Eindhoven University of Technology

MASTER

Design of a RSA crypto-processor using a systolic array

Kuipers, E.A.M.

Award date:
1996

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

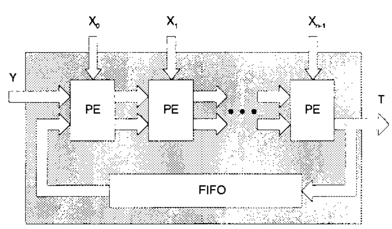
- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EB 616

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section Information and Communication Systems

Design of an RSA crypto-processor using a systolic array



Graduation report

Author : E.A.M. Kuipers
Coach : R. Joosten
Supervisor : Prof. Ir. M.P.J. Stevens
Date : June 1996

The faculty of Electrical Engineering of Eindhoven Univerisy of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.

Prof. Ir. M.P.J. Stevens,
vakgroep Informatie en Communicatie Systemen,
faculteit Elektrotechniek,
TU Eindhoven.

19 Juni 1996

Betreft: *Geheimhouding van afstudeerverslag*

Geachte heer Stevens,

Vanwege het feit dat het afstudeerproject betreffende het ontwerp van een RSA cryptie-processor heeft plaatsgevonden in het bedrijf Pijnenburg Custom chips B.V., wordt verzocht het verslag betreffende dit afstudeerproject niet openbaar te maken voor het jaar 2001.

Hartelijk dank voor de medewerking.

Met vriendelijke groet,

Erwin Kuipers

Abstract

This report describes the design of a scalable RSA device, which is suited for public-key encryption and decryption according to the Rivest, Shamir and Adleman method [Riv77]. This design has been developed in the context of a graduation assignment at the section Information and Communication Systems of the faculty Electrical Engineering of the Eindhoven University of Technology. This assignment is characterized as follows:

> *Design a parameterizable RSA cryption-processor, which can be optimized on either chip-size or cryption speed. The goal is to achieve maximum flexibility, which allows the processor to be used in any environment using an optimal configuration.*

The RSA design is based on a modular multiplication core, which executes the Montgomery algorithm [Mon85]. This algorithm requires conversions to and from an N-residue domain, but it is faster than the conventional 'paper & pencil' method and is easier to implement in hardware.

The multiplication core (MMM) is a systolic array, which consists of a number of processing elements (PE's), which can be varied in number and size. The number and size of the PE's are parameters which can be used to configure the RSA design to optimally perform in it's environment.

For this purpose the Montgomery algorithm has been adapted for systolic arrays, which results in a PE design which is proposed by Iwamura et al. [Iwa94]. In this report the steps are described, which are required to adapt the Montgomery algorithm to an efficient algorithm suited for systolic arrays. All conditions, which are required for this algorithm in order to prevent overflow or underflow are described. Further a schematic of the systolic array is presented, which shows the data flow in the PE's. Finally a schematic of an RSA processor is presented, which is based on the MMM-core.

The MMM-core has been simulated and functionally tested, from which can be concluded that the adapted Montgomery algorithm is working correctly. The PE's of the MMM-core have been described in VHDL and compiled to hardware-design. These compilations show, that using mimimal hardware optimization of the PE's, a (best case) cryption speed of 80 cryptions (1024 bits) per second can be achieved at a clock frequency of 66 MHz using a datapath of about 70 Kgates. When small chip size is required, the RSA design can be adapted to perform at 27 MHz using 6 Kgates. Using this configuration the RSA device can calculate about 5 cryptions of 1024 bits per second.

Table of contents

1 Introduction

In this chapter public-key crypto-systems are explained, and the encryption/authentication methods are described. Then the need for a flexible RSA cryption-device is explained, from which the graduation assignment can be characterized. Finally a number of situations are described which this device is suitable for.

1.1 Public-key cryptography

Today's communication is largely based on production and transport of digital information. The largest part of this information consists of private data, which may not be read or changed by unauthorized persons. This requires the application of safety measures, like isolated communication channels or data encryption. Because the latter is far more inexpensive, many crypto-systems have been developed to secure communication channels.

One crypto-system which has been in use for over 10 years now, is the DES-algorithm. DES is still commonly used, for it allows data blocks to be encrypted and decrypted fast and easily (over 20 Mbit/s in hardware), and still has withstood cryptanalysis attacks successfully. However, to decrypt a message, DES requires that both sender and receiver possess the cryption-key, which must be transferred using a safe communication channel.

The necessity for safe key-transfer can be avoided by using a public-key cryption method like RSA instead. Public-key cryptography, and RSA in particular, has no need for transmitting keys, for it is based on key-pairs: each sender/receiver has it's own public and private (secret) key. Because of the use of key-pairs, an identical algorithm can be used for both encryption and decryption. A message which has been *en*crypted using the public key, can be *de*crypted using the private key if and only if the private and public keys form a key-pair. This can be illustrated using the following example:
Sender 1 would like to send message M to receiver 2 using public-key cryptography. Sender 1 has key pair (*p1*, *s1*), receiver 2 has key pair (*p2*, *s2*) for public and secret keys. Using encryption/decryption function F_{key}(data), the data transfer can be illustrated using figure 1.1.

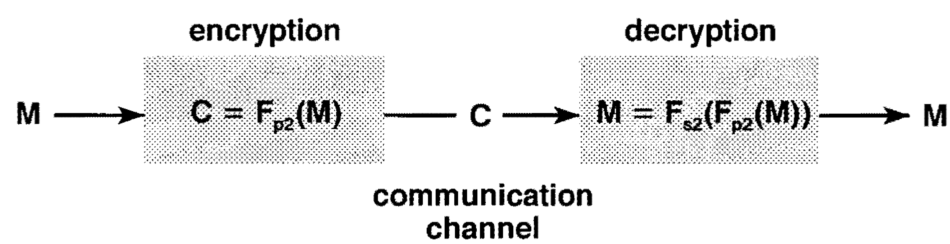


***figure 1.1**: Encryption/decryption using a public-key crypto-system*

Because the cyphertext C has been encrypted using the public key *p2*, it can only be decrypted using the secret key *s2*, which is only known by receiver 2. Therefore, receiver 2 is the only person who can decrypt cyphertext C to message M.

Public-key cryption implies that, using the cryption function F and the correct keypair, encryption and decryption should be calculated relatively easy. However, breaking the crypto-system by finding the inverse function $F_{p2}^{-1}(C) = F_{p2}^{-1}(\, F_{p2}(M)\,) = M$ (which means uncovering secret key *s2*) should require more time than the expiration date of the message (after which encryption is no longer necessary). More about the cryption function F is explained in chapter 2.

Because of the large digits used by the cryption function F (over 1024 bits), public-key encryption and decryption take too long for large messages. Therefore this cryption method is generally used in combination with DES: messages are encrypted fast using DES, and are sent to the receiver with the DES-key, which has been encrypted using a public-key encryption method. This allows the sender to safely send the DES-key with the DES-encrypted message using the same (unsafe) communication channel.

Besides encryption and decryption, public-key crypto-systems can also be used for authentication and verification of the sender's identity. The sender can send his signature by first encrypting a message using his own private key (authentication). Then the encryption using the receiver's public key is executed. On the receiver's side, the original message can be retrieved by decryption using the receiver's own private key, followed by verification using the sender's public key. The transferred message M can only be retrieved correctly if the public key *p1* matches the secret key *s1*. Because sender 1 is the only person who could have encrypted M using secret key *s1*, the identity of the sender of message M has been verified (see figure 1.2).

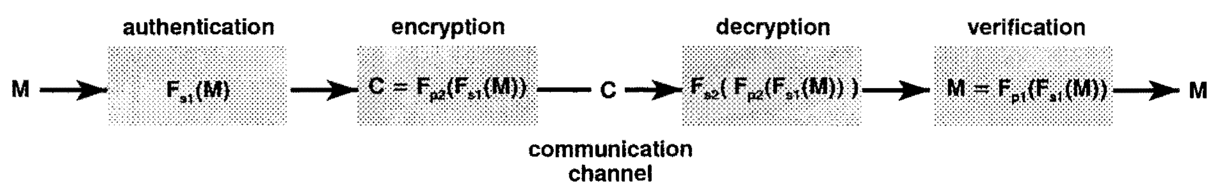


***figure 1.2**: Authentication/verification using a public-key crypto-system*

Today's most popular public-key system is RSA, because for many years now it has withstood numerous attempts to break this system, and it allows authentication and verification easily.

1.2 A scalable hardware RSA cryption-device

The last few years the RSA crypto-system has become more popular than ever, strongly encouraged by the increasing demand for data security on the Internet. Because of it's considerable amount of arithmetic operations and it's demand for still larger digits, software RSA cryption has proven to be too slow for many applications. The increasing demand for high-speed RSA cryption however requires custom hardware devices, which are suitable for fast arithmetic operations on digits of width 1024 bits and larger. These high-speed hardware devices are mainly designed to perform at maximum cryption speed at the cost of a large chip size. Other RSA applications however impose less severe restrictions on cryption speed; but demand small chip size. The following two examples illustrate this:

Network server:
- Cryption-time less than 10 ms.
- Chip size approximately 200 Kgates.
- Maximum clock frequency.

Chip-card:
- Cryption time approximately 0.5 seconds.
- Chip size less than 10 Kgates.
- Clock frequency 5 - 20 MHz.

Because usually DES and RSA are used together, for security reasons it is preferred to place hardware for both encryption methods, including memory, on one chip, which constrains available space. Other applications (e.g. cryptography in portable devices) constrain the operating frequency of the RSA device. These conditions require a flexible hardware design, which provides a trade-off of chip size against cryption speed. Now the graduation assignment, as described in this report, can be characterized as follows:

> *Design a parameterizable RSA cryption-processor, which can be optimized on either chip-size or cryption speed. The goal is to achieve maximum flexibility, which allows the processor to be used in any environment using an optimal configuration.*

To achieve flexibility in both time and space, the RSA cryption-device should be designed using a systolic array, which consists of a number of identical processing elements (PE's). The number and size of these PE's are parameters which directly relate to the number of clockcycles required for an RSA-cryption, the maximum clock-frequency, and the chip-size. These parameters can be adjusted to meet the requirements imposed by the hardware environment.

The scalable RSA cryption-device is applicable in the following situations:

- ***When chip size is constrained: Optimization on cryption time.***
  In the case of several hardware-devices on one chip (e.g. DES, RSA, memory, control, security hardware) only limited space is left for the RSA cryption part. Also, the maximum clock-frequency can be constrained by the processing speed of the environment.
  The clock-frequency determines the *size* of the PE's; the chip space determines the *number* of PE's. When both parameters are fixed, the maximum cryption speed for these parameters is achieved.

- ***When a reduced cryption speed is sufficient: Optimization on chip size.***
  In this case a minimum chip size can be obtained by adjusting the parameters (size and number of PE's) and clock-frequency. Because this application does not require high-speed RSA cryption, the internal bus-width can be made significantly smaller than the full cryption-width (>1024 bits), which reduces chip size considerably.

- ***When RSA-cryption is applied using variable cryption widths.***
  When using *smaller* cryption widths, less PE's can be activated, which reduces the number of clock-cycles, resulting in less cryption time.
  Using *larger* cryption widths (>1024 bits) can easily be achieved by increasing the number of PE's or connecting multiple RSA cryption-devices in cascade.

To find the optimal RSA cryption-device for any environment, it is desirable to make use of a graph which indicates the optimal size and number of PE's, given a specific chip size or cryption speed, as indicated in figure 1.3.

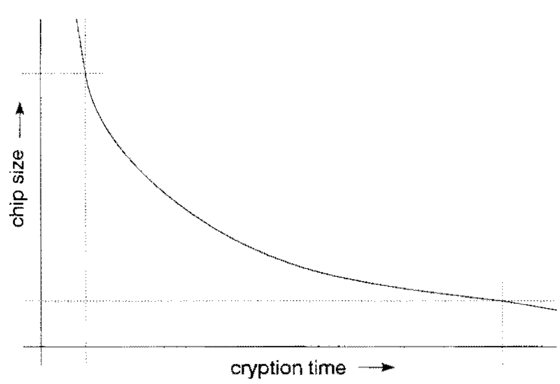


figure 1.3*: Trade-off of the optimal parameters of the RSA cryption-device*

This report describes the design of such a flexible RSA cryption-device, using PE's which have been designed to execute a modular multiplication algorithm in a pipeline structure.

2 RSA exponentiation

In this chapter the RSA public-key crypto-system is explained, and how the public and private keys are used in RSA calculations. An algorithm is presented, which can execute RSA exponentiation fast, using only modular multiplications.

2.1 What is RSA?

RSA is a public-key crypto-system for both cryption and authentication, introduced in 1977 by Rivest, Shamir and Adleman [Riv77]. RSA uses public key (N, e) and private key (N, d), where N, d, e are positive integers, N is odd and $d,e<N$. The cryption function $F_{key}(M)$ is defined as $M^{key} \bmod N$ $(M<N)$, so the data must be 'chopped' into digits smaller than N.
Using the keys (N, e) and (N, d), RSA cryption operates as follows:

- *Encryption of message M to cyphertext C:*
  $C \quad = F_e(M) = M^e \bmod N$

- *Decryption of cyphertext C to message M:*
  $M \quad = F_d(C) = C^d \bmod N = M^{ed} \bmod N$

The decryption of C using private key d returns the original message M if e, d and N are defined according to a set of rules. The modulus N is the product of two large primes, say p and q. Choose a private key $d<N$, which is relatively prime to $(p\text{-}1)(q\text{-}1)$. The public key e is defined as the multiplicative inverse of $d \bmod (p\text{-}1)(q\text{-}1)$, which means, that
$ed \bmod (p\text{-}1)(q\text{-}1) = 1$, so
$ed \equiv 1 + k\cdot(p\text{-}1)(q\text{-}1), \quad k \in \mathbb{N}.$

According to Euler and Fermat [Niv72], for any integer M relatively prime to $N=pq$ goes:
$M^{k\cdot(p-1)(q-1)} = 1 \bmod pq$

Now the decrypted message $F_d(C)$ can be written as:

$$\begin{aligned}
F_d(C) &= M^{ed} \bmod N \\
&= M^{1+ k\cdot(p-1)(q-1)} \bmod pq \\
&= M \bmod pq \cdot 1 \bmod pq \\
&= M \bmod N \\
&= M \qquad \forall_{0\leq M<N}
\end{aligned}$$

Which proves that decryption of C (using the private key d) results in the original message M.

RSA security is based on the assumption that factorization of large digits into prime numbers is very difficult (see [Pol74], [Dix92]). When the modulus N is factorized in the two primes p and q, the private key d can be revealed easily by calculating the multiplicative inverse (mod $(p\text{-}1)(q\text{-}1)$) of public key e. If however N is chosen large enough, the factorization problem is too complex to solve within the expiration time of the encrypted message. Currently an RSA modulus N of 130 decimal digits (432 bits) has been factorized with great effort.
An other way to uncover the private key d is exhaustive search. However, also this technique to break the RSA-code requires too much calculation effort if N is chosen large enough.
In today's RSA cryptography a modulus of 1024 bits or larger is recommended.

2.2 An RSA exponentiation algorithm

As mentioned before, RSA encryption calculates M^e mod N, RSA decryption calculates C^d mod N. Because both calculations are equivalent ($M,C<N$ and $d,e<N$), let's focus on the modular exponentiation $C = M^e \bmod N$.

Define $n=\lceil {}^2\log N \rceil$, so n is the number of bits of the RSA modulus. Because $e<N$, this exponent can be represented using binary digits:

$$e = \sum_{i=0}^{n-1} 2^i e_i = (e_{n-1} \ldots e_1 e_0) \tag{1}$$

Using this notation, C can be written as:

$$\begin{aligned} C &= M^{2^{n-1}e_{n-1} + \ldots + 2e_1 + e_0} \bmod N \\ &= \Big(\big((1 \cdot M^{e_{n-1}})^2 \cdot M^{e_{n-2}} \big)^2 \cdot \ldots \cdot M^{e_1} \Big)^2 \cdot M^{e_0} \bmod N \end{aligned} \tag{2}$$

So M can be exponentiated using n-1 squarings and n multiplications. However, in many cases the number of multiplications can be reduced, because if the exponent bit e_i is zero, the multiplication by M^{e_i} can be skipped.
The exponentiation algorithm for calculating C according to equation (2) now is as follows:

{input M, e, N}
C := 1
for i = (n-1) **downto** 0 **do**
begin
  if e_i=1 **then** $C := C \cdot M \bmod N$
  if i>0 **then** $C := C \cdot C \bmod N$
end
{output $C = M^e \bmod N$}

Note that during this algorithm the intermediate result C never exceeds N.

All most significant zero bits of exponent e, which precede the most significant '1'-bit, can be skipped, because for each of these zero exponent bits the algorithm will square the initial '1'.

Let ε be the number of bits required to represent e binary, so $\varepsilon = \lceil {}^2\log e \rceil \le n$. Now ε indicates the number of mod N-squarings executed by the exponentiation algorithm. Let η be the number of '1'-bits of exponent e, so $\eta \le \varepsilon$. Now η indicates the number of mod N-multiplications executed by the algorithm, and $\varepsilon+\eta$ modular multiplications are required to calculate M^e mod N.
The upper bound of the required number of modular multiplications thus is $2n$, which can be reduced to $1\frac{1}{2}n$ using the following exponentiation method, based on [Bri82]:

- if $\eta \le \frac{1}{2}\varepsilon$, e contains at most $\frac{1}{2}\varepsilon$ '1'-bits, so use the algorithm as presented before:

```
{input M, e, N}
C := 1
for i = (ε-1) downto 0 do
begin
   if e_i=1 then C := C·M mod N
   if i>0 then C := C·C mod N
end
{output C = M^e mod N}
```

- if $\eta > \frac{1}{2}\varepsilon$, e contains less than $\frac{1}{2}\varepsilon$ '0'-bits, so the inverse of e contains at most $\frac{1}{2}\varepsilon$ '1'-bits. Now the following algorithm can be applied using the precomputed value M^{-1} mod N:

```
{input M, e, N, M^-1 mod N}
e' := 2^ε - e
C := M
for i = (ε-1) downto 0 do
begin
   C := C·C mod N
   if e_i'=1 then C := C·M^-1 mod N
 end
{output C = M^e mod N}
```

Both algorithms require at most $\varepsilon+\frac{1}{2}\varepsilon \le 1\frac{1}{2}n$ modular multiplications.

Using the presented exponentiation method, RSA exponentiation boils down to repeated calculation of $C = A \cdot B$ mod N, where A, B, $C < N$ (so all can be represented using n bits). Other, more efficient exponentiation algorithms are presented in [Knu69], [Zha93], [Dim95] and [Kaw93], but all are based on repeated modular multiplications.
In the next chapters the design of a modular multiplier is described, which is particularly suited for RSA-exponentiation.

3 Modular Multiplication

This chapter describes how two digits of width n bits can be multiplied modulo N. First a 'paper & pencil' method is explained, which requires large bit comparisons. Then an alternative algorithm is presented, which has no need for bit comparisons, at the cost of necessary transformations to and from an N-residue domain. Finally some modifications are described which improve the performance of this alternative algorithm.

3.1 The 'paper & pencil' method

The modular multiplication $C = A \cdot B \bmod N$ $(A,B < N)$ can be calculated straightforward by first multiplying A and B, and then reducing the product by a multiple of N such that the result does not exceed N. This method is known as the 'paper & pencil' method and can be applied using the following algorithm:

$$C = A \cdot B \bmod N = A \cdot B - q \cdot N \qquad (A,B < N)$$

- *Multiplication* : Calculate $A \cdot B$
- *Trial division* : Find q with $0 \le q < N$ such that $0 \le C < N$

If n (the width of modulus N in bits) is large, the calculation of the full product $A \cdot B$ of width $2n$ should be avoided. This can be done by splitting both A and q in k digits of width α bits:

$$k = \left\lceil \frac{n}{\alpha} \right\rceil \tag{3}$$

$$\begin{aligned} A &= \sum_{i=0}^{k-1} 2^{\alpha i} a_i = (a_{k-1} \cdots a_1 a_0) \\ q &= \sum_{i=0}^{k-1} 2^{\alpha i} q_i = (q_{k-1} \cdots q_1 q_0) \end{aligned} \tag{4}$$

Now C can be calculated by multiplying each digit a_i by B, and by immediately reducing the result modulo N:

$$C = \sum_{i=0}^{k-1} \left(a_i B - q_i N\right) 2^{\alpha i} \tag{5}$$

The product $a_i B$ has only width $n+\alpha$ bits instead of $2n$, which reduces the multiplier size considerably if k is large.

This 'paper & pencil' method requires that for each product term a_iB a q_i is found in order to reduce it modulo N (trial division). The number of comparisons and subtractions can be reduced by skipping the modulo reduction (subtraction of $q_i N$) several multiplication steps and subtracting a larger multiple of N. This method however increases the size of the q-digits, which requires much additional hardware and a longer critical path. This issue returns in many hardware designs which are based on optimized 'paper & pencil' methods, as presented in [Bri82], [Mor90], [Wal93] or [Iwa93].
In [Mon85] an alternative algorithm is presented, which is based on transformations to and from an N-residue domain. In [Eld93], optimized 'paper & pencil' methods are compared to this Montgomery algorithm. It is concluded that the Montgomery algorithm can achieve twice the speed of the optimized 'paper & pencil' method described in [Bri82], at the cost of two extra registers. The operation and advantages of the Montgomery algorithm are shown in the next paragraph.

3.2 The Montgomery algorithm

Peter L. Montgomery has developed a method for calculating $C = A \cdot B \bmod N$ without the need for trial division. In [Mon85] he shows that the modulo reduction factor q does not have to be found using bit comparison, but can be calculated. This requires however conversion of A and B to an N-residue domain and conversion of the calculation result back to C in the integer domain.
The Montgomery method for modular multiplication can be described as follows:

1 Let N be a positive odd integer such that $2^{n-1} < N < 2^n$.
  Choose an $R = 2^r$, r a positive integer, which satisfies
  - $R > N$ ($r \geq n$)
  - $\gcd(R,N)=1$ (R is coprime to N, which is satisfied by N being odd).

2 Find integers R^{-1} and N' satisfying $0< R^{-1}<N$ and $0< N'<R$,
  such that $RR^{-1} - NN' = 1$, so
  - $RR^{-1} \bmod N = 1$ (R^{-1} is the multiplicative inverse of R modulo N)
  - $N' = (RR^{-1} - 1) \operatorname{div} N$

3 Let A, B, C be integers, $0 \leq A, B, C < N$.
4 Let X, Y, T be integers, $0 \leq X, Y < N$ and $0 \leq T < 2N$.
5 Let λ and μ be integers, $0 \leq \lambda, \mu < N$
6 Define function $\mathrm{MMM}(\lambda,\mu) = \lambda\mu \cdot R^{-1} \bmod N$.

Using the function $\mathrm{MMM}(\lambda,\mu)$ and a precalculated value $R_N=R^2 \bmod N$, the modular multiplication $C = A \cdot B \bmod N$ can be calculated as follows:

- *Convert the integers A and B to the N-residue domain using $MMM(..,R_N)$:*
  $X = \mathrm{MMM}(A,R_N) = A \cdot R^2 \cdot R^{-1} \bmod N = AR \bmod N$
  $Y = \mathrm{MMM}(B,R_N) = B \cdot R^2 \cdot R^{-1} \bmod N = BR \bmod N$

- *Calculate in the N-residue domain the modular multiplication MMM(..,..):*
  $T = \mathrm{MMM}(X,Y) = ABR^2 \cdot R^{-1} \bmod N = ABR \bmod N$

- *Convert T from the N-residue domain to C in the integer domain using MMM(..,1):*
  $C = \mathrm{MMM}(T,1) = ABR \cdot R^{-1} \bmod N = AB \bmod N$

The N-residue transformations are illustrated in figure 3.1.

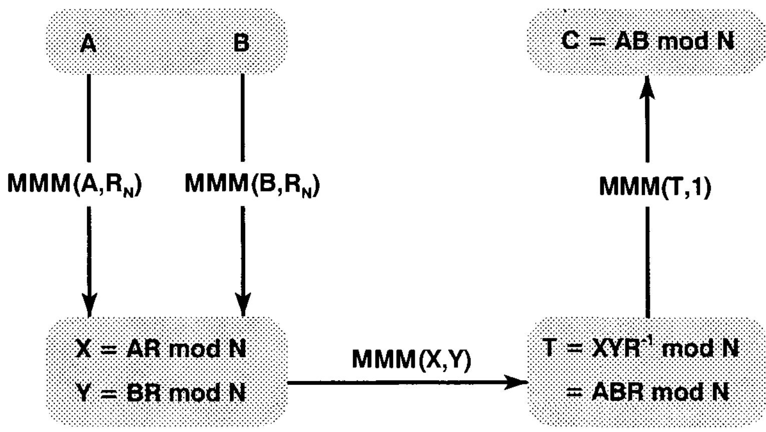


***figure 3.1**: Montgomery transformations to and from the N-residue domain*

Montgomery defines the function $\mathrm{MMM}(X,Y)$ as:

$$T = MMM(X,Y) = \frac{XY + mN}{R} = XY \cdot R^{-1} \bmod N \tag{6}$$

The factor m is defined as:

$$\begin{aligned} m &= (X \cdot Y \bmod R) \cdot N' \bmod R \\ &= XY \cdot N' \bmod R \equiv XY \cdot N' + k \cdot R \qquad (k \in \mathbb{I}) \end{aligned} \tag{7}$$

So $0 \leq m < R$.

Equation (6) can be proven by simply substituting (7):

$$\begin{aligned} T &= \frac{XY + XY \cdot NN' + k \cdot NR}{R} \\ &= \frac{XY(1 + NN') + k \cdot NR}{R} \end{aligned} \tag{8}$$

Using the Montgomery property $RR^{-1} - NN' = 1$, T can be written as:

$$T = \frac{XY \cdot RR^{-1} + k \cdot NR}{R} = XY \cdot R^{-1} + k \cdot N \equiv XY \cdot R^{-1} \bmod N \tag{9}$$

which proves that $T = \mathrm{MMM}(X,Y) = XY \cdot R^{-1} \bmod N$.

Because $X \cdot Y$, N and R^{-1} are integers, T can be shown to be an integer by calculating:

$$\begin{aligned} mN &= XYNN' + kNR = XY(RR^{-1} - 1) + kNR \\ &= -XY + (XYR^{-1} + kN)R = -XY + l \cdot R \end{aligned} \qquad \textbf{(10)}$$

Since $l = (XYR^{-1} + kN) \in \mathbb{I}$, the numerator of T is a multiple of R, which proves that T is an integer.

3.2.1 Adjustment of the modular multiplication result

The Montgomery algorithm shows that the product $X \cdot Y$ can be reduced modulo N using a division by $R = 2^r$, with $R > N$ and N is odd (so $r \geq n$). This integer division is allowed, for the lower r bits of the product XY are set to zero by adding an m-multiple of N, which does not affect the final result in the N-residue domain. This concept is illustrated in figure 3.2.

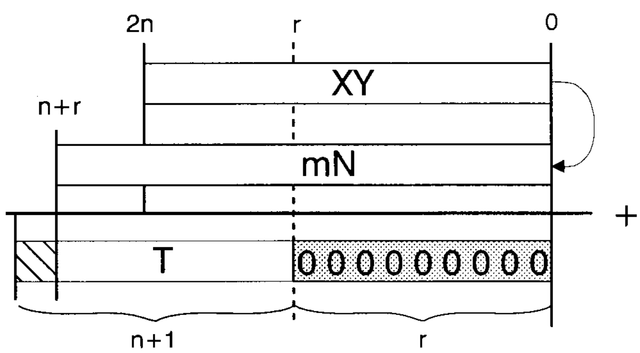


figure 3.2: Principle of the Montgomery algorithm

However, the MMM(X,Y) output can be equal to $XY \cdot R^{-1} \bmod N + N$ instead of the desired $XY \cdot R^{-1} \bmod N$. This can be demonstrated as follows:

Using $X, Y < N$ and $m, N < R$, an upper bound of $T = \mathrm{MMM}(X,Y)$ can be determined using equation (6) and the conditions imposed at X, Y and m:

$$T = \frac{XY + mN}{R} < \frac{N \cdot N + R \cdot N}{R} < \frac{R \cdot N + R \cdot N}{R} = 2N \qquad \textbf{(11)}$$

So if the MMM result T equals or exceeds N, T should be adjusted to $T - N$. After this adjustment T is an N-residue value smaller than N, so it satisfies the input conditions imposed at he input multiplicands X and Y. This means that (after this N-adjustment) MMM output values can be used as input values of a new MMM.

The necessity for N-adjustment of the Montgomery multiplication result (subtraction of N if $\mathrm{MMM}(X,Y) \geq N$) can be avoided by choosing R large enough. Because it is desired to use the MMM-output T ($< 2N$) directly for input to a new $\mathrm{MMM}(X,Y)$, the input conditions for X and Y should become $0 \leq X,Y < 2N$. With $R = 2^r$, the new condition for r can be found as follows:

1	$2^{n-1} < N < 2^n,\ n \geq 1$	$\Rightarrow\ N \leq 2^n - 1$
2	$0 \leq X,\ Y < 2N$	$\Rightarrow\ X,\ Y \leq 2^{n+1} - 3$
3	$R = 2^r,\ r = n + d$	$(d \in \mathbb{N})$
4	$m < R = 2^{n+d}$	$\Leftarrow\ m \leq 2^{n+d}$

Find a minimal d, such that $T = \mathrm{MMM}(X,Y) < 2N$ for all $X,\ Y < 2N$.

$$T = \frac{XY + mN}{R} \leq \frac{(2^{n+1}-3)\cdot(2^{n+1}-3) + 2^{n+d}\cdot(2^n-1)}{2^{n+d}} < 2(2^n-1) \quad \textbf{(12)}$$

This can be rewritten as:

$$\begin{aligned} &2^{2n+2} - 6\cdot 2^{n+1} + 9 < 2^{2n+d} - 2^{n+d} \quad \Rightarrow \\ &2^{2n}(4 - 2^d) - 2^n(12 - 2^d) + 9 < 0 \end{aligned} \quad \textbf{(13)}$$

This condition is satisfied for all $n \geq 1$ and $d \geq 2$. This means, that if $R = 2^r$, $r \geq n+2$ and $X,\ Y < 2N$, the calculated $T = \mathrm{MMM}(X,Y) < 2N$. The MMM-function now can be applied for repeated modular multiplications (as in exponentiation algorithms) without N-adjustment. However, the final result C after conversion back to the integer domain using $C = \mathrm{MMM}(T,1)$ may *not* exceed N. This modular multiplication requires adjustment only if $C = N$, for backward conversion of N-residue values $<2N$ always returns integers $\leq N$, which can be shown as follows:

1	$2^{n-1} < N < 2^n,\ n \geq 1$	$\Rightarrow\ N \leq 2^n - 1$
2	$0 \leq T < 2N$	$\Rightarrow\ T \leq 2^{n+1} - 3$
3	$R = 2^r,\ r = n + d$	$(d \in \mathbb{N})$
4	$m < R = 2^{n+d}$	$\Rightarrow\ m \leq 2^{n+d} - 1$

If C is the integer after conversion of T from the N-residue domain, so $C = \mathrm{MMM}(T,1)$, the upper bound of C can be determined as follows:

$$C = \frac{T\cdot 1 + mN}{R} \leq \frac{(2^{n+1}-3)\cdot 1 + (2^{n+d}-1)\cdot(2^n-1)}{2^{n+d}} \quad \textbf{(14)}$$

This can be rewritten as:

$$C \leq \frac{(2\cdot 2^n-3) + 2^{2n+d} - 2^{n+d} - 2^n + 1}{2^{n+d}} = 2^n - 1 + \left(\frac{2^n-2}{2^{n+d}}\right) < 2^n \quad \textbf{(15)}$$

So $C<2^n$ for all $n\geq 1$, implying that C can be at most equal to N after conversion of T back to the integer domain. Only then C must be set to zero in order to reduce C modulo N.

3.2.2 The Montgomery multiprecision algorithm

Because $R = 2^r$, $r \geq n+2$, the maximum $m < R$ is represented by at least n+2 bits. To avoid the calculation of the full-width product XY during a Montgomery modular multiplication (MMM), both X and m are split into k digits of width α bits, with $0 < \alpha$, $k \leq n+2$, so

$$k = \left\lceil \frac{n+2}{\alpha} \right\rceil \tag{16}$$

Now let $r = k\alpha$, so r is the smallest multiple of α which equals or exceeds n+2, indicating the number of bits which are used to represent X and m. These values can be written using base 2^α as

$$\begin{aligned} X &= \sum_{i=0}^{k-1} 2^{\alpha i} x_i = (x_{k-1} \ldots x_1 x_0) \\ m &= \sum_{i=0}^{k-1} 2^{\alpha i} m_i = (m_{k-1} \ldots m_1 m_0) \end{aligned} \tag{17}$$

under the condition that x_i and $m_i < 2^\alpha$.
Using equations (7) and (18), the Montgomery algorithm becomes:

$$R \cdot T = \sum_{l=0}^{k-1} \left(x_l Y + m_l N\right) 2^{\alpha l} \tag{18}$$

Division by $R = 2^{\alpha k}$ yields:

$$T = \sum_{l=0}^{k-1} \left(x_l Y + m_l N\right) 2^{-\alpha(k-l)} \tag{19}$$

Now the partial sum $T(i)$ can be defined using index i = 0, 1, ... , k-1:

$$T(i) = \sum_{l=0}^{i} \left(x_l Y + m_l N\right) 2^{-\alpha((i+1)-l)} \tag{20}$$

so $T(k\text{-}1) = T = XY\,R^{-1} \bmod N$.
If the last sum term is extracted from the entire sum of equation (21), $T(i)$ can be written as:

$$\begin{aligned} T(i) &= \sum_{l=0}^{i-1} \left(x_l Y + m_l N\right) 2^{-\alpha((i+1)-l)} + \left(x_i Y + m_i N \right) 2^{-\alpha} \\ &= 2^{-\alpha} \sum_{l=0}^{i-1} \left(x_l Y + m_l N\right) 2^{-\alpha(i-l)} + \left(x_i Y + m_i N \right) 2^{-\alpha} \\ &= \left(T(i-1) + x_i Y + m_i N \right) 2^{-\alpha} \end{aligned} \tag{21}$$

So instead of dividing the sum of products $xY + mN$ once by $R = 2^{\alpha k}$, now during k iteration steps the partial sum is divided by 2^{α}. This division is only permitted if the division result is an integer, so if the α least significant bits (LSB's) of the numerator of $T(i)$ are zero. Therefore m_i is defined as:

$$m_i = \left(T(i-1) + x_i Y \right) N' \bmod 2^{\alpha} \tag{22}$$

The principle of the Montgomery multiprecision case can be illustrated using figure 3.3, which shows two consecutive iteration steps.

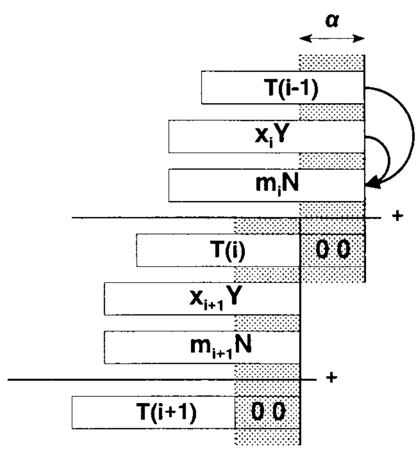


figure 3.3: The Montgomery multiprecision case

Notice that each intermediate result $T(i)$ has maximum width n+2 bits, for x_iY has width α+n+1 bits, and m_iN has width α+n bits. The final result $T(k\text{-}1)$ = MMM(X,Y) however has width n+1 bits, for equation (12) shows that T = MMM(X,Y) < $2N$ < 2^{n+1}.

3.2.3 Scaling the Montgomery multiprecision algorithm

As just has been shown, an MMM can be calculated using k iteration steps. In each step a digit of X is multiplied by Y, the result is added with the result of the previous iteration step and the whole is divided by 2^{α}. This division is only allowed if the numerator of $T(i)$ is a multiple of 2^{α}. For this purpose an m_i-multiple of N is added to this numerator, which is calculated using equation (22). However, this m_i cannot be calculated until the product x_iY is available, so the product m_iN can only be calculated afterwards.

The calculation of m_i can be simplified by shifting each product x_iY over α bits to the left, out of the grey area of figure 3.3. The scaled multiprecision Montgomery algorithm can be determined as follows:

If digits x_k and m_{-1} are set to zero, it follows from equation (18) that

$$\begin{aligned} R \cdot T &= \sum_{l=0}^{k-1} x_l Y \, 2^{\alpha l} + \sum_{l=1}^{k} m_{l-1} N \, 2^{\alpha(l-1)} \\ &= \sum_{l=0}^{k} (2^{\alpha} \cdot x_l Y) \; 2^{\alpha(l-1)} + \sum_{l=0}^{k} m_{l-1} N \, 2^{\alpha(l-1)} \end{aligned} \tag{23}$$

Left and right division by $R = 2^{\alpha k}$ yields:

$$T = \sum_{l=0}^{k} \left(2^{\alpha} \cdot x_l Y + m_{l-1} N\right) 2^{-\alpha(k-l+1)} \tag{24}$$

Now the partial sum $T(i)$ is redefined using index $i = 0, 1, \ldots, k$:

$$T(i) = \sum_{l=0}^{i} \left(2^{\alpha} \cdot x_l Y + m_{l-1} N\right) 2^{-\alpha(i-l+1)} \tag{25}$$

so $T(k) = T = XY\,R^{-1} \bmod N$.

By separating the last term of the entire sum of equation (26), $T(i)$ can be written as:

$$\begin{aligned} T(i) &= 2^{-\alpha} \sum_{l=0}^{i-1} \left(2^{\alpha} \cdot x_l Y + m_{l-1} N\right) 2^{-\alpha((i-1)-l+1)} \; + \; \left(\, 2^{\alpha} \cdot x_i Y + m_{i-1} N \,\right) 2^{-\alpha} \\ &= \left(\, T(i-1) + 2^{\alpha} x_i Y + m_{i-1} N \,\right) 2^{-\alpha} \end{aligned} \tag{26}$$

Now the lower α bits of the *numerator* of $T(i)$ depend only on the lower α bits of $T(i\text{-}1)$, which simplifies the calculation of m_{i-1} :

$$m_{i-1} = T(i-1) \cdot N' \bmod 2^{\alpha} \tag{27}$$

These results are also presented in [Iwa94] and [Dus90].

The multiprecision Montgomery algorithm scaled over α bits can be illustrated by figure 3.4, which shows two successive iteration steps.

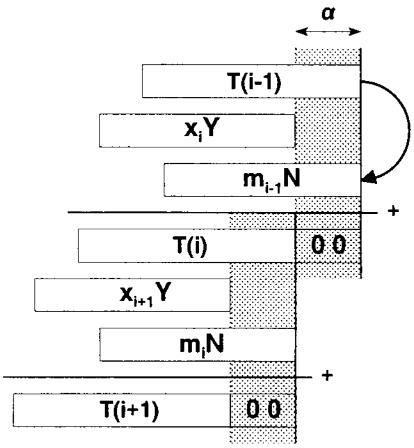


figure 3.4: The Montgomery multiprecision case scaled over α bits

Notice that the scaled Montgomery algorithm produces intermediate results $T(i)$ of width $n+2+\alpha$ bits, for the MSB of the product $2^{\alpha} \cdot x_i Y$ is located at bit position $n+1+2\alpha$. Again, by equation (12) the final result $T(k) = T < 2N$ has maximum width $n+1$ bits.

The Montgomery multiprecision algorithm scaled over α bits can be described as follows:

Montgomery conditions:

1 $n \geq 1$
2 $2^{n-1} < N < 2^n$, N is odd
3 $0 \leq X, Y, T < 2N$
4 $1 \leq \alpha \leq n+2$
5 $k = \lceil (n+2)/\alpha \rceil$
6 $r = k\alpha \geq n+2$
7 $R = 2^r = 2^{k\alpha}$
8 $RR^{-1} - NN' = 1$

{*input X, Y, N*}
$T(-1) = 0$
$m_{-1} = 0$
$x_k = 0$
for $i = 0$ **to** k **do**
begin
  $T(i) = (\, T(i-1) + 2^{\alpha} \cdot x_i Y + m_{i-1} N \,)$ **div** 2^{α}
  $m_i = T(i) \cdot N'$ **mod** 2^{α}
end
{*output* $T = T(k) = MMM(X,Y) = XYR^{-1} \bmod N$}

Each m_i is the product of the lower α bits of the currently calculated $T(i)$ and the lower α bits of the precalculated constant N'. For example, if α=1, m_i is the product of the LSB of $T(i)$ and N'. Because N' is always odd (see paragraph 6.4.1), it's LSB is always '1', so m_i can be retrieved straight from $T(i)$ without any calculation!
By choosing small values for α, m_i is calculated using a simple α x α multiplier, which can start multiplying while the higher order bits of $T(i)$ are calculated. This parallel arithmetic gives great benefit over the 'paper & pencil' method, which cannot start the q_i-determination until a great number of bits of the product a_iB has been calculated.

4 Montgomery in Systolic Arrays

In this chapter is described how the multiprecision Montgomery algorithm can be applied for systolic arrays. To reduce the internal bus width, the algorithm is adapted, which provides a new parameter which relates to the PE size. Although this adapted algorithm cannot be realized directly, after some modifications a flexible design is obtained which is suitable for systolic arrays.

As described in chapter 1, a flexible RSA cryption-device can be obtained by hardware design using a systolic array. The array consists of identical processing elements (PE's), of which the number and size can be adjusted in order to optimally perform in the environment. The multiprecision case of Montgomery's algorithm is well suited for hardware design using systolic arrays, for each iteration step can be calculated by one PE. If each PE processes a digit of X of width α bits, k+1 PE's would be required to execute a modular multiplication, as illustrated in figure 4.1.

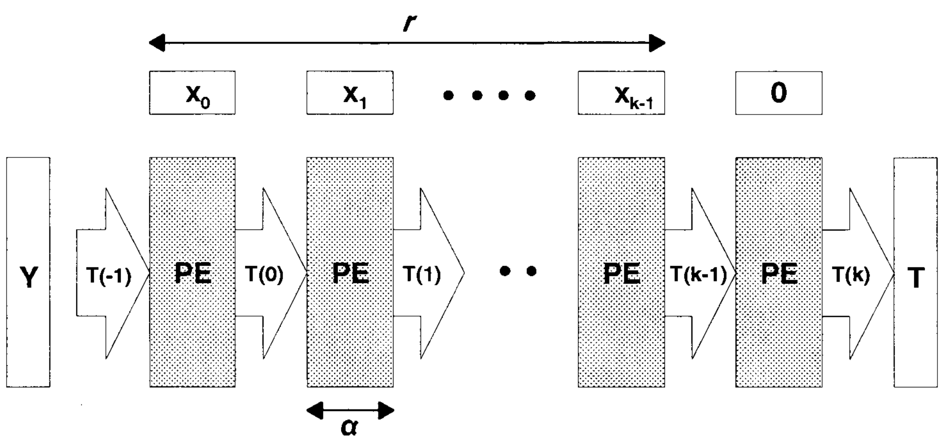


figure 4.1: MMM iteration steps in a systolic array

The size of the PE's is defined by α, and the number of PE's can be decreased. If less than k+1 PE's are used, the intermediate result T(i) (i<k) on the output of the last PE must be stored in a register (width n+2+α bits). The register contents can be loaded in the first PE, which then will calculate $T(i+1)$ using digit x_{i+1}. So if p is the number of PE's ($1 \leq p \leq k+1$), $\lceil (k+1) / p \rceil$ MMM-cycles are required.

4.1 Reducing the internal bus width

Although α provides some flexibility for the size of a PE, still (n+2+α) bits are processed each iteration step. This width may be too large for applications which require a small internal bus width. Therefore it is desired to split T, Y and N into smaller digits of width β bits.

If β is constrained to $\beta \leq n$, integer l can be defined as

$$l = \left\lceil \frac{n}{\beta} \right\rceil \tag{28}$$

Now let $s = l\beta$, so s is the smallest multiple of β which equals or exceeds n. Then $n \leq s < n+\beta$ and $N < 2^n$ can be represented binary using l digits of width β bits.

If also $\beta \geq \alpha+2$, both $T(i)$ and Y can be represented binary using $l+1$ digits of β bits:

$$T(i) = \sum_{j=0}^{l} t_j(i)\, 2^{\beta j}$$

$$Y = \sum_{j=0}^{l} y_j\, 2^{\beta j} \tag{29}$$

$$N = \sum_{j=0}^{l-1} n_j\, 2^{\beta j}$$

under the condition that $t_j(i)$, y_j and $n_j < 2^{\beta}$.

The most significant digits $t_l(i)$ contain the calculation overflow bits generated during the Montgomery modular multiplication. The digits $t_l(k)$ and y_k have at most bit n+1 of $T(k)$ and Y placed at the least significant bit position (only if n=$l\beta$), for both $T(k)$ and $Y < 2N < 2^{n+1}$.
The contents of the digits of Y, $T(i)$ and N are illustrated in figure 4.2.

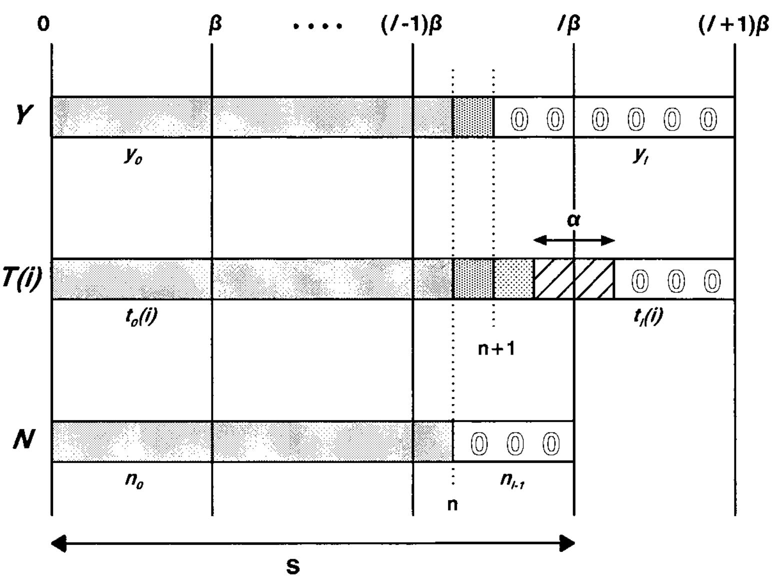


figure 4.2: Contents of digits Y, T(i) and N

The purpose of splitting Y, $T(i)$ and N into digits of β bits is to calculate one digit $t_j(i)$ of β bits in each PE. Now also β provides a parameter which directly relates to the PE size.

From equations (26) and (29) follows, that

$$T(i) = \sum_{j=0}^{l} t_j(i)\, 2^{\beta j} = \sum_{j=0}^{l} \frac{t_j(i-1) + 2^{\alpha} x_i y_j + m_{i-1} n_j}{2^{\alpha}}\, 2^{\beta j} \tag{30}$$

Notice that the total summations are equal, which does *not* imply that all individual sum terms are equal (each $t_j(i) < 2^{\beta}$, while the fraction on the right side of equation (30) is smaller than $2^{\alpha+\beta+1}$).

To determine how each $t_j(i) < 2^{\beta}$ can be calculated in one PE, we define:

$$U'(i) = 2^{\alpha} T(i) \tag{31}$$

Write $U'(i)$ redundant using $l+1$ digits of width $2\alpha+\beta+1$:

$$U'(i) = \sum_{j=0}^{l} u_j'(i)\, 2^{\beta j} = 2^{\alpha} T(i) = \sum_{j=0}^{l} \left(t_j(i-1) + 2^{\alpha} x_i y_j + m_{i-1} n_j \right) 2^{\beta j} \tag{32}$$

The summation term $u'_j(i)$ is defined by

$$u_j'(i) = t_j(i-1) + 2^{\alpha} x_i y_j + m_{i-1} n_j < 2^{2\alpha+\beta+1} \tag{33}$$

The upper bound for $u'_j(i)$ is justified by $t_j(i\text{-}1) < 2^{\beta}$, by $2^{\alpha} x_i y_j < 2^{2\alpha+\beta}$ and by $m_{i-1} n_j < 2^{\alpha+\beta}$.

Now if $u'_j(i)$ is split in three parts:

$$\begin{aligned} \delta_j(i) &= u_j'(i) \bmod 2^{\alpha} \\ t_j'(i) &= (u_j'(i)\, div\, 2^{\alpha}) \bmod 2^{\beta} \\ \gamma_j(i) &= u_j'(i)\, div\, 2^{\beta+\alpha} \end{aligned} \tag{34}$$

this value can be written as:

$$u_j'(i) = 2^{\beta+\alpha} \gamma_j(i) + 2^{\alpha} t_j'(i) + \delta_j(i) \tag{35}$$

The use of $U'(i)$ can be illustrated using figure 4.3.

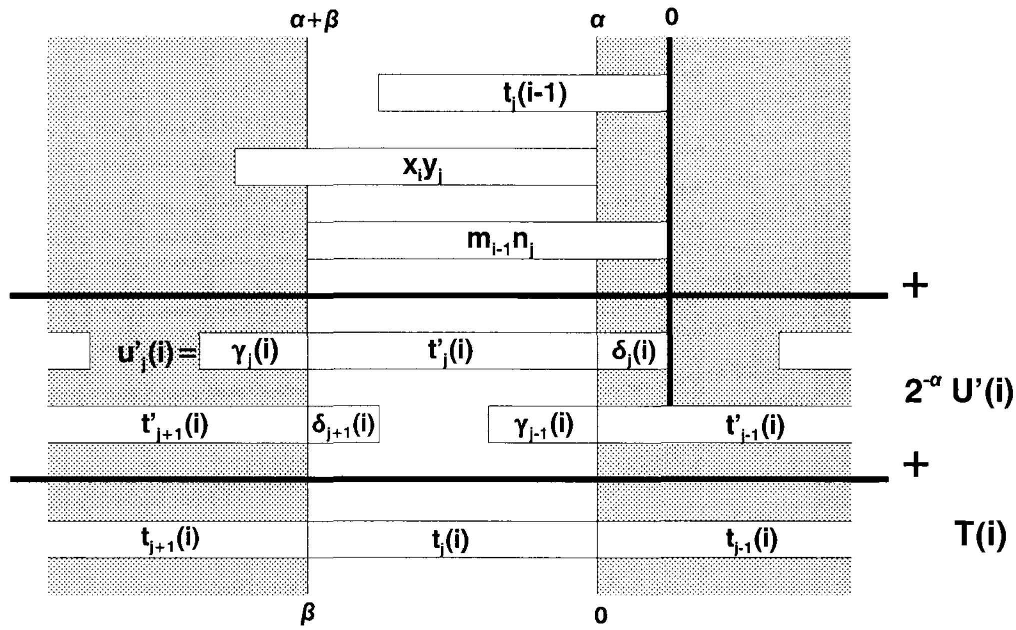


figure 4.3: Calculation of adjacent digits of T(i)

Using equations (30), (32) and (33), $T(i)$ can be written as:

$$
\begin{aligned}
T(i) &= \sum_{j=0}^{l} t_j(i) 2^{\beta j} = 2^{-\alpha} \sum_{j=0}^{l} u_j'(i) 2^{\beta j} \\
&= 2^{-\alpha} \sum_{j=0}^{l} \left(2^{\beta+\alpha} \gamma_j(i) + 2^{\alpha} t_j'(i) + \delta_j(i) \right) 2^{\beta j} \\
&= \sum_{j=0}^{l} \gamma_j(i) 2^{\beta(j+1)} + \sum_{j=0}^{l} t_j'(i)\, 2^{\beta j} + \sum_{j=0}^{l} 2^{-\alpha} \delta_j(i)\, 2^{\beta j} \\
&= \sum_{j=1}^{l+1} \gamma_{j-1}(i) 2^{\beta j} + \sum_{j=0}^{l} t_j'(i) 2^{\beta j} + \sum_{j=-1}^{l-1} 2^{-\alpha} \delta_{j+1}(i)\, 2^{\beta(j+1)} \\
&= \sum_{j=0}^{l} \gamma_{j-1}(i) 2^{\beta j} + \left(\gamma_l(i) 2^{\beta(l+1)} - \gamma_{-1}(i) \right) + \sum_{j=0}^{l} t_j'(i) 2^{\beta j} + \\
&\quad 2^{\beta-\alpha} \sum_{j=0}^{l} \delta_{j+1}(i)\, 2^{\beta j} + \left(\delta_0(i) - \delta_{l+1}(i) 2^{\beta(l+1)} \right)
\end{aligned}
\tag{36}
$$

Now both $\gamma_{-1}(i)$ and $\delta_{l+1}(i)$ are 0 by definition, for both values are out of the digit index range $j = 0 \ldots l$. Because $T(i) = 2^{-\alpha}U'(i)$ is an integer, $U'(i) \bmod 2^{\alpha} = 0$ for all $i = 0 \ldots k$, which implies by equation (34) that $\delta_0(i) = 0$. Also, $\gamma_l(i) = 0$, for this digit is located at bit position $n+\beta$ of $T(i) = 2^{-\alpha}U'(i) < 2^{n+\alpha+2} < 2^{n+\beta}$. So no more than the lower $\alpha+2$ bits of digit $t_l(i)$ are used.

According to equations (34) and (36), $T(i)$ can be written as:

$$T(i) = \sum_{j=0}^{l} t_j(i)2^{\beta j} = \sum_{j=0}^{l} \left(\gamma_{j-1}(i) + (u_j'(i)\, div\, 2^{\alpha}) \bmod 2^{\beta} + 2^{\beta-\alpha}\delta_{j+1}(i) \right) 2^{\beta j} \qquad \textbf{(37)}$$

Now the complete sumterm can be constrained modulo 2^{β}, for all addition overflow bits will ripple into $\gamma_j(i)$, which is processed in the next digit $t_{j+1}(i)$.

Using the definition of $u'_j(i)$ in equation (33), $T(i)$ can be written as

$$\begin{aligned} \sum_{j=0}^{l} t_j(i)2^{\beta j} &= \sum_{j=0}^{l} \left(\frac{u_j'(i) + 2^{\beta}\delta_{j+1}(i) + 2^{\alpha}\gamma_{j-1}(i)}{2^{\alpha}} \right) \bmod 2^{\beta} \cdot 2^{\beta j} \\ &= \sum_{j=0}^{l} \left(\frac{t_j(i-1) + 2^{\alpha}x_i y_j + m_{i-1}n_j + 2^{\beta}\delta_{j+1}(i) + 2^{\alpha}\gamma_{j-1}(i)}{2^{\alpha}} \right) \bmod 2^{\beta} \cdot 2^{\beta j} \end{aligned} \qquad \textbf{(38)}$$

The fraction represents an integer division (the lower α bits of the addition of the numerator can be ignored, for they are processed as $\delta_{j-1}(i)$ in the previous digit $t_{j-1}(i)$). If finally the numerator of the fraction of equation (38) is defined as:

$$u_j(i) = t_j(i-1) + 2^{\alpha}x_i y_j + m_{i-1}n_j + 2^{\beta}\delta_{j+1}(i) + 2^{\alpha}\gamma_{j-1}(i) \qquad \textbf{(39)}$$

with $j = 0 \,..\, l$ and $i = 0 \,..\, k$, the required digits can be calculated as follows:

$$\begin{aligned} \delta_j(i) &= u_j(i) \bmod 2^{\alpha} \\ t_j(i) &= (\, u_j(i)\ div\ 2^{\alpha}\,) \bmod 2^{\beta} \\ \gamma_j(i) &= u_j(i)\ div\ 2^{\alpha+\beta} \end{aligned} \qquad \textbf{(40)}$$

Because $T = T(k) = \sum_{j=0}^{l} t_j(k)\, 2^{\beta j}$, and all digits $t_j(k) < 2^{\beta}$, the Montgomery modular multiplication result is obtained by concatenation of all digits $t_j(k)$, so no post-processing (extra addition) is required.

The expression of equation (39) can be illustrated using figure 4.4.

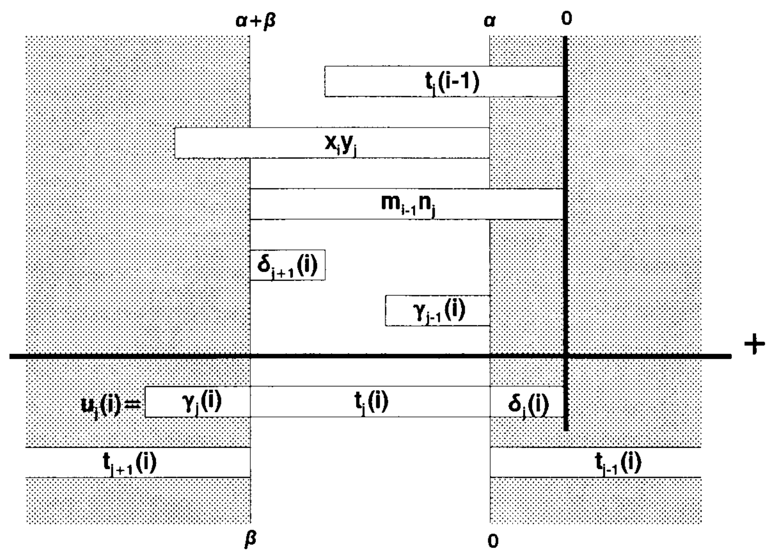


figure 4.4: Calculation of digit $t_j(i)$

In order not to lose any information bits during the division of $U(i)$ by 2^{α}, the lower α bits of the numerator of $t_0(i)$ are set to zero by adding $m_{i-1}N$, which corresponds to adding $m_{i-1}\, n_j$ to each digit $t_j(i)$. Because $t_0(i\text{-}1)$ is the only value which has effect on these lower α bits, m_{i-1} is defined as:

$$m_{i-1} = t_0(i-1) \cdot N' \bmod 2^{\alpha} \qquad \textbf{(41)}$$

The expressions (39), (40) and (41) provide an adapted algorithm of the Montgomery method for modular multiplication. The next step is to implement it in a systolic array.

4.2 The Montgomery algorithm adapted for systolic arrays

The MMM-result $T(k)$ can be retrieved by concatenation of all digits $t_0(k)$ to $t_l(k)$. The adapted algorithm of expression (39) can be implemented in a systolic array by loading x_0 to x_k in the consecutive PE's of the array, and inputting the digits of $T(\text{-}1)$, Y and N from least to most significant digit in the first PE serially. Each PE now has a set of registers, in which the input digits and temporary result are stored and passed on to the next PE. Each clock cycle PE number i calculates $t_j(i)$ using $t_j(i\text{-}1)$ from the preceding PE. The systolic array looks like the schematic of figure 4.5.

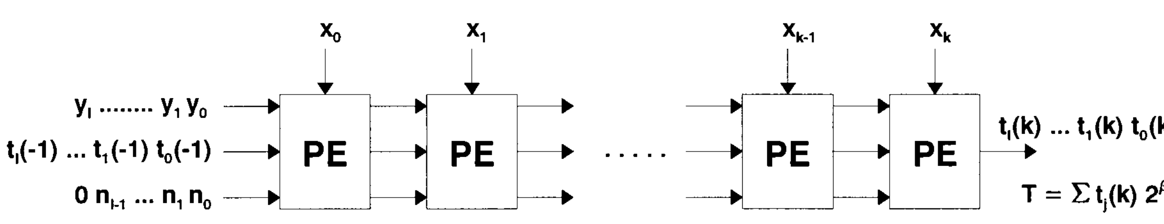


figure 4.5: Systolic array which calculates MMM(X,Y) using digits of X, Y, N and T

The data-flow in the systolic array can be illustrated by taking a 'snapshot' of a number of PE's for several clock cycles, as in figure 4.6.

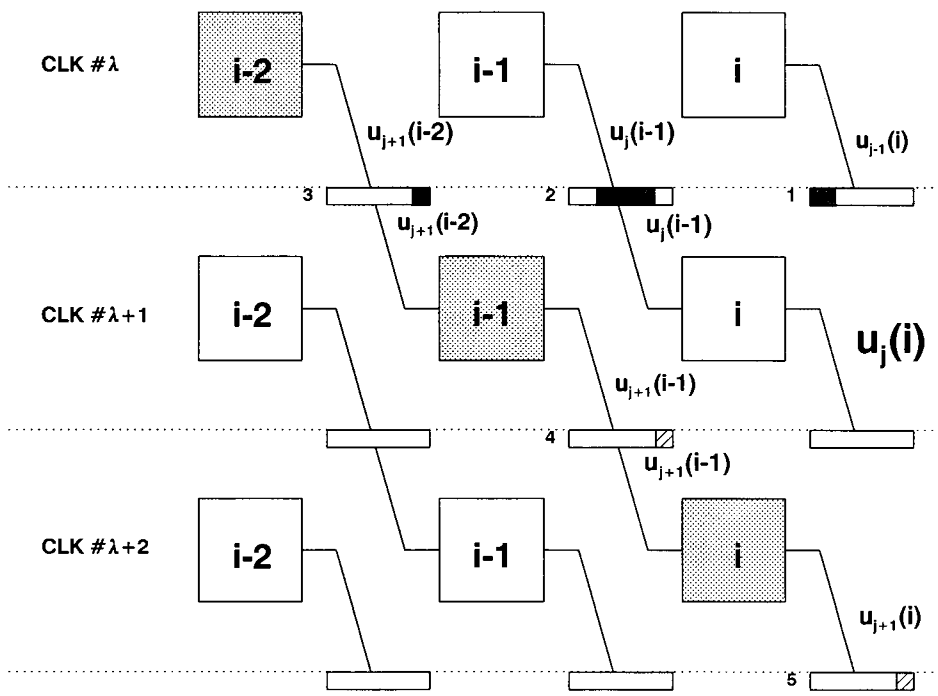


figure 4.6: 'Snapshots' of the systolic array

This figure shows the input and output values of PE nrs. i-2 to i during three clock cycles. So each horizontal row shows what the three consecutive PE's are calculating simultaneously.
The dotted horizontal lines indicate a clock edge on which the input values of the PE input registers (placed on the dotted lines) are loaded. The grey marked PE's indicate the datapath of digit u_{j+1}.

Notice that the sum of the digit index j and the iteration step index i is the same for all output values $u_j(i)$ of the PE's in the systolic array at a certain clock cycle. This sum, which is referred to as *time-index*, increases by one each next clock cycle. The time-indices of figure 4.6 thus are respectively i+j-1, i+j and i+j+1

By expression (40) $u_j(i)$ can be written as binary vector ($\gamma_j(i)$:$t_j(i)$:$\delta_j(i)$). PE #i calculates in clock cycles λ to (λ+2) digits $u_{j-1}(i)$, $u_j(i)$ and $u_{j+1}(i)$. To calculate $u_j(i)$ in clock cycle λ+1, PE #i needs the following input parameters:

- $t_j(i\text{-}1)$: Can be loaded directly from register #2 (the input register of the current PE)
- $m_{i\text{-}1}$: Can be calculated directly using $t_j(i\text{-}1)$ and constant N'
- x_i, y_j and n_j : Can be loaded directly from external memory or internal shift registers
- $\gamma_{j\text{-}1}(i)$: Can be loaded directly from register #1 (the input register of the next PE)
- $\delta_{j+1}(i)$: Cannot be loaded from register #5 until clock cycle λ+3

There seems to be a problem calculating $u_j(i)$, for the required $\delta_{j+1}(i)$ can only be read from register #5 after two clock cycles. However, this $\delta_{j+1}(i)$ can be added in PE's which are placed further in the systolic array (PE numbers i+1 ... k).
PE #i cannot read $\delta_{j+1}(i\text{-}1)$ from register #4 until the next clock cycle λ+2. However, $\delta_{j+1}(i\text{-}2)$ is available in the current clock cycle (λ+1) and can be read directly from register #3 (the input register of the previous PE). So if $\delta_{j+1}(i\text{-}2)$ is used for calculation of $u_j(i)$, all input parameters are available and $t_j(i)$ can be output by PE #i.

In order to find out which modifications to the Montgomery algorithm have to be made to add $\delta_{j+1}(i\text{-}2)$ instead of $\delta_{j+1}(i)$, digit $t_j(i)$ in expression (40) is reduced to $t_j(\text{-}1)$. This is done using a temporary identifier $D(i)$, defined as:

$$D(i) \;=\; 2^{\alpha}x_i y_j + m_{i-1}n_j + 2^{\beta}\delta_{j+1}(i) + 2^{\alpha}\gamma_{j-1}(i) \tag{42}$$

Using this definition and applying integer division, from equation (40) $t_j(i)$ can be written as:

$$\begin{aligned} t_j(i) &= 2^{-\alpha}\left(t_j(i-1) + D(i) \right) \bmod 2^{\beta} \\ &= 2^{-\alpha}\left(2^{-\alpha}\left(t_j(i-2) + D(i-1) \right) + D(i) \right) \bmod 2^{\beta} \\ &= 2^{-\alpha}\left(2^{-\alpha i}t_j(-1) + \sum_{l=0}^{i} D(l)\,2^{\alpha(l-i)} \right) \bmod 2^{\beta} \end{aligned} \tag{43}$$

By the Montgomery algorithm, $T(\text{-}1)$ is set to zero, which implies that $t_j(\text{-}1) = 0$ for $j = 0$ ... l.
Using this property and substituting $D(l)$, we obtain:

$$\begin{aligned} t_j(i) &= 2^{-\alpha}\left(2^{-\alpha i}\sum_{l=0}^{i}\left(2^{\alpha}x_l y_j + m_{l-1}n_j + 2^{\beta}\delta_{j+1}(l) + 2^{\alpha}\gamma_{j-1}(l) \right)2^{\alpha l} \right) \bmod 2^{\beta} \\ &= 2^{-\alpha(i+1)}\left(\sum_{l=0}^{i}\left(2^{\alpha}x_l y_j + m_{l-1}n_j + 2^{\alpha}\gamma_{j-1}(l) \right)2^{\alpha l} + \right. \\ &\qquad\qquad \left. \sum_{l=2}^{i+2} 2^{\beta}\delta_{j+1}(l-2)\;2^{\alpha(l-2)} \right) \bmod 2^{\beta} \end{aligned} \tag{44}$$

The last summation of δ's can be rewritten as:

$$\sum_{l=2}^{i+2} 2^{\beta}\delta_{j+1}(l-2)\, 2^{\alpha(l-2)} = \sum_{l=0}^{i} 2^{\beta-2\alpha}\delta_{j+1}(l-2)\, 2^{\alpha l} + 2^{\alpha i}\left(2^{\beta-\alpha}\delta_{j+1}(i-1) + 2^{\beta}\delta_{j+1}(i) \right) \tag{45}$$

under the condition that both $\delta_{j+1}(-2)$ and $\delta_{j+1}(-1)$ are zero.

This 'rescaling' of δ over two iteration steps results in the following $t_j(\mathrm{i})$:

$$t_j(i) = \Bigg(2^{-\alpha} \sum_{l=0}^{i} \left(2^{\alpha}x_l y_j + m_{l-1}n_j + 2^{\alpha}\gamma_{j-1}(l) + 2^{\beta-2\alpha}\delta_{j+1}(l-2) \right) 2^{\alpha(l-i)} + \left(2^{\beta-2\alpha}\delta_{j+1}(i-1) + 2^{\beta-\alpha}\delta_{j+1}(i) \right) \Bigg) \bmod 2^{\beta} \tag{46}$$

Instead of calculating $t_j(i)$, it is possible to calculate in each PE a digit $v_j(i) < 2^{\beta}$, defined as:

$$v_j(i) = 2^{-\alpha} \sum_{l=0}^{i} \left(2^{\alpha}x_l y_j + m_{l-1}n_j + 2^{\alpha}\gamma_{j-1}(l) + 2^{\beta-2\alpha}\delta_{j+1}(l-2) \right) 2^{\alpha(l-i)} \bmod 2^{\beta} \tag{47}$$

By splitting the expression above in a sum from $l = 0 \ldots i\text{-}1$ and $l = i$ (as in expression (22)), $v_j(i)$ can be calculated recursively by:

$$v_j(i) = \frac{v_j(i-1) + 2^{\alpha}x_i y_j + m_{i-1}n_j + 2^{\alpha}\gamma_{j-1}(i) + 2^{\beta-2\alpha}\delta_{j+1}(i-2)}{2^{\alpha}} \bmod 2^{\beta} \tag{48}$$

In order not to lose any information bits after the integer division by 2^{α}, the lower α bits of the numerator of $v_0(i)$ are set to zero by adding $m_{i-1}N$, which corresponds to adding $m_{i-1}n_j$ to the numerator of each digit $v_j(i)$. Analogous to equation (41), m_{i-1} can be calculated as:

$$m_{i-1} = v_0(i-1)\cdot N' \bmod 2^{\alpha} \tag{49}$$

only if the data bits of $2^{\beta-2\alpha}\ \delta_{j+1}(i\text{-}2)$ are located outside the α least significant bits of the numerator of $v_0(i)$, thus if $2^{\beta-2\alpha} \geq 2^{\alpha}$, or $\beta \geq 3\alpha$. This is a stronger condition than the earlier imposed $\beta \geq \alpha+2$, but it is essential for preventing underflow while calculating digits $v_0(i)$.

If the numerator $w_j(i)$ of the integer division of (48) is defined as:

$$w_j(i) = v_j(i-1) + 2^{\alpha}x_i y_j + m_{i-1}n_j + 2^{\alpha}\gamma_{j-1}(i) + 2^{\beta-2\alpha}\delta_{j+1}(i-2) \tag{50}$$

the digits $v_j(i)$, $\delta_j(i)$ and $\gamma_j(i)$ can be defined as follows:

$$\begin{aligned}\delta_j(i) &= w_j(i) \bmod 2^{\alpha} \\ v_j(i) &= (w_j(i)\, div\, 2^{\alpha}) \bmod 2^{\beta} \\ \gamma_j(i) &= w_j(i)\, div\, 2^{\beta+\alpha}\end{aligned} \tag{51}$$

The expressions (49) to (51) provide a modified Montgomery algorithm which is suitable for execution in a dedicated systolic array, which we call MMM (Montgomery Modular Multiplier). However, the final result needs some δ-correction, for we wish to calculate $t_j(i)$ instead of $v_j(i)$.

4.3 The final delta correction

After the digits $v_j(k)$ have been calculated in the PE's, according to (46) all digits $t_j(k)$ of $T(k)$ = MMM(X,Y) are calculated as

$$t_j(k) = \left(v_j(k) + \left(2^{\beta-2\alpha}\delta_{j+1}(k-1) + 2^{\beta-\alpha}\delta_{j+1}(k) \right) \right) \bmod 2^{\beta} \tag{52}$$

In the systolic array this corresponds to adding the δ's, which are generated in the last two PE's, to the addition result of the last PE. For this purpose two extra (small) PE's are required, which add these δ's at the right place at the proper moment.
As $t_j(k)$ contains only the lower β bits of the addition of (52), the overflow bits should be added to the next (more significant) digit $t_{j+1}(k)$. If this overflow is defined as $\mu_j(k)$, the final δ-correction is executed as shown in figure 4.7.

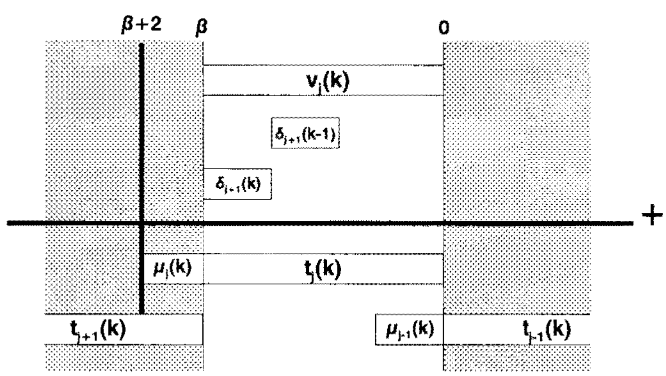


figure 4.7: Final delta-correction

If $\mu_{-1}(k)$ is set to zero, the calculation of $t_j(k)$ can be rewritten as:

$$t_j(k) = \left(v_j(k) + \left(2^{\beta-2\alpha}\delta_{j+1}(k-1) + 2^{\beta-\alpha}\delta_{j+1}(k) \right) + \mu_{j-1}(k) \right) \bmod 2^{\beta} \tag{53}$$

by which the addition overflow is defined as:

$$\mu_j(k) = \left(v_j(k) + \left(2^{\beta-2\alpha}\delta_{j+1}(k-1) + 2^{\beta-\alpha}\delta_{j+1}(k) \right) + \mu_{j-1}(k) \right) div\, 2^{\beta} \tag{54}$$

Using equations (53) and (54), the output digits $v_j(k)$ of the last PE can be corrected by adding the δ-values, generated in the last two PE's. After this correction the desired Montgomery result $t_j(i)$ is obtained.

4.4 Summary of the adapted algorithm for systolic arrays

We have seen that the multiprecision case of Montgomery's algorithm can be executed by a systolic array using large PE's. To reduce the PE size, the *T, Y* and *N* values have been split in digits of β bits, which results in PE's of size α x β bits (indicated as PE(α,β)).
However, digit $t_j(i)$ cannot be calculated directly in a systolic array, for the required $\delta_{j+1}(i)$ is not yet available at the time of calculation. Instead digit $v_j(i)$ is calculated using $\delta_{j+1}(i\text{-}2)$, which is stored in the input register of the previous PE at the time of calculation. This method however requires two extra (small) PE's, which add the last digits $\delta_{j+1}(k\text{-}1)$ and $\delta_{j+1}(k)$ to $v_j(k)$ in order to obtain the desired digit $t_j(k)$.

Figure 4.6 shows, that the required $\gamma_{j-1}(i)$ which is necessary to calculate $v_j(i)$ can be loaded from the input register of the next PE, and the required $\delta_{j+1}(i\text{-}2)$ can be loaded from the input register of the preceding PE. The data flow during the calculation of $v_{j+1}(i\text{-}1)$ and $v_j(i)$ in two consecutive PE's is shown in figure 4.8.

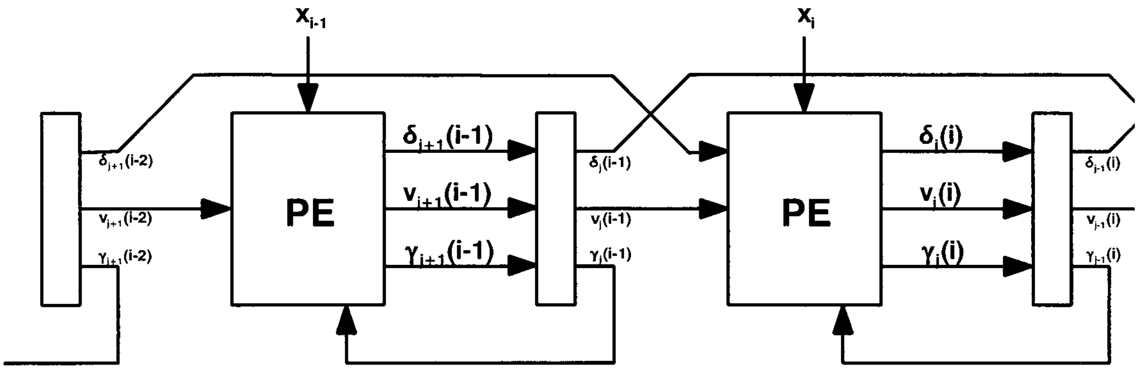


figure 4.8: Register and PE output digits of two consecutive PE's

The Montgomery algorithm which is suitable for PE's of size α x β is as follows :

Montgomery conditions:

1 $n \geq 1$
2 $2^{n-1} < N < 2^n$, N is odd
3 $0 \leq X, Y, T < 2N$
4 $1 \leq \alpha \leq n+2$
5 $k = \lceil (n+2)/\alpha \rceil$
6 $r = k\alpha \geq n+2$
7 $R = 2^r = 2^{k\alpha}$
8 $RR^{-1} - NN' = 1$
9 $3\alpha \leq \beta \leq n$
10 $l = \lceil n/\beta \rceil$
11 $s = l\beta \geq n$

{*input X, Y, N*}
$m_{-1} = 0$
for $i = 0$ **to** k **do**
begin
  $\gamma_{-1}(i) = 0$
  $\delta_{l+1}(i) = 0$

  for $j = 0$ **to** l **do**
  begin
    {*initialize input digits of first PE (i=0)*}
    $v_j(-1) = 0$
    $\delta_{j+1}(-2) = 0$
    $\delta_{j+1}(-1) = 0$

    $w_j(i) = v_j(i-1) + 2^{\alpha} x_i y_j + m_{i-1} n_j + 2^{\alpha}\gamma_{j-1}(i) + 2^{\beta-2\alpha}\delta_{j+1}(i-2)$

    $\delta_j(i) = w_j(i) \bmod 2^{\alpha}$
    $v_j(i) = (w_j(i) \operatorname{div} 2^{\alpha}) \bmod 2^{\beta}$
    $\gamma_j(i) = (w_j(i) \operatorname{div} 2^{\alpha+\beta}) \bmod 2^{\alpha+1}$
  end {for j}
  $m_i = v_0(i) \cdot N' \bmod 2^{\alpha}$
end {for i}

{*execute final delta-correction*}
$\mu_{-1}(k) = 0$
for $j = 0$ **to** l **do**
begin
  $t_j(k) = (v_j(k) + 2^{\beta-2\alpha}\delta_{j+1}(k-1) + 2^{\beta-\alpha}\delta_{j+1}(k) + \mu_{j-1}(k)) \bmod 2^{\beta}$
  $\mu_j(k) = (v_j(k) + 2^{\beta-2\alpha}\delta_{j+1}(k-1) + 2^{\beta-\alpha}\delta_{j+1}(k) + \mu_{j-1}(k)) \operatorname{div} 2^{\beta}$
end {for j}

{$T = T(k) = \Sigma_j\, t_j(k) \cdot 2^{\beta j} = MMM(X,Y) = XYR^{-1} \bmod N$}

5 Hardware Design of the RSA-device

In the previous chapter it has been shown, that the Montgomery algorithm after some transformations can be executed by a dedicated systolic array (MMM) using a number of identical PE's which can process α bits of X and β bits of Y within one clock cycle. The next step is to create a hardware design of a PE which executes the adapted Montgomery algorithm. Then an MMM-design is presented which consists of a cascade of these PE's. Finally an RSA chip design is proposed, which executes an exponentiation algorithm adapted for the MMM.

5.1 Design of a PE

Before proceeding, the choice of α and β is constrained to powers of 2, for this simplifies the hardware implementation of the integer division and multiplications significantly. Therefore, the constraint of $\beta \geq 3\alpha$ implies that $\beta \geq 4\alpha$ when the MMM is implemented in hardware.

To determine the size of the PE register for storage of $\gamma_j(i)$, we need to define an upper bound for digit $w_j(i)$. Because in equation (50) the summation term $2^{\alpha}x_iy_j < 2^{2\alpha+\beta}$ and at least *one* addition carry bit is generated, it is stated that this upper bound is $2^{2\alpha+\beta+1}$.

Proof:
Assume that $w_j(i) < 2^{2\alpha+\beta+1}$, then by expression (51) $\gamma_j(i) < 2^{\alpha+1}$. Then by (50) $w_j(i)$ is bounded by:

$$w_j(i) \;<\; 2^{\beta} + 2^{\alpha}\cdot 2^{\alpha+\beta} + 2^{\alpha+\beta} + 2^{\alpha}\cdot 2^{\alpha+1} + 2^{\beta-2\alpha}\cdot 2^{\alpha} \;=\; 2^{2\alpha+\beta+1}\,(\,2^{-2\alpha-1} + 2^{-1} + 2^{-\alpha-1} + 2^{-\beta} + 2^{-3\alpha-1}\,)$$

To make an upper boundary estimate of the expression between parenthesis, the minimum value $\beta = 4\alpha$ is used:

$$w_j(i) \;<\; 2^{2\alpha+\beta+1}\,(\,2^{-2\alpha-1} + 2^{-1} + 2^{-\alpha-1} + 2^{-4\alpha} + 2^{-3\alpha-1}\,) \;\leq\; 2^{2\alpha+\beta+1} \quad \text{for all } \alpha \geq 1,\ \beta \geq 4\alpha.$$

(the expression between parenthesis equals 1 for $\alpha = 1$).
So the assumption is true, and $\gamma_j(i)$ can be stored in a register of width α+1 bits.

Now expression (54) implies, that $w_j(i)$ is a binary vector of $\delta_j(i)$ (α LSB's), $v_j(i)$ (bits α to $\alpha+\beta-1$), and $\gamma_j(i)$ (α+1 MSB's).

Next to these values an m_i must be calculated, which will be used in the next PE. Equation (49) shows, that m_i can be calculated according to:

$$m_i \;=\; v_0(i)\cdot N' \bmod 2^{\alpha} \qquad \textbf{(55)}$$

This means, that a PE only needs to calculate a new m_i if it is calculating the first digit $w_0(i)$ using x_i, $v_0(i\text{-}1)$, y_0 and n_0, so when the PE is starting a new modular multiplication.
Additionally, because m_i has width α bits, only the *lower* α bits of $v_o(i)$ and constant N' are required for the m_i calculation. The fact that carries propagate away from this m_i makes this Montgomery algorithm superior to the paper & pencil method.

For the calculation of $w_j(i)$ digits y_j and n_j are required, which are loaded from the previous PE and passed on to the next PE each clock cycle. Digit x_i is loaded in PE #*i* (together with $m_{i\text{-}1}$) each time a new modular multiplication is started, and remains there until all digits y_j, n_j and $v_j(i\text{-}1)$ have been loaded and processed in this PE (until $w_l(i)$ has been calculated and a new modular multiplication can be started).

Furthermore it has been shown that the required $\delta_{j+1}(i\text{-}2)$ can be loaded from the input register of the preceding PE, and $\gamma_{j\text{-}1}(i)$ can be loaded from the input register of the next PE.

Using this description, the PE's of the MMM can be outlined as in figure 5.1.

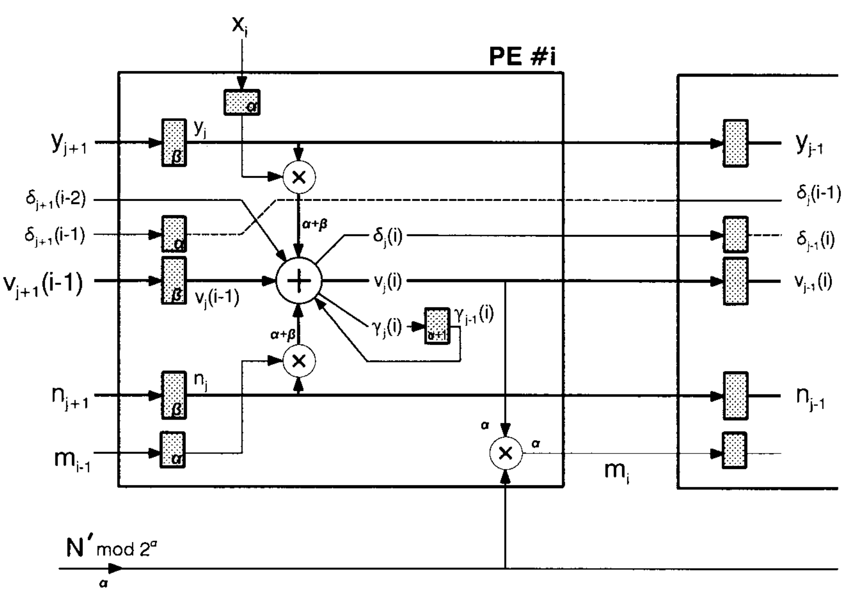


figure 5.1: Outline of the PE's of the MMM

The 5-input adder, which is the core of the PE, adds the input values according to equation (50), which by expression (51) results in the desired $\delta_j(i)$, $v_j(i)$ and $\gamma_j(i)$.
The schematic shows that x_i , $m_{i\text{-}1}$ and $\gamma_{j\text{-}1}(i)$ are local values which belong to PE #*i*. All other values are passed on to the next PE.

5.2 Design of the MMM

As has been shown in the previous chapter, the MMM is a dedicated systolic array, consisting of a number of identical PE's which each execute part of an adapted version of the Montgomery algorithm. Because most exponentiation algorithms are based on repeated modular multiplications, the MMM provides a powerful core for a scalable RSA device. However, the MMM is not entirely compatible with RSA exponentiation because of two reasons:

- Two conversions to the N-residue domain and one conversion back to the integer-domain are required at the start and ending of an exponentiation, for all modular multiplications are executed in the N-residue domain.

- N-residue values have width n+1 bits, while RSA values all have width n bits.

The required conversions have little impact on the exponentiation performance, for exponentiation needs at most 1,5n modular multiplications (see paragraph 2.2). Because RSA security is based on a large value of n (1024 bits), the conversions take about 0.2% of the whole exponentiation time. However, these conversions still need precalculation of the constant R^2 mod N.
The second drawback can be minimized by feeding back bit n+1 of the MMM-result internally, so only digits $t_0(i)$ to $t_{l-1}(i)$ (containing n databits) will be stored in memory for storage of intermediate results. Equation (15) shows that the final conversion back to the integer domain reduces the n+1 bit N-residue value to an n-bit integer value. In this way the user does not have to concern about the Montgomery algorithm, except for providing the constant R^2 mod N.

5.2.1 Delta-correction

Figure 4.8 shows how the PE's of the MMM are mutually connected. This PE interconnection can also be used for the final δ-adjustment, which adds the δ-digits generated in the last two PE's of the MMM to digit $v_j(i)$ (equation (52)). If this addition is split into:

$$\begin{aligned} t_j(k) &= \Big(\big(v_j(k) + 2^{\beta-2\alpha}\delta_{j+1}(k-1) \big) \bmod 2^\beta + 2^{\beta-\alpha}\delta_{j+1}(k) \Big) \bmod 2^\beta \\ &= \big(v_j(k+1) + 2^{\beta-\alpha}\delta_{j+1}(k) \big) \bmod 2^\beta \qquad \textbf{(56)} \\ &= v_j(k+2) \bmod 2^\beta \end{aligned}$$

digits $v_j(k+2)$ and $v_{j+1}(k+1)$ can be calculated by two extra (smaller) PE's in cascade, placed behind the last PE of the MMM. This principle is shown in the schematic of figure 5.2.

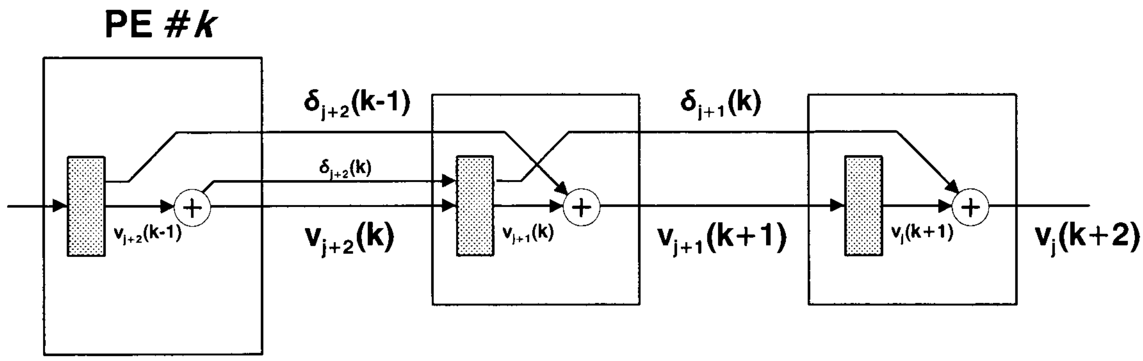


figure 5.2: δ-correction using two small dedicated PE's

Using these two dedicated PE's for δ-correction at the end of the last PE, digits $v_j(k+2)$ are calculated ($j = 0 \dots l$) which by (56) are equal to the Montgomery output digits $t_j(k)$.

5.2.2 Pipelined multiplication in the MMM

If a PE has processed all l+1 digits of one modular multiplication, it is ready and can start a new calculation by loading the first digit $v'_0(i)$ of the next modular multiplication. This pipelining can be illustrated by figure 5.3, which shows the transition of two consecutive multiplications.

In this figure the grey PE's are calculating digits of the first modular multiplication, the white PE's are calculating the next multiplication. If all digits y_j, n_j and $v_j(-1)$ ($j = 0 \dots l$) have been loaded in a PE of the MMM, this PE is ready and can start loading the first digits of the next modular multiplication.

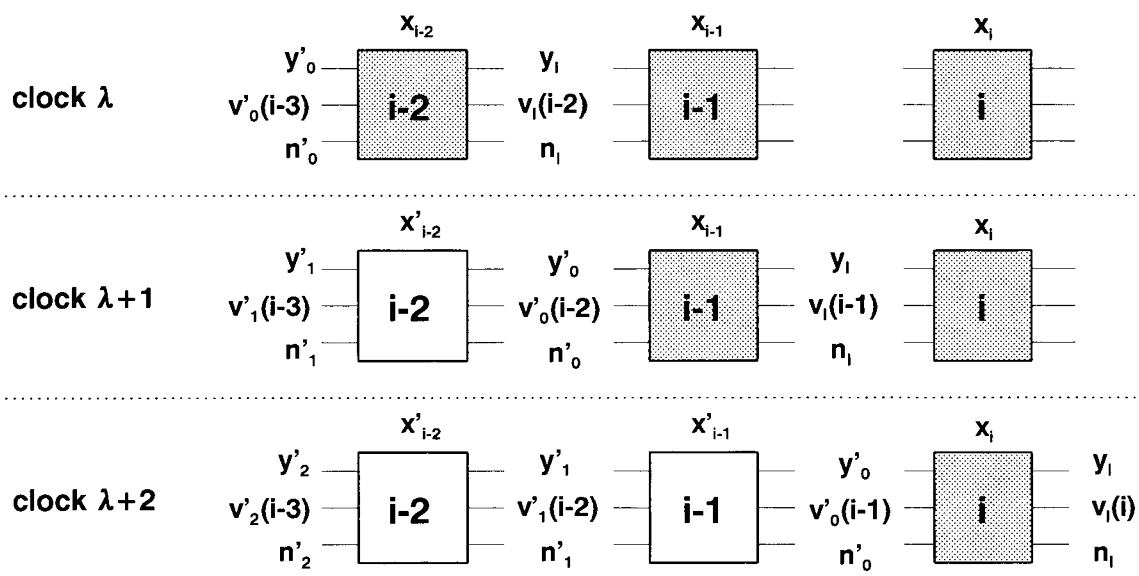


figure 5.3: Data flow of two consecutive modular multiplications in the MMM

Two consecutive multiplications in the MMM do not mutually interfere:

- In figure 5.3, in clock cycle $\lambda+2$ PE #i calculates the last digit $v_l(i)$ of the first multiplication, and should add $\delta_{l+1}(i\text{-}2)$, which is zero by definition (there are only l digits of $\delta_j(i)$). However, instead $\delta'_0(i\text{-}2)$ which has been calculated in PE #i-2 in clock cycle $\lambda+1$ (and loaded in PE #i-1 in clock cycle $\lambda+2$) is added! This is allowed, for all $\delta_0(i)$ digits which are calculated starting a new modular multiplication are zero. The proof for this is as follows:
  By equations (51) and (50) and $\beta \geq 4\alpha$ the following applies:

$$\begin{aligned}\delta_0(i) &= w_j(i) \bmod 2^\alpha = \\ &= \left(v_0(i-1) + 2^\alpha x_i y_0 + m_{i-1}n_0 + 2^\alpha \gamma_{-1}(i) + 2^{\beta-2\alpha}\delta_1(i-2) \right) \bmod 2^\alpha \\ &= \left(v_0(i-1) + m_{i-1}n_0 \right) \bmod 2^\alpha\end{aligned} \quad \textbf{(57)}$$

  Equation (49) shows that:

$$m_{i-1}n_0 = v_0(i-1)\cdot N'n_0 \bmod 2^\alpha \quad \textbf{(58)}$$

  Using this equation and that $N'n_0 \bmod 2^\alpha = -1$ (see paragraph 6.4.1), $\delta_0(i)$ can be written as:

$$\delta_0(i) = \left(v_0(i-1) + (\, v_0(i-1)N'n_0 \,) \bmod 2^\alpha \right) \bmod 2^\alpha = 0 \quad \textbf{(59)}$$

  which shows that all digits $\delta_0(i)$ are zero and will not interfere with the calculation of the last digit of the preceding modular multiplication.

- The PE which calculates the last digit of the preceding multiplication, stores the overflow bits in the γ-register ($\alpha+1$ bits). The next clock cycle, when this PE starts to calculate the first digit of the next modular multiplication, the contents of this γ-register will be added. Therefore digit $\gamma_l(i)$ must be zero (for all $i = 0 \ldots k$) in order not to interfere with the next multiplication.
  This can be shown using equation (50):

$$\begin{aligned}w_l(i) &= v_l(i-1) + 2^\alpha x_i y_l + m_{i-1}n_l + 2^\alpha \gamma_{l-1}(i) + 2^{\beta-2\alpha}\delta_{l+1}(i-2) \\ &= v_l(i-1) + 2^\alpha x_i y_l + 2^\alpha \gamma_{l-1}(i)\end{aligned} \quad \textbf{(60)}$$

  Because digits $y_0 \ldots y_{l-1}$ contain *at least* n bits of Y ($l = \lceil n/\beta \rceil$), digit y_l can contain at most bit $n+1$, which implies that $2^\alpha x_i y_l < 2^{2\alpha}$. Now $w_l(i)$ can be bounded by:

$$w_l(i) < 2^\beta + 2^{2\alpha} + 2^{2\alpha+1} < 2^{\beta+1} \quad \textbf{(61)}$$

  By expression (51), $\gamma_l(i) = w_l(i) \text{ div } 2^{\beta+\alpha} = 0$ for all $i = 0 \ldots k$, which proves that the γ-register only contains zeroes when the PE starts a new modular multiplication.

So because both $\delta_0(i)$ and $\gamma_l(i)$ are zero for each i, two multiplications in the MMM can be executed after each other without interference. However, each PE which starts a new modular multiplication must be initialized by loading the new x_i and m_{i-1} digits and resetting the γ-register (contents are undefined after a chip reset, and must be set to zero).

5.2.3 Reducing the number of PE's

For purpose of scalability it is desired to change the number of PE's to optimally fit the environment in which the RSA device is used. We have seen before:

- $k = \lceil (n+2)/\alpha \rceil$: There are k+1 digits of X to be loaded in the MMM, so k+1 PE's are used during one modular multiplication.
- $l = \lceil n/\beta \rceil$: There are l+1 digits of Y, T and N to be loaded in the MMM, so each PE must process l+1 digits during one modular multiplication.

Because $\beta \geq 4\alpha$, $k > l$. This means, that there are more PE's to be used (X-digits to be loaded) than Y, N and T digits to be loaded by each PE.
Choosing $p = k+1$ provides just enough PE's to load all X-digits, but because l+1 (the number of digits running through the MMM) is smaller than k+1, there are p - (l+1) PE's not active. These non-active PE's cannot already start a new multiplication, for this requires the output digits of the current multiplication.

In order to keep all PE's of the MMM active all the time, at most l+1 PE's should be used and the intermediate results are fed back into the first PE internally. In this case data would run through the same PE multiple times (MMM cycles) during one modular multiplication, each time processing a new digit of X. Because the MMM must be able to contain all intermediate results (all $v_j(i)$'s), a FIFO buffer of depth (l+1) - p is required (l+1 digits, of which p are stored in the PE registers). This would yield an MMM which looks like figure 5.4.

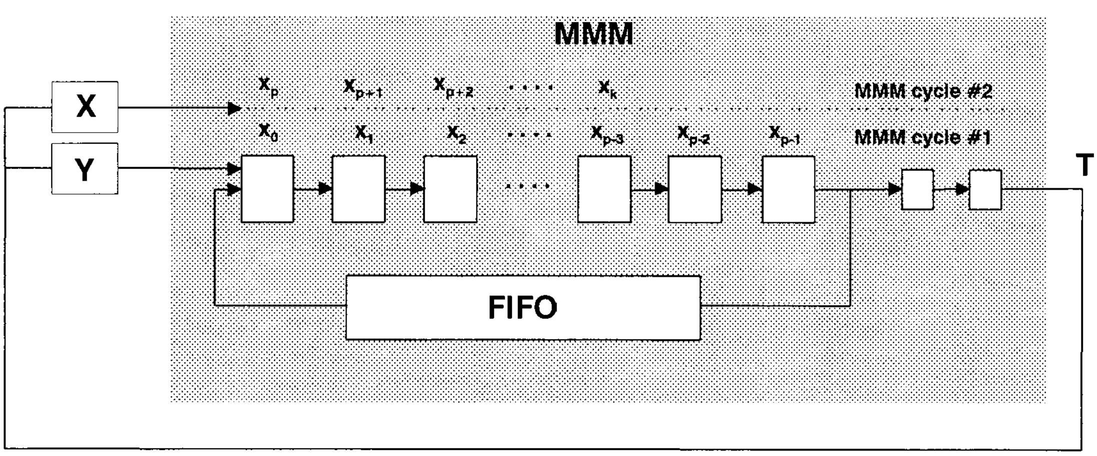


figure 5.4: MMM configuration with internal feedback

The first MMM cycle digits x_0 ... x_{p-1} are loaded in the PE's. The first PE in the MMM is ready for a new MMM cycle if it has calculated the last digit $v_l(0)$. At that time the next digit x_p can be loaded to calculate $v_0(p)$, which starts the second MMM cycle. One modular multiplication requires $\lceil (k+1)/p \rceil$ MMM cycles. As we have seen before, in two successive PE's the first digit of a new MMM cycle does not affect the calculation of the last digit of the preceding MMM cycle.
All digits $v_j(p\text{-}1)$ which leave the last PE of the MMM while the first PE is not ready yet ($p < l+1$) are stored in the FIFO buffer until the second MMM cycle can start. Because PE #i needs both $\delta_{j+1}(i\text{-}2)$ and $\delta_{j+1}(i\text{-}1)$ and the intermediate result $v_j(i)$ from the preceding PE, all must be stored in the FIFO. This FIFO therefore will have $(l+1) - p$ levels of width $\beta+2\alpha$ bits.

When the last MMM cycle (in which $v_j(k)$ is calculated) has been completed, after δ-correction the final digits $t_j(k)$ with $j = 0$ ... $l\text{-}1$ are written to external memory and input to the MMM for the next modular multiplication. Digit $t_l(k)$, which contains at most bit $n+1$ of T (if n is a multiple of β, see figure 4.2) can be stored internally and fed back to one of the MMM inputs (multiply) or both inputs (squaring) for the next modular multiplication of the exponentiation algorithm. In this way in the external memory at most $s = l\beta$ bits (s is the smallest multiple of β larger than n, $s < n+\beta$) have to be stored, and the user will not be confronted with extra memory space for storage of the overflow digit $t_l(k)$ of the Montgomery algorithm.
A new modular multiplication can start as soon as the first PE of the MMM has calculated it's last digit in the last MMM cycle, so when the FIFO is empty.

Because the number of PE's p can be chosen arbitrarily, it is possible that the final $v_j(k)$ digits are calculated by a PE in the middle of the MMM. All next PE's must then be set in a 'bypass mode', which forces them to pass on the input result to the output without any modification.
However, the δ-correction needs not only $v_j(k)$, but also the digits $\delta_j(k\text{-}1)$ and $\delta_j(k)$, which must also be passed on by the 'bypass' PE's. Because each $\delta_j(i\text{-}2)$ is input to the next PE directly (not loaded in an input register), PE's in bypass mode need to store this δ digit in an extra register of width α bits. Also, multiplexers are required in each PE to select the input digits or calculated digits for output.

There is however a way to ensure that $v_j(k)$ will always be calculated by the last PE of the MMM. In that case $v_j(k)$ and matching δ's are directly input in the δ-correction logic, and no bypass mode is required. In equation (16) we have chosen k, the number of X-digits, such that $r = k\alpha$ is the smallest multiple of α larger than $n+2$. If $k+1$ is chosen to be a multiple of p, $v_j(k)$ will always be calculated by the last PE. Although this can increase the number of X-digits considerably, no extra external memory is required, because all extra X-digits are zero and can be generated in the MMM. Using $k+1$ digits with $k+1$ is a multiple of p, all PE's which originally were in the bypass mode are now calculating an iteration step of the Montgomery algorithm with a zero on the x_i-input. Therefore the multiplication time will not change using this method. Notice that choosing $k+1$ as a multiple of p, $R = 2^r = 2^{k\alpha}$ will become larger, which can lengthen the calculation of R^2 mod N.

5.2.4 Control of the MMM

Because data flow in the MMM is constant, the MMM control has little complexity. It mainly consists of comparison logic and two index counters i and j, which address digits of x_i respectively y_j and n_j (stored outside the MMM in external memory), and x_k and y_k (generated internally). Therefore, index counter i counts upwards from 0 to k and j counts upwards from 0 to l.
If index counter j = 0, PE #0 loads digits y_0 , n_0 and v_0 , so PE #0 must be initialized (reset γ-register, load x_i and m_{i-1} registers) . The next clock cycle j = 1 and PE #1 needs initialization. For this purpose all PE's have an address decoder, which forces a PE to be initialized when it is addressed by index counter j. If however $j \geq p$, a non-existing PE is addressed so no x_i digit can be loaded. In that case the i counter must hold it's current value until PE #0 is addressed again (new MMM cycle).
If $j = l$, an input multiplexer must select the internally stored overflow digit $t_l(k)$ of the previous multiplication (there is no digit y_l stored outside the MMM) and place it on the y-input of the first PE, together with digit v_l (internal overflow digit) and n_l , which is zero. The next clock cycle the j register can be reset to zero, which starts a new MMM cycle processing the next series of X-digits.
If $i = k$, the zero-digit x_k is loaded in the last PE of the MMM (if k+1 is a multiple of p). Also this digit must be selected by a multiplexer, for it is not stored in the external memory.

Because it is desired to store only l digits of with β in the external memory, all overflow bits caused by the Montgomery algorithm should be processed in the MMM internally.
The processing of overflow digit $t_l(k)$ on the X-data input depends on the values $r = k\alpha$, $s = l\beta$, the cryption width n and the size of α. There are two situations which should be treated seperately:

- $n = l\beta$: This implies that bit n+1 of $T(k)$ is located at the LSB of overflow digit $t_l(k)$, as indicated in figure 5.5.

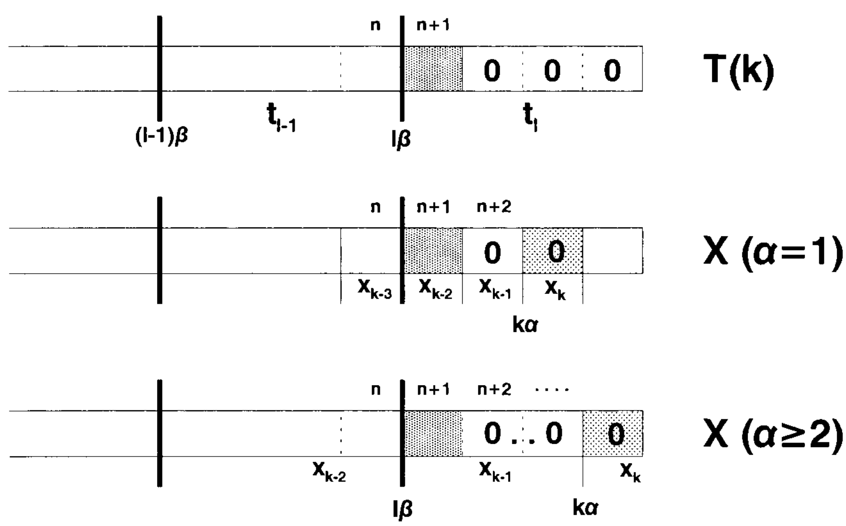


figure 5.5: Location of bit n+1 in overflow digit t_l if $n = l\beta$

If N-residue value $T(k)$ is loaded in the MMM on the X-data input while $n = l\beta$, there are two situations that can occur:

- $\alpha = 1$: Digits x_0 .. x_{k-3} are loaded from the external memory (which stores l digits of width β). Three digits x_{k-2} ... x_k must be concatenated internally, of which digit x_{k-2} must contain bit n+1 (stored in the LSB of the internal overflow register), and x_{k-1} and x_k must be zero.
- $\alpha \geq 2$: Digits x_0 .. x_{k-2} are loaded from the external memory. Two digits x_{k-1} and x_k must be concatenated internally. The LSB of digit x_{k-1} must contain bit n+1 (stored in the LSB of the internal overflow register), and x_k must be zero.

- $n < l\beta$: This implies that bit n+1 of $T(k)$ is stored in the external memory, and overflow digit $t_l(k)$ is always zero, as indicated in figure 5.6.

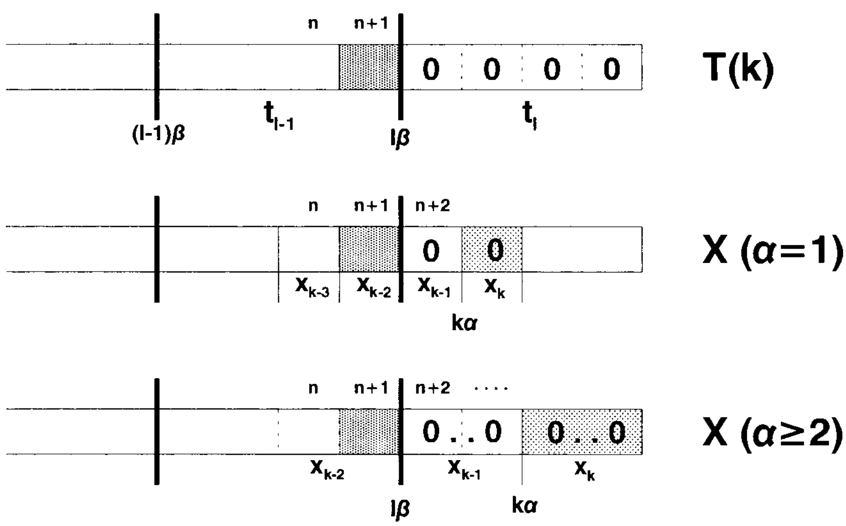


figure 5.6: *Location of bit n+1 if n < lβ*

Also in this case two situations can be distinguished:

- $k\alpha > l\beta$: Digits x_0 .. x_{k-2} are loaded from the external memory. Two digits x_{k-1} and x_k must be concatenated internally. The LSB of digit x_{k-1} (which represents bit n+2 of X) is always zero, so the LSB of $t_l(k)$ (which is always zero if $n < l\beta$) can be placed here.
- $k\alpha \leq l\beta$: Digits x_0 .. x_{k-1} are loaded from the external memory (containing all n+2 bits of X). One digit x_k is concatenated internally. The LSB of digit x_k is always zero, so the LSB of $t_l(k)$ (which is always zero if $n < l\beta$) can be placed here.

In all cases zero-digit x_k must be generated internally. In some cases extra X-digits x_{k-2} or x_{k-1} must be concatenated internally, for then these cannot be stored in the external memory.
All cases however show that the LSB of the (internally stored) overflow digit $t_l(k)$ can be placed at the LSB of the first internally generated X-digit digit which is concatenated to the last X-digit loaded externally: It is of no concern whether this LSB is really bit n+1 of the Montgomery result.

To make a selection between external stored digits and internal digits, two input multiplexers *Xmux* and *Ymux* are placed in the MMM. *Xmux* adds digits x_{k-2} , x_{k-1} and/or x_k , possibly added with bit n+1 if $n = l\beta$. *Ymux* concatenates digits y_l and n_l (which is zero) to y_{l-1} and n_{l-1} . Digit y_l contains the overflow digit $t_j(k)$ of the previous modular multiplication, which is fed back internally. *Ymux* also generates zero-digits to fill all PE's if there are more PE's than digits of Y and N.

Because the initial digits $v_j(-1)$, $\delta_{j+1}(-2)$ and $\delta_{j+1}(-1)$ are zero, the FIFO must provide zero-digits to be loaded in the first PE *only* during the first MMM-cycle. The next MMM cycle the digits stored in the FIFO are loaded in the first PE.

Using the four described situations, the MMM control functions can be described as listed in appendix A. This control description of the MMM provides a simple control structure, which is primarily based on comparison of counters and constants.

Notice that if l+1 < p, the MMM is not completely filled and the j-counter will not address all PE's. Therefore the number of digits is extended to p using overflow digit $y_l = t_l(k)$ and zero digits y_{l+1} to y_{p-1} and n_l to n_{p-1}. . The j-counter will not be resetted until j = p-1, so if all PE's have been addressed and have loaded a digit of X.

5.3 Design of the RSA processor

Using the MMM core which has been described in the previous paragraph, an RSA-exponentiation can be executed using repeated modular multiplications according to the Montgomery algorithm. For this purpose the multiplicands must be converted to the N-residue domain, and the final result should be converted back to the integer domain. Now the exponentiation algorithm becomes:

{input M, e, N, n, $R_N = R^2 \bmod N$}

C' := MMM(1, R_N) = $R \bmod N$
M' := MMM(M, R_N)

for $i = (n\text{-}1)$ **downto** 0 **do**
begin
  if e_i=1 **then** C := MMM(C', M')
  if i>0 **then** C := MMM(C', C')
end

C := MMM(C', 1)
if $C = N$ **then** $C = 0$

{output $C = M^e \bmod N$}

Also this algorithm may skip all succeeding most significant '0' bits, for then only the initial $C' = R \bmod N$ is squared, which does not change this initial value:

$$MMM(R\bmod N, R\bmod N) = (R^2 \bmod N)R^{-1} \bmod N = R \bmod N \quad \textbf{(62)}$$

This algorithm needs storage of M, N, e, C and $R_N = R^2 \bmod N$. Using the MMM as multiplication core, the RSA device can be modelled as in figure 5.7.

In this figure the exponent e is loaded bit by bit in the chip control block, which executes the exponentiation algorithm as described above. The constant $R^2 \bmod N$ must be provided by the user and will be stored in the R_N momory until a new modulus N is required. Using this constant, the originally loaded message M is converted to the N-residue domain, and the result M' is written back in the M-memory. The C memory is used for storage of intermediate exponentiation results. The initial C = '1', which is used by the exponentiation algorithm, can be generated internally in the MMM, so this '1' does not have to be written in the C memory by the user. The internal generation of this '1' can also be used for transformation of the final C' value back to the integer domain.

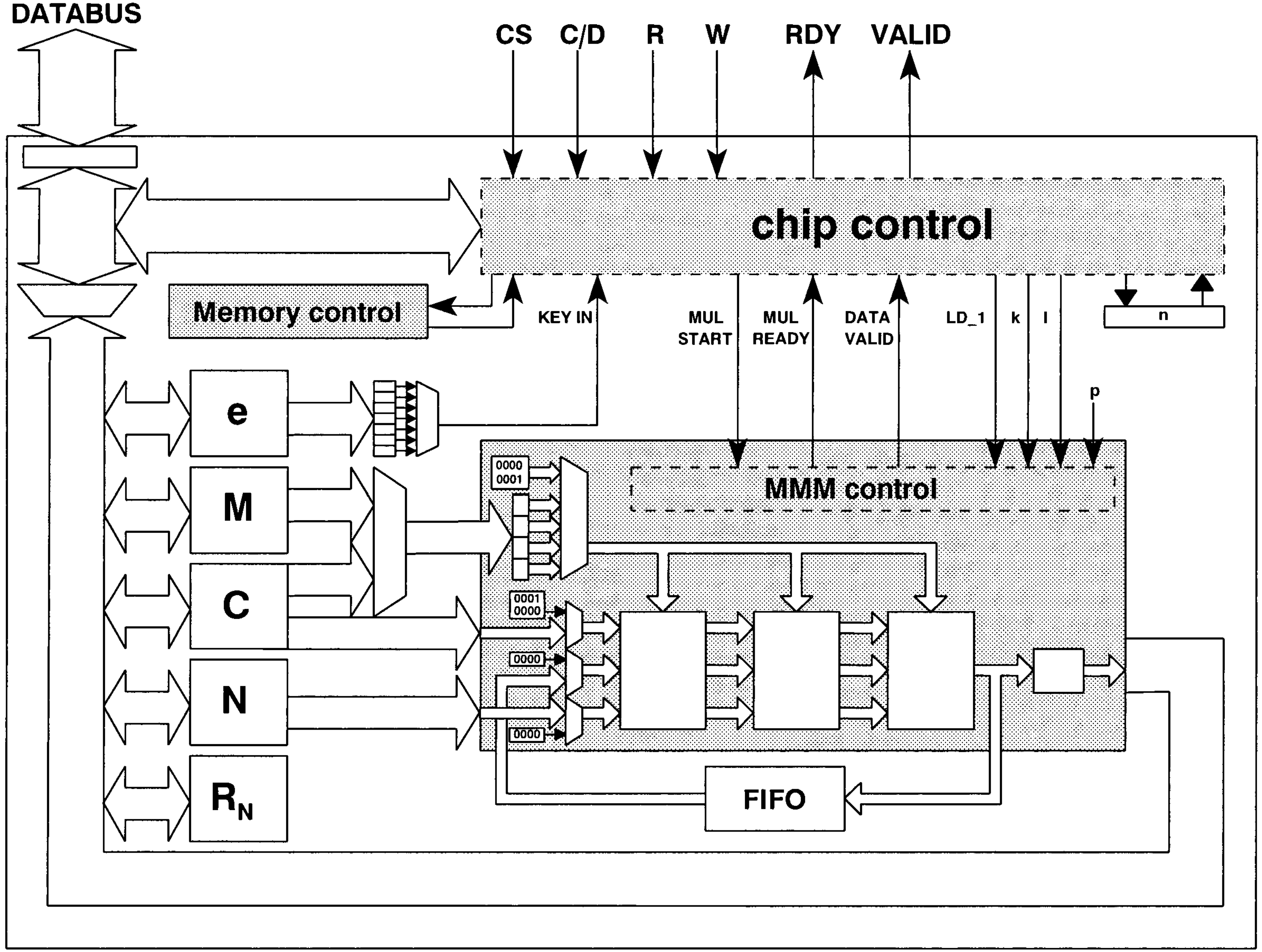


figure 5.7: Schematic of the scalable RSA device using the MMM-core

The multiplexer in front of the MMM selects if a multiplication MMM(C', M') or a squaring MMM(C', C') is to be executed by the MMM. The input multiplexers inside the MMM select when external digits are loaded or when internal digits are generated.

The digits of width β bits, stored in the M and C memories, are converted to digits of α bits using the input multiplexer inside the MMM.

For security reasons it is essential that the internal bus, which has width β bits, is seperated from the external databus, for no (intermediate) results or memory contents may be read during exponentiation.

6 Performance of the RSA core

6.1 Number of clock cycles of an MMM

We have seen that each PE of the MMM processes l+1 digits, and that k+1 PE's are used for one modular multiplication. The number of clock cycles of a Montgomery modular multiplication can be determined using :

- One MMM cycle takes (l+1) clock cycles (or p clock cycles if $p > l+1$)
- It takes $\lceil (k+1)/p \rceil$ MMM cycles to calculate a modular multiplication

This means, that the first PE of the MMM is available after $(l+1) \cdot \lceil (k+1)/p \rceil$ clock cycles. It takes another (p-1) + 2 clock cycles before the last digit $t_l(k)$ leaves the MMM, but due to pipelining (starting the next modular multiplication in the first PE's while the last PE's are calculating the preceding multiplication), these extra clock cycles will only be evident after the last multiplication of the exponentiation has been calculated.

It has been shown in paragraph 2.2, that at most 1.5n modular multiplications are required for an exponentiation. Because RSA uses large values of n, the three MMM's required for conversions are ignored. Now the number of clock cycles G_n can be defined as:

$$\begin{aligned} G_n &= 1.5n \cdot (l+1) \cdot \left\lceil \frac{k+1}{p} \right\rceil + (p+1) \\ &= 1.5n \cdot \left(\left\lceil \frac{n}{\beta} \right\rceil + 1 \right) \cdot \left\lceil \frac{\left\lceil \frac{n+2}{\alpha} \right\rceil + 1}{p} \right\rceil + (p+1) \end{aligned} \tag{63}$$

For reasons of simplicity this is approximated by

$$G_n(\alpha,\beta) = 1.5n \cdot \left(\frac{n+\beta}{\beta} \right) \cdot \left(\frac{n+\alpha}{\alpha} \right) \cdot \frac{1}{p} \tag{64}$$

So an MMM containing PE's of size α x β requires approximately $G_n(\alpha,\beta)$ clock cycles to execute an RSA exponentiation of width n bits. If the maximum clock frequency of such a PE is defined by f(α,β), The number of n-bits RSA cryptions per second can be defined as:

$$E_n(\alpha,\beta) = \frac{f(\alpha,\beta)}{G_n(\alpha,\beta)} = \frac{\alpha\beta \cdot f(\alpha,\beta) \cdot p}{1.5n(n+\alpha)(n+\beta)} \tag{65}$$

If p is equal to the numer of gates of the MMM divided by the number of gates of a PE(α,β), we can define the performance index Pi(α,β) as:

$$Pi(\alpha,\beta) = \frac{\alpha\beta \cdot f(\alpha,\beta)}{\#gates\,(PE(\alpha,\beta))} \cdot 10^{-6} \tag{66}$$

This performance index can be interpreted as the maximum speed of a PE per unit of area.

Now the number of RSA cryptions per second equals:

$$E_n(\alpha,\beta) = Pi(\alpha,\beta) \cdot \frac{\#gates(MMM)}{1.5n(n+\alpha)(n+\beta)} \cdot 10^{6} \tag{67}$$

under the condition that $p = \{\ \#gates(MMM)\ /\ \#gates(PE((\alpha,\beta))\ \} \leq l+1$ (more than $l+1$ PE's will not improve the performance of the MMM). Using the performance index $Pi(\alpha,\beta)$, the performance of PE's of different sizes can be compared.

6.2 Performance of PE type 1

Using the PE schematic of figure 5.1, the datapath from the PE input registers to the PE output can be modelled as in figure 6.1. This PE, which has not been optimized involving hardware, is called PE type 1.

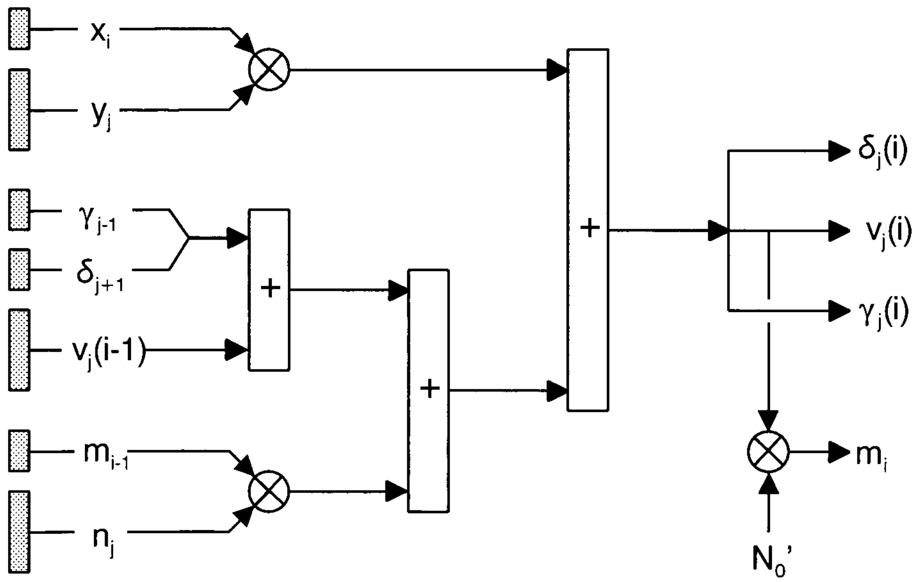


figure 6.1: Schematic of the datapath in PE type 1

This figure shows that the critical path contains at least an α x β multiplier and two adders. Because m_i is calculated using the lower α bits of $v_j(i)$ and the last adder has width $\alpha+\beta$ ($\beta \geq 4\alpha$), it is likely that m_i is calculated before the carry of the last adder has rippled to the MSB of $\gamma_j(i)$.

The $\delta_{j+1}(i\text{-}2)$ digit is directly loaded from the input register of the preceding PE. Digit $\gamma_j(i)$ is fed back to the γ input-register. The addition of $2^{\beta-2\alpha}\delta_{j+1}(i\text{-}2)$ and $2^{\alpha}\gamma_{j-1}(i)$ is only required if $\beta=4\alpha$, for then the MSB of γ will overlap the LSB of the δ-digit. Then a ripple-carry adder of width α bits is required. If $\beta > 4\alpha$ there is no overlap and δ and γ can be treated as one digit.

This PE has been described in VHDL and compiled to a hardware design for several values of α and β. The PE has been compiled using the ES2 0.5µ library, using standard components.
No use has been made of scanpath registers, because for security reasons the contents of internal registers may not be read during or after a calculation. An other possibility to test the MMM core is to execute a number of exponentiations. Statistical analysis must indicate the fault coverage of this testing method.

The compiler results are shown in table 6.1. Of each PE(α,β) is indicated the number of gates, the maximum clock frequeny, and the performance index Pi(α,β). If the working frequency of a PE is halved, also Pi(α,β) will be reduced by a factor 2.

***Table 6.1**: Performance indices of PE type 1*

		β			
		4	8	16	32
α	1	225 83 MHz **1.48**	292 58 MHz **1.60**	858 57 MHz **1.06**	1556 42 MHz **0.86**
	2		601 61 MHz **1.62**	1053 39 MHz **1.19**	2128 23 MHz **0.69**
	4			1858 32 MHz **1.10**	3895 24 MHz **0.79**
	8				5632 20 MHz **0.91**

This table shows that an MMM using PE(2,8) or PE(1,8) can achieve the largest number of cryptions per second. For example, 1024 bits RSA cryption requires:

- PE(2,8): p = 128, #gates(MMM) = 77 Kgates. $G_{1024}(2,8)$ = 76 cryptions/second.
- PE(1,8): p = 128, #gates(MMM) = 37 Kgates. $G_{1024}(1,8)$ = 37 cryptions/second.
- PE(8,32):p=1, #gates(MMM) = 5632 Kgates (without FIFO). $G_{1024}(8,32)$ = 3.4 cryptions/second

Notice that these figures are best-case indications, which do not take into account wire load or extra output buffers, which most likely are required when connecting many devices in cascade.

6.3 Performance of PE type 2

Because PE type 1 has a multiplier and two cascaded adders in the critical path, PE performance stays low due to the large carry-ripples. The carry ripple in the last two adder stages can be eliminated by replacing them by a three-input carry-save adder. A carry save adder consists of a number of full adders, of which the carry-input is used as data-input, and the carry-output is part of the addition result, which is represented redundantly using S° (XOR result) and $S^\wedge$ (carry out). Because also the generated carries must be stored, this notation requires double register space.

The result can be converted back to an integer using a ripple-carry adder which calculates $S^\circ + 2S^\wedge$. In figure 6.2 a carry-save adder is shown.

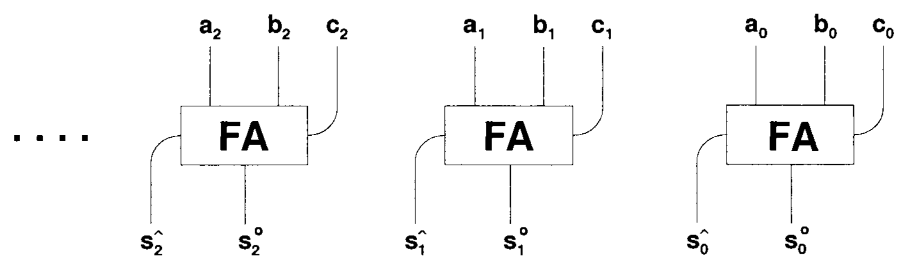


***figure 6.2**: Schematic of a carry-save adder*

A great advantage is that this adder does not have a carry ripple. Instead, all generated carries are stored with the addition result, by which the carry propagation can be postponed.

If the last two adder stages of PE type 1 are replaced by a carry-save adder, $\delta_j(i)$, $v_j(i)$, and $\gamma_j(i)$ are represented redundantly. Because the extra carry bit generated in the most significant full-adder stage is stored in the MSB of $\gamma_j^\wedge(i)$, both $\gamma_j^\wedge(i)$ and $\gamma_j^\circ(i)$ have width α bits. So extra register space of total length $2\alpha+\beta$ bits is required. Also, the addition of $2^{\beta-2\alpha}\delta_{j+1}^\wedge(i-2)$ and $2^{\alpha}\gamma_{j-1}^\wedge(i)$ will no longer be required if $\beta = 4\alpha$, for no overlap will occur. The same goes for δ° and γ°.

The m_i calculation needs the integer representation of the lower α bits of $v_0(i)$. Therefore, these must be converted using a ripple-carry addition of $(w_j^\circ(i) \bmod 2^{2\alpha})$ and $(2 \cdot w_j^\wedge(i) \bmod 2^{2\alpha})$, which has width 2α bits.

The conversion of the complete result back to integers is executed at the beginning of the next PE, where the carry can ripple during the multiplications x_iy_j and $m_{i-1}n_j$. PE type 2 is shown in figure 6.3.

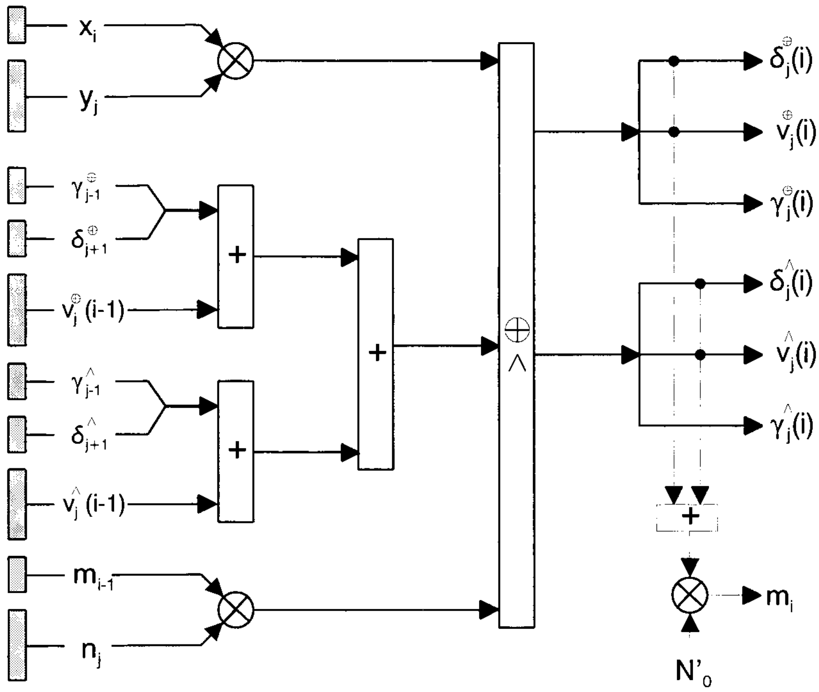


figure 6.3: Schematic of the datapath in PE type 2

Also this PE has been described in VHDL and compiled to hardware using the ES2 0.5μ library and standard components without scanpath registers. The compiler results are shown in table 6.2.

***Table 6.2**: Performance indices of PE type 2*

		β			
		4	8	16	32
α	1	210 88 MHz **1.68**	387 68 MHz **1.41**	1030 76 MHz **1.18**	2140 60 MHz **0.90**
	2		531 66 MHz **1.99**	1120 42 MHz **1.20**	2821 57 MHz **1.29**
	4			2102 45 MHz **1.37**	4442 39 MHz **1.12**
	8				5952 27 MHz **1.17**

This table shows that an MMM using PE(2,8) or PE(1,8) can achieve the largest number of cryptions per second. For example, 1024 bits RSA cryption requires:

- PE(2,8): $p = 128$, #gates(MMM) = 68 Kgates. $G_{1024}(2,8) = 83$ cryptions/second.
- PE(8,32): $p = 1$, #gates(MMM) = 6 Kgates (without FIFO). $G_{1024}(8,32) = 4.1$ cryptions/second

Again, these are best-case estimates of the overall-performance of the MMM.

6.4 Optimization of the PE's

There are some more possibilities for optimizing the hardware design of the PE, which in general all apply to reduction of the adder depth and carry propagation, or easier calculation of m_i.

6.4.1 Optimization of the m_i calculation

Equation (49) shows, that m_i can be calculated by:

$$m_i = v_0(i) \cdot N' \bmod 2^\alpha \tag{68}$$

For this purpose the calculation of N' is required, which is defined by the Montgomery algorithm:

$$RR^{-1} - NN' = 1 \tag{69}$$

Because only the lower α bits of N' are required, only the lower α bits if this comparison are used:

$$(RR^{-1} - NN') \bmod 2^\alpha = (RR^{-1} \bmod 2^\alpha - NN' \bmod 2^\alpha) \bmod 2^\alpha = 1 \tag{70}$$

Because $R = 2^r$, $r \geq n + 2 > \alpha$, $RR^{-1} \bmod 2^\alpha$ is zero.
Further, because $\beta \geq 4\alpha$, $N \bmod 2^\alpha = n_0 \bmod 2^\alpha$. Now equation (70) becomes:

$$-n_0 N' \bmod 2^\alpha = 1 \tag{71}$$

If the negative product is written in two's complement notation, this becomes:

$$(1 + \overline{n_0 N'}) \bmod 2^\alpha = 1 \bmod 2^\alpha \tag{72}$$

Which implies:

$$\overline{n_0 N'} \bmod 2^\alpha = 0 \Rightarrow n_0 N' = (11\ldots11)_\alpha \tag{73}$$

So each bit of the product n_0N' must yield a binary '1'.

If both product terms are represented binary as:

$n_0 \bmod 2^\alpha = (n_{\alpha-1}\ n_{\alpha-2} \ldots n_0)_2$
$N' \bmod 2^\alpha = (n'_{\alpha-1}\ n'_{\alpha-2} \ldots n'_0)_2$

The product $n_0N' \bmod 2^\alpha$, $\alpha \leq 4$ can be calculated using a 'paper & pencil' method:

$$
\begin{array}{cccc l}
n_3 & n_2 & n_1 & n_0 & \\
n'_3 & n'_2 & n'_1 & n'_0 & \\
 & & & & \times \\
\hline
n'_0n_3 & n'_0n_2 & n'_0n_1 & n'_0n_0 & \\
n'_1n_2 & n'_1n_1 & n'_1n_0 & & \\
n'_2n_1 & n'_2n_0 & & & \\
n'_3n_0 & & & & \\
 & & & & + \\
\hline
1 & 1 & 1 & 1 &
\end{array}
$$

Taking the carry bits into account, which are generated during the addition and should be added to the next bit, each product bit can be determined seperately:

$n'_0n_0 = 1 \qquad \Rightarrow n'_0 = 1 \qquad$ (N is odd)

$n'_0n_1 \oplus n'_1n_0 = 1 \qquad \Rightarrow n_1 \oplus n'_1 = 1 \qquad \Rightarrow n'_1 = \overline{n}_1$

$n'_0n_2 \oplus n'_1n_1 \oplus n'_2n_0 \oplus (n'_1n_0 \wedge n'_0n_1) = n_2 \oplus n_1\overline{n}_1 \oplus n'_2 \oplus (n_1 \wedge \overline{n}_1) = 1$

$\Rightarrow n_2 \oplus n'_2 = 1 \qquad \Rightarrow n'_2 = \overline{n}_2$

$n'_0n_3 \oplus n'_1n_2 \oplus n'_2n_1 \oplus n'_3n_0 \oplus (n_2 \wedge n'_2) = n_3 \oplus n_2\overline{n}_1 \oplus n_1\overline{n}_2 \oplus n'_3 \oplus (0) = 1$

$\Rightarrow n_3 \oplus n_2 \oplus n_1 \oplus n'_3 = 1 \qquad \Rightarrow n'_3 = \overline{n_1 \oplus n_2 \oplus n_3}$

So using the Montgomery condition $RR^{-1} - NN' = 1$ the lower 4 bits of N' can be derived directly from the lower digit of N. It is not recommended to do this for large α ($\alpha \geq 8$), for at each new bit more carry bis are generated which would increment the logic depth to determine the lower α bits of N' considerably.

So using:
$n'_0 = 1$
$n'_1 = \overline{n}_1$
$n'_2 = \overline{n}_2$
$n'_3 = \overline{n_1 \oplus n_2 \oplus n_3}$

up to $\alpha = 4$ the Montgomery constant $N' \bmod 2^\alpha$ does not have to be calculated externally, for a PE can do this using the first input digit of N.

Now m_i can be calculated by equation (68) using the product of the α lower bits of N' and $v_o(i)$. If $v_o(i)$ and m_i *are* written using binary digits as:

$$v_o(i) \bmod 2^\alpha = (v_{\alpha-1}\ v_{\alpha-2} \ldots v_0)_2$$
$$m_i \bmod 2^\alpha = (m_{\alpha-1}\ m_{\alpha-2} \ldots m_0)_2$$

the lower four bits of the product result m_i can be determined seperately again using the 'paper & pencil' method:

$$\begin{array}{ccccc}
v_3 & v_2 & v_1 & v_0 & \\
n'_3 & n'_2 & n'_1 & n'_0 & \\
\hline
 & & & & \times \\
n'_0v_3 & n'_0v_2 & n'_0v_1 & n'_0v_0 & \\
n'_1v_2 & n'_1v_1 & n'_1v_0 & & \\
n'_2v_1 & n'_2v_0 & & & \\
n'_3v_0 & & & & \\
\hline
 & & & & + \\
m_3 & m_2 & m_1 & m_0 & = m_i \bmod 2^4
\end{array}$$

Adding the generated carry bits of an addition to the addition of the next bit of m_i mod 2^α, the lower four bits of m_i are defined by:

$$m_0 = n'_0v_0 = v_0$$

$$m_1 = n'_0v_1 \oplus n'_1v_0 = v_1 \oplus \overline{n}_1 v_0$$

$$m_2 = n'_0v_2 \oplus n'_1v_1 \oplus n'_2v_0 \oplus (v_1 \wedge \overline{n}_1 v_0)$$

$$= v_2 \oplus \overline{n}_1 v_1 \oplus \overline{n}_2 v_0 \oplus (v_1 \wedge \overline{n}_1 v_0)$$

$$m_3 = n'_0v_3 \oplus n'_1v_2 \oplus n'_2v_1 \oplus n'_3v_0 \oplus c_{m2}$$

$$= v_3 \oplus \overline{n}_1 v_2 \oplus \overline{n}_2 v_1 \oplus (\overline{n_1 \oplus n_2 \oplus n_3})v_0 \oplus c_{m2}$$

If the following definitions are made:

a $= v_2$
b $= \overline{n}_1 v_1$
c $= \overline{n}_2 v_0$
d $= v_1 \wedge \overline{n}_1 v_0$

carry bit c_{m2} can be determined by:

$$c_{m2} = (a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$$

$$= a \wedge (b \oplus d) \;\vee\; c \wedge (a \oplus b) \;\vee\; d \wedge (b \oplus c)$$

Using these definitions of the lower four bits of m_i, a PE can calculate m_i using only the lower α bits of digits $v_0(i)$ and n_0 without the use of a multiplier of size α x α and without the need for precalculation of N' mod 2^{α}.

6.4.2 Optimization of the adders

By replacing the last two 2-input adder stages by one three-input carry-save adder, PE type 2 has a better performance index than PE type 1. This optimization step can be applied once more by adding the second adder stage of figure 6.3 to the carry-save adder, which results in a 4-input delayed-carry adder. This adder, of which is a larger version has been used in the Brickell design [Bri82], has a logic depth of four full adders.

Because hardware compilations show that PE's perform best using $\alpha = 2$, further optimization of PE's are focussed on PE's with $\alpha = 2$. Now the multiplications can be replaced by addions ($x_{i,0}y_j + 2x_{i,1}y_j$) and ($m_{i-1,0}n_j + 2m_{i-1,1}n_j$). In this way the eight PE input digits (δ and γ are considered to be one adder input digit) can be added using two more of these 4-input delayed-carry adders. PE type 3 will then look like figure 6.4.

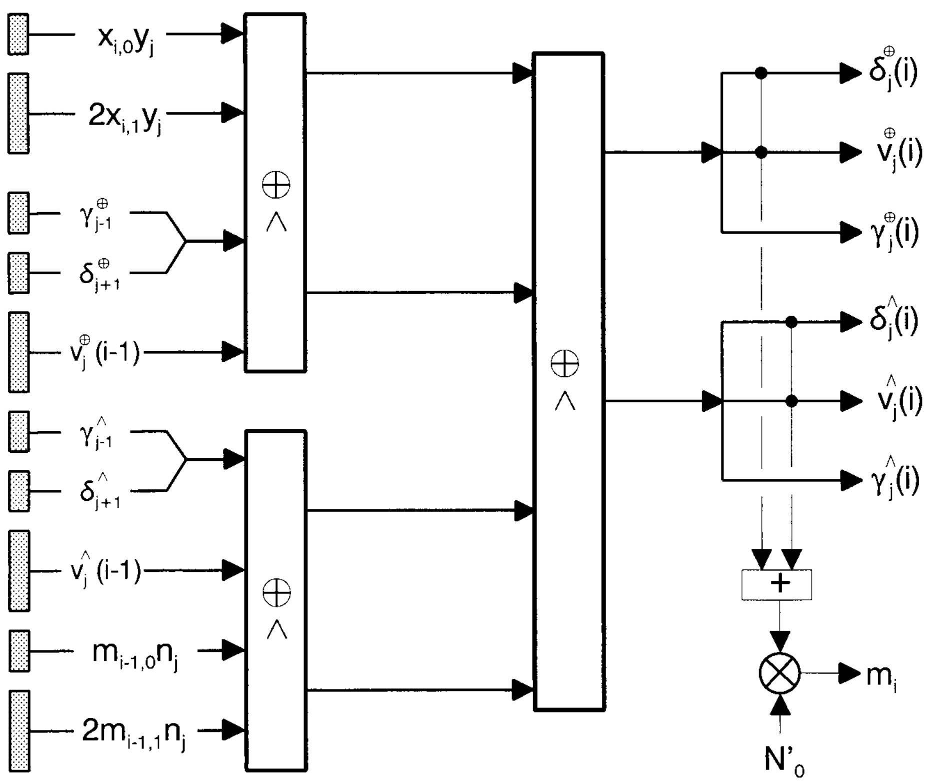


figure 6.4: Schematic of PE type 3 using delayed-carry adders

This type of PE shows a logic depth of eight full adders and little logic for the calculation of m_i. Although this type of PE has not been compiled to hardware design yet, it is estimated that this type of PE can run at a clock frequency of over 90 MHz.

7 Conclusions and Recommendations

In this report the design of an RSA crypto-processor has been presented, using an MMM-core consisting of PE's. The desing is flexible by choice of parameters α and β, which have effect on the size and maximum clock frequency of a PE, and by choice of the number of PE's p, which has effect on the size of the MMM-core. All parameters directly relate to the number of clock cycles which is required for one exponentiation.

The RSA processor is based on a common exponentiation algorithm, which makes use of at most $1.5n$ repeated modular multiplications based on the Montgomery algorithm. The MMM-core, which executes an adapted version of this algorithm (suited for systolic arrays) has been simulated and funtionally tested. The simulations show that the adapted algorithm is working correctly.

The performance of the RSA processor has been estimated by hardware compilations of PE's using several values of α and β. These compilations show, that an MMM of 128 PE's (each PE of size $\alpha = 2$, $\beta = 8$) can calculate about 83 (1024 bits) cryptions/second at a clock frequency of 66 MHz. The size of the MMM-core in that case is approximately 68 Kgates (best-case estimation).
Low-speed RSA cryption can be executed using an MMM consisting of 1 PE of 6 Kgates ($\alpha = 8$, $\beta = 32$), which can reach 4.1 (1024 bits) cryptions/second at a clock frequency of 27 MHz. However, this MMM requires a FIFO buffer of 32 levels of width 48 bits.

The RSA cryption device is also flexible regarding cryption width: A smaller cryption width results in less X and Y-digits, which reduces cryption time. The upper bound of the cryption width is only determined by the size of the on-chip memory.
Using a double cryption width will decrease the cryption time only by a factor 8. Because of the regular structure of the MMM, multiple MMM-cores can be connected in cascade, decreasing the number of clock cycles required for an exponentiation.

The performance of the presented RSA design can be improved by:

- *Optimization of the PE's*
  - Using dedicated delayed-carry adders (as in PE type 3), high-speed PE's can be designed.
  - Designing a PE using customized hardware can decrease the size of the MMM-core considerably. Because all PE's are identical, customization only has to be executed for a single PE.

- *Optimization of the exponentiation algorithm*
  Literature provides several improved exponentiation algorithms based on repeated modular multiplications. The improvement generally applies to reduing the number of multiplications required for an exponentiation. If these improved algorithms can be adjusted such that the MMM-core can execute these multiplications, exponentiation is speeded up.

It can be concluded, that a flexible RSA device has been developed which can operate in both a high-speed and a low-area environment, using different parameters.

Literature references

[Bri82] Brickell, E.F.
A Fast Modular multilplication algorithm with application to two key cryptography
Crypto '82, p. 51-60
Albuquerque, New Mexico

[Dim95] Dimitrov, V. and T. Cooklev
Two algorithms for modular exponentiation using nonstandard arithmetics
Journal: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences
Vol: E78-A Iss:1 p.82-87
Jan. 1995, Country of publication: Japan

[Dix92] Dixon, B. and A.K. Lenstra
Massively parallel elliptic curve factoring
EUROCRYPT '92 EXTENDED ABSTRACTS, pp. 169-179

[Dus90] Dusse, S.R. and B.S.Kaliski
A cryptographic library for Motorola DSP56000
Advances in cryptology, Eurocrypt '90, p.230-244,
Springer-Verlag (1990)].

[Eld93] Eldridge, S.E. and C.D. Walter
Hardware Implementation of Montgomery's Modular Multiplication Algorithm
IEEE Transactions on Computers, Vol. 42, No. 6, June 1993 p. 693-699]

[Iwa93] Iwamura, K. , T. Matsumoto and H. Imai
A Parallel Processing Method for Implementing the RSA Cryptosystem
Electronics and Communications in Japan,
Part 3, Vol. 76, No 5, May 1993 p. 14-27

[Iwa94] Iwamura, K. , T. Matsumoto and H. Imai
Montgomery modular-multiplication method and systolic arrays suitable for modular exponentiation
Electronics and Communications in Japan, Part 3, Vol. 77, 1994, No. 3, p. 40-50.
Translated from: Denshi Joho Gakkai Ronbunshi *Japan).
Vol. 76-A (1993), No. 8, p. 1214-1223

[Kaw93] Kawamura, S. and A. Shimbo
Fast sever-aided secret computation protocols for modular exponentiation
Journal: IEEE Journal on Selected Areas in Communications
Vol: 11 Iss: 5 p.778 - 784
June 1993, USA

[Knu69] Knuth, Donald E.
Seminumerical algorithms
Volume 2 of The Art of Computer Programming.
Addison Wesley. Reading, Massachusetts, 1969. (RSA paper)

[Mor90] Morita, H.
A Fast Modular-Multiplication Algorithm Based on a Higher Radix
Advances in cryptology, Crypto '89, p. 387-399
Springer-verlag 1990

[Niv72] Niven, I. and H. S. Zuckerman.
An Introduction to the Theory of Numbers
John Wiley & Sons, New York 1972

[Pol74] Pollard, J.M.
Theorems on factorization and primality testing
Proc. Camb. Phil. Soc. (1974), pp 521-528

[Wal93] Walter, C.D.
Systolic Modular Multiplication
IEEE Transactions on Computers
Vol. 42, ISs:3, p. 376-378, March 1993, USA

[Zha93] Zhang, C.N.
An improved binary algorithm for RSA
Journal: Computers & Mathematics with Applications
Vol: 25 Issue: 6 p. 15-24
March 1993 Country of publication: UK

Appendix A: MMM controller functions

DESCRIPION OF MMM THE CONTROLLER FUNCTIONS

Register j	INC		(load next Y and N-digits the next clock cycle)
if $j \geq p$ then			
	Register i	HOLD	(stop loading X-digits if no PE addressed)
else			
	Register i	INC	
if j = MAX(l+1, p) then			(reset j if all digits y_l are loaded ($l+1 \geq p$))
	Register j	RESET	(or all PE's are addressed ($l+1 < p$))
if ($j < l$) then			
	Ymux	$y_j = y_j$	(load Y-digits from external memory)
else			
if ($j = l$) then			
begin			
	Ymux	$y_j = t_l(k)$	(feed back internally stored overflow digit)
	Ymux	$n_j = 0$	(by adding an extra digit after y_{l-1})
end			
else {$j > l$}			(more PE's than Y-digits: add zero-digits)
begin			(to fill all PE's of the MMM)
	Ymux	$y_j = 0$	
	Ymux	$n_j = 0$	
end			
if $n = l\beta$ then			
begin			
if $\alpha = 1$ then			
if $i \geq k$-2 then			
begin			
	Xmux	$x_i = 0$	(concatenate three digits x_{k-2} to x_k)
	Xmux	LSB(x_{k-2}) = LSB($t_l(k)$)	(feed back LSB of internal overflow digit)
else			
	Xmux	$x_i = x_i$	(load X-digits from external memory)
end			

```
    else {α ≥ 2}
        if i ≥ k-1 then
        begin
            Xmux      x_i = 0                       (concatenate internal digits x_{k-1} and x_k)
            Xmux      LSB(x_{k-1}) = LSB(t_l(k))    (feed back LSB of internal overflow digit)
        else
            Xmux      x_i = x_i                     (load X-digits from external memory)
        end
    end
else {n < lβ}
    if kα > lβ then
        if i ≥ k-1 then
            Xmux      x_i = 0                       (concatenate internal digits x_{k-1} and x_k)
        else
            Xmux      x_i = x_i                     (load X-digits from external memory)
        end
    else {kα ≤ lβ}
        if i = k then
            Xmux      x_i = 0                       (concatenate x_k internally)
        else
            Xmux      x_i = x_i                     (load X-digits from external memory)
        end
    end
end

if (i = k) and (FIFO = empty) then                  (MMM is ready for a new multiplication)
    begin
            Register i    RESET
            Register j    RESET
    end
```

Appendix B: VHDL description of PE type 2

```
*******************************************************************
--
-- Company     :   Pijnenburg Custom Chips b.v.
--
-- Project     :  Pxxx
--
-- Designer    :   E.Kuipers
--
-- Hierarchy   :  ~/p900/synopsys/rtl
--
-- File        :  PE4.VHD
--
-- Creation    :  01/04/96
--
-- Description:  PE type 2
--
--
-- Changes     :
--
*******************************************************************
library IEEE ;
   USE IEEE.std_logic_1164.all;
   USE IEEE.std_logic_arith.all;
PACKAGE MMM_GLOBAL IS
        CONSTANT    A           : integer := 4;

        CONSTANT    B           : integer := 32;

END MMM_GLOBAL;

LIBRARY ieee;
   USE ieee.std_logic_1164.ALL;
   USE ieee.std_logic_misc.ALL;
   USE ieee.std_logic_arith.ALL;
   USE ieee.std_logic_unsigned.ALL;
LIBRARY MMM_RTL;
   USE MMM_RTL.MMM_GLOBAL.ALL;

entity PE1 is
       PORT  (  CLK      : In    std_logic;
                InitPE   : In    std_logic;


                Xi       : In    std_logic_vector (A-1 downto 0);
                mi       : In    std_logic_vector (A-1 downto 0);
                Yi       : In    std_logic_vector (B-1 downto 0);
                Ni       : In    std_logic_vector (B-1 downto 0);
```

```
                Ti_and  : In    std_logic_vector (B-1 downto 0);
                d1i_and : In    std_logic_vector (A-1 downto 0);
                d2i_and : In    std_logic_vector (A-1 downto 0);
                gi_and  : In    std_logic_vector (A-1 downto 0);
                Ti_xor  : In    std_logic_vector (B-1 downto 0);
                d1i_xor : In    std_logic_vector (A-1 downto 0);
                d2i_xor : In    std_logic_vector (A-1 downto 0);
                gi_xor  : In    std_logic_vector (A-1 downto 0);

                mo      : Out   std_logic_vector (A-1 downto 0);
                Yo      : Out   std_logic_vector (B-1 downto 0);
                No      : Out   std_logic_vector (B-1 downto 0);
                T_o_and : Out   std_logic_vector (B-1 downto 0);
                d1o_and : Out   std_logic_vector (A-1 downto 0);
                d2o_and : Out   std_logic_vector (A-1 downto 0);
                go_and  : Out   std_logic_vector (A-1 downto 0);
                T_o_xor : Out   std_logic_vector (B-1 downto 0);
                d1o_xor : Out   std_logic_vector (A-1 downto 0);
                d2o_xor : Out   std_logic_vector (A-1 downto 0);
                go_xor  : Out   std_logic_vector (A-1 downto 0)
             );
end PE1;

architecture BEHAVIORAL of PE1 is

             SIGNAL X      : std_logic_vector (A-1 downto 0);
             SIGNAL m      : std_logic_vector (A-1 downto 0);
             SIGNAL Y      : std_logic_vector (B-1 downto 0);
             SIGNAL N      : std_logic_vector (B-1 downto 0);
             SIGNAL T_and  : std_logic_vector (B-1 downto 0);
             SIGNAL d1_and : std_logic_vector (A-1 downto 0);
             SIGNAL d2_and : std_logic_vector (A-1 downto 0);
             SIGNAL g_and  : std_logic_vector (A-1 downto 0);
             SIGNAL T_xor  : std_logic_vector (B-1 downto 0);
             SIGNAL d1_xor : std_logic_vector (A-1 downto 0);
             SIGNAL d2_xor : std_logic_vector (A-1 downto 0);
             SIGNAL g_xor  : std_logic_vector (A-1 downto 0);

             SIGNAL p1     : std_logic_vector (B+A-1 downto 0);
             SIGNAL p2     : std_logic_vector (B+A-1 downto 0);
             SIGNAL s1_and : std_logic_vector (B-2*A-1 downto 0);
             SIGNAL s1_xor : std_logic_vector (B-2*A-1 downto 0);
             SIGNAL s2_and : std_logic_vector (B     downto 0);
             SIGNAL s2_xor : std_logic_vector (B     downto 0);
             SIGNAL s3     : std_logic_vector (B+2   downto 0);
             SIGNAL s4_and : std_logic_vector (B+2*A-1 downto 0);
             SIGNAL s4_xor : std_logic_vector (B+2*A-1 downto 0);
             SIGNAL N0     : std_logic_vector (A-1   downto 0);
             SIGNAL ZEROES : std_logic_vector (B-1   downto 0);
             SIGNAL s8     : std_logic_vector (A-1   downto 0);
```

```
begin

    registers: PROCESS(CLK)
    BEGIN
        IF (CLK'event) AND (CLK = '1') THEN

            -- load always Y, N, T, d1 and d2 on each clock
            Y        <= Yi;
            N        <= Ni;
            T_and    <= Ti_and;
            d1_and   <= d1i_and;
            T_xor    <= Ti_xor;
            d1_xor   <= d1i_xor;

           IF (InitPE = '1') THEN
                -- reset register g, load Xi and mi
                X        <= Xi;
                m        <= mi;
                g_and    <= (OTHERS => '0');
                g_xor    <= (OTHERS => '0');
            ELSE
                -- hold registers X, m, g
                X        <= X;
                m        <= m;
                g_and    <= g_and;
                g_xor    <= g_xor;
            END IF;
         END IF;
    END PROCESS registers;


    PE_input: PROCESS(d2i_and, d2i_xor)
    BEGIN
        d2_and    <= d2i_and;
        d2_xor    <= d2i_xor;
        ZEROES    <= (OTHERS => '0');
    END PROCESS PE_input;


    mul1: PROCESS(m,N)
    BEGIN
            -- m width > 1 bit
            p1       <= m * N;
    END PROCESS mul1;


    mul2: PROCESS(X,Y)
    BEGIN
            -- X width > 1 bit
            p2       <= X * Y;
    END PROCESS mul2;
```

```
PEadd1_and: PROCESS(d2_and, g_and, ZEROES)
BEGIN
        -- b >= 4a, d2_and and g_and don't overlap: construct s1
              as 1 digit of width B-2A, using B-4A zeroes
        s1_and  <= d2_and & ZEROES(B-(4*A)-1 downto 0) & g_and;
END PROCESS PEadd1_and;


PEadd1_xor: PROCESS(d2_xor, g_xor, ZEROES)
BEGIN
        -- b >= 4a, d2_xor and g_xor don't overlap: construct s1
              as 1 digit of width B-2A, using B-4A zeroes
        s1_xor  <= d2_xor & ZEROES(B-(4*A)-1 downto 0) & g_xor;
END PROCESS PEadd1_xor;


PEadd2_and: PROCESS(s1_and,T_and,ZEROES)
-- ripple-carry adder for redundant carry-bits of T,
VARIABLE    T1,
            T2,
            SUM : std_logic_vector(B-A downto 0);
BEGIN
    T1      := ZEROES(A downto 0) & s1_and;
    T2      := '0' & T_and(B-1 downto A);
    SUM     := T1 + T2;
    s2_and  <= SUM & T_and(A-1 downto 0);
END PROCESS PEadd2_and;

PEadd2_xor: PROCESS(s1_xor,T_xor,ZEROES)
-- ripple-carry adder for redundant sum-bits of T, width: 1+B-A
VARIABLE    T1,
            T2,
            SUM : std_logic_vector(B-A downto 0);
BEGIN
    T1      := ZEROES(A downto 0) & s1_xor;
    T2      := '0' & T_xor(B-1 downto A);
    SUM     := T1 + T2;
    s2_xor  <= SUM & T_xor(A-1 downto 0);
END PROCESS PEadd2_xor;
```

```
PEadd3: PROCESS(s2_and, s2_xor, ZEROES)
-- ripple-carry adder which converts the redundant sum T+g+d to
      normal representation
VARIABLE    T1,
            T2,
            SUM : std_logic_vector(B+1 downto 0);
BEGIN
    T1      := "00" & s2_xor(B downto 1);
    T2      := '0' & s2_and;
    SUM     := T1 + T2;
    s3      <= SUM & s2_xor(0);
END PROCESS PEadd3;


PEadd4: PROCESS(s3, p1, p2, ZEROES)
-- redundant adder width: B+2A (=number of XOR-ANDOR pairs)
VARIABLE    T1,
 T2,
 T3  : std_logic_vector(B+(2*A)-1 downto 0);
BEGIN
 -- A >= 2 !!
    T1      := ZEROES((2*A)-3 downto 0) & s3(B+1 downto 0);
    T2      := ZEROES(A-1 downto 0) & p1;
    T3      := p2 & ZEROES(A-1 downto 0);

    FOR I in (B+(2*A)-1) downto 0
    LOOP
 s4_xor(I)    <= T1(I) XOR T2(I) XOR T3(I);
 s4_and(I)    <= ( T1(I) AND T2(I) ) OR ( T1(I) AND T3(I) ) OR
                                  ( T2(I) AND T3(I) );
END LOOP;

END PROCESS PEadd4;


PE_output: PROCESS(d1i_xor, d1i_and, s4_and, s4_xor, Y, N)
BEGIN
    d2o_xor     <= d1i_xor;
    d2o_and     <= d1i_and;
    d1o_xor     <= s4_xor(A-1 downto 0);
    d1o_and     <= s4_and(A-1 downto 0);
    T_o_xor     <= s4_xor(B+A-1 downto A);
    T_o_and     <= s4_and(B+A-1 downto A);
    go_xor      <= s4_xor(B+(2*A)-1 downto B+A);
    go_and      <= s4_and(B+(2*A)-1 downto B+A);

    -- output input register values
    Yo      <= Y;
    No      <= N;
END PROCESS PE_output;
```

```
    sel_m: PROCESS(d1_and, d1_xor, s4_and, s4_xor, InitPE)
    VARIABLE    s5, s6, s7  : std_logic_vector((2*A)-1 downto 0);
    BEGIN
   s5     := s4_xor(2*A-1 downto 0); -- T_o_xor,d1_xor
   s6     := s4_and(2*A-1 downto 0); -- T_o_and,d1_and
   s7     := s5 + (s6(2*A-2 downto 0) & '0');
            -- define tri-state port (A>1)
            IF (InitPE = '1') THEN
                s8      <= s7((2*A)-1 downto A);
            ELSE
                FOR I in (A-1) downto 0
                LOOP
                        s8(I)       <= 'Z';
     END LOOP;
            END IF;
    END PROCESS sel_m;

    calc_m: PROCESS(s8, N0)
    VARIABLE      p3    : std_logic_vector((2*A)-1 downto 0);
    BEGIN
   p3      := N0 * s8;
   mo      <= p3(A-1 downto 0);
    END PROCESS calc_m;

end BEHAVIORAL;

configuration CFG_PE1_BEHAVIORAL of PE1 is
   for BEHAVIORAL
   end for;
end CFG_PE1_BEHAVIORAL;
```