lsb first. It is also assumed that n-bit encryption is being performed, and that the multiplier is able to handle (n+h) bit numbers.

Two n-bit registers, M and R, hold (-m); the two's complement of the modulus, and r, the current residue. The residue is initially set to 1. As the system is clocked, the residue register is reloaded with either (2r), or with (2r-m), depending on the sign bit of the (2r-m) calculation. An (n+h) bit accumulator sums those residues which are gated into it by the incoming 2(n+h) bits of the serial product P.

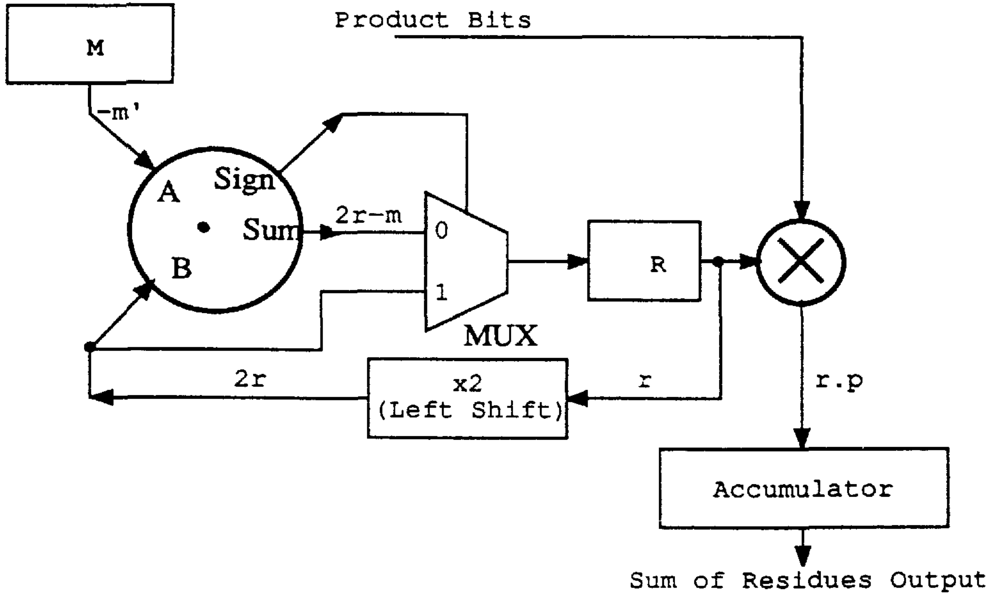

Figure 2: Parallel Sums-of-Residues Calculation

For practical sizes of n, the time taken for the sign bit to appear is prohibitive because of the long carry-propagate path. However, for a given modulus, the sequence of sign bits is always the same. Therefore, they can be precomputed by a host machine each time the modulus is changed, and stored as a sequence of bits in a shift register. (In an RSA system, the modulus changes only when the enciphering or deciphering keys are changed). This allows the carry chain to be pipelined.

In fact, not all of the sign-bit sequence needs to be stored. Suppose m is q bits long, $q \le n$, then

$$2^{q-1} < m < 2^{q}, \text{ so for } i = 1 \,..(q-1),\; r_{i+1} = 2r_i = 2^{i-1}.$$

and hence the sign bits are all the same.

To make the most of this fact, a 'working modulus', m', can be used; being the actual modulus multiplied by an appropriate power of 2 (left shifted) to make an n-bit number, i.e.

$$m' = m.2^{n-q}$$

The sum of residues derived from the working modulus will be congruent to that of the real modulus, and the bit length of the result will still not exceed the upper bound previously described. The first n bits of the product P, may be loaded directly into the accumulator, and the residue register, R, may be preloaded with (-m') in two's complement form; as this is the residue of 2^{n+1} mod m'.†

So far the SOR system has been described as being essentially parallel in operation, using the serial output of the multiplier to gate a parallel accumulation of the residues. This parallel architecture may be transformed into a bit serial array format by 'skewing' the original data paths in both time and space. The modulus and residue registers are now effectively shift registers, and the sign bit register is essentially static. Static storage is also required for the (n+2h) most significant data bits, hence (n+2h) 'cells' are needed if they are all to be identical (essential for expandability). Each cell contains two full adders, a 2:1 multiplexer, nine flip flops, and one AND gate. The functional description of a cell is shown in figure 3.

Each 'T' block in figure 3 represents a commonly-clocked storage element. The SELECT signal is a pulse that is passed from one cell to the next at each clock signal, and the arrival of this signal will cause the current input on the global PRODUCT line to be stored into the P-register. The two's complement of the working modulus (-m') circulates in the shift register formed by the MODULUS, MODULUS(out) chain. ADR1

---

† Note: if the lower n bits of P represent a number,≥m', then the preloading of the accumulator results in m' being added to the accumulation. However, this still yields a final sum that is congruent to the true residue, and within the maximum bit length defined. It does however mean that the sign bit register is (n+2h) long, as opposed to (n+2h+1).

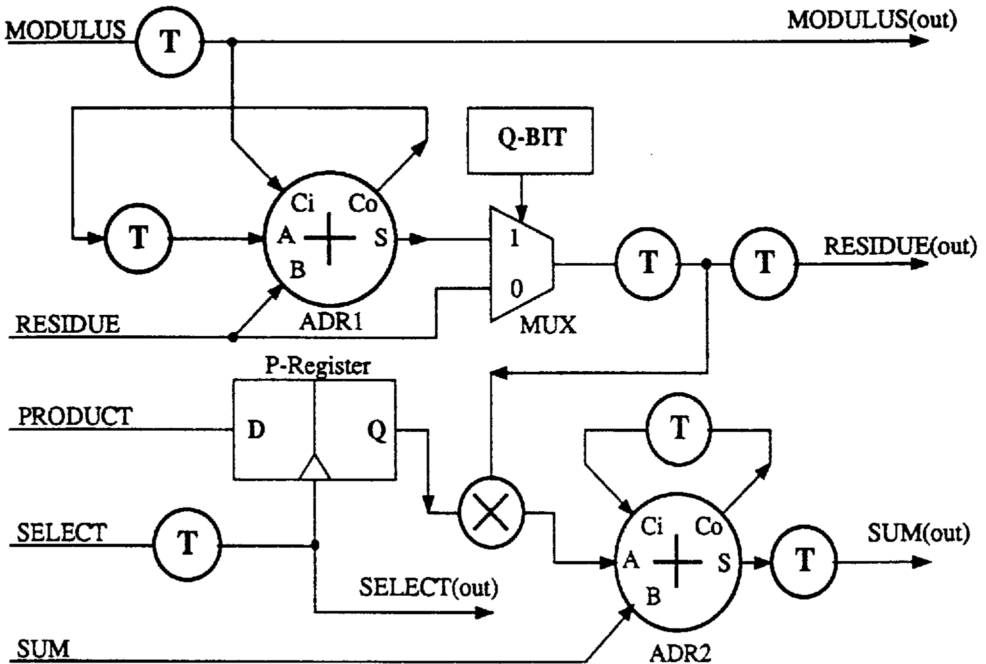

Figure 3: SOR reduction hardware cell

corresponds to the adder required in calculating the (2r-m′) term in the SOR residue calculation. The sign bit (Q-bit) for the cell is permanently stored, having previously been set up by the host system, and is used to gate either 2r or 2r-m′ via multiplexer MUX. The result is propagated to the next cell via double storage elements. This double delay achieves the doubling of the residue presented to the next cell. The single delayed ADR1 output is gated by the stored product bit into the carry-save adder ADR2. Using the single delayed residue result further pipelines the design, making the critical timing path within the cell little more than a single adder delay.

The sum of residues is calculated as follows: consider time step i=1..2(n+h), and cell number j=1..(n+2h).

During time steps i = 1..n, the array stores the least n bits of the product P.

During time steps i = n..2(n+h)-1, cell j calculates the (i-j-n+2)th bit of the residue modulo m′ of 2^{j+n}

During time steps i = (n+1)..2(n+h), cell j also holds the (j+n)th product bit, which gates the accumulation of the (i-j-n)th bit of the residue of 2^{j+n}, for all j ≤ (i-n).

The accumulator is effectively preloaded by bits 1..n of the product. In practice this is achieved by adding bit (i-n) of the product to the accumulating adder ADR2 during time steps i=(n+1)..2n. Array operation is illustrated for cells 1,2,3 at time steps i=n,n+1,n+2 in figure 4 below. Note that P is the product to be reduced, and that the carry terms for the accumulator are not shown.

Time Step		Processing Element		
		1	2	3
n	**ADR1 Calculates:**	**Bit 1 of** 2^{n+1} mod m'		
	ADR2 Accumulates:			
n+1	**ADR1 Calculates:**	**Bit 2 of** 2^{n+1} mod m'	**Bit 1 of** 2^{n+2} mod m'	
	ADR2 Accumulates:	**Bit 1 of P + Bit 1 of** 2^{n+1} mod m'		
n+2	**ADR1 Calculates:**	**Bit 3 of** 2^{n+1} mod m'	**Bit 2 of** 2^{n+2} mod m'	**Bit 3 of** 2^{n+3} mod m'
	ADR2	**Bit 2 of P + Bit 2** 2^{n+1} mod m'	**Bit 1 of** 2^{n+2} mod m'	

Figure 4: Array Operation after time step (n-1)

This results in the sum-of-residues appearing serially, lsb first, at cell (n+2h). The latency between input of the product msb, and output of the sum-of-residues lsb is just one clock cycle.

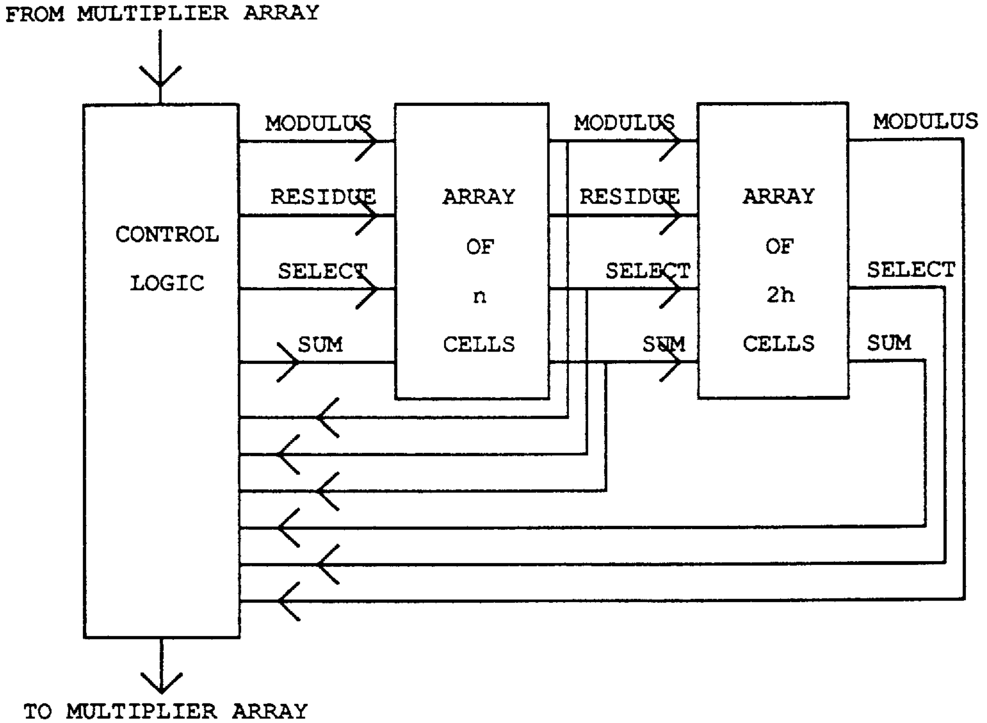

Figure 5: SOR Array interconnect

A scheme for interconnecting a practical sum-of-residues array is shown in figure 5. This also shows part of the accumulator being used as a shift register to buffer the first n bits of the product P. The feedback paths for the 'shifting' modulus and a circulating 'select' pulse are also included. The function of the select pulse allows the array to be self-sizing, and hence easily expandable.

3.4 Practical Exponentiator Hardware

A practical hardware modulo exponentiation system will therefore consist of a hardware bit serial multiplier, followed by a SOR reduction unit, as shown in figure 6. Extra hardware is required for the exponent and plaintext registers, and to control the flow of reduction unit output into the multiplier.

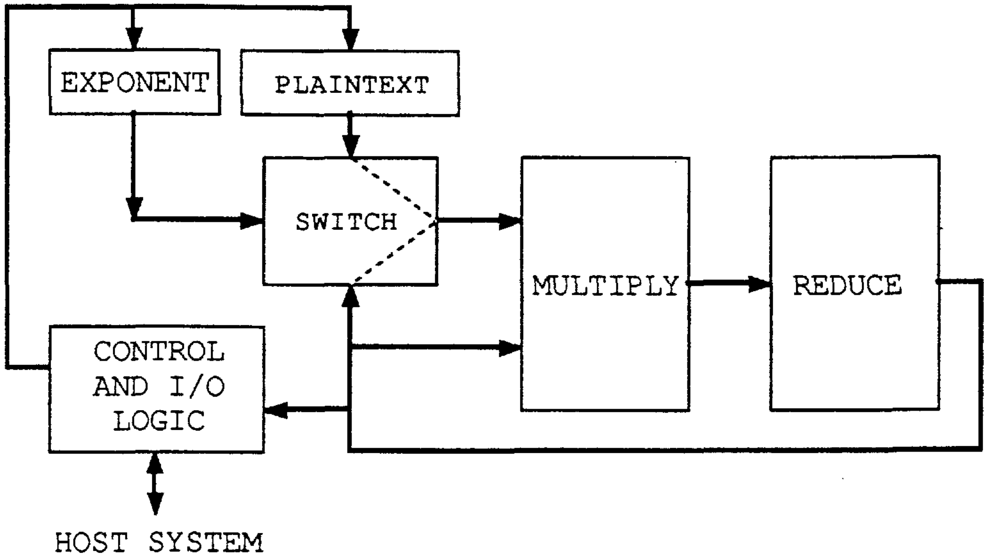

Figure 6: Practical Exponentiator Hardware System

3.5 Advantages and Disadvantages of this Architecture

This implementation of the RSA algorithm requires slightly more hardware than some of the existing systems [1,3]. It requires roughly twice as many clock cycles to perform a modular multiplication, but each clock cycle requires only 8 gate delays, and no broadcast fanout logic - as opposed to the 20-28 or so gate delays plus fanout time required for the two other schemes mentioned. Therefore, a speed improvement of approximately 50% would be expected using similar process technology.

Broadcasting is only required at the interface between the multiplier and the reduction unit, hence an extra clock cycle can be allowed for broadcast propagation with minimal loss of throughput.

The system needs minimal host support for loading the modulus, performing the final reduction, etc. These tasks could be handled by a dedicated single chip microcomputer.

4. Partitioned Sums-of-Residues

We now give a generalisation of the method that can be used with both hardware and software, at the cost of higher overheads, either in hardware or operational complexity.
In a parallel system, treating the multiplier output product, P, one bit at a time is inefficient, as observed in the previous section. However, for the price of introducing greater redundancy in the result, the product can be partitioned into discrete words; each word being multiplied by the residue of the least significant power of two that the word begins with, and the results accumulated. This still gives a result congruent to the true reduced result.

Consider a single multiply and SOR step. Suppose the product is 2n bits long, and is divided up into c-bit long words. This is illustrated in figure 7, where n = 2c.

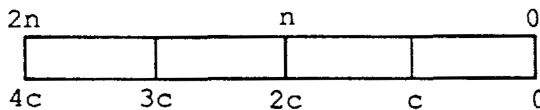

Figure 7: Partitioning the Product into c-bit Segments

A look-up table holds the residues of $2^0, 2^c, 2^{2c}, 2^{3c}$, etc.
Each c-bit segment of the product is then multiplied by the appropriate residue, and the results accumulated. For instance, if n=8, and c=4, then 50000 mod 23 (=21) is computed as:
$(0*2^0 \bmod 23) + (5*2^4 \bmod 23) + (3*2^8 \bmod 23) + (12*2^{12} \bmod 23) = 113$
which is congruent to the true result.

The overhead incurred during exponentiation using recursive partitioned sums-of-residues (PSOR) must be calculated. Using a simple approach to maximum bit-length calculation, the ith multiplication followed by by PSOR reduction will yield a SOR of bit-length not more than:

$$L_{i+1} = \left(n + c + \left\lceil \log_2 \left\lceil 2.L_i/c \right\rceil \right\rceil\right).$$

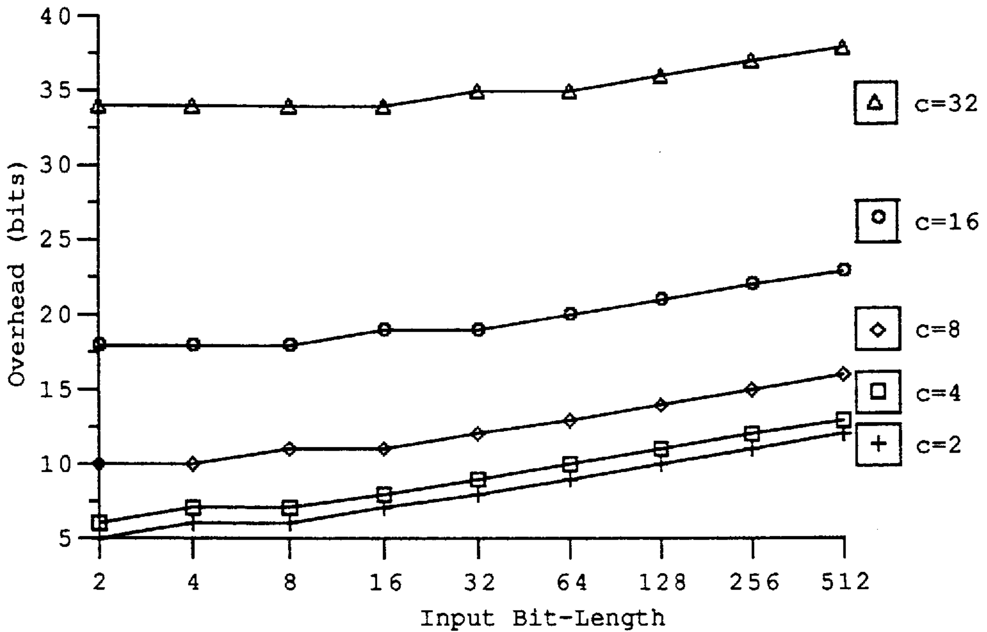

Figure 8: Graph of bit-length overheads for PSOR scheme

Note: this expression is not valid for the case c=1, as multiplying a number by 1 does not increase the bit-length.

To simulate exponentiation, and hence find the maximum bit-length obtained, the above equation is relaxed starting with L=n, until L reaches a maximum. Figure 8 is a graph of maximum bit length overhead, h, versus modulus bit-length n, for various values of c, the partition width.

It is feasible to use the "working modulus" optimisation again, and thereby simply accumulate the first n bits of the non-reduced product. This decreases the size of any look-up table, and the number of multiplies required to perform the PSOR.

The operations used in PSOR could be performed by dedicated hardware multiplier-accumulator chips, or as multiply and add instructions in software. The same hardware or software instructions could be used to generate the non-reduced product.

The total number of fixed-length multiply-accumulate cycles to perform the complete PSOR modulo multiplication is now determined. The multiplication to form the product requires

$$\left(\left\lceil (n+h)/c \right\rceil\right)^2$$

operations, using a c-bit multiply function. The PSOR reduction would be expected to take

$$\left\lceil (n+2h)/c \right\rceil * \left\lceil n/c \right\rceil$$

operations using the same function. In practice, it would make sense to adjust n such that either the left-hand or right-hand term above is an integer. Figure 9 shows total numbers of multiply-accumulate cycles necessary for a complete multiplication and PSOR reduction.

n	c	Mult-Accs.
256	8	2308
256	16	628
256	32	188
283	32	188
474	32	541

Figure 9: Typical numbers of Multiply-Accumulate cycles for PSOR

Finally, the storage requirements for the look-up table for PSOR operation will be considered. Assuming a working modulus is used, a table of

$$\left\lceil (n+2h)/c \right\rceil$$

n-bit residues will be needed, i.e. a memory requirement of

$$\left\lceil (n+2h)/c \right\rceil * n \text{ bits.}$$

Figure 10 below shows the residue look-up capacity for the set of "typical" values given in figure 9 above. The memory requirement is also expressed as number of registers, or memory locations, assuming that they are the same bit-width as the multiplier.

n	c	Residue Bits	No.c-bit Registers
256	8	17408	2167
256	16	8960	560
256	32	4864	152
283	32	5120	160
474	32	8192	256

Figure 10: Typical Residue Table sizes for PSOR algorithm

5.Conclusions

An optimised hardware algorithm has been proposed for performing RSA public key encryption. This algorithm lends itself to a high speed efficient VLSI implementation of an encryption system, using serial data and one-dimensional semi-systolic arrays. The system consists of a serial multiplier array coupled with a unique serial sum-of-residues reduction array. Based on this architecture, it is possible to build an easily expandable RSA engine with hardware complexity O(n) and speed proportional to $1/n^2$.

The partitioned sums-of-residues method has significant advantages over the bit-level method as far as operation on standard hardware/software is concerned. It is readily implemented with typical Digital Signal Processing chips, and could be programmed efficiently on any general-purpose machine with a fast integer multiply function, and a large register set (or a data cache). No bit testing or conditional branching need be performed, hence the algorithm would run extremely efficiently on highly pipelined processors, or RISC machines, since no instruction queue flushing would be required.

6.Acknowledgements

The authors would like to thank John Guppy, the British Aerospace (Dynamics) Technology Executive, for the contribution

of his mathematical expertise. The original theoretical work behind this paper formed part of B.A.Johnson's PhD programme, which was funded by the UK Science and Engineering Research Council. The work in this paper is the subject of a patent application.

7.References

[1] Baker, P.W.
'Fast computation of A*B mod N'
IEE Electronics Letters, Vol.23, No.15,16 July 1987, pp794.

[2] Blakley, G.R.
'A Computer Algorithm for Calculating the Product AB Mod M'
IEEE Trans. Comp. Vol.C32 No.5 May 1983

[3] Brickell, E.F.
'A Fast Modular Multiplication Algorithm with application to two-key Cryptography'
In "Advances in Cryptology",conf. proc. CRYPTO'82, Plenum Press, 1982.

[4] Ngo-Chen, I., Willoner, R.
'An O(n) Parallel Multiplier having Bit Sequential Input and Output'
IEEE Trans. Comp. Vol.C28 No.10 Oct.1979

[5] Ngo-Chen, I., Willoner, R.
'An Algorithm for Modular Exponentiation'
Proc. 5th Symp. Comp. Arith. IEEE 1981

[6] Rivest, R.L., Shamir, A., and Adleman, L.
'On Digital Signatures and Public Key Cryptosystems',
Comms. ACM, Vol.21, No.2, Feb. 1978 pp120-126

[7] Selby, A., Mitchell, C.
'Algorithms for Software Implementations of RSA'
IEE Proc. May 1989 Vol.136 Part E No.3 p166

[8] Strader,N.R., Rhynne, V.T.
'A Canonical Bit Sequential Multiplier'
IEEE Trans. Comp. Vol.C31 No.8 Aug.1982