Resource Requirements for the Application of Addition Chains in Modulo Exponentiation

Jörg Sauerbrey
Andreas Dietel

Lehrstuhl für Datenverarbeitung
Technische Universität München
P. O. Box 20 24 20
W-8000 München 2
Germany

sy@ldv.e-technik.tu-muenchen.de

Abstract

Addition chains or sequences can be used to reduce the amount of multiplications to accomplish an exponentiation at the cost of more memory required. We examine known methods of exponentiations based on addition sequences and derive the parameters determining operation count and number of required registers for storing intermediate results. As a result an improved method is proposed to choose window distributions as a basis for using known addition sequence heuristics.

1. Introduction

A lot of cryptographic methods and protocols rely on the fast evaluation of powers modulo a large number n. One of the famous members of this class of methods is RSA. Exponentiation is usually based on modulo multiplications which can be broken down into additions. A lot of research work has been done to implement fast modulo multiplication [AliMar91, CuBoKa91, Even90, LipPos90, LuHaLH88, Montgo85, Moraga89, Morita90] and to speed up addition [KocHun90, Hwang79]. This paper deals with reducing the number of modulo multiplications for one modulo exponentiation. The aim is to obtain a computation rule for a specific exponent which leads to less multiplications than the usual methods. The effort for deriving this computation rule pays off, if one has to compute a lot of exponentiations with the same exponent. This holds for example for RSA, where the exponent is part of the key and subsequent encryptions use the same key. The computation rule could then be stored together with the key.

In this paper different computation rules are compared with regard to the number of multiplications and the amount of memory required to perform the exponentiations. We focus our investigation mainly on the amount of memory, because memory is a scarce resource for a VLSI implementation. We propose an improved method for finding computation rules, where the memory requirements are economic.

2. Exponentiation and Addition Chains

According to the rule $b^x b^y = b^{x+y}$ the computation of a power b^{x+y} corresponds to the problem of finding a sequence of increasing integers approaching the exponent. The sequence has to begin with 1 and every integer is the sum of two preceeding integers in the sequence. Such sequences are called addition chains. In general, an addition chain for n is a sequence of integers $(1=a_0, a_1, \ldots, a_r=n)$ with the property that $a_i = a_j + a_k$ for some k and j, $k \le j < i$, for all $i=1, 2, \ldots, r$ ([Knuth69], p. 402).

R.A. Rueppel (Ed.): Advances in Cryptology - EUROCRYPT '92, LNCS 658, pp. 174-182, 1993.

For example, the exponentiation b^x accomplished by repeated multiplications by b results in the addition chain $(1, 2, 3, \ldots, x)$. The well known binary algorithm (repeated squaring), which is based upon the binary representation of x, defines an addition chain $(1, 2, \ldots, a_i, \ldots, x)$, where

$$a_{i-1} = \begin{cases} a_i - 1 & a_i \text{ odd} \\ a_i/2 & a_i \text{ even} \end{cases} \qquad (1)$$

We will show later that it is desirable to compute several predefined powers within the application of a single sequence of exponentiations. According to this requirement we define an addition sequence for $n_0,\ldots,n_k$ as an addition chain, which at least contains the elements $n_0,\ldots,n_k$. A star chain is an addition chain or sequence $(a_0,\ldots,a_r)$, where each term a_i is the sum of a_{i-1} and a previous a_k.

We define the following: $l(n_0,n_1,\ldots)$ is the length of the addition chain or sequence containing $n_0,n_1,\ldots$ The length of the addition chain or sequence with the star property (star chain) is defined by $l^*(n_0,n_1,\ldots)$. The Hamming weight $v(n)$ denotes the number of 1's in the binary representation of n. For $l(n)$ of the shortest addition chain it is known that $\log_2 n + \log_2 v(n) - 2.13 \le l(n)$. For addition sequences it is [Yao76]:

$$l(n_0, n_1, \ldots, n_k) \le \log n_k + c \sum_{i=0}^{k} \frac{\log n_i}{\log\log(n_i + 2)} \text{ where } c \text{ is a constant.} \qquad (2)$$

The star property may be suitable to reduce the amount of memory required to accomplish the exponentiation, but it does not guarantee that we get the shortest chain as $l(n) \le l^*(n)$ (Theorem of W. Hansen in [Knuth69], p. 413). For example the addition chain defined by equation (1) has the property, that only the value of the preceding step of the chain (a_{i-1}) and the value of b have to be stored for computing b^x. So using the binary algorithm, only two n bit registers are necessary to compute $b^x \bmod n$.

There are plenty of theoretical results and asymptotic bounds concerning different kinds of addition chains (see references in [BosCos89]), but much less practical hints for building and using addition chains. The main problem is, that computing the shortest addition chain is an NP-complete problem [DoLeSe81].

[McCart86] discusses the interaction between the efficiency of the basic multiplication algorithm and the addition chain used to compute b^x. If the cost of a multiplication is bound to the length of the operands, the multiplicative cost of evaluating b^x is minimized by using repeated squaring. But for modulo multiplications the cost of a multiplication is nearly constant. In this case it is clear, that the shortest chain for x will yield to the cheapest evaluation of b^x, in terms of number of multiplications. The binary algorithm does not define the shortest chain, as it can be shown for x=15. Here the shortest chain is (1, 2, 3, 6, 12, 15), whereas the binary algorithm yields to (1, 2, 3, 6, 7, 14, 15). However in most cases, saving of multiplications results in the necessity of storing more intermediate results.

[BosCos89] have shown some heuristics to compute addition sequences, which are mostly better than those derived from the standard binary method for exponentiation. With these heuristics it is possible to produce addition chains, which are on the average 21% shorter than those of the binary algorithms (using 512 bit exponents).

3. Known methods for the application of addition chains

Since even with these heuristics it is not feasible to compute an addition chain for an exponent x with length of 512 or 1024 bits, the goal of determining the computation rule is achieved in two steps. The first step is to reduce the computation of an addition chain for a large x to the computation of an addition sequence by choosing an appropriate set of numbers (window values) which are much smaller than x. The last step is to compute a 'good' addition sequence for these numbers.

There are different methods to accomplish the first step. They are all based on the m-ary method described in [Knuth69, p. 404]. If an exponentiation with the m-ary method has to be computed, x is rewritten as $x = d_0 m^l + d_1 m^{l-1} + \ldots + d_l$. This means that with $m = 2^k$ the binary representation of the exponent is divided in t windows with the width of k bit each and window values $d_i \in [0, m\text{-}1]$. The corresponding addition chain is as follows:

$$
\begin{aligned}
&1,2,3,\ldots,(m-2),(m-1),\\
&\quad 2d_0,4d_0,\ldots,md_0,(md_0+d_1), \qquad (3)\\
&\qquad 2(md_0+d_1),4(md_0+d_1),\ldots,m(md_0+d_1),(m^2d_0+md_1+d_2),\\
&\qquad \ldots,(m^l d_0+m^{l-1}d_1+\ldots+d_l)
\end{aligned}
$$

The first row serves to compute all possible window values (d_i), whereas each following row 'shifts' a new window value to the next window position in the binary representation of the exponent. Note that the width of the windows not necessarily have to be fixed.

There are several ways to decrease the length of the chain in order to reduce the number of multiplications for the exponentiation.

1. The last operation of a line of equation (3) correspondent to a particular window can be omitted if the window value is zero. If we take each string of one or more zeros of the binary representation of the exponent as a window, the LSBs of the remaining window values cannot be zero, Now the values of the first line of equation (3) decrease to the odd values between 1, ..., m-1.
2. Compute only those values in the first line, which are used as d_i's in the following lines. This can be accomplished with an addition sequence algorithm. The maximum size of the windows can now be much larger, since we don't have to store all (odd) values of [1,m-1] in a table.
3. The window distribution should be optimized. For example: the window values should be chosen, such that a short addition sequence can be constructed, with respect to the preceding point; there should be as few windows as possible; the same window values should appear more than once.
4. Choose the windows from right to the left such that many windows contain patterns of bits from windows standing to the right of them. This method is analog to the compression algorithm of Ziv and Lempel [ZivLem78]. If the values are stored in a tree, the derivation of new values is done easily using the values already computed. This method has been published in [Yacobi90] and is called compression method.

Of course, these reductions can be combined. The methods for defining windows have a great effect on the operation count, but their effect on the number of intermediate results to be stored (memory demand) is not obvious.

4. Effects on operation count and memory demand

The effect on the number of operations is rather clear. In detail there are four factors:

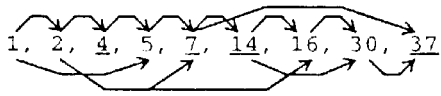
1. The length of the exponent x determining the number of squarings.
2. The length of the leftmost window (MSB-window), which reduces the number of squarings.
3. The number of non-zero windows minus one, which determines the number of multiplications. (Note: windows with the value of zero don't need a multiplication.)
4. The length of the addition sequence minus one, which defines the number of operations needed to compute the different window values. (Note: This is not the case for the method of [Yacobi90])

Figure 1 gives an example:

Window distribution (non-zero windows underlined):

exponent in binary representation:	111	00	100	1110	000	111	100101	0000	100
decimal window values:	7		4	14		7	37		4

Addition sequence, corresponding to the first row of equation 3 (window values underlined; the arrows indicate which numbers are added to get a new one):



number of operations: (length of the exponent)-(length of the MSB-window)+(number of non-zero windows-1)+(length of the addition sequence-1) 37-3+5+8=47

for comparison the number of operations with the binary method (length of the exponent-1)+(number of '1's in the exponent): 30+14=44

Figure 1: Window distribution and number of operations

The importance of examining the memory demand for computing b^x with the methods described in section 3 becomes obvious, if we look at the correspondent hardware realization.

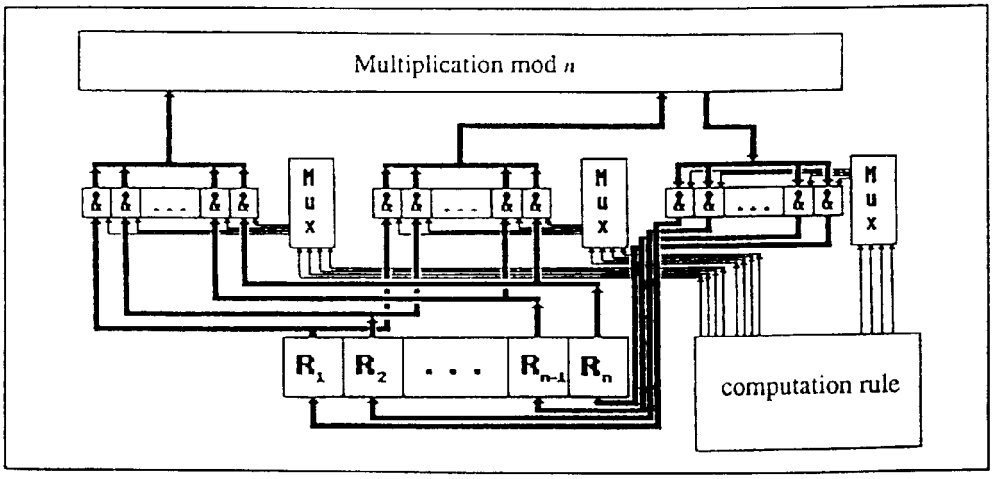


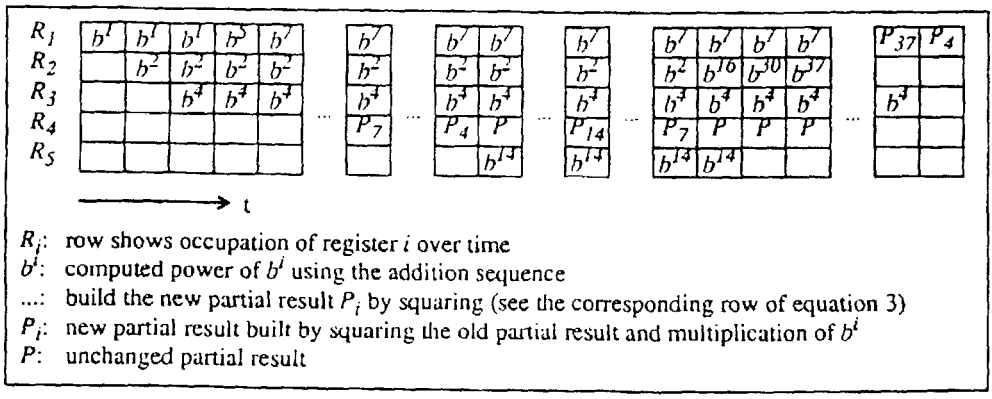
Figure 2: Hardware implementation of an exponentiation unit using addition chain methods

Figure 2 shows an example for the implementation of a hardware unit to compute b^x. There is a fixed number of registers (R_i) available for storing intermediate results. Initially R_1 is loaded with b. The box 'computation rule' is a controller selecting the correct source and destination registers R_i for each computation step. The computation rule for the actual x to accomplish the exponentiation is loaded into this block in advance. It implicitly contains the window distribution and the addition sequence, found by one of the algorithms above.

The effects of the computation rule on the required number of registers, cannot be summarized in a simple formula whereas the effects on the number of operations can be. The effect on the register count is investigated in the following points:

1. Does the window distribution contain non-adjacent windows sharing a common value?
   Once this value is computed, it has to to be stored for later use. The window value 4 in the example shown in figure 1 explains this point.
2. Is a window value necessary for the computation of the following window values?
   The value 14 in the addition sequence is an example for this point, because it is needed to create the value of 30.
3. Are the results, arising from the computation of the present window value necessary for the computation of the following window values?
   The value of 2 shows this aspect, because 2 is necessary to build 4, but has to be stored longer to create 16.
4. If traversed from left to right, how well are the windows sorted for increasing values?
   This point is important because we can start to compute the corresponding row of equation 3 once we have computed the corresponding window value within the addition sequence. If this value is not required any more we can drop it.

In the example of figure 1 at least 5 registers are needed to compute b^x. The content of the registers during the computation is as follows (figure 3):



R_i: row shows occupation of register i over time
b^i: computed power of b^i using the addition sequence
....: build the new partial result P_i by squaring (see the corresponding row of equation 3)
P_i: new partial result built by squaring the old partial result and multiplication of b^i
P: unchanged partial result

Figure 3: Content of the registers during the computation

5. Operation count and memory requirements of different exponentiation methods

In this section we investigate three different exponentiation methods, which are combinations of the reduction methods discussed in section 3:

1. A combination of the standard m-ary method, reduction 2, and the addition sequence heuristics of [BosCos89], called 'modified m-ary method'.
2. A combination of the reductions 1, 2 and 3, suggested by [BosCos89]. We will refer to this method as 'optimized m-ary method'.
3. A new combination of reductions 1, 2 and 3, with a special emphasis on the reduction of the number of intermediate results to be stored, which we will call 'size oriented window distribution'.

The following figures show the average number of the required operations and registers for 100 exponentiations with randomly selected 512 bit exponents ('1' and '0' are distributed evenly). The effect of different window sizes on the number of operations and registers is presented.

Figure 4 shows the simulation results for the 'modified m-ary method'. On the horizontal axis the window size is given in bit.

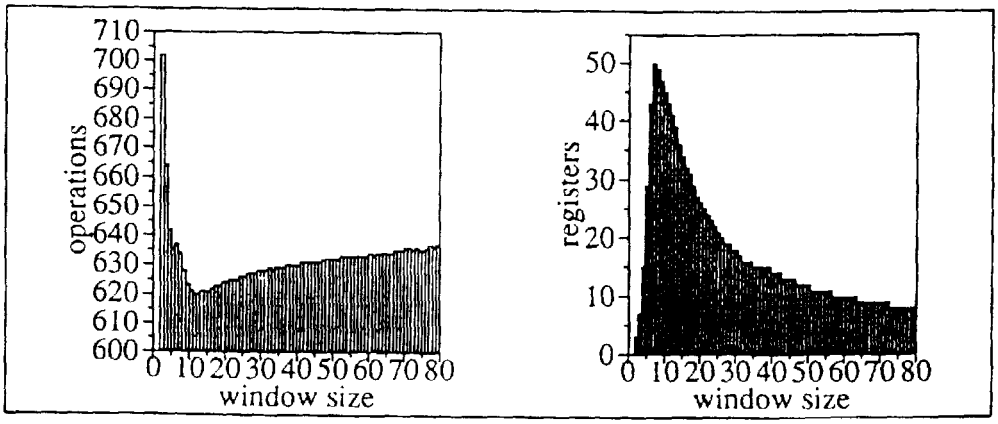


Figure 4: Number of operations and registers when applying the 'modified m-ary method'

Figure 5 shows simulation results for the 'optimized m-ary method'. The horizontal axis shows the maximal permitted window size in bits. In this algorithm the MSB-window has always the maximum size, thus reducing the number of squaring operations, due to the effect on the operation count as stated in section 4.

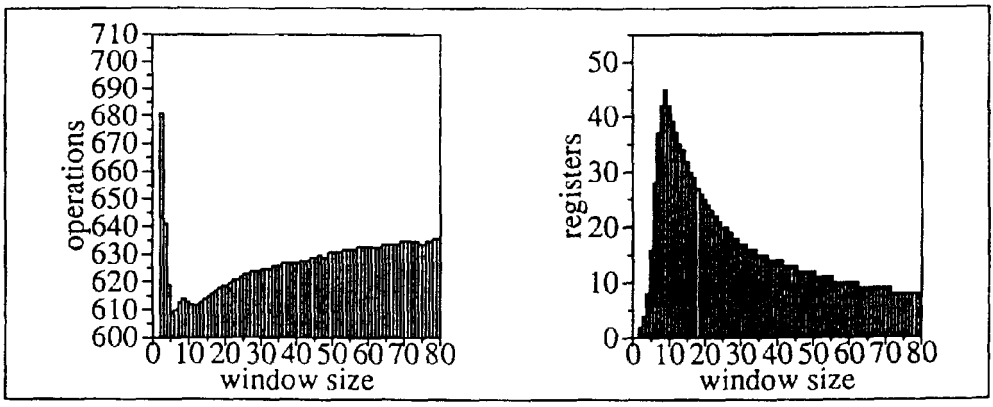


Figure 5: Number of operations and registers when applying the 'optimized m-ary method'

For comparison it should be recalled, that for 512 bit exponents the standard binary algorithm leads on the average to 767 operations and 2 required registers.

In both figures the number of operations has a distinct minimum. This is due to the fact that with increasing window sizes the number of windows decreases while the size of the MSB-window increases. However, when increasing the window sizes further, the addition chains for creating the window values become longer, because the employed heuristics are inefficient in identifying short sequences out of large numbers.

For both methods there is a rapid increase of the number of required registers with increasing window sizes, because of the exponential increase of the number of possible window values. This trend stops at some maximum value and the development reverses with further increasing window size. The reason for this effect is, that the bigger the window size the smaller the probability of the existence of all possible window values in the set of window values. For example, if the window size is 5 there are 16 possible window values (optimized m-ary method). The probability is very high that these window values exist more then once in the set of required window values of a 512 bit exponent. Thus a lot of different window values have to be stored for a long time, because they are needed later. With further increase of the window size, this probability decreases rapidly. Now only some of the existing window values have to be stored for later use.

The results of figures 4 and 5 illustrate the following problem: window sizes leading to good results for the number of operations require a lot of registers, and vice versa.

To reduce the memory requirements while keeping the number of operations low, we developed a new algorithm for selecting windows. The idea is to generate a window distribution with constantly increasing window values from left to right. This overcomes some reasons of storing values, because now window values appeare once and are sorted. However, this does not entirely eliminate the need for storing values (see section 4). The algorithm starts with a given size of the MSB-window and chooses new window values such that these values are greater than those of the windows to the left. Figure 6 shows an example of a window distribution generated with the 'size oriented window distribution'.

Example of a 'size oriented window distribution'
(windows underlined; initial window size: 3):

exponent in binary representation:	111	001001	1100	001111	0010100	0010...
decimal window values:	7	9	12	15	20	

Figure 6: Window distribution with the 'size oriented window distribution'

Figure 7 shows simulation results for the 'size oriented window distribution'. The horizontal axis shows the initial window size in bit.

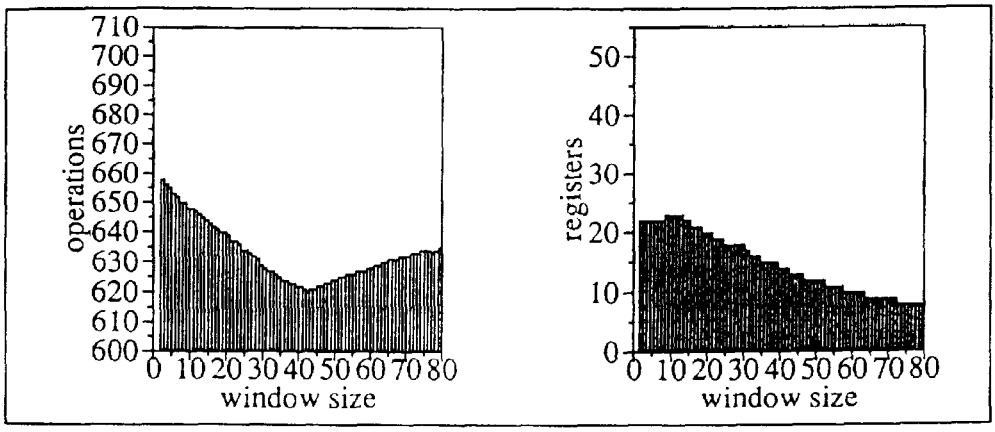


Figure 7: Number of operations and registers needed applying the 'size oriented window distribution'

The new algorithm actually provides a good compromise balancing the number of operations and the registers needed to accomplish the exponentiation. It is conceivable that an even better ratio can be achieved by a further improvement of the 'size oriented window distribution' algorithms and of the addition chain heuristics according to the effects explained in section 4.

6. Conclusion

We have examined different methods of exponentiation using addition chains. These methods have been compared with respect to the average number of multiplications and registers needed to accomplish exponentiations with randomly chosen exponents. The main factors influencing the operation count and memory requirements have been stated. A new method to choose window distributions (size oriented window distribution) has been proposed. It shows a better compromise between operation count and memory demand than previously known techniques.

7. References

[AliMar91] Alia, Giuseppe; Martinelli, Enrico: "A VLSI Modulo m Multiplier", IEEE Transactions on Computers, Vol. 40, No. 7, p. 873-878, July 1991

[BosCos89] Bos, Jurjen; Coster, Matthijs: "Addition Chain Heuristics", in Brassard, G. (Ed.): "Advances in Cryptology - Crypto '89", Proceedings (Lecture Notes in Computer Science 435), p. 400-407, Springer, 1989

[CuBoKa91] Curiger, A.V.; Bonnenberg, H.; Kaeslin, H.: "Regular VLSI Architectures for Multiplication Modulo (2 exp n + 1)", IEEE Journal on Solid State Circuits, Vol. 26, No. 7, p. 990-994, July 1991

[DoLeSe81] Downey, P.; Leong, B.; Sethi, R.: "Computing Sequences with Addition Chains", SIAM Journal on Computing, Vol. 3, No. 3, p. 638-646, August 1981

[Even90] Even, Shimon: "Systolic Modular Multiplication", in Menezes, A.J.; Vanstone, S.A.(Eds.): "Advances in Cryptology - Crypto'90 Proceedings (Lecture Notes in Computer Science 537), p. 619-624, Springer, 1990

[Hwang79] Hwang, Kai: "Computer Arithmetic: Principles, Architecture, and Design", John Wiley & Sons, New York, 1979

[Knuth69] Knuth, Donald E.: "The Art of Computer Programming, Vol. 2: Seminumerical Algorithms", Addison-Wesley, Reading, Massachusetts, 1969

[KocHun90] Koc, C. K.; Hung, C. Y.: "Multi-Operand Modulo Addition Using Carry Save Adders", Electronics Letters, Vol. 26, No. 6, p. 361-363, IEE, March 1990

[LipPos90] Lippitsch, P.; Posch, K.C.; Posch, R.: "Multiplication As Parallel As Possible", Institute for Information Processing Graz, Report 290, October 1990

[LuHaLH88] Lu, E.H.; Harn, L.;Lee, J.Y.; Hwang, W.Y.: "A Programmable VLSI Architecture for Computing Multiplication and Polynomial Evaluation Modulo a Positive Integer", IEEE Journal on Solid State Circuits, Vol. 23, No. 1, p. 204-207, February 1988

[McCart86] McCarthy, D. P.: "Effect of Improved Multiplications Efficiency on Exponentiation Algorithms Derived from Addition Chains", Mathematics of Computations, Vol. 46, No. 174, p. 603/608, American Mathematical Society, April 1987

[Montgo85] Montgomery, P. L.: "Modular Multiplication without Trial Division", Mathematics of Computation, Vol. 44, No. 170, p. 519-521, April 1985

[Moraga89] Moraga, Claudio: "Design of a Modulo p Multiplier", International Journal on Electronics, Vol. 67, No. 5, p. 819-827, Taylor & Francis, 1989

[Morita90] Morita, Hikaru: "A Fast Modular-multiplication Module for Smart Cards", Proceedings of AUSCRYPT '90 (Lecture Notes in Computer Science 453), p. 406-409, Springer, January 1990

[Yacobi90] Yacobi, Y.: "Exponentiation Faster with Addition Chains", in Damgard, I.B. (Ed.): "Advances in Cryptology - EUROCRYPT '90", Proceedings (Lecture Notes in Computer Science 473), p. 222-229, Springer, 1990

[Yao76] Yao, Andrew: "On the Evaluation of Powers", SIAM Journal on Computing, Vol. 5, No. 1, pp. 100-103, March 1976

[ZivLem78] Ziv, Jacob; Lempel, Abraham: "Compression of Individuel Sequences via Variable-Rate Coding", IEEE Transactions on Information Theory, Vol. IT-24, No. 5, pp. 530-536, September 1978