Techniques for the Hardware Implementation of Modular Multiplication

by

Colin D. Walter

Computation Department, U.M.I.S.T.,
PO Box 88, Sackville Street, Manchester M60 1QD, U.K.

C.Walter@co.umist.ac.uk http://www.co.umist.ac.uk/

Index Terms: *Computer arithmetic, cryptography, RSA cryptosystem, Montgomery modular multiplication, redundant number systems, higher radix representation, optimal speed, area complexity.*

Abstract

Hardware for modular multiplication is required for strong cryptosystems so that large volumes of encrypted data can be safely stored in public areas and obtained over a public network (e.g. the WWW) but only understood by authorised users. This article reviews the main bottlenecks which may arise in the more obvious implementations of RSA and outlines a variety of solutions to them so that plain text can be recovered in real time.

1. Introduction

RSA [1] is an arbitrarily strong, two key cryptosystem which is generally only currently used for authentication protocols and for exchange of session keys to enable secure communication by arithmetically less intensive encryption methods. However, there is a growing need for secure encryption of large volumes of data. In particular, this is the case for data accessible over the internet which is commercially sensitive and should only be comprehensible to certain readers. It is also the case for much private data which is stored on public servers. The quantity of such data can make weak cryptosystems essentially useless in such situations. However, software implement-ations of a strong cryptosystem such as RSA are too slow for decrypting retrieved data in real time for viewing or editing purposes. Only dedicated hardware can achieve a speed equal to the disk access times, internal bus speeds, etc. which dictate the retrieval times for unencrypted data.

An RSA cryptosystem consists of a modulus M 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 $A' = A^e \bmod M$ and so uniquely decrypted by $A = A'^d \bmod M$ using the same algorithm for both processes. $M = PQ$ is chosen as a product of two large primes, e is often small with few non-zero bits so that encryption is relatively fast, and d chosen to satisfy $de \equiv 1 \bmod (P{-}1)(Q{-}1)$. Thus d has as many bits as M. The owner of the cryptosystem publishes M and e but keeps secret the factorization of M and the key d. Breaking the system requires discovering P and Q, which is computationally infeasible. Indeed, each addition of a small, almost constant number of bits (around 15) to the size of M doubles the effort required for this [2].

This paper discusses the major problems associated with space and time efficient hardware implementation of the cryptosystem and reviews their solution. Among the issues of concern are carry propagation, digit distribution, buffering, communication and use of available area.

2. Notation

The computation of $A^e \bmod M$ is split into two main processes: modular multiplication and exponentiation. The exponentiation is discussed last as it has to make repeated use of the modular multiplication hardware. Thus we look first at computing $(A\times B) \bmod M$.

Each number X has a representation of the form $X = \sum_{i=0}^{n-1} x_i r^i$ where r is the radix or base (usually a power of 2) and x_i is the ith digit (usually $0 \le x_i < r$). Let n be the number of digits for this representation of M. The representation is *redundant* if numbers can be represented in more than one way. Here this occurs by allowing digits in a range larger than $0..r-1$, and typically that given by an extra (carry) bit, that is, $0..2r-1$. For example, the output from a carry-save adder provides two bits for each digit and so, in effect, is a redundant representation where digits lie in the range 0..2 rather than the usual 0..1. We use k for the number of bits used to represent a digit, so that kn is approximately 2^{10} here, and $O(kn)$ is effectively constant.

The choice of k and n splits the formation of $A\times B$ into two multiplication levels: forming products of k-bit digits and forming products of n-digit numbers. More levels could be created. Different algorithms are used for each level. The lower level is of combinational logic, possibly pipelined to increase throughput. The hardware will be built around n k-bit digit multipliers, and this defines our choice of k. The upper level views a product as a sequence of additions of digits multiples a_iB of the multiplicand, and so will take a number of clock cycles.

3. Multiplication

Practical planar designs are well known for multipliers which are optimal with respect to some measure of time and area [3]-[8]. Under a model which assumes that wires take area but do not contribute to time, $Area\times Time^2$ 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 $\sqrt{k}$ [6]. 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. Since the cross-over point is claimed to be around $k = 3\times10^3$ bits [5], i.e. greater than the size of the numbers here, classical methods are preferable. Indeed, it makes sense to pick a standard k-bit combinational multiplier off the shelf since it will contain years of optimisation, it will have a known latency and will be known to be correct. For a current standard chip of perhaps 10^7 transistors devoted entirely to RSA, $k = 32$ or 64 is the maximum practical [9] since there must be space for registers and for other operations such as modular reduction.

There is a direct trade-off between time and area. Doubling the number of digit multipliers allows the parallel processing of twice as many digits and so halves the time taken. If the k-bit multiplier works in one cycle with no pipelining and k is roughly the bandwidth of the internal bus, it is easy to calculate that sufficient throughput for real-time decryption requires n multipliers so that a complete multiplier digit times multiplicand a_iB (or equivalent) can be computed in one cycle. Then, in effect, each cycle processes all the multiplicand bits with k of the multiplier bits. A different regime would lead to more complex data paths and hence less efficient use of chip area, and so is not considered.

Suppose therefore that the n digit multipliers are used to add a_iB to a partial product in one clock cycle. If the carries are propagated then this takes extra time beyond the digit multiplic-ations. The digit multipliers ought not to lie idle while this happens. The equivalent of a carry save adder might be used to avoid carry propagation, so that the partial product has a redundant representation [10]. Alternatively, successive a_iB can be pipelined: either $a_{i+1}B$ can be formed as the previous carries are propagated or, if a_ib_j and its carry are calculated on one cycle, the next multiplier along should use this carry on the next cycle to compute a_ib_{j+1} and another carry [9].

4. Modular Reduction

The reduction of $A\times B$ to $(A\times B)$ mod M can be carried out in several ways, but each involves repeatedly choosing a suitable digit q and subtracting a shifted qM from the current remainder. The successive choices of digit q can be pieced together to form the integer quotient $Q = (A\times B)$ div M or a closely related quantity if desired. Classically, shifted multiples of M are subtracted from the most significant end of $A\times B$:

```
{ Pre-condition: 0 ≤ A×B < M×r^n }
R := A×B ;
For i := n–1 downto 0 do
Begin
    q := R div (M×r^i) ;
    R := R – q×M×r^i   ;
    { Invariant:  0 ≤ R < M×r^i
                  & R ≡ (A×B) mod M }
End ;
{ Post-Condition: R = (A×B) mod M }
```

This requires waiting for a full carry propagation with each subtraction if the largest possible multiple is to be removed. A better solution is just to use the top digit or two of M and the remaining product to determine a sufficiently good multiple of M to remove [10]. This reduces the result enough to guarantee an upper bound of rM, say, when the process terminates. This can be cleaned up properly at the end of the de-cryption. The critical path is now in the circuitry for computing q, but this can be reduced by scaling M [11,12]. M is replaced by a small multiple so that its top digits are known and simple, say, 10... Minor post-processing will again recover the correct residue.

If the product $A \times B$ is fully computed before the modular reduction starts, then a register of $2n$ digits is required. However, if the product is performed by repeated shifting and addition, the modular reductions can be interleaved with the additions to keep the partial sum down to only about n digits, thereby saving space. This has the added advantage that the multiplier hardware and modular reduction hardware can work simultaneously on the same modular product; otherwise full utilisation of these two parts of the hardware would require the complication of handling two modular products at once.

The modular reduction requires another n digit multipliers to compute qM, so that the main area taken up by the RSA circuitry is $2n$ $k \times k$-bit multipliers and an adder to combine their outputs. It has been suggested that a table be formed containing some or all digit multiples of M in order to avoid re-computing them so many times. This is unwise as it requires $O(2^k)$ entries, so the time and space requirements would exceed those of re-computation unless k were very small.

5. Montgomery's Algorithm

The above modular reduction method has several disadvantages. It requires a redundant representation (which takes up more space) to avoid carry propagation, makes a poor choice of multiple to subtract, takes time to compute the digit q, and requires the global broadcasting of q to each digit position. Peter Montgomery [13] has shown how to use the *least* significant digit of an accumulating product to determine the multiple of M to subtract. He reverses the usual multiplication order by choosing multiplier digits from *least* to *most* significant and shifting *down* on each iteration. If R is the current partial modular product, then q is chosen so that $R+qM$ is a multiple of r, and this is shifted down (i.e. divided by r) for use in the next iteration. Consequently, $(A \times B \times r^{-n}) \bmod M$ is computed:

```
{ Pre-condition: 0 ≤ A < r^n }
R := 0 ;
For i := 0 to n−1 do
Begin
    q := (-(r_0+a_i×b_0)m_0^-1) mod r;
    R := (R+a_i×B+q×M) div r ;
    { Invariant: 0 ≤ R < M+B }
End ;
{ Post-Condition: R ≡ (A×B×r^−n) mod M }
```

The extra factor, a power of r, is easily cleared up in minor post-processing [14]. Any extra multiple of M is also easily removed.

With this algorithm, the digit q is computed from the lowest digits of R and M without waiting for any carry propagation. So pipelining of the digits can now take place with $a_i b_{j+1}$ computed on the cycle after $a_i b_j$ using its carry and the same values of q and a_i. Thus, a non-redundant representation can be used and q and a_i no longer need to be broadcast to all digit slices in the same clock cycle.

Once more the critical path length is in computing q. To reduce this path M is again scaled [15], this time ensuring that its lowest digits are known and simple, say ...01. This moves the critical path to within the multiplier of each digit slice, so that most of the hardware is operating at full capacity. Further optimisation must concentrate on the digit slice and improved communications.

6. Communications

Our goal is to have only local inter-communication because of the delays and wiring associated with global movement of data. The standard approach, namely parallel processing of digit operations for the same multiplier digit, requires broadcasting q and a_i to all digit slices on each cycle [10,12]. Avoiding this requires pipelining digits as described above and yields a systolic array [16,17,9].

If the term $a_i b_j$ were added to the partial result in digit slice j on cycle $i+j$, the first output digit would occur at time $n-1$. Full utilisation of the hardware could only occur if another modular multiplication were to start at time *n* when digit slice 0 becomes free again. Fortunately, the exponentiation involved in the cryptosystem means that the output from one modular multiplication is the input to the next modular multiplication, and so no time is lost and little wiring and buffering is required to achieve this. The precise timing details are actually slightly more complex because of the shift down in Montgomery's algorithm [16,17,9].

Since this system requires input digit serially, *k* bits at a time for each number and produces output similarly, its I/O matches internal bus speeds and bandwidth and so reduces the need for on chip buffering of data.

7. Exponentiation

The exponentiation required for encryption and decryption is generally achieved simply by incorporating several registers and making repeated use of the modular multiplier hardware. If the exponent *d* has the same size as *M*, namely about *nk* bits, then the usual square and multiply algorithm for exponentiation takes between *nk* and 2*nk* modular multiplications, and 1.5*nk* on average. There are ways to reduce this towards *nk* [18,19] but the possible improvements are very limited. Much more is achieved by good design for the modular multiplier.

8. Conclusion

We have reviewed the main bottlenecks which may arise in hardware for implementing the RSA cryptosystem and shown how to make the most efficient use of area. The key solutions are to use Montgomery's modular multiplication algorithm [13], scale the modulus, pipeline digit products and use an existing *k*-bit combinational multiplier. Finally, *k* is chosen as large as possible to make full use of the available chip area so that real time decryption is achieved.

References

[1] 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.

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

[3] R. P. Brent & H. T. Kung, "The Area-Time Complexity of Binary Multiplication", *J. ACM*, vol. **28**, 1981, pp. 521-534.

[4] R. P. Brent & H. T. Kung, "A Regular Layout for Parallel Adders", *IEEE Trans. Comp.*, vol. **C-31**, no. 3, March 1982, pp. 260-264.

[5] 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, pp. 155-168.

[6] K. Mehlhorn & F. P. Preparata, "Area-Time Optimal VLSI Integer Multiplier with Minimum Computation Time", *Information & Control*, vol. **58**, 1983, pp. 137-156.

[7] F. P. Preparata & J. Vuillemin, "Area-Time Optimal VLSI Networks for computing Integer Multiplication and Discrete Fourier Transform", *Proc. ICALP*, Haifa, Israel, 1981, pp. 29-40.

[8] C. S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Trans. Electronic Computers*, vol. **EC-13**, no. 2, Feb. 1964, pp. 14-17.

[9] C. D. Walter, "Redundant Arithmetic is not necessary for Fast Modular Exponentiation", submitted to *IEEE Trans on Comp*.

[10] 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.

[11] C. D. Walter, "Faster Modular Multiplication by Operand Scaling", *Advances in Cryptology - CRYPTO '91*, J. Feigenbaum (ed.), Lecture Notes in Comp. Sci. vol. **576**, 1992, pp. 313-323, Springer-Verlag.

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

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

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

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

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

[17] 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.

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

[19] C. D. Walter, “Exponentiation using Division Chains”, *IEEE Trans. Comp*., vol. **47**, 1998, no. 7, *to appear*.

26th May 1998