A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N

Holger Orup

Computer Science Department
Aarhus University
DK-8000 Aarhus C, DENMARK
e-mail: orup@daimi.aau.dk

Peter Kornerup

Dept. of Math. and Computer Science
Odense University
DK-5230 Odense M, DENMARK
e-mail: kornerup@imada.ou.dk

Abstract

In a class of crypto systems fast computation of modulo exponentials is essential. The popular RSA protocol uses operands of more than 500 bits to achieve a sufficient security. We present a parallel version of a well known exponentiation algorithm that halves the worst case computing time. It is described how a high radix modulo multiplication can be implemented by interleaving a serial-parallel multiplication scheme with an SRT division scheme. The problems associated with high radices are efficiently solved by the use of a redundant representation of intermediate operands. We show how the algorithms can be realized as a highly regular VLSI circuit. Simulations indicate that a radix 32 implementation of the algorithms is able of computing 512 bit operand exponentials in 3.2 msec. This is more than 5 times faster compared to other known implementations.

1 Introduction

The concept of two-key crypto systems was introduced by Diffie and Hellman [5] in 1976. In 1978 Riverst, Shamir and Adleman [13] published an encryption scheme based on computing exponentials. The RSA scheme realizes encryption as put forth by Diffie and Hellman. Both encryption of a message and decryption of an encrypted message are done by computing an exponential $M^E \bmod N$, $M \in [0, N[$. The message is denoted M and the key (E, N). Modulus N is a product of two very large primes and the security of the system depends on the length of the keys. To achieve a sufficient security key lengths of 500-600 bits are necessary.

For an implementation of the RSA protocol it is crucial to calculate modulo exponentials in a rate corresponding to the transmission rate between transmitter and receiver to avoid the encryption to be a bottle neck in the communication system. Brickell [4] has made a survey of hardware implementations of RSA. In his paper chips from Cryptech [7] are the fastest, with a computing time of 512 bit messages corresponding to a rate of $17\frac{\text{Kbit}}{\text{sec}}$. Thorn EMI [15] has made chips that encrypt 512 bit messages at $29\frac{\text{Kbit}}{\text{sec}}$.

In this paper we will present a method for obtaining rates of more than $150\frac{\text{Kbit}}{\text{sec}}$. Section 2 describes how we construct a new and faster algorithm for performing exponentials by splitting the computation into parallel multiplications. The implementation of a high radix modulo multiplication is elaborated in sections 3, 4 and 5, where it is shown how to interleave a multiplication with a SRT division scheme and how the algorithm can be realised in VLSI design. Section 6 discusses the performance of a VLSI implementation. The summary indicates how the methods can be generalised to even higher radices.

2 Exponentiation

A commonly known algorithm for performing an exponentiation is named *Russian Peasant* [8]. It performs the computation as a number of multiplications and squarings proportional to the bit length of the exponent. The algorithm can be modified to perform a modulo exponentiation by substituting the multiplications and squarings for modulo multiplications and modulo squarings [13]. Below is shown a variant in which the exponent E is read from the least significant bit. The i'th bit of E is denoted e_i. In the curly brackets is an invariant for the loop.

If we denote the bit length of M,E and N by n the worst case time is

$$T[\text{Exp}, n] = 2nT[\text{Mult}, n]$$

and the average time is

$$T[\text{Exp}, n] = \frac{3}{2}nT[\text{Mult}, n],$$

where it is assumed that the computing time for squaring and multiplication is identical.

Observing that the three statements in the loop are independent of each other it is possible to speed up the algorithm by performing *two* multiplications in *parallel*. In this way the computing time is reduced to

$$T[\text{Exp}, n] = nT[\text{Mult}, n], \tag{1}$$

CH3015-5/91/0000/0051$01.00 © 1991 IEEE

```
Algorithm:
    Modulo exponentiation.
Stimulation:
    E, M, N, where E ≥ 0 and 0 ≤ M < N.
Response:
    X = M^E mod N
Method:
    i := 0;
    X := 1; Y := M;
    WHILE i < n DO
        {*X · Y^(E div 2^i) ≡_N M^E*}
        IF e_i = 1 THEN
            X := (X · Y) mod N
        END;
        Y := (Y · Y) mod N;
        i := i + 1;
    END;
```

Algorithm 1: *Variant of Russian Peasant for modulo exponentiation.*

which is independent on the number of 1–bits in E. It is hard to imagine how an exponentiation can be performed with less than n squarings.

Note that by distributing the modulo reduction to squaring and multiplication the bit-length of intermediate operands is bounded. This makes the exponentiation algorithm feasible to implement for large values of n.

3 Multiplication

To implement the exponentiation algorithm mentioned above we need an efficient way to perform modulo multiplication. Several algorithms have been presented [3] [2] [1] [10] [7]. All of them use radix 2. We will follow the approach in e.g. [3], where the modulo reduction is further distributed in the algorithm for multiplication. A similar approach is followed in [9] where a radix 4 algorithm is elaborated.

The usual way of multiplying is by scanning the multiplier serially from the *least* significant digit and parallelly adding a multiple of the multiplicand followed by a *right* shift of the partial product [14]. This gives a maximal carry ripple length of the parallel additions corresponding to the length of the multiplicand.

In the algorithm shown below we scan the multiplier from the *most* significant digit and the partial product is *left* shifted. In an ordinary multiplication this would result in a maximal carry ripple length equal to the sum of the multiplier length and the multiplicand length. But since we perform a modulo reduction on the partial product in every iteration, the length of the intermediate operands will be limited to n plus a few digits. Assume we scan the multiplier k bits at a time, corresponding to processing a digit in radix 2^k, we can express the serial-parallel multiplication scheme as in Algorithm 2.

```
S := 0; i := n' − 1;
WHILE i ≥ 0 DO
        S := (2^k S + a_i B) mod N;
        i := i − 1;
END;
```

Algorithm 2: *Serial-parallel multiplication with integrated modulo reduction.*

In this algorithm S is the accumulator, n' the number of radix 2^k digits, a_i is digit number i of the multiplier, B the multiplicand and N the modulus.

The modulo reduction can be carried out by *interleaving the multiplication with a division.* Division is usually performed by inspecting the partial remainder and subtracting a multiple of the divisor, followed by a *left* shift of the resulting partial remainder. In Algorithm 3 the variable S has the dual role of a partial remainder in a SRT division scheme [14] and of a partial product in a serial-parallel multiplication. This is the reason why we want to scan the multiplier from the most significant digit and left shift the partial product. Note that this

```
S := 0; i := n' − 1;
WHILE i ≥ 0 DO
        q := Estimate(S div N);
        S := 2^k S + a_i B − 2^k qN;
        i := i − 1;
END;
Correction of S;
```

Algorithm 3: *Modulo multiplication with quotient estimation.*

method is only feasible if we are able to generate the multiples $a_i B$ and qN rapidly.

Instead of calculating the *exact* value of the quotient digit, which is a tedious task for long operands, we *estimate* a value of q. This, of course, implies that the final result is not necessarily completely modulo reduced, and a correction must be performed after the loop by subtracting N until S belongs to the correct interval. It is then required that the precision of the estimate is chosen such that the range of S does not diverge during a computation. Assuming that

$$\begin{aligned} a &\in \{0, 1, \ldots, 2^k - 1\} \\ B &\in [0; 2N[\\ q &\in \{0, 1, \ldots, q_{max}\} \end{aligned}$$

we can derive a range restriction, which must be satisfied by the estimated q, in the following way

$$\begin{aligned} 2^k S + aB - 2^k qN &\le q_{max} N \\ S - qN &\le \frac{q_{max} N - aB}{2^k} \\ S - qN &\le \frac{q_{max} - 2(2^k - 1)}{2^k} N. \end{aligned} \tag{2}$$

As we shall see later we can construct hardware that generates multiples qN efficiently if the range of q belongs to $[0; 42]$. Since q is non negative it is required that S is non negative. We achieve this by restricting $S - qN$ to be non negative. The restriction then implies that the maximal radix 2^k is 16, i.e. the scan factor is limited to 4.

However, we are able to increase the scan factor. The idea is to reduce the contribution of the multiple aB in (2) by choosing a larger divisor. Recall that we just want to modulo reduce the partial product, so we can choose an easily generated multiple of N, e.g. $2^r N$. This gives the following range restriction

$$\begin{aligned} 2^k S + aB - 2^k q 2^r N &\le q_{max} 2^r N \\ S - q 2^r N &\le \frac{q_{max} 2^r N - aB}{2^k} \\ S - q 2^r N &\le \frac{q_{max} 2^r - 2(2^k - 1)}{2^k} N. \end{aligned} \tag{3}$$

Since we want to generate multiples aB with the same hardware as qN, a is limited to 42 and the maximal radix is therefore 32, corresponding to a scan factor of 5. To achieve this, restriction (3) implies that r must be greater than or equal to 1.

We are now ready to present the final algorithm for modulo multiplication. Note that the final corrections can be made by iterating two extra cycles, while setting $a_i = 0$ and furthermore assuming $r = k$.

The final result is read from S discarding the $2k$ least significant bits, and belongs to the interval $[0; 2N[$. A further reduction is not necessary since, according to the stimulation conditions, we can directly start up a new multiplication in the exponentiation algorithm with inputs in the interval $[0; 2N[$. When the exponentiation algorithm terminates the result will also belong to $[0; 2N[$, here a reduction is necessary. This reduction is easily carried out while outputting the result serially. The correctness of Algorithm 4 is proven in [11]. The time complexity is:

$$T[\text{Mult}, n] = \left(\left\lceil \frac{n+1}{k} \right\rceil + 2\right) T[\text{iteration}] \tag{4}$$

In the rest of this paper we will describe how to perform the central operations of the loop, and take a closer look at the hardware architecture of the multiplication unit.

Algorithm:
Modulo multiplication.

Stimulation:
$A = a_{n'-1} a_{n'-2} \cdots a_0 a_{-1} a_{-2}$,
where $a_i \in [0; 2^k - 1]$,
$a_{-1} = a_{-2} = 0$,
$n' = \lceil \frac{n+1}{k} \rceil$;

B, where $B \in [0; 2N[$;
N, where $N \in]2^{n-1}; 2^n[$;
k, where $k \geq 3$;
r, where $r = k$.

Response:
S div $2^{2k} \equiv_N AB$ and
S div $2^{2k} \in [0; 2N[$.

Method:
$S := 0$; $i := n' - 1$;
WHILE $i \geq -2$ DO
  {$*S \equiv_N (A \text{ div } 2^{k(i+1)}) \cdot B*$}
  $q :=$ Estimate(S div $2^r N$);
  $S := 2^k S + a_i B - 2^{k+r} qN$;
  $i := i - 1$;
END;

Algorithm 4: *Modulo multiplication.*

4 Calculation of $2^k S + aB - q2^{k+r} N$

Because we are dealing with very long operands we use redundant carry save adders. This implies that the result of a multiplication is represented in *two* words. To avoid an area or time consuming carry-completing adder in the circuit we represent the multipliers and multiplicands in carry save form during the *complete* computation of an exponential. The modulus is represented in 2'complement form and in one word. We get the expression

$$2^k (S_s + S_c) + a(B_s + B_c) - q 2^{k+r} N \tag{5}$$

The multiplier digit a in radix 32 is recoded from the carry-save representation of A through two levels. The 5 digit positions of A is interpreted as three digits in radix 4: d_2, d_1, d_0, where $d_2 \in [0; 2]$ and $d_1, d_0 \in [0; 6]$. These are first recoded into digit sets $[-1; 1]$ repectively $[-1; 3]$, possibly generating and absorbing carries. Secondly these digits are recoded into $\alpha_2, \alpha_1, \alpha_0$ with $\alpha_i \in [-1; 2]$, where α_2 may absorb a carry without generating a carry out. Hence the radix 32 digit a is represented as:

$$a = 4^2 \alpha_2 + 4\alpha_1 + \alpha_0, \quad \alpha_i \in \{-1, 0, 1, 2\}, \tag{6}$$

thus aB can be computed as the sum of three shifted versions of B in a multiplexing network as shown in

Figure 5. The result is again represented in carry save form $(aB)_s$ and $(aB)_c$. The computation of $-qN$ is performed the same way, noting that $q \in [0;42]$ can be represented in radix 4 as in (6).

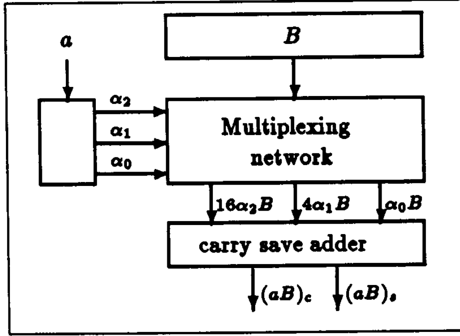

Figure 5: *Unit for generating a multiple.*

Expression (5) is now expanded to the form

$$\begin{array}{c} 2^k(S_s + S_c) \\ +((aB_s)_s + (aB_s)_c) + ((aB_c)_s + (aB_c)_c) \\ +2^{k+r}((-qN)_s + (-qN)_c) \end{array} \quad (7)$$

The cost of keeping the operands in carry save form is that we get an expression of eight terms instead of three terms. To reduce the hardware we perform the computation in a pipelined fashion, and share a single unit for generating multiples and a single 4-2 adder for summing terms. The adder is constructed of two carry save adders. In Figure 6 the timing of the computation of expression (7) is illustrated. Each row shows the activity of a hardware component by enclosing the activity in a box. In the left column the components are described. A single iteration of the loop in Algorithm 4 is computed by six cycles of the hardware. The iterations are overlapped, as illustrated by the dashed lines, resulting in a throughput of one iteration per 3 hardware cycles. (U_s, U_c), (V_s, V_c) and (S_s, S_c) denote registers in the pipeline for saving results in carry save form. The computation is performed from left to right.

5 Estimation of S div $2^r N$

Several implementations or suggestions of how to implement the quotient estimation have been presented in the past. According to [1] the estimate can be found by multiplying a few of the most significant digits of the partial remainder S and the reciprocal of the divisor $2^r N$. This assumes that the necessary amount of digits of $\frac{1}{2^r N}$ is part of input to the chip or that it is computed on the chip. The standard approach for SRT division, or as extended in [6], is to use table lookup, implemented as a PLA circuit. This method seems to be infeasible for radices as high as in this paper. We will follow the approach in [3] which is based on what we identify as a "parallel exhaustive search" for the quotient digit. In parallel we compute the sign of resulting partial remainders when the quotient digit assumes all possible values, i.e. we perform in parallel

$$S - q2^r N, \;\; q \in \{0, 1, \ldots, q_{max}\},$$

where S is in carry save form. The sign is detected as the carry out of the computation. A high carry out indicates a non negative partial remainder. Then we determine the quotient digit as the smallest value of q which results in a positive remainder. Since the sign computation involves a carry ripple we only use a few of the most significant digits of S and $-q2^r N$, and consequently the quotient digit will just be an *estimate*. The necessary number of digits is determined by the range restriction (3). In Figure 7 is shown a single cell in the quotient estimation unit for calculating the sign of $S - q2^r N$. There is one cell for each possible value of q. In the figure we denote by p the position of the most

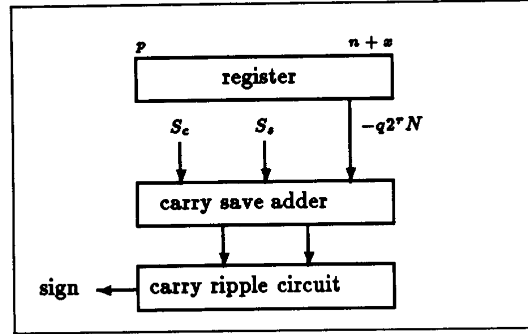

Figure 7: *A cell in the quotient estimation unit.*

significant bit, the sign bit, in S_s, S_c and in $-q2^r N$. According to restriction (3) $p = n + r + \lceil \log_2 q_{max} \rceil$ and for $q_{max} = 42$ this gives $p = n + r + k + 1$. n is the bit length of N, and $n + x$ denotes the position of the least significant bit in the quotient estimation. Since the weights of the discarded parts of S_s, S_c and $-q2^r N$ are all positive and belongs to $[0; 2^{n+x}[$, this computation gives the following range for the estimated value of q:

$$\begin{array}{rcl} S - q2^r N & = & S' + (-q2^r N)' + \epsilon \\ & \in & [0; 2^r N + 3 \cdot 2^{n+x}[\,, \end{array}$$

where S' and $(-q2^r N)'$ denotes S and $-q2^r N$ with the bits from 0 to $n + x - 1$ set to zero. Now x can be determined from the range restriction (3), where the redundant digit set $\{0, 1, \ldots, q_{max}\}$ is assumed for the multiplier:

$$\begin{array}{rcl} 2^r N + 3 \cdot 2^{n+x} & < & \frac{q_{max} 2^r - 2q_{max}}{2^k} N \\ 2^x & < & \frac{1}{6}(\frac{2^r - 2}{2^k} q_{max} - 2^r), \end{array}$$

where the smallest value 2^{n-1} of N has been inserted.

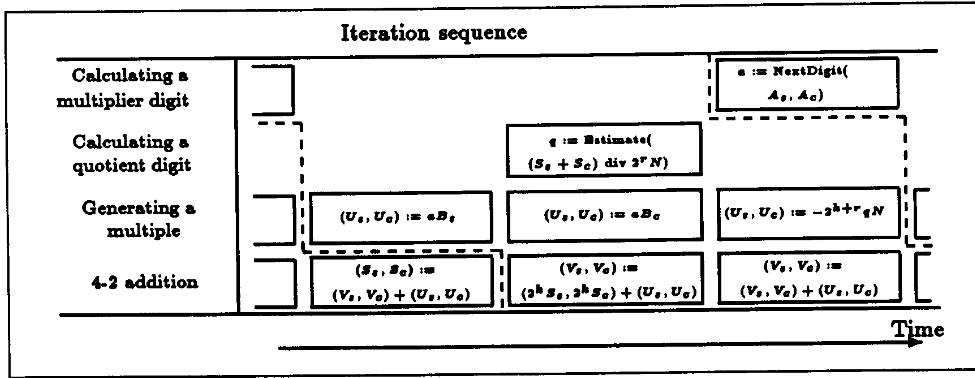

Figure 6: *Timing diagram for an iteration in the multiplication algorithm. The iterations are overlapped.*

Substituting 5 for r and k, and 42 for q_{max} we get $x \leq 0$. Thus the necessary number of bits in the quotient estimation is $p - (n + x) + 1 = 12$ in the case of radix 2^5.

This method for quotient estimation seems very area consuming but compared to the sise of the multiple generating unit and the 4-2 adder for operands longer than 500 bits this area is reasonable. As usual in VLSI design a high degree of concurrency results in faster circuits at the cost of area.

6 Performance

Equation (1) and (4) gives an expression for the computing time of an exponentiation

$$T[\text{Exp}, n] = n(\left\lceil \frac{n+1}{k} \right\rceil + 2)T[\text{iteration}].$$

As explained in Figure 6 an iteration in the multiplication loop is performed in three cycles through the hardware. Anticipating that the quotient estimation unit is the critical path in the circuit we get

$$T[\text{Exp}, n] = n(\left\lceil \frac{n+1}{k} \right\rceil + 2)3T[\text{quotient estimation}].$$

A cell of the quotient estimate unit has been designed in a 2μ CMOS process and simulations shows a delay less than 20 ns. For $n = 512$ and $k = 5$ we achieve a computing time of 3.2 ms, corresponding to a bit rate of

$$\frac{n}{T[\text{Exp}, n]} = 159 \frac{\text{Kbit}}{\text{sec}}.$$

Compared to the hardware implementation from Thorn EMI [15] with a bit rate of 29 $\frac{\text{Kbit}}{\text{sec}}$ this design improves the speed by a factor of more than 5.

The calculation on computing time assumes that we have implemented the parallel version of the exponentiation algorithm. This can be done in two ways: By replicating the multiplication unit or by pipelining a single multiplication unit. With respect to area the last approach is preferable. Observing that the two parallel multiplications have the modulus N and the multiplicand B in common we only need to add extra registers for a multiplier in carry save form and latches to implement the pipeline. An iteration in the multiplication loop now consist of six clock cycles at approximately 10ns. Clock frequencies as high as this can be hard to achieve. A way to avoid the use of a clock to synchronize the circuit is to use self-timed circuit schemes [16].

The VLSI design of the hardware components shows high regularity and the area for wiring is minimized through the use of carry save adders. At the expense of regularity and area, the speed of quotient estimation can be increased by replacing the carry ripple circuit in Figure 7 by carry look-ahead circuit.

We can obtain a rough estimate of the area by comparing the proposed architecture to the one described in [12], which has been laid out using a silicon compiler: The area is approximately 200 mm^2 in a 1.2μ CMOS process technology for a chip capable of modulo exponentiating 561 bit operands. The architecture presented here include *one* unit for generating multiples and *one* 4-2 adder where [12] include two of each and additionally a 561 bit ripple adder. The adder is used to convert the result of a multiplication from carry save form to a non redundant representation. Taking into account the extra latches for pipelining a multiplication unit we believe that the area will be less than 200 mm^2 in a 1.2μ process if we use the silicon compiler and its library cells. We can reduce the area significantly by making a full custom layout of the design, since the library cells are designed in a conservative manner using static registers and static logic gates. Another way to reduce the area (and the computing time) is by choosing a smaller process technology, e.g. 0.8μ which approximately halves the area.

All of the fastest implementations in Brickells survey [4] include more than one chip. Cryptechs 712 bit solution [7] comprise 6 chips, where each chip contains a

datapath for 120 bit. Thorn EMIs 768 bit solution [15] comprise a controller chip and 3 datapath chips for 256 bit each.

7 Summary

We have presented a way to speed up a well known exponentiation algorithm by performing two multiplications in parallel, and we have shown how these multiplications can be performed efficiently using high radices. Further more we have developed a highly regular hardware architecture, based on the redundant carry save addition technique, implementing the multiplication algorithm with a radix of 32. Simulations indicates a resulting speed improvement of more than 500% compared to other known implementations.

Currently we are working on generalising the multiplication to even higher radices. By expressing the quotient and multiplier in a symmetric redundant digit set it seems simple to modify the hardware architecture to radix 64.

References

[1] Paul Barrett. Implementing the Riverst Shamir and Adleman public key encryption system on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86*, pages 311–323, 1986.

[2] G.R. Blakely. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Trans. Computers*, C-32:497–500, 1983.

[3] Ernest F. Brickell. A fast modular multiplication algorithm with applications to two key cryptography. In D. Schaum, R.L. Riverst, and A.T. Sherman, editors, *Advances in Cryptology, Proceedings of Crypto '82*, pages 51–60, New York, 1982. Plenum Press.

[4] Ernest F. Brickell. A Survey of Hardware Implementations of RSA. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89*, pages 368–370. Springer-Verlag, 1990.

[5] W. Diffie and M.E. Hellman. New directions in cryptography. In *IEEE Trans. on Info. Theory*, volume IT-22(6), pages 644–654, Nov. 1976.

[6] Jan Fandrianto. Algorithm for high speed shared radix 8 division and radix 8 square root. In *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 68–75. IEEE, 1989.

[7] Frank Hoornaert, Marc Decroos, Joos Vandewalle, and René Govaerts. Fast RSA-Hardware: Dream or Reality ? In *Advances in Cryptology - EUROCRYPT '88*, pages 257–264, 1988.

[8] Donald E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, volume 2. Addison-Wesley, 2. edition, 1981.

[9] Hikaru Morita. A Fast Modular-multiplication Algorithm based on a Higher Radix. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89*, pages 387–399. Springer-Verlag, 1990.

[10] G.A. Orton, M.P. Roy, P.A. Scott, and L.E. Peppard. VLSI implementation of public-key encryption algorithms. In *Advances in Cryptology - CRYPTO '86*, pages 277–301, 1986.

[11] Holger Orup and Erik Svendsen. VICTOR. Forbedringer og videreudviklinger af VICTOR - en integreret kreds til understøttelse af RSA-kryptosystemer. Computer Science Department of Aarhus University - Internal report, 1990.

[12] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an Efficient RSA Hardware Implementation. In I.B. Damgård, editor, *Advances in Cryptology - EUROCRYPT '90*, pages 245–252. Springer-Verlag, 1991.

[13] Ronald L. Riverst, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, volume 21, pages 120–126, Feb. 1978.

[14] Norman R. Scott. *Computer Number Systems & Arithmetic*. Prentice-Hall, 1985.

[15] THORN EMI. RSA Evaluation Board. Technical Report 10, Thorn EMI Central Reasearch Laboratories, 1988.

[16] T.E. Williams, M. Horowitz, R.L. Alverson, and T.S Yang. A self-timed chip for division. In Paul Losleben, editor, *Advanced Research in VLSI. Proceedings of the 1987 Stanford Conference*, pages 75–95. The MIT Press, 1987.