Exponentiation using Division Chains

Colin D. Walter, *Member, IEEE*

Computation Department, U.M.I.S.T.,
PO Box 88, Sackville Street, Manchester M60 1QD, U.K.
`C.Walter@umist.ac.uk`   `http://www.co.umist.ac.uk/`

Abstract

Exponentiation may be performed faster than the traditional square and multiply method by iteratively reducing the exponent modulo numbers which, as exponents themselves, require few multiplications. This mainly includes those with few non-zero bits. For a suitable choice of such divisors, the resulting mixed basis representation of the exponent reduces the expected number of non-squaring multiplications by over half at the cost of a single extra register. Pre-processing effort depends entirely on the exponent and can be kept down to the work saved in a single exponentiation. Moreover, no pre-computed look-up tables are required, so the method is especially applicable where space is at a premium. In particular, it out performs the instance of the m-ary method which uses the same space. However, for 512-bit exponents, it beats every instance of the m-ary method, achieving well under 635 multiplications on average. Both hardware and software implementations of the RSA crypto-system can benefit from this algorithm.

Key Words Modular exponentiation, bit recoding, RSA cryptosystem, addition chains, *m*-ary method, mixed basis arithmetic, radix representation.

1 Introduction

Fast exponentiation is becoming increasingly important with the widening use of encryption. Whereas the most startling improvements in speed are achieved through the use of dedicated hardware for multiplication, some small gains can also be made through a good choice of algorithm for organising the order of multiplications at either the hardware or software level.

The expected number of multiplications (including squarings) to compute A^e in the traditional way, using square and multiply, is approximately $\frac{3}{2}\log_2 e$ [14], [15]. The pre-calculation or intermediate calculation of auxiliary powers of *A* enables some unnecessary repetition of work to be avoided. There are several methods described below for reducing the average number of operations to about $\frac{4}{3}\log_2 e$ using storage for only one or two extra powers of *A*, but the variance of the distribution is high: there is a very real chance of still requiring $\frac{3}{2}\log_2 e$ or even more operations. With more storage, the coefficient $\frac{4}{3}$ can be further reduced [22], but the lower limit is still above 1 [11], [14] since $\lfloor \log_2 e \rfloor$ squarings are certainly necessary to generate a number as large as A^e. However, fast registers are normally very limited in number, so the use of too much more storage is actually likely to *retard* the exponentiation: the cost of communication with slower memory may easily outweigh any minimal reduction in the coefficient.

This paper describes another technique which uses only *one* register more than the minimum for the standard square and multiply method. It achieves an average coefficient below $\frac{5}{4}$, and is *consistently* close to its average. Some pre-computation is required, but only enough to establish a suitable sequence of squarings and multiplications. Thus the pre-computation need not be repeated in a sequence of exponentiations using the same exponent. A simple version of the technique achieves $\frac{5}{4}$ with a pre-computation effort equivalent to only a few multiplications of integers the size of the exponent. So the overall gain is small for a single exponentiation, but when the same key is used repeatedly as in the RSA cryptosystem, the pre-computation cost can be ignored and the full reduction claimed. The method is very adaptable, with better results easily obtained from greater effort or from allowing more register space. It also works equally well for any group, so that the speed-ups work as well for an elliptic curve group as for an integer residue class group.

2 Notation and Literature Review

On a sequential machine, any exponentiation by e can be described by an *addition chain* $a_0, a_1, a_2, a_3, ..., a_n$, where $a_0 = 1$, $a_n = e$ and, for each $i > 0$, $a_i = a_j + a_k$ for some $j, k < i$ [14]. The ith multiplication performed is $A^{a_i} = A^{a_j} \times A^{a_k}$ and exponentiation by e takes n multiplications. Storage requirements for any chain can be worked out easily from the sequence itself, although if there is a choice of j and k for any i then the minimum storage might not be clear. For given small exponents e, the minimal number of multiplications can be found by a search of all addition chains for e. As an NP-hard problem [8], however, such a search is impractical for the typical decryption keys e found in RSA cryptography. Nevertheless, here we present a technique which is easily tailored to the available time and space resources and ultimately would yield a best addition chain.

Suppose $e = \sum_{j=0}^{n} r_j 2^j$ is the binary representation of e. The standard method of square and multiply can be performed by scanning the bits of e in either direction. First, a Horner-style evaluation

$$A^e = ((\ldots((A^{r_n})^2 A^{r_{n-1}})^2 \ldots)^2 A^{r_1})^2 A^{r_0}$$

corresponds to an addition chain with $a_{i+1} = 2a_i$ (square) or $a_{i+1} = a_i + a_0$ (multiply). This requires just 2 storage registers, containing $A = A^{a_0}$ and the partial result A^{a_i} respectively. Alternatively, and dually, rather than squaring the partial result and multiplying in A as required, A can be repeatedly squared and the resulting power multiplied into the partial result when needed:

$$A^e = (A^{2^0})^{r_0} (A^{2^1})^{r_1} (A^{2^2})^{r_2} \ldots (A^{2^n})^{r_n}$$

Here the first of the two registers now contains A^{2^i} for $i = 0,1,..., n$. For natural number arithmetic this requires larger registers than Horner's method, but there is no difference for finite rings, such as the residues of RSA cryptosystems, or for real approximations. The number of multiplications, excluding squarings, is one less than the Hamming weight of e, i.e. one less than the number of non-zero bits in e; on average $\lfloor \log_2 e \rfloor / 2$.

By expressing the exponent using the radix m instead of 2 and pre-computing the powers A^i for $i = 1, 2, ..., m-1$ we obtain the m-ary method [14]. This follows the Horner style evaluation above, requiring repeatedly raising the partial result to the mth power and then multiplying by an A^i. It is usually convenient to pick m as a power of 2. Then the number of squarings is roughly the same as before, but the number of other multiplications excluding pre-computations reduces to $\lfloor \log_m e \rfloor (m-1)/m$ on average. This is good for larger m, but storage requirements quickly become prohibitive for RSA applications. Moreover, the hardware to select one of m registers on each iteration has a critical path length of order $O(\log_2 m)$. Thus, without considerable care, the complexity of the non-squaring multiplications for this method might still be essentially $O(\log_2 e)$.

Half the memory can be saved by pre-computing only A^i for odd $i < m$ [22]. The exponent is recoded as

$$e = r_0 + m_0(r_1 + m_1(r_2 + m_2(\ldots(r_{n-1} + m_{n-1} r_n)\ldots))) \quad (1)$$

where $m_i = m$ is chosen whenever it makes r_i odd, and otherwise $m_i = 2$ is chosen with $r_i = 0$. Then A^e is evaluated starting at the innermost bracketed expression:

$$A^e = ((\ldots((A^{r_n})^{m_{n-1}} A^{r_{n-1}})^{m_{n-2}} \ldots)^{m_1} A^{r_1})^{m_0} A^{r_0} \quad (2)$$

This requires $m/2$ registers besides the partial result, about $\lfloor \log_2 e \rfloor - \frac{1}{2}\log_2 m + 1$ squarings if m is a power of 2, and about $\lfloor \log_2 e \rfloor / (\log_2 m + 1)$ other multiplications on average besides any used for the pre-computations. (For $i < n$ there are on average as many cases of $m_i = m$ as $m_i = 2$.) Taking $m = 2$ gives the binary method above with coefficient $\frac{3}{2}$ for $\log_2 e$ in the total number of multiplications. Taking $m = 4$ means a third register, which holds A^3, and this reduces the coefficient to $\frac{4}{3}$.

If the inverse A^{-1} exists and can be calculated cheaply, then any sequence of 1s can be replaced by the sequence $10...0\bar{1}$ as follows. Starting from the least significant bit of e, whenever two adjacent 1s are encountered with no carry from lower down, the lower 1 is replaced by $\bar{1}$ and a carry of 1 generated to the upper 1. Similarly, for a carry of 1 into 10, the lower 0 is replaced by $\bar{1}$ with a carry up to the 1. Otherwise carries propagate up as usual. On average the carry is 1 in 50% of cases, 2/3rds of these are into a following 1. Similarly, a carry of 0 is to a digit 0 in 2/3rds of cases. Thus, after this modified Booth recoding, a 0 digit occurs on average in 2/3rds of all cases. So again about $\frac{4}{3}\log_2 e$ squarings or multiplications by A or A^{-1} are required, with one multiplication for each non-zero digit [26]. Storage is 3 registers, one for each of A, A^{-1} and the partial result. This method can be combined with the m-ary method [9], [10], [15]. Further similar methods include [12].

The m-ary method has been developed in several important ways. One of these uses the ideas of data compression: common subsequences of bits are identified and the corresponding powers of A computed once and stored for use each time the bit sequence is encountered. In effect, this windowing technique corresponds to allowing the m_i in (1) to be any power of 2 [1], [2], [16]. Another method employs *vector* addition chains so that work involved in computing several different powers of A can be shared wherever possible [2],[18],[19],[20],[23]. Similar sharing is also considered in [17], [24], [25].

In many of these methods, the trade-off between space and time is poor when e is static and A variable. In hardware implementations, a speed-up of less than two-fold, as here, should require less than twice the area. Since the total hardware requirement for a digit parallel modular multiplier is equivalent to only a small number of registers [21], few of these techniques are likely to be of much value to hardware designers. However, we will produce more than half of the maximum possible savings by using only 1 more register than for the square and multiply method.

3 Mixed Basis Representations.

In a *mixed basis* number representation, an exponent e is written as a digit linear combination

$$e = r_0b_0 + r_1b_1 + \ldots + r_nb_n$$

of basis elements b_i. Normally, b_i is a multiple of b_{i-1}, with $b_i = m^i$ in a standard radix m representation. The divisibility property enables the nested representation of e necessary in the m-ary method. In the variation where half the memory is saved, the basis elements are $b_i = m_0m_1\ldots m_{i-1} = b_{i-1}m_{i-1}$ with $m_{i-1} = 2$ or m. Dimitrov *et al* [6] consider the particular case of this representation with $m = 3$, illustrating the value of m not being a power of 2. There the digits r_i are 0 or 1 and basis elements b_i are of the form 2^j3^k. This leads to $1.42361\times\log_2 e$ multiplications on average. They suggest that each m_i could be any prime up to some bound m. Then the powers A^i could be pre-computed for $i < m$ and A^e calculated following the m-ary method, that is, with Horner style evaluation. In fact, the algorithm presented in [6] for $m = 3$ processes the digits in the opposite order; the cost is the same for that case. If many of the possible b_ith powers of A are precomputed, then this latter order can lead to considerable time savings when a number of exponentiations with fixed A and variable e are involved [4], [7]. However, this is a different problem and the space cost may be large.

Here we retain the key divisibility property of the basis elements, namely $b_i = b_{i-1}m_{i-1}$, and develop algorithms for making good choices of the m_i. The divisibility property enables the digits to be processed in either order. Where space is expensive, using Horner's order of evaluation without precomputed powers leads to duplication of effort when digits are equal since then the r_ith power of A must be re-calculated. We concentrate on scanning the digits in the reverse order: the b_ith power of A will be computed dynamically from the b_{i-1}st and held in a single register, just as the 2^ith power is held in the corresponding binary method. At the same time, the $r_{i-1}b_{i-1}$st power will be formed and multiplied into the partial result.

4 The Division Chain Method

The new means of reducing the number of multiplications used in exponentiation arises from the iterative application of a decomposition $e = me' + r$ where r is usually the least non-negative residue of e modulo m. At each repetition the divisor m is selected by reference to a pre-determined set of inexpensive pairs (m,r) and the powers A^m and A^r are computed. Since A^e satisfies the relationship

$$A^e = (A^m)^{e'}A^r \qquad (3)$$

it suffices to multiply A^r into a partial product register and re-apply the process to the remaining problem of raising A^m to the power e'. Including all residues for some divisor such as 2, 3 or 5 guarantees each step is possible and that termination occurs. When $e' = 0$ has been processed the partial product register contains the required output. So, if m_0, m_1, ..., m_n is the list of divisors which this generates, and r_0, r_1, ..., r_n are the associated remainders then, as in (2),

$$A^e = A^{r_0+m_0(r_1+m_1(\ldots+m_{n-1}(r_{n-1}+m_{n-1}r_n)\ldots))}$$

and evaluation is performed by processing the exponent expression from left to right. Note that the divisors for the algorithm can be chosen on the fly as the exponentiation is performed or in advance whenever the exponent is known.

We assume that it is known how best to calculate A^m and A^r. So divisor / residue pairs (m,r) must be selected from a set for which this information is known. Then at each step the cheapest such decomposition can be chosen from this fixed set of pairs. A sequence of pairs (m,r) used to direct an exponentiation in this way will be called a *division chain* by analogy with the addition chain description. Although it is usually the case, we see later that the residue need not always be the least non-negative one. However, if no alternative choice of residue is ever permitted, then the sequence of divisors suffices to determine the division chain.

When $m = 2$ at each step, this technique is just the standard square and multiply method described above. If $r = 0$ at each step it becomes the *factor* method [14]. So the division chain method here generalises both of these. Moreover, if $m_i = m$ always, it is the dual of the m-ary method, processing digits in the opposite order. Bos and Coster [2] provide some heuristics for their MakeSequence method of constructing addition chains. This method shows how these can be realised.

If $F(e)$ is the minimum number of multiplications used to compute A^e by any method, then the decomposition (3) shows $F(e) \leq F(e')+F(m)+F(r)+1$. Inequality will arise if either r is 0 or some multiplications which are used to form A^m may also be used in constructing A^r. However, $F(e) \approx F(e')+F(m)$ because $e \approx me'$ and $F(x)$ is always of order $\log(x)$. So the main step of the algorithm can only be useful if A^m and A^r are both cheap to compute, with much of this work in common. In particular, this is the case when $m = 2^n+1$ and $r = 0$, $2^{n-1}+1$ or 2^i for some $i \leq n$. Then the formation of A^r is a by-product of computing A^m. In general, useful pairs (m,r) normally have the property that r lies in an addition chain for m of minimal length or, at worst, r is the sum of two or three members of such a chain. Consequently, if $f(e)$ is the number of multiplications this method yields, and there are sufficient such pairs (m,r), then we will have $f(e) \approx f(e')+f(m)$ at each step and can expect to obtain a reasonably efficient scheme for exponentiation.

With suitable restrictions on the divisors m and residues r, only three storage registers are needed, although more can be used. One register holds the partial result which is the product of all the A^r from previous steps. The other two are used to form A^m using an addition chain that requires only two values to be kept. When the components required for A^r are formed, they are multiplied into the partial result register and so do not interfere with the calculation of A^m.

As an illustration, we show how to compute A^{349} starting with the divisor 17. First, $A^{349} = (A^{17})^{20}A^9$ requires computing $B = A^{17}$ and A^9. These are found in 5 multiplications using the addition chain (1, 2, 4, 8, 9, 17), and A^9 is placed in the partial product register. This leaves B^{20} to be calculated and multiplied into the accumulating product. Using the divisor 4 next, $B^{20} = (B^4)^5$. We obtain $C = B^4$ with 2 squarings and must then compute C^5 for multiplying into the partial product. Using the divisor 4 again, $C^5 = (C^4)^1C^1$. The C^1 is multiplied into the partial product, then C^4 is computed with two squarings and the result finally multiplied into the partial product to yield A^{349}. So exponentiation by 349 can be done this way with 11 multiplications rather than the 13 required by the binary and A, A^3 methods.

5 Calculation of the Coefficient

We will demonstrate the value of the method with a small set of divisors. Suppose the strategy is as follows:

```
If      e ≡ 0 mod 2         then m = 2
elseif  e ≡ 0 mod 3         then m = 3
elseif  e ≡ 1,2,5,8 mod 9   then m = 9
else   {e ≡ 4,7 mod 9}      then m = 3
```

with r chosen as the least non-negative residue modulo m. Let $f(e) = c\log_2 e$ be the average number of squarings plus multiplications expected for a random exponent e. Then $f(e)$ can be expressed fairly accurately as a sum of terms $p\times(v+f(e/m))$, one for each case (m,r), where p is the probability of the case arising, m is the associated divisor, and v is the number of multiplications required to form A^m and A^r and multiply A^r into the result. For the case $e \equiv 5 \bmod 9$, we can use the addition chain (1, 2, 4, 5, 9) to show that 4 multiplications yield both A^m and A^r, giving $v = 5$. The other values of v are clear.

e mod 18	m	Rel freq	Distrib of new e mod 6					
			0	1	2	3	4	5
0,6,12	2	4	2	-	-	2	-	-
2,8,14	2	10	-	5	-	-	5	-
4,10,16	2	10	-	-	5	-	-	5
3	3	6	-	2	-	-	-	-
9	3		-	-	-	2	-	-
15	3		-	-	-	-	-	2
1	9	9	1	-	1	-	1	-
7	3		-	-	3	-	-	-
13	3		-	-	-	-	3	-
5	9	9	1	-	1	-	1	-
11	9		-	1	-	1	-	1
17	9		-	1	-	1	-	1
Freq sums: 48			4	9	10	6	10	9

Table 1. Frequency counts for the Markov process.

The values of the probabilities p are less obvious since application of the method causes the residue classes to be no longer uniformly distributed. Action depends entirely on the residue of e modulo the lowest common multiple of the divisors, namely 18, and not on any previous exponent. The sequence of residues for successive exponents forms a Markov chain for which the stochastic transition matrix $P = (p_{ij})$ is easy to construct: p_{ij} is the probability of obtaining the new exponent $e' \equiv j \bmod 18$ from an exponent $e \equiv i \bmod 18$. For example, $e = 18k+7$ uses $m = 3$, yielding $e' = 6k+2 \equiv 2$, 8 or 14 mod 18. So $p_{7,2} = p_{7,8} = p_{7,14} = 1/3$. The probabilities p_i of exponents which are i mod 18 eventually stabilise: the row vector $\boldsymbol{p} = (p_i)$ satisfies $\boldsymbol{p} = \boldsymbol{p}P$ and can be calculated easily. It turns out that p_i only depends on i mod 6, with $p_0 = p_6 = p_{12} = 1/36$, etc. The relative frequencies of the 18 classes are given in Table 1, which can readily be used to check that these do indeed yield the equilibrium values.

So the 4 cases of the algorithm have probabilities 1/2, 1/8, 1/4 and 1/8 respectively. In consequence,

$$f(e) = (1/2)(1+f(e/2)) + (1/8)(2+f(e/3)) + (1/4)(5+f(e/9)) + (1/8)(3+f(e/3))$$

Then using $f(e/m) \approx f(e)-f(m) \approx f(e)-c\log_2 m$ we obtain

$$0 = (1/2)(1-c\log_2 2) + (1/8)(2-c\log_2 3) + (1/4)(5-2c\log_2 3) + (1/8)(3-c\log_2 3)$$

So $c \approx 1.4064$ (cf. [6] with $c \approx 1.42361$ for a slightly simpler example).

Better values for the coefficient c arise from the use of more divisors. Consider:

```
If      e ≡ 0 mod 33      then m = 33
elseif  e ≡ 0 mod 17      then m = 17
elseif  e ≡ 0 mod 2       then m =  2
elseif  e ≡ 0 mod 5       then m =  5
elseif  e ≡ 0 mod 3       then m =  3
elseif  e ≡ 1,2,4,8,16,17,32 mod 33
                      then m = 33
elseif  e ≡ 1,2,4,8,9,16 mod 17
                      then m = 17
else case e mod 90 of
 1,17,77,29,43,47,67,83       : m = 3
 7,11,13,31,41,49,61,71,79,89: m = 2
 19,23,37,53,59,73           : m = 9
end.
```

Again, choose r minimally. The more complex final case is engineered to enable useful factors such as 2, 3, 5 or 9 to be picked up on the next iteration. As before the residue classes modulo 90×11×17 are not equi-probable. The transition matrix P has grown to around 2^{28} entries and made the direct solution of $\boldsymbol{p} = \boldsymbol{p}P$ infeasible. However, if $\boldsymbol{i}$ is the suitably scaled all 1s row vector then $\boldsymbol{p} = \lim_{n\to\infty} \boldsymbol{i}P^n$. In practice convergence is fairly rapid. The Euclidean distance between successive values of $\boldsymbol{p}$ is almost halved at each iteration, and so another decimal place of c is established on every fourth iteration. After just 7 iterations we obtain the correctly rounded $c = 1.3566$. When the divisor 65 is included in a similar fashion to 33, c is reduced to 1.343.

The time complexity of finding $\boldsymbol{p}$ is driven by the number of non-zero entries in P. The matrix has size $l \times l$ where l is the lowest common multiple of the moduli used, and it has not far short of ln non-zero entries where n is the number of different divisor/residue pairs considered. Thus, including more than a few small divisors makes the vector $\boldsymbol{p}$ too big and the computation too time consuming for a direct solution, and it forces a statistical approach to estimating c.

To achieve a coefficient noticeably less than the $\frac{4}{3}$ of the A, A^3 method, take the following divisors:

```
2, 3, 5, 17, 33, 49, 65, 97
    129, 257, 513, 1025
```

and allow residues which are either 0, or are in a minimal addition chain which uses just two memory fields, or are the sum of two residues in such a chain. (Details are contained in the Appendix.) At each iteration simply choose the divisor m for which the ratio $v/\log_2 m$ is least, where v is the number of multiplications associated with the residue which occurs. Then, on average, random 512-bit integers require 672.15 multiplications – almost 1.5% fewer than the average of 682 for the A, A^3 method.

How much work is involved in choosing the division chain for a given exponent and a given such strategy? A small, variable but bounded number of modular divisions are performed on the exponent at each step to decide the next divisor. Each of these divisions requires at most a small constant times more work to perform than the division which is actually chosen. Thus choosing the divisor m at each step requires on average a constant, say c', times more work than simply dividing the remaining expon