Exponentiation using Division Chains

Colin D. Walter

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

cdw@sna.co.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 themselves require few multiplications, such as those with few non-zero bits. For a suitable choice of such divisors, this reduces the expected number of non-squaring multiplications by over half at the cost of a single extra register. The method is applicable to exponentiation in any multiplicative group where squaring is as expensive as multiplication and not cheaper than integer division. In particular, both hardware and software implementations of the RSA crypto-system can benefit.

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 at the software level.

The expected number of multiplications (including squarings) to compute A^e in the traditional way, using square and multiply, is approximately $\frac{3}{2}\log_2 e$. The pre-calculation or intermediate calculation of auxiliary powers of A enables some unnecessary repetition of work to be avoided. There are several methods 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 4/3 can be further reduced [6] but the lower limit is still above 1 [4] since $\lfloor\log_2 e\rfloor$ squarings are certainly necessary to generate a number as big as A^e. Thus, as fast registers are normally very limited in number, the use of 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, achieves an average coefficient below 5/4, and is consistently close to its average. Some pre-computation is required to establish a suitable sequence of squarings and multiplications, with better results obtained from greater effort. A simple version of the technique achieves 5/4 with a pre-computation effort equivalent to only a few multiplications of integers the size of the exponent.

2 Notation and Literature Review

On a sequential machine, any exponentiation by e can be described by an *addition chain* $a_0, a_1, a_2, a_3, \ldots, a_n$, where $a_0 = 1$, $a_n = e$ and, for each $i > 0$, $a_i = a_j + a_k$ for some $j, k < i$ [4]. 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 [1], however, such a search is impractical for the typical decryption keys e found in RSA cryptography.

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

$$A^e = ((((A^{e_n})^2 A^{e_{n-1}})^2 \ldots)^2 A^{e_1})^2 A^{e_0}$$

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, 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:

$$A^e = (A^{2^0})^{e_0}(A^{2^1})^{e_1}(A^{2^2})^{e_2}\ldots(A^{2^n})^{e_n}$$

Here the first of the two registers now contains A^{2^i} for $i = 0,1,\ldots, n$. For natural number arithmetic this requires larger registers than Horner's method, but there is no difference for finite rings or 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 and pre-computing the powers A^i for $i = 1, 2, \ldots, m-1$ we obtain the m-ary method [4]. This follows the Horner style evaluation above, requiring repeated raising to the mth power and 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 become prohibitive very quickly.

Half the memory can be saved by pre-computing A^i for odd i only [6]. The exponent is recoded as $e = e_0 + m_0(e_1 + m_1(e_2 + m_2(\ldots(e_{n-1} + m_{n-1}e_n)\ldots)))$ where $m_i = m$ is chosen whenever it makes e_i odd, and otherwise $m_i = 2$ is chosen with $e_i = 0$. Then

$$A^e = ((((A^{e_n})^{m_{n-1}} A^{e_{n-1}})^{m_{n-2}} \ldots)^{m_1} A^{e_1})^{m_0} A^{e_0}$$

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 3/2. Taking $m = 4$ means a third register, which holds A^3, and this reduces the coefficient to 4/3.

If the inverse A^{-1} exists and can be calculated cheaply, then any sequence of 1s can be replaced by the sequence $10\ldots0\bar{1}$. 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 $\frac{4}{3}\log_2 e$ multiplications by A or A^{-1} are required [7]. 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 (see [2], [5]). Further similar methods include [3].

3 The Division Chain Method

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

$$A^e = (A^d)^{e'} A^r$$

it suffices to multiply A^r into a partial product register and re-apply the process to the remaining problem of raising A^d 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.

We assume that it is known how best to calculate A^d and A^r. So divisor / residue pairs (d,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 (d,r) used to direct an exponentiation 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 choice of residue is ever permitted, then the sequence of divisors suffices to determine the division chain.

When $d = 2$ 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 of [4]. So the division chain method here generalises both of these.

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

With suitable restrictions on the divisors d 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^d 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^d.

As an illustration, we show how to compute A^{349} starting with the divisor 17. First, $A^{349} = (A^{17})^{20}A^9$ where $B = A^{17}$ and A^9 are computed 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.

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

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 $p\times(m+f(e/d))$, one for each case (d,r), where p is the probability of the case arising, d is the associated divisor, and m is the number of multiplications required to form A^d and A^r and multiply A^r into the result. For $e \equiv 5 \bmod 9$, we can use the addition chain (1, 2, 4, 5, 9) so that 4 multiplications yield both A^d and A^r, giving $m = 5$. The other values of m are clear.

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, $e = 18k+7$ requires $d = 3$, 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 $p = (p_i)$ satisfies $p = pP$ 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 the following table, which can readily be used to check that these yield the equilibrium values.

e mod 18	d	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		-	2	-	-	-	-
9	3	6	-	-	-	2	-	-
15	3		-	-	-	-	-	2
1	9		1	-	1	-	1	-
7	3	9	-	-	3	-	-	-
13	3		-	-	-	-	3	-
5	9		1	-	1	-	1	-
11	9	9	-	1	-	1	-	1
17	9		-	1	-	1	-	1
Freq sums:		48	4	9	10	6	10	9

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 $f(e/d) \approx f(e) - f(d) \approx f(e) - c\log_2 d$ 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 $c \approx 1.4064$.

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

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

The 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 90x11x17 are not equi-probable. The transition matrix P has grown to around 2^{28} entries and made the direct solution of $p = pP$ infeasible. However, if i is the suitably scaled all 1s row vector then $p = \lim_{n\to\infty} iP^n$. In practice convergence is fairly rapid. The Euclidean distance between successive values of 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 p is driven by the number of non-zero entries in P. The matrix has size lxl 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, adding more than a few small divisors makes the vector 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 less than the 4/3 of the A, A^3 method, take the following divisors:

```
    2, 3, 5, 9, 17, 33, 49, 65,
      97, 127, 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. At each iteration simply choose the divisor d for which the ratio $m/\log_2 d$ is least, where m is the number of multiplications associated with the residue which occurs. Then, on average, random 512-bit integers require under 670 multiplications – almost 2% fewer than the A, A^3 method.

In each of these strategies, there is a constant ratio between the average number of divisions of the exponent by a divisor and the average factor by which the exponent is decreased at each step. So generating the division chain has the same time complexity as forming the product of two numbers with the size of the exponent. If squaring the base number requires a comparable effort (as in the RSA cryptosystem) then the effort to create the division chain is equivalent to a constant number of squarings. This is therefore cheaper than the A, A^3 method as long as the exponent is large enough.

5 Choice and Ordering of Divisors

From the last section the formula for c is

$$c = \frac{\sum_i p_i m_i}{\sum_i p_i \log_2 d_i}$$

where the sum extends over the cases i which occur with probability p_i, have divisor d_i and require m_i multiplications. Thus, if the relative probabilities of these cases remain essentially unchanged, it is worth adding any new pair (d,r) with a ratio $m/\log_2 d$ which is better than the current value of c. In particular, in addition to using numbers with the form 2^n+1, we might also consider divisors d of Hamming weight 3 (i.e. 3 non-zero bits) with residues r which are 0 or are generated en route to d. Then a minimal addition chain using just two registers will require $n+2$ multiplications where the highest power of 2 in d has order n. Useful examples therefore include 49 and 97, both with ratios under 5/4 for these residues.

It is a matter of only a few minutes computing to generate all optimal and near optimal addition chains for all potentially useful divisors up to 10 or so bits in length, say, and hence establish the smallest number of extra additions which will generate each residue. Here we should relax the apparent restriction $r < d$ since, to minimise multiplications, it may be necessary to go beyond the least non-negative residue. Thus 11 mod 13 is only obtained with a minimal number of multiplications within the suggested memory restrictions by allowing $r = 12+12$. However, if both r and $r+d$ are equally cheap to compute, it may make sense to select the one which makes the next exponent e' even so that the beneficial (2,0) could then be applied.

Using all d up to over $3\text{x}2^{10}$, optimal division chains were constructed for e up to over 2^{20}. The frequencies of pairs (d,r) were recorded, and this confirmed the relative uselessness of pairs with a poor ratio $m/\log_2 d$ unless d was small. It also showed that composite d in this range are often superfluous since it frequently requires no more multiplications to use their factors as divisors instead.

The obvious order in which to arrange the divisor/residue pairs is in increasing cost per bit, i.e. following the order of the ratios $m/\log_2 d$. However, larger divisors have a longer lasting effect than smaller ones, and so they are better (resp. worse) choices if they have similar but above (resp. below) average ratios. To be more precise, if d is applied to the exponent e, then we can expect $m + c\log_2(e/d) = m - c\log_2(d) + c\log_2(e)$ multiplications on average. This is minimised if a divisor is chosen for which $m - c\log_2 d$ is minimal. Thus, the best order for selecting pairs (d,r) should be close to that determined by the values of $m - c\log_2 d$. Indeed, this explains the order in the second example of Section 4 where (2,0) is not the best first choice. In situations such as the last case of that example, any divisor that will be picked up automatically for the next iteration needs to be taken into account when assessing relative merits.

6 Large Exponents

The well-known algorithms described in Section 2, such as square and multiply, all prescribe a single course of action, as do the strategies described here so far for division chains. In all cases the high variance means frequent poor results. However, for division chains considerable choice is possible. Fixing a specific order for trying divisors usually yields a sub-optimal method. A better coefficient c is obtained when the best chain is selected from all those generated by extending partially constructed chains using every possible divisor. This also reduces the variance and hence gives a good value more reliably. Unfortunately, it is not feasible for a large exponent – there are too many combinations to evaluate all possibilities. The search space must be reduced.

Suppose the division chain (d_1,r_1), (d_2,r_2), (d_3,r_3), ..., (d_k,r_k) reduces the exponentiation problem from the power e to the power e'. Then $e = r + de'$ where $d = d_1d_2...d_k$ and (normally) $r < d$. The number of multiplications m associated with the chain is the sum of the numbers m_i associated with each pair (d_i,r_i), namely the number of multiplications required to form the d_i and r_ith powers and multiply the r_ith power into the existing partial product. If e' is still large compared to d then e is essentially reduced by a factor d for the cost of m multiplications. Thus, given a number of such chains, the best one to choose is the one for which $m/\log_2 d$ or, better, $m - c\log_2 d$ is minimal. This process can be repeated to reduce e' in its turn. Eventually e' becomes small enough for the value of r to affect the choice of chain noticeably (say $e' < d$), and then a table might be used to complete the division chain optimally.

This yields an algorithm for large exponents. Fix a suitable length k for the division chain segments to be considered. The larger the choice of k, the longer the algorithm takes to complete, but the better the result. Now repeatedly perform the following:

i) generate all reasonably priced chains of k divisors to reduce the current value of the exponent;
ii) select the cheapest one under the chosen criterion;
iii) apply this sub-chain to reduce the exponent.

The iterative procedure should terminate when the current value of the exponent becomes less than the upper limit of a pre-calculated table of optimal chains for small exponents. Of course, if in the last few iterations k divisors were to reduce the exponent to close to 0 so that the costing criterion becomes inaccurate, then the offending chains can be curtailed earlier, say at the point where the exponent falls into the range of the table. Finally, with the exponent in the range of the table, the best way of completing the chain can be looked up.

To cost this, suppose S is the sum over all divisors of the probability of the divisor being useable to extend a division chain. Assuming a close to uniform distribution on residues of exponents modulo the lowest common multiple of the divisors, S is approximately $\sum_i n_i / d_i$ where n_i is the number of residues associated with the divisor d_i. Thus S will be at most the number of divisors being used. To extend any division chain by one pair, there are roughly S possible choices (assuming successive choices are mostly independent). Suppose that L is the result of averaging the logarithms of the divisors used in an optimal sub-chain, weighted according to their frequencies, i.e. the average number of bits by which a divisor in such a chain reduces the exponent. Assume finally that T is the logarithm of the maximum exponent in the look-up table. Then, for exponents of N bits, the above algorithm will require about $(N-T)/Lk$ iterations, each of which generates about S^k sub-chains. So the work involved is roughly proportional to

$$S^k(N-T)/Lk$$

times the effort required to perform a single division of the exponent by a divisor. This shows that very large exponents are not much more difficult to deal with than smaller ones; the work is proportional to the square of the number of bits.

For good performance, k must not be too small since then the sub-chains would not be particularly good on average. Hence the best savings in time are made through a sensible choice of acceptable pairs, thereby reducing S. An obvious choice is the set of pairs used most frequently over a large range of optimal chains. Furthermore, picking large k may be a waste of effort since the average reduction in numbers of multiplications declines exponentially with k (see Table 2 below).

Many variations in the algorithm are clearly possible. For example, the subchains could be bounded by limiting the divisor product rather than by k, or k might be varied when there is no good subchain of a specific length at some point. Also, only the initial few pairs of optimal sub-chains might be selected at each iteration and the test for optimality (which contains an approximation to c) might be varied dynamically. If a good chain is still not obtained, choosing a different first divisor or using all initial subchains is also possible.

7 Test Results

We next consider test results from a specific implementation[†] of the algorithm in order to establish that a coefficient of 5/4 is generally obtainable for exponents of the size used in, say, the RSA cryptosystem. None of the improvements suggested in the last paragraph were included for the tabulated results.

First, for divisors d up to 2^8 the minimal addition chains using only two registers were generated and a table was constructed of optimal division chains for exponents up to over 2^{20}. In any subrange the method was found to use fewer than $\frac{5}{4}\log_2 e$ multiplications and squarings on average. In particular, between 2^{15} and 2^{16}, apart from a few short chains such as those for 2^{15} itself and $2^{15}+2^a$ with $a < 15$, almost 90% require 19 or 20 multiplications, under 1.5% require 21, and none require more. Hence the variation in numbers of multiplications is very small, unlike for the binary or A, A^3 methods. Essentially no exponents require too many multiplications, but there is an increasing tail of exponents which can be dealt with using fewer multiplications than expected, and it is this that makes searches for better division chains worthwhile.

Next, over ranges of at least 2^{17} exponents of order above 2^{20} and divisors up to 3×2^9, the average value of $m/\log_2 e$ was calculated for optimal chains and found to be about 2% under 5/4. Indeed, successive ranges show this ratio has a tendency to decrease as the exponents increase. This is because once the reduction operations have made the exponent less than the maximum divisor, the choice of future divisors becomes progressively more limited. This leads to a poorer ratio for smaller exponents.

The coefficient for the whole of a large exponent is essentially of the form $\left(\sum_i m_i\right)/\left(\sum_i \log_2 d_i\right)$ where each $m_i/\log_2 d_i$ comes from an optimal subchain. From the above experimental data each such term should be less than 5/4 on average since each d_i will fall within the range investigated (unless k is large). So the expected coefficient ought to be better than 5/4. Although a given sub-chain being optimal at one step undoubtedly confers some properties on the next exponent, the division process seems to minimise most of these effects. So the relevant multiplicative properties of successive exponents appear to be mostly independent for successive steps of the algorithm. Thus, individual steps resulting in a large number of multiplications should not affect the rest of the algorithm. Indeed, for larger exponents which require a greater number of steps, the variance in the distribution of the number of multiplications should be smaller. Thus extreme cases will be evened out and this virtually guarantees finding a sequence for e with under $\frac{5}{4}\log_2 e$ multiplications.

To test 512-bit exponents and investigate how small a coefficient c might be possible, 2^{12} of the most frequently used divisor/residue pairs were chosen from optimal chains of exponents up to over 2^{20} using divisors up to about 3×2^{10}. Choosing $k = 4$, an average of under 631 multiplications was achieved for randomly generated 512-bit exponents, with a standard deviation of under 4 (see Table 1). This clearly improves on the coefficient 5/4. Assuming the distribution is close to normal, a total of at most 640 multiplications or squarings can be effectively guaranteed for over 98% of all 512-bit exponents. In comparison, the square & multiply and A, A^3 methods average $\frac{3}{2}\times511 = 766.5$ and about $\frac{4}{3}\times511 = 681.3$ multiplications with standard deviations of about 11.3 and 6.2 respectively.

k	1	2	3	4
Av. No. Mults.	649.4	636.1	632.3	630.9
Stand. Dev.	3.45	3.3	3.4	3.9

Table 1. Multiplication numbers for 2^{12} (d,r) pairs.

Only a few divisors are actually required in order to achieve an average of under 5/4. Consider again the 12 divisors 2, 3, 5, 17, 33, 49, 65, 97, 129, 257, 513 and 1025. The 182 most frequently occurring divisor/residue pairs were selected. Then with $k = 5$ the algorithm yielded an average of around 639 multiplications, so that $c < 5/4$ for all larger k (see Table 2).

k	1	2	3	4	5
Av. No. Mults.	667.6	647.5	644.0	641.3	639.5
Stand. Dev.	4.9	4.7	4.5	4.4	4.8
Av. No. Divs.	188	198	211	207	202

Table 2. Multiplication numbers for the 12 listed divisors.

The cost of this particular scheme for a random exponent of $N = 512$ bits is easy to estimate. The sum in

[†] Demonstrated at the conference.

section 6 for approximating S shows there are about 4.85 ways of extending any subchain by one more pair, and, if 202 divisors are applied on average, the typical divisor has $N/202$ bits. So $L \approx 2.5$. Averaging over a few subchains would also give L. If no look-up table is used, there are around $202/k \approx 40$ iterations of the algorithm in which $S^k \approx 2620$. So a complete chain is obtained with an effort of roughly 2620×40 divisions of 512-bit numbers by numbers averaging 6 or 7 bits.

As this is the equivalent of about 2^{10} multiplications of pairs of 512-bit numbers, the effort is only justified in RSA cryptography if the key is re-used a number of times. However, for $k = 1$ or 2 the effort is reduced to around 10 or 28 such multiplications respectively. This leads to about the same computational effort as the A, A^3 method if searching for a multiplication scheme must be done for every iteration.

Finally, about 2^9 pairs for 29 divisors with low Hamming weight were considered: the 12 above together with 9, 41, 43, 81, 83, 161, 163, 193, 321, 323, 385, 641, 643, 769, 1281, 1283 and 1537. Here numbers of the forms $5\times2^n+1$ and $5\times2^n+3$ have a minimal addition chain of length $n+3$ by using the sequence 1, 2, 3, 5, 10, 20, ..., 5×2^n, $5\times2^n+1$ or $5\times2^n+3$. The work involved can be estimated from Table 3 using $S \approx 7.1$: about 13 and 53 512-bit multiplications' worth of effort for $k = 1$ and 2 respectively. Overall there is again little difference with the A, A^3 method unless the same RSA key is re-used.

k	1	2	3	4	5
Av. No. Mults.	655.4	639.9	636.3	634.1	632.9
Stand. Dev.	4.0	3.6	3.6	4.0	4.7
Av. No. Divs.	139	157	160	164	169

Table 3. Statistics for the 29 listed divisors.

Comparing the contents of the three tables, it is clear that adding further divisors brings diminishing returns. It would be interesting to know the theoretical limit of the method: on average, how many multiplications are required in an optimal division chain if all divisor/ residue pairs are possible and a given fixed number of registers are allowed?

8 Conclusion

A straightforward, cheap algorithm has been described for reducing the number of multiplications normally performed in an exponentiation. Almost no extra memory is required. The average improvement is better than established methods with the same memory requirements. Furthermore, the method is adaptable to a wide range of space and time resources, providing a variable search space from which better evaluation orders can be found. For any exponent e and very high probability of success, it has been shown how to find a scheme requiring under $\frac{5}{4}\log_2 e$ multiplications or squarings when only three registers are available.

Acknowledgement: The author would like to thank the anonymous referee who corrected the argument in Section 4 and kindly improved the presentation by supplying a table similar to that given.

Bibliography

[1] P. Downey, B. Leony & R. Sethi, "Computing Sequences with Addition Chains", *SIAM J. Comput.* vol. **3** (1981), pp. 638-696.

[2] Ö. Eğecioğlu & Ç. K. Koç, "Fast Modular Exponentiation", *Comm., Control & Signal Processing*, ed. E. Arikan, Elsevier Science, 1990, pp.188-194.

[3] L. C. K. Hui & K.-Y. Lam, "Fast square-and-multiply exponentiation for RSA", *Electronics Letters*, vol. **30**, no. 17 (Aug 1994), pp. 1396-1397.

[4] D. E. Knuth, *"The Art of Computer Programming"*, vol. **2**, *"Seminumerical Algorithms"*, §4.6.3, 2nd Edition, Addison-Wesley, 1981, pp. 441-466.

[5] Ç. K. Koç, "High radix and bit recoding techniques for modular exponentiation", *International J. Computer Mathematics*, vol. **40** (1987), nos. 3-4, pp 139-156.

[6] Y. Yacobi, "Exponentiating Faster with Addition Chains", *Advances in Cryptology - Eurocrypt 90*, Lecture Notes in Computer Science, vol. **473**, Springer-Verlag, 1991, pp. 222-229.

[7] C. N. Zhang, H. L. Martin, & D. Y. Y. Yun, "Parallel Algorithms and Systolic Array Designs for RSA Cryptosystem", *Proc. of the International Conference on Systolic Arrays*, K. Bromley, S.Y. Kung & E. Swartzlander (Eds.), Computer Society Press, San Diego, California, May 25-27, 1988, pp. 341-350.