A Fast Method for Modulus Reduction and Scaling in Residue Number System

Oswin Aichholzer *
Hannes Hassler †

Abstract. Over the last three decades there has been considerable interest in the implementation of digital computer elements using hardware based on the residue number system. We propose a technique to compute a residue in this number system using a parallel network. Our technique enables scaling, too. We improve a former result of $O(n)$ cycles to $O(\log n)$, where n is the number of moduli. The hardware expense is the same, $O(n^2)$. Further advantages are that scaling factors can be chosen almost freely allowing scaling with radix 2. Negative numbers are covered as well, requiring no additional effort. Applications are RSA encryption and scaling.

Index Terms – DSP, multiplication, residue computation, residue number system, RSA encryption, scaling algorithms

1 Introduction

Over the last three decades there has been considerable theoretical interest in the *Residue Number System* (RNS) as applied to the implementation of digital arithmetic processors. The two main advantages are that multiplication is fast and binary operations $(+, -, *)$ can be performed without carry delay because of separability on each residue digit. This feature enables very fast arithmetic operations and is well suited for a parallel application.

However, there are also disadvantages since the implementation of operations such as division, scaling, sign detection and magnitude comparison is cumbersome and demands great efforts in time and/or hardware. Thus there seem to be no striking advantages of RNS in general purpose arithmetic units. However, there are high speed applications needing only the binary operations $(+, -, *)$ and a method for scaling or residue computation. We develop a parallel approach for residue computation on an arbitrary modulus with the mild restriction that there must be no common divisors with the RNS moduli set. This also provides scaling as a very welcome by-product.

In [2] a method of scaling is developed, needing $O(n)$ cycles with n the number of RNS moduli. The hardware expense is $O(n^2)$ look-up tables. Using properties of [9] and showing further features we develop a method for residue computation and scaling which needs only $O(\log n)$ cycles. Our hardware expense is $O(n^2)$, too. Another advantage is that scaling factors are almost independent of the moduli set.

*Department of Theoretical Computer Science, Graz University of Technology
†Department of Applied Information Processing and Communications Technology, Graz University of Technology

We have to take care only, that there is no common divisor with our RNS moduli set, which is no problem. Moreover, we show that our approach covers also negative numbers as well, requiring no change of the algorithm.

An application for the residue computation of numbers with many digits is the RSA encryption, see [8]. Here we develop a *relaxed residue computation* (RRC) enabling a parallelization of the needed multiplications. Note that the terminus *relaxed residues* has already been introduced in [5].

Scaling is important in digital signal processing (DSP), e.g. FFT or recursive filter. These algorithms rely on fast *multiply-and-add* techniques. A list of further applications can be found in [12].

The paper is organized as follows. In Sec.2 we explain RNS, how Chinese Remainder Theorem works, and how we can extend RNS with a redundant modulus. In Sec.3 we find out that extended RNS covers negative numbers and learn how to preprocess most of residue computation. In Sec.4 we consider two applications, RSA-encryption and scaling. We present an example which covers both residue computation and scaling.

2 Calculating in RNS

2.1 RNS, an alternate number system

RNS has a base of relatively prime moduli, $\{p_1, \dots p_n\}$. Let $P := \prod_{i:=1}^{n} p_i$ and we denote with $\langle a \rangle_k := a \bmod k$, a residue. According to the Chinese Remainder Theorem each number $x \in [0, P-1]$ is uniquely represented by the residues $(\langle x \rangle_{p_1}, \dots, \langle x \rangle_{p_n}) =: (x_1, \dots, x_n)$. Addition, subtraction and multiplication can be done in parallel with RNS. Assume we have two numbers a, b $a \hat{=} (a_1, \dots, a_n)$ $b \hat{=} (b_1, \dots, b_n)$. Then one can perform the operations $\otimes$ in parallel

$$a \otimes b \hat{=} (\langle a_1 \otimes b_1 \rangle_{p_1}, \dots, \langle a_n \otimes b_n \rangle_{p_n}) \quad \textit{where} \quad \otimes \in (+, -, \cdot) \tag{1}$$

However, when $a \otimes b \geq P$ a reduction modulo P is performed on the result. Hence, to achieve an exact result, overflow has to be prevented.

2.2 Transform to RNS

Fast two's-complement-to-residue conversion has been treated in [4]. However, for our purposes it is only an ancillary function. To receive RNS representation we compute $x_i = \langle x \rangle_{p_i}$ for each i by some conventional algorithm.

2.3 Transform from RNS

In the following we present residue-to-two's-complement conversion. Note that we essentially discuss the Chinese Remainder Theorem (CRT).

Let $\hat{P}_i := P/p_i$ and the notation $\hat{P}_i^{-1}$ will be used for the multiplicative inverse of $\hat{P}_i$ according to modulus p_i.

$$\textit{It holds} \quad \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} \equiv \begin{cases} x_i \bmod p_j & j = i \\ 0 \bmod p_j & j \neq i. \end{cases} \tag{2}$$

So by adding these products one gets a number $x^{'}$

$$x^{'} = \sum_{i:=1}^{n} \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} \qquad x^{'} \equiv x \bmod P \tag{3}$$

which is equivalent to the number x represented by the residues x_i (the range of summation is always considered going from 1 to n and therefore dropped in the sequel). We can express the exact value x by

$$x = \sum \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} - r_x P. \tag{4}$$

where r_x is integer. Formula (4) will be useful in the further discussion.

2.4 Calculation of r_x

For a correct conversion we have to compute r_x. One way is to divide equation (4) by P and compute the sum with fractional numbers. Truncate the result to an integer and you have r_x. This method has been introduced by Soderstrand et al. [10]. Many works on RNS made use of it, e.g. [3, 4, 6, 7, 13]. The drawback of fractional numbers is truncation error. To compute r_x in all cases correctly we need at least an accuracy of $\log_2 Pn$ bits. Vu analyses it in [13], formula (14).

However, we want to stay in RNS and deal with small items which have only $\log_2 p_i$ bits. Shenoy and Kumaresan [9] propose an elegant method to compute r_x by a redundant residue. They use it to compute base extension. We give a short description. We make an observation on the sum term in (4), i.e.

$$\sum_{i:=1}^{n} \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} < nP. \tag{5}$$

Hence, the above sum can exceed the range P by at most $(n-1)P$. Therefore, the value of r_x has to be $n-1$ or less. This bound on the value of r_x can be used for choosing an adequate redundant modulus. Let p_r be a redundant modulus, such that

$$p_r \geq n+1 \quad \textit{and} \quad gcd(P, p_r) = 1 \tag{6}$$

and x_r be the corresponding residue. This redundant residue is available from start to finish along with the residues $x_1, x_2, \ldots, x_n$. Reducing (4) to the redundant residue we get (7).

$$x_r = \left\langle \sum \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} - \langle r_x P \rangle_{p_r} \right\rangle_{p_r} \tag{7}$$

We arrange it to

$$\langle r_x \rangle_{p_r} = \left\langle P^{-1} \sum \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} - x_r \right\rangle_{p_r} \tag{8}$$

Since r_x is strictly bound from above by n $(< p_r)$, it follows that $r_x = \langle r_x \rangle_{p_r}$. r_x can be calculated from (8), using the network in Fig.1. Note, that all operations performed for moduli $p_1 \ldots p_n$ are performed as well on p_r so our number will be represented as $(x_1 \ldots x_n, x_r)$. We refer to this representation as *extended RNS*.

3 Calculations within extended RNS

In this section we show how to compute numbers in extended RNS. We show that r_x is even correct for negative numbers. From now on we assume that all binary operations $(+, -, *)$ are performed in parallel on each modulus including the redundant modulus p_r. We call them the *internal moduli*.

3.1 Negative Numbers

The discussion of CRT assumes x to be a nonnegative integer. However, formulas (4) and (8) work also for the negative case. We show the following

Theorem: Any number $x \in [-P+1 \dots P-1]$ represented by $(x_1, \cdots, x_n, x_r)$ will be correctly calculated by formulas (4) and (8).

Proof. (a) Given

$$-P < x < P \tag{9}$$

there are numbers $x' < x < x''$ so that x, x' and x'' all yield the same representation $(x_1 \dots x_n, x_r)$. There must hold $x' \leq x - p_r P$ and $x + p_r P \leq x''$ since x' and x'' must be at least $p_r P$ apart from x and therefore by using (9) holds

$$x' \leq P - 1 - p_r P \tag{10}$$

$$\textit{and} \quad -P + 1 + p_r P \leq x'' \tag{11}$$

(b) When we consider the range which (4) can produce by either assuming r_x or $\sum \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i}$ being maximal we get an upper and lower bound and can apply (10) and (11) to it which results in

$$x' < -(p_r - 1)P \leq \sum \hat{P}_i \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} - r_x P < nP \leq (p_r - 1)P < x'' \tag{12}$$

and therefore (4) never produces either x' or x''. Since there is no other number within $[-(p_r - 1)P \dots (p_r - 1)P]$ which has the same representation as x it follows that x is produced. □

Thus binary operations can be carried out between signed numbers without explicit knowledge of sign of either number. One only has to care about overflow prevention. Negation of a number is easy since multiplying by -1 is a defined operation.

3.2 Residue Computation

We want to perform reductions on some arbitrary modulus L with $gcd(L, P) = 1$ by our given residues. We call L the *external modulus*. Note that L is rather large compared to single p_i's. In our applications it is about $\log_2 L = (1/2) \log P$. We preprocess the residues $mod\, L$ of all large constants.

$$\tilde{x} = \sum \left\langle \hat{P}_i \right\rangle_L \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} + r_x \langle -P \rangle_L \tag{13}$$

We get a *relaxed residue computation* (RRC), i.e. not the residue $x^* < L$ but some number $\tilde{x} \equiv x^* \bmod L$. $\tilde{x}$ is in general case somewhat larger than L. We denote a

Figure 1: *A-Tree: Calculation of* r_x *with a binary tree where* $K = \langle 1/P \rangle_{p_r}$, $R(x_r) := \langle -x_r K \rangle_{p_r}$ *and* $A_i(x_i) := \langle K \langle \hat{P}_i \rangle_{p_r} \langle \hat{P}_i^{-1} x_i \rangle_{p_i} \rangle_{p_r}$

RRC of some number x by $\langle x \rangle_L^*$. RRC produces errors. We can show both for scaling and RSA that these errors are not severe, see Sec.4.

Formula (13) has still large constants, i.e. $\langle \hat{P}_i \rangle_L$ and $\langle -P \rangle_L$. We cut them down in the preprocessing and compute the residue for each internal modulus. With these small constants we can compute the L-residue of some number directly in RNS. We compute the sum in equation (13) for each specific modulus p_j.

Formulas (8) and (13) give us a parallel algorithm to compute a L-residue in RNS. The implementation of functions A, B, C, R (see Fig.1 and 2) can be done by look-up tables or processors performing the necessary multiplications. We compute each sums in parallel by a binary tree as shown in Fig.2. At the same time we compute r_x in a binary tree network which additionally uses the value x_r, see Fig.1. Fig.4 depicts a parallel network of the whole residue computation.

3.3 Algorithm and Analysis

The reduction algorithm works as shown in Fig.3. The corresponding tree scheme is depicted in Fig.4. The tasks which we discuss in Sec.4 will need another $n+1$ multiplications. To compare our approach with other results we assume that we can feed two numbers into one look-up table and one binary adder, respectively. Hence, we can merge two A (and B) functions and one addition into one look-up table. Note that multioperand adders would further reduce hardware requirements and the number of adding cycles.

The hardware expense for one reduction, i.e. the A-Tree and all B-Trees is $L_o = n(n+2)/2+1$ look-up tables and $A_o = (n+1)^2/2-1$ adders. The longest path needs

Figure 2: *B_j-Tree: Calculation of $\tilde{x}_j$ with a binary addition tree $\mod p_j$ where $B_i(x_i) := \langle\langle \hat{P}_i\rangle_L \langle \hat{P}_i^{-1} x_i\rangle\rangle_{p_j}$ and $C_j(r_x) := \langle r_x \langle -P\rangle_L\rangle_{p_j}$. The calculation of r_x is done in parallel by an A-Tree as illustrated by Fig. 1*

$Tl_o = 2$ look-ups and $Ta_o = \lceil \log_2 n + 1\rceil + 1$ additions, see Fig.1 and 2. As promised we have a hardware complexity of $O(n^2)$ and time complexity $O(\log n)$.

In an approach with larger moduli we would replace look-up tables by fast multipliers. For instance in a RSA application one would choose moduli too large for look-up tables, see [6].

4 Applications

4.1 RSA encryption

For an RSA encryption we compute $cipher := (message)^e \, mod\, L$, where e is the encryption key. We need a repeated modulus multiplication for RSA. As was pointed out in (13) we perform RRC's. We must be sure that the product of two such numbers does not exceed P.

Let $\tilde{x}_1$ and $\tilde{x}_2$ denote two numbers on which we did RRC. We require that P is an upper bound for the product of them. Formula (13) allows to establish

$$\tilde{x}_1\tilde{x}_2 \leq (L-1)^2(\sum_{i:=1}^{n}(p_i - 1) + p_r - 1)^2 < P \tag{14}$$

as a sufficient condition to cover RRC-error and to prevent overflow for repeated modulus multiplications. Note that the p_i-s can always be chosen that (14) will be satiesfied to guarantee that there will be no overflow of P. After the final multiplication

	Reduce(x)
(A1)	Feed each modulus including x_r into an A-Tree performing the calculations of formula (8) in parallel, see Fig. 1
(A2)	Feed each modulus $x_1 \cdots x_n$ and r_x simultaneously into $n+1$ B-Trees which compute (13) in parallel for each modulus, see Fig. 2

Figure 3: *Algorithm to perform a reduction* $\mod L$

we compute an exact residue $mod\, L$ by some auxiliary device. Since RSA consists of several hundred modulus multiplications this task has almost no time significance. We can resolve it by some conventional method.

A RSA modulus L has about 500 bits and more. One needed some P with more than 1000 bits. A full parallel scheme to implement RSA would consist of $n+2$ trees as shown in Fig.4 plus a multiplier for each modulus, rather too much for such a task.

However, there are *smaller scale* parallel solutions, which employ a linear array architecture. For instance Posch et al. [6] deal with RNS as a means to implement a distributed and faster RSA encryption than single chip solutions. Their approach is related to Soderstrand's technique, see Subsec.2.4.

4.2 Scaling

We show how to do scaling and compare it with a result of Jullien [2]. Scaling is similar to division. However, rather than division by an arbitrary number we know the divisor in advance. Multiplication of fractionals needs scaling. Consider for example the product $.568 \times .731 = .415208 \approx .415$. We compute exact integer multiplication and scale down to three digits, $\lfloor 415208/10^3 \rfloor$. Other than Jullien [2] we can choose the scaling factor to be independent of the moduli set. Naturally we take powers of two.

In RNS it is difficult to separate higher significant bits from lower significant ones. Scaling up is easy. We simply multiply $x2^l$. Note that 2^l has been transformed to RNS. We must be sure that the product $x2^l$ will not exceed P.

Scaling down is a much more difficult task. Simply multiplying by the inverse of 2^l obviously does not work. A residue computation by 2^l is a way to cut off l low significant bits. We compute $x_c := x - \langle x \rangle_{2^l}$. All l lower significant bits of x_c are zero. Now we can compute $x_s := x_c 2^{-l}$ in RNS, which is $x_s = \lfloor x/2^l \rfloor$.

Next we estimate the error caused by RRC instead of an exact reduction. Instead of x_c we get $x'_c := x - \langle x \rangle^*_{2^l}$. We know $x'_c \equiv x_c\, mod\, 2^l$ and all lower significant bits of x'_c are zero. With formula (13) we have an upper bound for the difference $\langle x \rangle^*_{2^l} - \langle x \rangle_{2^l} < np_{max} 2^l$. The error after scaling down is $|x_s - x'_s| < np_{max}$. In other words there are $t = \log_2 np_{max}$ lower significant bits faulty.

In an actual DSP application we need multiplication and scaling many times. We

Figure 4: *Whole reduction cycle. It consists of $n+1$ B-Trees and one A-Tree. Each B-Tree has n moduli inputs and one for r_x. The A-Tree has $n+1$ moduli inputs including the redundant one.*

must avoid overflow. Hence, all integers Y must be $Y < \sqrt{P}$. This integer represents $Y/2^l$. After multiplication we have a number $Z = Y_1 * Y_2$, which represents $Z/2^{2l}$. We scale it down with some error to $Z' \approx \lfloor Z/2^l \rfloor$, which represents $Z'/2^l$. This condition demands a scaling factor of 2^l and $l := \lfloor (1/2) \log_2 P \rfloor$. Note that each scaling operation adds only an error of 2^{t-l} at most. Hence, we have an actual accuracy of $l - t$ bits. See also the example in Subsec.4.3.

We consider a multiplier unit with scaling. We compare our hardware expenses with that of [2] for different quantities of moduli n.We deal with expenses for reduction in Subsec.3.3. For the multiplier and scaling we must add another $n+1$ look-up tables, $n+1$ adders, two look-up cycles and one adder cycle. For instance consider the case of 16 moduli. The approach of Jullien needs 164 look-up tables and 13 look-ups. Our method employs 162 look-up tables, 159 binary adders. We need 4 look-ups and 6 additions, see Tab.1.

4.3 Example:

We choose the moduli set $p_i \in (11, 13, 15, 17, 19, 23)$ (therefore $P = \prod_{i:=1}^{6} = 15935205$) and the redundant modulus $p_r = 7$. We want to perform calculations of the form $a_1 + a_2 b$ and compute the residue $mod\, L = 2^{11}$. Thus we preprocess the constants $\hat{P}_i$ =(1448655, 1225785, 1062347, 937365, 838695, 692835)' and $\left\langle \hat{P}_i^{-1} \right\rangle_{p_i} =$ $(10, 7, 8, 9, 6, 4)'$. For the RRC we get $\langle \hat{P} \rangle_L = (719, 1081, 1483, 1429, 1063, 611)'$ and we have $\langle -P \rangle_L = 283$.

	Jullien		Our Method			
	Hardware	Time	Hardware		Time	
n	L_j	T_j	L_o + A_o		$Tl_o + Ta_o$	
8	42	8	50	47	4	6
12	93	10	98	95	4	6
16	164	13	162	159	4	6
20	255	15	242	239	4	7
24	366	17	338	335	4	7
28	497	19	450	447	4	7

Table 1: *Comparison of multiply-scaling-by-estimate of Jullien with our approach for a growing amount of moduli n. Jullien uses for all binary operations including additions look-up tables. We separated look-up tables from adders. Hardware costs grow quadratically with either method, where our method needs roughly twice as much. Time grows linear with Jullien's method whereas our method needs only logarithmic time. Naturally, one could reduce further A_o and Ta_o by multioperand adders.*

We perform the calculation $a_1 + ba_2$ using some decimal constants a_1, a_2 and a number b from the intervall $(0, 1)$. For this example we represent the numbers in RNS as integers with an implizit fixed exponent of 2^{-11}. Thus after each multiplication we have to rescale the result by $L = 2^{11}$ showing how our reduction works.

Let $a_1 = 0.1875 = 384 \cdot 2^{-11} \hat{=} (10, 7, 9, 10, 4, 16 \mid 6)$, $a_2 = 0.5625 = 1152 \cdot 2^{-11} \hat{=} (8, 8, 12, 13, 12, 2 \mid 4)$, and $b = 0.4375 = 896 \cdot 2^{-11} \hat{=} (5, 12, 11, 12, 3, 22 \mid 0)$. We first perform $a_2 b$ and a reduction $mod\, L$ to rescale the result.

$$
\begin{aligned}
& x = a_2 b = (7, 5, 12, 3, 17, 21 \mid 0) \\
& \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} = (10 \cdot 7, 7 \cdot 5, 8 \cdot 12, 9 \cdot 3, 6 \cdot 17, 4 \cdot 21)' = (4, 9, 6, 10, 7, 15)'
\end{aligned} \tag{15}
$$

For calculation of r_x we need $K = \langle P^{-1} \rangle_{p_r} = 6$ and $\left\langle \hat{P} \right\rangle_{p_r}' = (5, 1, 6, 2, 4, 3)$, respectively. We get

$$
\begin{aligned}
r_x &= \left\langle -x_r K + K \left\langle \hat{P} \right\rangle_{p_r}' \left\langle \hat{P}_i^{-1} x_i \right\rangle_{p_i} \right\rangle_{p_r} = \\
&= \langle -0 \cdot 6 + 6 \cdot (5 \cdot 4 + 1 \cdot 9 + 6 \cdot 6 + 2 \cdot 10 + 4 \cdot 7 + 3 \cdot 15) \rangle_7 = 3
\end{aligned} \tag{16}
$$

One could compute now formula (4) directly, but in our network we do the calculation for each modulus independently. For example for $p_1 = 11$ we calculate

$$
\begin{aligned}
& \left\langle \left\langle \hat{P} \right\rangle_L \right\rangle_{11} \langle \hat{P}_i^{-1} x_i \rangle_{p_i} + r_x \langle -P \rangle_L = 4 \cdot 4 + 3 \cdot 9 + 9 \cdot 6 + \\
& \quad + 10 \cdot 10 + 7 \cdot 7 + 6 \cdot 15 + 3 \cdot 283 = 1185 \equiv 8 \bmod 11
\end{aligned} \tag{17}
$$

and we compute similarly for $p_2, \cdots p_7$. We get $\tilde{x} = \langle x \rangle_L^* = (8, 0, 13, 4, 10, 3 \mid 6)$.

To rescale we compute $\hat{x} = x - \tilde{x} = (10, 5, 14, 16, 7, 18 \mid 1)$. Now we multiply it by $L^{-1} = 2^{-11}$, i.e. the inverse of $2048\, mod\, (P p_r)$. $\langle L^{-1} \rangle_{p_i} = (6, 2, 2, 15, 14, 1 \mid 2)$.

Finally we get $a_1 + L^{-1} \cdot \hat{x} = (10, 7, 9, 10, 4, 16 \mid 1)$. Transforming back to decimals this results in $862 \cdot 2^{-11}$=0.4209. Thus due to RRC our result differs about 3 percent to the expected result of 0.4336.

Summary and Conclusions

We adapt a base extension method of Shenoy and Kumaresan [9] to compute residues and scale in RNS. We preprocess constants which allow an easy mapping from RNS to RNS for these functions. We show that the algorithm covers negative numbers, too.

Residue computation and scaling is almost independent from the RNS moduli set. Hence, extern moduli and scaling factors can be chosen arbitrarily. We use residue computation to do scaling. We can reduce the time for scaling in RNS from $O(n)$ [2] to $O(\log n)$. The hardware expense is for both methods $O(n^2)$.

RNS has also been considered as a means to encrypt by RSA in parallel. We show how to apply our method to it. The most important applications of RNS lie in DSP. Scaling is needed to manage the overflow problem. Higher precision arithmetic requires more moduli. Our algorithm scales in a very short time even for large systems which employ many moduli.

Acknowledgments

We thank Professor Reinhard Posch for valuable discussions.

References

[1] H. Hassler and O. Aichholzer, *A fast method for modulus reduction in Residue Number System*, TU-Graz, IIG-Report 312 Sep. (1991), and TU-Wien, EPP'93 (Economical Parallel Processing), Proceedings, 40–56, Feb. (1993)

[2] G. A. Jullien, *Residue Number Scaling and Other Operations Using ROM Arrays*, IEEE Trans. Comput. 27, 325–336, Apr. (1978)

[3] J. Y. Kim, K. H. Park and H. S. Lee, *Efficient Residue-to-Binary Conversion Technique with Rounding Error Compensation*, IEEE Trans. Circuits Syst. 38, 315–317, Mar. (1991)

[4] S. J. Meehan, S. D. O'Neil and J. J. Vaccaro *An Universal Input and Output RNS Converter* IEEE Trans. Circuits Syst. 37, 799–803, June (1990)

[5] K. C. Posch and R. Posch, *Approaching encryption at ISDN speed using partial parallel modulus multiplication*, Microprocessing and Microprogramming 29, 177–184, May (1990)

[6] K. C. Posch and R. Posch, *Residue number systems, a key to parallelism in public key cryptography*, Proceedings of Fourth IEEE Symposium on Parallel and Distributed Processing, 432–435, Dec. (1992)

[7] K. C. Posch and R. Posch, *Base Extension Using a Convolution Sum in Residue Number System*, Computing 50, 93–104, Jan. (1993)

[8] R. L. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM 21, 120–126, Feb. (1978)

[9] A. P. Shenoy and R. Kumaresan, *Fast Base Extension Using a Redundant Modulus in RNS*, IEEE Trans. Comput. 38, 292–297, Feb. (1989)

[10] M. A. Soderstrand, C. Vernia and J.-H. Chang, *An improved residue number system digital-to-analog converter*, IEEE Trans. Circuits Syst. CAS-30, 903–907, Dec. (1983)

[11] N. Takagi, H. Yasuura and S. Yajima, *High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree*, IEEE Transactions on Computers 34, 789–796, Jul. (1985)

[12] F. J. Taylor, *Residue Arithmetic: A tutorial with examples*, IEEE Computer 17, 50–62, May (1984)

[13] T. V. Vu, *Efficient Implementation of the Chinese Remainder Theorem for Sign Detection and Residue Decoding*, IEEE Trans. Comput. c-34, 646–651, July (1985)