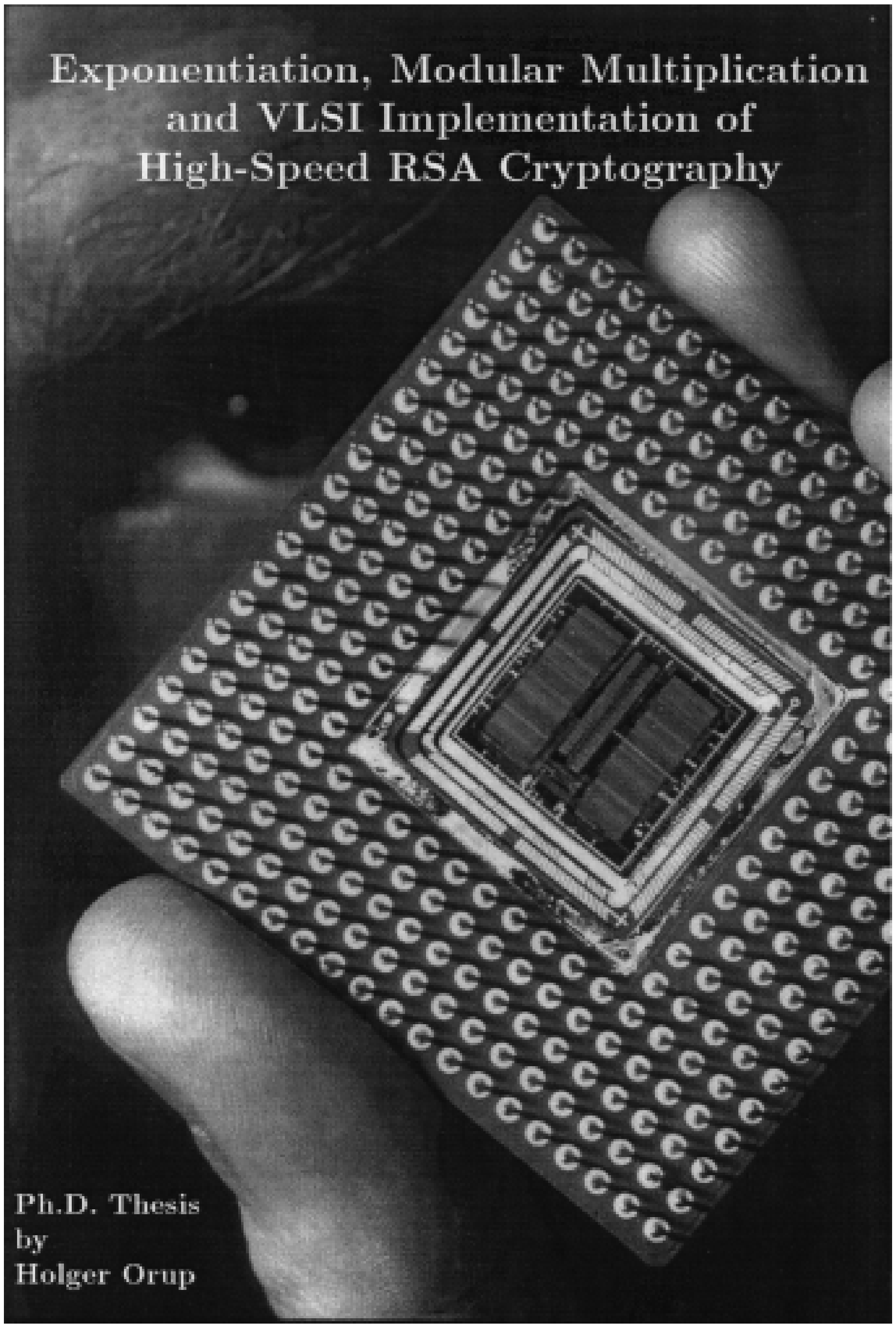
Exponentiation, Modular Multiplication and VLSI Implementation of High-Speed RSA Cryptography



Ph.D. Thesis
by
Holger Orup

Exponentiation, Modular Multiplication and VLSI Implementation of High-Speed RSA Cryptography

Ph.D Thesis

by

Holger Orup

Department of Computer Science,
University of Aarhus,
DK-8000 Aarhus C, Denmark

August 1995

Danish Summary (Dansk Resumé)

Denne afhandling er resultatet af et Ph.D. studium udført i perioden fra 1990 til 1995. Studiet foregik ved Datalogisk Afdeling, Aarhus Universitet under vejledning af Lektor Peter Møller-Nielsen. I perioden fra september 1990 til januar 1993 blev en del af arbejdet udført ved Udviklingsafdelingen, Tele Danmark/Jydsk Telefon.

Problemstilling

Afhandlingen er motiveret af en problemstilling, der især er kendt fra anvendelsen af RSA kryptosystemet: Den praktiske anvendelighed af systemet er afhængig af hastigheden, hvormed man er i stand til at udføre de tilhørende transformationer af data. For at undgå at anvendelsen af kryptografi bliver en flaskehals i de elektroniske kommunikationssystemer, er det vigtigt at opnå en tilstrækkelig høj transformationshastighed.

Transformationerne, der benyttes i RSA kryptosystemet, er baseret på en aritmetisk operation på formen M^e mod m kaldet modulo eksponentiering. Da operandernes størrelse typisk er på 500 bit eller mere, er modulo eksponentiering en forholdsvis tidskrævende operation. I 1990, da projektet startede, var den hurtigste implementering af modulo eksponentiering i stand til at foretage beregningerne med en hastighed på 29 Kbit/sek ved en operandstørrelse på 512 bit.

Formål

Hovedformålet med arbejdet præsenteret i afhandlingen er at undersøge mulighederne for at forøge hastigheden af modulo eksponentiering ved hjælp af specialdesignede VLSI (Very Large Scale Integrated) kredsløb. Den konkrete målsætning for projektet udført ved Tele Danmark/Jydsk Telefon var at konstruere et VLSI kredsløb, som kunne foretage module eksponentiering af operander på 561 bit med en hastighed på minimum 64 Kbit/sek. Endvidere skulle kredsløbet implementeres på en enkelt chip.

Formålet med selve afhandlingen er at beskrive resultaterne opnået under studiet og at sætte disse i perspektiv ved en diskussion og sammenligning med resultater kendt fra litteraturen. Desuden er det hensigten at give læseren indsigt i de teknikker, der kan anvendes til at effektivisere modulo exponentiering.

Tilgangsvinkel

Den valgte tilgangsvinkel er baseret pa en simpel analyse af VLSI implementeringer, der var kendt i 1990: Alle de hurtigste implementeringer bruger det samme antal klokke perioder til at foretage en modulo eksponentiering, og forskellen i hastighed kan tilsyneladende alene tilskrives forskellen i den anvendte klokkefrekvens. Dette kunne tyde på, at fremskridt i hastighed alene er nået på baggrund af hurtigere kredslebsteknologier, og ikke på grund af forbedrede beregningsmetoder. Derfor blev det besluttet at undersøge mulighederne for at udvikle nye og mere effektive beregningsmetoder, som er velegnede til VLSI implementering.

Afhandlingens struktur

Afhandlingen er opdelt i to separate dele. Den sidste del består af fire artikler, gengivet i appendices A-D, samt en rapport med beskrivelser af RSA kredsen i appendix E. Alle dokumenterne er skrevet i perioden fra 1990 til 1995. Artiklerne præsenterer nogle af de opnåede resultater. Den første del af afhandlingen indeholder en relativt udtømmende oversigt over relevante resultater fra litteraturen. Disse resultater diskuteres og sammenholdes med resultaterne opnået under studiet. Desuden er projektet med konstruktion af RSA kredsen beskrevet.

Oversigtsdelen

De første fire kapitler udgør en hierakisk struktureret fremstilling, hvor problemstillingerne identificeret på ét niveau løses ved et sæt teknikker, der introducerer et nyt sæt lavere liggende problemstillinger. Det nederste niveau i hierakiet er kredsløbsniveauet, hvor løsningerne realiseres ved en kredsløbsarkitektur, der er i stand til at udføre en specifik beregningsmetode. Det femte kapitel beskriver den senest udviklede beregningsmetode. Metoden repræsenterer en markant fremgang i de opnåelige beregningshastigheder og må ses som et produkt af at samle erfaringerne fra de første fire kapitler. Endelig afsluttes oversigtsdelen med en kortfattet konklusion, der præciserer de gennemgående løsningsstrategier og -teknikker.

I det følgende er de væsntligste opnåede resultater kort skitseret. Resultaterne er opdelt efter emne:

Eksponentierings metoder. Da eksponentiering foretages ved gentagen multiplikation, er litteraturen præget af metoder, der forsøger at minimere antallet of multiplikationsoperationer. Imidlertid er det muligt at parallellisere beregningen og herved opnå en kortere beregningstid end det er muligt med sekventielle metoder. Dette er på trods af, at det samlede antal multiplikationer ikke er minimalt. Det væsentligste bidrag til dette emneområde er en metode, der anvender den parallelle beregningsstrategi uden en samtidig signifikant forøgelse af VLSI kredsløbets størrelse. Dette opnåes ved at anvende en samlebåndsteknik (eng: pipelining). Det synes vanskeligt at opnå yderligere fremgang i metoderne til modulo eksponentiering.

Modulo multiplikations metoder. Der er til gengæld store muligheder for at forøge hastigheden af modulo multiplikation. I denne afhandling er den gennemgående strategi at opnå mere effektive beregninger af modulo multiplikation ved at benytte såkaldte høj-radix metoder. Den basale idé bag høj-radix metoder er at reduce antallet af additioner. Dette kan udtrykke sig ved, at antallet af klokkeperioder per multiplikation reduceres. Et af de største problemer ved høj-radix metoderne er, at det reducerede antal klokkeperioder ledsages af en forøgelse af klokkeperiodens længde. Et af de væsentligste resultater i afhandlingen er en særdeles effektiv høj-radix metode, hvor klokkeperiodens længde ikke forøges ved stigende radices.

VLSI implementering. En RSA processor er blevet konstrueret, fabrikeret og afprøvet. Den er fuld funktionsdygtig og lever op til kravene, der blev specificeret ved projektets begyndelse. Det viser sig at være den hidtil hurtigste kendte implementation af modulo exponentiering.

Artikeldelen

I de fem appendices A-E er gengivet fire artikler samt en kortfattet rapport. Den første artikel, publiceret i [OSA90a], danner grundlag for beregningsmetoderne, der benyttes af RSA processoren. Den anden artikel, publiceret i [OK91], præsenterer en module multiplikations metode, der i udstrakt grad udnytter egenskaberne ved såkaldte redundante repræsentationer. Endvidere beskrives en velegnet kredsløbsarkitektur, som udnytter samlebåndsteknikken til at formindske størrelsen af kredsløbet. Den tredie artikel beskriver strategien bag arbejdet med at reducere arealet af RSA processoren. Artiklen indeholder en oversigt over effekten af de anvendte teknikker. Den fjerde artikel, publiceret i [Oru95], præsenterer en særdeles effektiv høj-radix modulo multiplikationsmetode. Metoden baserer sig på en kombination af tidligere kendte teknikker. Endelig er en rapport inkluderet. Rapporten indeholder nogle af de væsentligste data vedrørende RSA processorens funktionalitet.

Preface

This thesis is the result of a Ph.D. study carried out in the period from 1990 to 1995. The study was done at the Department of Computer Science, University of Aarhus, Denmark, and supervised by Associate Professor Peter Møller-Nielsen. As an integral part of the study I was at the Research and Development Department, Tele Danmark/Jydsk Telefon, Denmark, in the period from September 1990 to January 1993.

The study was driven by a specific problem known from the field of public key cryptography: In order to avoid the application of cryptography to be a bottle-neck in a communication system, it is necessary to perform the cryptographic transformations at a rate corresponding to the transmission rate of the communication channel. In the RSA public key crypto system, and in other public key systems as well, the transformation rate is limited by the speed of which modular exponentiation can be performed. The aim of the study was, primarily, to implement a special-purpose processor by means of the VLSI circuit technology and, hereby, to demonstrate that the transformations used in RSA cryptography can be performed at 64 Kbit/sec, corresponding to the transmission rate of an ISDN channel.

The thesis is organised as two major parts. The first part constitutes a relatively exhaustive description of the relevant research results on methods for fast computation of modular exponentials. The description provides an insight into the properties of the arithmetical operations used. It is discussed how these properties, through various optimisation techniques, can be utilised to obtain efficient computation methods. Finally, the hardware implementation project carried out at Tele Danmark/Jydsk Telefon is described. The second part of the thesis is structured as five separate papers. The papers presents the results obtained during the study.

My interest for the problem treated in this thesis was born in 1989 during a conversation with two fellow students. One of the fellows was taking a course in Cryptology and had been faced with the problem of RSA cryptography. Hence, he asked

> "Is it possible to construct a VLSI processor that performs fast computations of expressions of the form $M^e \bmod m$?"

We got the opportunity to study the problem in further detail and in January 1990 we completed our Master's thesis [OSA90b] on the subject. The results were described in an article [OSA90a] and presented at the Eurocrypt '90 Conference in Aarhus, Denmark, May 1990. Furthermore, a report [OS90] on improvements of the results of the Master's thesis was completed in June 1990.

After receiving my Master's degree I applied for, and got, a Ph.D. scholarship in the summer 1990. During the Eurocrypt '90 Conference a contact to Tele Danmark/Jydsk Telefon was established. A cooperation was initiated in September 1990 with the purpose of implementing a VLSI processor. A computation method and an associated architecture, mainly based on the article from Eurocrypt '90, was chosen for the implementation. The processor was sent for fabrication in January 1993. It was presented at the Hot Chip VI Symposium, Standford, California, August 1994 [Oru94]. In the first stage of the project, for a period of approximately a year, the work was done in cooperation with Erik Svendsen.

In the fall 1990 Professor Peter Kornerup, Odense University, Denmark, gave me an introduction to the field of Computer Arithmetic and explained that some of the ideas we had presented at Eurocrypt '90 were well-known techniques from that field. This resulted in the article [OK91], presented at the 10th Symposium on Computer Arithmetic, Grenoble, France, June 1991. Finally, at the 12th Symposium on Computer Arithmetic, Bath, England, July 1995, the results of my latest work were presented [Oru95].

The photography on the front cover of the thesis shows the processor that was implemented through the cooperation with Tele Danmark/Jydsk Telefon.

Aarhus, Denmark *Holger Orup*
August, 1995

Acknowledgements

I am truly grateful for the inspiration, advice and support I have received from many people during my work on this thesis. It is not possible to mention everyone whose interaction contributed to this work, but I would like to thank some of the people that have helped me the most.

First of all I would like to express my deepest gratitude to my wife Hanne for her all-out support.

At University of Aarhus I would like to thank my supervisor Peter Møller-

Nielsen for his help, advice and support during my studies. I would also like to thank Ole Caprani and Henrik Esbensen for many fruitful discussions and suggestions.

At Tele Danmark/Jydsk Telefon I would like to thank my former colleagues at the Research and Development Department. A special thank to Poul Gødsvang and Finn Barrett for many inspiring discussions. I would also like to thank Erik Svendsen, who was directly involved in the design of the RSA processor, as well as John Thorup, who designed and constructed the boards for testing the processor. Finally, I owe thanks to Palle Brandt for taking the initiative to establish the project of implementing the RSA processor.

Thanks to Peter Kornerup, Odense University, for introducing me to the field of Computer Arithmetic and for constructive discussions, advice and support.

I am also very grateful for the financial support provided by the Department of Computer Science, University of Aarhus and by the AIDA research project[1].

Aarhus, Denmark *Holger Orup*
August, 1995

[1]Danish Natural Science Research Council, grant no, 5.21.08.02

Contents

Chapter 1

Introduction

This chapter provides an introduction to the subject and purpose of this thesis. Section 1.1 gives a motivation for the work presented. After a brief description of the services provided by cryptography, the RSA public key crypto system is introduced. The complexity of computing modular exponentials, expressions of the form $M^e \bmod m$, represents the most serious limitation on the applicability of public key cryptography. In Section 1.2 the purpose of this thesis is described: To develop fast hardware implementations for supporting the computations required in RSA cryptography. Section 1.3 describes the approach chosen for achieving the goal of the presented work. Section 1.4 describes the overall organisation of the thesis. Finally, Section 1.5 provides a brief description of the papers included in the Appendices.

1.1 The Need for Crypyography

Electronic communication system are quickly overtaking paper-based systems and face-to-face meetings [DMW94]. Every day, millions of people use telephones, fax machines, and computer network for interactions. Sensitive data are transmitted over insecure channels. Landau et al. [LKB$^+$94] describe cases, where weak links in the communication system have been used for penetration: In the 1970s thousands of phone conversations about business at IBMs private microwave network were systematically eavesdropped by Soviet intelligence agents. A group of students at the University of Wisconsin forged an email letter of resignation from the director of housing to the chancellor of the university.

The accelerating introduction of electronic communication will increase the importance of *information integrity* [Sim92a], i.e. all the questions concerning privacy, authenticity, authority etc.: Business will tele-connect with customers to sell and bill. Manufactures will electronically query suppliers to check product availability. Insurance companies, doctors and medical centres will carry on electronic exchanges about patient treatment.

Vulnerable communication can easily undermine user's confidence in a system. Hence, there is an urgent need for means to provide for the integrity of information. A very important part of the solutions on the information integrity is cryptographic in nature.

1.1.1 Information Integrity Functions

Information integrity functions for paper-based communication systems have evolved over thousands of years. The needs for parties of the paper-based communication include such functions as listed in Table 1.1 [DMW94]. The table lists a few a the more common functions and is far from complete. The conventional paper-based information integrity functions motivate the need for similar functions in electronic communication and provide a basis for analogies. The same message or transaction that now is handled by a paper-based system will soon be sent by an electronic system, to improve the speed of communication and the cost of handling. As this evolution progresses, the security needs of users are not diminishing. In fact, with easy access to more information on-line, the threats to the information integrity are likely to increase.

Attacks on paper-based systems generally require *physical access* to one of the few copies of a paper message. In contrast, an electronic system may store multiple copies of messages. A sender and recipient of a message generally does not know exactly which nodes of a network carried the message. Furthermore, there is a potential for undetectable, *remote access* to systems storing or transmitting electronic messages. Some basic threats to the information integrity in electronic communication systems are listed in Table 1.1.1 [Fum91].

A more elaborate presentation of the information integrity functions needed

Signature	Verify the identity of the originator. Written signatures are the primary form of identifying the originator. In early times wax seals were imprinted with the symbol of an important individual or office.
Integrity	Ensure that the message or transaction received is the same as that sent, without accidental or intentional modification. Transactions are recorded in ink on non-erasable documents. Special papers have been developed that display certain indicators if they are modified.
Non-repudiation	Prove to a third party that the transaction actually took place. This prevents an individual from denying having engaged in a transaction. Procedures have been established to verify individual signatures, keep duplicate copies of transactions, and entrust third parties to adjudicate disputes.
Privacy (secrecy)	Keep communications private. Physical protection mechanisms have evolved to ensure the privacy of transactions. The glue seal on an envelope and the wax seal on a document are used to discourage others from reading a message. Cryptography, the basis of much of the security in electronic communication systems, is a method almost as old as writing itself.

Table 1.1: Information integrity functions in paper-based communication systems.

Interception of data	Network meadia can be tapped.
Modification of message	Modification of a message occurs when the contents of a message is altered without detection and results in unauthorised effects
Replay of message	A replay occurs when a message, or a part of it, is repeated to produce an unauthorised effect.
Masquerading	A masquerade is when an entity pretends to be a different entity. This can be used to introduce invalid messages that are delivered as if they were genuine.
Repudiation	This refers to a senders (or recipients) denial of participation in a transaction.

Table 1.2: Some information integrity threats.

in electronic communication systems, and of the corresponding threats, can be found in text-books on cryptography, e.g. [Den82, Bra88, PGV91, Sim92b].

1.1.2 Cryptography

As mentioned above, a means for providing some of the important information integrity functions is cryptography. In general, there are two types of cryptography denoted *secret key cryptography* and *public key cryptography*. The crypto systems for performing secret key cryptography are also known as conventional crypto systems. Until 1976, when the concept of public key cryptography was introduced by Diffie and Hellman [DH76], all crypto systems were secret key systems. The following brief description of both types of crypto systems gives the reader an introduction to the advantages, and the disadvantages, of the public key crypto systems. The description is based on

[Nec92].

Secret Key Cryptography

A secret key system consists of two transformations: An encryption transformation E_K used for encryption of a message M, and a decryption transformation D_K used for decryption of the encrypted message, i.e. $D_K(E_K(M)) = M$. The transformations are parameterised with a parameter K, denoted the key. By imposing certain requirements to the transformations, it is possible to withstand some of the information integrity threats in Table 1.1.1. Suppose two parties, say Alice and Bob, are communicating messages on a public communication channel. Furthermore, suppose that a third party, say Charlie, has access to the communication channel:

- To prevent Charlie from intercepting a message M send from Alice to Bob, Alice encrypts M using E_K. Then, the resulting so-called ciphertext $C = E_k(M)$ is sent to Bob. Finally, Bob decrypts C using D_K. The key K, used as parameter in the encryption and the decryption, is kept secret from Charlie. Hence, by requiring that it is infeasible for Charlie to compute $D_K(C)$ without knowledge of the key value, Alice and Bob have achieved privacy in their communication.

- Furthermore, if it is infeasible for Charlie to compute $E_K(M)$ without knowledge of the key value, Charlie cannot pretend to be Alice in a communication where M is send to Bob.

The secret key crypto systems are mainly used for providing privacy in a communication between two parties. The *Data Encryption Standard* (DES) system is the most widely used secret key crypto system, see e.g. [SB92].

Public Key Cryptography

One of the reasons [Dif92] for proposing public key cryptography was the problem of key distribution: If two people, who have never met before, are to communicate privately using secret key cryptography, they must somehow agree in advance on a key that will be known to themselves and to no one else. Another reason was the problems of signatures and of non-repudiation: A method was needed for providing the recipient of a purely digital electronic message with a way of demonstrating to other people, that the message had come from a particular person. Hence, the signature should allow the recipient to hold the author to the contents of the message.

Public key systems differ from secret key systems in that there is no longer a single secret key shared by a pair of users. Rather, each user has each own key material. Furthermore, the key material of each user is divided into two portions, a private component and a public component. The public component generates a public transformation E, and the private component generates a private transformation D. Often, E and D is denoted the encryption transformation and the decryption transformation, respectively. This is, however, an imprecise terminology: Depending on the actual system, it may be the case that $D(E(M)) = M$, $E(D(M)) = M$, or both. A common requirement to the public transformation E is that it must be a so-called *trapdoor one-way function*. "One-way" refers to the fact that E should be easy to compute from the public component of the key but hard to invert unless one possesses the corresponding private transformation D, or equivalently, the private component of the key. The private component thus yields a "trapdoor" which makes the problem of inverting E seem difficult from the point of view of all but the possessor of D.

The following examples show how privacy, signatures, and non-repudiation may be provided by a public key crypto system. The transformations D_A and E_A are those generated by Alice's key, and the transformations D_B and E_B are those generated by Bob's key:

- To prevent Charlie from intercepting a message M send from Alice to Bob, Alice encrypts the message by means of Bob's public available transformation E_B. Then, the ciphertext $C = E_B(M)$ is sent to Bob, who decrypts C by means of his own private transformation, $M = D_B(C)1$. So, when the public key crypto system is used for obtaining privacy, only the transformations of the recipient are used. The requirement to the transformations is that $D_B(E_B(M)) = M$. It should be emphasised, that Bob never needs to share D_B with Alice.

- To convince Bob that the message M indeed originates from Alice and, hence, cannot have been generated by Charlie, Alice is able to sign the message: Alice transforms the message by means of her own private transformation. Then, the resulting signed message $S = D_A(M)$ is sent to Bob. Finally, in order to verify the signature, Bob applies Alice's public transformation to obtain $M = E_A(S)$. Since D_A is strictly private to Alice, Charlie could not possibly have generated the signed message. Note that only the transformations of Alice's are used. In order to provide signatures, the transformations must obey $E_A(D_A(M)) = M$.

- The signed message, $S = D_A(M)$, could not even have been generated by Bob. Furthermore, the signature can be verified by every person who has access to Alice's public transformation. Hence, Bob can prove to a third party that Alice indeed was the author of the signed message, and Alice cannot deny having signed the message.

To provide privacy, the transformations used in a public key systems must obey the condition $D(E(M)) = M$, and to provide signatures they must obey $E(D(M)) = M$. According to [Nec92] there is only one major system, the *Rivest-Shamir-Adleman* (RSA) system, that satisfies both conditions. This system will be introduced below.

Compared to the secret key systems, the public key systems provide a wider range of information integrity functions. Furthermore, the key distribution problem is significantly reduced: There is no longer a need for exchanging secret keys. Apart from the private transformation of a user, only the public available transformations of the other users are required in order to apply public key cryptography.

There are, however, a disadvantage of the public key systems: Compared to the secret key systems, they are based on very slow transformations, i.e. the obtainable bandwidths associated with public key cryptography are limited. A state-of-the-art dedicated hardware implementations of the DES secret key system is able to perform the transformations at a rate of up to 90 Mbit/set [Pij91]. This is close to 1000 times faster than the fastest known implementations of the RSA public key system. Indeed, *the bandwidth problem represents the most serious limitation on the practical applicability of public key systems.*

1.1.3 The RSA Public Key Crypto System

The RSA public key crypto system is the best known public key crypto system. Many authors regard the system as the most versatile system that have been proposed. The system is invented by Rivest, Shamir and Adleman [RSA78]. It was published for the first time in 1977 [Gar77].

Both the public transformation E and the private transformation D are performed by a so-called modular exponentiation. The transformations have a common modulus m. They only differ in the value of the exponent. The pair (e, m), where e is the public exponent, constitutes the public component of a user's key. Similarly, the pair (d, m), where d is the private exponent, constitutes the private component,

$$\begin{aligned} E(M) &= M^e \bmod m \\ D(M) &= M^d \bmod m, \text{ where } M \in [0; m[\end{aligned} \tag{1.1}$$

In order to obey $D(E(M)) = E(D(M)) = M$, and simultaneously to achieve that E has the properties of a trapdoor one-way function, the values of m, e and d must be selected with care. A brief introduction of how to generate the keys is given below.

In a typical application of RSA cryptography, the digital representation of the message M will be much greater than the modulus m. In this case the message is divided into a number of blocks, $M = M_1 M_2 \ldots$, where each block is less than m. Then, each block is separately transformed using E or D. Hence, a typical application consists of a (long) series of transformations, where the modulus and exponent is fixed.

Generating Keys

The following is a brief description of the basic requirements to a user's public key (e, m) and private key (d, m). For a more detailed treatment of the key generation topic, the reader is referred to e.g. [BO92, Moo92]. The keys are generated through the following steps:

1. Two large prime numbers p and q are chosen. Then, the modulus is computed as the product of the primes, i.e. $m = pq$. The so-called Euler totient function of m (see e.g. [Den82, p. 41],) is computed as well, $\varphi(m) = \varphi(p)\varphi(q) = (p-1)(q-1)$.

2. Choose an integer exponent, say e, such that $e \in [1; \varphi(m)[$ and $\gcd(e, \varphi(m)) = 1$. The last condition ensures the existence of a multiplicative inverse of e modulo $\varphi(m)$. Then, let the other exponent d be a multiplicative inverse, i.e. an integer d satisfying $ed \bmod \varphi(m) = 1$.

When the modulus m and the exponents e and d are chosen in accordance to these rules, the following equation holds for all $M \in [0; m[$,

$$D(E(M)) \quad = \quad E(D(M)) = M \tag{1.2}$$

A proof of (1.2) is included in the original article [RSA78] by Rivest, Shamir and Adleman. It should be mentioned, that several descriptions of the RSA system contain an *inadequate* proof of Equation (1.2). Even though the complete proof is pretty short, it is beyond the scope of this introductory description. So, the interested reader is referred to [RSA78].

Security of RSA

The public transformation $E(M) = M^e \bmod m$ is a one-way function since it is relatively easy to compute $E(M)$ and relatively hard to invert the transformation without the knowledge of the private exponent d. Moreover, because it is relatively easy to invert E using the private transformation D, E is a trapdoor function.

Using a very rough measure of computing time, it takes about $n = \log_2 m$ primitive operations to perform a modular exponentiation assuming all of the operands are of the same bit length n. The unit “primitive operations” is a very imprecise measure. However, the measure is satisfactory for the purpose of this introduction: To give a feeling of the computationally effort required to perform the transformations E and D, and to invert E without knowing D. Hence, the time for computing E, or D, is about n operations. The fastest known methods for inverting E without knowing d require a factorisation of m, i.e. a computation of the prime factors p and q of m. The problem of prime factorisation is believed to be a hard problem in the sense, that it is very resource demanding: One of the fastest methods, denoted the *quadratic sieve*, requires in the order of $\exp(\sqrt{n \cdot \log n})$ primitive operations to factorise an n-digit number [Nec92].

In order to obtain a sufficient degree of security of the RSA crypto system, the length of the keys must be chosen such that it becomes infeasible to factorise the modulus. Until recently, it was believed that a key length of 512 bits was adequate. However, there have been a substantial progress in the development of methods for prime factorisation: In 1977 it was estimated that several billions of years was required to factorise a number of 129 decimal digits [Gar77]. Indeed, the inventors of the RSA system challenged the public by offering a $100 prize to the first successful decoder of an encrypted message. The message was encrypted using a modulus of 129 decimal digits, or in a binary representation, 426 bits. In April 1994 the message was finally decoded in a factorisation project running over 8 months [AGLL94]. The computations was distributed to about 600 sites by means of the Internet, and a total of about 1600 machines were used. A 512 bit modulus has not yet been factorised. The authors of [AGLL94] estimate that such a project probably would require at least 100 times the computing power available in the factorisation of the 426 bit modulus. Similar projects are described in [LM89, DDLM93]. Today, a key length of 700-1000 bits is believed to sufficiently safe.

1.1.4 Other Public Key Crypto Systems

Several public key crypto systems have been proposed since public key cryptography was invented in 1976. Several of these systems have been broken, as well. It is remarkable that virtually all systems, that have not been broken, employ transformations based on exponentiation [Dif92, p. 166]. In general, the "one-way" property of transformations based on exponentiation is due to either the *prime factorisation problem* or to the *discrete logarithm problem*:

The prime factorisation problem refers to the problem of finding the prime factors of a modulus $m = pq$. As discussed above, a factorisation of m can be used for inverting the public transformations of the form,

$$E(M) = M^e \bmod m.$$

The discrete logarithm problem refers to the problem of inverting a public transformation of the form,

$$E(M) = a^M \bmod p.$$

Here, both the prime number p and the base a are part of the public key. If the result of applying E to M is denoted C, then the discrete logarithm to the base a of C modulus p reveals the message, $M = \log_a C \bmod p$.

The fastest methods for computing discrete logarithms, using n digit operands, have resource requirements of the same order as the requirements for factoring n digit moduli. Hence, the keys used in any of the transformations based on exponentiation is expected to be of about the same length. However, compared to the RSA system, some public key systems based on the discrete logarithm problem require longer operands to achieve the same degree of security [vO92].

1.1.5 Hardware Support

Shortly after the invention of the RSA public key crypto system in 1977, researchers began to develop dedicated hardware to support applications of RSA cryptography with more computing power. The very first hardware implementations of modular exponentiation were boards of discrete components. These were shortly after followed by special purpose VLSI (Very Large Scale Integrated) circuits. In 1980 Rivest, Shamir and Adleman developed a single-chip implementation for modular exponentiation of 512 bit operands [Riv80]. The chip should have been capable of encrypting at a rate of about 1.2 Kbit/sec. However, the chip never gotten to work correctly [Riv82, Riv84]. In 1981 a 336 bit chip was developed at Sandia National Laboratories, California. By combining two of these chips an encryption rate of 420 bit/set using 336 bit keys was achieved [Riv84]. In 1990, when the work presented in this thesis was initiated, the fastest known implementation had an encryption rate of 29 Kbit/sec using 512 bit keys. Today, in 1995, hardware implementations of the modular exponentiation operation have increased the available encryption rate to more than 100 Kbit/sec for key lengths of more than 512 bit. One of the implementations is a product of the work presented in this thesis. Furthermore, the development of new methods and the development of faster technologies implies that encryption rates of several Mbit/set soon will be achievable.

The voluminous literature on implementation of fast RSA cryptography witnesses the great interest of finding a solution to the bandwidth problem. Since 1980 more than twenty hardware implementations supporting RSA cryptography have been made[1], and several methods and architectures suited for hardware implementation have been proposed.[2] Some of the im-

[1][Riv80, ST83, Koc85, Bar86, GD88, HDVG88, Tho88, Bri89, ICHO89, DK90, VVDJ90, SVB91, Dif92, IWSD92, Lin, Pij92, SV93, Sch93, Oru94]

[2][NS81, WC81, Bri82, Miy82, QC82, Bla83, Slo85, MA85, Mon85, ORS$^+$86, SG86, Bak87, Sed87, KH88, Gib88, LHLH88, ZMY88, BG89, JM89, MP89, Mor89, KH90a, KH90b, Eve90, FDG90, OSA90a, WE90, Eld91, KH91, OK91, Wal91a, Wal91b, Tak92,

plementations are aimed for Smart Cards, where the available amount of circuitry is quit limited [Kno88, Mor89, dWQ90]. Methods, and hardware support, for other public key crypto systems have been developed as well [ORS$^+$86, GG90, AMOV91, ABMV93]. A description of the first years of hardware development for public key cryptography is included in [Dif92], and partial lists of existing RSA chips can be found in [Bri89, BFS91, Sch93].

1.2 Purpose of the Thesis

In 1990, when the work presented in this thesis was initiated, the following specific subgoal was set up:

> A VLSI implementation capable of performing modular exponentiation should be realised. The implementation should be able to compute the transformations used in RSA cryptography at a rate of at least 64 Kbit/sec. Hereby, it should be possible to apply real-time RSA cryptography to the data transmitted on an ISDN channel. To obtain a satisfactory degree of security, the length of the keys was specified to 561 bit. Furthermore, to enable the implementation to be embedded in telecommunication equipment, it should be implemented as a single VLSI chip. Finally, in order to demonstrate the capabilities of the chip, it should support a special interface used internally in some ISDN telephones.

Apart from describing the various decisions taken during the process of realising the VLSI chip, the purpose of this thesis is to provide the reader with an insight into the methods for performing modular exponentiation of operands of several hundreds of bits.

1.3 Chosen Approach

In the work presented in this thesis the strategy for achieving faster transformation rates for the RSA system is to develop *comptation methods*, that are more "efficient" than other known methods. The fastest implementation of RSA cryptography is achieved by constructing *dedicated hardware circuits*.

TY92, IMI92a, IMI92b, Sau92, EW93, Kor93b, OPT93, Wal93, Zha93, Kor94b, Oru95]

In contrast to methods aimed for standard micro-processors, the methods of this thesis are not constrained by a standard architecture. Indeed, the additional freedom of specifying special-purpose hardware architectures is utilised to obtain optimal solutions where the *hardware architecture are developed in harmony with the computation method*.

In general, the means for obtaining a fast hardware implementation can be divided into two independent contributions:

The technology used for the implementation has a major influence on the obtainable performance for a given computation method and architecture. One of the reasons that the hardware implementations are becoming increasingly faster, compared to the initial initiatives in 1980, is the improvement of the CMOS technology.

The effect of using a faster technology is often seen as increasing clocking frequencies of VLSI chips. A good illustration of this effect is reported in the article [IWSD92]: Ivey et al. have implemented the same architecture and computation method in two different technologies. In a bulk CMOS process they obtain a clocking frequency of 100 MHz, and in a Silicon On Insulator (SOI) CMOS process they obtain a frequency of 150 MHz. In [MP89] it is considered to use a Gallium Arsenide (GaAs) process.

The computation method and the architecture do, of course, influence the obtainable speed of an implementation. In the article reprinted in Appendix A it is observed that all of the fastest implementations known in 1990 use about the same number of clock cycles for performing a modular exponentiation. Hence, the underlying computation methods are characterised by requiring the same number of cycles, and the difference in computation speed can be attributed to the varying clocking frequencies. So, it is tempting to claim that, until 1990, the difference in the computation speed is mainly due to the difference in technology and to the skills of the VLSI circuit designers—it is not due to varying "efficiencies" of the computation methods. This basic observation is the reason for the approach chosen in this thesis: *Through the development of efficient computation methods and architectures, the speed of hardware implementations for performing modular exponentiation is increased independently of the technology chosen.* Of course, a combination of "efficient" methods and fast technologies leads to even better

	Clock	Throughput	Cycles	Reference
Years 1980-1990:				
Cryptech	14 MHz	17 Kbit/sec	$42 \cdot 10^5$	[Bri89, Sch93]
AT&T	15 MHz	19 Kbit/sec	$40 \cdot 10^5$	[Bri89, Sch93]
Thorn EM1 board	24 MHz	29 Kbit/sec	$42 \cdot 10^5$	[Tho88]
Years 1990-1995:				
Calmos Syst. Inc.	20 MHz	28 Kbit/sec	$37 \cdot 10^5$	[Sch93]
Cryptech PQR512	25 MHz	32 Kbit/sec	$40 \cdot 10^5$	[Lin]
Pijnenburg PCC200	25 MHz	40 Kbit/sec	$32 \cdot 10^5$	[Pij92]
University of Sheffield	150 MHz	92 Kbit/sec	$83 \cdot 10^5$	[IWSD92]
VICTOR	25 MHz	111 Kbit/sec	$12 \cdot 10^5$	[Oru94]
Utilisation of CRT:				
Thorn EMI board	24 MHz	56 Kbit/sec	$22 \cdot 10^5$	[Tho88]
DEC Perle-0 board	26 MHz	200 Kbit/sec	$6.7 \cdot 10^5$	[SVB91, BRV93]
DEC Perle-1 board	40 MHz	300 Kbit/sec	$6.8 \cdot 10^5$	[SV93, BRV93]
Without CRT:				
DEC Perle-0 board	26 MHz	50 Kbit/sec	$27 \cdot 10^5$	[SVB91, BRV93]
DEC Perle-1 board	40 MHz	75 Kbit/sec	$27 \cdot 10^5$	[SV93, BRV93]

Table 1.3: Existing hardware implementations performing 512 bit exponentiation.

performance.

The article of Appendix A defines an “efficiency” measure of the underlying computation methods. The measure is basically a measure of the number of clock cycles required for a modular exponentiation: A high number of cycles implies a low efficiency, and vice versa. All of the fastest hardware implementations known in 1990 had the same efficiency. It should be emphasised, that none of the slower performing implementations known in 1990 had higher efficiencies. So, in some sense, all of the fast performing implementations in 1990 used a state-of-the-art method.

In the meanwhile, since 1990 when the article of Appendix A was written, more hardware implementations have been made. Table 1.3 lists the fastest implementations known by the author.[3] The clocking frequency, the throughput (i.e. the computation rate) for a modular exponentiation using 512 bit operands, and the number of clock cycles required for an exponentiation are shown in the table. Some uncertainty in the performance of the implementations should be expected: Often the obtainable throughput depends on the actual data values. For the most common method of exponentiation, the worst case computation requires twice the number of cycles of the best case computation. For some of the implementations it is not known whether the performance refers to the best case, the average case, or the worst case. The first section of Table 1.3 lists the fastest implementations known in 1990. The second section lists the implementations that have appeared since 1990. The third, and fourth, section lists implementations that utilise the Chinese Remainder Theorem (CRT) to reduce the computational effort required to perform the private transformation of the RSA system. As described in Section 2.4 this can reduce the computing time by a factor close to four. Since the computing time for performing a general 512 bit modular exponentiation, where the CRT cannot be used, is not known for the DEC implementations, the fourth section of the table shows the expected performance when the effect of the CRT is removed.

As seen by Table 1.3, the Thorn EMI board does not fully utilise the potential of the Chinese Remainder Theorem: The number of cycles is decreased by less than a factor of two. Removing the effect of the CRT, it is seen that only four of the implementations made after 1990 have significant changes in the "efficiency" of the computation method: The Sheffield chip, the DEC boards, and the chip denoted VICTOR. The implementation of the latter chip is part of the work presented in this thesis.

- The Sheffield chip represents an approach where the "efficiency" of the

[3]In the article [SV93] the performance for the DEC Perle-1 implementation is specified as 600 Kbit/sec. Therefore, many authors have been lead to the belief, that the DEC Perle-1 implementation is capable of performing a single 512 bit modular exponentiation in 0.85 msec. This is, however, not the case: According to a personal communication on July 4 1995 with Mark Shand, one of the authors of [SV93], the specified throughput of 600 Kbit/sec is an *estimate of the total performance of two independent modular exponentiation units, each performing a 512 bit exponentiation*. Therefore, a throughput of 300 Kbit/sec must be expected for a single exponentiation unit.

The throughput of 600 Kbit/sec have never been measued for the DEC Perle-1 implementation. However, a throughput of 185 Kbit/sec have been measured for a single unit performing modular exponentiations using 970 bit operands.

computation method has *decreased* in comparison to the implementations made prior to 1990. On the other hand, the clocking frequency is significantly higher than the rest of the implementations listed in Table 1.3. This illustrates the fact, that the "efficiency" measure is a bad stand-alone measure of the performance potential of a computation method: Even though a fast technology is used for implementing the Sheffield chip, the high clocking frequency is partly due to a short so-called *critical path* of the circuitry, i.e. the longest delay of the circuitry activated in a clock cycle.

- The VICTOR chip, and the DEC Perle-0 board, represents an approach where the increased performance is obtained by an increased "efficiency" of the computation method. Even though the technology used to implement the VICTOR chip is expected to be faster than the technologies used prior to 1990, the clocking frequency has not increased significantly. This indicates that the cost of using a more "efficient" computation method is a relatively longer critical path.
- The varying performance of the two DEC boards is not due to a difference in the "efficiency". The variation is expressed by a difference in clocking frequency. However, it is not solely due to the variation of the technology used for the implementation: In the Perle-1 implementation another computation method with the same "efficiency" as the Perle-0 and a shorter critical path has been used.

Hence, *to obtain a high performance of a computation method, and the associated hard-ware architecture, both the required number of cycles and the critical path of the circuitry must be considered.* For a further discussion on how to measure the performance, and on how to separate the contributions from technology and method, the reader is referred to e.g. [PH94, Chapter 2]. In the following chapters of the thesis, the term "efficiency" will not refer to a specific well-defined measure of performance.

1.4 Organisation of the Thesis

The thesis consists of two parts: The first part comprises six chapters, and the second part comprises five appendices. Each appendix contains a paper that has been written during the period from 1990 to 1995. The overall aim of the first part is to provide the reader with an insight into the research on computation methods for performing modular exponentiation. Furthermore, the aim is to report the work done by the author. Through a discussion of the existing literature on the research topic, the contributions represented by the papers in the appendices is set into a consistent frame. The chapters of the first part are on various topics. They can be read independently of each other. Each chapter ends with a summary and a discussion of the topic. It is assumed that the reader is familiar with the basic methods of computer science. The papers in the appendices assume some knowledge of the methods and techniques known from the fields of computer arithmetic and VLSI design.

The first four chapters are structured as a hierarchal presentation where the problems identified at one level are solved by introducing a new set of problems at a lower level. The lowest level in the hierarchy is the hardware level, where the solutions are realised as a hardware architecture capable of executing a specific computation. The fifth chapter describes a particular efficient computation method. The method can be viewed as the result of combining the experiences gained from the preceeding chapters:

Chapter 1 gives a brief motivation to the subject of this thesis by considering applications of cryptography. The principles of public key cryptography are briefly introduced with an emphasis on the RSA crypto system. The problem of achieving sufficiently fast computation of modular exponentials is identified.

Chapter 2 is a relatively exhaustive description of methods for computation of exponentials. Since an exponentiation is performed by a series of multiplications the focus is directed toward methods using as few multiplications as possible and toward methods that can utilise a parallel computation scheme. An effort has been put into identifying various properties of the multiplication operation and explaining how these properties can be utilised to achieve efficient methods for computation of exponentials.

Chapter 3 treats modular multiplication—the arithmetical operation used in modular exponentiation. As the previous chapter, this chapter contains a relatively exhaustive description of the methods and techniques used for computing modular products. The important concept of "representation" is introduced, and it is described how the properties of the chosen representation can be utilised to achieve efficient methods for addition, subtraction, multiplication and division. These are the fundamental operations used in modular multiplication. The problem of determining quotient digits is treated in detail. It turns out that this is one of the major problems in the approach of high-radix modular multiplication described by the papers in the appendices.

Chapter 4 is a description of a project of implementing a VLSI processor for computing modular exponentials. The style of the chapter is more descriptive than discussing. The chapter includes a description of the history of the project, the hardware architecture and the computation methods, and the results of the tests and performance measurements.

Chapter 5 describes an efficient solution of the problem of determining quotient digits. The solution represents a break-through in the high-radix approach followed by the author. The chapter describes how the performance of future hardware implementations of modular exponentiation can be increased by more than an order of magnitude compared to the fastest implementations known today.

Chapter 6 contains a brief conclusion on the work presented in this thesis.

1.5 Description of Papers

As previously mentioned the second part of the thesis consists of five papers written during the period from 1990 to 1995 In the Appendices A through E the original papers are printed in the original typesetting. Except from the paper in Appendix E, all of the papers are research articles. The paper in Appendix E is a document providing some of the essential data on the VLSI processor described in Chapter 4. The following listing of the papers provides information on co-authorship, publication status etc.,

1. Holger Orup, Erik Svendsen, and Erik Andreasen, "VICTOR, an Efficient RSA Hardware Implementation", in Ivan B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 245–252, Aarhus, Denmark, May 21–24 1990. Springer-Verlag, Berlin, 1991.

This article includes a short motivation for focussing on more efficient computation methods. The basic idea of using high-radix modular multiplication is introduced. However, the term “multiple bit scan” is used in place of “high-radix”. It is noteworthy that the estimated speed of a suggested VLSI implementation is quit close to the speed of the fabricated VLSI processor described in Chapter 4.

2. Holger Orup and Peter Kornerup, “A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N”, in Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 51–56, Grenoble, France, June 26–28 1991. IEEE Computer Society Press, Los Alamitos, California, 1991.

   In this article the terminology known from the field of computer arithmetic is used. The article suggests an extended use redundant representations. Furthermore, a computation schedule based on pipelining is proposed.

3. Holger Orup, “Area Reduction for Bit-Sliced Layouts using a Commercial Development System”, Unpublished article, Department of Computer Science, University of Aarhus, 1994.

   This article describes the experiences obtained from the work of reducing the area of the VLSI processor. The various techniques for reducing the area, and the effect of applying them, are described.

4. Holger Orup, “Simplifying Quotient Determination in High-Radix Modular Multiplication”, in Simon Knowles and William H. McAllister, editors, *Proceedings. 12th IEEE Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, California, 1995.

   This article describes how a combination of optimisation techniques leads to a very simple quotient determination in high-radix modular multiplication. Furthermore, a pipelining technique is utilised to obtain a very short critical path in the hardware architecture.

5. Holger Orup, “RSA Processor, Preliminary Engineering Data”, Internal document, Department of Computer Science, University of Aarhus, 1993.

This is a document that provides some of the essential data of the VLSI processor. It should be emphasised, that the document *in no means* pretends to be a satisfactory complete product description. It is a document providing preliminary descriptions of a prototype. The document is included in order to provide an impression of the functionality of the VLSI processor.

Chapter 2

Exponentiation

Exponentiation refers to the process of evaluating exponentials or powers b^e, where b is the base and e is the exponent. The eth power of b is defined recursively by

$$\begin{aligned} b^0 &= 1_G \\ b^e &= b^{e-1} *_G b,\ e \in \{1, 2, 3, \dots\}, \end{aligned} \tag{2.1}$$

where b is element in a set G with a *mutiplication* composition $*_G : G \times G \mapsto G$ and $1_G \in G$ is a neutral element for multiplication. An example is the set $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$ with the multiplication composition *modular multipication* $(x \cdot y)$ mod n and the neutral element 1. In this case, Equation 2.1 defines modular exponentials as in the RSA crypto system. The RSA crypto system is described in Subsection 1.1.3. Taking into consideration, that this thesis primarily is directed toward efficient methods for computing modular exponentials, the definition seems very general. But, since the methods of this chapter only uses a few properties of modular multiplication, the methods apply to all sets, where a multiplication composition is defined. Other sets, that are used in crypto systems, are the Galois Fields GF(2^n), where the elements are polynomials and the multiplication composition is defined as polynomial multiplication modulo an irreducible polynomial [Den82, p. 48].

A common characteristic for the computation of exponentials in crypto systems is the very large exponents. E.g., at present, exponent values in the range from 2^{512} to 2^{1024} are considered to be necessary to achieve a sufficient degree of security for the RSA system. In the future, even larger exponent values may be necessary.

A straight forward method for exponentiation is derived from Equation 2.1. It will be called the *unary method*,

$$\begin{aligned} b^0 &= b^{e-1} * b \qquad (2.2) \\ &= (\cdots((1*b)*b)\cdots)*b. \end{aligned}$$

To simplify the notation the subscripts of the multiplication composition and the neutral element are made implicit. The unary method needs e multiplications and two registers: One register for b and one for intermediate results. Since $(1*b)$ equals b the very first multiplication can be avoided, resulting in $e-1$ multiplications when $e>0$.

In the following, exponentiation methods will be discussed and compared in terms of the resource requirements: *Computing time* and *memory requirements*. The *computing time* is expressed as the necessary number of multiplications. It is assumed that the time for other operations such as additions and comparisons are negligible. In certain algebraic systems there may be a substantial difference in the computing time for a general multiplication and a squaring. E.g. Agnew et al. [AMV88] utilise that squaring can be performed much faster than multiplication in GF(2^n). Consequently, whenever feasible, the total computing time will be split into a number of multiplications and a number of squarings. The *memory requirements* will be expressed as the number of registers needed, where a register is assumed to contain a single element from the set G over which exponentiation is performed. This is a rough measure since the binary encoding of two elements may require very different amounts of space in terms of bits. Indeed, this happens if exponentiation is performed by integer multiplication over $\mathbb{N}$, the non-negative integers. However, this rough measure is useful in the present context, since in most crypto systems the set of elements is finite and a modulo reduction is part of the multiplication. Memory for the exponent and for constants, such as the neutral element 1, will be implicit in the discussions of memory consumption. Furthermore, the *required number of processing elements* will be part of the resource requirements when parallel methods are described.

This chapter is divided into five sections. Section 2.1 describes different sequential computation methods. It is shown that if the multiplication composition is associative the computing time can be reduced to a logarithmic number of multiplications. Moreover, it is utilised that by precomputing often used values and by saving these in a table the computing time can be further reduced. Section 2.2 gives some theoretical lower bounds on the number of multiplications. This is used to access how well the different sequential computation methods perform. In Section 2.3 parallel computation methods are described. It turns out that a pipelined method is superior to the fastest sequential methods, both with respect to computing time and to hardware consumption. Section 2.4 describes how the algebraic properties of modular multiplication can be utilised to speed up the computation of modular exponentials in the RSA crypto system. Finally, Section 2.5 contains a summary and a discussion.

2.1 Fewer Multiplications

Compared to the unary method the number of multiplications can be dramatically reduced if the multiplication composition is associative, i.e. $(a*b)*c = a*(b*c)$. This property implies that $b^{2e} = b^e * b^e = (b^e)^2$. Hence, if the exponent is even the number of multiplications can be reduced to nearly half. Furthermore, if the exponent is odd the rule $b^{2e+1} = (b^e)^2 * b$ also reduces the computationally effort. These rules can be used recursively and the result is a logarithmic number of multiplications. This is the well known *binary method* described by Knuth [Knu81, p. 441]. According to Knuth the binary method was described as early as 200 B.C.. If the exponent is binary encoded as a string of n bits, $e_{n-1}e_{n-2}\dots e_0$, the value of e can be expressed as

$$\begin{aligned} e &= \sum_{i=0}^{n-1} e_i 2^i,\ \ e_i \in \{0,1\}, e_{n-1} > 0 \qquad (2.3) \\ &= ((\cdots((e_{n-1})2 + e_{n-2})2 + \cdots)2 + e_1)2 + e_0, \end{aligned}$$

using Horner's rule. Because of the constraint $e_{n-1} > 0$ the trivial case, where $e = 0$, is disregarded. This constraint ensures that the string of binary digits does not contain any leading zeroes and, hence, that $n-1$ equals $\lfloor \log_2 e \rfloor$. The eth power of b can now be written as

$$\begin{aligned} b^e &= b^{((\cdots((e_{n-1})2+e_{n-2})2+\cdots)2+e_1)2+e_0} \qquad (2.4) \\ &= ((\cdots((b^{e_{n-1}})^2 * b^{e_{n-2}})^2 * \cdots)^2 * b^{e_1})^2 * b^{e_0}. \end{aligned}$$

This shows that exponentiation can be performed with $n-1$ squarings and $n-1$ multiplications. The memory requirement is one register for b and another for the intermediate results. Since $n-1$ equals $\lfloor \log_2 e \rfloor$ the method is logarithmic.

If bit e_i is 0 the multiplication by b^{e_i} reduces to a multiplication by 1 and can be neglected. Usually the number of bits, that are non-zero in the binary encoding of e, is denoted $\nu(e), \nu(e) = \sum_i e_i \in \{1, 2, \ldots, \lfloor \log_2 e \rfloor + 1\}$. This function is called the *Hamming weight* of e. The number of squarings and multiplications can then be expressed by respectively $\lfloor \log_2 e \rfloor$ and $\nu(e) - 1$. Since the method scans the exponent from most significant bit to least significant bit it is denoted the *left-to-right* binary method.

It is also possible to scan the exponent from right to left. Still using a binary encoding of e, the eth power of b can be written as

$$\begin{aligned} b^e &= b^{e_0 2^0 + e_1 2^1 + \cdots + e_i 2^i + \cdots + e_{n-1} 2^{n-1}} \qquad (2.5) \\ &= (b^{2^0})^{e_0} * (b^{2^1})^{e_1} * \cdots * (b^{2^i})^{e_i} * \cdots * (b^{2^{n-1}})^{e_{n-1}}. \end{aligned}$$

The sequence of squares, $b^{2^0}, b^{2^1}, \ldots, b^{2^i}, \ldots, b^{2^{n-1}}$, can be computed in $n-1$ squarings by using the rule $b^{2^i} = (b^{2^{i-1}})^2$. After computation of the ith square, b^{2^i}, exponent bit e_i is inspected. If it is 1 the ith square is multiplied onto an intermediate result, giving $b^{e_0 2^0 + e_1 2^1 + \cdots + e_i 2^i}$. Again, the method uses $[log_2\ e]$ squarings and $\nu(e) - 1$ multiplications. It requires two registers: One register for the squares and one for the intermediate results. As described in Section 2.3, the right-to-left method is easy to rewrite into a parallel method.

Shand and Vuillemin compare the left-to-right and the right-to-left binary method [SV93]. It is stated that the right-to-left method requires two registers while the left-to-right method gets away with only one register. It is correct that the left-to-right method only has a single intermediate result. But, contrary to the right-to-left method, the left-to-right method also requires a register for b which is needed through the whole computation.

2.1.1 The m-ary Method

Knuth generalises the left-to-right binary method to a left-to-right *m-ary method*. For $m > 2$ the exponent is m-ary encoded as a string of n digits, such that e is expressed as

$$\begin{aligned} e &= \sum_{i=0}^{n-1} e_i m^i, \ e_i \in \{0, 1, \ldots, m-1\}, e_{n-1} > 0 \qquad (2.6) \\ &= ((\cdots((e_{n-1})m + e_{n-2})m + \cdots)m + e_1)m + e_0, \end{aligned}$$

which for $m = 2$ is identical to Equation (2.3). The unary method is also a specialisation of the m-ary method: Let $m = 1$ and let all digits e_i in the unary encoding be 1. This means that the unary encoding is a string of e 1's. Hence, the following m-ary computation method applies to all $m > 1$:

$$\begin{aligned} b^e &= b^{((\cdots((e_{n-1})m+e_{n-2})m+\cdots)m+e_1)m+e_0} \\ &= ((\cdots((b^{e_{n-1}})^m * b^{e_{n-2}})^m * \cdots)^m * b^{e_1})^m * b^{e_0}. \end{aligned} \tag{2.7}$$

When $m > 2$ the exponent digits take values from the set $\{0, 1, 2, \ldots, m-1\}$. Consequently, the powers b^{e_i} are no longer the simple values 1 and b. Instead of computing b^{e_i} for each i, implying that the same power of b may be recomputed several times, they can be *precomputed* and stored into a table of size $m - 1$. Stinson [Sti90] describes exponentiation methods for an algebraic system where the time for squaring is negligible compared to multiplication. One of Stinson's results is an algorithm that computes all powers $b, b^2, b^3, \ldots, b^{m-1}$ in not more than $\frac{1}{2}m - 1$ multiplications, where m is assumed to be a power of two. The algorithm starts from b and proceeds to the next even power by using the rule $b^e = (b^{\frac{e}{2}})^2$. This costs a squaring. Then the next odd power is computed by the rule $b^e = b^{e-1} * b$ at a cost of a multiplication. By alternating application of these two rules the complete table is builded up. If Stinson's algorithm is applied to general values of m, i.e. values that are not restricted to powers of two, and if the squarings are brought into the analysis a total of $m - 2$ operations is used: $\lfloor \frac{1}{2}(m-2) \rfloor$ multiplications and $\lfloor \frac{1}{2}(m-1) \rfloor$ squarings. The unary method also requires a total of $m - 2$ operations to compute the table. However, only one of these operations is a squaring.

After the table is precomputed the remaining computation uses $\nu_m(e) - 1$ multiplications and $n - 1 = \lfloor \log_m e \rfloor$ exponentiations with the (small) exponent m. The function $\nu_m(e) \in \{1, 2, \ldots, \lfloor \log_m e \rfloor + 1\}$ denotes the number of non-zero digits in the m-ary encoding of e. It is convenient to choose m to be a power of two, i.e. $m = 2^k$ for some k. Then the exponentiations with exponent m can be performed by $k = \log_2 m$ squarings. This gives a total of $(\log_2 m)\lfloor \log_m e \rfloor + \lfloor \frac{1}{2}(m-1) \rfloor$ squarings and $\nu_m(e) - 1 + \lfloor \frac{1}{2}(m-2) \rfloor$ multiplications. Besides storage for the table a register for the intermediate results is needed, a total of m registers. When $m = 2^k$ the digits e_i of an exponent e can be interpreted as groups of k bits in a binary encoding of the exponent. This is the reason that the m-ary method is also named the *k-window method*. It should be mentioned that an exponentiation method identical to the k-window method was described by Brauer [Bra39] in 1939. Brauer developed the method to improve an upper bound on the number of multiplications needed for computing exponentials.

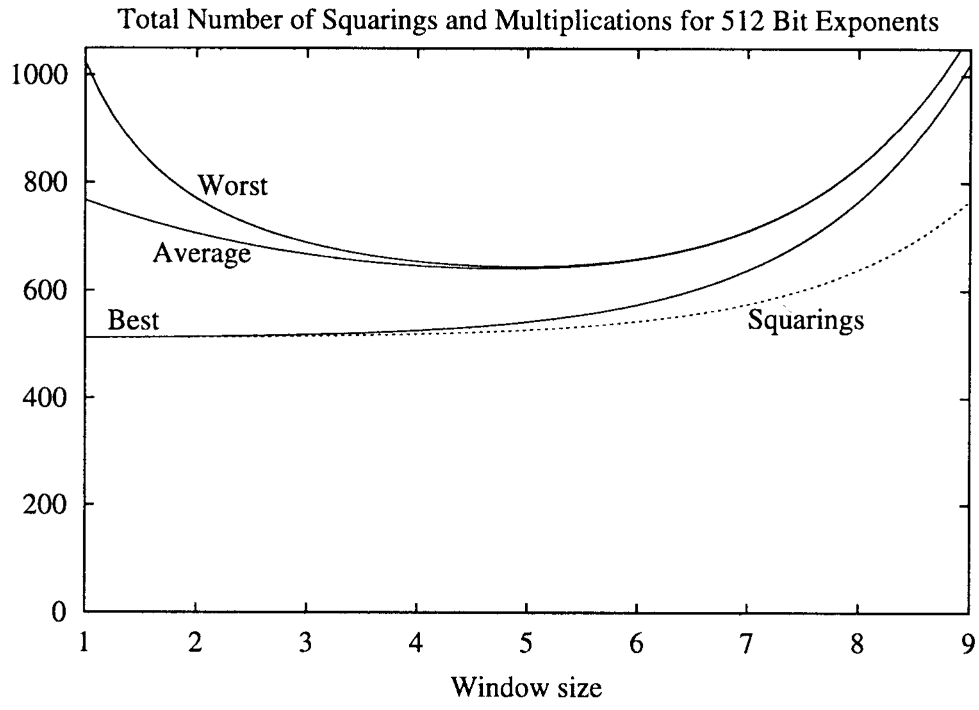


Figure 2.1: The total number of operations for the m-ary method in the worst, average and best case. Also shown is the number of squarings.

Figure 2.1 shows the computing time for different sizes of the window $k = \log_2 m$ when e is a 512 bit number. The curves are derived from the above expressions, which are based on the assumption that m is a power of two. So for non-integer window sizes the curves are approximations. The worst case time is for all digits e_i non-zero. The best case time is when all but the most significant digit are zero. The best case time does not represent a lower bound for the computing time for exponentiation but illustrates the interval of computing time for various exponent values when the m-ary method is applied. Indeed, when only the most significant digit is non-zero there is no reason to precompute a table that is never used. Also shown in the figure is the average time: If the digits are random in the set $\{0, 1, \ldots, m-1\}$ the

$\lfloor \log_2 e \rfloor + 1$	Window size	Squarings	Multiplications	Total	Registers
256	4	259	70	329	16
512	5	525	117	642	32
1024	6	1051	201	1252	64

Table 2.1: Minimal, worst case, computing time for m-ary method.

probability for a non-zero digit is $\frac{m-1}{m}$. Recalling that $e_{n-1} > 0$, this implies that $\nu_m(e) = \frac{m-1}{m} \lfloor \log_m e \rfloor + 1$ in the average case. The number of squaring operations is also depicted in the figure. If similar curves for 256 bit and 1024 bit exponents are plotted these will be shaped very much like the curves for 512 bit exponents.

In Table 2.1 the minimal total number of operations, in the worst case, is listed for 256, 512 and 1024 bit exponents. With respect to the worst case computing time a window size of 5 is optimal for 512 bit exponents. Compared to the binary method (window size 1) the worst case total number of operations is reduced by 37 percents for 512 bit exponents when a window size of 5 is chosen. Although the optimal window size decreases with decreasing exponent bit-lengths and increases for increasing exponent bit-lengths a window size of 5 gives a worst case total number of operations that is fairly close to minimum for exponents with bit-lengths from 256 bits to 1024 bits.

In the m-ary method the reduction in computing time time is achieved at the expense of additional registers for holding a table of precomputed values. The main contribution to the reduction is the reduced number of *multiplications* performed in the last phase of the computation, i.e. in the phase after the precomputation. In this phase the number of multiplications is reduced by approximately a factor of $k = \log_2 m$ from $\nu(e) - 1$ to $\nu_m(e) - 1$. But also the number of *squarings* performed in this phase is (sometimes) slightly reduced from $\lfloor \log_2 e \rfloor$ to $(\log_2 m) \lfloor \log_m e \rfloor$. Since $\lfloor \log_2 e \rfloor$ equals $(\log_2 m) \lfloor \log_m e \rfloor + \lfloor \log_2 e_{n-1} \rfloor$ the reduction in the number of squarings is $\lfloor \log_2 e_{n-1} \rfloor \in \{0, 1, \ldots, k-1\}$. This means that the number of bits, $\lfloor \log_2 e_{n-1} \rfloor + 1$, used in the binary encoding of the most significant digit e_{n-1} determines the reduction of squarings. Since the value $b^{e_{n-1}}$ has been precomputed and stored into a table the savings in squarings corresponds to a reuse of squarings already performed during the precomputation. It is possible to get full advantage of these squarings already performed and always achieve a total reduction of $k - 1$ squarings, independent of the number of bits used for e_{n-1}. The trick is to allocate windows to the binary encoding of exponent e from left to right instead of from right to left. The following illustration shows how windows are allocated from right to left in accordance with Equation 2.6.

$$e = \overbrace{\boxed{\underbrace{00\ldots0}_{j}1x\ldots x}}^{e_{n-1}}\ \overbrace{\boxed{xx\ldots xxx\ldots x}}^{e_{n-2}}\cdots\overbrace{\boxed{xx\ldots xxx\ldots xx}}^{e_1}\ \overbrace{\boxed{xx\ldots xxx\ldots x}}^{e_0} \tag{2.8}$$

The bits denoted by x symbolise an arbitrary bit value. Since all windows have the same window size k the most significant digit e_{n-1} is padded with j zero-bits, so that the most significant window has $k = j + \lfloor \log_2 e_{n-1} \rfloor + 1$ bits in total. Now, if the windows are allocated from left to right instead and the least significant digit e_0 is padded with j zero-bits the following picture is obtained

$$e' = \overbrace{\boxed{1x\ldots xxx\ldots x}}^{e'_{n-1}}\ \overbrace{\boxed{xx\ldots xxx\ldots x}}^{e'_{n-2}}\cdots\overbrace{\boxed{xx\ldots xxx\ldots xx}}^{e'_1}\ \overbrace{\boxed{xx\ldots x\underbrace{00\ldots0}_{j}}}^{e'_0} \tag{2.9}$$

Since $e' = 2^j e$ the computation of b^e can be done by using the following expression for e instead of (2.6)

$$e = \frac{e'}{2^j} = ((\cdots((e'_{n-1})m + e'_{n-2})m + \cdots)m + e'_1)\frac{m}{2^j} + \frac{e'_0}{2^j}, m = 2^k \tag{2.10}$$

The number of squarings is seen to be $k(n-1) - j = k(n-1) + \lfloor \log_2 e_{n-1} \rfloor - (k-1) = \lfloor \log_2 e \rfloor - (k-1)$, which independent of the number of bits used for e_{n-1}. The number of multiplications is unchanged. If Equation (2.10) is used for the computation of exponentials a further reduction of the computing times in Table 2.1 is obtained. The reduction is 3 squarings for 512 bit exponents and 2 squarings for 1024 bit exponents. There is no change in the computing times for 256 bit exponents. This is shown in Table 2.2.

The direction in which the windows are allocated should not be confused with direction in which the exponent digits are scanned. In the above description of the m-ary method the exponent digits are scanned from left to right. Thus the description is still a generalisation of the left-to-right binary method.

$\lfloor \log_2 e \rfloor + 1$	Window size	Squarings	Multiplications	Total	Registers
256	4	259	70	329	16
512	5	522	117	639	32
1024	6	1049	201	1250	64

Table 2.2: Minimal, worst case, computing time for m-ary method when windows are allocated left-to-right.

Knuth writes [Knu81, p. 444] that there is also a less obvious right-to-left m-ary method that takes more registers but only a few more operations compared to the left-to-right method. In fact, it is possible to derive a right-to-left method that requires exactly the same number of registers and operations as the left-to-right method. The strategy follows an idea by Yao [Yao76]. Using an m-ary encoding of e, the eth power of b is written as,

$$\begin{aligned} b^e &= b^{e_0 m^0 + e_1 m^1 + \cdots + e_i m' + \cdots + e_{n-1} m^{n-1}} \\ &= (b^{m^0})^{e_0} * (b^{m^1})^{e_1} * \cdots * (b^{m^i})^{e_i} * \cdots * (b^{m^{n-1}})^{e_{n-1}}. \end{aligned} \tag{2.11}$$

If m is a power of two, say $m = 2^k$, the sequence $b^{m^0}, b^{m^1}, \ldots, b^{m^i}, \ldots, b^{m^{n-1}}$, can be computed in $(n-1)k$ squarings. Instead of using registers for a precomputed table, $m-1$ registers are used for intermediate results. The registers are initialised to 1 and are denoted $c_1, c_2, \ldots, c_{m-1}$. After the computation of the ith sequence element, b^{m_i}, exponent digit e_i is inspected. Digit e_i can take one of the values in $\{0, 1, \ldots, m-1\}$. If $e_i = j$ and $j > 0$ the ith sequence element is multiplied onto c_j. In this way all sequence elements with *common exponent digit value* j are multiplied onto c_j. It can be expressed as

$$c_j = 1 * \prod_{i:e_i=j} b^{m^i} \ , \text{ for all } j = 1, 2, \ldots, m-1 \tag{2.12}$$

Since the first multiplication onto each c_j is a multiplication onto 1 and therefore negligible, the number of multiplications required to compute $c_1, c_2, \ldots,$ c_{m-1} is $\nu_m(e) - \delta$. Here δ denotes the number of different digit values from $\{1, 2, \ldots, m-1\}$ represented by the exponent digits, i.e. the number of non-empty index sets $\{i : e_i = j\}$ in (2.12).

The final phase of the computation uses an algorithm by Brickell et al. [BGMW92]. In terms of the intermediate results, the eth of b can be expressed as

$$\begin{aligned} b^e &= c_{m-1}^{m-1} * c_{m-2}^{m-2} * \cdots * c_1^1 \qquad (2.13)\\ &= (c_{m-1}) * (c_{m-1} * c_{m-2}) * \cdots * (c_{m-1} * c_{m-2} * c_{m-3} \cdots * c_1).\\ &= c'_{m-1} * c'_{m-2} * \cdots * c'_1. \end{aligned}$$

This phase of the computation is accomplished by two sweeps through the intermediate results. Start by setting $c'_{m-1} = c_{m-1}$ and then calculate $c'_{m-2} = c'_{m-1} * c_{m-2}$, $c'_{m-3} = c'_{m-2} * c_{m-3}, \dots, c'_1 = c'_2 * c_1$ in $\delta - 1$ multiplications. If the value c'_j is stored in register c_j no further registers are required. Through the second sweep $c'_{m-2} = c'_{m-1} * c'_{m-2}$, $c'_{m-3} = c'_{m-2} * c_{m-3}, \dots, c'_1 = c'_2 * c'_1$ are calculated in $m - 2$ multiplications. Now the value c'_1 equals b^e. In total, including a register for computing the sequence values b^{m^i}, m registers are required. The total number of operations is $(n-1)k$ squarings plus $\nu_m(e)+m-3$ multiplications. The only difference from the left-to-right m-ary method is how the operations are divided into squarings and multiplications.

The right-to-left m-ary method described above uses a right-to-left window allocation. By using a left-to-right window allocation the number of squarings also reduces to $\lfloor \log_2 e \rfloor - (k-1)$ for the right-to-left m-ary method.

2.1.2 Thurber's Modification

The m-ary method can be modified, so that the computing time and the required number of registers is decreased. An observation, by Thurber [Thu73], that halves the table size for a given m reduces the time for precomputing the table and therefore the total computing time. Thurber observes that only the odd powers, $b, b^3, b^5, \dots, b^{2\lfloor \frac{1}{2}m \rfloor - 1}$, are needed in the table. These $\lfloor \frac{1}{2}m \rfloor$ powers can be precomputed by the rule $b^{2e+1} = b^{2e-1} * b^2$, which requires a single squaring and $\lfloor \frac{1}{2}(m-2) \rfloor$ multiplications. The modification is based on a rewriting of all non-zero exponent digits, such that e_i is written as $2^{r_i} e'_i$, where e'_i is odd. The number of zero-bits in the least significant end of the binary encoding of e_i is expressed by r_i, which takes a value from $\{0, 1, \dots, \lfloor \log_2 e_i \rfloor\}$. As an example, assume the binary encoding of e_i is 101000, then r_i is 3 and e'_i is 5. In general, assume that all non-zero digits are expressed as $x_i e'_i$ where $x_i \in \{1, 2, \dots, m-1\}$ and $b^{e'_i}$ is a precomputed power. Equation (2.6) is then written as

$$
\begin{aligned}
e &= ((\cdots((x_{n-1}e'_{n-1})m + x_{n-2}e'_{n-2})m + \cdots)m + x_1e'_1)m + x_0e'_0 \quad (2.14)\\
&= (((\cdots((e'_{n-1}\frac{x_{n-1}}{x_{n-2}}m + e'_{n-2})\frac{x_{n-2}}{x_{n-3}}m + \cdots)\frac{x_2}{x_1}m + e'_1)\frac{x_1}{x_0}m + e'_0)x_0.
\end{aligned}
$$

In Thurber's special case where $x_i = 2^{r_i}$ and $m = 2^k$ this leads to the following expression of the exponent e,

$$
\begin{aligned}
e \;=\; &(((\cdots((e'_{n-1})2^{k+r_{n-1}-r_{n-2}} + e'_{n-2})2^{k+r_{n-2}-r_{n-3}} \\
&+ \cdots)2^{k+r_2-r_1} + e'_1)2^{k+r_1-r_0} + e'_0)2^{r_0}.
\end{aligned} \quad (2.15)
$$

Apart from the precomputation the number of squarings in this exponentiation method is $(k + r_{n-1} - r_{n-2}) + (k + r_{n-2} - r_{n-3}) + \cdots + (k + r_1 - r_0) + r_0$ which reduces to $(n-1)k + r_{n-1}$. This is r_{n-1} squarings more than used in the m-ary method with right-to-left window allocation. However, it is possible to get rid of the r_{n-1} extra squarings. By rewriting the beginning of the exponentiation based on expression (2.15)

$$
\cdots(b^{e'_{n-1}})^{2^{k+r_{n-1}-r_{n-2}}} * \cdots \;=\; \cdots(b^{2e_{n-1}})^{2^{k-1-r_{n-2}}} * \cdots, \quad (2.16)
$$

it is seen that the first $k + r_{n-1} - r_{n-2}$ squarings have been transformed into $k - 1 - r_{n-2}$ squarings plus the calculation of $b^{2e_{n-1}}$. This calculation is performed by a single multiplication: If e_{n-1} is odd then $b^{e_{n-1}}$ is in the table and $b^{e_{n-1}} * b^{e_{n-1}} = b^{2e_{n-1}}$. If e_{n-1} is even then both $b^{e_{n-1}-1}$ and $b^{e_{n-1}+1}$ are in the table and $b^{e_{n-1}-1} * b^{e_{n-1}+1} = b^{2e_{n-1}}$. Hence, the $(n-1)k + r_{n-1}$ squarings is transformed into $(n-1)k-1$ squarings and a single multiplication. Furthermore, by allocating the windows left-to-right as in Section 2.1.1 it is possible to reduce the $(n-1)k - 1$ squarings to $\lfloor \log_2 e \rfloor - k$ squarings. The number of multiplications is $\nu_m(e) - 1$ plus an extra multiplication for the above calculation. In total, including the precomputation, Thurber's modification of the m-ary method performs an exponentiation in $\lfloor \log_2 \rfloor - (\log_2 m - 1)$ squarings and $\lfloor \frac{1}{2}(m-2) \rfloor + \nu_m(e)$ multiplications. Including a register for the intermediate results a total of $\lceil \frac{1}{2}m \rceil$ registers are required. For a fixed value of m this method is an improvement over the m-ary method with left-to-right window allocation by $\lfloor \frac{1}{2}(m-1) \rfloor$ squarings and $\lceil \frac{1}{2}m \rceil$ registers while the number of multiplications has increased by one. In Table 2.3 the minimal total number of operations, in the worst case, is listed for 256, 512 and 1024 bit exponents. Note that Thurber's modification has increased the optimal

$\lfloor\log_2 e + 1$	Window size	Squarings	Multiplications	Total	Registers
256	5	251	67	318	16
512	6	506	117	623	32
1024	6	1018	202	1220	32

Table 2.3: Minimal, worst case, computing time for Thurber's modification.

window size for 256 and 512 bit exponents. Compared to the m-ary method in Table 2.1 the computing time for 512 bit exponents has been reduced by 3 percents.

No exponentiation method have been proposed that uses a *fewer number of operations* than is used in Thurber's modification of the m-ary method, when nothing can be assumed about the value of exponent e and base b. Although many other methods require fewer operations for special exponent values or base values, no method performs better in the worst case.

When discussing the m-ary method and Thurber's modification some authors note that in the computation of b^e it may happen that some of the precomputed table values are never used. Thus, by avoiding the precomputation of these unused values the table size will be reduced and the number of operations for performing the precomputation might be reduced. The actual number of unused values depends on the value of the exponent in use. According to [HPS71, pp. 44–46] the probability $p_k(r,n)$ that exactly k out of the r precomputed values are unused when n random table lookups are done can be expressed as,

$$p_k(r,n) = \binom{r}{k} \sum_{j=0}^{r-k} (-1)^j \binom{r-k}{j} \left(1 - \frac{j+k}{r}\right)^n, \text{ where } 1 \le r \le n. \quad (2.17)$$

Figure 2.2 shows how the number of unused table values is distributed when the m-ary method and Thurber's modification is applied for exponentiation with random 512 bit exponents. For both methods the table size is 31 and the average number of table lookups is approximately equal to the number of non-zero m-ary digits in the exponent, $\frac{m-1}{m}\lfloor \log_m e \rfloor + 1$. This gives 100 lookups for the m-ary method and 85 lookups for Thurber's modification. The expected number of unused table values, $\sum k \cdot p_k(r,n)$, is 1.2 for the m-ary method and 1.9 for Thurber's modification. As seen in Figure 2.2 the probability that more than 4 values are unused is very small for both

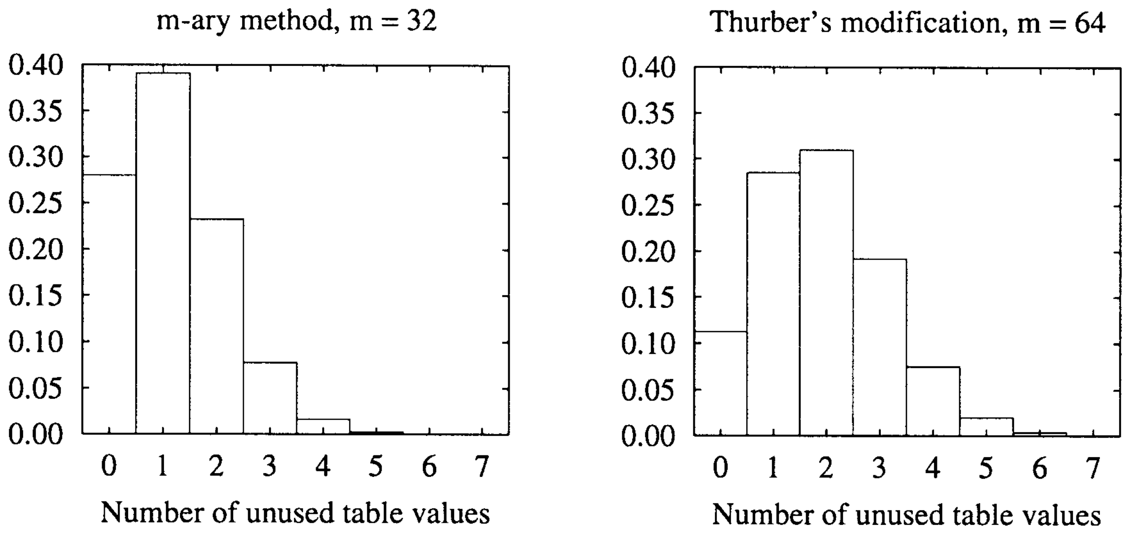


Figure 2.2: Probability distribution of the number of unused table values for random 512 bit exponents.

methods and, hence, the reduction in resource requirements is minor. Furthermore, unless the unused table values are the last computed in the original precomputation, it is not obvious that the absent of these values will lead to a reduced precomputation time.

2.1.3 Methods Based on Heuristics

The strength of the m-ary method and Thurber's modification is the upper bound on the worst case computing time and on the number of required registers. Furthermore, the methods are easy to express and therefore they are easy to implement in a computer program or into a dedicated hardware circuitry. For both methods the resource requirements in the average case are very close to the requirements in the worst case.

In the methods based on heuristics the aim is to find a good *individual computation rule* for each individual exponent value such that a better performance is obtained. For the particular exponent value in consideration, heuristics are used to find a computation rule which, hopefully, is superior to the rules obtained from other methods. Of course the rules can be compared, such that the best one is selected for the exponentiation. The drawbacks of the heuristic methods are the complicated descriptions and, for some heuristics, the required computing time for finding a good rule. This implies that the heuristic methods are not well-suited for hardware implementation. They are best suited for applications where the particular exponent value is used for many consecutive exponentiations, i.e. applications with a fixed exponent such as the RSA crypto system, or in applications where the exponent value has been known for long prior to the exponentiation. As a consequence, Sauerbrey and Dietel [SD92] suggest that an exponentiation rule for a particular exponent be value can be *precomputed* by heuristic methods and then be part of the input to dedicated hardware.

The strategy of most heuristic methods follows the strategy of the m-ary method and Thurber's modification: Split the binary encoded representation of the exponent e into windows, where the value of the ith window is denoted e_i. Then precompute a table that holds a power b^{e_i} for each window value e_i. Finally, perform the exponentiation, by means of the table, in approximately $\lfloor \log_2 e \rfloor - \lfloor \log_2 e_{n-1} \rfloor$ squarings and $n-1$ multiplications, where n is the number of windows and e_{n-1} is the value of the most significant window. The way in which the exponent is split into windows is called the *window distribution*. As seen in Section 2.1.1 and 2.1.2 the window distribution has influence on the computing time and on the number of required registers. By choosing a large window size the number of windows becomes small and the number of multiplications for the final phase of the computation becomes small. But simultaneously the number of possible window values increases and a large number of operations may be required to precompute the table. So the window distribution is closely related to the precomputation time, and a tradeoff between the number of operations required in the last phase of the computation and the number of operations required in the precomputation must be done.

Bos and Coster [BC89] were first to apply heuristics in exponentiation. Bos and Coster proposed a window distribution with much bigger window sizes than in Thurber's modification. The idea is to precompute only the powers of b that are *needed* and hereby avoid to precompute *all possible* powers. To be successful this idea demands an efficient precomputation technique. In addition to some guidelines to optimise the window distribution, Bos and Coster provide heuristics to construct an efficient precomputation of the powers $b^{e_0}, b^{e_1}, \ldots, b^{e_{n-1}}$. It is found that "in all cases we tried ... with

$e_j \leq 1000$" [BC89, p. 405] the number of operations can be estimated to be less than or equal to $\frac{3}{2}\log_2 e_j + n + 1$, where e_j is the largest of the n window values. Even though it is an *estimate* this result is remarkable: The absolute minimal number of operations required to compute b^{e_j} is $\log_2 e_j$ (see Section 2.2) and all methods described so far have a worst case computing time very close to $\frac{3}{2}\log_2 e_j$ for $e_j \leq 1000$. Hence, the overhead for computing $n-1$ additional powers of b is only slightly more than $n-1$ operations. Bos and Caster state that their method is capable of compute exponentials with 512 bit exponents in 605 operations on average. This is an improvement of 3 percents over the worst case time in Thurber's modification.

In [SD92] Sauerbrey and Dietel examine different window distribution methods with emphasis on the relationship between the number of operations and the number of required registers. The examination is based on experiments with 100 random selected 512 bit exponents. The heuristics by Bos and Coster are applied for the precomputation. None of the methods use less than 610 operations on average. It is not clear whether one of the examined window distribution methods is identical to the method of Bos and Coster. Sauerbrey and Dieted observe that window distributions leading to good results for the number of operations require a lot of registers, and vice versa. Motivated by the fact that memory is a scarce resource for a VLSI implementation, Sauerbrey and Dietel propose a new window distribution method. It shows a better compromise between operation count and register demand than previously known methods: For 512 bit exponents about 620 operations are used on average and 10-15 registers are required on average. Compared to the worst case of Thurber's modification in Table 2.3 this is an improvement of less than 0.5 percent (3 operations) in computing time and about 50-70 percents in register demand.

The results obtained by applying heuristics show a better *average* performance than Thurber's modification. But, if an exponentiation method is going to be hardware implemented the adequacy of the average case analysis can be questioned: The hardware implementation must be able to cope with even the worst case input. This means that the available memory must fit the worst case register demand. Furthermore, if the exponentiation is part of a real-time application, the worst case computing time is often of greater importance than the average computing time.

2.2 Theoretical Limits

In the search for fast exponentiation methods the question "how few operations are needed?" arises. Theoretical questions about exponentiation methods is usually formulated as *addition chain* problems. An addition chain for the integer e is defined as a sequence of integers

$$\begin{array}{l} a_0, a_1, \ldots, a_r \text{ where } a_0 = 1, a_r = e \text{ and for all } i = 1, 2, \ldots, r \\ a_i = a_j + a_k \text{ for some } k \leq j < i \end{array} \tag{2.18}$$

This means that every element, except a_0, in the sequence can be expressed as the sum of two preceding elements. The *length* of an addition chain with $r + 1$ elements is said to be r. Three examples of addition chains for 15 are 1, 2 ,3 ,5, 10, 15 with length 5 and 1, 2, 3, 4, 7, 8, 15 with length 6 and 1, 2, 3, 4, 7, 8, 10, 15 with length 7. An addition chain for the integer e can be viewed as a method to compute e by additions starting from 1. As illustrated by the examples there is a variety of methods for computing the same integer. Some methods compute intermediate results that are never used, so another method using fewer additions must exist. Even though a method uses all intermediate results, there might be other methods with fewer additions. The length of an addition chain expresses the number of additions used by the method. Hence, a chain for a given integer with the *shortest length* prescribes a method with fewest additions. Because of the rule $b^{e_1+e_2} = b^{e_1} * b^{e_2}$ an addition chain for the exponent e also prescribes an exponentiation method for computing b^e where the number of multiplications is equal to the addition chain length. In this section there will be no distinction between squaring and multiplication.

The shortest length, for which there exists an addition chain for e, is denoted $l(e)$. All the exponentiation methods described so far can be formulated as addition chains and the lengths of these chains give upper bounds for $l(e)$. The interesting question is how close these methods are to an optimal method. A trivial lower bound for $l(e)$ is $\log_2 e$ since the value of a chain element a_i can be no more than twice the value of element a_{i-1}. Another lower bound for $l(e)$ has been given by Schönhage [Sch75],

$$\log_2 e + \log_2 \nu(e) - 2.13 \leq l(e) \tag{2.19}$$

For $\log_2 \nu(e) - 2.13 > 0$ Schönhage's bound is the best lower bound known that applies *for all* e. An asymptotic better lower bound is given by Erdős in [Erd60] where it is stated that for *allmost all* e and an arbitrary chosen positive real ϵ,

$$\log_2 e + (1-\epsilon)\frac{\log_2 e}{\log_2 \log_2 e} \leq l(e) \tag{2.20}$$

The meaning of this formulation is that *for suitably large values of* e the number of addition chains, that are shorter than the left side expression, is substantially less than e. Thus, it is not possible to derive an exponentiation method that *for all* e performs the computation in a number of multiplications less than the left side expression. Erdős also showed that the bound given by (2.20) is tight, i.e. very close to known upper bounds of $l(e)$. In fact, an upper bound of $l(e)$ is given by the m-ary method: Recall the worst case computing time, $(\log_2 m)\lfloor \log_m e \rfloor + \lfloor \log_m e \rfloor + m - 2$. By insertion of $\log_2 m = \lceil c \log_2 \log_2 e \rceil$, where $c^{-1} = 1 + \frac{\epsilon}{2}$ and ϵ is an arbitrary chosen positive real, the following bound is derived,

$$\begin{aligned}
l(e) &\leq \log_2 e + \frac{\log_2 e}{\lceil c \log_2 \log_2 e \rceil} + 2^{\lceil c \log_2 \log_2 e \rceil} - 2 \\
&< \log_2 e + (1 + \frac{\epsilon}{)}2\frac{\log_2 e}{\log_2 \log_2 e} + 2(2^{\log_2 \log_2 e})^c \\
&\leq \log_2 e + (1+\epsilon)\frac{\log_2 e}{\log_2 \log_2 e} + f(e), \text{ where} \qquad (2.21) \\
f(e) &= 2(\log_2 e)^c - \frac{\epsilon}{2}\,\frac{\log_2 e}{\log_2 \log_2 e} \\
&= \frac{\log_2 e}{\log_2 \log_2 e}(2\,\frac{\log_2 \log_2 e}{(\log_2 e)^{1-c}} - \frac{\epsilon}{2}) \qquad (2.22)
\end{aligned}$$

Since $1 - c > 0$ the expression $(\log_2 e)^{1-c}$ grows faster than $\log_2 \log_2 e$ when e is increasing. For a fixed value of ϵ this implies that $f(e)$ eventually becomes negative when e is increasing. This result states that asymptotically no methods exists that performs better than the m-ary method. However, researchers should not be discouraged in the attempts to find methods that *for the given range of exponent values of interest* exponentiates faster than the known methods.

An indication of the hardness of finding an addition chain with the shortest length is given by Downey, Leong, and Sethi in [DLS81]. The authors proves that the *addition sequence problem* is NP-complete. The addition sequence problem is formulated as: "Given a sequence $e_1, e_2, \ldots, e_s$ of positive integers, what is the smallest number of additions to compute all s integers starting with 1?". This is a generalisation of the addition chain ($s = 1$) problem. In fact, in this chapter addition sequences have already been used for precomputing a table of powers. It is NP-hard to find the shortest addition sequence length and, of course, at least as hard to find an addition sequence with the shortest length. Even though the complexity of the addition *chain* problem still remains open, the results on addition sequences indicates that it might be very hard to find shortest addition chains. However, it cannot be *concluded* that the addition chain problem is hard. The article [DLS81] is often, incorrectly, referenced for proving that the addition chain problem is NP-complete.

The lower bounds by Schönhagen and Erdős give some answers to the question asked in the beginning in this section: For arbitrary exponent values the m-ary method is about the best obtainable. Other methods may perform the exponentiation in fewer operations, but the difference is vanishing for increasing exponent values. For exponent values of interest in this thesis Schönhage's bound (2.19) implies that at most 17 percents improvement of Thurber's method for 512 bit exponents is possible. Furthermore, the complexity of the addition sequence problem indicates that it might be very hard to improve the methods.

2.3 Parallel Computation

In Section 2.1 the aim was to reduce the required number of operations for computing exponentials and hereby to obtain a fast computation. In this section another approach for obtaining fast computation of exponentials is taken. Instead of reducing the *number* of operations, the aim is now to reduce the *time* for performing the computation. This can be accomplished by computing some of the operations in *parallel*. In a sequential computation the number of operations is a direct measure of the computing time. This is not always the case for parallel computation. It might very well happen that methods with comparatively many operations are better suited for fast parallel computation than methods with comparatively few operations.

Kung [Kun88] has made a thorough treatment of the problem of how to systematically map an algorithm onto an array of processing elements. Although the techniques described by Kung have not been systematically used in the work presented in this thesis, some of the descriptive techniques are applied for visualising and clarifying purposes. As noted by Rivest [Riv84] the *right-to-left binary method*, described by Equation (2.5), is suited for a parallel computation. To see how the operations of the right-to-left binary method can be computed in parallel the method is expressed as a set of recursive equations,

$$b^e = X_{n-1} \text{ where } \begin{array}{rcrlcl} X_{i+1} & = & X_i * (Y_{i+1})^{e_{i+1}}, & X_{-1} & = & 1, \\ Y_{i+1} & = & Y_i * Y_i, & Y_0 & = & b. \end{array} \tag{2.23}$$

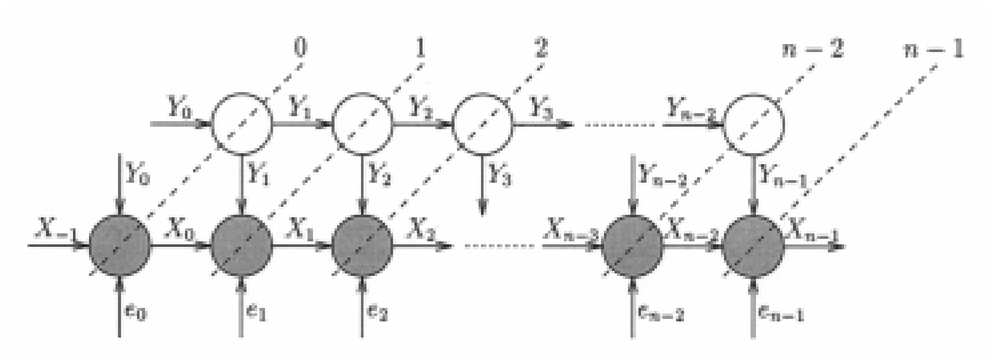
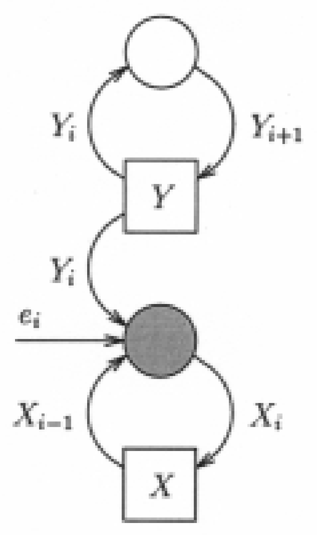


Figure 2.3: Dependence graph for right-to-left binary method.

A useful tool for revealing the possibilities for parallel computation of such a set of equations is a *dependence graph*. Figure 2.3 shows the dependence graph for the right-to-left binary method. The graph gives an illustration of the Equations (2.23) when they are folded out. Each node correspond to an operation: The unshaded nodes correspond to one of the $Y_i * Y_i$ operations and the shaded nodes correspond to one of the $X_i * (Y_{i+1})^{e_{i+1}}$ operations. An arc into a node represents a value on which the operation depends, and an arc out from a node represents a result of the operation. So, a dependence graph gives an explicit view of how an operation depends on the results from other operations and, hence, dictates the sequence of computation. The dependence graph can guide the designer when decisions about the parallel computation are taken: How many processing elements should be used, how should the operations be divided between the processing elements and when should each individual processing element execute the operations? The decision on how to configure processing elements and allocate operations onto the processing elements is described as a *mapping* from the dependence graph onto a configuration of processing elements. A straightforward mapping is the one-to-one mapping of the nodes in Figure 2.3 onto an array of $2n-1$ processing elements, one for each node. In this case, the execution sequence of each individual processing element, the *schedule of computation*, is indicated by the stippled lines in the figure: All nodes on the same line are processed at the same time. Such lines are called *equitemporal hyperplanes*. The lines have been numbered from 0 to $n-1$. The numbering gives an ordering in time for the execution of the operations. In the following the line numbered i will also be referred to as *time step number* i.



```
$X := 1$; $Y := b$;
for $i := 0$ to $n-1$ do
L: in parallel
        # if $e_i = 1$ then $X := X * Y$
        # $Y := Y * Y$
    end
end
```

The following invariant holds at label L:

$$Y = b^{2^i} \wedge X = \prod_{j=0}^{i-1} (b^{2^j})^{e_j}.$$

Figure 2.4: Parallel computation of right-to-left binary method.

Assume that the operation corresponding to a node takes one time unit. Then the number of nodes on the longest path from input to output corresponds to the longest sequence of operations and, therefore, is a measure of the minimal computing time or *latency* obtainable from the method. The actual latency depends on how the computation, expressed by the dependence graph, is mapped onto a number of processing elements and how the computation is scheduled on the processing elements. In the one-to-one mapping the latency is n time units, which is minimal for the right-to-left binary method. Note that the array can be *pipelined*, implying that a new computation can be initiated in each time unit. Each computation then ripples through the array and the stage of each computation is shown by one of the equitemporal hyperplanes. The result is a *throughput* of one exponential per time unit.

Unless there is a need for high throughput the utilisation of processing elements is very inefficient in the one-to-one mapping for a single exponentiation. Instead the dependence graph can be mapped onto two processing elements, where all the unshaded nodes in Figure 2.3 are mapped onto one processing element and all the shaded nodes are mapped onto the other processing element. The schedule of this computation remains as indicated by the stippled lines at the dependence graph. In Figure 2.4 a hardware architecture, consisting of two processing elements (multiplication units), two registers and data connections between processing elements and registers, is depicted together with an algorithmic description of the computation schedule. Furthermore, an invariant that holds prior to each cycle of the loop in the algorithm is given. The variables X and Y denote the contents of the registers. The hardware architecture and algorithm in Figure 2.4 is also presented in the articles reprinted in Appendix A and B. Note that the numbering of the stippled lines in the dependence graph is closely related to the value of i in the loop. Indeed, the ith cycle of the loop corresponds to a processing of all the nodes on the ith stippled line of the dependence graph and the invariant describes a relation between the inputs to these nodes. The invariant is also a *snapshot* of the register contents just prior to the processing of cycle i. The labels on the data connections in the hardware architecture have the following meaning: A label on an input to a processing element denotes the data value just prior to the processing of cycle i, and a label on an output from a processing element denotes the data value just after the processing of cycle i.

The mapping onto two processors also gives the minimal latency of n time units. If it is assumed that the time for a squaring equals the time for a general multiplication the sequential computation methods in Section 2.1 can be compared to the parallel computation on two processing elements. The parallel computing time is equal to the time for n multiplications and it is *independent* of $\nu(e)$, the number of non-zero bits in the binary encoding of e. The sequential computing time for the binary methods is $n + \nu(e) - 2$, and it depends on $\nu(e)$ which varies from 1 to n. So, compared to the worst case sequential computing time, a speed-up very close to 2 is achieved by using two processing elements. The number of required registers is two for both the sequential and the parallel computation of the right-to-left binary method.

A remark on the difference between the best case sequential computing time of $n-1$ time units and the parallel computing time of n time units is due here. In Equations (2.23) the value of X_0 is expressed as $1 * b^{e_0}$. This expression is transformed into a node in the dependence graph. Even though X_0 can be "calculated" by a simple selection between 1 and b a multiplication is performed in the parallel computation shown in Figure 2.4. In the analysis of the sequential computation in Section 2.1 a multiplication by 1 is *not* counted as a multiplication. Furthermore, the minimal sequential computing time, $n-1$ time units, arrises when all exponent bits $e_0, e_1, \ldots, e_{n-2}$ are zero and only the most significant bit e_{n-1} is one. (As written in the beginning of Section 2.1 it is assumed that $e_{n-1} > 0$). For this special exponent value the parallel computing time could also be reduced to $n-1$ time units since the last operation performed is a multiplication by 1. But in general the *latency* of the parallel computation is n time units. However, a *throughput* of one exponential per $n-1$ time units can be achieved if X_0 is not computed by a multiplication but simply initialised to b^{e_0}. Then, as seen at the dependence graph, the next computation can *overlap* the current computation by one time unit, i.e. the next computation of Y_1 can be performed by one processing element simultaneously with the current computation of X_{n-1} by the other processing element.

Compared to the worst case computing times for Thurber's method in Table 2.3 the improvement is 19 percents for 256 bit exponents, 18 percents for 512 bit exponents and 16 percents for 1024 bit exponents. According to the lower bounds in Section 2.2 the parallel computing time is smaller than the computing time for *any* sequential method when $\nu(e) \geq 8$. Obviously, the strength of the parallel computation lies in the reduction of the *worst* case sequential computing time. If the value of $\nu(e)$ is decreased the speed-up is also decreased. But, as already noted in Section 2.1.3 the worst case computing time is often of greater importance than the average computing time when the exponentiation is part of a real-time application.

2.3.1 Pipelined Computation

For the parallel computation on two processing elements the reduced computing time is obtained at the cost of an extra processing element. Now, as observed in Appendix A and B it is also possible to perform the parallel computation on a *single pipelined* processing element. Then the overhead in hardware is reduced to the extra registers that implement the pipeline buffers. In general, to implement a two-stage pipelined computation of an operation denoted by the function f it must be possible to decompose f into functions f_1 and f_2 such that $f = f_2 \circ f_1$, i.e. f is equal to the composition of f_1 and f_2. The efficiency of the pipelined computation is highly dependent on how the decomposition is done: First, the computing time for f_1 and f_2 should be balanced and both computing times should be about the half of the computing time for f. This ensures that the neither f_1 or f_2 is a bottleneck in the pipeline and that the throughput of the pipeline is about twice the throughput of a non-pipelined computation. Finally, the required number of registers for buffering the result from f_1 should be as small as possible.

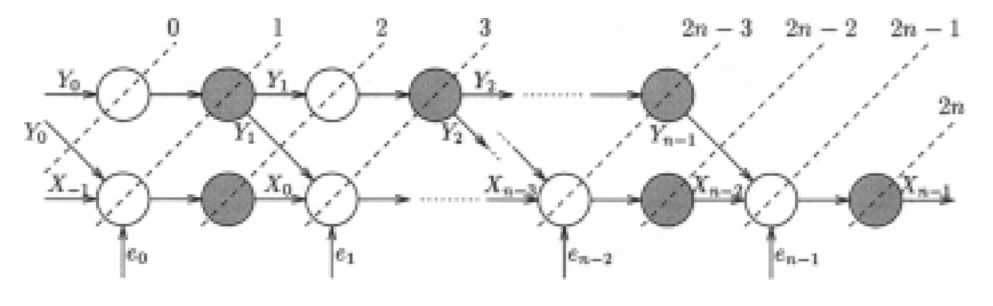


Figure 2.5: Dependence graph for pipelined right-to-left binary method.

In the following a pipelined computation of the binary right-to-left method will be developed. The main operation in Equations (2.23) is the multiplication operation. Assume that the multiplication operation is decomposed into the operations denoted by f_1 and jf_2, such that the composition of these gives,

$$f_2 \circ f_1(X, Y, e) = \quad X * (Y)^e \quad = \begin{cases} X * Y & \text{if } e = 1 \\ X & \text{if } e = 0. \end{cases} \tag{2.24}$$

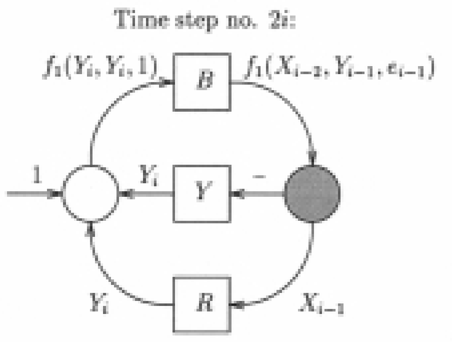
Then, if this decomposition substitutes the multiplication operation in Equations 2.23, the following set of recursive equations describes a pipelined right-to-left binary method,

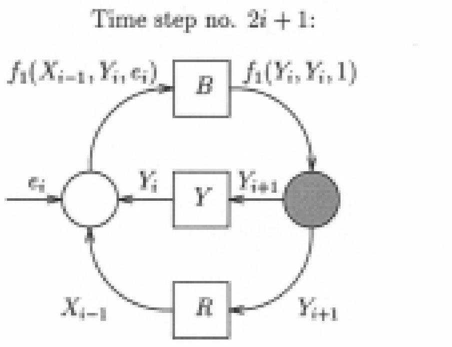
$$\begin{aligned} b^e = X_{n-1} \text{ where } X_{i+1} &= \quad f_2 \circ f_1(X_i, Y_{i+1}, e_{i+1}), & X_{-1} &= 1, \\ Y_{i+1} &= \quad f_2 \circ f_1(Y_i, Y_i, 1), & Y_0 &= b \end{aligned} \tag{2.25}$$

The dependence graph for these equations is shown in Figure 2.5. In this graph an unshaded node corresponds to one of the f_1 operations and a shaded node corresponds to one of the f_2 operations. The new dependence graph can be viewed as a modification of the old graph in Figure 2.3 where each old node is expanded into two new nodes. A multiplication now takes two operations. However, if it is assumed that the time for operation performing an f_1 or an f_2 operation is the half of the time for performing a multiplication then the computing time for a multiplication is unchanged. The idea behind the pipelined computation becomes clear when the dependence graph is mapped onto two processing elements, where one processing element performs f_1 operations and the other processing element performs f_2 operations. The expected total hardware consumption for these two processing elements is about the same as the hardware consumption for a single processing element that is capable of performing a complete multiplication operation. Hence, the new hardware architecture contains two processing elements that together form a *pipelined* version of one of the processing elements used in the previous hardware architecture in Figure 2.4.

A computation schedule is given by the equitemporal hyperplanes in Figure 2.5. The schedule has been chosen such that only one f_1 operation and only one f_2 operation is performed at the same time. The latency for the proposed pipelined computation is $2n+1$ time units. This corresponds to the time for performing $n + \frac{1}{2}$ multiplications. The small increase in latency is common for all pipelined computations: A small overhead due to the buffering in the pipeline will appear, and the overhead, measured in time units, will usually be equal to the number of inserted buffers. In this case the pipeline has two stages and a single buffer is inserted between the stages. Regarding the throughput, then it is obvious from the dependence graph that time step number 0 in the next computation can be overlapped with the current time step number $2n$. This results in a throughput of one exponential per $2n$ time units. Furthermore, a repetition of the discussion on page 42, of how the computation of X_0 can be replaced by a simple initialisation, will conclude that it is also possible to achieve a throughput of one exponential per $2(n-1)$ time units when the computation is pipelined.

In Figure 2.6 the pipelined hardware architecture is shown together with an algorithmic description of the computation schedule and two invariants. The hardware architecture is depicted in two states: The upper state corresponds to one of the even numbered time steps in Figure 2.5 and the lower





```
$B := f_1(1,1,0)$; $Y := b$; $R := b$;
for $i := 0$ to $n$ do
L$_0$: in parallel
        # $B := f_1(R, Y, 1)$
        # $R := f_2(B)$
    end;
L$_1$: in parallel
        # $B := f_1(R, Y, e_i)$
        # $Y := f_2(B)$
        # $R := f_2(B)$
    end
end
```

The following invariant holds at label L$_0$:

$$\begin{aligned} B &= \begin{cases} f_1(1,1,0) & \text{if } i = 0 \\ f_1(\prod_{j=0}^{i-2}(b^{2^j})^{e_j}, b^{2^{(i-1)}}, e_{i-1}) & \text{if } i > 0 \end{cases} \\ \bigwedge Y &= b^{2^i} \\ \bigwedge R &= b^{2^i}, \end{aligned}$$

and the following invariant holds at label L$_1$:

$$\begin{aligned} B &= f_1(b^{2^i}, b^{2^i}, 1) \\ \bigwedge Y &= b^{2^i} \\ \bigwedge R &= \prod_{j=0}^{i-1}(b^{2^j})^{e_j}. \end{aligned}$$

Figure 2.6: Pipelined computation of right-to-left binary method.

state corresponds to one of the odd numbered time steps. There are three registers in the hardware architecture. Register B is the pipeline buffer. It contains the result from an f_1 operation. The actual size of register B depends on how f_1 is chosen. Register R alternately holds the value Y_i and the value X_{i-1}. Finally, register Y is used for saving the value Y_i during a period of two time units. As seen in Figure 2.5, Y_i is needed in two consecutive time steps. The algorithm consists of an initialisation and a loop. Each cycle in the loop is divided into two phases: The first phase corresponds to an even numbered time step in Figure 2.5 and the second phase corresponds to an odd numbered time step. So, a cycle in the loop describes the computation of two consecutive time steps. According to Figure 2.5 register R must be initialised to b prior to the first phase of the first cycle. This is done before entering the loop. Furthermore, register Y and register R must be initialised to respectively b and 1 prior to the second phase of the first cycle. Since Y is not altered during the first phase, Y can also be initialised before the loop is entered. The initialisation of R prior to the second phase can be done by initialising register B to the value $f_1(1,1,0)$ prior to the first phase. Then the computation in the first phase will result in an assignment of the value $f_2(f_1(1,1,0)) = 1$ to register R. Of course the initialisation of register R could also have been carried out by a simple assignment in a conditional control structure, e.g. # **if** $i = 0$ **then** $R := 1$ **else** $R := f_2(B)$, during the first phase. The time step numbering in Figure 2.5 relates closely to i in Figure 2.6. The even numbered time step $2i$ corresponds to the first phase of the ith cycle and the odd numbered time step $2i+1$ corresponds to the second phase of the ith cycle.

The hardware requirement for the pipelined right-to-left binary computation corresponds to *one* multiplication unit, two registers and a pipeline buffer. Except for the buffer this is equal to the hardware requirement for the sequential right-to-left binary method. In Chapter 4 a VLSI implementation of a processor, that can compute modular exponentials, is described. The processor is using a pipelined right-to-left binary computation method for the exponentiation. The size of the pipeline buffer in this implementation is equal to the size of the other registers. So, at least when the multiplication composition is defined as modular multiplication, it is reasonable to count the buffer as a register. Hence, at the cost of a single additional register it is possible to halve the worst case computing time for the sequential binary method. The pipelined binary computation is superior to Thurber's modification—both with respect to computing time and with respect to hardware consumption. It can be concluded that *it is better to use additional registers for a pipelined computation than to use registers for a table of precomputed values.* However, as described by Brickell et al. in [BGMW92] it is possible to obtain very fast computation of exponentials by using a large table of precomputed values under the assumption that the computing time for this table can be disregarded. Indeed, in applications where the base is fixed over many consecutive exponentiations with varying exponents the table is computed only once.

Shand and Vuillemin [SV93] have analysed the pipelined binary right-to-left method. The analysis is based on the description of the method in Appendix B. Apparently the description is unclear, because Shand and Vuillemin assume that the computation is scheduled on a single processing element by *time-multiplexing*. In Figure 2.7 the dependence graph from Figure 2.3 is shown with a time-multiplexed computation schedule. It is seen that the computation can be scheduled on a single processing element that alternately executes an $X_i * (Y_{i+1})^{e_i+1}$ operation and an $Y_i * Y_i$ operation. This is what is meant by a time-multiplexed computation schedule. The computing time is $2n-1$ time units, where a time unit refers to the time for performing a complete multiplication, and the computing time is equal to the worst case time for the sequential binary method. Shand and Vuillemin observe that in the average case the sequential binary method is 33 percents faster than a time-multiplexed computation of the parallel binary method. Of course the time-multiplexed computation is slow. It corresponds to a sequential computation where *all* multiplications of the type $X_i * (Y_{i+1})^{e+1}$ is executed—even though e_{i+1} is zero. In fact, the *pipelined* right-to-left binary computation is 33 percents faster than the average case for the sequential binary computation.

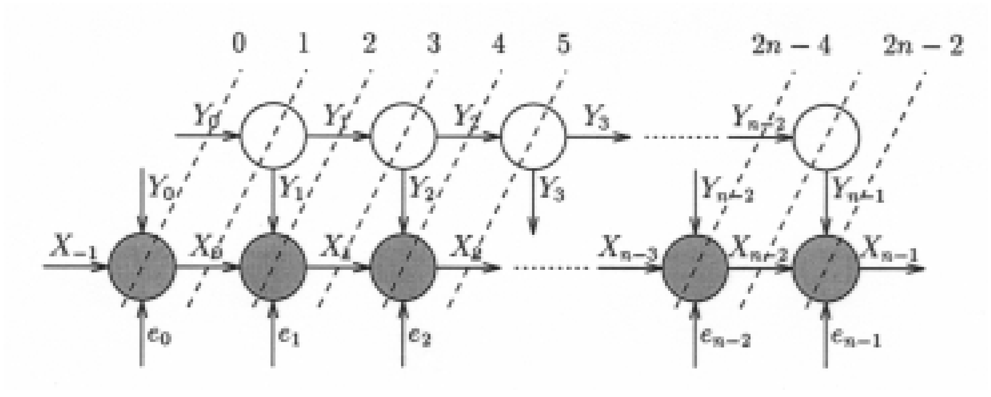


Figure 2.7: Dependence graph for right-to-left binary method. Time-multiplexed computation schedule.

2.3.2 From Computation Method to Implementation

In this section, dealing with parallel computation, several descriptive techniques have been used. It has been utilised that a *computation method* can be analysed by a *dependence graph*. The analysis revealed some options for the structure of the *hardware architecture* and for the *computation schedule*. The computation schedule has been formulated as *algorithms* and *invariants* have been given. In traditional programming disciplines the invariants are a useful means for proving the correctness of computer programs or algorithms. The invariants are also useful when dedicated hardware is constructed and used for a computation. Indeed, from an algorithmic point of view it does not matter whether standard micro-processors or dedicated hardware is used for the computation: An algorithm is a general description of the computation schedule, and the only requirement to the underlying hardware is, that the instructions in the algorithm are performed correctly. Hence, the invariants are useful for proving the correctness of the computation schedule. Moreover, as noted in the beginning of Section 2.3, an invariant can be viewed as being a *snapshot* of the register contents. This appeared to be very valuable in the process of implementing the exponentiation processor described in Chapter 4: The control logic was implemented by translating the algorithms to finite state machines, and invariants were used for keeping track of the states. The processing elements in the hardware architecture were translated to combinatorial logic and the registers were translated to flip-flops. During the validation of the combinatorial logic, invariants were used for expressing the correct register contents between consecutive clocking periods.

2.4 Modular Exponentiation

The exponentiation methods described in the previous sections only assume a single property about the multiplication composition: It must be associative. In this section the attention will be focussed on a specific multiplication composition: *Modular multiplication* as used in the RSA crypto system. It will be discussed how the algebraic properties of modular multiplication and of the exponents used in RSA cryptography can be utilised to improve the computing time for modular exponentiation.

In the RSA crypto system (see Section 1.1.3) the functions used for encryption and decryption have the form $b^e \bmod m$, where b is a block of data that is encrypted or decrypted and the pair (e, m) is a key. Modulus m is a composite of two prime numbers p and q, i.e. $m = pq$. To be useful as key, the exponent e must obey the restriction $\gcd(e, \varphi(m)) = 1$ where

$\varphi(m) = (p-1)(q-1)$. Furthermore, the primes p and q must be about the same bit-length [Nec92, p. 207]. The bit-length of modulus m and exponent e determines the security of the RSA crypto system. At present, bit-lengths in the range from 512 to 1024 are adequate. Since the number of multiplications needed for performing an exponentiation is about proportional to the exponent bit-length, it has been suggested to use small exponent values. It is possible to use a small *public* exponent value without compromising the security, but the length of the *private* exponent must be about the same length as modulus. Knuth [Knu81, pp. 386-389] suggests a public exponent of value 3. However, as observed by Hastad [Has88] small public exponent values may be vulnerable to crypto analytical attacks. So, some precautions should be taken when small public keys are used.

If small public exponents are used, the computing time for exponentiation with public exponents is significant shorter than the computing time for exponentiation with private exponents. Quisquater and Couvreur [QC82] show how the composite property of m, and the knowledge of the prime factors p and q, can be utilised to speed up exponentiation with private exponents. The trick is to apply the Chinese Remainder Theorem for dividing the computation of $b^e \bmod m$ into two faster computations, $b^{e \bmod (p-1)} \bmod p$ and $b^{e \bmod (q-1)} \bmod q$. The results of these computations can then be combined to the value of $b^e \bmod m$ by means of two multiplications. The following formulation of the Chinese Remainder Theorem is found in [Knu81, pp. 268-276], where Knuth discusses modular arithmetic,

Theorem 2.4-1 (Chinese Remainder Theorem) *Let $m_1, m_2, \ldots, m_r$ be positive integers that are relatively prime in pairs, i.e.,*

$$\gcd(m_j, m_k) = 1 \quad \text{when} \quad j \neq k. \tag{2.26}$$

Let $m = m_1, m_2, \ldots, m_r$, and let $a, u, u_1, u_2, \ldots, u_r$ be integers. Then there is exactly one integer u that satisfies the conditions

$$a \leq u < a + m, \quad \text{and} \quad u \equiv u_j (\text{mod } m_j) \text{ for } 1 \leq j \leq r. \tag{2.27}$$

A proof of Theorem 2.4-1 and some methods for computing a solution u to (2.27) are included in Knuth's book. To see how the theorem can be applied for computing a solution $u = b^e \bmod pq$ observe that if $u_p = b^e \bmod p$ and

$u_q = b^e \bmod q$ then,

$$\begin{aligned} u_p = b^e \bmod p = \quad & (b^e \bmod pq) \bmod p && \equiv u (\bmod p), \text{ and} \\ u_q = b^e \bmod q = \quad & (b^e \bmod pq) \bmod q && \equiv u (\bmod q). \end{aligned} \tag{2.28}$$

So, by computing u_p and u_q and combining these, a unique solution $u = b^e \bmod pq$ can be found. The advantage of splitting the computation of $b^e \bmod pq$ into the computations of $b^e \bmod p$ and $b^e \bmod q$ becomes clear when the following rewriting is done: According to Fermat's theorem (e.g. [Knu68, p. 69] $b^{p-1} \equiv 1 \pmod p$ whenever p is a prime and b is not a multiple of p. Hence, if it is assumed that $b \bmod p \neq 0$,

$$\begin{aligned} u_p &= b^e \bmod p \\ &\equiv b^{(p-1)(e \text{ div } (p-1))+e \bmod (p-1)} \pmod p \\ &\equiv (b^{(p-1)})^{e \text{ div } (p-1)} \cdot b^{e \bmod (p-1)} (\bmod p) \\ &\equiv 1^{e \text{ div } (p-1)} \cdot b^{e \bmod (p-1)} (\bmod p) \\ &\equiv b^{e \bmod (p-1)} (\bmod p) \end{aligned} \tag{2.29}$$

If $b \bmod p = 0$ Fermat's theorem cannot be used. However, for this special case the congruence be $b^e \equiv b^{e \bmod (p-1)} \equiv 0 \pmod p$ is valid when $e \bmod (p-1) \neq 0$. It should be noted that the congruence is *not* valid if $e > 0$, $e \bmod (p-1) = 0$ and $b \bmod p = 0$; by insertion into the definition of an exponential in (2.1) it follows that,

$$b^e \equiv b^{e-1} \cdot b \equiv 0^{e-1} \cdot 0 \equiv 0 \pmod p, \quad \text{and} \quad b^{e \bmod (p-1)} \equiv b^0 \equiv 1 (\bmod p).$$

A similar discussion leads to $u_q \equiv b^{e \bmod (q-1)} \pmod q$ when $e \bmod (q-1) \neq 0$. The restrictions on the exponent values do not imply difficulties for an application of the Chinese Remainder Theorem in the RSA crypto system. Indeed, since the exponent values are already restricted by the condition $\gcd(e, (p-1)(q-1)) = 1$ it is seen that $e \bmod (p-1) \neq 0$ and $e \bmod (q-1) \neq 0$.

Now, to obtain the solution $u = b^e \bmod pq$, the intermediate results u_p and u_q are combined by the computation,

$$u = (((u_p - u_q) \cdot q^{-1}) \bmod p) \cdot q + u_q \tag{2.30}$$

where q^{-1} is a precomputed integer satisfying $q \cdot q^{-1} \equiv 1 \pmod p$, i.e. q^{-1} is q's multiplicative inverse modulo p. According to Fermat's theorem, q^{-1}

can be computed as q^{p-2} mod p. Equation (2.30) was proposed by Garner in 1959 [Gar59]. Opposed to other ways of computing u from u_p and u_q, Equation (2.30) does *not* require multiplication modulo pq. This is a nice property when dedicated hardware is build for the computation.

The computing time for the exponentiation method based on the Chinese Remainder Theorem is smaller than the time for previous described methods. First, since the length of exponent e mod $(p-1)$ and e mod $(q-1)$ is approximate half the length of e, the number of modular multiplications for computing each of u_p and u_q is halved. If u_p and u_q are computed in parallel the time for computing both values will be reduced by a factor two due to the smaller exponent values. Second, the lengths of moduli p and q are half the length of m. According to Chapter 3 the computing time for modular multiplication is approximate proportional the operand length when dedicated hardware is used for the computation. This implies a reduction in computing time by a factor of two. If a standard micro-processor is used the computing time for multiplication is proportional to the square of the operand length [SV93], i.e. a reduction by a factor of four can be expected. The overhead for combining u_p and u_q is a multiplication module p, an ordinary integer multiplication, a subtraction and an addition. This overhead is negligible when p and q have lengths of more than hundred bits. The values e mod $(p-1)$, e mod $(q-1)$ and q^{-1} are also needed in the computation. However, these values can be thought of being a part of the private key, and the values can be precomputed once for all when the private key is generated. In total, the computing time can be reduced by a factor of four by using dedicated hardware to compute u_p and u_q in parallel. Such a dedicated hardware circuitry will contain two units for performing modular multiplication. But, since each unit operates on word sizes that are half the length of $m = pq$, the circuit size is expected to be equal to the size of a single unit capable of performing multiplication modulo m. If a single standard micro-processor is used the reduction factor is four, and if two processors are used for a parallel computation of u_p and u_q the reduction factor is eight.

It has been mentioned that dedicated hardware do not need a modulo m multiplication unit. This is true when *private* exponents are used. If a *public* exponent is used, the user is not expected to know the prime factorisation of m. Hence, in this case the Chinese Remainder Theorem cannot be applied, and a modulo m multiplication unit is needed. If a single dedicated hardware device shall be able to perform exponentiations with both private and public exponents, and be able to explore a parallel computation by the Chinese Remainder Theorem, it must be reconfigurable: Two "half-length" modular multiplication units should be configured into a single "full-length" modular multiplication unit. Of course, it would also suffice to include both kinds of modular multiplication units, but this would imply a large hardware consumption. Shand and Vuillemin describe a highly configurable implementation that is based on *field programmable gate arrays* in [SV93].

2.5 Summary and Discussion

The resource requirements and some characteristics for the exponentiation methods described in this chapter are summarised in Table 2.4. It should be remarked that the expressions for computing time and register requirement in the m-ary method and Thurber's modification assume m is a power of two. The unit of time is the computing time for a single multiplication or squaring, so the computing time expressions represent the total number of multiplication and squaring operations. The main interest in this thesis is to investigate efficient methods for performing modular exponentiation with very large operands. Therefore the table also shows the resource requirements for a modular exponentiation where the bit-length of the operands is 512 bit. (This operand length has become the standard length when different exponentiation methods or implementations for the RSA crypto system are being compared). Apart from the method based on heuristics all the requirements for 512 bit exponentiation are calculated for the worst case. The window sizes for the m-ary method and Thurber's modification is chosen such that a minimal worst case computing time is achieved. In Table 2.4 the number of processing elements corresponds to the number of multiplication units. In the pipelined method a single multiplication unit is split into two parts, where each part perform a computation in a half time unit. This is the reason for the awkward time expression.

In Section 2.1 the sequential computation methods has been treated. The binary method, it's generalisation to the m-ary method, Thurber's modification and methods based on heuristics have been described. By precomputing often used values and storing these into a table it is seen that the total computing time can be reduced. It is shown that it does not matter, with respect to resource requirements, whether the exponents are scanned left-to-right or right-to-left. Furthermore, it is described how a left-to-right window allocation can reduce the computing time. The fastest sequential method is Thurber's modification of the m-ary method. Compared to the binary method it reduces the worst case time for 512 bit exponents with 39 percents. Slightly shorter average computing times have been reported for methods based on heuristics. However, in real-time applications the worst case computing times are of greater importance, and when dedicated hardware is constructed the worst case memory requirement must also be considered.

Method	Resource requirements and characteristics		Example
Binary	Time:	$\lfloor \log_2 e \rfloor + \nu(e) - 1$	1022
	Registers:	2	2
m-ary	Time:	$\lfloor \log_2 e \rfloor - (\log_2 m - 1) + \nu(e) + m - 3$	639
	Registers:	m	32
	Window size:	$\log_2 m$	5
Thurber	Time:	$\lfloor \log_2 e \rfloor - (\log_2 m - 1) + \nu_m(e) + \frac{m}{2} - 1$	639
	Registers:	$\frac{m}{2}$	32
	Window size:	$\log_2 m$	5
Heuristics	Time:	Average based on experiments	(620)
	Registers:	Average based on experiments	(10–15)
Theoretical lower bound	Time:	$\log_2 e + \log_2 \nu(e) - 2.13$	519
Parallel	Time:	$\lfloor \log_2 e \rfloor + 1$	512
	Registers:	2	2
	Process. elem:	2	2
Pipeline	Time:	$\frac{1}{2}(2\lfloor \log_2 e \rfloor + 3)$	$512\frac{1}{2}$
	Registers:	3	3
	Process. elem.:	A single element split into two parts	1
Chinese Remainder Theorem	Modular exponentiation: Modulus and exponent as in RSA system. Gives 4 times speed up of a given method for exponentiation.		

Table 2.4: Summary of exponentiation methods, exemplified with worst case 512 bit modular exponentiation.

Section 2.2 gives answers to some theoretical questions about exponentiation. These questions are usually formulated in terms of addition chains. As a curiosity the following quotation of Brauer from 1939 [Bra39] shows that the research for fast methods to compute modular exponentials has taken place for quit a while:

> "The following question leads to addition chains: The least positive residue of $c^n \pmod{m}$ (c, m, n given integers) is to be formed using the smallest possible number of multiplications."

One of the theoretical results is a lower bound on the number of multiplications. This bound implies that no sequential method can compute exponentials with 512 bit exponents in less than 519 multiplications. Hence the computing time for the binary method cannot be reduced by more than 49 percents.

As described in Section 2.3 a change of approach can reduce the computing time. Instead of trying to reduce the number of multiplications a parallel computation can reduce the time for performing an adequate number of multiplications. It is shown how the right-to-left binary method can be computed in parallel. Hereby a computing time less than a theoretical lower bound for sequential computation is obtained. Furthermore, it is explained how a pipelined hardware architecture can be constructed. Compared to the binary method the pipelined method only requires a single additional register, and it reduces the worst case computing time by 50 percents. This is superior to any of the other methods.

In Section 2.4 a special class of exponentials is considered. This class is modular exponentials as defined by the RSA crypto system. The algebraic properties of the operands can be utilised to improve the computing time. It is discussed how the Chinese Remainder Theorem can be used to split a modular exponentiation with private exponent into two faster modular exponentiations. If dedicated hardware is used it is possible to reduce the computing time by approximately a factor of four. Table 2.4 do not show an explicit expression for computing time, register requirements etc. for this type of computation. This is because the trick of applying the Chinese Remainder Theorem in some sense is orthogonal to the other methods: The Chinese Remainder Theorem do not give an explicit method for computing exponentials, but the theorem tells how to speed up the computation of certain kinds of modular exponentials. A prerequisite is that a method for performing general modular exponentiation is given. Hence, any of the other methods in Table 2.4 can be selected as the general exponentiation method.

When studying papers on implementations of modular exponentiation it is remarkable that virtually all constructions of dedicated hardware do use the binary exponentiation method. The only exceptions are the first VLSI implementation from Sandia National Laboratories [Riv84], which uses the parallel right-to-left binary method, and the DEC Perle-1 implementation [SV93], which uses a 2^5-ary method. The VLSI implementation described in Chapter 4 uses, of course, the pipelined method.

The literature on exponentiation methods, both regarding the issues of practical implementations and the theoretical aspects, is extensive. One of the most thorough descriptions has been made by Knuth in [Knu81, pp. 441–466]. An approach, not mentioned in this chapter, for computing exponentials is described by Zhang, Martin and Yun in [ZMY88]. In this article it is utilised that in some algebraic systems a multiplicative inverse b^{-1} exists for all elements b, i.e. $b * b^{-1} = 1$, where $b \neq 0$.[1] This implies that negative digits are acceptable in an encoding of the exponent. The presented method results in a computing time about equal to the time for a 4-ary method while the number of required registers is decreased by one compared to the later method. The suggested encoding of the exponent is also known as a binary encoding with the redundant symmetric digit set $\{-1, 0, 1\}$. The same method is described in [JM89, Zha93]. In [Bri82] a similar method, with the same performance, is described. It is also possible to describe Zhang et al.'s method in terms of a generalisation of addition chains: addition/subtraction

[1]Zhang et al. apply this method for computing modular exponentials in the RSA crypto system. To ensure the existence of the multiplicative inverse $b^{-1} \bmod m$ the condition $gcd(b, m) = 1$ must be fulfilled. Since $m = pq$ is a composite of two primes this condition does not always hold. It does not hold when b is a multiple of p or q. However, in practice this is not of great concern because the probability, that b is a multiple of p or q, is vanishing when p and q are large and b is random in the set $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. Typical, the bit-lengths of p and q are more than 256 bits. The set $\{b \in \mathbb{Z}_m : b > 0 \text{ and } gcd(b, m) = 1\}$ is denoted $\mathbb{Z}_m^*$ and consists of $\varphi(m) = (p-1)(q-1)$ elements [Nec92, p. 258]. Hence, the probability that $b \in \mathbb{Z}_m$ and $b \notin \mathbb{Z}_m^*$ is

$$1 - \frac{\varphi(m)}{m} = 1 - \frac{(p-1)(q-1)}{pq} = \frac{p+q-1}{pq}.$$

If p and q is assumed to be 2^{256} the probability is about equal to 2^{-255} which indeed is vanishing.

chains. The theoretical lower bound by Schönhage, Equation (2.19), is extended to addition/subtraction chains in Schönhage's original article [Sch75]. An exponentiation method, inspired from a data compression algorithm, is presented by Yacobi in [Yac90]. It uses the precomputation technique known from the m-ary method, but the window size is varied during a scan of the exponent. The computing time for this method depends on the "compressibility" of the exponents. Compressible exponents results in faster computing times, and vice versa. Finally, methods especially suited for computations where the based is fixed for many consecutive exponentiation while the exponent is varying are described by Brickell et al. in [BGMW92]. These methods result in very fast computations when a large table has been precomputed. Since the base is fixed the precomputation is only performed once. Further development of these methods is described by de Rooij in [dR94] and by Lim and Lee in [LL94]. A warning to the time unit of this chapter is given by McCarthy in [McC86]. In this chapter a time unit corresponds to the time for performing a multiplication. When the multiplication composition is known to be modular multiplication the intermediate results will remain limited to the range given by modulus. Hence, it is fair to assume that the multiplication time is independent on the operands. But for other multiplication compositions the time for a multiplication may be highly dependent on the operand bit-length and, consequently, other exponentiation methods may be the most efficient.

Chapter 3

Modular Multiplication

The previous chapter discussed the evaluation of exponentials. The definition of an exponential refers to an abstract multiplication composition. In this chapter a specific multiplication composition, *modular multiplication*, is studied. In general, *multiplication* refers to the process of evaluating products $a * b$ where $*$ is a multiplication composition. As for the definition of exponentials a general definition of products refers to another abstract composition, now called *addition*. Since the main interest of this thesis is fast evaluation of modular exponentials the descriptions in this chapter will be restricted to modular multiplication. Certainly, many of the methods and ideas of this chapter can also be used for other multiplication compositions that possess algebraic properties similar to modular multiplication.

A recursive definition of the modular product $a *_{\mathbb{Z}_m} b = (a \cdot b) \bmod m$, where a is the multiplier, b the multiplicand and m the modulus, can be formulated as

$$\begin{aligned} 0 *_{\mathbb{Z}_m} b &= 0 \\ a *_{\mathbb{Z}_m} b &= ((a-1) *_{\mathbb{Z}_m} b) +_{\mathbb{Z}_m} b. \end{aligned} \tag{3.1}$$

It is assumed that m is an integer, $m > 0$, and that both a and b belongs to the set of non-negative integers less than m, $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$. The addition composition, $+_{\mathbb{Z}_m} : \mathbb{Z}_m \times \mathbb{Z}_m \mapsto \mathbb{Z}_m$, is *modular addition* and it is defined by $x +_{\mathbb{Z}_m} y = (x + y) \bmod m$.

The reason for formulating modular products by the above equations becomes clear when a comparison to the definition of exponentials, Equation (2.1), is done. It is seen that the definition of modular products is a specialisation of the general definition of exponentials: The multiplier a is an exponent, the multiplicand b is a base and the addition composition, with the neutral element 0, is a multiplication composition in Equation (2.1). Therefore, the exponentiation methods in Chapter 2 also can be formulated as modular multiplication methods, and the results obtained on exponentiation methods can be used when discussing modular multiplication. All efficient exponentiation methods require that the multiplication composition is associative and, indeed, modular addition is associative.

When studying the literature on modular multiplication it appears that most methods can be identified as analogous forms of one of the exponentiation methods in Chapter 2. (A class of modular multiplication methods, that cannot be properly described by Equation (3.1), is called Montgomery multiplication. This class will be treated in Chapter 5). Hence, a complete description of the methods used in an implementation of modular exponentiation can be made by identifying the exponentiation method and the multiplication method as one of the methods from Chapter 2, and by characterising the modular addition method.

The exponentiation methods in Chapter 2 were formulated in general terms. The methods were based on very few assumptions about the algebraic properties of the multiplication composition and of the application of the exponentials. In this chapter another approach will be taken: The algebraic properties of modular addition will be explored and utilised as much as possible to achieve fast modular multiplication methods. Furthermore, the knowledge of the application area of the modular products will be utilised to formulate techniques that may only be advantageous in the specific applications considered in this thesis. The applications are characterised by very large operands and by many consecutive computations of modular products using a fixed modulus. Moreover, the majority of the modular products are intermediate results in the application.

In this chapter the emphasis will be on the so-called high-radix modular multiplication methods. These methods are the analogous forms of the β-ary exponentiation methods. (The β-ary encoding was denoted the m-ary encoding in Section 2.1.1. Because the symbol m is denoting a modulus in this chapter it may be a source for confusion, so the new symbol β is used). Radix 2 and radix 4, i.e. 2-ary and 4-ary, modular multiplication methods dominate the literature. For radices greater than 4 the computation becomes more complicated. However, if these complications are solved efficiently, there is a potential for faster evaluation of modular products and, hence, modular exponentials by using high-radix methods. Opposed to the description of the exponentiation methods in Chapter 2 the resource requirements of the modular multiplication methods in this chapter will not be precisely stated. The large number of design tradeoffs and possibilities for combining the various techniques makes this an infeasible task. Instead the chapter will be of a more descriptive nature. To illustrate the implications of the presented methods and techniques some of the discussions comprise an analysis and comparison of examples.

This chapter is divided into eleven sections. Section 3.1 introduces a simple method for performing modular additions. The purpose of the description is to give the reader an introduction of the basic types of operations required in the computation of modular multiplication methods. In Section 3.2 the representation of integers is discussed. It turns out that the representation of the operands is very important for the efficiency of addition and subtraction operations, and for the efficiency of the formation of multiples. In particular, so-called redundant number representations are useful. Section 3.3 discusses how a residue modulo m can be used for representing intermediate results in the computation of modular operations. As for the redundant number representation the residue representation also introduces a kind of redundancy in the representation. The redundancy in the residue representation is utilised to achieve an efficient quotient determination. It is also shown how the rules of modular arithmetic make it possible to replace the basic modular addition operation by other kinds of basic operations. In Section 3.4 the left-to-right modular multiplication method is treated. This method is similar to the left-to-right exponentiation method. The basic operation is of the form $(2^k s + a_i b) \bmod m$. Instead of subdividing this basic operation into a number of modular additions another approach for the computation is followed: The basic operation is expressed as the intermediate operation $2^k s + a_i b - q_i m$ which is subdivided into computation of multiples $a_i b$ and $q_i m$, and into determination of a quotient digit q_i. Section 3.5 shows how a parallel computation of the left-to-right modular multiplication method leads to a computing time that is about equal to the computing time for SRT division (e.g. [Kor93a]). Moreover, a hardware architecture for this parallel computation is presented. Section 3.6 explains a general scheme for the computation of modular operations. The scheme keeps track of the representations of the intermediate operands by means of restrictions on the input and output of modular operations. In Section 3.7 the computation of multiples is discussed. The representations of the operands and the resulting multiple have influence on the computing time and on the required circuitry for a dedicated hardware implementation. The next three sections are devoted to the quotient determination operation: Section 3.8 provides an analysis of the interdependencies of the parameters that characterise the quotient determination. The implications of the parameter values are discussed. Section 3.9 shows how a simple scaling of the modulus can be used for improving the parameter values and, hence, for reducing the complexity of the quotient determination. Section 3.10 comprises a description of methods for determination of quotient digits. In particular, methods based on table-lookup are treated in detail. Furthermore, the quotient determination complexity in modular multiplication and in SRT division is compared. Finally, the results of this chapter is summarised and discussed in Section 3.11.

3.1 A Simple Modular Addition Method

To illustrate some of the techniques used for performing modular multiplication and to introduce some terminology the following straightforward modular addition method is discussed,

$$x +_{\mathbb{Z}_m} y = x + y - qm, \quad \text{where} \quad q = \begin{cases} 0 & \text{if } x + y - m < 0 \\ 1 & \text{otherwise.} \end{cases} \tag{3.2}$$

Here, modular addition is implemented by ordinary integer addition followed by a *modular reduction*. The reduction is implemented by ordinary integer comparison and subtraction. In general, a modular operation can be implemented by applying the related integer operation followed by a modular reduction. A reduction modulo m of the integer z refers to the process of computing $z \bmod m$. This means to find the non-negative integer r such that $z = qm + r$, where $0 \leq r < m$ and q is some integer. Modular reduction is closely related to integer division, where the aim is find the integer q, denoted z div m, such that $z = qm + r$, where $0 \leq r < m$ for some integer r. Since q and r are unique for a given pair (z, m) it makes sense to name q and r respectively *the quotient* and *the remainder*. In (3.2) a comparison determines the quotient value. Since $x \in \mathbb{Z}_m$ and $y \in \mathbb{Z}_m$ the sum $x + y$ is restricted to the range $\{0, 1, \ldots, 2(m-1)\}$ and, therefore, the quotient is restricted to the range $\{0, 1\}$. It is seen that the quotient is 0 when $x + y - m < 0$

and 1 when $x + y - m \geq 0$. This calculation of a quotient will be called *quotient determination*. It is also known as *quotient selection*. To obtain an efficient quotient determination the knowledge of the operand ranges is utilised. In this case a single comparison is sufficient. After the quotient q has been determined the modular reduction is completed by subtraction of the multiple qm. Since all operations used in the evaluation of $(x + y) \bmod m$ are integer operations (addition, subtraction and comparison) it is natural to apply well known techniques (e.g. [Kor93a]) for efficient computation of integer expressions: Assume x, y , and m are binary encoded in n bits and also the result $(x + y) \bmod m$ is delivered in a binary encoding. Further assume that negative integers are represented by two's complement. Then all integer operations needed can be performed by an $n+1$ bit wide addition unit. In total, two integer additions are used in Equation (3.2): One addition for computing $x + y$ and one addition for computing $(x + y) + (-m)$. The comparison is done by inspection of the resulting sign of the latter addition, so the subtraction and comparison are accomplished by the same operation.

3.2 Integer Representation and Arithmetic

As seen from the simple method in Equation (3.2) modular addition can be computed by means of integer addition, subtraction and comparison. Hence, by applying the techniques known from integer arithmetic an efficient computation of modular sums may be obtained. One of the most important issues in computer arithmetic is the *representation* or *encoding* of numbers. Numbers can be represented in several ways: Outside the community of computer scientists and electrical engineers the decimal representation is the most common representation. When computations are executed on electronic computers the binary representation is better suited. The number representation has great influence on the efficiency of a computation. Some representations are better suited for one type of computation while other representations may be preferable for another. As a simple illustration of this, consider multiplication by a constant: If a number is binary encoded multiplication by 2 is obtained by a left-shift of the binary digits. A decimal encoded number is multiplied by 10 when left-shifting the decimal digits. It is much more inconvenient to multiply a binary encoded number by 10 or to multiply a decimal encoded number by 2. In Section 2.1.1 the β-ary encoding was introduced. Both the binary encoding ($\beta = 2$) and the decimal encoding

($\beta = 10$) are specialisations of the β-ary encoding. An example of an unconventional number representation is the Residue Number System, see e.g. [Gar59, ST67, Tay84]. Indeed, this representation was implicitly introduced in Section 2.4 where the Chinese Remainder Theorem was utilised to compute modular exponentials. According to this theorem an integer u in the range $\{0, 1, \ldots, pq - 1\}$ can be uniquely represented by a pair of residues (u mod p, u mod q), where p and q are primes. In Section 2.4 it was shown how an exponential u^e mod pq can be efficiently computed in this representation.

Until now, no specific assumptions about the number representation have been made. One of the advantages when constructing dedicated hardware to support a particular computation is the possibility to choose between different ways to represent the operands. This freedom is not present when a standard micro-processor is used for executing the computation. In standard micro-processors all operands are binary encoded. In the remaining of this chapter it will be utilised that some arithmetic operations can be performed very efficiently when certain integer representations, different from the binary encoding, are chosen. There is, however, one limitation on the representation; it is assumed that the input to a modular exponentiation and the resulting output are binary encoded. This implies that a conversion between binary encoding and the representation chosen for the intermediate operands is necessary. So, before another number representation is applied in the computation the cost of the conversion must be considered.

3.2.1 Non-Redundant Representation

The β-ary encoding of an integer is an example of a *non-redundant fixed-radix representation* [Kor93a, Chapter 1]. Let the integer x be β-ary encoded as a string of n digits, $x_{n-1}x_{n-2}\ldots x_0$. Then each digit belongs to the *digit set* $\{0, 1, \ldots, , \beta - 1\}$ and the value of x is expressed by

$$x \;=\; \sum_{i=0}^{n-1} x_i \beta^i \tag{3.3}$$

In this sum the weight of digit x_i is the ith power of a *fixed* integer β, which is called the *radix*. The smallest possible value of x is 0 and the largest possible value is $\beta^n - 1$. Therefore the range $[0; \beta^n - 1]$ is called the *range of the representable integers*. Obviously, all possible values of x are integer values. In total there are β^n different strings with n digits. It is easy to see that an n-digit string is a unique representation of an integer x: Each digit $x_i, i = 0, 1, \dots, n-1$, is uniquely determined by the expression $x_i = (x$ div $\beta^i)$ mod β. Therefore, all β^n integers in the range $[0; \beta^n - 1]$can be represented by an n-digit string. Since no two different strings represent the same value, the representation is said to be *non-redundant*.

The most natural representation of integers in computer systems is the binary encoding, which is identified as a non-redundant radix 2 representation with the digit set $\{0, 1\}$. However, a string of bits can also be interpreted as a radix 2^k representation with the digit set $\{0, 1, \dots, 2^k - 1\}$. Then each group of k bits, starting from the least significant bit, is a radix 2^k digit. The value of such a digit is binary encoded. It is common to denote radix 2^k representations *high-radix representations* whenever k is greater than 1. Since it is just a matter of *interpretation* of the meaning of a string of bits there is no additional cost when a representation is changed between a binary encoding and a 2^k-ary encoding. In the literature on computer arithmetic the term *high-radix multiplication* usually refers to a method where the number of multiplier-bits scanned in each iteration is more than one bit, say k bits. This corresponds to the 2^k-ary exponentiation method from Section 2.1.1. Similarly, the term *high-radix modular multiplication* corresponds to the 2^k-ary exponentiation method.

The literature on computer arithmetic contains many different methods for efficient addition of binary numbers. Since these methods are described in standard textbooks on computer arithmetic (e.g. [Hwa79, Obe79, Spa81, Sco85, Kor93a]) and on VLSI design (e.g. [GD85, WE92]) the methods will not be treated in detail in this thesis. A simple addition method is known as *carry ripple addition*. The worst case computing time for this method is proportional to the operand bit-lengths. In fact, if the bit-lengths are n bits the computing time is equal to the delay of n *full adders*. The hardware architecture of a carry ripple adder is simple and very regular; it is a row of n full adders, where the communication is limited to the nearest neighbours. The fastest addition methods, e.g. *carry lookahead addition*, perform the addition in a time proportional to $\log_2 n$. However, the architectures for these methods are more complex and less regular than a carry ripple adder. Assuming that negative integers are represented properly, e.g. by two's complement, subtraction is accomplished by addition of a negative integer. Furthermore, comparison is done by subtraction followed by inspection of the resulting sign. Hence, both addition, subtraction and comparison of non-redundant binary numbers can be done in a time proportional to $\log_2 n$. No faster methods for these operations are known. Indeed, according to Koren [Kor93a, pp. 80–81] theoretical results indicate that the lower bound for binary addition is logarithmic in the number of bits.

Integer Multiplication and Modular Multiplication

In the beginning of this chapter it was explained how (modular) multiplication can be viewed similarly as exponentiation. It was argued that if the addition composition is associative the results obtained from exponentiation can be reused in multiplication. However, integer multiplication is inherently more efficient than integer exponentiation when the integers are represented by a fixed-radix representation. E.g. assume the integers are binary encoded and the bit-length is n. Further, assume the binary left-to-right exponentiation method in Section 2.1 is applied for both exponentiation and multiplication. The worst case computing time for an exponentiation is $n-1$ squarings plus $n-1$ multiplications. Integer squaring cannot be done essentially faster than integer multiplication[1], so the total time is about $2(n-1)$ multiplications. This indicates that the time for integer multiplication is about $2(n-1)$ additions. But, observe that a squaring operation $b * b$ in the exponentiation process corresponds to a doubling operation $b + b$ in the multiplication process. Since doubling can be done by a simple left-shift it is seen that the time for this operation is negligible in comparison to the time for a general addition. Hence, the worst case time for integer multiplication is about $(n-1)$ additions when the binary method is used. Now, the computing time for doubling is negligible in *integer* multiplication but how about doubling in *modular* multiplication ? According to the simple modular addition method in Equation (3.2) a modular doubling can be achieved by a left-shift and a subtraction. The subtraction is needed to perform a modu-

[1]If some very fast integer squaring method appeared then an integer multiplication could be implemented by the *quarter-square multipliccation method*. According to Chen [Che71] this method was used in analog computation. The method is based on the expression,

$$a \cdot b = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2$$

Hence, integer multiplication is not significantly slower than squaring.

lar reduction of the left-shifted intermediate result. Even though a modular doubling is faster than a general modular addition it is not negligible. A modular doubling requires one integer addition while a general modular addition requires two integer additions. Hence, the worst case computing time for modular multiplication performed by the binary method corresponds to $n-1$ plus $2(n-1)$ integer additions; a total of $3(n-1)$ integer additions. This is three times more than the computing time for integer multiplication. If the parallel computation of the binary right-to-left method in Section 2.3 is applied, the computing time for the integer multiplication corresponds to n integer additions while the computing time for the modular multiplication corresponds to $2n$ integer additions.

3.2.2 Redundant Representation

In the previous section the non-redundant radix β representation of an integer x was defined to be a string of digits $x_{n-1}x_{n-2}\ldots x_0$ where a digit is restricted to the digit set $\{0, 1, \ldots, \beta-1\}$. A *redundant* radix β representation is characterised by having more than β values in the digit set. Avižienis [Avi61] have studied a class of redundant representations where the digit set is a symmetric set of negative and positive digit values $\{-\sigma, -\sigma+1, \ldots, 0, \ldots, \sigma-1, \sigma\}$ where σ is a positive integer not greater than $\beta-1$. If $2\sigma+1 > \beta$ this set has more than β digit values and, hence, is redundant. Avižienis denotes this class of redundant representations for *signed digit number representation*. The value of a redundant represented integer is given by the same expression as non-redundant representations in Equation (3.3). As an example, consider a string of n digits in radix 4 representation with digit set $\{\overline{2}, \overline{1}, 0, 1, 2\}$ where $\overline{d}$ denotes the negative digit value $(-d)$: The range of representable integers is $[-\frac{2}{3}(4^n-1); \frac{2}{3}(4^n-1)]$. Each of the $\frac{4}{3}(4^n-1)$ integers in this range can be represented by some n digit string. There are 5^n different strings with n digits, so when $n > 1$ some integers must be represented by more than one string. E.g. the strings $1\overline{2}$ and 02 both represents the value 2. Avižienis restricts the largest absolute value σ of a digit by $\sigma \leq \beta - 1$. Hereby it is ensured that the representation for a zero valued integer is unique.

The advantage of redundant representations is that the computing time for addition turns out to be *independent on the operand digit length*. I.e. there is no carry ripple effect when two redundant represented integers are added. The sum is represented by the same redundant representation. To be

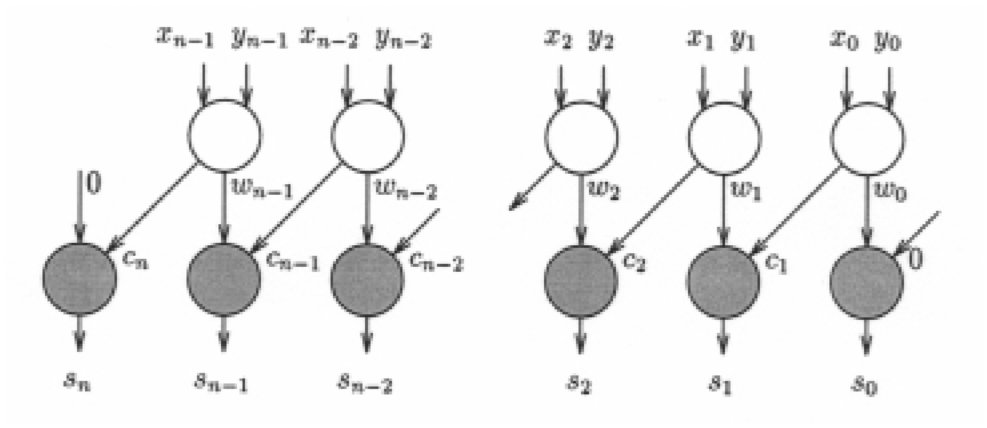


Figure 3.1: Addition of two redundant represented integers.

more specific, Avižienis shows that if the allowable number of digit values is greater than or equal to $\beta+2$ then the ith sum digit can be determined from the ith and $(i-1)$th operand digits. This means that a carry (or borrow) cannot ripple more than one digit position. This is illustrated in Figure 3.1. The feature in *redundant addition*, that makes the computing time independent on the operand digit length, is the possibility for *absorbing an incoming carry* without generating a new carry. Assume x and y are encoded in a redundant radix β representation with symmetric digit set $\{\overline{\sigma}, \overline{\sigma-1}, \ldots, \sigma\}$, where $\sigma \leq \beta-1$. Then w_i and c_{i+1} are computed by the unshaded processing elements in Figure 3.1 such that

$$x_i + y_i = w_i + \beta_{c_i+1} \quad \text{where} \quad w_i \in \{\overline{\sigma-1}, \overline{\sigma-2}, \ldots, \sigma-1\} \quad \text{and}$$
$$c_{i+1} \in \{\overline{1}, o, 1\}.$$

Now the final sum digit $s_i = w_i + c_i$ can be computed by the shaded processing element. The constraints on the values of w_i and c_i ensure that s_i belongs to the digit set $\{\overline{\sigma}, \overline{\sigma-1}, \ldots, \sigma\}$. Hence, a carry generated at position i will be absorbed at position $i+1$ without generating a new carry. Figure 3.1 also illustrates that the hardware architecture of redundant adders has a very regular structure where the communication is limited to the nearest neighbours. In [Avi61] Aviženis also shows that the (least) redundant representation with $\beta+1$ allowable digit values has similar properties. However, then the ith sum digit must be determined from the ith, $(i-1)$th and $(i-2)$th operand digits.

Another often used redundant representation is called *carry save representation.* A carry save representation is a radix 2 representation with the asymmetric digit set $\{0, 1, 2\}$. The sum of two binary numbers s and c can be interpreted as a single number in carry save representation by assigning the sum of the ith bits $s_i + c_i$ to the ith redundant digit. Indeed, the result of a *carry save addition* is two binary numbers s and c. A carry save adder resembles of carry ripple adder. However the carry output from the full adder at position i is not connected to the carry input of the full adder at position $(i + 1)$. Instead the carry outputs are part of the result, just as the sum outputs are part of the result, and the carry inputs are being an extra operand. So, a carry save adder takes three binary numbers as input and outputs two binary numbers whose sum is equal to the sum of the three input numbers. The output is in carry save representation where the ith digit is encoded as the pair (s_i, c_i). This is illustrated in Figure 3.2 where a node symbolises a full adder. A carry save adder is also called a "3-2 adder" or a "3-2 compressor". These names mirrors the fact that a carry save adder reduces the representation of a sum from three numbers to two numbers. A carry save adder *cannot* be used for addition of two carry save represented numbers. It may add three binary numbers or it may add a binary number to a carry save number. For addition of two carry save numbers a 4-2 adder is required. Obviously, a 4-2 adder can be constructed from two 3-2 adders.

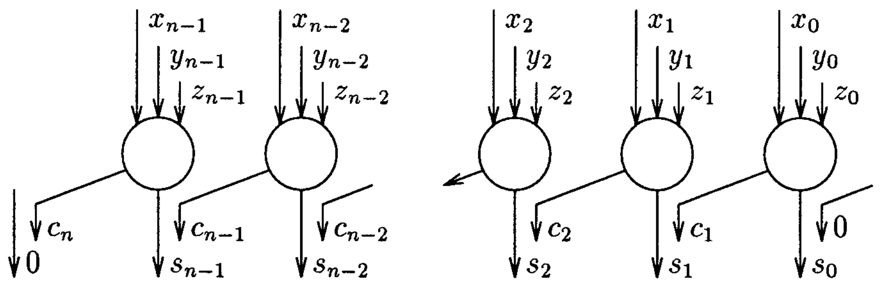


Figure 3.2: A carry save adder.

Redundant addition is much faster than non-redundant addition when the operands are relatively long. Furthermore, the hardware architectures of redundant adders are more regular than the architectures of fast non-redundant adders. Thus, if several additions or subtraction have to be performed on intermediate operands, that may be kept in redundant representation, redundant adders are preferable with respect to computing time and regularity. In the literature on modular multiplication there are several proposals for utilisation of redundant represented intermediate operands. Most of these proposals can be grouped into one of three different radix 2 representations: The *carry save representation* (e.g. [Miy82, Bak87, GD88, HDVG88, ICHO89, Mor89, KH90a, IWSD92, OPT93]), the *borrow save representation* (e.g. [VVDJ90, TY92, Tak92]), which is identical to the radix 2 signed digit representation with the digit set $\{\bar{1}, 0, 1\}$, and the *delayed carry save representation* (e.g. [Bri82, Gib88, FDG90, WE90]). The delayed carry save representation is a slightly modified version of the carry save representation. There are of course many possible choices of a redundant digit set for a given radix. Parhami [Par93] has analysed the properties of a generalisation of the redundant signed digit representation.

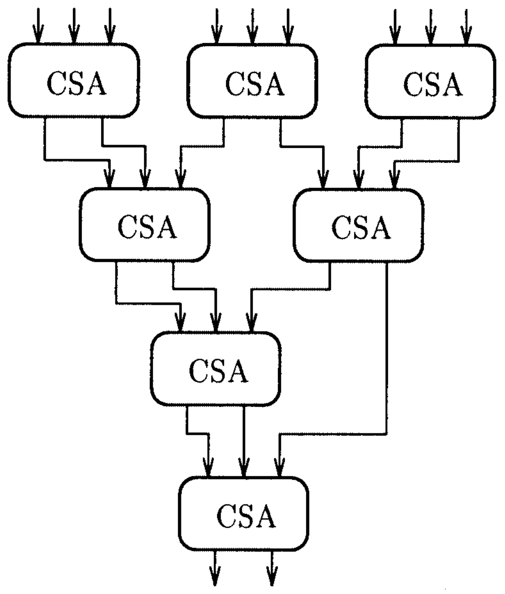


Figure 3.3: A Wallace Tree for computing a sum of nine binary numbers.

Utilisation of Redundant Addition in Multiplication

A classic application of redundant addition is integer multiplication. This operation is computed by a number a consecutive additions and only the final result has to be converted into non-redundant representation. A multiplication unit that uses the carry save representation is suggested by Wallace in

[Wal64]. The time for performing an integer multiplication by Wallace's multiplication unit is proportional to $\log_2 n$ where n is the operand bit-length. This computing time is achieved by an architecture that is structured as a tree of carry save adders. Such a tree of carry save adders is often denoted a *Wallace Tree*. The Wallace Tree in Figure 3.3 depicts a tree where each node is a carry save adder (CSA). The tree is capable of computing a sum of nine binary encoded numbers in a time equal to the time for four carry save additions. The sum is in redundant carry save representation which can be converted to non-redundant representation by a non-redundant addition. Note that the tree in Figure 3.3 implements a 9-2 adder. Since a multiplication operation is computed by means of a number of additions a Wallace Tree is useful for obtaining a fast multiplication. Indeed, Wallace's multiplication unit is remarkable fast. The time for computing the sum of n binary numbers is comparable to the time for an addition of two n bit numbers by one of the fastest non-redundant adders. In [TYY85] Takagi et al. describe a multiplication method that uses a tree of redundant signed digit adders, and a VLSI implementation of this method is described in [HNN$^+$87].

Multiplier Recoding

Another important application of redundant representations is *multiplier recoding*. Consider the multiplication $a \cdot b$ where the multiplier a is expressed as n non-redundant radix β digits $a_{n-1}a_{n-2}\dots a_0$. Then the multiplication may be performed by summing together the multiples $a_{n-1}\beta^{n-1}b, a_{n-2}\beta^{n-2}b, \dots$, $a_0\beta^0 b$. The number of non-zero multiples is equal to $\nu_\beta(a)$, the number of non-zero digits of multiplier a. In Section 2.1 the function ν_β determines the data-dependent part of the required number of multiplications for performing an exponentiation. Similarly, in multiplication, ν_β determines the number of multiples to be summed. In worst case $\nu_\beta(a)$ is n and, hence, the worst case number of additions is $n - 1$. First consider the radix 2 case with digit set $\{0, 1\}$. As hinted by Booth [Boo51] it is possible to reduce the value of $\nu_2(a)$ by recoding the multiplier digits into the redundant signed digit set $\{\bar{1}, 0, 1\}$. Booth's recoding is based on the observation that a bit sequence of ones in a can be replaced by a redundant digit sequence containing precisely two non-zero digits, e.g. 01111 is equal to $1000\bar{1}$. This corresponds to replacing the sum $2^3 + 2^2 + 2^1 + 2^0$ by $2^4 - 2^0$. So, by recoding the multiplier to a redundant signed digit set the value of $\nu_2(a)$ may be reduced. Note, that the signed digits implies that both a positive and a negative version of the multiplicand b must be present.

In Section 2.5 is mentioned that an exponentiation method, suggested by Zhang et al. [ZMY88], is based on a recoding of the exponent to the radix 2 digit set $\{\bar{1}, 0, 1\}$. Zhang proves in [Zha93] that this recoding technique leads a value of $\nu_2(a)$ that is the least possible. According to [Kor93a, p.104] such a technique is called *the canonical recoding*. In 1960 Reitwiesner [Rei60] proposed a recoding technique that gives the representation of a with minimal $\nu_2(a)$. Reitwiesner also proved that a representation having this property is unique (i.e. Zhang et al's recoding is identical to Reitwiesner's recoding) and is characterised by the following constraints: Let $a_n a_{n-1} \ldots a_0$ be the non-redundant two's complement representation of a where bit a_n determines the sign of a, i.e. $a_n = 1$ if a is negative and $a_n = 0$ if a is non-negative. Further, let the "minimal" encoding of a be denoted $a'_n a'_{n-1} \ldots a'_0$. Then $a'_{i+1} \cdot a'_i = 0$ for all $i \in \{0, 1, \ldots, n-1\}$, i.e. no two consecutively indexed digits are both non-zero. Hence, the canonical recoding gives the bound $\nu_2(a) \leq \lceil \frac{n+1}{2} \rceil$ which compared to the non-redundant encoding approximately halves the worst case value of $\nu_2(a)$. This means that the number of multiples that have to be summed in a multiplication may be limited to $\lceil \frac{n+1}{2} \rceil$ for $n+1$ bit multipliers. Consequently, a faster computation must be expected and, furthermore, for some multiplication units, e.g. units based on a Wallace Tree, a reduction in hardware consumption is achieved. Reitwiesner also analyses the average value of $\nu_2(a)$ when the canonical recoding is applied. He finds that $\nu_2(a)$ approximates $\frac{n}{3}$ for large values of n. If a is non-redundant encoded the average value is $\frac{n}{2}$. The canonical recoding can be recursively described by the following equations where $c_{i+1} \in \{0, 1\}$ is a carry propagated from position i to position $i+1$,

$$\begin{aligned} c_{i+1} &= (a_i + a_{i+1} + c_i) \text{ div } 2, \quad \text{where } c_0 = 0 \\ a'_i &= a_i + c_i - 2c_{i+1} \quad \text{for} \quad i \in \{0, 1, \ldots, n\}. \end{aligned} \tag{3.4}$$

It is assumed that $a_{n+1} = a_n$, i.e. the sign of the two's complement representation of a is extended to position $n+1$ during the recoding. This implies that $c_{n+1} = a_n$. The canonical recoding results in a representation with value

$$\sum_{i=0}^{n} a'_i 2^i = \sum_{i=0}^{n} a_i 2^i - c_{n+1} 2^{n+1} = \sum_{i=0}^{n} a_i 2^i - a_n 2^{n+1} = \sum_{i=0}^{n-1} a_i 2^i - a_n 2^n,$$

which, indeed, is the value of the two's complement representation of a. The equations in (3.4) show that a carry may ripple from position 0 to position

n and, hence, that the digits a'_i must be computed from right to left. So, in multiplication methods where the multiplier digits are scanned from left to right or in methods where all digits must be simultaneously available the time for the canonical recoding may be significant. In right-to-left methods of multiplication the canonical recoding can be done on-the-fly while the multiplier is scanned. (In [ZMY88, Zha93] the left-to-right binary exponentiation method is used together with a canonical recoding of the exponent. Even though the time for the recoding, compared to exponentiation, is negligible it would be more natural to use the right-to-left method). Although the canonical recoding gives the minimal value of $\nu_2(a)$ it is not in wide use. Other recodings, with a worst case value of $\nu_2(a)$ equal to the worst case value of the canonical recoding, allow parallel computation of the multiplier digits.

An often used recoding method known as *Booth modified recoding* is suggested by MacSorley [[Mac61]. The idea in this method is to recode two consecutive bits in a two's complement representation into a single radix 4 signed digit belonging to the set $\{\overline{2}, \overline{1}, 0, 1, 2\}$. (This can also be identified as a radix 4 digit set conversion from the non-redundant digit set $\{0, 1, 2, 3\}$ into the redundant digit set $\{\overline{2}, \overline{1}, 0, 1, 2\}$). Since a radix 4 digit in the set $\{\overline{2}, \overline{1}, 0, 1, 2\}$ can be interpreted as two consecutive radix 2 digits in the set $\{\overline{1}, 0, 1\}$, where at most one of these digits is non-zero, Booth modified recoding also halves the worst case value of $\nu_2(a)$. A generalisation of Booth modified recoding, a radix 2^k digit set conversion from $\{0, 1, \ldots, 2^{k-1}\}$ into $\{\overline{2^{k-1}}, \overline{2^{k-1} - 1}, \ldots, 2^{k-1}\}$, can be described by the following equations where $i \in \{0, 1, \ldots, n\}$ and $c_0 = 0$,

$$\begin{aligned} c_{i+1} &= a_i \text{ div } 2^{k-1} \\ a'_i &= a_i + c_i - 2^k c_{i+1}. \end{aligned} \tag{3.5}$$

The carry c_{i+1} simply is the most significant bit in the binary encoding of a_i and, therefore, a'_i must belong to $\{\overline{2^{k-1}}, \overline{2^{k-1} - 1}, \ldots, 2^{k-1}\}$. Since a'_i is determined from a_i and the most significant bit of a_{i-1} there is no carry ripple effect. Consequently, the digits a'_i may be computed left-to-right, right-to-left or simultaneously. In particular, the radix 4 version of Booth modified recoding is widely used in the implementation of multiplication units. The reason is that multiples of the multiplicand b are restricted to the set $\{\overline{2b}, \overline{b}, 0, b, 2b\}$ and, hence, the multiples needed in the addition are just shifted versions of b or $\overline{b}$. If, instead, the radix 4 multiplier digits are encoded in the digit set $\{0, 1, 2, 3\}$ the multiple $3b$ is source for inconveniences: Since

$3b$ cannot be computed by a simple shift operation it must be precomputed and stored into a table in order not to delay the multiplication operation.

Some references for a more throughout treatment of multiplier recoding and its application in multiplication are [Rub75, VSH89, SG90, Kat94, Kor94a].

Comparison Operation

The methods for computation of modular addition use integer addition and integer comparison. In this section it has been shown how redundant representations lead to a computing time of addition that is independent on the operand lengths and, hence, that a very fast addition operation can be obtained. Unfortunately, a comparison of two redundant represented numbers cannot be performed faster than a comparison of two non-redundant represented numbers. For example, consider a redundant signed digit represented number as defined by Avižienis. Then a comparison can be done by a subtraction followed by an inspection of the resulting sign. The sign of the result is equal to the sign of the most significant non-zero digit and, consequently, in the worst case all digits must be inspected to find the sign of the number. This implies that the fastest methods for comparison must require a computing time that is proportional to the logarithm of the operand length. Thus, the comparison operation is a critical operation in modular multiplication methods.

Now, modular reduction and integer division are closely related, so the techniques known from division may be used in modular reduction too. A class of division methods is named *SRT division* after Sweeney, Robertson [Rob85] and Tocher [Toc58] who independently discovered the method. SRT division uses subtraction and shifting similar to the paper-and-pencil method. However, instead of an exact calculation of the quotient digits, which involves a slow comparison operation, an *estimate* of the quotient digits is used [Rob85, Atk68]. Opposed to an exact quotient determination a quotient estimation only uses a few of the most significant digits of the partial remainder and the divisor. Therefore, a faster computation can be achieved. The penalty of using a quotient estimate in modular reduction is that, sometimes, the result will differ from the correct result by a multiple of the modulus. This can be illustrated by a version of the simple modular addition method in Equation (3.2) where an estimate is used.

$$x +_{\mathbb{Z}_m} y = x + y - qm, \quad \text{where} \quad q = \begin{cases} 0 & \text{if } x + y - m - \Delta < 0 \\ 1 & \text{otherwise.} \end{cases} \tag{3.6}$$

Assume the comparison $x + y - m < 0$ in Equation (3.2) is replaced by a comparison where only a few of the most significant digits are used. This corresponds to the comparison $x+y-m-\Delta < 0$ where Δ is the truncation error. The possible sign and magnitude of Δ depends on how the number $x + y - m$ is represented and on the number of truncated digits. If the representation is the redundant signed digit representation Δ may take both positive and negative values, and if the carry save representation is used, Δ is non-negative. So, if $x + y - m - \Delta$ turns out to be zero, or close to zero, the correct sign of $x + y - m$ may depend on the sign of Δ and, hence, the quotient estimation technique in (3.6) may assign a wrong value to q. In such a situation the result from Equation (3.6) will be $(x + y) \bmod m \pm m$. However, in Section 3.3 it will be demonstrated that the quotient estimation technique can be applied during the computation of intermediate results. Efficient estimation of quotient digits is the subject of Section 3.8, 3.9 and 3.10.

3.2.3 Comparison of Non-Redundant and Redundant Representations

It has been shown that redundant number representations often are preferable in applications where a large number of additions are needed. Multiplication is an example of such an application. The time for performing an addition becomes independent of the operand digit-length. The time for adding two carry save represented numbers is about equal to the delay of two full adders. Similar computing times are expected for other radix 2 redundant represented numbers. The fastest non-redundant addition is proportional to the logarithm of the operand digit-lengths. Hence, for very long operands the redundant number representation is superior to the non-redundant number representation. It has also been shown how a multiplier efficiently can be recoded into a redundant representation. Hereby, the number of terms to be summed in a multiplication is about halved. There are, however, some costs implied by redundant representations. First, an *increased hardware consumption* may be expected. The registers for holding a redundant represented number consume more circuitry than registers for non-redundant represented numbers: Due to the enlarged digit set the circuitry for holding a single digit must be able to hold more digit values. Second, the enlarged digit set also implies an increased complexity for the circuitry to add two redundant digits. Furthermore, even though redundant addition is fast, the overhead for *conversion from redundant into non-redundant representation* should be considered before a redundant representation is applied for a computation. According to Kornerup [Kor94a], a conversion from redundant into non-redundant representation can be done in a time that is proportional to $\log_2 n$. Further, Kornerup notes, that for *any* conversion from a redundant digit set into a non-redundant digit set there exist situations where the most significant digit of the result depends on the least significant digit of the number being converted and, hence, such a conversion must take logarithmic time. As an example consider a conversion from redundant signed digit representation into non-redundant representation. This can be achieved by a non-redundant addition: Split the redundant integer into two non-redundant integers, such that one integer consists of all positive digits and the other integer consists of all negative digits from the redundant represented integer. Then these two non-redundant integers are added by a non-redundant adder, and a conversion into non-redundant representation is achieved.

3.3 Residue Representation and Arithmetic

In the preceeding section it was shown how redundant representations can lead to very fast addition and multiplication of integers. Unfortunately, comparison cannot be performed as fast as addition and, consequently, the comparison operation needed in the modular addition method is limiting the efficiency of modular addition. Further, while integer doubling is obtained by a simple shift operation, the comparison operation makes modular doubling about just as hard as modular addition. In the preceeding section it was also discussed how the quotient estimation technique can improve the efficiency of modular reduction. However, the result from a modular reduction based on a quotient estimate, see p. 73, may differ from the correct result by a multiple of the modulus. In the following it will be discussed how these results can be viewed as other valid representations of the result obtained from a correct computation.

Consider a modular reduction of z, i.e. the computation of z mod m. If the quotient estimation technique is applied the result of the computation may not be z mod m but, certainly, it will belong to the set

$$[z] = \{x : x = z \bmod m + k \cdot m,\ k \in \mathbb{Z}\} = \{x :\ x = z + k \cdot m,\ k \in \mathbb{Z}\}. \tag{3.7}$$

An element $x \in [z]$ is called a *residue of* z *modulo* m and $[z]$ is called the *residue class of* z *modulo* m. If two elements, say x and y , both are residues of z module m then x is *congruent to* y *modulo* m, which is written by the notation $x \equiv y \pmod{m}$, or by $x \equiv_m y$. Now, the aim in a modular reduction of z is to compute the smallest positive residue, denoted z mod m, in $[z]$. The result of a quotient estimation technique is not necessarily z mod m but, indeed, it is a residue of z modulo m. Such a residue can be viewed as another representation of z mod m. By allowing more than a single representation of z mod m a kind of *redundant residue representation* is obtained. The redundancy of this representation must not be mixed up with the redundancy of the integer representation in Section 3.2.2. Indeed, these two types of redundancies are independent of each other. *The advantage of the redundant integer representation is a fast addition operation while the advantage of the redundant residue representation is a fast quotient determination.*

The following well known arithmetic rules, e.g. [Knu68, p. 39] or [Den82, p. 37], show that modular addition, subtraction and multiplication can be replaced by the corresponding ordinary integer operations without the residue class of the result is being changed. Since modular exponentiation is computed by modular multiplication this observation applies to modular exponentiation as well.

$$\begin{aligned} x + y &\equiv_m (x + y) \bmod m \\ x - y &\equiv_m (x - y) \bmod m \\ x \cdot y &\equiv_m (x \cdot y) \bmod m \end{aligned} \tag{3.8}$$

So, if it is allowed to represent an intermediate result, say z mod m, by other residues in the same residue class $[z]$ then the basic modular operations may be simplified. As example consider modular multiplication. This operation is computed by a number of modular additions. If redundancy is allowed in the representation of all intermediate results the quotient estimation technique can be used for obtaining a faster computation. It is only the final result that has to be converted into the (non-redundant) residue in the interval $\mathbb{Z}_m$. Further, observe that if the modular multiplication is part of a computation of a modular exponential it is not necessary to perform this conversion on the intermediate modular products.

According to Equation (3.8) there is really no need to complicate the intermediate computations with modular reductions and, hence, quotient determinations. This is only required in the final conversion. In fact, an addition or subtraction of a multiple of m is just a change of representation in the same residue class,

$$z + q \cdot m \equiv_m z \bmod m, \quad \text{where } q \in \mathbb{Z}.$$

However, since the cardinality of a residue class is infinite it is necessary to limit the number of allowable representations in order to limit the hardware consumption for registers and processing elements. Of course, if the number of intermediate operations is finite and the input is finite a computation will lead to a finite output. But for some modular operations the operand length may be significant if all intermediate modular operations are performed by the corresponding integer operation. An extreme example is modular exponentiation $b^e \bmod m$ as used in the RSA crypto system where the input operands are, at least, 512 bits integers. Without any modular reduction of the intermediate results these might achieve lengths up to $512 \cdot 2^{512}$ bits! (This is a tremendous huge number. The reader is encouraged to estimate the number of atoms in the complete universe and make conclusions of his/her own). A less extreme example is modular multiplication $(a \cdot b) \bmod m$. If a and b are n bit numbers then the product $a \cdot b$ may consume up to $2n$ bits. There are, indeed, several proposals for computing $(a \cdot b) \bmod m$ by an ordinary integer multiplication followed by a modular reduction, e.g. [NS81, MA85, KH88, Eve90, Sau92]. In particular the early designs of dedicated hardware for modular exponentiation uses this strategy for modular multiplication, e.g. [Riv80, ST83, VVDJ90]. Compared to a computation method based on the simple modular addition method, described in Section 3.1, these methods require an addition unit and registers that must be able to handle operands of double the length. Hence, an increased circuit size must be expected for dedicated hardware implementations of these modular multiplication methods. Furthermore, as will be shown in Section 3.5, the computing time for performing a modular reduction $z \bmod m$ by a SRT division method is about equal to the computing time for performing a modular multiplication $(a \cdot b) \bmod m$ by a similar method. So, there seems to be no reasonable argument in favour of splitting the modular multiplication operation into an ordinary integer multiplication followed by a modular reduction.

The quotient determination method controls the range of the intermediate operands. The smallest range is obtained by an exact calculation of the quotients while the estimation technique leads to an increased range and, consequently, an increased hardware consumption. In general, the operand range increases with the "inaccuracy" of the quotient determination while the time for determining the quotient is expected to decrease. Further, the final conversion of a result into a residue in the range $\mathbb{Z}_m$ increases in complexity when the range of the intermediate operands increases. So, the decision on the allowable range of intermediate operands must be a tradeoff between the hardware consumption, the complexity of quotient determination and the complexity of the final conversion. In some computations, e.g. modular exponentiation, the computation is dominated by many intermediate operations and, hence, the computing time for performing the final conversion may be neglected.

3.4 Left-to-Right Modular Multiplication Method

Nearly all contributions in the literature on modular multiplication methods are variations on the β-ary left-to-right exponentiation method described by Equation (2.7) in Section 2.1.1. In this chapter the radix is restricted to a power of two, i.e. $\beta = 2^k$ for some integer $k > 0$. The 2^k-ary left-to-right modular multiplication method can be described by the congruence,

$$(a \cdot b) \bmod m \equiv_m 2^k(\cdots 2^k(2^k(a_{n-1}b) + a_{n-2}b) + \cdots) + a_0 b. \tag{3.9}$$

The operation $(a \cdot b) \bmod m$ is a specialisation of the general operation $(2^{kn}r + a \cdot b) \bmod m$, which can be described by

$$(2^{kn}r + a \cdot b) \bmod m \equiv_m 2^k(\cdots 2^k(2^k(2^k r + a_{n-1}b) + a_{n-2}b) + \cdots) + a_0 b. \tag{3.10}$$

Both the computation of (3.9) and of (3.10) can be done by an intermediate operation that computes a residue modulo m of expressions of the form $2^k s +$

a_ib, i.e. an operation that computes $2^ks + a_ib - q_im$ for some quotient digit q_i[2]. This operation is, indeed, a very useful and important operation: If $a \cdot b$ is zero the operation becomes $2^ks - q_im$, which is the basic operation in some division methods [Rob85], and if q_im is zero for all i, the operation becomes $2^ks + a_ib$ which is the basic operation in left-to-right integer multiplication methods. Hence, the basic operation $2^ks + a_ib - q_im$ of the left-to-right modular multiplication method can be seen as a merge, or generalisation, of the known basic operations of integer multiplication and division.

As described in Section 3.1 a computation of a residue modulo m of 2^ks+a_ib may be subdivided into a number of modular doublings and modular additions. Indeed, this is the approach in [TY92] where Takagi and Yajima develop a radix 2 and a radix 4 modular multiplication method. Takagi and Yajima use redundant signed digit representation to achieve a fast addition and, moreover, by allowing residues in the range $]-m; m[$ they can utilise the quotient estimation technique in the computation. Another approach is to postpone the quotient determination until after the calculation of $2^ks + a_ib$ instead of using modular doubling and modular addition as primitives in the computation. This is the most commonly used approach. In the next section it will be shown that the computation of the intermediate operation $2^ks+a_ib-q_im$ can be made about as fast as the computation of 2^ks-q_im. The latter operation computes a residue modulo m of 2^ks which corresponds a modular k-fold doubling. So, exemplified by the case where $k = 1$, this shows that it is advantageous to postpone the modular reduction until after the doubling and the addition is performed. Apparently, the additional time for performing the addition operation is vanishing in comparison with the time for just performing a modular doubling. This may also be the reason that right-to-left modular multiplication method, which is similar to the right-to-

[2]Lu et al. [LHLH88] have proposed a generalisation of $(a \cdot b) \bmod m$ into $(a \cdot b + c) \bmod m$. Lu et al. use this operation in place of modular multiplication in the left-to-right exponentiation method and achieve a method for evaluation of polynomials module m. If c is 2^k-ary encoded, $c = \sum_{i=0}^{n-1} c_i 2^{ki}$, the intermediate operation in the computation of $(a \cdot b + c) \bmod m$ becomes $(2^ks + c_i) + a_ib - q_im$. The computation of $2^ks + c_i$ is obtained by the usual left-shift of s with the modification that c_i, in place of a zero valued digit, is shifted into the least significant digit position of s. So, compared to the intermediate operation $2^ks + a_ib - q_im$ Lu et al.'s generalisation do not increase the computationally efforts.

In summary, if Lu et al.'s generalisation is unified with the generalisation in (3.10) it is seen that a residue modulo m of the expression $(2^{kn}r + a \cdot b + c)$ may be computed by intermediate operations of the form $(2^ks + c_i) + a_ib - q_im$.

left exponentiation method in Section 2.1.1, only has been considered in a single article [FDG90]. Suppose the parallel computation of the right-to-left method in Section 2.3 is utilised. Then, two basic operations of the form $2^k y - q_j m$ and $x + a_i y - q_i m$, respectively, are computed in parallel. The first of these operations is seen to be identical to the above discussed modular k-fold doubling operation. So, unless some method, that is faster than the computation of $2^k s + a_i b - q_i m$, is proposed for the computation of $2^k s - q_i m$ there will be no benefit of using the right-to-left modular multiplication method—not even if the parallel computation is utilised.

As seen from Equation (3.9) the left-to-right modular multiplication can be computed by n executions of the intermediate operation $2^k s + a_i b - q_i m$ where n denotes the number of radix 2^k digits required to represent the multiplier a. Assume ℓ binary digits is sufficient to represent the value of a. Then, compared to a radix 2 modular multiplication method, the required number of intermediate operations can be reduced to about $\frac{\ell}{k}$ by using a radix 2^k version of the operation. Indeed this observation is the motivation for considering the use of high radices: *The aim of the high-radix modular multiplication approach is to obtain a faster computation by reducing the required number of intermediate operations. However, if the computing time for a single radix 2^k version of the intermediate operation increases by a rate greater than or equal to k, there may be no point in using a high radix.* The remainder of this chapter is devoted to efficient computation of intermediate operations of the form $2^k s + a_i b - q_i m$ or of a form similar to this.

3.5 Utilisation of Parallel Computations

Using the notation introduced in Section 2.3 the left-to-right method in Equation (3.9) can be described by a set of recursive equations,

$$(a \cdot b) \bmod m \equiv_m S_0, \quad \text{where} \quad \begin{array}{rcll} S_i &=& R_i - q_i m, & S_n = 0, \\ R_i &=& 2^k S_{i+1} + a_i b. & \end{array} \tag{3.11}$$

The value of R_i is the result of a "shift-and-add" operation and S_i denotes the result of a modular reduction of R_i. The quotient digit q_i must be determined from the the value of R_i and modulus m. In Figure 3.4 a slice of the dependence graph for this method is depicted. Note that the dependence graph is more "fine-grained" than the equations in (3.11), i.e. there is more nodes in the graph than there is equations in (3.11): The black node corresponds to an equation that expresses the determination of quotient digit q_i and the shaded nodes correspond to equations that compute the multiples a_ib and q_im. To keep the notation simple these equations have been left out. The dashed lines in the figure symbolise the start-point of a new cycle in the computation of the recursive equations. The computation of the nodes between two consecutive start-points is denoted the computation of a *recursion cycle*. As seen from the data dependencies the computing time for a single recursion cycle is determined by the time for adding the multiple a_ib to 2^kS_{i+1}, the time for determining a quotient digit q_i, the time for computing the multiple q_im, and the time for subtracting q_im from R_i. Note, that the multiple a_ib can be computed *in parallel* with the other operations and, hence, the time for computing a_ib has no influence on the time for a recursion cycle. This is because a_i and b are part of the input to the modular multiplication, so the only demand is that the computation of a_ib should by completed before it is needed in a recursion cycle.

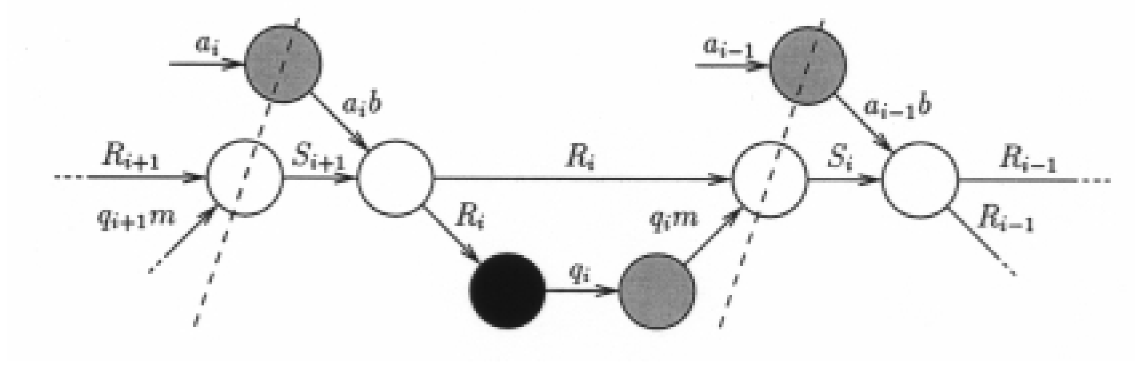


Figure 3.4: Slice of dependence graph for the left-to-right method based on a recursive evaluation of $S_i = (2^kS_{i+1} + a_ib) - q_im$.

Now, it is possible to reduce the recursion cycle time by rearranging the computation of the values in (3.11). In Figure 3.4 it is seen that the determination of q_i is delayed by the computation of $R_i = 2^kS_{i+1} + a_ib$. It is, however, not necessary to perform the addition of a_ib after the modular reduction of R_{i+1}: It may be done *before* the subtraction of $q_{i+1}m$, and *in parallel* with the determination of q_{i+1} and the computation of multiple $q_{i+1}m$. Another set of recursive equations that mirrors such a computation can be developed by rewriting (3.11),

$$\begin{aligned} R_i &= 2^k S_{i+1} + a_i b \\ &= 2^k (R_{i+1} - q_{i+1} m) + a_i b \\ &= (2^k R_{i+1} + a_i b) - 2^k q_{i+1} m \\ &= T_i - 2^k q_{i+1} m, \quad \text{where} \\ T_i &= 2^k R_{i+1} + a_i b. \end{aligned}$$

Hence, the computation of T_i can be performed in parallel with the computation of $2^k q_{i+1} m$. If the initial value $S_n = 0$ is replaced by the initial value $R_n = 0$, (3.11) can be written as,

$$(a \cdot b) \bmod m \equiv_m S_0, \quad \text{where} \quad \begin{aligned} S_0 &= R_0 - q_0 m, \\ R_i &= T_i - 2^k q_{i+1} m, \quad R_n = 0, \\ T_i &= 2^k R_{i+1} + a_i b \end{aligned}$$

Further, to simplify these equations the recursion depth is enlarged by one, which means $i \in \{-1, 0, \dots, n-1\}$, and the multiplier digit $a_{-1} = 0$ is introduced. This gives,

$$R_1 = T_1 - 2^k q_0 m = (2^k R_0 + a_{-1} b) - 2^k q_0 m = 2^k (R_0 - q_0 m).$$

Therefore, the result S_0 can be expressed as R_{-1} div 2^k which is easily obtained by a right-shift of R_{-1}. Hereby, the rearrangement of (3.11) is completed and the following set of recursive equations is obtained,

$$(a \cdot b) \bmod m \equiv_m R_{-1}, \quad \text{where} \quad \begin{aligned} R_i &= T_i - 2^k q_{i+1} m, \quad R_n = 0, \\ T_i &= 2^k R_{i+1} + a_i b. \quad a_{-1} = 0 \end{aligned} \tag{3.12}$$

The new dependence graph in Figure 3.5 shows how a parallel computation of T_i and of $2^k q_{i+1} m$ can be utilised to improve the computing time for a recursion cycle. Indeed, assuming that the most time consuming path in the figure is the path through the quotient determination node and the node computing the multiple $2^k q_{i+1} m$, *a recursion cycle time for modular multiplication that is comparable to the recursion cycle time of SRT division methods is achieved.* As long as the computing times for determination of a quotient digit and for computation of a multiple $2^k q_{i+1} m$ have not be discussed, the recursion cycle times cannot be properly compared. However, this will be done in detail in the sections from 3.7 to 3.10. The computing time for a SRT division recursion cycle turns out to be about equal to the computing time for a modular multiplication recursion cycle. According to (3.12) the cost of using this computation strategy is an additional recursion cycle. For large operands this additional cost is vanishing.

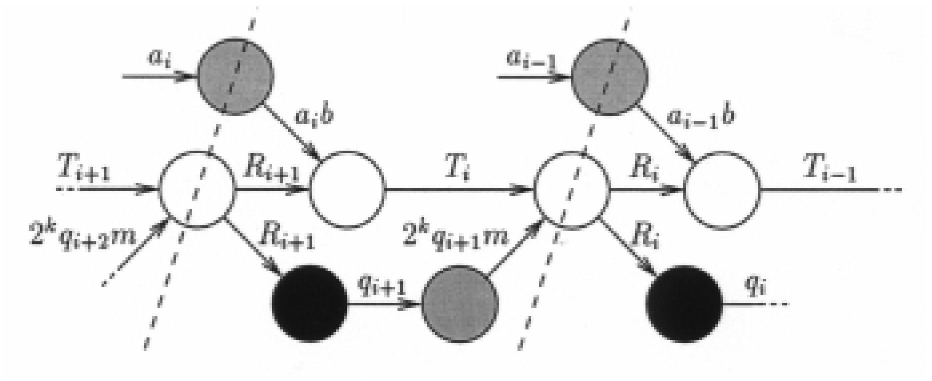


Figure 3.5: Slice of dependence graph for the left-to-right method based on a recursive evaluation of $R_i = (2^kR_{i+1} + a_ib) - 2^kq_{i+1}m$.

It should be mentioned that the method illustrated by Figure 3.5 is similar to the method described in the article in Appendix A. This article describes a hardware architecture that, indeed, is a direct mapping of the five nodes in a single recursion cycle onto five processing elements. In Figure 3.6 the hardware architecture is shown. The VLSI processor described in Chapter 4 has a very similar hardware architecture. However, since the pipelined exponentiation method, see Section 2.3.1, requires that two modular multiplications are performed simultaneously the architecture is pipelined. This implies that the computation of a recursion cycle is divided into two sub-computations and that a pipeline buffer is inserted somewhere in the architecture. Furthermore, the architecture of the VLSI processor has an additional multiplier register.

Assume a modular multiplication is based on (3.12) and assume the computation utilises the possibilities for improving the recursion cycle time by performing some of the operations in parallel. Then, as indicated by the dependence graph in Figure 3.5 and by the hardware architecture in Figure 3.6, the computing time is determined by the time for determining a quotient digit q_{i+1}, the time for computing the multiple $2^kq_{i+1}m$, and the time for subtracting this multiple from T_i. These three operations will be the issue of the discussion in the following sections.

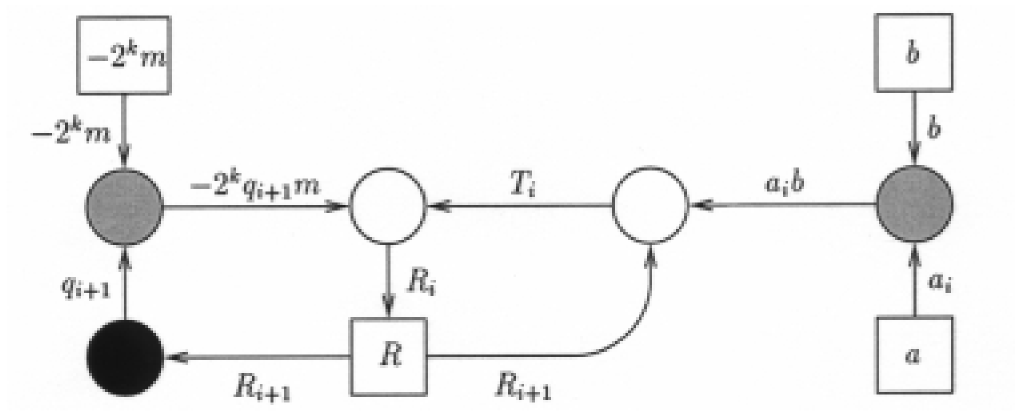


Figure 3.6: Hardware architecture for computation of $R_i = (2^k R_{i+1} + a_i b) - 2^k q_{i+1} m$.

3.6 Representation of Intermediate Operands

In the light of the discussions on integer representations in Section 3.2 and on residue representations in Section 3.3 a general scheme for methods of computations of modular operations can be made. The scheme described below is based on the very general, and simple, observation that if the benefits of converting a computation into another "domain" are greater than the costs imposed by the conversion into and back from this domain then a more efficient computation is achieved. Furthermore, the scheme is directed toward recursive or iterative computation methods where a modular operation is computed by repeated application of an intermediate operation. Indeed, all the methods for computation of modular operations considered in this thesis have this property: Modular exponentiation is computed by repeated application of modular multiplication, and modular multiplication is computed by repeated application of an intermediate operation, e.g. $2^k R_{i+1} + a_i b - 2^k q_{i+1} m$.

Stimulus: All inputs that may be a result from a previous application of the present modular operation are imposed by a *restriction on the residue range and on the integer representation*. Denote this restriction by $\mathbb{D}$.

Response: The result from the present modular operation must fulfil restriction $\mathbb{D}$. This ensures that the present modular operation may be invoked repeatedly without any concerns about the validity of the residue range or integer representation of the intermediate results.

Method: The present modular operation is computed by means of repeated application of an intermediate modular operation. First, if the intermediate operation has a restriction on the input values that differs from $\mathbb{D}$, say $\mathbb{E}$ a conversion of the input values from $\mathbb{D}$ into $\mathbb{E}$ must be performed. Then the intermediate operation can be applied repeatedly until the computation is completed. Finally, a conversion of the result from $\mathbb{E}$ back to $\mathbb{D}$ is done.

As example, consider the evaluation of modular exponentials $b^e \bmod m$ in the RSA crypto system. In this application $\mathbb{D}$ denotes non-redundant binary represented integers in the range $\mathbb{Z}_m$. The intermediate operation in exponentiation is a multiplication operation with the restriction $\mathbb{E}$. An often used restriction on the multiplier, the multiplicand and, hence, the resulting product is that these must be non-redundant binary represented integers in the residue range $[0; 2m[$. Since $\mathbb{D}$ is included in $\mathbb{E}$ no conversion is needed before the intermediate modular multiplication operations are applied. A conversion of the final result from $\mathbb{E}$ into $\mathbb{D}$ is, however, required. This can be accomplished by a single non-redundant subtraction of m.

Until now, the only restriction on the quotient digit q_i is that it must be an integer. However, if the allowable range of residues is bounded an enhanced restriction on the quotient determination is imposed. Suppose the range of residues is bounded to the symmetric range $[-\alpha m; \alpha m]$[3]. Then the quotient determination in (3.11) and in (3.12) may be described by the operation,

$$\{ \text{ Determine integer } q_i \text{ such that } |R_i - q_i m| \leq \alpha m \ \}. \tag{3.13}$$

Obviously, the residue range must include at least m integers, so $\alpha \geq \frac{1}{2}$. The parameter α, in some sense, specifies the redundancy of the residue range. If $\alpha = \frac{1}{2}$ the residue range is non-redundant and the quotient estimation technique cannot be applied. For increasing values of α the computational efforts required to perform the quotient determination in (3.13) are expected to be decreasing. However, the complexity of the quotient determination also

[3]The notation and analysis in this chapter is inspired by Kornerup's description in [Kor93b]. Kornerup considers *symmetric* ranges because he uses a redundant signed digit integer representation with digit set $\{\overline{\sigma}, \dots, \sigma\}$ to encode the multiplier digits a_i and the quotient digits q_i. There are several descriptions of modular multiplication methods that use non-negative residue ranges and non-negative digit sets. The article in Appendix A is one example. In [Wal91a] Walter presents an analysis of a high-radix modular multiplication method with non-negative residue ranges and non-negative digit sets.

depends on the range of R_i. If this range is large compared to $[-\alpha m; \alpha m]$ the *quotient digit set*, i.e. the set of quotient digit values that may be the result of (3.13), will be comparatively large. Therefore, the complexity of the quotient determination is expected to increase for increasing cardinality of the quotient digit set, too. Also the complexity of the *computation of multiples* $q_i m$ is increasing for increasing cardinality of the quotient digit set. In the following, the range $[-\delta\alpha m; \delta\alpha m]$ will denote the range of R_i, and $q^{\max}$ will denote the maximal required absolute value of a quotient digit in (3.13), i.e. $|q_i| \leq q^{\max}$. With these range restrictions imposed on the intermediate operation $2^k S_{i+1} + a_i b - q_i m$ the computation can be described as,

Algorithm 3.6—1 (Intermediate operation $2^k S_{i+1} + a_i b - q_i m$)

Stimulus: S_{i+1}, a_i and b, where $|S_{i+1}| \leq \alpha m$ and $|a_i b| \leq (\delta - 2^k)\alpha m$.

Response: S_i, where $|S_i| \leq \alpha m$.

Method:
```
$R_i := 2^k S_{i+1} + a_i b$;
    { Determine integer $q_i$ such that $|R_i - q_i m| \leq \alpha m$ and $|q_i| \leq q^{\max}$ }
    $S_i := R_i - q_i m$;
```

Note that no restrictions on the integer representation of S_i and S_{i+1} are stated in this description. However, in order to obtain fast addition and subtraction a redundant representation is usually applied. The range restriction on $a_i b$ ensures that $|R_i| \leq \delta\alpha m$. Obviously, $\delta \geq 2^k$. In the same way, and with the same comments, the computation of the intermediate operation $2^k R_{i+1} + a_i b - 2^k q_{i+1} m$ can be described by,

Algorithm 3.6—2 (Intermediate operation $2^k R_{i+1} + a_i b - 2^k q_{i+1} m$)

Stimulus: R_{i+1}, a_i and b, where $|R_{i+1}| \leq \delta\alpha m$ and $|a_i b| \leq (\delta - 2^k)\alpha m$.

Response: R_i, where $|R_i| \leq \delta\alpha m$.

Method:
```
$T_i := 2^k R_{i+1} + a_i b$;
    { Determine integer $q_{i+1}$ such that $|R_{i+1} - q_{i+1} m| \leq \alpha m$ and $|q_{i+1}| \leq$
    $q^{\max}$ }
    $R_i := T_i - 2^k q_{i+1} m$;
```

The computing time for these intermediate operations depends on the time for performing addition and subtraction. As seen from the range restrictions the operand lengths can be expected to be longer than modulus m. Hence, in applications with moduli of several hundreds of bits, as in the RSA crypto system, *the only reasonable choice of addition techique is redundant addition* when aiming for fast modular multiplication. This implies that the operands S, R and T in the above descriptions are redundant represented. The times for computing multiples and for determining quotient digits also have influence on the total computing time for the above intermediate operations. The next section will discuss methods for computation of multiples. Hereafter, the interdependencies of α, δ and $q^{\max}$ will be made clear and techniques for estimation of quotient digits will be discussed.

3.7 Computation of Multiples

In Section 3.5 it was seen that a fast computation of multiples $q_i m$, or $2^k q_{i+1} m$, is important for obtaining a fast recursion cycle time in the modular multiplication methods. In this section the computation of multiples of modulus m and of multiplicand b will be discussed. There are, in general, three issues to consider in the approaches to camputation of the multiples: The digit set for the multiplier digit a_i and the quotient digit q_i, the integer representation of the multiplicand, and the integer representation and residue range of the resulting multiple.

3.7.1 Multiplier Digit Set and Quotient Digit Set

If a digit set is restricted to zero and to powers of two, e.g. $\{\overline{2}, \overline{1}, 0, 1, 2\}$ the computation of a multiple is particularly easy. Then, as mentioned in Section 3.2.2 page 71, the computation of a multiple requires, in worst case, a simple shift of b or $\overline{b}$. However, if only powers of two are accepted as valid digit values the radix 2^k is bounded to 2 or to 4. This is one of the main reasons that radix 2 and radix 4 modular multiplication methods dominate the literature. In fact, the only dedicated hardware construction that applies a high-radix method with radix greater than 4 is the VLSI processor described in Chapter 4 where the radix is 32. In this processor the multiplier digit set is $\{0, 1, \ldots, 31\}$ and the quotient digit set is $\{0, 1, \ldots, 42\}$. Instead of precomputing a table of the possible multiples $a_i b$ and $q_i m$ these multiples are computed "on-the-fly" when they are needed in each cycle of the multiplication algorithm. The drawback, compared to the precomputation approach, is an increased computing time for a cycle in the multiplication. However, in the precomputation approach additional time for precomputing the tables must be included in the total computing time. Furthermore, when the multiplication method is hardware implemented the circuitry for tables of these sizes consumes a significant area—an area that, in this radix 32 implementation, is estimated to be significantly larger than the area for the circuitry to compute the multiples.

Now, suppose a hardware implementation is constructed and there is room for a table of precomputed multiples. How is this table best utilised? According to Section 3.5 the computation of $a_i b$ can be performed in parallel with the quotient determination and with the computation of a multiple of m, so the computing time for $a_i b$ is not as essential as the computing time for $q_i m$. Furthermore, a precomputation of the possible values of $q_i m$ is only required once for each change of m. In modular exponentiation the modulus is fixed for each modular multiplication, so the additional time for performing this precomputation is negligible. This is not the case if the table is used for precomputed values of $a_i b$. Consequently, *the best solution for a hardware implementation is a "hybrid" consisting of a table for holding precomputed values of $q_i m$ and consisting of circuitry for on-the-fly computation of $a_i b$.*

The principle behind the technique for computation of multiples in the VLSI processor is described in the articles in Appendix A and B. The technique can be identified as a *multiplier recoding combined with a redundant addition by a Wallace Tree* (see Section 3.2.2): In the processor a single radix 32 digit, say q_i, in the set $\{0, 1, \ldots, 42\}$ is recoded into three radix 4 digits d_0, d_1, d_2 such that $q_i = 4^2 d_2 + 4^1 d_1 + 4^0 d_0$ and $d_0, d_1 \in \{\overline{1}, 0, 1, 2\}$ and $d_2 \in \{0, 1, 2\}$. Then the multiple $q_i m$ is computed as the sum of the three terms $4^2 d_2 m, 4^1 d_1 m$ and $4^0 d_0 m$. Since all three terms merely are shifted versions of m or $\overline{m}$ they are easy obtainable. The sum is computed by a carry save adder which, indeed, is a three-input Wallace Tree. Thus, the time for computing a multiple is equal to the recoding time plus the time for adding three integers by a redundant carry save adder.

It should be noted that the computation scheme used in the VLSI processor can be improved: As described by Kornerup in [Kor93b] a radix 64 digit in the *symmetric* set $\{\overline{42}, \ldots, 42\}$ can be recoded into three radix 4 digits in the set $\{\overline{2}, \overline{1}, 0, 1, 2\}$ and, hence, the radix can be increased from 32 to 64 without increasing the time for computing a multiple. In general, if a radix 2^{2p} digit q_i is restricted to the symmetric set of digit values $\{\overline{q^{\max}}, \ldots, q^{\max}\}$, where $q^{\max} \leq \frac{2}{3}(2^{2p} - 1)$, then q_i can be recoded into p radix 4 digits in the set $\{\overline{2}, \overline{1}, 0, 1, 2\}$ and the corresponding multiple can be computed as a sum of p terms. A fast way of computing this sum is to use a Wallace Tree or a similar tree that is constructed by redundant adders.

In the above discussions on how to compute multiples two alternatives are mentioned: Precompute a table or use a multiplier recoding technique combined with a tree of redundant adders. If it is advantageous for a concrete implementation these alternatives may be combined, such that a small table of precomputed multiples can be used as input to a tree of adders. For example, if the value of $3m$ is precomputed all multiples $q_i m$ where $q_i \in \{\overline{63}, \ldots, 63\}$ can be expressed as a sum of three terms such that $q_i m = 4^2 d_2 m + 4^1 d_1 m + 4^0 d_0 m$ and $d_0, d_1, d_2 \in \{\overline{3}, \ldots, 3\}$. Hence, by precomputing the multiple $3m$ it is possible to compute a wider range of multiples by the same tree of redundant adders.

3.7.2 Representation of Multiplicand and Modulus

The representation of b an m has influence on the time and on the circuitry required to compute the multiples $a_i b$ and $q_i m$. Almost all proposed methods for modular multiplication use a non-redundant binary encoding of b and m. Regarding the representation of modulus m this is natural since, mostly, m is part of the input for an application that performs some modular operations. Also, if modular multiplication is viewed as an application in its own the multiplicand b may be assumed to be non-redundant binary represented. However, if b is an intermediate operand, as in the computation of modular exponentials, it is a result from a previous computation. Therefore, if redundant adders is used in the multiplication, a conversion from the intermediate redundant representation into non-redundant binary representation is required after each multiplication. This will enlarge the total computing time. In modular exponentiation it is not even possible to utilise a computation scheme where the next multiplication is performed in parallel with a conversion of the current result: In all the exponentiation methods presented in Chapter 2, the result from the current modular multiplication is used as multiplicand in the immediate succeeding multiplication.

Redundant Representation of the Multiplicand

As a consequence of this *conversion overhead* it is suggested in the article in Appendix B that *no conversion performed after each modular multiplication* and, therefore, that the multiplicand b is redundantly represented. Tagaki and Yajima [TY92, Tak92] have proposed a radi.x 4 modular multiplication method that is based on the same idea. Also Morita [Mor89] uses a redundant representation of the intermediate operands in a radix 4 method. However, in Morita's method the sign of the result is inspected after each multiplication. Since such an inspection, or comparison, is about as fast (or slow) as a conversion into non-redundant representation Morita does not get full advantage of the redundant representation. The penalty for using a redundant representation of b is a more complicated computation of the multiple a_ib. For example, if b is in carry save representation it is encoded in two non-redundant binary integers b_s and b_c such that $b = b_s + b_c$. This implies that the formation of multiple a_ib expands to a formation of two multiples plus an addition, $a_ib = a_ib_s + a_ib_c$. Hence, if a tree of adders is used for computing a_ib, as described in Section 3.7.1, it must be able to add twice as many terms as in the case of a non-redundant represented multiplicand. This means an increased computing time and an increased hardware consumption for the computation of a_ib. However, as explained in Section 3.5 the computation of a_ib may be performed in parallel with the other computations, so the time for a recursion cycle does not have to be affected by a redundant representation of multiplicand b.

The VLSI processor in Chapter 4 uses a non-redundant binary representation of the multiplicand. To perform a conversion from carry save representation into this non-redundant binary representation a binary adder has been included in the architecture. The article in Appendix C lists the area consumption of various parts of the processors circuitry. In the article the binary adder is denoted a “high-speed adder”. It is seen that this adder consumes about 14 percents of the area occupied by circuit cells. Suppose the processor is using the carry save representation for both the multiplicand and the multiplier. *Then there is no need for the binary adder since the final conversion into non-redundant representation may be performed by a full adder during the process of outputting the result serially from the processor.* The additional circuitry required to handle carry save represented multiplicands and multipliers is three registers for holding the operands (the processor utilises the pipelined computation of modular exponentials described in

Section 2.3.1, hence the circuitry includes one multiplicand register and two multiplier registers), a unit for computing a multiple (two 4-1 multiplexers, a 3-1 multiplexer and a carry save adder), and two carry save adders for adding the multiples a_ib_s and a_ib_c. The additional area for this circuitry is, according to the data in the article, about equal to 2.2 times the area of a binary adder. Hence, in total such a modification will require an extra area corresponding to 1.2 times the size of a single binary adder; a total area expansion of about 17 percents. How much, then, is the total computing time improved? The VLSI processor uses 115 recursion cycles to perform a modular multiplication plus a time corresponding to 7 recursion cycles for the conversion. So, the computing time for this particular processor may be improved by about 6 percents.

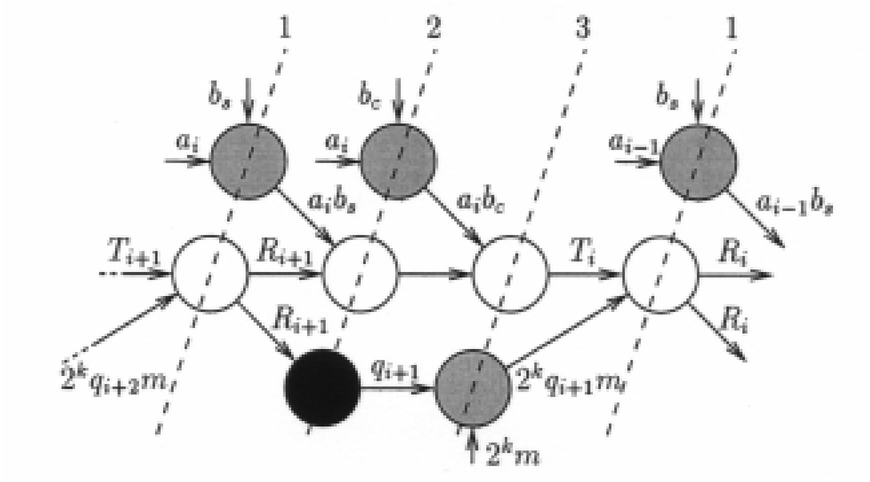


Figure 3.7: Slice of dependence graph for the left-to-right method based on a recursive evaluation of $R_i = ((2^kR_{i+1} + a_ib_s) + a_ib_c) - 2^kq_{i+1}m$.

Time-Multiplexed Computation

In the VLSI processor the overhead for conversion is about 6 percents. This processor uses a radix 32 multiplication method. But, if higher radices is used for the implementation the time spend for conversion will be more significant and, regarding the computing time, the advantage of using redundant represented multiplicands will be greater. The drawback of using even higher radices is an increase of hardware consumption. Motivated by this, a computation schedule that reduces the required amount of circuitry is proposed in the article in Appendix B. When the multiplicand is redundantly represented, e.g. in carry save representation, the intermediate operation $2^k R_{i+1} + a_i b - 2^k q_{i+1} m$ expands to $2^k R_{i+1} + a_i b_s + a_i b_c - 2^k q_{i+1} m$. A slice of the dependence graph for this operation is shown in Figure 3.7. As indicated in the figure the dependence graph can be mapped onto three processing elements: The white nodes are mapped onto a processing element that performs redundant addition, the shaded nodes are mapped onto an element that computes multiples, and the black node is mapped onto an element that performs the quotient determination. The computation of a single recursion cycle is then scheduled into three sub-cycles as shown by the numeration of the dashed lines. This computation schedule can be identified as a *time-multiplexed computation schedule* where three additions are performed by a single shared processing element and, similarly, the computations of three multiples are performed by another shared processing element.

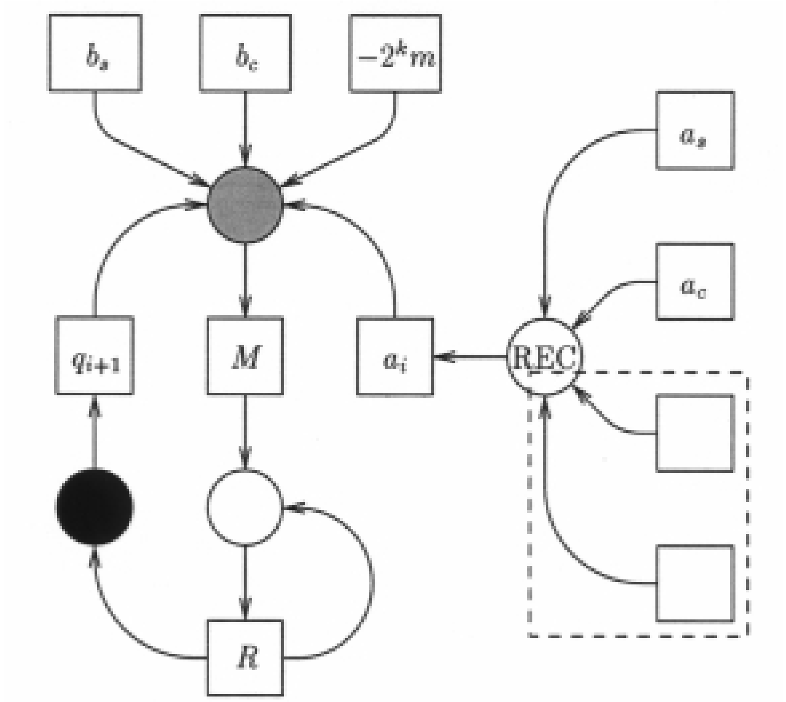


Figure 3.8: Hardware architecture for time-multiplexed computation of $R_i = ((2^k R_{i+1} + a_i b_s) + a_i b_c) - 2^k q_{i+1} m$.

A hardware architecture for the time-multiplexed computation of $2^k R_{i+1} + a_i b_s + a_i b_c - 2^k q_{i+1} m$ is shown in Figure 3.8. The processing element denoted REC performs the recoding of the multiplier. This recoding is not stated explicitly in the recursive equations describing the computation method, but, as mentioned in Section 3.7.1, it is used for obtaining an efficient computation of multiples. An example is the radix 4 recoding from carry save representation into the digit set $\{\overline{2}, \overline{1}, 0, 1, 2\}$ described by Morita in [Mor89]. A similar proper encoding of the quotient digit is assumed to be performed during the quotient determination by the black processing element. To simplify the computation of the multiples $a_i b_s, a_i b_c$ and $-2^k q_{i+1} m$ into a computation of the general form $x_i y$ where x_i is some radix 2^k digit and y is some full-length operand, the value of modulus m has been replaced by $-2^k m$. This makes the architecture of the shaded processing element simpler. Although it is not illustrated at Figure 3.8 the computation performed by the white processing element may by simplified too: The white processing element performs additions of the general form $y + z$ and $2^k y + z$, where y and z denote some full-length operands. If $\frac{k}{3}$ is an integer value, the computation of the white processing element can be simplified to a single type of addition with the general form $2^{\frac{k}{3}} y + z$. This is seen by the following rewriting,

$$\begin{gathered} ((2^k R_{i+1} + a_i b_s) + a_i b_c) - 2^k q_{i+1} m = \\ 2^{\frac{k}{3}} (2^{\frac{k}{3}} (2^{\frac{k}{3}} R_{i+1} + a_i 2^{-\frac{k}{3}} b_s) + a_i 2^{-\frac{2k}{3}} b_c) - 2^k q_{i+1} m \end{gathered}$$

Thus, if the right-shifted version $2^{-\frac{2k}{3}} b_s$ is replacing b_s and $2^{\frac{k}{3}} b_c$ is replacing b_c in Figure 3.8 the simplification is obtained. The part of the architecture in the dashed box shows the extra multiplier required if the pipelined computation of exponentials is supported by the architecture. This pipelined computation also requires that each processing element is pipelined into a two-stage pipeline and, hence, that additional pipeline buffers must be inserted into each processing element.

The time for computation of a recursion cycle by the time-multiplexed computation schedule is equal to the sum of the times for the three sub-cycles in Figure 3.7. This time can be written as,

$$\max(white, shaded) + \max(white, shaded, black) + \max(white, shaded), \tag{3.14}$$

where *white*, *shaded* and *black* are synonyms for the computing time of an operation performed by a processing element of the corresponding colour.

Using the same notation, the computing time of a recursion cycle performed by Algorithm 3.6–2, and described by the dependence graph in Figure 3.5, Section 3.4, can be written as,

$$\mathit{white} + \max((\mathit{black} + \mathit{shaded}), \mathit{white}). \tag{3.15}$$

Now, to be able to compare the recursion cycle time of these methods it is assumed that the computing times for the processing elements are ordered by $\mathit{white} \leq \mathit{shaded} \leq \mathit{black}$. The validity of this assumption is not obvious. However, if the radix is sufficiently large the computation of a multiple includes a redundant addition and, hence, it is reasonable to assume $\mathit{white} \leq \mathit{shaded}$. Further, as will be discussed in Section 3.10, the time for determination of a quotient digit q_{i+1} is comparatively large for all known methods that are feasible to implement in hardware. If this assumption is accepted the recursion cycle time in (3.14) becomes *shaded* + *black* + *shaded* and the time in (3.15) becomes *white* + *black* + *shaded*. So, the recursion cycle time is increased by a time corresponding to *shaded* – *white* when the multiplicand and the multiplier is redundant represented and the time-multiplexed computation schedule is used. To make any comparisons of the resulting total computing time for a modular multiplication it is necessary to know the concrete times *shaded* and *white* in the time-multiplexed method and the concrete time for the conversion into non-redundant representation required by the method in Figure 3.5. Compared to the architecture of the latter method the architecture in Figure 3.8 requires three extra registers: two registers for holding redundant representations of a and b and one unspecified register, M, for holding multiples in some representation. The circuitry for the registers holding the digits a_i and q_{i+1} are negligible. On the other hand the number of white and shaded processing elements are halved and there is no longer a need for circuitry that implements the conversion from redundant into non-redundant representation.

It should be noted that the time-multiplexed computation schedule also can be applied for computations where the multiplier and multiplicand is non-redundant represented. Indeed, if these operands are non-redundant represented the hardware architecture is reduced by one of the registers holding b_s, or b_c and by one of the registers holding a_s or a_c. Then, compared to the architecture in Figure 3.6 the architecture in Figure 3.8 requires an extra register M for holding multiples while the number of white and shaded processing elements are halved. Further, if the computation of $a_i b$ is performed in sub-cycle 2 (see Figure 3.5) the computing time for a recursion cycle in the time-multiplexed computation schedule in (3.14) becomes *white* + *black* + *shaded* which is equal to the time in (3.15). Hence, when the multiplier and multiplicand are non-redundant represented the time-multiplexed computation schedule results in the same computing time and, moreover, in a reduced hardware consumption.

The above considerations on the recursion cycle time for the time-multiplexed computation schedule are only valid if each of the three sub-cycles is performed as fast as possible. That is, the discussions on the computing time are not valid if the *same* period of time is assigned to each sub-cycle. In a hardware implementation, that uses a *global clock signal* to synchronise the operations performed in various parts of the architecture, it is therefore not optimal to assign a single clock period to each of the sub-cycles. In such an implementation the minimal clock period is bounded by the time for performing the slowest of the sub-cycles which probably is the quotient determination. Hence, to get full advantage of the time-multiplexed computation schedule each sub-cycle must be divided into a number of clock periods that corresponds to the required computing time for each particular sub-cycle. The drawback of this approach is that a very fast global clock signal must be generated and distributed to the processing elements in the implementation. For very fast clock signals this might be impossible or, at least, a highly demanding technical challenge. There is, however, a way of avoiding the global clock signal in a hardware implementation. This is by using *asynchronous circuit design* or *self-timed circuit design* where the synchronisation between communicating processing elements is performed locally between the involved processing elements. Williams and Horowitz [WH91, Wil93] have developed such techniques and applied these in a hardware implementation of SRT division.

3.7.3 Representation and Range of Resulting Multiple

Consider the computation of $(2^k R_{i+1} + a_i b) - 2^k q_{i+1} m$ where the redundant addition technique is applied for computing $T_i = 2^k R_{i+1} + a_i b$ and $R_i = T_i - 2^k q_{i+1} m$. As explained in Section 3.2.2 it is faster to add a redundant represented integer and a non-redundant represented integer than it is to add two redundant represented integers. Further, the adder used for the latter addition is more complex. In a computation where T_i is computed in parallel with the determination of q_{i+1} and the computation of $q_{i+1}m$ the recursion cycle time cannot be improved by using a non-redundant representation of a_ib. However, since the recursion cycle time depends on the time for computing R_i the time can be (slightly) reduced by using a non-redundant representation of $q_{i+1}m$. The only feasible way of obtaining a non-redundant representation of $q_{i+1}m$ is by precomputing all possible multiples of m and holding these multiples in a table: If the multiples are computed on-the-fly, the time for performing the conversion into non-redundant representation will be much longer than the time saved by the subtraction in the computation of expression $R_i = T_i - 2^k q_{i+1}m$.

The recursion cycle time does not depend on the representation chosen for the resulting multiple a_ib. But, the *residue range of a_ib* does affect the quotient determination and, hence, the recursion cycle time. In Section 3.6 the range of a_ib is given by the stimulus restriction of Algorithm 3.6–1 and 3.6–2 $|a_ib_i| \leq (\delta - 2^k)\alpha m$. The parameter α is determined by the precision of the quotient estimation and δ is determined by the range of a_ib. As discussed in Section 3.6 a small value of δ leads to a small quotient digit set and, therefore, the complexity of the quotient determination is expected to be comparatively smaller. Furthermore, a small quotient digit set leads to a less complex computation of the multiples $q_{i+1}m$.

Now, if multiplicand b is the result from a previous modular multiplication, it will be restricted to some residue range given by the response restriction of that modular multiplication method. In some multiplication methods this range is positive, e.g. $[0; 2m[$. Then, the maximal absolute value of b can be halved by converting b into a *symmetric residue range*, e.g. a residue in $[0; 2m[$ can be converted into a residue in $[-m; m]$ by a subtraction of m. In [Mor89, Mor90] Morita uses this technique for converting a multiplicand from the range $[0; m[$ into the range $[-\frac{1}{2}m; \frac{1}{2}m]$. Furthermore, by recoding the multiplier digit a_i into a *symmetric digit set* the smallest absolute value of $|a_ib|$, and therefore δ, is obtained. It is, however, possible to obtain a further reduction of the value of δ. The idea is to perform a modular reduction of the multiple a_ib and hereby to replace the computation of $T_i = 2^k R_{i+1} + a_ib$ by $T_i = 2^k R_{i+1} + a_ib - q_i'm$, where q_i' is a quotient digit determined from the value of a_ib and m. The computation of T_i can still be scheduled such that it is completed before the value of T_i is needed in the subsequent computation of $R_i = T_i - 2^k q_{i+1}m$. At very best, if an exact determination of q_i' is performed, this idea leads to a residue of a_ib modulo

m in the range $[-\frac{1}{2}m; \frac{1}{2}m]$. So, $|a_ib| \le \frac{1}{2}m = (\delta - 2^k)\alpha m$ which gives the value $\delta = 2^k + \frac{1}{2\alpha}$. This is the smallest obtainable value of δ when a modular reduction of a_ib is performed.

In [Mor89, Mor90] Morita describes another approach for keeping the value of δ small. Morita's goal is to obtain the symmetric radix 2^k digit set $\{\overline{2^{k-1}}, \dots, 2^{k-1}\}$ for both the multiplier digits and for the quotient digits. As will be shown in the next section, this is the smallest possible quotient digit set. Therefore, the computationally effort required to compute the multiples will be the least possible. In Morita's method the penalty for achieving this digit set is a more complex quotient determination: In the determination of quotient digit q_i Morita includes the value of the multiple $a_{i-1}b$ of the next recursion cycle. Morita's method can be described by the following intermediate operation

Algorithm 3.7–1 (Morita's intermediate operation)

Stimulus: S_{i+1}, a_i, a_{i-1} and b, where $|2^kS_{i+1} + a_ib| \le 2^k\alpha m, |a_{i-1}| \le 2^{k-1}$ and $|b| \le \frac{1}{2}m$.

Response: S_i, where $|2^kS_i + a_{i-1}b| \le 2^k\alpha m$.

```
Method: $R_i := 2^kS_{i+1} + a_ib$;
    { Determine integer $q_i$ such that $|R_i + 2^{-k}a_{i-1}b - q_im| \le \alpha m$ };
    $S_i := R_i - q_im$;
```

So, Morita achieves a simple computation of multiples a_ib and q_im at the cost of a more complex quotient determination. In this method the value of δ is specified by the bound $|R_i + 2^{-k}a_{i-1}b| \le \delta\alpha m$ which gives $2^k\alpha m + 2^{-k}2^{k-1}\frac{1}{2}m = \delta\alpha m$. Hence, $\delta = 2^k + \frac{1}{4\alpha}$ which is smaller than the value of δ obtained by a modular reduction of a_ib.

Fortunately, there is another, much simpler, way to obtain a small value of δ. In Section 3.9 it will be shown how a simple scaling of modulus m can give values of δ that are even smaller than the minimal values obtained above. Furthermore, the scaling technique neither implies a major modification of the architecture, as imposed by the modular reduction approach, or an increased quotient determination complexity, as imposed by Morita's approach.

3.8 Residue Range and Quotient Digit Set

In this section the interdependencies of the residue range of R_i, $[-\delta m; \delta m]$, the residue range of $R_i - q_i m$, $[-\alpha m; \alpha m]$, and the required quotient digit set, $\{\overline{q^{\max}}, \ldots, q^{\max}\}$, will be discussed. Recall the specification of the quotient determination operation in Section 3.6,

$$\{ \text{ Determine integer } q_i \text{ such that } |R_i - q_i m| \leq \alpha m \text{ and } |q_i| \leq q^{\max} \ \}$$

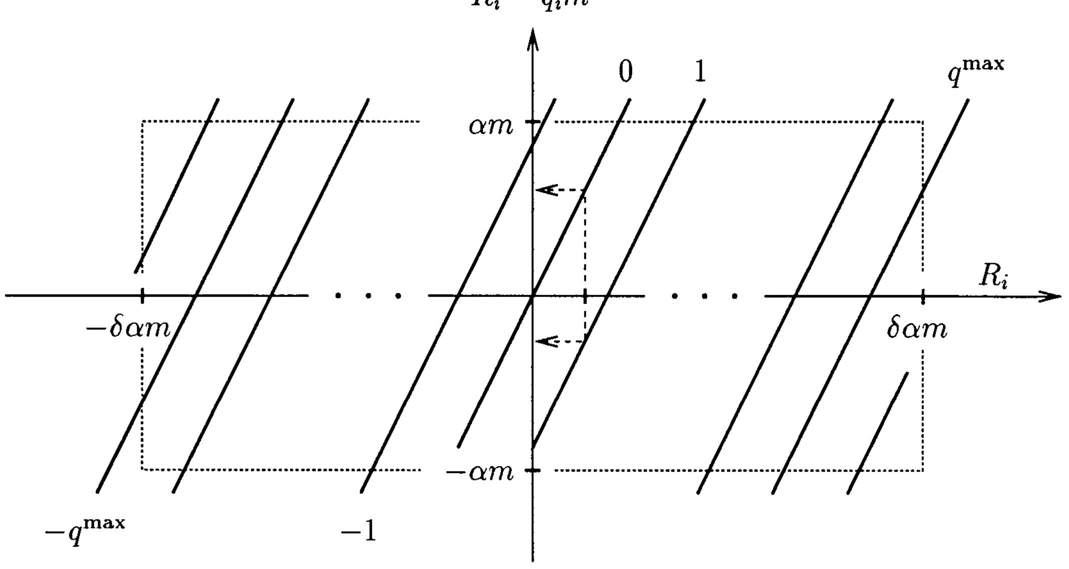


Figure 3.9: Robertson Diagram showing the modular reduction of R_i.

The quotient determination can be visualised by the *Robertson Diagram* in Figure 3.9. This type of diagram was used by Robertson [Rob85] for explaining the arithmetic operations performed during division. The figure shows how a modular reduction of a value R_i (the horizontal axis) maps R_i onto the residue $R_i - q_i m$ (the vertical axis) for various choices of q_i. Since the aim is to obtain a residue in the range $[-\alpha m; \alpha m]$ only a few values of q_i can be selected. An example where either 0 or 1 can be selected for q_i is shown by the dashed arcs in the figure. If more than a single quotient digit can be selected for some values of R_i, the value of α is greater than $\frac{1}{2}$. If $\alpha = \frac{1}{2}$ there is only a single proper quotient digit for each R_i and, consequently, an exact determination of q_i is required. So, *an increasing value of α is expected to decrease the computationally effort required to determine a proper quotient digit.* On the other hand, a large value of α makes the final conversion of a residue from the range $[-\alpha m; \alpha m]$ into the range $[0; m[$ more complicated. To ensure that the final conversion can be done by a single inspection of the sign and a single addition of m, some authors require that $\alpha < 1$[4]. This is, however, not particularly important for the application in this thesis: In a computation of modular exponentials the intermediate operands do not need such a conversion after each multiplication, so the time for performing the final conversion is negligible compared to the total computing time for a modular exponentiation.

From Figure 3.9 the maximal required absolute value, $q^{\max}$, of a quotient digit is seen to be determined by the constraint,

$$\begin{aligned} \delta\alpha m - q^{\max} m &\in [-\alpha m; \alpha m] \\ q^{\max} &\in [(\delta-1)\alpha; (\delta+1)\alpha]. \end{aligned} \tag{3.16}$$

Hence, $q^{\max} = \lceil(\delta-1)\alpha\rceil$ is sufficient to perform the modular reduction in the figure. Since this equation is central for the complexity of quotient determination and the computation of the multiples $q_i m$ it deserves to be stated as a theorem,

Theorem 3.8–1 (Interdependencies of α, δ and $q^{\max}$)
All values of R_i where $|R_i| \le \delta\alpha m$ can be reduced to $R_i - q_i m$, such that $|R_i - q_i m| \le \alpha m$, by selection of a quotient digit q_i in the digit set $\{\overline{q^{\max}}, \dots, q^{\max}\}$ where,

$$q^{\max} = \lceil(\delta-1)\alpha\rceil. \tag{3.17}$$

In Section 3.6 it was seen that the parameter δ is bounded by $\delta \ge 2^k$, so Equation (3.17) states that it is not possible to both obtain a large value of α and a small value of $q^{\max}$. This means *the complexity of the determination of quotient digit q_i cannot be reduced without increasing the complexity of the computation of multiples $q_i m$, and vice versa.*

[4]Some modular multiplication methods do not use symmetric digit sets for the multiplier digits and quotient digits. These methods are based on positive digits and positive multiplicands. In these methods the corresponding requirement as that $R_i - q_i m$ must be a residue in the range $[0; 2m[$ which has the same width as the range $]-m; m[$. This is also the case for the methods described in the articles in Appendices A and B.

Equation (3.17) is useful for analysis of a given modular multiplication method. For example, Kornerup [Kor93b] derives the maximal allowable value of α when $q^{\max}$ is bounded by $q^{\max} \leq \frac{2}{3}(2^k - 1)$ for even values of k. As explained in Section 3.7.1, this particular bound on $q^{\max}$ gives an advantageous computation of multiples. So, by this analysis Kornerup aim for the simplest possible quotient determination given a fixed method for computation of multiples.

Since $\alpha \geq \frac{1}{2}$ the following lower bound of $q^{\max}$ can be derived by inserting this bound and the bound $\delta \geq 2^k$ into Equation (3.17),

$$q^{\max} \geq \left\lceil (2^k - 1)\frac{1}{2} \right\rceil = 2^{k-1} \text{ for all } k \in \{1, 2, \ldots\}. \tag{3.18}$$

This shows that the quotient digit set $\{\overline{2^{k-1}}, \ldots, 2^{k-1}\}$ used in Morita's method is the least possible. The maximal allowable value of α in Morita's method, described by Algorithm 3.7–1, can be derived from Equation (3.17). Using $q^{\max} = 2^{k-1}$ and $\delta = 2^k + \frac{1}{4\alpha}$ the following upper bound of α is achieved,

$$\begin{aligned} 2^{k-1} \geq (\delta - 1)\alpha &= (2^k - 1)\alpha + \tfrac{1}{4} \\ \alpha \leq \tfrac{2^{k-1} - \frac{1}{4}}{2^k - 1} &= \tfrac{1}{2} + \tfrac{1}{4(2^k - 1)}. \end{aligned} \tag{3.19}$$

k	$\alpha \leq$
1	$\frac{3}{4}$
2	$\frac{7}{12}$
3	$\frac{15}{28}$
4	$\frac{31}{60}$

The table shows some sample values of the bound. The upper bound is greater than $\frac{1}{2}$ for all $k \in \{1, 2, \ldots\}$, so it is indeed possible to perform a modular multiplication with the constraint $q^{\max} = 2^{k-1}$ and still use the quotient estimation technique. Note, the bound decreases for increasing values of k. Hence, the *required precision of the quotient determination increases for increasing radices* and, consequently, Morita's "high-radix" method is best suited for small radices.

From Equation (3.17) it follows that $q^{\max} \geq (\delta - 1)\alpha$. If this bound is inserted into the stimulus restriction of Algorithm 3.6–1 and 3.6–2, the following bound on the range of $a_i b$ is obtained,

$$\begin{aligned} |a_i b| &\leq (\delta - 2^k)\alpha m \\ &\leq (q^{\max} - (2^k - 1)\alpha)m. \end{aligned} \tag{3.20}$$

Further, by insertion of $q^{\max} = 2^{k-1}$ it follows that $|a_i b| \leq ((2^k - 1)(\frac{1}{2} - \alpha) + \frac{1}{2})m$. So, even if the minimal parameter value $\alpha = \frac{1}{2}$ is used, the range of $a_i b$ is bounded by $|a_i b| \leq \frac{1}{2}m$. Thus, as discussed in the previous section, it requires a modular reduction of $a_i b$ to achieve this minimal range of the quotient digits. Moreover, an exact quotient determination is required.

3.9 Scaling of Modulus

In this section it will be shown how a simple scaling of the modulus m can improve the value of parameter δ and, therefore, allow a larger value of α in a computation where some fixed value of $q^{\max}$ has been chosen.

In Algorithm 3.6–1 and 3.6–2 the stimulus sets up a restriction on the range of $a_i b$, $|a_i b| \leq (\delta - 2^k)\alpha m$. The left-to-right modular multiplication methods given by (3.11) and (3.12) in Section 3.5 compute a residue of $(a \cdot b) \bmod m$ in the residue range $[-\alpha m; \alpha m]$. Since, it is desirable to use the result of a previous multiplication as input to the next multiplication without performing any conversion of the residue range, it is assumed that multiplicand b is bounded by $|b| \leq \alpha m$. Further, it is assumed that no modular reduction of the multiple $a_i b$ is performed during the recursion cycles of the modular multiplication methods. Let $a^{\max}$ denote the maximal absolute value of the multiplier digits, i.e. $|a_i| \leq a^{\max}$ for all indices i. The minimal value of $a^{\max}$ is obtained by using a symmetric digit set for the encoding of multiplier a. Obviously, $a^{\max} \geq 2^{k-1}$. Under these assumptions the bound of $a_i b$ can be expressed by,

$$|a_i b| \leq a^{\max} \alpha m \leq (\delta - 2^k)\alpha m. \tag{3.21}$$

This gives the bound $\delta \geq 2^k + a^{\max}$. Further, by insertion into (3.17) it is seen that,

$$q^{\max} \geq (2^k + a^{\max} - 1)\alpha. \tag{3.22}$$

Now, suppose the intermediate operation in Algorithm 3.6–1 or 3.6–2 is performed by a scaled version m' of modulus m, i.e. $m' = c \cdot m$ for some positive integer constant c, and suppose the multiple b still is bounded by $|b| \leq \alpha m$. Then the new stimulus restriction becomes $|a_i b| \leq (\delta - 2^k)\alpha m'$ and, consequently, parameter δ is bounded by $\delta \geq 2^k + c^{-1}a^{\max}$. The term $c^{-1}a^{\max}$ is the only part of δ that depends on the range of $a_i b$. Hence, *by scaling* *the modulus it is possible to make the contribution from the range of $a_i b$ arbitrary small.* The scaling of m changes (3.22) to,

$$q^{\max} \geq (2^k + c^{-1}a^{\max} - 1)\alpha. \tag{3.23}$$

Apparently, the scaling technique gives a solution of one problem at the cost of introducing another problem: How can the restriction $|b| \leq \alpha m$ be maintained when a modular multiplication is performed with the modulus $m' = c \cdot m$? According to the above discussion the result will be in the range $[-\alpha cm; \alpha cm]$. The trick, that gives an efficient solution to the latter problem, is to scale the multiplier a as well. Thus, a residue of $(a \cdot b)$ mod (mc) is computed in place of $(a \cdot b)$ mod m. Then, a residue of $(a \cdot b)$ mod m in the range $[-\alpha m; \alpha m]$ may be computed by an integer division,

$$\begin{aligned} x &\equiv (ac \cdot b) \bmod (mc), \text{ where } |x| \leq \alpha cm \\ x \text{ div } c &\equiv (a \cdot b) \bmod m, \text{ where } |x \text{ div } c| \leq \alpha m. \end{aligned} \tag{3.24}$$

By choosing a scaling constant equal to a power of two, i.e. $c = 2^r$ where $r \in \{0, 1, \ldots\}$, the scaling of m and a just requires a simple left-shift. The integer division by c requires a simple right-shift. However, there may be an additional cost imposed by scaling the multiplier a: The modular multiplication methods in Section 3.5 need n, respectively $n + 1$, recursion cycles where n is the number of radix 2^k multiplier digits. If the scaled multiplier ac requires more than n digits the number of recursion cycles must be increased. Using the notation introduced in Section 3.6, the left-to-right modular multiplication method in (3.12), based on the intermediateoperation in Algorithm 3.6–2 and the scaling technique, can be described by,

Algorithm 3.9–1 (Modular multiplication with scaling of modulus)

Stimulus: Scaled multiplier $a' = a2^r = \sum_{i=0}^{n'-1} a'_i 2^{ki}$ and $a'_{-1} = 0$, where $|a'_i| \leq a^{\max}$.
Multiplicand b, where $|b| \leq \alpha m$.
Scaled modulus $m' = 2^r m$.
Scaling constant 2^r, where $r \in \{0, 1, \ldots\}$.

Response: $S_0 \equiv_m (a \cdot b)$, where $|S_0| \leq \alpha m$.

```
Method: $R_{n'} := 0$;
    for $i := n'-1$ downto $-1$ do $R_i := 2^k R_{i+1} + a'_i b - 2^k q_{i+1} m'$;
    $S_0 := R_{-1}$ div $2^{k+r}$;
```

In Algorithm 3.9–1 the symbol n' denotes the number of radix 2^k digits of the scaled multiplier a'. Hence, $n'-n$ additional recursion cycles are introduced by the scaling technique.

In [Bri82] Brickell describes a radix 2 modular multiplication method that utilises the scaling technique. In this method the scaling factor corresponds to choosing $r = 10$, so 10 additional recursion cycles is needed. In the articles in Appendix A and B, and in Kornerup's method in [Kor93b], the value $r = k$ is chosen. This results in a single additional recursion cycle. Also the VLSI processor described in Chapter 4 utilises the scaling technique with $r = k = 5$.

The effect of the scaling technique can be illustrated by Kornerup's modular multiplication method, which is identical to Algorithm 3.9–1. Kornerup introduces the bound $q^{\max} \le \frac{2}{3}(2^k - 1)$ and restricts k to even values. From (3.23) it follows that,

$$\begin{aligned} \alpha &\le \frac{q^{\max}}{2^k + c^{-1}a^{\max} - 1} && (3.25) \\ &\le \frac{2}{3} \cdot \frac{(2^k - 1 + c^{-1}a^{\max}) - c^{-1}a^{\max}}{2^k + c^{-1}a^{\max} - 1} \\ &= \frac{2}{3} - \frac{2}{3} \cdot \frac{1}{\frac{c}{a^{\max}}(2^k - 1) + 1} && (3.26) \end{aligned}$$

Hence, by this bound of $q^{\max}$ the value of α is smaller than $\frac{2}{3}$ for all k. Through the scaling constant c it is possible to adjust the bound of α such that it is arbitrary close to $\frac{2}{3}$, In Kornerup's quotient determination method it is essential to maximise the value $\lfloor \log_2(\alpha - \frac{1}{2}) \rfloor$. Since $\alpha < \frac{2}{3}$ this maximum is -3, and it is obtained if $\alpha \ge \frac{1}{8} + \frac{1}{2} = \frac{5}{8}$. So, how should the scaling constant c be selected to maximise $\lfloor \log_2(\alpha - \frac{1}{2}) \rfloor$? Using (3.26) it follows that the bound,

$$c \ \ge \ \frac{15a^{\max}}{2^k - 1}$$

is sufficient to ensure the existence of a value $\alpha \ge \frac{5}{8}$. Kornerup assumes $a^{\max}$ is bounded by the same value as $q^{\max}$. By insertion of this bound, it is seen that $c \geq \frac{45}{2}$. So, if $c = 2^r$ it is sufficient to choose $r \geq 5$. The bound (3.26) becomes a constant, $\frac{2}{3} \cdot \frac{48}{49}$, when c is equal to 2^5. Kornerup uses the value $c = 2^k$ which, consequently, is suboptimal for radices given by $k \leq 4$.

In [Tak92] Takagi describes a radix 4 modular multiplication method that is a variant of Algorithm 3.9–1. Takagi's method is based on the other intermediate operation, given by Algorithm 3.6–1, in Section 3.6. Takagi uses a scaling constant of 4 and the restriction $q^{\max} = a^{\max} = 2$. It appears this method is very similar to Kornerup's method for $k = 2$. However, the quotient determination methods differ for these modular multiplication methods. Takagi restricts the range of α to range $]\frac{9}{16}; \frac{5}{8}[$. Although not explicitly stated by Takagi, this is equivalent to demanding $\lfloor \log_2(\alpha - \frac{1}{2}) \rfloor = -4$. So, also Takagi's method might benefit from choosing a scaling constant of 2^5.

In Section 3.7.3 it was claimed that the scaling technique is superior to Morita's method. Morita uses the restriction $q^{\max} = a^{\max} = 2^{k-1}$. If a scaling constant $c = 2^{k+1}$ is applied, Equation (3.25) reduces to,

$$\alpha \quad \leq \quad \frac{1}{2} + \frac{3}{8} \cdot \frac{1}{2^k - \frac{3}{4}}$$

By comparing this bound to the bound derived from Morita's method in Equation (3.19) it is seen that the scaling technique allows greater values of α and, hence, is superior to Morita's method. Note, the bound for α is decreasing for increasing values of k. This is also the case for Morita's method, so none of these methods are suited for high radices.

Finally, according to (3.25) the bound of α is limited by the value of $q^{\max}$ and the radix 2^k. It is seen, that *no matter how large the scaling constant is chosen and how $a^{\max}$ is restricted* the bound of α is limited by,

$$\alpha \quad \leq \quad \frac{q^{\max}}{2^k - 1} \tag{3.27}$$

Using the fact that $\delta \geq 2^k$, this bound is seen to be a direct consequence of Theorem 3.8–1.

3.10 Quotient Determination Methods

In this section methods for determination of quotient digits will be discussed. Recall the specification of the quotient determination operation,

$$\{ \text{ Determine integer } q \text{ such that } |R - qm| \leq \alpha m \text{ and } |q| \leq q^{\max}\}.$$

The indices, referring to the recursion cycle of the operation, are left out of the notation in this section. The range of R is bounded by $|R| \leq \delta\alpha m$. When $\alpha > \frac{1}{2}$ there may be more than a single value of q that fulfils the requirement of the quotient determination operation, $|R - qm| \leq \alpha m$. Then, as indicated in Section 3.2.1, it is possible to perform this operation by inspecting just a few of the most significant digits of R and of m and, thus, providing an estimate of the quotient digit. For high-radix SRT division methods the quotient determination operation has been identified as a very time-critical operation of a recursion cycle [Tay85]. This is also the case for high-radix modular multiplication methods.

In Section 3.5 and 3.7 the quotient determination operation was symbolised by a black node that computes some proper quotient digit *value* in some proper *encoding*. In the previous sections the emphasis has been on the value of the quotient digit. This will be the case in this section as well. But, according to Section 3.7 the encoding of the value q is important for the efficiency of the computation of the multiple qm. So, regarding the computing time there are two issues to consider in the quotient determination operation: The computing time for determining the value of q and, if a subsequent recoding is necessary, the computing time for this recoding.

The methods for determination of quotient digits can be divided into two classes which will be denoted respectively *on-the-fly computation methods* and *table look-up methods*: The methods in the first class perform a computation of the quotient digit in each recursion cycle. The computation is based on a few of the most significant digits of R and m. Since the computation is done in every recursion cycles this class of methods is denoted on-the-fly computation methods. The methods of this class can be further divided into two subclasses. The subclasses are characterised by the approach used for the computation: The first approach is based on *comparison operations*. By comparing a few of the most significant digits of R to a number of comparing constants the value of R is estimated. More precisely, an interval to which R must belong is identified. A quotient digit value is associated to each of these intervals. This quotient digit value is then returned as the result of the quotient determination operation. This subclass of methods can be seen as a generalisation of the modular reduction method used for performing the simple modular addition described in Section 3.1 and further discussed in Section 3.2.2, page 73. Some approaches based on comparison operations are described in [Bri82, Tay85, Bak87, FDG90, KH90a, Mor90, Tak92] and in the article in Appendix B. The second approach of the on-the-fly computation methods is based on a *multiplication by an approximation of the reciprocal of modulus m.* Here, the idea is to obtain an approximation to the exact quotient digit $q = \lfloor \frac{R}{m} \rfloor$ by performing a multiplication of truncated versions of R and $\frac{1}{m}$. This approach for determining a quotient digit in modular multiplication methods is described in [NS81, Miy82, Bar86] and it is used in the article in Appendix A.

The methods of the other class, the table look-up methods, utilise a precomputation strategy. The idea of these methods is to minimise the required time of the quotient determination operation performed in each recursion cycle by precomputing a table of quotient digit values. These values are stored in the encoding required for the subsequent formation of multiple qm. A quotient digit is then found by a simple table look-up. The table look-up methods have a shorter computing time than the on-the-fly computation methods. However, the table look-up methods require additional time for precomputing the table. The remainder of this section will be devoted to table look-up methods. In 3.10.1 the method and the notation used in the subsequent descriptions are introduced. In 3.10.2 the required number of most significant digits of R and m is analysed. Subsection 3.10.3 comprises a detailed analysis of the case where operand R is redundant represented. As well, the complexity for the quotient determination operation is compared for a SRT division method and a modular multiplication method with the same fixed value of $q^{\max}$. Finally, the required number of table entries is discussed. In Section 3.10.4 it is shown how the quotient determination complexity can be reduced by adjusting the range of modulus.

3.10.1 Table Look-Up Methods

The idea of the table look-up methods is to precompute a table with an entry for each pair $(\hat{R}, \hat{m})$ where $\hat{R}$ and $\hat{m}$ refers to a sufficient number of the most significant digits of R respectively m. Then, the quotient determination operation can be expressed as the table look-up,

$$q := \text{QuotientDigitTable}(\hat{R}, \hat{m}).$$

To minimise the required time for a computation of the table and to minimise the need for storage it is important to use as few digits as possible of R

and m. Of course, in a hardware implementation the table can be placed in a permanent storage at the time of construction and, then, no time for the precomputation is required. In applications where modulus m is fixed during many quotient determination operations it is advantageous to utilise the knowledge of the value of m during the precomputation. Hereby, the number of entries is reduced to the number of values of $\hat{R}$ and the quotient determination operation reduces to,

$$q := \text{QuotientDigitTable}(\hat{R}).$$

The penalty is that the table must be computed each time the modulus is changed. In, for example, modular exponentiation the modulus is fixed for many intermediate modular multiplications and, furthermore, in RSA applications the modulus may very well be fixed for several modular exponentiations. In stand-alone applications like division the "modulus" (in division this operand is usually denoted the divisor) cannot be expected to be fixed for more than a single operation. So, the application of the quotient determination operation must be taken into account when the operation is implemented. Obviously, the time for precomputation of the quotient digit table is of minor importance in the applications considered in this thesis. Hence, by utilising the precomputation technique a *quotient determination operation that is more efficient than those known from the literature on division methods can be expected.* In the next subsection a condition that expresses a *necessary number of digits*, i.e. a minimal required number of digits of R and m, will be derived. Further, a condition that expresses a *sufficient number of digits* will be derived.

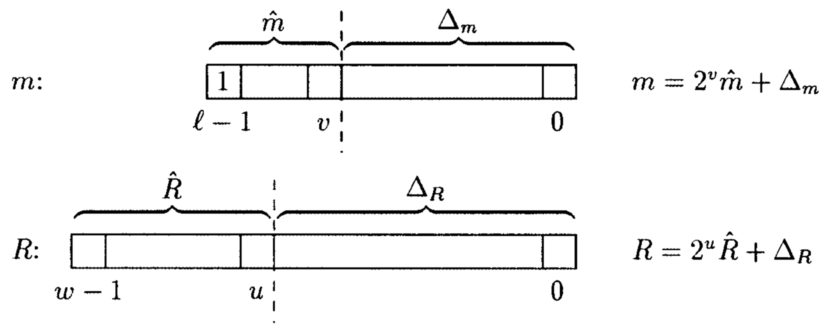


Figure 3.10: Truncation of operands used in the quotient determination

In Figure 3.10 the notation used in the descriptions of the quotient determination methods is summarised. It is assumed that both m and R are expressed in radix 2 number systems. A natural choice of representation for m is the (non-redundant) binary representation, where the digit set is $\{0, 1\}$. It is assumed that m is represented in ℓ digits such that the most significant digit is equal to 1. Hence, the range of m is $[2^{\ell-1}; 2^{\ell}[$[5]. For R a redundant representation is more common: For example the carry save representation, where the digit set is $\{0, 1, 2\}$ or the borrow save representation, where the digit set is $\{\bar{1}, 0, 1\}$. The required number of digits for representing R is denoted w and is determined from the range of R. The most significant part of R and of m used in the quotient determination is denoted $\hat{R}$ respectively $\hat{m}$. The u digits truncated from R is denoted Δ_R and the v digits truncated from m is denoted Δ_m. The range of a truncation error Δ depends on the number of truncated digits and on the representation of the operand under consideration. Table 3.1 lists the minimal value $\Delta^{\min}$ and the maximal value $\Delta^{\max}$ for an x digit truncation error in the above mentioned representations. The range of $\hat{R}$ follows from the expression $R = 2^u\hat{R} + \Delta_R$ and from the condition $|R| \leq \delta\alpha m$. By Theorem 3.8–1 the value of $\delta\alpha$ is seen to be less than or equal to $q^{\max} + \alpha$. Hence, $\hat{R}$ must be bounded by,

$$\left\lceil \frac{-(q^{\max} + \alpha)m - \Delta_R^{\max}}{2^u} \right\rceil \leq \hat{R} \leq \left\lfloor \frac{(q^{\max} + \alpha)m - \Delta_R^{\min}}{2^u} \right\rfloor$$

Equation (3.28) limits the number of values of $\hat{R}$ and, hence, expresses the maximal number of table entries of the form QuotientDigitTable($\hat{R}$) required in the table look-up methods.

[5]Note that ℓ a fixed parameter in a dedicated hardware implementation. This does not imply maybe that the implementation is limited to perform modular multiplication with a modulus of exactly ℓ digits: It can perform the computation with all possible values of modulus bounded by $0 < m < 2^{\ell}$. The trick is to perform a "normalisation" by left-shifting modulus m and multiplicand b until the most significant modulus digit is positioned at digit position $\ell - 1$. Hereafter, the modular multiplication is performed using the shifted operands. Let the number of left-shifts be denoted by r. Then the computation corresponds to a computation of $x \equiv (a \cdot b2^r) \bmod (m2^r)$. According to Equation (3.24) the desired result $(a \cdot b) \bmod m$ is equal to x div 2^r and, therefore, it can be achieved by r right-shifts of x.

Representation	$\Delta^{\min}$	$\Delta^{\max}$
Binary	0	$2^x - 1$
Carry save	0	$2(2^x - 1)$
Borrow save	$-(2^x - 1)$	$2^x - 1$

Table 3.1: Range of an x digit truncation error for some radix 2 representations.

3.10.2 Analysis of Selection Intervals

The aim of a quotient determination method is to determine some "valid" quotient digit value for a given value of $\hat{m}$ and some value of $\hat{R}$ in the range (3.28). A valid quotient digit value j is obeying $|R - jm| \leq \alpha m$. The *selection interval* for quotient digit j is denoted I_j and is defined as the interval of $\hat{R}$ where j is a valid quotient digit value. So, if a value of $\hat{R}$ belongs to I_j then the table entry of this $\hat{R}$ value can be initialised with the quotient digit value j. From $(j - \alpha)m \leq R \leq (j + \alpha)m$ it follows that I_j can be determined from,

$$\begin{array}{ccccc} (j-\alpha)(2^v\hat{m} + \Delta_m) & \leq & 2^u\hat{R} + \Delta_R & \leq & (j+\alpha)(2^v\hat{m} + \Delta_m) \\ \frac{(j-\alpha)(2^v\hat{m}+\Delta_m)-\Delta_R}{2^u} & \leq & \hat{R} & \leq & \frac{(j+\alpha)(2^v\hat{m}+\Delta_m)-\Delta_R}{2^u} \end{array} \tag{3.28}$$

Since the explicit values of the truncation errors are unknown during the quotient determination process the worst case truncation errors must be considered in order to ensure that j is a valid quotient digit value. Consequently, the cardinality of the selection intervals decreases for increasing worst case truncation errors. If the lower bound and the upper bound of selection interval I_j is denoted $I_j^{\min}$, respectively $I_j^{\max}$, the selection intervals are expressed as,

$$I_j = \{\lceil I_j^{\min} \rceil, \lceil I_j^{\min} \rceil + 1, \ldots, \lceil I_j^{\max} \rceil$$

Equation (3.28) gives the value of $I_j^{\min}$ when $j = -q^{\max}$ and the value of $I_j^{\min}$ when $j = q^{\max}$. The remaining bounds of the selection intervals are derived by insertion of the worst case truncation error into (3.29),

$$I_j^{\min} = \begin{cases} \dfrac{(j-\alpha)(2^v\hat{m} + \Delta_m^{\max}) - \Delta_R^{\min}}{2^u}, & \text{if } (j-\alpha) \geq 0 \\[2ex] \dfrac{(j-\alpha)(2^v\hat{m} + \Delta_m^{\min}) - \Delta_R^{\min}}{2^u}, & \text{if } (j-\alpha) \leq 0 \end{cases} \tag{3.29}$$

$$I_j^{\max} = \begin{cases} \dfrac{(j+\alpha)(2^v\hat{m} + \Delta_m^{\min}) - \Delta_R^{\max}}{2^u}, & \text{if } (j+\alpha) \geq 0 \\ \dfrac{(j+\alpha)(2^v\hat{m} + \Delta_m^{\max}) - \Delta_R^{\max}}{2^u}, & \text{if } (j+\alpha) \leq 0 \end{cases} \tag{3.30}$$

To ensure that all possible values of $\hat{R}$ indeed are represented in some selection interval it must be demanded that the distance between two neighbouring intervals, say I_{j-1} and I_j, is less than or equal to one, i.e.,

$$\lceil I_j^{\min} \rceil - \lfloor I_{j-1}^{\max} \rfloor \leq 1 \text{ for all } j \in \{-q^{\max}+1, \ldots, q^{\max}\}. \tag{3.31}$$

From this demand a condition that expresses a sufficient precision of the quotient determination procedure can be derived. The demand (3.32) is fulfilled if $I^{\min} \leq I_{j-1}^{\max}$. First assume that $(j-\alpha) \geq 0$. Then, by using the expression (3.30) for $I_j^{\min}$ and the expression (3.31) for $I_{j-1}^{\max}$ the following *sufficient condition* is achieved,

$$(j-\alpha)(\Delta_m^{\max} - \Delta_m^{\min}) + (\Delta_R^{\max} - \Delta_R^{\min}) \leq (2\alpha - 1)(2^v\hat{m} + \Delta_m^{\min}). \tag{3.32}$$

The worst case restriction on the truncation errors is when j is the largest possible, i.e. $j = q^{\max}$, and when $\hat{m}$ is the smallest possible, i.e. $2^v\hat{m} = 2^{\ell-1}$. Exactly the same worst case restriction on the truncation errors is derived under the assumption $(j+\alpha) \leq 0$. For the remaining values of j, where $-\alpha < j < \alpha$, the restriction on the truncation errors is weaker than (3.33). Observe that Theorem 3.8–1 gives $q^{\max} \geq (\delta - 1)\alpha$ and since $\delta \geq 2^k$ this results in $q^{\max} \geq \alpha$ for all $k \geq 1$. So, there will always exist a value of j such that $(j-\alpha) \geq 0$ and, therefore, the weaker restriction is of no importance.

In some cases Equation (3.33) is too restrictive. When $\lceil I_j^{\min} \rceil = I_j^{\min}$ and $\lfloor I_{j-1}^{\max} \rfloor = I_{j-1}^{\max}$ it is sufficient and necessary to require $I_j^{\min} \leq I_{j-1}^{\max} + 1$. Indeed, for all cases this requirement is necessary. Hence, the following *necessary condition* is derived,

$$(j-\alpha)(\Delta_m^{\max} - \Delta_m^{\min}) + (\Delta_R^{\max} - \Delta_R^{\min}) \leq 2^u + (2\alpha - 1)(2^v\hat{m} + \Delta_m^{\min}).$$

For this condition the worst case restriction on the truncation errors also appears when $j = q^{\max}$ and when $2^v\hat{m} = 2^{\ell-1}$. In conclusion the following theorem can be stated,

Theorem 3.10–1 (Necessary precision and sufficient precision)
Giuen $q^{\max}$, the parameter α and the worst case truncation errors $\Delta_m^{\max}, \Delta_m^{\min}$ and *$\Delta_R^{\max}$, $\Delta_R^{\min}$ in some radix 2 representation. Then, a necessary condition expressing the maximal allowable number of truncated digits u of R and v of m is,*

$$(q^{\max} - \alpha)(\Delta_m^{\max} - \Delta_m^{\min}) + (\Delta_R^{\max} - \Delta_R^{\min}) \leq 2^u + (2\alpha - 1)(2^v \hat{m} + \Delta_m^{\min}). \tag{3.33}$$

Furthermore, a sufficient condition is,

$$(q^{\max} - \alpha)(\Delta_m^{\max} - \Delta_m^{\min}) + (\Delta_R^{\max} - \Delta_R^{\min}) \leq (2\alpha - 1)(2^v \hat{m} + \Delta_m^{\min}). \tag{3.34}$$

In the next subsection some implications of this theorem will be discussed. It is be assumed that modulus m belongs to the interval $[2^{\ell-1}; 2^{\ell}[$ and that m is binary represented. Since the only difference between the quotient determination methods in SRT division and left-to- right modular multiplication is the value of parameter α and of the maximal absolute quotient digit value $q^{\max}$, Theorem 3.10–1 can be applied for the analysis of SRT division methods as well. Indeed, a comparison of a modular multiplication method and an SRT division method with the fixed value of $q^{\max}$ will be done. According to Table 3.1 the term $(\Delta_R^{\max} - \Delta_R^{\min})$ evaluates to $2(2^u - 1)$ when the representation of R is either borrow save or carry save. Therefore, Theorem 3.10–1 reveals the same results for both these representations of R.

3.10.3 Borrow Save and Carry Save Representation

Consider the case where the intermediate operand R is redundant represented in borrow save or carry save representation. Then, by insertion of the minimal and maximal truncation errors from Table 3.1 the necessary condition (3.34) reduces to,

$$\begin{aligned} (q^{\max} - \alpha)(2^v - 1) + 2(2^u - 1) &\leq 2^u + (2\alpha - 1)2^v \hat{m}. \\ (q^{\max} - \alpha)(2^v - 1) + 2^u &\leq 2 + (2\alpha - 1)2^v \hat{m} \end{aligned} \tag{3.35}$$

A bound on largest possible number u of truncated digits of R is obtained by setting v to zero and by inserting the minimal value $2^{\ell-1}$ of modulus. Hence, the necessary condition becomes,

$$2^u \leq 2 + (2\alpha - 1)2^{\ell-1}$$

In [Kor93b] Kornerup maximises the value of $\lfloor \log_2(\alpha - \frac{1}{2}) \rfloor$ in order to minimise the number of required digits of $\hat{R}$ in the quotient determination. This is seen to be in good correspondence with this bound on u. In Kornerup's modular multiplication method α is restricted to the range $[\frac{1}{2}; \frac{2}{3}[$. As explained in Section 3.9 it is possible to obtain the constant value $\alpha = \frac{2}{3}\frac{48}{49}$, independent on the radix of the method, by scaling the modulus. So, the necessary condition is fulfilled if $u \leq \ell - 3$. In [Atk68] Atkins analyses the required precision in SRT division methods. Atkins restricts the analysis to radix 2^k methods with $\alpha = \frac{2}{3}$. Further, both Kornerup's modular multiplication method and Atkins' SRT division method use $q^{\max} = \frac{2}{3}(2^k - 1)$ and both methods are limited to even values of k. Obviously, also Atkins' SRT division method must fulfil $u \leq \ell - 3$. By insertion of the maximal and minimal truncation errors from the Table 3.1, the sufficient condition reduces to,

$$\begin{aligned}(q^{\max} - \alpha)(2^v - 1) + 2(2^u - 1) &\leq (2\alpha - 1)2^v\hat{m} \\ (q^{\max} - \alpha)(2^v - 1) + 2^{u+1} &\leq 2 + (2\alpha - 1)2^v\hat{m} \end{aligned} \tag{3.36}$$

When $v = 0$ and $m = 2^{\ell-1}$ the sufficient condition becomes,

$$2^{u+1} \leq 2 + (2\alpha - 1)2^{\ell-1}.$$

Hence, both for Kornerup's method with $\alpha = \frac{2}{3}\frac{48}{49}$ and for Atkins' method with $\alpha = \frac{2}{3}$, Theorem 3.10–1 gives that $u = \ell - 4$ is a sufficient precision of $\hat{R}$ when no truncation of m is done. Since the above analysis reveals the necessary precision $u \leq \ell - 3$, it might be possible that $u = \ell - 3$ is sufficient. Indeed, Atkins finds that $u = \ell - 3$ is sufficient. However, the following calculation of the selection intervals for Atkins' radix 4 method shows that if $u = \ell - 3$ then *it is not all possible values of $\hat{R}$ that are represented in the selection intervals*. Using the value $\alpha = \frac{2}{3}$ and $m = 2^{\ell-1}$ the bounds for the selection intervals $I_{\bar{2}}, I_{\bar{1}}, \ldots, I_2$ are computed by Equations (3.30) and (3.31) and by Equation (3.28). The result, when $\hat{R}$ is borrow save represented, is:

j	$\lceil I_j^{\min} \rceil$	$\lfloor I_j^{\max} \rfloor$
$\bar{2}$	-11	-7
$\bar{1}$	-5	-3
0	-1	1
1	3	5
2	7	11

It is seen that the values $-6, -2, 2$ and 6 of $\hat{R}$ are missing in the selection intervals and, therefore, there must be an error in Atkins' analysis of the sufficient precision of $\hat{R}$. Further observe that according to expressions (3.30) and (3.31) of the bounds for the selection interval, the best condition for improving the sufficient precision of $\hat{R}$ is when a a full precision of m, i.e. $v = 0$ is used. If $v = 0$ then the bounds of the selection interval I_j for a given j are independent on the radix. Hence, *the precision $u = \ell - 4$ is necessary and sufficient for Atkins' SRT division method and Kornerup's modular multiplication method.* The reason that the conclusion also holds for the modular multiplication method is, that α is smaller than for the SRT division method. So, for a given j the selection interval of the modular multiplication method will be smaller than or equal to the selection interval for the division method and, therefore the restrictions on the modular multiplication method will be stronger than or equal to the restrictions on the SRT division method.

In [Atk70] Atkins describes the arithmetic unit of the computer ILLIAC III developed at University of Illinois. The unit uses a radix 4 SRT division method with a borrow save representation of the intermediate operand R a binary representation of $m, \alpha = \frac{2}{3}$ and quotient digit set $\{\bar{2}, \bar{1}, 0, 1, 2\}$. In this implementation the precision corresponds to choosing $u = \ell - 4$ and $v = \ell - 4$. So, fortunately, the erroneous result was not utilised in the implementation of the arithmetic unit of ILLIAC III. (This might, of course, also be the reason, that the error has not been corrected). If the precision given by $u = \ell - 4$ and $v = \ell - 4$ are chosen for $\hat{R}$ and $\hat{m}$ the selection intervals for the minimal value of $\hat{m}$ i.e. $2^v\hat{m} = 2^{\ell-1}$ are:

j	$\lceil I_j^{\min} \rceil$	$\lfloor I_j^{\max} \rfloor$
$\bar{2}$	-22	-13
$\bar{1}$	-12	-4
0	-4	4
1	4	12
2	13	22

Since no values of $\hat{R}$ are missing, this precision of $\hat{R}$ and $\hat{m}$ is sufficient for radix 4 and $\alpha = \frac{2}{3}$. A similar computation of the selection intervals when $\alpha = \frac{2}{3}\frac{48}{49}$ shows that $v = \ell - 4$ is insufficient. However, $v = \ell - 5$ turns out to be sufficient. So, *for Atkins' SRT division method and Kornerup's modular multiplication method the only difference in the complexity of the quotient determination is in the required precision of $\hat{m}$.* Using the necessary and sufficient precision $u = \ell - 4$ of $\hat{R}$ in Equations (3.36) and (3.37) the following conditions for the precision of $\hat{m}$ is obtained when $q^{\max} = \frac{2}{3}(2^k - 1)$,

α	NecessaryPrecision	SufficientPrecision
$\frac{2}{3}$	$v \leq \begin{cases} \ell - 2 - k & \text{for } k = 2 \\ \ell - 3 - k & \text{for } k > 2 \end{cases}$	$v \leq \begin{cases} \ell - 3 - k & \text{for } k = 2 \\ \ell - 4 - k & \text{for } k > 2 \end{cases}$
$\frac{2}{3} \cdot \frac{48}{49}$	$v \leq \begin{cases} \ell - 2 - k & \text{for } k = 2 \\ \ell - 3 - k & \text{for } k > 2 \end{cases}$	$v \leq \begin{cases} \ell - 4 - k & \text{for } k = 2 \\ \ell - 5 - k & \text{for } k > 2 \end{cases}$

So, the necessary precision obtained by Theorem 3.10–1 for the two choices of parameter α is identical. The sufficient precision differs by a single digit. As indicated by the radix 4 case, these bounds on the sufficient precision of $\hat{m}$ may very well be improved by a more detailed analysis of the selection intervals. However, in the application area of this thesis the precision of $\hat{m}$ is of minor importance: Assume the table look-up method is used for determining the quotient digit and assume the computation of the table is performed after each change of modulus. Then, the number of table entries only depends on the range of $\hat{R}$. It does not depend on the range of $\hat{m}$. The only implication of the precision of $\hat{m}$ is in the computation of the table. It is sufficient to use the above derived precision of m in the computation of the table. Indeed, it might be a waste of computation time if a higher precision is used.

Table Size and Representation of Operand R

The number of table entries is determined by the number of possible values of $\hat{R}$. The range of $\hat{R}$ is given by Equation (3.28). Observe that the bounds of $\hat{R}$ depends on the representation of $\hat{R}$. First, assume that $\hat{R}$ is borrow save represented. By insertion of the expressions for the worst case truncation errors $\Delta_R^{\min}$ and $\Delta_R^{\max}$ it is seen that the bounds are symmetric around zero, i.e.,

$$|\hat{R}| \leq \left\lfloor \tfrac{(q^{\max}+\alpha)m+(2^u-1)}{2^u} \right\rfloor \leq \left\lfloor \tfrac{(q^{\max}+\alpha)m}{2^u} \right\rfloor \tag{3.37}$$

Hence, the number of possible values of $\hat{R}$ is bounded by,

$$2\lfloor (q^{\max} + \alpha)m2^{-u} \rfloor + 3 \leq 2\lfloor (q^{\max} + \alpha)2^{\ell-u} \rfloor + 3.$$

The largest number is when $m \in [2^{\ell-1}; 2^{\ell}[$ is maximal. Now, because $\hat{R}$ is redundant represented there exist values of $\hat{R}$ that have more than a single encoding. Hence, if the redundant encoding of $\hat{R}$ is used as an address in the table, the number of entries must be larger than the number of possible values of $\hat{R}$. For example, suppose the range of $\hat{R}$ is $\{-31, -30, \ldots, 31\}$. Then, the number of possible values is 63, and the number of digits required in the borrow save representation of $\hat{R}$ is 5. There are $3^5 = 243$ different encodings of $\hat{R}$ and each of these encodings represents some value in the range of $\hat{R}$. Hence, in this example, the number of encodings is nearly 4 times the number of values. In methods with an even larger range of $\hat{R}$, e.g. Atkins' radix 4 division method or Kornerup's radix 4 modular multiplication method, this effect is even more dramatic. Therefore, it is common to *convert the redundant represented $\hat{R}$ into a non-redundant representation* before the table look-up is performed. In [Kor93b] Kornerup suggests to convert $\hat{R}$ into a signed-magnitude representation, i.e. into the (non-redundant) binary representation of the absolute value of $\hat{R}$ and into the sign of $\hat{R}$. Then, by utilising a symmetry property of the table the number of table entries can be reduced to the number of absolute values of $\hat{R}$,

$$\lfloor (q^{\max} + \alpha) m 2^{-u} \rfloor + 2 \quad \leq \quad \lfloor (q^{\max} + \alpha) 2^{\ell - u} \rfloor + 2 \tag{3.38}$$

The symmetry property of the table, when $\hat{R}$ is borrow save represented, can be expressed by,

$$\text{QuotientDigitTable}(\hat{R}) = \begin{cases} \text{QuotientDigitTable}(\hat{R}) & \text{if } \hat{R} \geq 0 \\ -\text{QuotientDigitTable}(|\hat{R}|) & \text{if } \hat{R} < 0 \end{cases}$$

The validity of this property follows from Equations (3.30) and (3.31) where it is seen that the selection interval bounds obey $I_j^{\min} = -I_{\bar{j}}^{\max}$ and, equivalently, $I_j^{\max} = -I_{\bar{j}}^{\min}$. In table (3.40) the maximal required quotient digit table size for Kornerup's method is shown. The table sizes are calculated with the value $\frac{2}{3}\frac{48}{49}$ of α and $\frac{2}{3}(2^k - 1)$ of $q^{\max}$. The precision $u = \ell - 4$ is used for $\hat{R}$. In addition, the maximal required number of digits of $\hat{R}$ is showen,

k	Table Size	$\hat{R}$ Digits
2	44	6
4	172	8
6	684	10
8	2732	12

(3.39)

In general, for this method, the maximal table size is $\frac{2}{3}(2^k - 1)16 + 12$ and, hence, the maximal required number of digits of $\hat{R}$ is $\lfloor \log_2(\frac{2}{3}(2^k - 1)16 + 12)\rfloor = k + 4$. Also the maximal required number of digits w in the borrow save representation of R can be calculated: $w = u + k + 4$ which is equal to $\ell + k$ digits. The maximal table size gives a bound of the required storage for the quotient determination method, the maximal number of $\hat{R}$ digits gives a bound on the time for the conversion into non-redundant represent ation, and the maximal number of R digits gives a bound of the size of the register for holding R. Furthermore, in a hardware implementation the width of the circuitry is bounded by w.

If R is carry save represented, the number of possible values of R is equal to the number of values in the borrow save representation: By insertion of the worst case truncation errors into Equation (3.28),

$$\left\lfloor \frac{(q^{\max} + \alpha)m + 2(2^u - 1)}{2^u} \right\rfloor \leq \ \hat{R} \ \leq \left\lfloor \frac{(q^{\max} + \alpha)m}{2^u} \right\rfloor$$

Since $\lfloor (q^{\max} + \alpha)m2^{-u}\rfloor - 2$ is less than or equal to the left side of this expression, the number of possible values of $\hat{R}$ is bounded by,

$$2\lfloor (q^{\max} + \alpha)m2^{-u} - 3 \leq 2\lfloor (q^{\max} + \alpha)2^{\ell - u}\rfloor - 3.$$

Because of the symmetry property of the quotient digit table it might seem that the borrow save representation of R is superior to the carry save representation. However, there is a similar symmetry of the table when R is carry save represented. From Equations (3.30) and (3.31) it follows, that the selection interval bounds obey $I_j^{\min} = -I_{\bar{j}}^{\max} - (2 - 2^{1-u})$ and, equivalently, $I_j^{\max} = -I_{\bar{j}}^{\min} - (2 - 2^{1-u})$. So, the selection intervals are symmetric around the value $-(1 - 2^{-u})$. Unless the selection interval bounds are exact integers, i.e. $\lfloor I_{\bar{j}}^{\max} - (2 - 2^{1-u})\rfloor < \lfloor I_{\bar{j}}^{\min}\rfloor - 2$, the quotient digit table will be symmetric around -1 when 2^{-u} is small. Indeed, for the operand sizes considered in this thesis 2^{-u} is vanishing. So, the following symmetry of the quotient digit table can be used for reducing the table size to exactly the same size as obtained by the borrow save representation,

$$\text{QuotientDigitTable}(\hat{R}) = \begin{cases} \text{QuotientDigitTable}(\hat{R}) & \text{if } \hat{R} \geq -1 \\ -\text{QuotientDigitTable}(|\hat{R}| - 2) & \text{if } \hat{R} < -1 \end{cases}$$

For example, this symmetry applies both for Atkins' division method and Kornerup's modular multiplication method. Hence, for many cases *it does not* *matter whether the operand R is represented in borrow save representation or in carry save representation.*

Considering the Value of Modulus

Until now the only assumption about modulus m is that m is a binary represented ℓ digit integer, i.e. the value of m is known to be in the range $[2^{\ell-1}; 2^{\ell}[$. All the previous analysis of the required precision of $\hat{R}$, the maximal number of $\hat{R}$ digits to be converted into non-redundant representation prior to each table look-up, and the maximal number of entries in the quotient digit table has been based on a worst case assumption about the value of m. This worst case value is, dependent on the specific analysis, either the maximal value of m or the minimal value of m. The only moment, where the actual value of m is used in the quotient determination process, is when the quotient digit table is computed. It might, however, be advantageous to further utilise the knowledge of the actual value of modulus and obtain a reduction in the resource requirements.

First, consider the precision of $\hat{R}$. The largest precision is required when the value of m is minimal. For Kornerup's method it was shown that the precision $u = \ell - 4$ is necessary and sufficient. How large should m be to achieve a sufficient precision of $u = \ell - 3$? Using Equation (3.37) the sufficient condition $m \geq \frac{1}{2(2\alpha-1)} 2^{\ell-1}$ is obtained. If $\alpha = \frac{2}{3}\frac{48}{49}$ the condition becomes $m \geq \frac{49}{50} 2^{\ell-1}$. So, for values of m satisfying this condition the precision of $\hat{R}$ can be reduced to $u = \ell - 3$. According to (3.39) this reduces the maximal table size to $\frac{2}{3}(2^k - 1)8 + 6$ which, compared to the results in (3.40), is a reduction of 50 percents:

k	Table Size	$\hat{R}$ Digits
2	22	5
4	86	7
6	342	9
8	1366	11

(3.40)

The maximal number of $\hat{R}$ digits is reduced by one. So, the maximal number of $\hat{R}$ digits becomes $k + 3$. In Atkins' method, where $\alpha = \frac{2}{3}$, the bound corresponding on the modulus is $m \geq \frac{3}{2} 2^{\ell-1}$.

Next, consider the number of $\hat{R}$ digits. This number is determined by the range of $\hat{R}$ and the number increases with increasing values of m. In the worst case, for Kornerup's method, the number of $\hat{R}$ digits is $k+4$. Suppose $\hat{R}$ is borrow save represented. How small should m be to achieve a range of $\hat{R}$ such that the number of digits is $k+3$? Using Equation (3.38) the sufficient condition $m \leq \frac{3}{2}2^{\ell-1}$ is obtained. So, for values of m satisfying this condition the table size is less than or equal to 2^{k+3}. This corresponds to approximately 75 percents of the results in (3.40). In Atkins' method the corresponding bound is $m < \frac{3}{2}2^{\ell-1}$.

For Atkins' SRT division method it is seen that the quotient determination can be reduced to an inspection of the $k+3$ digits of $\hat{R}$ and, hence, that the maximal table size can be reduced to 2^{k+3}. Note that the positions of these $k+3$ digits vary with the value of modulus m: If $m \geq \frac{3}{2}2^{\ell-1}$ the digits of $\hat{R}$ correspond to the digits of R from position $u = \ell - 3$ up to position $\ell + k - 1$, and if $m < \frac{3}{2}2^{\ell-1}$ they correspond to the digits of R from position $u = \ell - 4$ up to position $\ell + k - 2$. Hence, to utilise the knowledge of the value of m to reduce the resource requirement, an implementation of the quotient determination must be reconfigurable. For Kornerup's modular multiplication method the above optimisation considerations divides the value of m into three ranges: $[2^{\ell-1}; \frac{3}{2}2^{\ell-1}],]\frac{3}{2}2^{\ell-1}; \frac{49}{30}2^{\ell-1}[$ and $[\frac{49}{30}2^{\ell-1}; 2^{\ell}[$. The range giving the smallest table size is $[\frac{49}{30}2^{\ell-1}; 2^{\ell}[$. The middle range sets the maximal resource requirement and, unfortunately, this range is non-empty. There is, however, a way to avoid values of modulus m in the middle range: In the next subsection it will be discussed how an adjustment of the range of modulus m can provide a more efficient quotient determination.

3.10.4 Adjusting the Range of Modulus

As described above there are some ranges of modulus m that gives an improvement in the maximal table size and in the maximal number of digits of $\hat{R}$. Consider Kornerup's modular multiplication method with $\alpha = \frac{2}{3}\frac{48}{49}$. As discussed, values of m in the range $]\frac{3}{2}2^{\ell-1}; \frac{49}{30}2^{\ell-1}[$ are inconvenient. Now, suppose the range of m is checked each time an application is initialised with a new value of m. Then, an inspection of the two most significant digits of m can decide the relationship between m and the ranges,

$$[2^{\ell-1}; \tfrac{3}{2}2^{\ell-1}[\text{ and } [\tfrac{3}{2}2^{\ell-1}; 2^{\ell}[$$

The idea is now to *adjust the range by scaling the modulus* with some scaling constant c before the modular multiplication is computed. The scaling constant is determined from the present range of m. Suppose $c = 4$ is used when m is in the first range, and $c = 3$ is used when m is in the last range. The effect of this transformation of the ranges is shown in the following table:

Range of m	$[2^{\ell-1}; \frac{3}{2}2^{\ell-1}[$	$[\frac{3}{2}2^{\ell-1}; 2^{\ell}[$
c	4	3
Range of $c \cdot m$	$[2^{\ell+1}; \frac{3}{2}2^{\ell+1}[$	$[\frac{9}{8}2^{\ell+1}; 2^{\ell+1}[$

Hence, if $\ell' = \ell + 2$ denotes the number of digits of the scaled modulus $m' = c \cdot m$, it is seen that all values of m' are bounded to the range $[2^{\ell'-1}; \frac{3}{2}2^{\ell'-1}[$. Consequently, if m' is replacing m during all the subsequent computations of intermediate modular products a more efficient quotient determination is achieved for Kornerup's method. As discussed in the previous subsection the range $[\frac{49}{30}2^{\ell-1}; 2^{\ell}[$ of modulus leads to the smallest quotient digit table. It is also possible, through a similar scaling of m, to obtain this range. Then, an inspection of the four most significant digits of m is needed. The following table shows how the scaling constant could be chosen to achieve $m' \in [\frac{49}{50}2^{\ell'-1}; 2^{\ell'}[$ where $\ell' = \ell + 3$:

$\frac{m}{2^{\ell-1}}$	$[\frac{8}{8}; \frac{9}{8}[$	$[\frac{9}{8}; \frac{10}{8}[$	$[\frac{10}{8}; \frac{11}{8}[$	$[\frac{11}{8}; \frac{12}{8}[$	$[\frac{12}{8}; \frac{13}{8}[$	$[\frac{13}{8}; \frac{14}{8}[$	$[\frac{14}{8}; \frac{15}{8}[$	$[\frac{15}{8}; \frac{16}{8}[$
c	14	12	11	10	9	9	8	7, 8

(3.41)

As shown, there might be more possible choices of integer values of c that can be used in a subrange. In general, the smallest value of c for a given range leads to the smallest quotient digit table.

The technique of adjusting the range of modulus is known from division methods. Some of the early articles describing this technique are [Svo63] by Svoboda and [Kri70] by Krishnamurthy. By scaling both the dividend and the divisor with the same scaling constant, it is assured that the resulting quotient is unaffected by the adjustment. Ercegovac and Lang [Erc83, EL85, EL90] have improved the quotient determination in SRT division methods by means of the range adjustment technique. In [EL90] a radix 4 SRT division method with quotient digit set $\{\bar{2}, \bar{1}, 0, 1, 2\}$ and $\alpha = \frac{2}{3}$ is developed. So, the method has the same parameter values as Atkins' division method. In this article, the aim of adjusting the range is to obtain *a quotient determination operation that is independent of the divisor*. This implies that the selection intervals are constant for all divisor values and, hence, the computation of the quotient digit table can be done once for all at the time of implementation of the division method. Ercegovac and Lang transform the divisor into a small range around a power of 2. The resulting range can be expressed as $[\frac{63}{64}2^{\ell'}, \frac{9}{8}2^{\ell'}[$. So, the position of the most significant digit of the scaled divisor can be either $\ell' - 1$ or ℓ'. The range of the unscaled divisor is subdivided into eight subranges as in (3.42). The parameter ℓ' is equal to $\ell + 3$ too. However, the scaling constants differ: The values are 16, 14, 13, 12, 11, 10, 9 and 9, where 16 is used in the first subrange in (3.42) and 9 is used in the last subrange. Ercegovac and Lang show that it is sufficient to inspect the 6 most significant digits of the partial remainder R in the quotient determination operation. In fact, it is sufficient to inspect 5 digits: A calculation of the selection intervals, given by Equations (3.30) and (3.31), shows that $u = \ell' - 3$ is sufficient. Furthermore, from (3.38) it follows that $|\hat{R}| \leq 25$. So, it is sufficient to inspect the 5 most significant digits of the partial remainder. The quotient digit table has 26 entries when the symmetry property is utilised. Ercegovac and Lang emphasise that it is important to use an efficient computation of the scaled dividend and divisor: In general, the values of the dividend and the divisor cannot be expected to be fixed for more than a single division operation. Hence, the range adjustment must be performed prior to each application of the division operation. In SRT division methods with radices higher than 4 it is not possible to obtain a quotient determination that is independent on the divisor without performing a more complex range adjustment. A calculation of the selection intervals for a radix 2^k method with $\alpha = \frac{2}{3}$, $q^{\max} = \frac{2}{3}(2^k - 1)$ and precision $u = \ell' - 3$ shows that if the range of the adjusted divisor is $[(1 - 2^{-(k+2)})2^{\ell'};\ (1 + 2^{-(k+4)})2^{\ell'}[$, or $[(1 - 2^{-(k+4)})2^{\ell'};\ (1 + 2^{-(k+2)})2^{\ell'}[$, then the quotient determination is independent on the divisor.

Although the division operation and modular reduction operation is closely related, there is a difference in the way the range adjustment technique is applied. Denote the dividend by z and the divisor by m. In division both the dividend and the divisor is scaled by the same constant c. This is done to assure the correctness of the resulting quotient value, (cz) div $(cm) = z$ div m. However, the resulting remainder r' is not equal to the correct remainder r. Indeed, if $r = z \bmod m$ then $r' = (cz) \bmod (cm) = cr$. So, to obtain the correct remainder a division by c is required. In general, r' and r do not belong to the same residue class. In modular reduction only the modulus is scaled. This implies that the result $r' \equiv z \bmod (cm)$ is belonging to the same residue class as $r \equiv z \bmod m$, i.e. $r' \equiv_m r$. So, the value r' can replace the value of r when r is an intermediate operand in the computation of a modular operation. Only the final result of the computation needs a complete modular reduction modulo m. Furthermore, in the applications in this thesis, the computing time for the initial scaling of the modulus is not as important as in the general division operation. So, even though the range adjustment might be relatively complex it is advantageous to utilise the range adjustment technique in high-radix modular multiplication methods.

The final correction is an additional cost compared to a computation with unscaled moduli. However, when the number of consecutive modular multiplications is large the time for performing the initial adjustment and the final correction is negligible in comparison with the total computing time. Indeed, since the quotient determination complexity is reduced, both a reduction in the total computation time and a reduction in the required quotient digit table size may be expected. However, because the range of the intermediate results increases by a factor of c the number of recursion cycles of the modular multiplication method increases. In a dedicated hardware implementation the required circuitry also increases with the range of the operand. So, a more detailed analysis, taking the specific parameter setting of $\alpha, q^{\max}, \ell, k$, etc. into account, should be completed before any qualified statements on costs and benefits of the range adjustment technique is postulated.

In [Wal91b] Walter utilises the range adjustment technique to obtain a simple quotient determination operation in modular multiplication. Walter suggests to adjust the modulus range into a range of the form $[(1-2^{-x})2^{\ell'}; 2^{\ell'}[$ where x is a sufficiently large integer. Indeed, Walter obtains a very simple quotient determination where *there is no need for a quotient digit table*. The idea is to obtain the relation,

$$\hat{R} = \text{QuotientDigitTable}(\hat{R}). \tag{3.42}$$

Apart from saving memory for holding the quotient digit table and saving time for a computation of the table it might also be possible to avoid the conversion of $\hat{R}$ into non-redundant representation. This conversion is performed in order to reduce the required number of table entries. So, if a redundant represented value of the quotient digit may be accepted in the computation of multiple qm, the conversion can be avoided. In general, it requires a relatively large value of the parameter α to obtain relation (3.43). By insertion of the expression $R = 2^u\hat{R} + \Delta_R$ into the range restriction $|R - qm| \leq \alpha$

of the quotient determination operation and using $q = \hat{R}$, the restriction is written as,

$$|\hat{R}(2^u - m) + \Delta_R| \leq \alpha m. \tag{3.43}$$

If m is known to be in the range $[(1-2^{-x})2^{\ell'}; 2^{\ell'}[$ and u equals ℓ' then the value of the term $\hat{R}(2^u - m)$ is in the range $[\hat{R}; 2^{\ell'-x}\hat{R}]$. Therefore, by choosing a sufficiently large value of x the contribution from this term can be made negligible in comparison to the contribution from the term Δ_R. (Walter restricts the range of m to $[(1-2^{-x})2^{\ell'}; 2^{\ell'}[$. When R is carry save represented this ensures that $R - qm$ is non-negative. If negative values of $R - qm$ is allowed it would be advantageous to use the range $[(1-2^{-x})2^{\ell'}; (1+2^{-x})2^{\ell'}[$ of m. This would reduce the required value of the scaling constant and, therefore, reduce the required value of the scaling constant and, therefore, reduce the computationally effort required value of the scaling constant and, therefore, reduce the computationally effort required in the scaling of modulus and in the correction of the final result.) A lower bound on α is then seen to be determined from $|\Delta| \leq \alpha m$. Using the worst case value $m = (1-2^{-x})2^{\ell'}$ the following bound is derived,

$$\alpha \geq \frac{|\Delta_R|}{(1-2^{-x})2^{\ell'}}$$

According to Tabler 3.1 the worst case truncation errors is $2(2^{\ell'} - 1)$ for the carry save representation and $(2^{\ell'} - 1)$ for the borrow save representation. So, no matter how large x is chosen the value of α will be greater than or equal to 2 for the carry save representation an greater than or equal to 1 for the borrow save representation. Theorem 3.8–1 states that an increasing value of α implies an increasing value of $q^{\max}$ and, therefore, the simplification in the quotient determination operation is optained at the cost of a more complicatedcomputation of multiples qm. A rough lower bound on the value of $q^{\max}$ follows from $q^{\max} \geq \alpha(\delta - 1)$ and $\delta > 2^k$. So, for the carry save representation $q^{\max} > 2(2^k - 1)$ and for the borrow save representation $q^{\max} > 2^k - 1$. In [OPT93] Orton, Peppard and Tavares are proposing a modular multiplication method similar to the method proposed by Walter. They determine the minimal required value of the scaling constant for various upper bounds of $q^{\max}$ when R is carry save represented. The upper bounds of $q^{\max}$ are $2^{k+1}, 2^{k+2}$ and 2^{k+3}. It is noted that the bound $q^{\max} < 2^{k+1}$ only is practical for the radix 2 case.

The idea of using the quotient digit estimate $q = \hat{R}$ can be viewed af a technique where *only the overflow digits of R are reduced modulo m*. Here, the overflow digits are the digits given by $\hat{R}$, i.e. the digits at a position greater than or equal to u. The adjustment of the ragne of the modulus to a value close to 2^u then enhances the "quality", i.e. reduces the value of α, of the modular reduction. There have been several proposals for using the overflow digits as a quotient digit estimate without utilising the range adjusment technique. All articles [QC82, MA85, KH88, Kno88, CBK91, IMI92a] describe some kind of variation on thes quotient determingation technique.

3.11 Summary and Discussion

In the beginning of this chapter it was explained how multiplication can be interpreted as a specialisation of exponentiation, and hence, that exponentiation methods can be formulated as multiplication methods. The similarity of exponentiation and multiplication has been noted by Knuth [Knu81, p. 443]. Knuth writes that the right-to-left binary exponentiation method (Equation (2.5)) is closely related to a procedure for multiplication that was acutally used by Egyptian mathematicians as early as 1800 B.C.. Further, Knuth writes that this multiplication method is often called the "Russian peasant method" of multiplication, since Western visitors to Russia in the nineteenth centry found the method in wide use there. The name "Russian peasant method" is now adopted by the exponentiation community as a synonym for the right-to-left binary exponentiation method. Because of this similarity the results on exponentiation methods from Chapter 2 can be reused in the discussion of multiplication methods. In the exponentiation methods the basic operation is multiplication. To obtain the analogous multiplication methods the basic operation simply is replaced by addition.

In exponentiation methods the required number of multiplications often is formulated in terms of addition chain lengths. In the analogous multiplication methods the addition chain length expresses the required number of additions. The length of an addition chain is a direct measure of the required number of basic operations. Hence, assuming that the computing time for each basic operation is one time unit the addition chain length gives a measure of the total computing time for a *sequential* computation. As shown in Chapter 2 it is possible to utilise a parallel computation and, hereby, to obtain a computing time that is less than a theoretical lower bound for the addition chain length. The validity of the assumption about equality of the computing time for the basic operations is highly dependent on the specific multiplication composition (or addition composition) under consideration. Hence, the addition chain length may not be a good measure of computing time. Some compositions have the property that squaring (or doubling) turns out to be much more efficient than the general basic operation. This is the case when integer addition is the basic operation. An integer doubling or even a many-fold doubling can be performed in a single very fast shifting operation. Indeed, by combining this property with a parallel computation it is possible to perform an *integer multiplication* in a time proportional to the logarithm of the operand lengths. This was demonstrated by the Wallace Tree computation in Subsection 3.2.2. No methods have been invented that are able to perform the computation of *modudar products* in a similar time. Opposed to integer doubling the computing time for a modular doubling is non-negligible.

In Chapter 2 various methods for evaluation of exponentials was described. The methods are formulated in general terms and they utilise very few of the algebraic properties of a specific multiplication composition. In fact, apart from the modular exponentiation method utilising the Chinese Remainder Theorem in Section 2.4, the only demand is that the composition must be associative. The variations of the methods are not due to variations of the algebraic properties. The methods differ in the strategy of computation: By increasing the available memory the total computing time can be reduced by means of the *precomputation technique*. Similarly, by increasing the number of processing elements a *parallel computation* that reduces the computing time can be scheduled. In a dedicated hardware implementation these two techniques are seen to represent the well known tradeoff between computing time and circuitry consumption.

In this chapter methods for evaluation of modular products have been described. The addition composition used in the definition of a modular product is modular addition. The presented methods and techniques are highly dependent on the *specific algebraic properties* of modular addition. Moreover, the knowledge of the *application area* of the modular products is utilised in the development of the methods. The purpose of the discussions in this chapter is to clarify how the properties of modular addition and the conditions of the application area can be utilised to obtain a fast computing time of modular products. Again, the techniques of precomputation and of parallel computation are the tools for improving the computing time. However, by studying the properties of modular addition and the conditions of the application area other possibilities reveal for exploring these techniques.

In Section 3.1 the primitive types of operations needed in the computation of modular addition were introduced. It was shown that integer addition and integer comparison are fundamental operations. The comparison operation is used for determining quotient digits in the modular reduction stage. In Section 3.2 it was discussed how the properties of various integer representations influence the computing time for the addition operation. Since the application area considered in this thesis is characterised by very long operands and by many intermediate computations it is beneficial to use redundant representations. Hereby, the time for performing an integer addition of intermediate operands becomes very fast and independent on the operand length. It was also discussed how the technique of multiplier recoding can be utilised to reduce the number of terms to be summed in an integer multiplication. This was utilised in the computation of multiples in Section 3.7. In Section 3.3 it was discussed how another kind of redundancy in the representation of the result of a modular operation can be utilised for improving the computing time of the quotient determination. It was shown that a residue can represent the desired intermediate result and, hence, that an exact integer comparison can be replaced by an estimate. Furthermore, the properties of modular arithmetic made it possible to transform the basic modular addition composition into other kinds of modular operations. This was utilised in Section 3.4 where the left-to-right 2^k-ary modular multiplication method was introduced. The basic operation used in this method is of the form $2^k s + a_i b - q_i m$, which computes a residue modulo m of $2^k s + a_i b$. When $k > 1$ the 2^k-ary method is also called a high-radix method. In the left-to-right 2^k-ary exponentiation method the analogous operation is $(s^{2^k}) * b^{a_i}$. In Chapter 2 the precomputation technique was utilised to improve the computing time by initialising a table of all possible values of b^{a_i}. As shown in Section 3.5 and further discussed in Section 3.7 it is not optimal to do the analogous precomputation of the multiples $a_i b$ in the modular multiplication method. It is better to utilise a parallel computation strategy where the multiple $a_i b$ is computed on-the-fly and in parallel with the determination of quotient digit q_i and in parallel with the subsequent formation of multiple $q_i m$. In the exponentiation method the time-critical operations were the k-fold squaring of s and the multiplication by b^{a_i}. This is not the case for modular multiplication: By utilising a parallel computation the computing time for the analogous k-fold doubling and the addition of $a_i b$ become secondary with respect to the computing time for $2^k s + a_i b - q_i m$. Indeed, it turns out that time-critical operations in this basic operation are the quotient determination, the computation of multiple $q_i m$ and the subsequent subtraction of this multiple. So, the time-critical operations in modular multiplication $(a \cdot b) \bmod m$ are the operations used for performing the modular reduction—not the operations used for performing the multiplication$a \cdot b$. These time-critical operations are identical to the operations used in SRT division methods. Having efficiently solved the addition and subtraction operation by redundant addition techniques the focus of the remaining sections was shifted toward methods for computation of multiples and methods for determination of quotient digits. Section 3.7 demonstrated how redundant addition combined with multiplier recoding can be utilised to develop a fast on-the-fly computation of multiples $a_i b$ and $q_i m$. By performing the addition in a tree-structured parallel computation scheme, as illustrated by Wallace's multiplication method, the computing time for $q_i m$ becomes proportional to the logarithm of the length of the binary representation of q_i. In most radix 2^k modular multiplication methods the quotient digit set is restricted to values of q_i with a maximal binary length of approximately k bits. Hence, for these methods the time for computing a multiple $q_i m$ will be about proportional to $\log_2 k$. This is of course a very rough measure. It should be emphasised that this time measure just gives an indication of the effect of increasing the radix. The on-the-fly computation method can be made even faster by combining it with a precomputation of some multiples. The method giving the fastest formation of multiples $q_i m$ is the table look-up method where all possible multiples are precomputed and stored in a table. Since the value of modulus is fixed for many consecutive modular multiplications the time for this precomputation will be vanishing. However, for high radix values this approach may require too much memory in a dedicated hardware implementation. The implications of the representation of b in the computation of $a_i b$ was discussed as well. In particular it was discussed how a modular multiplication method that allows redundant represented values of b can be scheduled. Since many intermediate modular multiplications are performed in the applications considered in this thesis it is natural to keep the intermediate operands in the redundant representation. Hereby, the computing time and the circuitry for performing the conversion between redundant and non-redundant represent ation can be avoided. In Section 3.8 the interdependencies of the parameters characterising the quotient determination was analysed. It was shown that the precision of the quotient determination operation cannot be reduced without increasing the quotient digit set, and vice versa. So, in the modular multiplication methods there is a tradeoff between the complexity of the quotient determination operation and the computation of the multiples $q_i m$. Section 3.9 demonstrated how a very simple scaling of the value of modulus can be utilised to obtain conditions for the quotient determination operation that are arbitrary close to the conditions for the analogous quotient determination operation in SRT division. This implies that the time for computation of the operation $2^k s + a_i b - q_i m$ becomes about equal to the computation of $2^k s - q_i m$ which is the basic operation in SRT division methods. Section 3.10 comprised a detailed discussion of quotient determination methods based on table look-up techniques. Compared to the general purpose SRT division methods, where the operands cannot be expected to be fixed for more than a single division operation, the precomputation technique can be further explored to simplify the quotient determination operation. Because the precomputation of the quotient digit table only has to be performed once for each change of modulus and because the modulus is fixed for many consecutive modular multiplication it is advantageous to perform a comparatively complicated precomputation. Indeed, it was discussed how an adjustment of the range of modulus can lead to a very simple quotient determination method, where the quotient digit value becomes equal to the value encoded by the most significant digits of the intermediate operand R to be modular reduced. In the table look-up methods the address of the entry containing the quotient digit is given by the value of the most significant digits of R. Since these digits are redundant represented it is required to perform a conversion into non-redundant representation in order to limit the table size. The conversion can be done in a time proportional to the logarithm of the number of radix 2 digits. According to the examples in the section this number can be expected to be about k plus a few digits. So, roughly estimated, the computing time for determination of a quotient digit by the table look-up method is proportional to $\log_2 k$.

As explained in Section 3.4, the aim of the high-radix modular multiplication approach is to obtain a faster computation by reducing the required number of intermediate operations $2^k s + a_i b - q_i m$. This number of operations is about proportional to $\frac{1}{k}$. The indications of the $\log_2 k$ rate by which the time for computation of multiples and for quotient determination increases for increasing radices show that the total computing time for the high-radix modular multiplication method indeed is decreasing for increasing radices.

Chapter 4

Modular Exponentiation Processor

In the previous two chapters several methods for evaluation of exponentials and modular products have been presented and discussed. The objective of this thesis is to investigate the possibilities for supporting applications based on evaluation of modular exponentials, like the RSA crypto system, with adequate computing power. Since the hardware implementation medium—in particular in the form of special-purpose VLSI circuits—is among the fastest available media, the descriptions have been biased toward methods that are suited for hardware implementation. In order to obtain experimental evidence for the soundness of the methods and, moreover, to obtain insight into the problems and limitations of such a VLSI implementation, a project of implementing a modular exponentiation processor was initiated in the fall 1990.

In Section 4.1 the background and the history of the project is briefly described. Section 4.2 gives an overview of the methods chosen for evaluation of modular exponentials and modular products. Furthermore, a relatively detailed description of the hardware architecture is provided. The layout of the processor is shown in Section 4.3. In Section 4.4 testing procedure, and the performance of the fabricated processor, is presented. Finally, Section 4.5 comprises a summary and a discussion of the experiences obtained from the project.

4.1 Background and History of the Project

After a period of studying methods for computation of modular exponentials, resulting in the Master's thesis [OSA90b], the internal report [OS90] and the article in Appendix A, a cooperation with the Department of Research and Development, Jydsk Telefon/Tele Danmark, was established in September 1990. The aim of the cooperation was to demonstrate, that it is possible to implement a single-chip modular exponentiation processor capable of performing real-time RSA encryption of digital data transmitted on an ISDN channel. The transmission rate of an ISDN channel is 64 Kbit/s. At that time, the fastest known implementation was a PC plug-in board from the company Thorn EMI [Tho88] capable of evaluating modular exponentials at a rate of 29 Kbit/s for 512 bit operands. In RSA applications the length of the modulus and of the exponent corresponds to the key length. It should be mentioned that the Thorn EMI board supports RSA encryption based on a utilisation of the Chinese Remainder Theorem (see Section 2.4). Hereby, the encryption rate of the board may be increased to 56 Kbit/s for 512 bit key lengths.

The project of implementing the modular exponentiation processor was based on the above mentioned studies of computation methods. The work of designing the circuitry and doing the physical layout of the processor was performed in the environments of the VLSI design group at Jydsk Telefon. Before the project of implementing the professor was started, the following requirements of the processor were specified:

- It should be able to perform a real-time RSA encryption of data at a transmission rate of at least 64 Kbit/s. As mentioned, this is the transmission rate of an ISDN channel.

- The key length should be 561 bits. That means, the processor should perform the evaluation of modular exponentials with 561 bit moduli. In RSA encryption a block b of a message is encrypted by evaluation of $b^e \bmod m$. To be able to uniquely determine b during the subsequent decryption process, it is required that $b < m$. By limiting the length of b to 560 bits, i.e. 70 bytes, and using 561 bit moduli this requirement is fulfilled.

- The processor should be implemented by a single VLSI chip. This limitation was imposed in order to minimise the physical size of the implementation. The physical size influences the applicability of the processor to be embedded into telecommunication equipment. Furthermore, the cost of a single-chip implementation is expected to be less than the cost of an implementation comprising more chips.

- The processor should support a special interface used internally in a specific ISDN telephone. The aim was to demonstrate the capabilities of the processor by embedding it into an ISDN telephone and, hereby, be able to demonstrate a real-time RSA encryption of digitised voice transmitted on an ISDN channel.

- The functionality of the processor should be testable. In case of a malfunction it should be possible to track down the source causing the error.

It was decided to use the methods and the hardware architecture described in the article in Appendix A with a minor modification: Since the quotient determination was expected to be one of the time critical operations, the quotient determination method in Appendix A was replaced by the method described in the article in Appendix B. The latter method was believed to be faster.

For the implementation of the processor a commercial chip development system, ChipCrafter [Cas91b] from the company Cascade Design Automation Corporation, was used. The system is a so-called silicon compiler, where most of the tasks of the design process are automated. A brief description of the ChipCrafter development system is included in the article in Appendix C. A collection of chip development tools from Valid Logic Systems and a circuit simulator, Saber [Ana92] from Analogy, were used as well. Finally, a design rule checker from Cadence Design Systems was applied.

The following description gives an overview of the history of the project:

September 1990 – November 1990. During the first three months some initial experiments on the expected area consumption of the processor were done. The parts of the circuitry, expected to be the most area consuming, were specified at a schematic entry level. Then, using ChipCrafter's automatic tools, some estimates of the area were obtained. The area estimate was 175 mm^2 for a design resulting from a pure application of the ChipCrafter development system. An area of 110 mm^2 was estimated for a design resulting from a mixed design strategy, where the area critical parts were full custom designed and the remaining parts were designed in ChipCrafter's environment. Since the aim of the project was to develop a prototype—not a product ready for commercial sale—it was decided to stay in the ChipCrafter environment. The time for implementing the processor was of greater importance than the area of the prototype. Of course, due to limitations of the fabrication facilities, there was an upper bound of the physical dimensions of the processor. This bound was approximately 200 mm^2.

As a curiosity, it should be mentioned, that the prototype was believed to be ready for fabrication in February 1991. This turned out to be far too optimistic: The processor was shipped for fabrication in January 1993!

December 1990 – May 1991. During the next six months the processor was specified in detail. This included the design of the state machines for controlling the sequence of computations and for controlling the interfaces. In case the final processor should be malfunctioning, a mechanism for tracking down the source for the errors was included. This mechanism allows the processor to be configured into a test mode, where all internal registers are connected into so-called scan-chains, e.g. [WE92, Chapter 7]. The scan-chains behave as ordinary shift-registers, allowing the user to stop the operation of the processor and to inspect the internal state of the processor by shifting out the register values by means of the scan-chain. Similarly, the processor can be brought into an arbitrary internal state by shifting in a new value specified by the user.

The functionality of the parts were validated through extensive simulations. The design was structured in a hierarchy, where the root was the complete processor design and the leafs were the primitive cells of the cell library. The simulation were carried out in a modular way, starting from the level just above the leaf cells and proceeding to the top level. Hereby, the functionality of a part of the design was validated before this part was included in another part at a higher level.

Finally, the timing of the parts in the design was analysed. According to the analysis, there would be no problem in meeting the timing requirements of the processor. There was, however, another serious problem: The design consumed by far too much area. This problem became the main issue in the remaining work until the design was sent for fabrication. The article in Appendix C reports some of the work and the results of the efforts of reducing the area.

June 1991 – December 1991. Under the constraints of the chosen chip development system the area of the processor layout was reduced. This involved optimisation of the placement of the circuit parts and of the routing of the wires connecting the parts. Furthermore, some of the most area consuming parts were redesigned in order to provoke the system to return a smaller layout. In spite of these actions it was not possible to obtain an area below 340 mm^2. The layout was still too large to fabricate. The design consisted of about 550,000 transistors and the power analysis showed a power consumption of approximately 2.5 W at a 25 MHz clocking frequency.

Simultaneously, the sources for the large area consumption were analysed. Apart from a relatively bad placement and routing, the leaf cells from the cell library had a large area consumption in comparison with other known implementations. So, experiments on replacing some of the cells with other user designed cells were initiated. Moreover, a set of programs, that enforced a certain placement of the leaf cells by manipulating the internal representation of the development systems database, was written.

January 1992 – April 1992. It was decided to replace some of the leaf cells in the cell library with smaller user designed cells. A new set of leaf cells was developed. This included detailed simulations of the electrical characteristics of the circuitry. The new cells were ported to the development system.

May 1992 – November 1992. It was necessary to make some adjustments of the design specification in order to include the new leaf cells. Consequently, a new set of simulations of the functionality was required. The optimisation of the placement and routing was repeated. The resulting area of the final design were approximately 210 mm^2. The transistor count was reduced to about 300,000 transistors and the power consumption was expected to be about 1.5 W at a 20 MHz clocking frequency.

A timing analysis indicated that the processor should be able to work at a 25 MHz clocking frequency under the worst case conditions, i.e. for a statistically slow process technology, a voltage of 4.5 V and a temperature of 85°C. The clocking frequency of 25 MHz corresponds to an encryption rate of about 100 Kbit/s.

Since the development system was unable to assign dimensions to the wires supplying the user designed leaf cells with power, it was necessary to do experiments and calculations on these dimensions. Furthermore, because the new flip-flop cells were known to be sensitive to the slope of the clocking signal, detailed simulations were made in order to assure that the clocking signal would be properly distributed on the chip.

December 1992 – January 1993. The final placement of the pad cells were done in accordance to the specification of the package for the processor.

The remaining two month, before the layout was sent for fabrication, were spent on checking the layout. First a so-called LVS (layout versus schematic) check was performed. This consists of a comparison of the transistor netlist, extracted from the final layout, with the transistor netlist obtained from the schematic specification level. The check revealed about ten signal wires, that had been left unconnected by the routing tool. These wires were manually connected by means of a layout editor. Furthermore, the power distribution was manually improved. The final check was the design rule check (DRC). This check revealed a set of design rule violations resulting from the routing of wires in the vicinity of the user defined cells. Fortunately, these violations were easily corrected by manually moving the wires.

February 1993 – December 1993. The chips were returned from fabrication in March 1993. In order to test the functionality of the processor and to measure the performance, a fairly large and complex circuitry board was constructed. The design, construction and test of the test board was completed around November 1993. The outcome of the test and the results of the measurements are described in Section 4.4.

4.2 Processor Description

From the user's point of view, the modular exponentiation processor can be divided into the three functional blocks depicted in Figure 4.1. The *control unit* is configuring the functionality of the processor in accordance to the configuration chosen by the user. Furthermore, the control unit is controlling the internal sequence of computations, and it is implementing the interface protocols. As described in the engineering data in Appendix E, the processor can be configured into several modes. One of these modes is the *test mode*, which is entirely used for testing the processor by means of the built-in scan-chains. The other modes are *normal operation modes*, which are used when the processor is used for evaluation of modular exponentials. The communication with the processor is controlled by means of a set of interface signals, provided by the user, and a set of flags generated by the processor. The data communication is bit-serial.

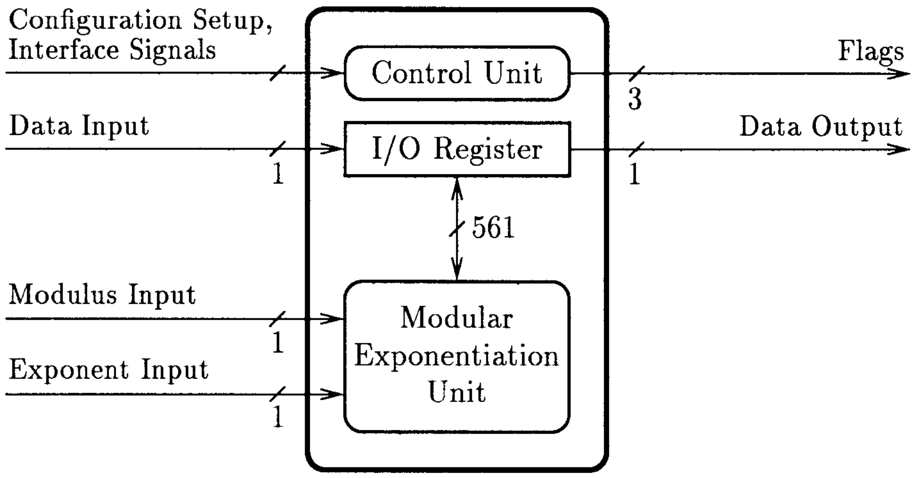


Figure 4.1: Functional blocks of the processor.

The processor's *I/O register* is used for collecting the input data and for holding the result of the previous evaluation. The I/O register is a shift-register. So, simultaneously with receiving a string of data input bits, a string a data output bits is shifted out from the register. When the I/O register is filled with a new operand, say b, the content of the register is swapped with the result, say c, from the *modular exponentiation unit*. Then, a new input operand can be shifted into the I/O register while the result c is shifted out from the processor. Simultaneously with this data communication, the modular exponentiation unit evaluates the exponential $b^e \bmod m$. The value e denotes the exponent, and the value m denotes the modulus. These values are shifted into the processor during an initialisation process, and the values are used during all the subsequent modular exponentiations until a new initialisation is done. Appendix E contains a more detailed description of the configuration possibilities, the interface protocols and the pins of the processor.

4.2.1 Modular Exponentiation Unit

The modular exponentiation unit is implementing a parallel computation of the right-to-left binary exponentiation method presented in Section 2.3. The parallel computation requires that two modular multiplications, a squaring of the form $y \cdot y \bmod m$ and a general multiplication of the form $x \cdot y \bmod m$, are computed in parallel. The parallel modular multiplications are characterised by having a common multiplicand. The parallel computation is accomplished by a single modular multiplication unit that is pipelined into two stages.

The hardware architecture of the modular exponentiation unit is illustrated in Figure 4.2. The figure shows the external data connections to the modular exponentiation unit, the internal functional blocks and the internal data connection of these blocks. Apart from the pipelined modular multiplication unit, the modular exponentiation unit consists of five registers and a small negation unit. All of the registers are holding binary represented operands, and the data connections are communicating binary represented data. Some of the data connections in Figure 4.2 are annotated with the width of the data bus. If no explicit width is given, the bus has a width corresponding to the full width of the communicated operand.

When the exponent e and the modulus m are feed serially into the processor, the exponent is shifted into the shift-register denoted the *E register*, and the modulus is shifted through the negation unit into the *M register*. The *negation unit* negates the value of m. Hence, prior to each modular exponentiation the E Register contains the value e, and the M Register contains the value $-m$. In the right-to-left exponentiation method, the bits of e are inspected from right to left. During the exponentiation process, the E

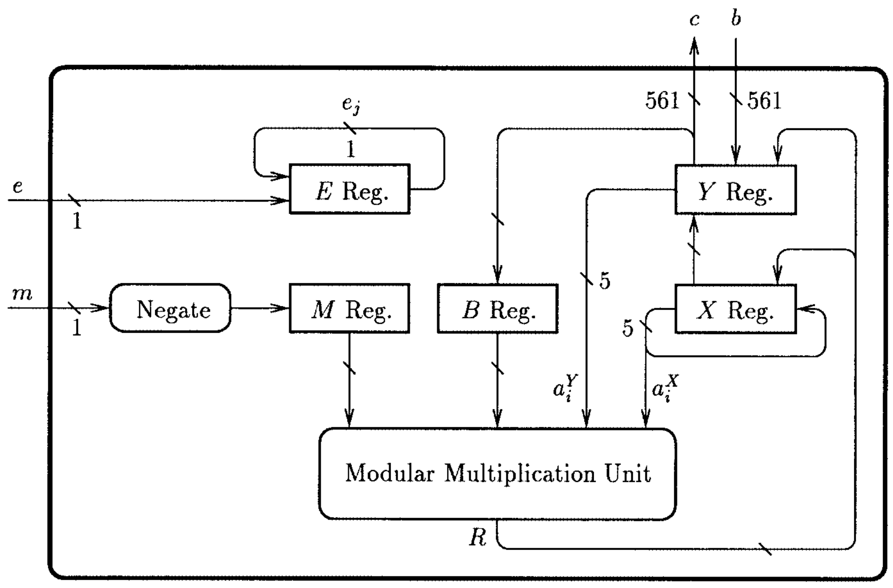


Figure 4.2: Hardware architecture of the modular exponentiation unit.

register is configured as a cyclic shift-register. Hereby, the actual exponent bit, denoted e_j, is positioned at the least significant end of the register. After completion of the exponentiation process, the content of the E register will be as prior to the start of the process, and E will be ready for the next exponentiation.

The purpose of the Y *register* is two-fold: First, this register is used for the exchange of new data, to be exponentiated, and for the result of an exponentiation. The exchange is implemented as a swap of the contents of the Y register and the I/O register. Second, the Y register is used for holding the multiplier operand of the modular squaring operation. The B *register* is holding the common multiplicand for both of the modular multiplications that are computed in parallel. So, prior to each parallel modular multiplication the B register is loaded with the content of the Y register. The X *register* is holding the other multiplier. During the modular multiplication process the multiplier registers, X and Y, are configured as shift-registers. Since the modular multiplication unit implements a radix 2^5 left-to-right method, the multiplier digits, denoted a_i^X and a_i^Y, are inspected from left to right. Each radix 2^5 multiplier digit is encoded in five bits and, therefore, the bits of the multiplier registers are shifted five positions at each shift operation. The X

register is cyclic. So, after each modular multiplication process, the content of X will be as prior to the process. It is only in case of a high exponent bit e_j, that X is loaded with a new value. After every multiplication, Y is loaded with the resulting modular square. Hence, there is no need for preserving the content of the Y register. Upon completion of the exponentiation process, the final result will be the content of the X register. The result is moved to the Y register, and the modular exponentiation unit is ready for a new exchange of data with the I/O register.

Denote the contents of the X, Y and B registers by, respectively, x, y and b. Then, the *modular multiplication unit* computes, in parallel, a residue module m of $y \cdot b$ and of $x \cdot b$. The resulting residues are binary represented, and they are restricted to the non-negative range $[0; 2m[$. In accordance to the discussion on the representation of intermediate operands in Section 3.6, the modular multiplication unit allows the result of a previous modular multiplication to be used, without any further conversion, as input to the next modular multiplication. This means that x, y and b can be arbitrary residues in the range $[0, 2m[$. The results from the modular multiplication unit are delivered on the data connection denoted by R. Since the unit is pipelined, the results are delivered in two consecutive time steps: First the result $R^Y \equiv y \cdot b \pmod{m}$, then the result $R^X \equiv x \cdot b \pmod{m}$. Because the results from the modular multiplication unit are residues in the range $[0, 2m[$, it is necessary to convert the final result of the modular exponentiation into the residue range $[0; m[$. This is accomplished by a subtraction of m followed by an inspection of the resulting sign. The final conversion is supported by the modular multiplication unit.

4.2.2 Modular Multiplication Unit

The modular multiplication unit is implementing a radix 2^5 left-to-right modular multiplication method. The method is based on a recursive evaluation of expressions of the form $R_i = (2^5 R_{i+1} + a_i b) - 2^5 q_{i+1} m$. As described in Section 3.5, it is possible to utilise a parallel computation strategy, where the evaluation of $T_i = 2^5 R_{i+1} + a_i b$ is overlapped with the evaluation of $2^5 q_{i+1} m$. Furthermore, in order to improve the quotient determination complexity, the modular multiplication unit utilises the scaling technique presented in Section 3.9. The scaling constant, by which the modulus m and the multiplier a are scaled, is $2^r = 2^5$. In fact, the modular multiplication method is identical to the method described by Algorithm 3.9–1 with the modification, that the resulting product is a residue in the non-symmetric range $[0; 2m[$. Finally, as mentioned above, the modular multiplication unit is pipelined. This means that two modular products are simultaneously computed. Since the multiplicand b and the modulus m are common operands for both multiplications, it is sufficient to implement a simultaneous evaluation of the expressions $R_i^Y = T_i^Y - 2^5 q_{i+1}^Y 2^r m$ and $R_i^X = T_i^X - 2^5 q_{i+1}^X 2^r m$, where $T_i^Y = 2^5 R_{i+1}^Y + a_i^Y b$ and $T_i^X = 2^5 R_{i+1}^X + a_i^X b$. The radix 2^5 digits a_i^X and a_i^Y denote the digits at position i of the scaled multiplier operands.

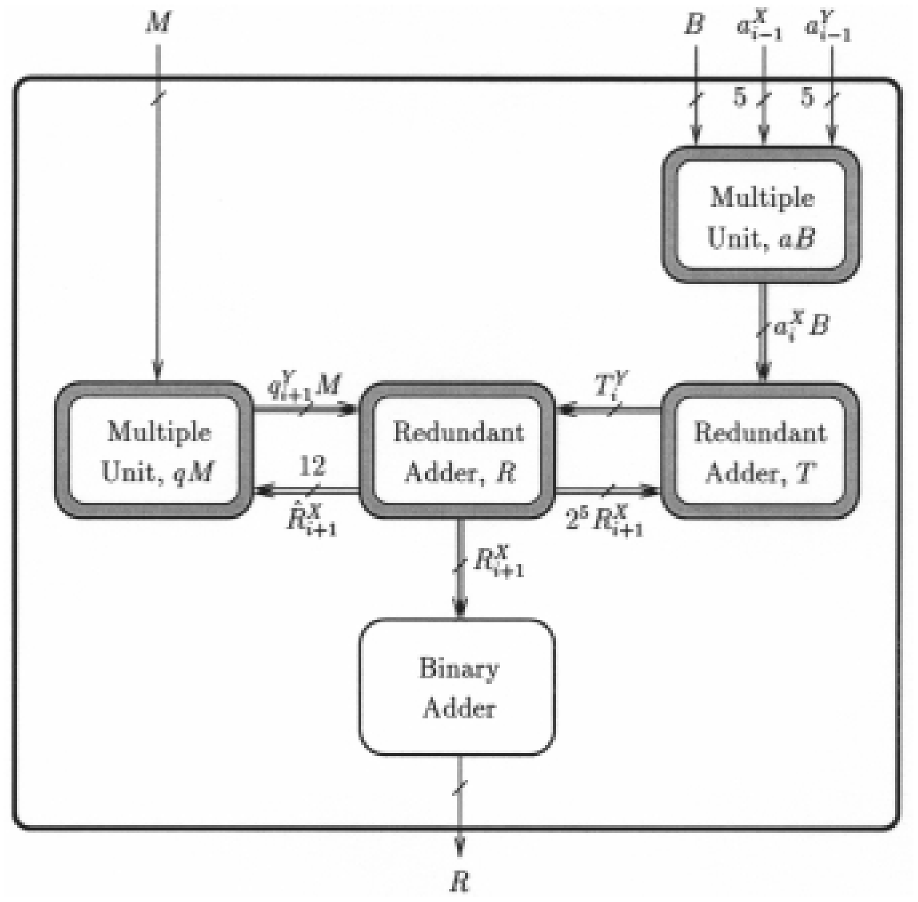


Figure 4.3: Hardware architecture of the modular multiplication unit.

The hardware architecture of the modular multiplication unit is illustrated in Figure 4.3. All of the internal connections are communicating data represented in the redundant carry save form. These connections are symbolised by double lines in the figure. The external connections are connected to the registers of the modular exponentiation unit. The external connections are communicating (non-redundant) binary represented data. Before a modular multiplication process is started, the external modulus register M and the multiplicand register B must be properly initialised. To compute residues modulo m of $y \cdot b$ and $x \cdot b$, M must hold the value $-2^{5+r}m = -2^{10}m$ and B must hold the value b. During the modular multiplication process the contents of the multiplier registers X and Y are shifted, digit by digit from the most significant end, into the modular multiplication unit. As stated by the stimulus condition of Algorithm 3.9–1, a multiplier register, say X, must hold the scaled multiplier $2^r x = a^X_{n'-1} a^X_{n'-2} \dots a^X_0$ plus an additional digit $a^X_{-1} = 0$. Since $2^r = 2^5$ this implies that X must be initialised with the value $2^{10}x$. Similarly, register Y must be initialised with the value $2^{10}y$. The number of radix 2^5 digits held by a multiplier register is $n_i = n' + 1$, where n' denotes the number of digits used for representing the scaled multiplier. Since a multiplier may be a result from a previous modular multiplication, it is known to belong to $[0; 2m[$ and, therefore, it may need 562 bits to be binary represented. So, when scaled by 2^5, the number of bits increases to 567. This means the number of radix 2^5 digits of the scaled multiplier is $n' = \lceil \frac{567}{5} \rceil = 114$. Hence, the number of radix 2^5 digits in the multiplier registers is $n_i = 115$.

The modular multiplication unit contains two pipelined units for redundant addition and two pipelined units for computation of multiples. Furthermore, a binary adder, used for converting the carry save represented results into binary representation, is included. The units with a grey-shaded frame in Figure 4.3 are the pipelined units. They are pipelined into two stages and, hence, each unit contains a register implementing the pipeline buffer.

The *multiple unit denoted aB* computes multiples of the multiplicand. The unit has three input operands: The multiplicand B, and the two multiplier digits a^X_{i-1} and a^Y_{i-1}. The actual multiplier digit used in the computation alternates between a digit from register Y and a digit from register X. A multiple is produced in each clock period. The sequence of computed multiples can be expressed as

$$a^Y_{113}B, a^X_{113}B, a^Y_{112}B, a^X_{112}B, \dots, a^Y_{-1}B, a^X_{-1}B, \tag{4.1}$$

The *multiple unit denoted qM* performs a similar computation. It computes multiples of the modulus value in register M. Instead of receiving the quotient digit q_{i+1} to be used in the computation, the unit receives a truncated version $\hat{R}_{i+1}$ of the intermediate result R_{i+1}. Hence, in this multiple unit, circuitry for determination of the quotient digits is included. The input $\hat{R}$ is equal to the 12 most significant carry save digits from position 561 to 572 of R, i.e. $\hat{R} = r_{572}r_{571}\ldots r_{561}$.

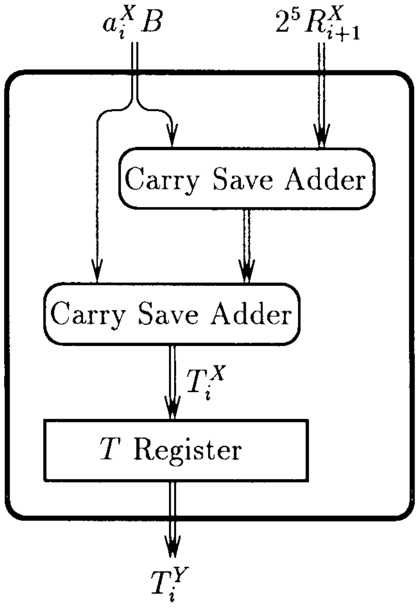


Figure 4.4: Hardware architecture of the redundant adder denoted T.

The *redundant adder denoted* T is implementing the addition operation in the expressions $T_i = 2^5 R_{i+1} + a_i B$. Since both terms in this expression are carry save represented, the adder can be identified as a 4–2 adder (see Subsection 3.2.2). The hardware architecture of the unit is shown in Figure 4.4. The register, T, implementing the pipeline buffer is buffering the result from the redundant addition. Since the result is carry save represented, register T corresponds to two registers for holding binary represented data. The *redundant adder denoted* R is implementing the addition operation in the expressions $R_i = T_i + q_{i+1}M$. (Recall that register M contains the value $-2^{10}m$. Hence, the subtraction is converted to an ordinary addition). The implementation of this adder is similar to the other redundant adder.

The *binary adder* is used for converting the final results, R_{-1}^Y and R_{-1}^X, into non-redundant binary representation. Since the computing time for a binary addition is relatively large compared to the computing time for the other units, more than a single clock period is assigned for this operation. The number of extra clock periods is denoted n_w, the number of *wait states*. The required number of wait states depends on the actual clocking frequency. Therefore, the parameter n_w can be configured by the user. While the processor is waiting for the conversion to be completed, the contents of all the registers, used in the computation of modular products, remain unchanged. The binary adder was generated by the chip development tools. According to the data book [Cas91a], this so-called high speed adder uses a *carry select* architecture with a *Manchester carry chain*. (These addition techniques are described in standard books on computer arithmetic and VLSI design, e.g. [WE92, Chapter 8]).

The data connections in Figure 4.3 and in Figure 4.4 are annotated with the values computed by the various units at a certain instant of time. The annotation can be viewed as a snapshot of the internal state of the modular multiplication unit. The snapshot shows the internal state just after the evaluation of R^X_{i+1}. Because of the pipelined architecture, all of the pipelined units produce an alternating sequence of results as illustrated by (4.1). In general, when a unit produces a result marked with Y, it simultaneously consumes inputs marked with X, and vice versa.

It should be mentioned, that the final results are left-shifted versions of the residues modulo m of $x \cdot b$ and $y \cdot b$. The results can be expressed by $R^Y_{-1}/2^{10} \equiv_m y \cdot b$ and by $R^X_{-1}/2^{10} \equiv_m x \cdot b$. According to the above discussion, the multiplier registers X and Y must be initialised with $2^{10}x$ and with $2^{10}y$ prior to each modular multiplication. Hence, the updating of these registers with a result from a previous modular multiplication can be achieved by a simple load of the value R_{-1}.

The modular multiplication unit supports the conversion of a residue, say R', in the range $[0; 2m[$ into the range $[0; m[$. As mentioned in the previous subsection, such a conversion is required for the final result of a modular exponentiation. The conversion is performed by a subtraction of m, i.e. the operation $R' - m$, and an inspection of the resulting sign: First register B is loaded with the value R'. Then, by enforcing certain values to the multiplier digits and quotient digits, the following computation implements the subtraction using the existing hardware architecture:

$$\begin{array}{lcll} R_1 & := & 0; & \\ R_0 & := & (2^5 R_1 + a_0 B) + q_1 M, & \text{where } a_0 = 2^5 \text{ and } q_1 = 0; \\ R_{-1} & := & (2^5 R_0 + a_{-1} B) + q_0 M, & \text{where } a_{-1} = 0 \text{ and } q_0 = 1; \end{array}$$

By insertion of the digit values, the computation is seen to result in $R_{-1} = 2^{10}B + M$, which equals the value $2^{10}(R' - m)$. Finally, by means of the binary adder the sign of $R' - m$ is computed.

4.2.3 Multiple Units

The hardware architecture of the multiple unit that computes multiples of B is shown in Figure 4.5. The input operands are the multiplier digit a^Y_{i-1} from register Y and the multiplier digit a^X_{i-1} from register X, and the multiplicand in register B. The input operands are binary represented. The digit set for the radix 2^5 multiplier digits is $\{0, 1, \ldots 31\}$. The multiple unit utilises the technique described in Subsection 3.7.1, where a multiplier digit, say a^X_i, is recoded into three radix 4 digits $(d_0, d_1, d_2)^X_i$ such that $a^X_i = 4^2 d_2 + 4^1 d_1 + 4^0 d_0$. The digit set for these radix 4 digits are $\{\bar{1}, 0, 1, 2\}$ for d_0 and d_1, and $\{0, 1, 2\}$ for d_2. Therefore, the multiple $a^X_i B$ can be computed as the sum of three shifted versions of B and $-B$. The sum is computed by a carry save adder and, hence, the resulting multiple is carry save represented. The multiple unit is pipelined into two stages. The pipeline buffer is a six bit register denoted the *a register*.

The 2–1 *multiplexer* in the figure is used for selection of either a multiplier digit from register Y or a multiplier digit from register X. As mentioned in the previous subsection, the actual multiplier digit used in the computation alternates between these two registers.

The unit denoted *recode a* is implementing the multiplier digit recoding. The value of each of the resulting radix 4 digits (d_0, d_1, d_2) is encoded into two bits. Hence, the encoding of a multiplier digit expands from five bits to six bits. It is possible to enforce the recode unit to produce the (d_0, d_1, d_2) encoding of the multiplier digit value 2^5 *independently* of the actual value of the multiplier digit at the input. As described in the previous subsection, this feature is needed for the final conversion of the result of the modular exponentiation. A special control signal *set*32 controls the feature. A synthesis tool named FINESSE [Cas91c] was used to generate the circuit for the

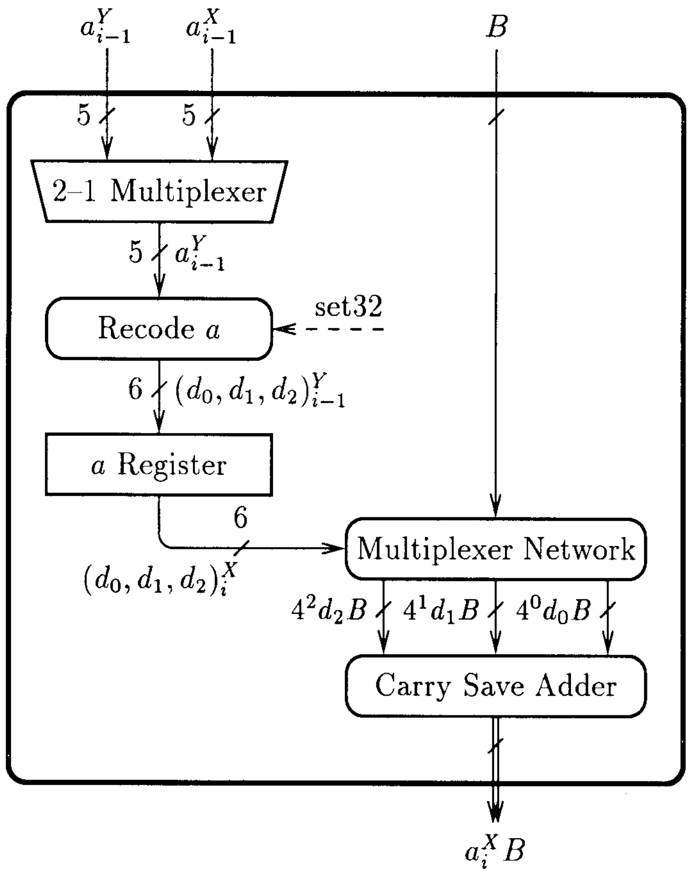


Figure 4.5: Hardware architecture of the multiple unit for computing multiples of B.

recode unit. The functionality of the unit was specified by a table. From this table, FINESSE generated a module comprising 15 instances of the basic combinatorial circuit cells from the cell library.

The unit denoted *multiplexer network* is producing the three terms $4^0 d_0 B$, $4^1 d_1 B$ and $4^2 d_2 B$ by a simple selection between shifted values of B and $-B$. The hardware architecture of this unit is shown in Figure 4.6. It consists of two 4–1 multiplexers for generation of $4^0 d_0 B$ and $4^1 d_1 B$ and a single 3–1 multiplexer for generation of $4^2 d_2 B$. As illustrated by the figure, the encoding (d_0, d_1, d_2) of the multiplier digit is used as selection signal inputs to the multiplexers. The input $\neg(4^1 B)$ denote the bit-wise inverted version of $4^1 B$:

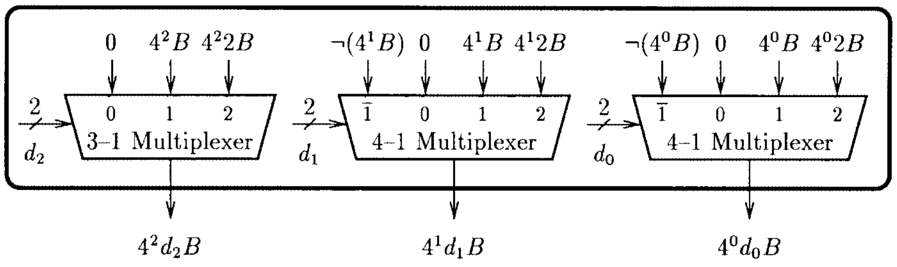


Figure 4.6: Hardware architecture of multiplexer network.

The two's complement is used for the representation of negative operands. Therefore, the negative value -4^1B can be computed by $\neg(4^1B) + 1$. The increment of $\neg(4^1B)$ is not shown in the figure: Note that the values of the four least significant bits of 4^2d_2B are always zero. Hence, the increment can be implemented by replacing the value of the least significant bit of 4^2d_2B with 1. Similar for the input denoted $\neg(4^0B)$. If both d_0 and d_1 are equal to $\bar{1}$, the binary value of the two least significant bits of 4^2d_2B is set to 10.

The hardware architecture of the multiple unit that computes multiples of M is shown in Figure 4.7. The principles for the computation of these multiples is similar to those presented above. However, in this unit, a *quotient determination unit* is included. The quotient determination unit computes the quotient digits q^Y_{i+1} to be used in the formation of the multiples $q^Y_{i+1}M$. The pipeline buffer is a 40 bit register denoted *q register*. It is holding the computed quotient digit. The possible values of the resulting quotient digit is known to be restricted to the quotient digit set $\{0, 1, \ldots, 39\}$. The representation of the resulting quotient digit is a bit-vector, where exactly one of the elements is equal to one. The bit-vector has 40 elements, and the position of the one-valued bit determines the value of the quotient digit: Position 0 represents the value 0, position 1 represents the value 1, etc.. The unit denoted *recode q* performs a recoding from the bit-vector representation into the (d_0, d_1, d_2) representation used by the multiplexer network. As for the above described recoder, this unit can be forced to produce a (d_0, d_1, d_2) encoding of a certain quotient digit value. This value is equal to one, and it is controlled by the signal *set01*. The functionality of the recode unit was specified by a table, and FINESSE was used for generating the circuit. The circuit comprises 30 instances of basic combinatorial cells.

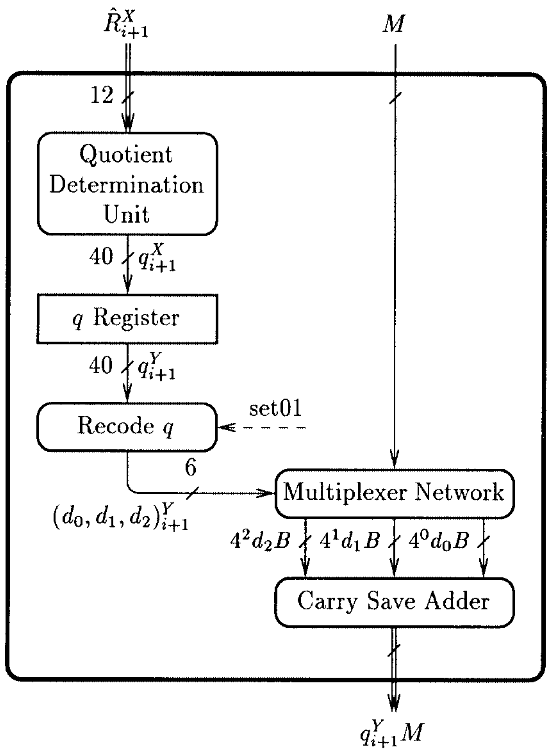


Figure 4.7: Hardware architecture of the multiple unit for computing multiples of M.

4.2.4 Quotient Determination Unit

The method implemented by the quotient determination unit is identical to the method described in the article in Appendix B. The residue range is non-negative, and the operation of the unit can be specified by

$$\{\text{Determine integer } q \text{ such that } R - qm' \in [0; \alpha m'[\text{ and } q \in \{0, 1, \dots, 42\}\}.$$

Since the unit merely computes an *estimate* of the exact quotient digit, the parameter α has a value greater than 1. The scaled modulus m' is equal to $2^5 m$, where $m \in [2^{560}; 2^{561}[$. The restriction on the maximal allowable quotient digit value, $q^{\max} = 42$, is imposed by the method for computation of multiples. However, as will be shown below, the quotient digit values determined by the method of this quotient determination unit will never exceed 39.

The basic principle of the method is to assign the value j to the quotient digit q if $R \in [jm'; (j+1)m'[$ and, hereby, obtaining that $R - qm' \in [0; m'[$. By inspecting the results of the comparison operations $R - jm' \geq 0$ and $R - (j+1)m' \geq 0$ it is checked if R indeed belongs to $[jm'; (j+1)m'[$. The comparison constants $c_j = -jm'$, where $j \in \{0, 1, \ldots, 39\}$, are computed simultaneously with the initialisation of the processor. Since the modulus m is put serially into the processor, a serial computation of the constants can be implemented by means of a few full adders and flip-flops.

However, in order to reduce the computing time for the comparison operations, the comparisons are limited to the *most signifiant parts of the operands:* The quotient determination unit performs the comparisons

$$\hat{R} + \hat{c}_j \geq 0, \text{ for all } j \in \{0, 1, 2, \ldots, 39\}, \tag{4.2}$$

where $\hat{R}$ and $\hat{c}_j$ refer to the most significant parts of R and c_j. Then, if $\hat{R} + \hat{c}_j \geq 0$ and $\hat{R} + \hat{c}_{j+1} < 0$ the value j is assigned to the quotient digit. Using the notation in Section 3.10, where Δ refers to the truncation error introduced by neglecting the u least significant digits, the operands are written as $R = 2^u \hat{R} + \Delta_R$ and $c_j = 2^u \hat{c}_j + \Delta_{c_j}$. Hence, $\hat{R} + \hat{c}_j \geq 0$ implies that $R - jm \geq \Delta_R + \Delta_{c_j}$, and $\hat{R} + \hat{c}_{j+1} < 0$ implies that $R - (j+1)m' < \Delta_R + \Delta_{c_{j+1}}$. So, $R - jm' \in [\Delta_R + \Delta_{c_j}; m' + \Delta_R + \Delta_{c_{j+1}}[$. According to Table 3.1, page 107, the truncation errors of the binary represented comparison constants belong to $[0; 2^u - 1]$, and the truncation error of the carry save represented R belongs to $[0; 2(2^u - 1)]$. Therefore, the resulting quotient digit value will ensure that

$$R - qm' \in [0, m' + 3(2^u - 1)[= [0; \alpha m'[. \tag{4.3}$$

According to the analysis in the article in Appendix B, it is sufficient to use the precision $u = 561$ to restrict the values of q to the set $\{0, 1, \ldots, 42\}$. Indeed, it turns out that q will never be greater than 39: Equation (4.3) gives that $\alpha m' = m' + 3(2^u - 1)$. By inserting $u = 561$ and using $m' = 2^5 m \geq 2^5 2^{560}$, it follows that $\alpha < \frac{19}{16}$. Furthermore, R is the result of a recursive computation of the form $R_i := 2^5(R_{i+1} - q_{i+1}m') + a_i b$, where $a_i \in [0; 31]$ and $b \in [0; \alpha m[= [0; \frac{\alpha}{32} m'[$. Hence, $R < 32(\alpha m') + \frac{31}{32}\alpha m'$. Finally, using $\alpha < \frac{19}{16}$, it is seen that $R < 40m'$ and, consequently, the quotient digit value will be less than 40.[1]

[1]In fact it is sufficient to use the quotient digit set $\{0, 1, \ldots, 38\}$: Theorem 3.8–1 states that a digit set bounded by $q^{\max} = [(\delta - 1)\alpha]$ is sufficient to achieve $R - qm' \leq \alpha m'$, when $R \leq \delta \alpha m'$. According to the above discussion, δ is bounded by $\delta < 32 + \frac{31}{32}$, and α is bounded by $\alpha < \frac{19}{16}$. Therefore, $\lceil (\delta - 1)\alpha \rceil$ is bounded by $\lceil (\delta - 1)\alpha \rceil \leq 38$.

In the quotient determination unit the comparison operations include the 12 most significant digits of R and $c_j = -jm'$ from position 561 up to position 572. Since an upper bound of R and jm', given by $40m' < 40\ 2^5 2^{561}$, is less than 2^{572}, the digits at position 572 determine the sign of the operands. (The two's complement representation is used for representation of negative operands).

The hardware architecture of the quotient determination unit is shown in Figure 4.8. It consists of 39 instances of a comparison unit and 40 exclusive or (XOR) gates. The input to the quotient determination unit, and to the instances of the comparison unit, comprises the 12 most significant digits, denoted $\hat{R}^X_{i+1}$, of the carry save represented intermediate operand R^X_{i+1}. The output of the quotient determination unit is a bit-vector representation $(v_0, v_1, \ldots, v_{39})$ of the value of quotient digit q^X_{i+1}: The bit-vector representation obeys that $v_j = 1$ if, and only if, $q^X_{i+1} = j$.

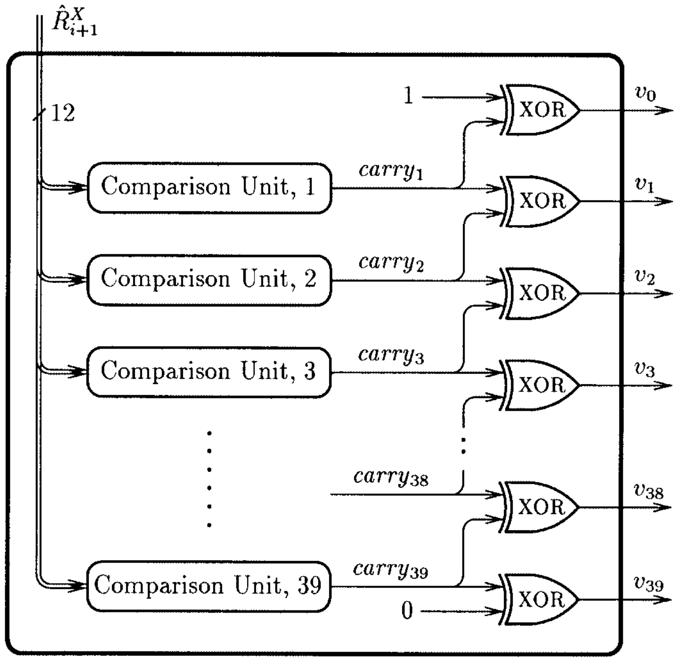


Figure 4.8: Hardware architecture of the quotient determination unit.

Each of the 39 comparison units performs one of the comparisons in (4.2). The unit denoted *comparison unit*, j performs the comparison $\hat{R}^X_{i+1} + \hat{c}_j \geq 0$.

The result of the comparison is denoted $carry_j$. Signal $carry_j$ is 1 if the answer is "true", and 0 if the answer is "false". So, according to the above discussion, the signal v_j (equal to the exclusive or of $carry_j$ and $carry_{j+1}$) takes the value 1 if, and only if, the value j should be assigned to the quotient digit. Since the intermediate operand R^X_{i+1} is restricted to the range $[0; 40m'[$, signal $carry_0$ is constantly 1, and $carry_{40}$ is constantly 0.

The hardware architecture of the *comparison unit* is shown in Figure 4.9. The unit contains a 12 bit register denoted $\hat{c}_j$ *register*. The register holds the most significant part of the comparison constant $c_j = -jm'$. A carry save represented sum of $\hat{R}^X_{i+1}$ and $\hat{c}_j$ is computed by means of a carry save adder. Finally, the sign of the sum is computed by a binary addition. The value of the carry signal, denoted $carry_j$, encodes the sign: Since $\hat{R}^X_{i+1}$ is non-negative and $\hat{c}_j$ is negative, a carry will be generated if $\hat{R}^X_{i+1} + \hat{c}_j \geq 0$. No carry will be generated if $\hat{R}^X_{i+1} + \hat{c}_j < 0$. The *binary adder* is an instance of the so-called high speed adder generated by the chip development tools. It should be mentioned, that only the resulting carry signal from the adder is needed. The binary represented sum is discarded. Therefore, a more efficient implementation of the comparison unit could be achieved by replacing the binary adder with a simpler circuit, where the functionality is limited to a computation of the resulting carry.

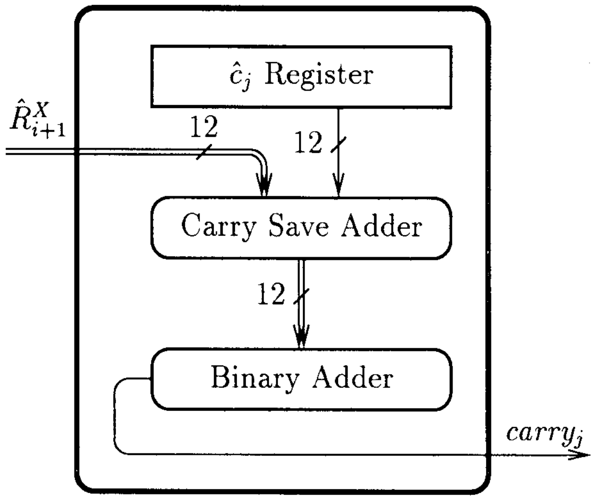


Figure 4.9: Hardware architecture of the comparison unit.

4.2.5 Control Unit

The control unit contains three *finite state machines*: The first state machine is implementing the interface protocols, where data to be exponentiated are collected in the I/O register. The second state machine is implementing the initialisation procedure, where new values of the exponent and the modulus are put into the processor, and where the $\hat{c}_j$ registers in the quotient determination unit are initialised. Finally, the third state machine controls the sequence of computations performed by the modular exponentiation unit. The following description is limited to the third state machine. The other state machines are similar. Apart from the state machines, the control unit contains some *counters*. E.g. four counters are associated with the control of the modular Exponentiation unit: Two counters are used for counting the number of exponent digits and the number of multiplier digits that have been processed in the exponentiation procedure. Furthermore, two counters are used for counting the number of wait states in the binary addition implementing the final conversion from carry save representation into binary representation. (See the description of the modular multiplication unit in Subsection 4.2.2).

The hardware architecture of the finite state machine for controlling the modular exponentiation unit is depicted in Figure 4.10. The input to the state machine is denoted *signals*. This input comprises external signals generated by the user of the processor. Moreover, it comprises internal signals generated by one of the other state machines, status signals from the counters, and the resulting sign from the binary adder in the modular multiplication unit. The output from the state machine comprises a set of *control signals* used for the internal control of the components in the modular exponentiation unit and for the control of the counters. E.g. control signals for the registers are controlling the moment of loading a new value, the moment of shifting the contents of a shift-register, or the moment of clearing the content. A set of *flags* is part of the output as well. Some of the flags are used internally in the control unit to inform the other state machines of the status of the modular exponentiation unit, and some of the flags are used for informing the user about the status of the processor.

The architecture in Figure 4.10 contains two registers. The *state register* holds the current state of the state machine, and the *signal register* holds the current values of the control signals and flags. The unit denoted *control*

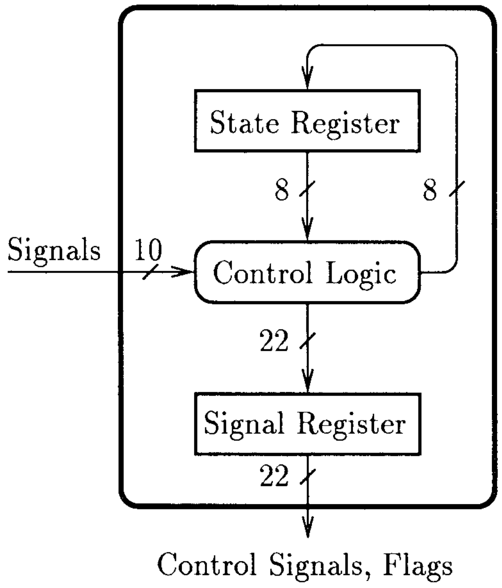


Figure 4.10: The finite state machine for controlling the modular exponentiation unit.

logic computes the next state of the state machine and the next value of the signal register. In this computation the current state and the current value of the input signals are taken into account.

For the implementation of the finite state machines, the synthesis tool FINESSE was applied. The functionality of the finite state machines was specified in a special purpose programming language. From this specification FINESSE generated a circuit module containing flip-flops for the implementation of the registers, and instances of basic combinatorial cells from the cell library for the implementation of the control logic. The finite state machine in Figure 4.10 comprises 30 flip-flops and 88 instances of the basic combinatorial cells.

4.3 Layout

The floorplan of the layout of the processor is illustrated in Figure 4.11, and a photography of the die of the fabricated processor is shown in Figure 4.12.

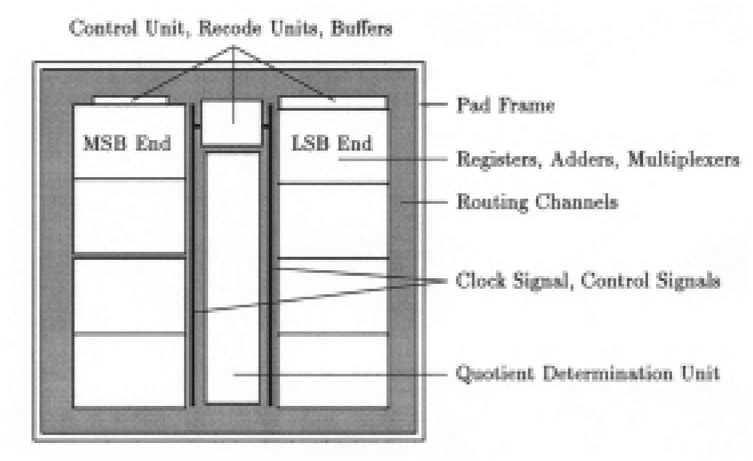

Control Unit, Recode Units, Buffers
MSB End
LSB End
Pad Frame
Registers, Adders, Multiplexers
Routing Channels
Clock Signal, Control Signals
Quotient Determination Unit

Figure 4.11: Floorplan of processor.

The three upper blocks of the floorplan contain all of the modules generated by the FINESSE synthesis tool: The control unit (including some counters) and the recoder units for the recoding of multiplier digits and quotient digits. Furthermore, these blocks contain buffers (or drivers) for driving the clock signal and the control signals. As indicated in Figure 4.11 these signals are distributed on each side of the quotient determination unit. The very wide circuitry consisting of registers, adders and multiplexers are laid out in bit-slices. Because of the width of this circuitry (about 576 bit-slices), it is divided into eight blocks of 72 bit-slices each and folded around the quotient determination unit. Some parts of this circuitry are not exactly 576 bits wide. Then, some of the bit-slices in the most significant (MSB) end, or in the least significant (LSB) end, are left unused. Each of the eight blocks of bit-slices, as well as the quotient determination unit, contains a second level of buffers for driving the clock signal and the control signals. The grey-shaded area inside the pad frame symbolises routing channels. The vast majority of this area is used for routing of power supply wires. The pad frame contains 111 pads, of which 32 are used for the power supply. Each side of the pad frame contains four Vdd pads and four GND pads.

The process technology used for the implementation of the processor is a 5 V, 1.2 μm, double metal layer, CMOS process technology, named ECPD12 [ES293], from the company European Silicon Structures (ES2). The exact dimension of the layout, including the pad frame, is 14,112.9 μm times

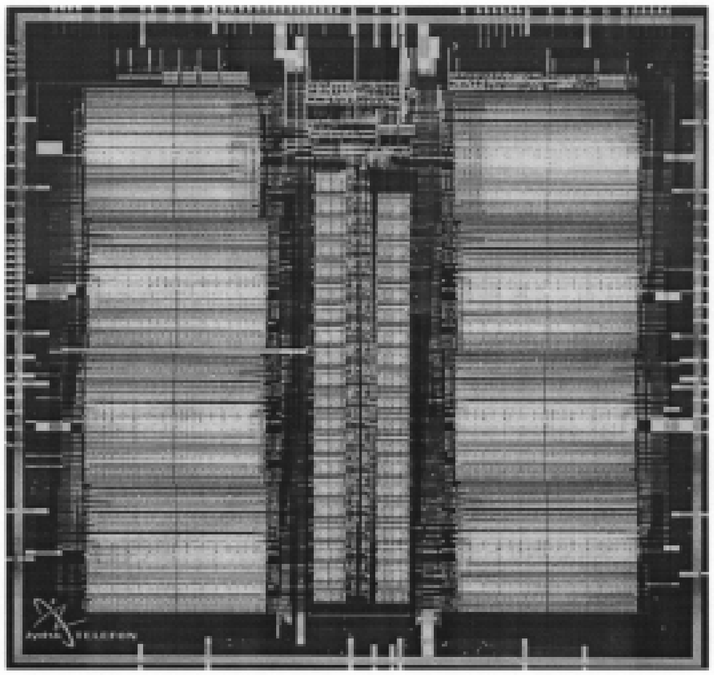


Figure 4.12: Photography of the die.

15,045.3 μm. This gives a total area of approximately 212 mm^2. The processor comprises 126,864 p-transistors and 177,113 n-transistors, a total of 303,977 transistors.

4.4 Test and Performance

For testing the functionality of the processor, and for measuring the performance, a series of experiments was done. The results reported in this section are based on experiments with a sample of 100 chips. According to the chip manufacturer, the company ES2, the expected yield should be in the range from 7 to 15 percents. So, it was hoped to find 7–15 well-functioning chips. Indeed, it turned out that 8 chips behaved in accordance to the specification of the design of the processor. In the following, these 8 chips will be denoted the “working chips” while the remaining 92 chips will be denoted the “failing chips”.

4.4.1 Check of Pin Connections

The very first experiment was a check of the pin connections. This was done to ensure that the actual connections of the pins were in accordance with the specifications. (The chips were packed and bonded by the manufacturer). The experiment was done by a series of measurements of the resistance between the pins of the package:

1. It was checked that all Vdd pins were electrically shorted internally at the chip. Similar for the GND pins. Furthermore, it was checked that the Vdd and GND pins were electrically isolated from each other.

2. It was checked that all of the other pins were isolated from the Vdd pins and from the GND pins.

The pin positions are described in Appendix E. The check of the pin connections was limited to a single chip. No errors in the connections were found.

4.4.2 Current Measurements on Reset

The next experiment was a measurement of the current consumption of the 100 chips. The purpose of the experiment was to check if the chips consumed a reasonable amount of power when clocked at varying frequencies. Furthermore, the responses of the chips on a *reset procedure* were observed: First, the voltage levels of four output pins were measured while the reset signal was activated. Then, the voltage levels were measured while the reset signal was deactivated. This gave a first indication of the functionality of the chips.

A simple test board, using an adjustable clock generating circuit, was built. The frequency of the system clock signal was set up via the parallel interface port from a Personal Computer (PC). Except from the reset signal and the system clock signal, all of the input pins were connected to either Vdd (5 V) or GND (0 V), giving a proper configuration of the processor. The processor was configured to use the so-called general purpose (GP) interface.

1. The measurements of voltage levels were done on the four output pins denoted by outputData, doneKey, doneExp and errorsync in the description in Appendix E. Pin *outputData* is the data output from the

I/O register. This pin is the only tristate output pin of the processor. *DoneKey* is a flag that signals the end of an initialisation procedure. This procedure is performed by the processor each time new values of the modulus and the exponent have been shifted into the processor. *DoneExp* is a flag that signals the end of an exponentiation process. Finally, *errorSync* is a flag that signals the detection of an error in the synchronisation pattern used in the self-synchronising SLD interface (see Appendix E). The correct voltage levels of the pins are shown in Table 4.1. Note that pin outputData is in the high impedance state, denoted by Z, when the reset signal is active. (The reset signal is active at a low voltage level). To check the high impedance of the outputData pin, the voltage level was forced to high through a pull up resistance and, similarly, forced to low through a pull down resistance.

Reset	OutputData	DoneKey	DoneExp	ErrorSync
0	Z	0	5	5
5	0	5	5	0

Table 4.1: Correct voltage levels of observed output pins.

2. Simultaneously with keeping the reset signal at the active level, the current consumption was measured at three clocking frequencies, 5 MHz, 10 MHz and 20MHz. During an activation of the reset signal, the control unit is brought into a well-defined state. In this state, some of the registers are cleared, and all of the remaining registers are holding their values. Consequently, the current consumed by the combinatorial circuitry between the registers will be minimal and, therefore, the influence on the current measurements from the (unknown) values in the registers after power-up will be relatively low.[2] It was expected to

[2]None of the combinatorial circuit cells are using dynamic logic families, where an internal node is pre-charged before the evaluation of the output values. Therefore, the current consumed by these cells is independent of the actual input values. However, the registers are using a type of D flip-flops [YS89], where an internal node is pre-charged in each clock period. Depending on the actual value held by the flip-flop, this internal node will be discharged in each clock period as well. In fact, if the value held by the flip-flop is zero, this internal node will alternately be charged and discharged. If the value is one, the node will remain charged. So, to some extent, the current measurements will be

see a linear relationship between the current consumption i_{total} and the clocking frequency f_Ω of the form

$$i_{\text{total}} = i_{\text{static}} + f_\Omega \cdot i_{\text{dynamic}} \tag{4.4}$$

The static current i_{static} is the contribution from leakages in the circuitry and from the output pins of the processor. Furthermore, there is a contribution from the test board. (This contribution was measured, without a processor in the test board, to be 256 μA when the reset signal was activated, and 40 μA when deactivated.) The dynamic current i_{dynamic} is the contribution from the circuitry for distribution of the clock signal and from the flip-flops clocked by this signal. The current supply to the external circuit for generating the clock signal was separated from the supply to the remaining parts of the test board. Hence, the current contribution from the clock generation circuit was excluded from the measurements.

3. Finally, after deactivating the reset signal, the current consumption was measured at a 20 MHz clocking frequency. This current is a measure of the power consumption when the processor is idle, i.e. in a state where the processor is ready for further processing. The internal operation of the processor, just prior to the event of deactivating the reset signal and just after this event, only differs in one respect: No registers are being actively cleared. However, since all registers are holding their values, the cleared registers will remain cleared.

The current measurements, and the observation of the voltage levels of the four output pins, implied that 17 chips were classified as failing chips: All of these chips consumed a current, that was significantly larger than the other 83 chips. Furthermore, some of the failing chips had erroneous voltage levels at the output pins. The difference in current consumption was most significant for the measurements at the 5 MHz clocking frequency, and least significant

influenced by the power-up state of the registers. Some of the registers are using a flip-flop variant that can be cleared, i.e. the value held by such a flip-flop can be forced to zero by activation of a "clear" input signal. When this signal is activated, the above mentioned internal node will remain discharged throughout a clocking period, and the pre-charging is disabled. Approximately 1,200 of the flip-flops have this clear option. When the external reset signal is activated, the clear signal for these flip-flops is activated, and when the reset signal is deactivated, the clear signal is deactivated, as well.

for the 20 MHz frequency. This indicated a large static current consumption for the failing chips, and it may be a symptom of, unintentionally, shorted circuitry in the chips.

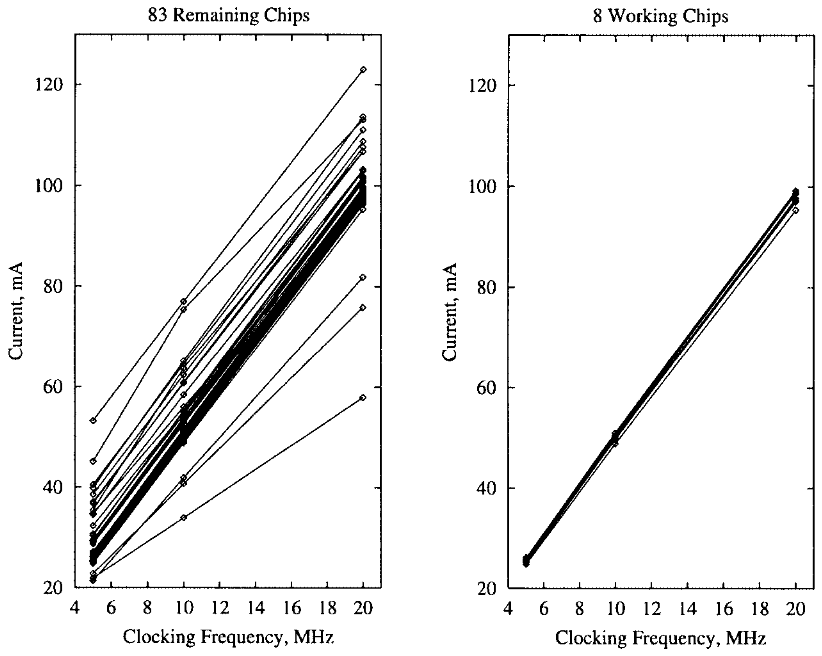


Figure 4.13: Plot of current as function of clocking frequency.

Two plots of the current measurements, with an active reset signal, for the remaining 83 chips are shown in Figure 4.13. The plot on the left comprises all 83 chips, while the plot on the right is limited to the data for the 8 working chips. Except from two sets of data in the left plot, all sets of data show an approximately linear relation between the current consumption and the clocking frequency. Furthermore, it is seen that the slopes of the lines are almost identical for the vast majority of the chips. So, assuming that the current consumption is given by expression (4.4), it is seen that the variation in current consumption of the chips are mainly due to variations in the static current consumption. The plot of the currents for the working chips shows a small variation in the static current consumption, and a small variation in the dynamic current consumption.

The current measurements for the 8 working chips are listed in Table

Chip No.	5 MHZ	10 MHZ	20 MHZ	20 MHZ, non-reset	i_{static}	i_{dynamic}
	mA	mA	mA	mA	mA	mA/MHz
01	25.2	49.7	97.0	107.6	1.6	4.8
11	25.7	50.5	98.7	109.4	1.6	4.9
15	24.8	48.8	95.4	105.8	1.5	4.7
37	25.7	50.5	98.6	109.4	1.7	4.9
53	25.3	49.7	97.2	107.8	1.6	4.8
63	25.6	50.4	98.6	109.3	1.5	4.9
66	26.1	51.0	99.2	110.0	2.0	4.9
89	25.4	50.0	97.7	108.4	1.6	4.8

Table 4.2: Current consumption of working chips

4.2. The first column contain the identification numbers of the chips. The next three columns lists the current measurements when the reset signal was at the active level. The fifth column contains the single measurement done when the reset signal was deactivated. Finally, in the last two columns, some estimated values of the static current and of the dynamic current are computed. These values are obtained by fitting a line, expressed by (4.4), to the three measurements of the current when the reset signal was active. It is seen that the static current for the working chips is in the range from 1.5 mA to 2.0 mA, and the dynamic current is in the range from 4.7 mA/MHz to 4.9 mA/Hz. A similar computation for the other $83 - 8 = 75$ chips confirms that the dynamic current is about equal for the vast majority of the chips: Of the 83 chips, 75 have a dynamic current in the same range as the working chips. The static current of the 83 chips shows a much larger variation: It varies from 1.4 mA to 30.2 mA, and the number of chips, having a static current in the same range as the working chips, is 37. Hence, it seems like this relatively simple measurements of currents might be a suitable method to classify the sample of chips into a class of candidates for working chips and a class of definitely failing chips. In this case, the method would classify approximately 60 percents of the chips as failing.

According to measurements at the 20 MHz clocking frequency in Table 4.2, the current consumption increases by 10.4–10.8 mA when the reset signal is deactivated. This is due to the internal operation of about 1,200 flip-flops that are no longer being actively cleared and, therefore, consume current through a process of pre-charging and discharging an internal node.

4.4.3 Test Board

For testing the functionality and for measuring the performance of the processor a new test board was built. The construction of this board was much more complicated than the board used in the previous current measurements. It should be mentioned, that both test boards were built by John Thorup, the Research and Development Department, Jydsk Telefon/Tele Danmark.

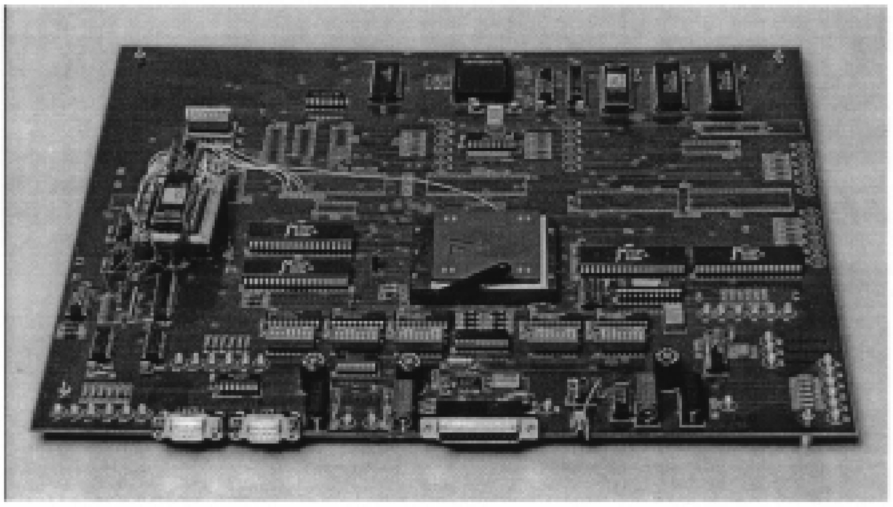


Figure 4.14: Photography of the test board.

Figure 4.14 shows a photography of the new test board. The large chip, placed in a socket near the centre of the board, is the processor to be tested. At the rear a micro-processor (in the following denoted by μP) is located. It controls the communication with the processor. At each side of the processor a set of FIFOs (First In, First Out queues) is placed. The FIFOs are holding the data to be serially shifted into, and out from, the processor. In front of the processor a set of switches is located. The switches are used for setting those of the configuration parameters, that are seldom changed. The remaining configuration parameters can be set via the μP. A circuit for generating an adjustable clocking signal is included on the board. The clocking frequency is controlled via the parallel interface port from a PC. The clock generation circuit provides two adjustable clocking signals for the processor: One for the system clock input pin, and one for the data clock input pin. The latter clocking signal determines the frequency by which data are shifted into, and out from, the I/O register. It is possible to choose a system clock frequency in the range from 320 KHz to 120 MHz, and a data clock frequency in the range from 160 KHz to 60 MHz. The test board is also equipped with a number of light-emitting diodes (LEDs). These monitor the status of the FIFOs (full, empty, etc.) and the values of the processor's output flags. As seen from the photography, several components have not been mounted on the test board. The board was prepared for supporting a scan-chain test. However, it turned out that a test of the functionality of the normal operation modes was adequate. Hence, the components for the scan-chain test were never mounted and, consequently, it is not known if the test mode of the processor is functioning.

A testing procedure can, briefly, be described as follows: After powering up the test board, a program denoted a "command interpreter" is loaded to the program memory of the μP. The command interpreter is loaded via one of the serial interface ports from the PC. The test to be performed is specified as a sequence of test commands. This specification has the form of a simple text file on the PC. During the testing procedure a series of test commands, including data, are transmitted to the μP from the PC via another serial interface port. Similarly, after execution of each test command, the result is returned to the PC. Then, the PC is used for comparing the result of the test with the expected, correct, result.

Apart from the command interpreter to be executed by the μP, some programs were written for implementation of the communication protocols between the test board and the PC. Furthermore, a simple test command "language" was specified. A parser/interpreter for checking the syntax and for interpreting the test command files was written. Figure 4.15 shows the listing of a typical test command file. The comments explain the meaning of the commands.

4.4.4 Test of Functionality

The testing of the functionality of the processor was carried out by means of a set of about 35 test command files. Apart from inspection of the results of the modular exponentiation, it was checked if the output flags behaved in accordance to the specifications. The data to be exponentiated and the values of the keys (i.e. the modulus and the exponent) were chosen from two categories: The first category was a set of "extreme" values. For example, the

Test Command File	Comments
`clock 0.320 2`	Set data clock to 320/2 KHz, and system clock to 2 MHz.
`crypt`	Set configuration to "crypt" mode.
`gpi`	Set configuration to GP interface.
`resetRSA`	Activate reset signal, and
`resetRSAno`	then deactivate reset signal.
`put key key002.bin`	Copy key value in file "key002.bin" to key FIFOs.
`start key`	Using the contents of the key FIFOs, perform an initialisation of the processor registers holding the modulus and the exponent.
`put gpi gpi000.bin`	Copy data value in file "gpi000.bin" to input FIFO.
`start gpi`	Shift data from input FIFO into I/O register and, simultaneously, shift contents of I/O register into output FIFO.
`get gpi dummy.bin`	Empty output FIFO by moving content to file "dummy.bin".
`startExp`	Generate a pulse on the startExp input pin. Hereby, the contents of I/O register is swapped with the result in the modular exponentiation unit, and an exponentiation process is started.
`startExp`	As above.
`start gpi`	As above.
`get gpi result.bin`	Move result from output FIFO to file "result.bin".

Figure 4.15: Example of a test command file.

smallest, and the largest, allowable value of the exponent and the modulus. The second category was a set of randomly selected values. Furthermore, the second category contained a number RSA key pairs. Hereby, some of the tests comprised a real encryption and a decryption of a series of data values. To check the results of the modular exponentiation a "reference" program was used. The reference program was *not* an implementation of the modular exponentiation method used by the processor. Instead a standard library routine was used. Hereby, it was ensured that errors in the method would not be inherited by the reference program.

The clocking frequencies chosen for the functionality test were rather low: 160 KHz for the data clock and 2 MHz for the system clock. This was done to ensure that failures would not be due to a too high clocking frequency. Later, when the working chips were found, the clocking frequencies were increased to measure the performance of these chips. The results of the performance measurements are reported in the next subsection. The following tests were performed:

1. During a process of initialising the key, the response of the doneKey flag was inspected. After reseting the processor, the flag must be high. Then, when the initialisation process is started, doneKey must go to a low level. Finally, when the key values have been shifted into the processor, and the processor have completed the internal initialisation process, the flag must go back to the high level.

2. Using the relatively simple GP interface, the functionality of the processor was tested. First, it was checked if data could be shifted into, and out from, the I/O register. Then, it was checked if an exponentiation process indeed was performed when the startExp input signal was activated. On an activation of the startExp signal, the doneExp flag must go low, and when the processor have completed the exponentiation, the flag must return to high. The correctness of the results of the exponentiation process was verified.

3. For the chips that passed the first two tests, a set of RSA encryption operations and decryption operation were performed. The idea of the test was to encrypt a large amount of random data and, then, after a decryption, to check if the data had changed. For each pair of encryption/decryption keys a total of 64,000 data blocks were processed. However, instead of shifting 64,000 data blocks, each consisting of 561 bits, through the processor, a 64,000-fold encryption, followed by a 64,000-fold decryption, of a single random data block was performed. Since the outcome of an encryption must look like a random number, this test was comparable to shifting 64,000 random data blocks through the processor. Furthermore, if any of the intermediate results were erroneous, the final result would not be identical to the original data block. During these tests, the frequency of the system clock was increased to 25 MHz, and the frequency of the data clock was increased to 512 KHz.

4. The self-synchronising SLD interface was tested. Since the former tests would reveal failures in the operation of the modular exponentiation unit, this test was primarily aimed at a verification of the functionality

of the various configuration possibilities of the SLD interface. The SLD interface is described in Appendix E.

The test of the functionality implied that 8 of the 83 remaining chips were classified as working. Using the GP interface no failures were detected in the functionality of these 8 chips. There was, however, detected a failure in the expected functionality of the SLD interface. The failure was due to an error in the specification of the finite state machine that implemented the SLD interface protocol. So, the failure was reproduced in all of the 8 "working" chips. Fortunately, the GP interface was included in the design of the processor. Hence, it was still possible to use the processor for computing modular exponentials. As a consequence of the failing SLD interface, the processor was never embedded in an ISDN telephone.

The failure was detected when the processor was configured to the *transmit and crypt mode*. In this mode, the resulting 561 bits modular exponential is merged with a 79 bits synchronisation pattern, giving a total of 640 bits. During a communication on an ISDN channel, these 640 bits is divided into 80 frames of 8 bits. Except from one frame, each of the 80 frames contains a synchronisation bit. In Table 4.3 the correct format of the output from the processor is shown together with the erroneous format actually produced by the chips. The 561 bits of the modular exponential is denoted $c_{560}c_{559}\dots c_0$, where c_0 is the least significant bit. As seen in the table, the synchronisation pattern is correct in the erroneous format. Furthermore, the format of the frames numbered from 1 to 79 is correct. The failure is in the last frame: Here a shift of the I/O register is missing in the specification of the controlling state machine. The consequence is that bit c_{553} is represented twice in the output format and that the most significant bit c_{560} is lost.

4.4.5 Performance Mesurements

The performance of the processor was measured while it was configured to use the GP interface. It was planned to do the measurements under varying external conditions for the processor, i.e. for varying supply voltages and for varying temperatures. However, it turned out that some of the components on the test board were quit sensitive to these conditions. So, the actual variations were limited to a minor variation in the supply voltage. All of the measurements were performed at room temperature. In the following, the results of the measurements are listed:

Frame	Correct Format	Erroneous Format
1	1 c_{006} $c_{005} \dots c_{000}$	1 c_{006} $c_{005} \dots c_{000}$
2	0 c_{013} $c_{012} \dots c_{007}$	0 c_{013} $c_{012} \dots c_{007}$
3	0 c_{020} $c_{019} \dots c_{014}$	0 c_{020} $c_{019} \dots c_{014}$
$\vdots$		
78	0 c_{545} $c_{544} \dots c_{539}$	0 c_{545} $c_{544} \dots c_{539}$
79	c_{553} c_{552} $c_{551} \dots c_{546}$	c_{553} c_{552} $c_{551} \dots c_{546}$
80	0 c_{560} $c_{559} \dots c_{554}$	0 c_{559} $c_{558} \dots c_{553}$

Table 4.3: Correct and erroneous format of output in transmit and crypt mode.

1. The minimal number n_w of wait states required by the binary adder in the modular multiplication unit, see Subsection 4.2.2, was found. Since n_w is increasing for increasing clocking frequencies, the number was found simultaneously with the maximal allowable frequency of the system clock. It turned out that the functionality of the 8 working chips was preserved for $n_w \geq 6$ when they were clocked at their maximal frequency at a supply voltage of 5.0 V.

2. At a supply voltage of 5.0 V, the maximal allowable frequency f_Ω of the system clock for the 8 chips varied from 26 MHz to 28 MHz. The maximal frequencies are shown in Table 4.4. There seems to be no correlation between these frequencies and the measurements of the current consumption during a reset operation in Subsection 4.4.2.

Chip No.	01	11	15	37	53	63	66	89
f_Ω (MHz)	28	28	27	27	28	26	27	27

Table 4.4: Maximal allowable system clock frequency for working chips.

3. The relation between the maximal allowable frequency and the supply voltage was measured for a single chip. The chip used in the experiment had identification number 66. For supply voltages less than 5.0 V the minimal number of wait states turned out to be 7. For voltages greater then or equal to 5.0 V the minimal number was 6. The result

State	Current (mA)	Power (W)
Idle	150	0.8
Busy	490	2.5

Table 4.5: Power consumption at 25 MHz.

of the experiment is illustrated by the plot in Figure 4.16. At the ± 10 percents voltage levels, i.e. at 4.5 V and 5.5 V, the maximal frequencies were 24.0 MHz and 29.5 MHz.

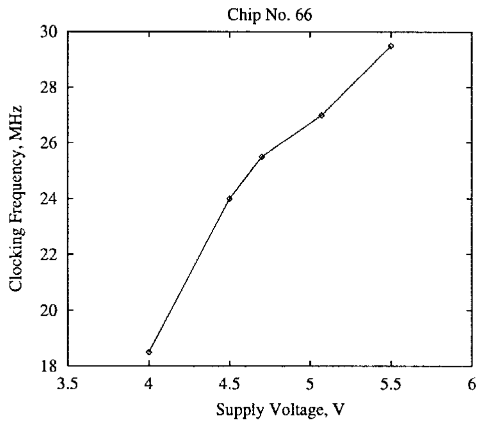


Figure 4.16: Maximal clocking frequency as function of supply voltage.

4. The current consumption of chip no. 66 was measured. A measurement was done while the processor was *idle*, i.e. the processor was waiting for new data to exponentiate. When idle, the doneExp flag is high. Furthermore, a measurement was done while the processor was *busy*, i.e. a modular exponentiation was being executed. When busy, the doneExp flag is low. The clocking frequency was 25 MHz, and the supply voltage was 5.0 V. Table 4.5 shows the current and the corresponding power consumption. The values of the current consumption in the table are equal to the measured values minus 250 mA, which was the current consumed by the test board without a processor in the socket.

5. The time for computing a modular exponential was measured. When

an exponentiation process is started by an activation of the startExp signal, the flag doneExp is pulled low by the processor. When the exponentiation is completed, the doneExp flag is returned to high. Hence, the computing time can be measured as the period from activation of the startExp signal to the moment where the doneExp signal is raised. This was done at a 25 MHz clocking frequency. The computing time was measured to be 5.47 ms. Hence, in an encryption application, the processor is able to encrypt at a rate of 561 bit per 5.47 ms, corresponding to about 102 Kbit/s. In Figure 4.17 the response of the doneExp flag is shown when the startExp signal is activated. Furthermore, the figure shows the measured times. The delay, from activating startExp until doneExp is pulled low, was measured as well. It was 188 ns at the 25 MHz clocking frequency.

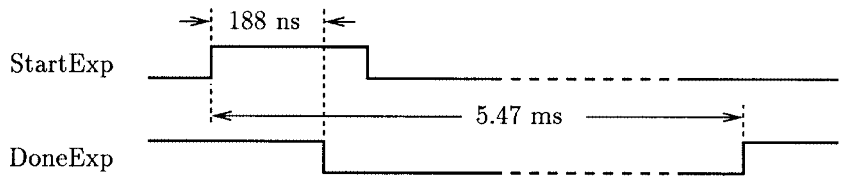


Figure 4.17: Measurements of computing time.

The time for computation of a modular exponential is proportional to the clocking period t_Ω. The clocking period is equal to the reciprocal value of the clocking frequency f_Ω. Furthermore, the time depends on the parameter setting of the processor: The number of exponent bits n_j, the number of radix 32 multiplier digits n_i, and the number of wait states n_w. In the above measurements, the parameter setting was $n_j = 561$, $n_i = 115$, and $n_w = 7$. In general, the computing time t_{exp} for a modular exponentiation operation can be approximated by the expression

$$t_{\text{exp}} = 2 \cdot (n_i + n_w) \cdot n_j \cdot t_\Omega. \tag{4.5}$$

4.5 Summary and Discussion

The data in Table 4.6 summarises the physical dimensions, the power consumption, and the performance of the modular exponentiation processor.

Technology	1.2 μm, 2 metal layers
Die area	212 mm^2
Transistor count	303,977
Power, at 25 MHz:	
— idle	0.8 W
— busy	2.5 W
Maximal clocking frequency:	
— at 5.0 V, room temperature	26.0 MHz
Computing time:	
— 561 bit operands, at 25 MHz	5.5 ms
Throughput:	
— 561 bit operands, at 25 MHz	102 Kbit/s

Table 4.6: Summary of data for the modular exponentiation processor.

One of the main requirements of the processor was a minimal throughput of 64 Kbit/s. As seen from the table there is a comfortable margin from this requirement to the actual maximal computing rate of 102 Kbit/s. Even though the performance, listed in the table, is at the condition of 5.0 V supply voltage and room temperature, it is believed that there will be no problems in meeting the requirement of 64 Kbit/s at the worst case conditions of 4.5 V and 85 °C: According to Equation (4.5) the throughput for 561 bit operand exponentiations are greater than 64 Kbit/s when the clocking frequency is greater than or equal to 16 MHz. An *indication* of the effect of simultaneously raising the temperature and lowering the voltage can be obtained from the performance measurements of a division chip reported by Williams and Horowitz in [WH91]. Here, the circuit delay increases from 2.8 ns at 5.0 V and 35 °C to 3.9 ns at 4.5 V and 125 °C. This corresponds to a performance degradation to about 70 percents. Hence, it is reasonable to estimate the frequency for the modular exponentiation processor at 4.5 V and 85 °C to be greater than $0.70 \cdot 26$ MHz, which is greater than 18 MHz.

Although a design error in the SLD interface was revealed, the project of implementing the modular exponentiation processor can be considered as a success: *It has been verified that it is possible to construct a single-chip processor, which performs real-time RSA encryption of data transmitted via a 64 Kbit/s ISDN channel.* According to the best knowledge of the author of this thesis, the processor is still *the fastest performing single-chip implementation for computation of modular exponentials.* Only one implementation has been reported to be faster: At Digital Equipment Corporation Paris Research Laboratory, Shand and Vuillemin [SV93] have made a 600 Kbit/s RSA encryption implementation, for 512 bit keys, using a so-called Programmable Active Memory (PAM) and a workstation. The PAM [BRV89, BRV93, VBR$^+$94] is a kind of universal hardware coprocessor based on a board of 23 field programmable gate arrays (FPGA). When the PAM is performing an RSA encryption the clocking frequency is 40 MHz. The PAM implementation uses a β-ary exponentiation method, where $\beta = 2^5$ (see Subsection 2.1.1), and a radix 4 modular multiplication method. It should be mentioned that the PAM implementation utilises the Chinese Remainder Theorem (see Section 2.4), which accounts for a speedup of about 4 when compared to the processor described in this chapter. So, for the general computation of modular exponentials, where either the prime factors of the modulus are unknown or the modulus is a prime, a throughput of about 150 Kbit/s for the PAM implementation may be expected.[3]

The modular exponentiation processor described in this chapter is designed to use 561 bit keys. Usually, when comparing the speed of modular exponentiation implementations, a key length of 512 bit is assumed. If the processor had been designed for this key length, the computing time would, according to (4.5), be about $2 \cdot (105 + 7) \cdot 512 \cdot 40$ ns, which is about 4.59 ms when a clocking frequency of 25 MHz is assumed. This corresponds to a throughput of more than 111 Kbit/s. It is 3.8 times faster than the speed

[3]According to a personal communication on July 4 1995 with Mark Shand, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, the throughput of 600 Kbit/s was, in fact, never *measured* for the PAM implementation. However, a throughput of 185 Kbit/s was measured for an implementation using a 970 bit key length. Therefore, if the effect of applying the Chinese Remainder Theorem is removed, the performance of a general computation of 970 bit modular exponentials is expected to be about 46.25 Kbit/s. Furthermore, since the throughput for such a general exponentiation is expected to be proportional to the reciprocal of the key length, the estimated throughput for 512 bit key lengths is approximately 88 Kbit/s. So, based on the comparisons of the *actual performance measurements* of the PAM implementation and the processor, it is fair to state, that the processor *is the fastest known implementation for computing modular exponentians, when no assumptions about the prime factors of the moduli are imposed.*

obtained by the board from Thorn EMI (without utilisation of the Chinese Remainder Theorem). The Thorn EMI board uses a clocking frequency of 24 MHz. Although an explicit description of the methods for modular exponentiation and for modular multiplication is missing in the data sheet for the Thorn EMI board [Tho88], there are some indications: It seems like a radix 2 modular multiplication method and a sequential binary exponentiation method are used. Furthermore, it seems like the throughput of 29 Kbit/s is for the average case, i.e. when the exponent e consists of $\nu(e) = 256$ non-zero bits (see Section 2.1). Then, for the worst case, where $\nu(e) = 512$, the throughput will be about 22 Kbit/s. So, for the worst case, the processor described in this chapter is about five times faster than the Thorn EMI board. Both implementations are clocked at about 25 MHz. The comparison illustrates the net effect of the computation methods used by the modular exponentiation processor:

- The parallel right-to-left binary exponentiation method improves the worst case computing time by a factor of two.
- The radix 2^5 modular multiplication method reduces the number of recursion cycles by a factor of five.
- The time for a recursion cycle is increased by a factor of two. Recall that the architecture has been pipelined into two stages. Hence, the time for a recursion cycle corresponds to two clocking periods. The increased recursion cycle time is due to the more complex quotient determination unit and to the more complex units for computation of multiples.

The resulting processor is a good illustration of the dilemma of the high-radix modular multiplication methods: By choosing a higher radix the number of recursion cycles is reduced. This does, hopefully, result in a reduction in the computing time. However, simultaneously with increasing the radix, the circuit complexity is increasing and, consequently, the recursion cycle time is increasing as well. In the next chapter it will be shown how a high-radix modular multiplication method *without this dilemma* can be developed. The method is a high-radix generalisation, and improvement, of the method used by Shand and Vuillemin for the PAM implementation.

It is worth noting that the processor was found well functioning after a single fabrication run. That means, there was no problems with the electrical properties of the chip or with the functionality of the modular exponentiation unit. Apart from a piece of luck, the reasons for this successful outcome can be attributed to:

- An extensive support provided by the chip development system. The chip designers were inexperienced with the design of large chips. So, probably, many pitfalls were avoided by remaining in the consistent environment of ChipCrafter, and by utilising the automatic tools for generating circuit modules, and for doing the work of circuit placement and routing. Finally, the analysis tools were of great help when decisions between various design tradeoffs were taken.

- The way of structuring the architecture. The architecture was structured as a hierarchy of units, where each unit had a well-defined functionality. Each unit was validated, in a bottom-up fashion, through a series of simulations. Then, when some units were included in the architecture of another unit at a higher hierarchal level, the focus was restricted to the functionality obtained by combining the "black-box" functionality of the lower level units. The strategy was to maintain an overview of the architecture, and the functionality, through the use of abstraction mechanisms.

- Simulations and theoretical verification. The algorithms describing the methods for exponentiation and modular multiplication were verified through formal proofs. Furthermore, the methods were verified through simulations of functional models of the architecture. Although simulations give an insight into the functionality of the design, it is not possible to obtain a full verification of the correctness. The correctness of an algorithmic description of the methods can be proved. However, since a proof is limited to the properties of an *abstract model* of the chip design, it must be supplemented with simulations.

  At the transistor level, user defined leaf cells were verified through extensive and detailed circuit simulations. Furthermore, the characteristics of the cells were compared to similar cells from the library of cells provided by the development system.

- Careful and disciplined style of work. Each time the design was modified, the validity was checked again. Furthermore, whenever possible, the design was double checked: E.g. the transistor netlist obtained from an extraction of the layout was compared to the netlist obtained from the schematic description. The quality of some parts of the power supply net was analysed through circuit simulations. Moreover, the quality of the clock signal distribution was checked.

The penalty for sticking to the ChipCrafter development system was a relatively large area consumption. Indeed, the problems with the area consumption implied that the project was delayed by at least a year. However, the lack of experience with design of large chips influenced the time for developing the processor as well.

An area consumption of 200 mm^2 for a 300,000 transistor chip is quit high when compared to (old) state-of-the-art designs: The Intel 80386 16 MHz micro-processor from 1985 was built using a 1.5 μm double metal layer CMOS process. The 80386 consisted of 275,000 transistors and consumed an area of 95 mm^2 [Bak90, Section 9.2]. Later, in 1989, a 33 MHz version was built using a 1.0 μm double metal layer process. In this version the area was reduced to 43 mm^2. Similarly, the Motorola 68030 from 1987 was built using a 1.2 μm double metal layer CMOS process. It consisted of 270,000 transistors and occupied an area of 55 mm^2. Therefore, an acceptable area consumption, and yield, for the modular exponentiation processor can be achieved through a redesign of the circuitry, where a full custom approach is followed, and a modern process technology is used.

Chapter 5

Montgomery Residues

In 1985 Montgomery [Mon85] proposed a new kind of modular multiplication method. The idea behind Montgomery's method is to obtain a quotient digit determination that is more efficient than those for the traditional modular multiplication methods described in Chapter 3. Because Montgomery's method requires some additional pre- and post-processing, it is best suited for applications where several modular multiplication are done using the same modulus. Hence, an application that may benefit from Montgomery's method is the computation of modular exponentials with very large operands.

Montgomery's contribution can be viewed as another way of representing residues modulo m, plus an efficient method for performing modular multiplication in the domain of this new representation. Kornerup [Kor93b] denotes the new residue representation for an *M-residue* after Montgomery. The M-residue of an integer x is equal to $(xr^n) \bmod m$, where r is a constant for the computation and n is the number of multiplier digits that are processed in the multiplication method. The value of r is chosen such that division and modular reduction by r becomes simple. Usually, r is chosen to be equal to the radix of the multiplication method, i.e. $r = 2^k$. Montgomery's multiplication operation, denoted by $*^M_{\mathbb{Z}_m} : \mathbb{Z}_m \times \mathbb{Z}_m \mapsto \mathbb{Z}_m$, can be defined by

$$a *^M_{\mathbb{Z}_m} b = (a \cdot b)r^{-n} \bmod m, \tag{5.1}$$

where r^{-n} denotes the multiplicative inverse of r^n modulo m. The product of multiplying the M-residue of a, say $x = (ar^n) \bmod m$, and the M-residue of b, say $y = (br^n) \bmod m$, is a new M-residue which, indeed, is the M-residue of $(a \cdot b)$:

$$x *_{\mathbb{Z}_m}^{M} y = (ar^n \cdot br^n)r^{-n} \bmod m = (a \cdot b)r^n \bmod m.$$

Hence, Montgomery's multiplication operation replaces ordinary modular multiplication in the domain of M-residue representations. To ensure the existence of the multiplicative inverse $r^{-n} \pmod m$, it is required that r and m are relatively prime, i.e. $\gcd(r, m) = 1$. When r is a power of two, this requirement is fulfilled if m is restricted to odd values. In cryptographic applications the modulus is always an odd value, so the requirement does not limit the usefulness of Montgomery's multiplication method for the application area considered in this thesis.

$$\begin{tikzcd}
b,\ e,\ m \arrow[r, bend left, "x = b *_{\mathbb{Z}_m}^{M} (r^n)^2"] & x = br^n \bmod m,\ e,\ m \arrow[d, bend left, "\text{Compute } y = x^e\text{, using the multiplication composition } *_{\mathbb{Z}_m}^{M} \text{ with the neutral } r^n \pmod m."] \\
z = b^e \bmod m & y = b^e r^n \bmod m \arrow[l, bend left, "z = y *_{\mathbb{Z}_m}^{M} 1"]
\end{tikzcd}$$

Figure 5.1: Modular exponentiation by means of Montgomery multiplication.

Because Montgomery's multiplication operation is associative, all of the efficient exponentiation methods in Chapter 2 can be used in the domain of M-residues as well. It should be noted that the neutral for the multiplication operation is the M-residue of 1, i.e. $r^n \bmod m$. Figure 5.1 illustrates how a modular exponentiation may be accomplished in the M-residue domain. Before the exponentiation process is initiated, the base b must be transformed into an M-residue. This can be done by a single Montgomery multiplication $b *_{\mathbb{Z}_m}^{M} (r^n)^2$, where $(r^n)^2 \pmod m$ is a constant that have to be precomputed once for each change of modulus. Similarly, after the exponentiation process is completed, the resulting exponential y must be transformed back from the M-residue representation. This can also be done by a single Montgomery multiplication $y *_{\mathbb{Z}_m}^{M} 1$. Hence, the Montgomery multiplication operation serves both the exponentiation process and the additional process of transforming operands to and from the M-residue domain. Compared to an exponentiation where the traditional modular multiplication composition is used, an exponentiation using Montgomery's multiplication composition requires two extra multiplications for the transformation to and from the M-residue domain.

5.1 Montgomery Multiplication

Montgomery's method for computation of a product of the form $(a \cdot b)r^{-n}$ mod m is analogous to the right-to-left binary exponentiation method given by Equation (2.5). The method is easily generalised to a radix 2^k version. In the following formulation it is assumed that $r = 2^k$ and, hence, that r^{-1} is the multiplicative inverse of 2^k modulo m:

$$\begin{aligned}(a \cdot b)r^{-n} &\equiv_m (a_0 b + a_1 2^k b + \ldots + a_i 2^{ki} b + \ldots + a_{n-1} 2^{k(n-1)} b)r^{-n} \\ &\equiv_m ((\cdots((a_0 b)r^{-1} + a_1 b)r^{-1} + \ldots + a_i b)r^{-1} + \\ &\quad \ldots + a_{n-1} b)r^{-1} \qquad (5.2)\end{aligned}$$

It is seen that Montgomery's multiplication operation can be computed by n recursive applications of an intermediate operation of the form $S_{i+1} := (S_i + a_i b)r^{-1}$ mod m, where S_i is the result of the preceeding operation. The intermediate operand S_0 must be initialised to zero.

At first glance the computation of $(S_i + a_i b)r^{-1}$ mod m looks more complicated than the computation of the intermediate operations of the form $(2^k S_{i+1} + a_i b)$ mod m used in the traditional modular multiplication methods in Chapter 3. However, as shown by Montgomery, the additional factor r^{-1} leads to an efficient quotient determination in the modular reduction phase. The idea is to transform the residue representation, $S_i + a_i b$, into another representation, $S_i + a_i b + q_i m$, of the same residue class, such that the least significant radix 2^k digit of $S_i + a_i b + q_i m$ is zero. This implies that

$(S_i + a_ib + q_im)r^{-1}$ can be written as,

$$[(S_i + a_ib + q_im) \text{ div } 2^k] \cdot 2^k \cdot r^{-1} \equiv_m (S_i + a_ib + q_im) \text{ div } 2^k.$$

Hence, the multiplication by r^{-1} is replaced by a simple right-shift. Note that the explicit value of r^{-1} is never used in the multiplication method. The quotient determination can be formulated as,

$$\{ \text{ Determine integer } q_i \text{ such that } (S_i + a_ib + q_im) \bmod 2^k = 0 \ \}.$$

There is a unique solution q_i (modulo 2^k) to this equation. In the following, the integer m' denotes the multiplicative inverse of $-m$ modulo 2^k. The existence of m' is ensured by the restriction $\gcd(m, r) = 1$. The value of m' must be calculated each time the modulus is changed. The quotient digit q_i is determined by,

$$\begin{aligned} -q_im &\equiv_{2^k} S_i + a_ib \\ q_i &\equiv_{2^k} (S_i + a_ib) \cdot m' \end{aligned} \tag{5.3}$$

So, the quotient determination is a computation of the residue $((S_i+a_ib)\cdot m')$ mod 2^k, which involves an addition of the least significant radix 2^k digits of S_i and a_ib, followed by a multiplication by the least significant radix 2^k digit of m'. This corresponds to a k bit addition followed by a $k \times k$ bit multiplication. The resulting quotient digit can be expressed in the digit set $\{0, 1, \ldots, 2^k - 1\}$ or, if convenient, in another residue range. Kornerup uses the (nearly) symmetric digit range $\{-2^{k-1}, -2^{k-1}+1, \ldots, 2^{k-1}-1\}$ [Kor93b]. As described in Section 3.7.1 it may be advantageous to use a symmetric digit set in the computation of the multiples q_im. The same observation holds for the encoding of the multiplier digits a_i.

Using the notation introduced in Section 3.6 the intermediate operation in Montgomery's modular multiplication method can be described as,

Algorithm 5.1–1 (Intermediate operation $(S_i + a_ib + q_im)r^{-1}$)

Stimulus: $S_i,\ a_i\ b$, where $0 \le S_i < b + m$ and $0 \le a_i < 2^k$.

Response: $S_{i+1} \equiv_m (S_i + a_ib)r^{-1}$, where $0 \le S_{i+1} < b + m$.

Method:
```
$q_i := ((S_i + a_ib) \cdot m') \bmod 2^k$
    $S_{i+1} := (S_i + a_ib + q_im) \text{ div } 2^k$;
```

It is seen that the residue range restrictions are identical for S_i and S_{i+1}. Hence, the intermediate operation can be recursively applied without additional processing. Algorithm 5.1–1 follows the style of the descriptions in the article in Appendix D, where the residue ranges are non-negative, and where the digit sets for the quotient and the multiplier are non-negative. Symmetric ranges could have been used as well. In Appendix D a series of algorithmic descriptions of the multiplication operation are given. It is shown, that if the number of recursion cycles, n, obeys $2^{kn} > 4m$, and if the input operands a and b is restricted to the residue range $[0; 2m[$, then the resulting product $(a \cdot b)r^{-n} \pmod{m}$ will belong to the same residue range. Hence, the multiplication operation also can be applied recursively without any additional processing.

5.2 Reducing the Recursion Cycle Time

The sub-operations of the intermediate operation given by Algorithm 5.1–1 are very similar to the sub-operations of the methods in Chapter 3. The observations regarding the representations of intermediate operands, the encoding of the digits, and the computations of multiples apply for Montgomery multiplication as well. Furthermore, the quotient determination operation is subject to considerations and tradeoffs similar to those presented in Section 3.10. The recursion cycle time for the intermediate operation given by Algorithm 5.1–1 is, however, expected to be slightly shorter than for the analogous traditional operation: The quotient digit set for the former operation is smaller, implying that the time for computing the multiples $q_i m$ is shorter. Furthermore, the length of the operands used in the quotient determination operation is smaller, giving a smaller computing time and, in case the quotient determination is based on a table look-up, a smaller number of table entries. These observations are also described by Kornerup in [Kor93b], where the traditional modular multiplication method is compared to Montgomery's multiplication method. The quotient determination is based on table look-up in both methods. Bosselaers, Govaerts and Vandewalle have made a similar comparison, based on experiments with software implementations, in [BGV93]. They conclude that Montgomery's multiplication method leads to a slightly faster evaluation of modular exponentials with large operands.

It is, however, possible to obtain a substantial speed improvement by combining the high-radix multiplication approach with a set of optimisation techniques. This is the main issue of the article in Appendix D, where it is shown that the *recursion cycle time can be made independent of the radix of the multiplication method.* Moreover, it is shown that *the time for a recursion cycle can be made as short as the time for a single redundant 4–2 addition.* These properties are superior to those of the traditional methods: As described in Section 3.11 an increased value of the radix implies an increased recursion cycle time for the traditional modular multiplication methods.

A detailed description of the optimisation techniques applied to Montgomery's multiplication method is included in Appendix D. In the following subsection, an overview of the relationship between these techniques and the optimisation techniques presented in Chapter 3 will be given.

5.2.1 Optimisation Techniques

The strategy of the optimisation techniques is identical to the strategy of the techniques described in Chapter 3: The complexity of the intermediate operation is reduced by means of *precomputations* that only have to be performed once for each change of the value of modulus. Since several consecutive modular multiplications are performed using the same value of modulus, the additional time for performing the precomputation becomes negligible. Furthermore, *parallel computations* are utilised. It turns out that the computation of consecutive intermediate operations can be overlapped and, therefore, that Montgomery's multiplication can be efficiently computed on a pipelined hardware architecture. Appendix D describes three optimisation techniques that, to some extent, can be applied independently of each other:

1. It is possible to *avoid the multiplication operation in the quotient determination* by adjusting the value of the modulus. A similar optimisation technique for the traditional modular multiplication operation was presented in Subsection 3.10.4, where the aim was to obtain a modulus value that is close to a power of two, i.e. the most significant digits have a fixed known value after the adjustment. In Montgomery's multiplication operation it is desirable to obtain a fixed known value of the *least significant digit* of modulus. Hereby, also the value of $m' \bmod 2^k$,

used in the quotient determination (5.3) becomes fixed. In Appendix D it is shown how to *adjust the value of the modulus such that* m' mod 2^k *becomes equal to* 1, i.e. the multiplication operation in the quotient determination is avoided. The new modulus value, say $\widetilde{m}$, is obtained through a simple scaling of m,

$$\widetilde{m} = c \cdot m, \text{ where } c = m' \bmod 2^k. \tag{5.4}$$

The penalty for replacing the value of modulus with $\widetilde{m}$ is a larger residue range $[0; 2\widetilde{m}[$ of the results of Montgomery's multiplication. However, in applications like modular exponentiation, where several intermediate multiplications are performed, the additional time for converting the final result into the residue range $[0; m[$ is vanishing. Since the scaling constant c belongs to $\{1, 3, \ldots, 2^{k-1}\}$ (note that m' is odd when m is odd) the residue range of the intermediate products will, at worst, be $[0; 2(2^k - 1)m[$. This implies that the required number of recursion cycles n in Montgomery's multiplication method may increase by one. As previously mentioned, the number of recursion cycles is constrained by $2^{kn} > 4\widetilde{m}$.

2. It is possible to *avoid the addition operation in the quotient determjnatjon* by scaling the value of the multiplicand. An analogous optimisation technique for the traditional modular multiplication operation was described in Section 3.5, where the aim was to achieve a higher degree of parallelism in the computation. The means was an implicit scaling of the *multiplier* a with the constant 2^k. Since a multiplication by 2^k is obtained through a left-shift, the scaling is implicit. In Montgomery's multiplication operation it is advantageous to implicitly scale the *multiplicand* b with the constant 2^k. Hereby, the value of $a_i b'$, where b' denotes the scaled multiplicand, in the quotient determination (5.3) becomes divisible by 2^k and, consequently, the addition operation in the quotient determination operation is avoided,

$$((S_i + a_i b') \cdot m') \bmod 2^k = (S_i \cdot m') \bmod 2^k, \text{ where } b' \bmod 2^k = 0.$$

In order to compensate for the multiplicand scaling, the number of recursion cycles in Montgomery's multiplication method is increased by one,

$$(a_i \cdot b)r^{-1} \equiv_m (a \cdot 2^k b)r^{-(n+1)}, \text{ where } r^{-1} \cdot 2^k \equiv_m 1.$$

This penalty is identical to the penalty of using the implicit scaling technique on the traditional modular multiplication operation.

When b is scaled, the updating statement in Algorithm 5.1–1 can be formulated as $S_{i+1} := (S_i + q_i m) \text{ div } 2^k + a_i b$. Hence, there is no explicit reference to the scaled multiplicand b'.

3. It is possible to *parallelise the computation of consecutive intermediate operations*. The idea of performing an overlapped computation of the intermediate operation is introduced by Shand and Vuillemin in [SV93], where the technique is denoted "quotient pipelining". The basic idea is to postpone the use of the quotient digit q_i, computed on basis of information available in the ith recursion cycle, by d recursion cycles. Hence, the term $q_i m$ does not appear in the updating statement until the $(i + d)$th recursion cycle. Hereby, it is possible to overlap the computation of d consecutive intermediate operations by means of a pipelined computation scheme.

   In Appendix D the quotient pipelining technique has been combined with the above described optimisation techniques. It is shown how to obtain a particular simple intermediate operation, where *the quotient determination is a simple inspection of the least significant digit of the intermediate operand*, $q_i := S_i \bmod 2^k$, *and the updating statement reduces to a shift-and-add operation*, $S_{i+1} := S_i \text{ div } 2^k + T_{i-d}$. The term T_{i-d} is the sum of the multiples $q_{i-d}\hat{m}$ and $a_i b$, where $\hat{m}$ is a precomputed operand. Both multiples are based on information available after the $(i - d)$th recursion cycle. Hence, the computation of T_{i-d} can be performed by an architecture that has been pipelined into d stages. The operand $\hat{m}$ is precomputed each time the value of modulus is changed. It is equal to $(\widetilde{m} + 1) \text{ div } 2^{k(d+1)}$, where $\widetilde{m}$ is the result of a scaling of m similar to (5.4),

$$\widetilde{m} = c \cdot m, \text{ where } c = m' \bmod 2^{k(d+1)}. \tag{5.5}$$

   Since the quotient determination reduces to a simple inspection of the least significant digit of S_i, *the quotient determination requires no circuitry and, hence, the area contribution from this operation is zero.*

Compared to the quotient determination method used in the exponentiation processor described in Chapter 4 this is a significant improvement. The processor's quotient determination unit had a significant contribution to the total area and, furthermore, it had a significant contribution to the recursion cycle time. In fact, the area of the processor's quotient determination unit increases by a rate of 2^k for increasing radix 2^k values.

In Appendix D an example hardware architecture for the computation of a radix 2^8 Montgomery multiplication is discussed. It is seen that the recursion cycle time can be reduced to the time for a single shift-and-add operation, which is efficiently performed by a redundant adder. Furthermore, it is seen that the recursion cycle time is independent of the chosen radix. This property makes Montgomery multiplication superior to the traditional modular multiplication methods. According to Section 3.11 the recursion cycle time for the traditional modular multiplication methods can be expected to increase by a rate of $\log_2 k$ for increasing radix 2^k values. Hence, *a significant step toward an efficient utilisation of high radices in modular multiplication* has been achieved.

There is, however, a penalty imposed by using the quotient pipelining technique:

- First, the number of recursion cycles increases with d since the use of quotient digit q_i is postponed for d recursion cycles.
- Second, because of the d pipeline buffers in the hardware architecture, the result of the multiplication process is further delayed by d recursion cycles. If the input operands for the next multiplication is known before the result of the present multiplication is completed, it is possible to start up the next multiplication as soon as the first stage in the pipeline has completed the present computation. Hence, it may be possible to overlap two consecutive modular multiplications and, hence, to avoid the penalty of d extra recursion cycles per multiplication. The required property is, however, dependent on the application. So, the actual scheduling of the multiplications may be improved after a further investigation of the application.

- Third, the implicit scaling of the multiplicand requires an additional recursion cycle.
- Finally, the residue range of the product increases to $[0; 2\widetilde{m}[$, which at worst is $[0; 2(2^{k(d+1)} - 1)m[$. So, due to the constraint $4\widetilde{m} < 2^{kn}$, the number n of basic recursion cycles increases by up to d cycles compared to the original radix 2^k version of Montgomery's method (5.2). Since $\widetilde{m}$ is odd, it cannot be a power of 2. Therefore, the constraint $4\widetilde{m} < 2^{kn}$ is obeyed by the following value of n:

$$\begin{aligned} n &= \left\lceil \frac{\log_2 4\widetilde{m}}{k} \right\rceil \\ &\leq \left\lceil \frac{\log_2 m + k(d+1) + 2}{k} \right\rceil \\ &= \left\lceil \frac{\log_2 m + 2}{k} \right\rceil + d + 1 \\ &\leq \left\lceil \frac{\log_2 m}{k} \right\rceil + d + 2, \text{ where } k \geq 2 \end{aligned}$$

In total, including all the additional penalty terms, the worst case number of recursion cycles for a radix 2^k version of the optimised Montgomery multiplication

$$n + d + d + 1 = \left\lceil \frac{\log_2 m}{k} \right\rceil + 3(d+1). \tag{5.6}$$

Therefore, the delay parameter d should be chosen as small as possible. In Appendix D it is indicated how the term T_{i-d} can be computed by means of a pipelined Wallace Tree, see Subsection 3.2.2, containing about $\log_2 k$ stages. The time for each stage is smaller than or equal to a redundant 4–2 addition, which corresponds to the delay of two full adders.

It should be mentioned, that the resulting modular product of the optimised Montgomery multiplication is redundantly represented. So, the time for a conversion into non-redundant representation should be added to the computing time indicated by (5.6). This issue is discussed in Appendix D as well. As for the traditional modular multiplication methods, it is possible to avoid the conversion by allowing the input operands a and b to be redundant represented. See the analogous discussion concerning the traditional modular multiplication methods in Subsection 3.7.2.

5.3 Additional Processing

In applications, like large operand modular exponentiation, where several intermediate modular multiplications are performed, the time for performing the additional processing required by Montgomery's method is usually negligible. The additional processing can be divided into two groups:

Pre-processing. A series of computations must be performed before the Montgomery multiplications are initiated. The computations may be seen as a part of the initialisation process of the application. For a modular exponentiation, the pre-processing consists of the following tasks:

- Compute the neutral, $r^n \bmod m$, for Montgomery's multiplication composition.
- Compute $r^{2n} \bmod m$, which is used in the transformation of operands into the M-residue domain.
- Compute $m' \bmod 2^{k(d+1)}$, the multiplicative inverse of $-m$ modulo $2^{k(d+1)}$.
- Compute $\hat{m}$ to be used in the updating statements. The expression for $\hat{m}$ is $(\widetilde{m} + 1) \text{ div } 2^{k(d+1)}$, where the scaled modulus $\widetilde{m}$ equals $(m' \bmod 2^{k(d+1)}) \cdot m$.

Some authors have a quit efficient solution to the pre-processing: In applications like the RSA crypto system the value of m is a part of the key. Therefore, it is known prior to the application and, consequently, it can be *assumed* that the "key" include the value of m plus the derivative values $r^n \bmod m$, $r^{2n} \bmod m$ and $\hat{m}$. Of course these additional values can be computed simultaneously with generating the RSA keys. However, it cannot be expected that all of the equipment for performing the encryption and decryption in a large (public) communication network are based on this particular implementation of the modular multiplications. There may very well be other implementations that uses a different set of derivatives. So, in a concrete application it cannot be assumed that these derivatives are part of the input.

Post-processing. After the final multiplication, and transformation from the M-residue range, a result in the residue range $[0; 2\widetilde{m}[$ has been obtained. Hence, a final conversion into the residue range $[0; m[$ is required. A similar conversion is required for the traditional modular multiplication methods.

Regarding the computing time, none of the above computations are particularly complex in comparison to a series of modular exponentiations. Of greater concern is the required circuitry of a hardware implementation that supports the additional processing. Hence, it is a challenging exercise to figure out how the additional processing may be performed by the existing hardware architecture without too many modifications and too much additional circuitry. This thesis does not include a solution to the exercise. However, some ideas and hints are provided:

- According to Kornerup [Kor94b], the modular reduction technique used by Montgomery is similar to a method, proposed by Hensel [Hen08] in 1908, for computation of the multiplicative inverse of an odd number modulo 2^n Indeed, Kornerup shows how a slightly modified version of Montgomery's multiplication method can be applied for computing the value of m' mod $2^{k(d+1)}$.

- When the scaling constant $c = m' \bmod 2^{k(d+1)}$ is computed, the scaled modulus $\widetilde{m} = c \cdot m$ may be computed by the existing hardware architecture. Finally, the operand $\hat{m} = (\widetilde{m} + 1) \operatorname{div} 2^{k(d+1)}$ is computed.

- It is not necessary to perform a complete reduction modulo m of r^n and r^{2n} since the multiplication method accepts residues in the range $[0; 2\widetilde{m}[$.

- Assume that r^n mod $\widetilde{m}$ has been computed. Then, it is possible to compute r^{2n} by an exponentiation process, where the exponent is set equal to kn, and the base is set equal to $2r^n \pmod{\widetilde{m}}$: Observe that $2r^n \pmod{\widetilde{m}}$ in the range $[0; 2\widetilde{m}[$ can be obtained by a left-shift of r^n mod $\widetilde{m}$. Furthermore, observe that $2r^n \pmod{\widetilde{m}}$ is the M-residue of 2. Therefore, the exponentiation in the domain of M-residues will result in the residue $2^{kn}r^n \pmod{\widetilde{m}}$ in the range $[0; 2\widetilde{m}[$. Hence, since r is equal to 2^k, the desired residue $r^{2n} \pmod{\widetilde{m}}$ has been computed.

- It is possible to convert the final result, say z, from the residue range $[0; 2\widetilde{m}[$ into the residue range $[0; 2m[$ by means of an *exact division* by the scaling constant c, where c is given by (5.5). Kornerup [Kor94b] has showed how to perform exact divisions by a slightly modified version of Montgomery's multiplication method.

  To be able to use the exact division technique, the operation used for transforming an operand from the M-residue domain, say y, is modified (See Figure 5.1, page 174): Instead of computing $z = y *_{\mathbb{Z}_m}^{M} 1$, the scaled result $z' = y *_{\mathbb{Z}_m}^{M} c$ is computed. Then from the congruence $z' \equiv c \cdot z \pmod{c \cdot m}$ it follows that $z' \text{ div } c \equiv z \pmod{m}$. Furthermore, since $z' \in [0; 2c \cdot m[$, it follows that $z \in [0; 2m[$. Hence, a conversion into the residue range $[0; 2m[$ has been obtained by an exact division that may be computed by a slightly modified version of the Montgomery multiplication method.

5.4 Summary and Discussion

In this chapter, another representation of residues, denoted M-residues after Montgomery, has been discussed. The advantage of this representation is a more efficient modular multiplication method, where the quotient determination operation is simpler than the corresponding operation for the traditional modular multiplication methods.

It has been discussed how a combination of two optimisation techniques, "adjustment of the modulus" and "shifting of the multiplicand", leads to a particular simple quotient determination method, *where a quotient digit is obtained through a simple inspection. Hence, there is no computation associated with the operation and, consequently, no circuitry is required for the operation.* This implies that the critical operation of the inter-mediate operation is the formation of the multiples $q_i m$ and $a_i b$. Moreover, it has been discussed how a further combination with the "quotient pipelining" technique, invented by Shand and Vuillemin [SV93], allows a pipelined computation of these multiples. The net effect of these optimisations is a modular multiplication method, where *the recursion cycle time is independent of the radix of the multiplication method.*

Indeed, the recursion cycle time can be made as short as the delay of a single redundant addition. This is a significant improvement of the recursion cycle time of the traditional modular multiplication methods (and the non-optimised Montgomery multiplication method, as well): According to Chapter 3 the recursion cycle time includes the time for determining a quotient digit q_i, the time for computing the multiple $q_i m$, and the time for subtracting $q_i m$ from the intermediate result. Compared to the new method for computing a Montgomery multiplication, this represents an overhead corresponding to the time for a quotient determination plus the time for computing a multiple. In fact, this overhead contributes significantly to the total recursion cycle time: Even though the time for the quotient determination may be reduced by adjusting the range of the modulus, see Subsection 3.10.4 the time for computing $q_i m$ will be longer than or equal to the delay of $\log_2 k$ redundant additions, where the radix is expressed as 2^k. Indeed, since the quotient digit range for the traditional methods are wider than the range for the optimised Montgomery method, the overhead will be longer than the delay of $\log_2 k$ redundant additions. Hence, *the recursion cycle time has been reduced by more than a factor* $\log_2 k$.

The aim of using high radices in the modular multiplication methods is to reduce the number of recursion cycles. In an *ideal* radix 2^k modular multiplication method the number of recursion cycles decreases by a factor of k, and the recursion cycle time is independent on the chosen radix. So, the computing time for an ideal high radix method decreases by a factor of k. This chapter has described a method with properties that approaches the ideal properties: The recursion cycle time is independent of the radix. However, the optimisation techniques add a penalty term, $3(\log_2 k + 1)$, to the ideal number of recursion cycles. It might be possible, through a more careful scheduling of the computation on the pipelined architecture, to reduce this penalty term. Compared to the computing time for an ideal high radix method, the computing time for the non-optimised Montgomery method, and for the traditional methods in Chapter 3, has a penalty *factor* proportional to $\log_2 k$. Hence, the optimisation techniques presented in this chapter can be viewed as a way of changing a multiplicative $\log_2 k$ time-penalty into an additive $\log_2 k$ time-penalty. It should be noted that the relative penalty for a given k, i.e. fraction of penalty-cycles in the total number of recursion cycles, is decreasing for increasing operands. This means that the total computing time is relatively closer to the ideal computing time for e.g. 512 bit modular multiplications than for e.g. 256 bit modular multiplications. Hence, the efficiency of the optimisation techniques is highest for computations with large operands.

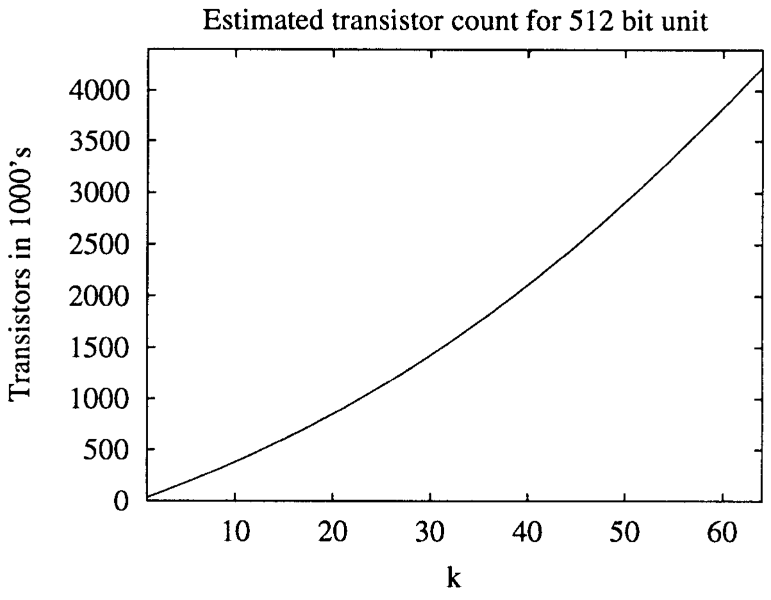


Figure 5.2: Transistor count of modular multiplication unit as function of the radix.

Orton, Peppard and Tavares [OPT93] have optimised a traditional high radix modular multiplication method using techniques very similar to the techniques described in this chapter. They utilise an adjustment of the modulus and a pipelined hardware architecture to obtain a recursion cycle time that is identical to the time for the optimised Montgomery method. The penalties are, however, larger than those for the Montgomery method: Since the range of the quotient digit values is wider, the number of pipeline stages is larger. The scaling constant, by which the modulus is scaled, is greater as well. Consequently, compared to the ideal high radix method, the number of additional recursion cycles is approximately $0.75k + 7.5$. This penalty is increasing linearly with k while the corresponding penalty for the Montgomery method is increasing with the logarithm of k. Moreover, due to the larger scaling constant, the intermediate operands are greater. This means that the required width of the hardware architecture is greater as well.

A pipelined hardware architecture for a radix 2^8 optimised Montgomery method has been presented in Appendix D. The required circuitry for this architecture is estimated to be about 300,000 transistors. This is about the same amount of circuitry that was required for the processor described in Chapter 4. This processor is implementing a traditional radix 2^5 modular multiplication method. Hence, the fact that no quotient determination unit is needed in the optimised Montgomery method implies that a higher radix can be utilised without increasing the amount of circuitry. Figure 5.2 shows an *estimate* of the required circuitry for a modular multiplication unit based on the optimised radix 2^k Montgomery method. The figure shows the transistor count as function of k for a unit that is used for modular exponentiation of 512 bit operands. Using a modern CMOS process technology it seems feasible to implement a single chip modular exponentiation processor, that is based on a high radix multiplication method with radices as high as 2^{30}: According to the estimate, such an implementation will require less than 1.5 million transistors.

The estimate in Figure 5.2 is based on the following observations: For a radix 2^k implementation the number of latched redundant adders is $\frac{k}{2}$ and the number of latched multiplexers is k. The width of the adders and multiplexers corresponds to the operand length, i.e. about $\log_2 \widetilde{m}$ bits, which is about $k(\log_2 k+1)+\log_2 m$ bits. Hence, the transistor count approximately increases by a rate of $\frac{3}{2}k \cdot (k(\log_2 k + 1) + \log_2 m)$.

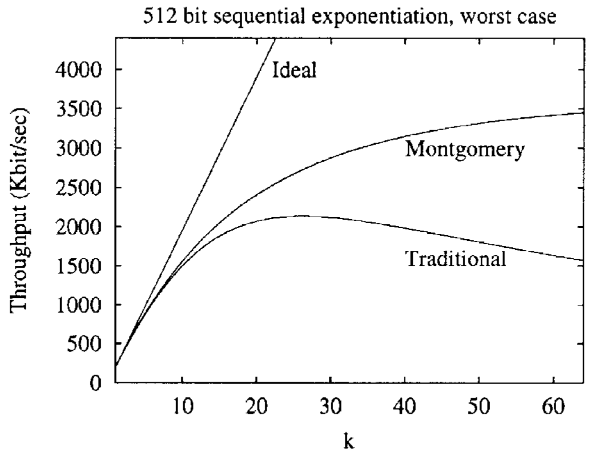


Figure 5.3: Throughput of 512 bit modular exponentiation as function of the radix.

To illustrate the speed potential of the optimised radix 2^k modular multiplication method, the estimated throughput of a modular exponentiation of 512 bit operands is shown as function of k in Figure 5.3. It is assumed that the recursion cycle time is 5 ns. Furthermore, it is assumed that a sequential binary exponentiation method is used. The throughput shown in the figure is for the worst case exponent value, i.e. two modular multiplications are performed for each exponent bit. (See Section 2.1). The figure shows three plots illustrating the obtainable throughput when one of the following modular multiplication methods is used: An ideal high radix method without any penalty terms, the optimised Montgomery method presented in this chapter, and the optimised traditional modular multiplication method developed by Orton, Peppard and Tavares. For radix values, where k is less than 10, the optimised methods are quit close to the ideal method. For higher radix values the penalty terms becomes more dominating. It is seen that the throughput for the traditional method is maximal for radices between 2^{20} and 2^{30}. Due to the smaller penalty term the Montgomery method is capable of utilising even higher radices. Even with a modest radix value, it is possible to obtain a throughput of more than 1 Mbit/set. *Hence, compared to the processor in Chapter 4, the throughput has increased by an order of magnitude.* This is in spite of the relatively primitive exponentiation method used in the computation of the throughput in Figure 5.3: The throughput may be increased by almost another order of magnitude by utilising more efficient exponentiation methods (see Chapter 2): The parallel exponentiation method doubles the worst case throughput. The Chinese Remainder Theorem may be used to further improve the throughput by a factor close to four.

The literature on modular multiplication methods includes several variations of Montgomery's method [SVB91, DK90, Eve90, Eld91, IMI92b, Sau92, Wal93, EW93] that have not been treated in this chapter. However, most of the articles describes methods—or applications using methods—based on radix 2 or radix 4. It should be mentioned that Arazi [Ara94] has observed that the quotient determination becomes simple in Montgomery's method for certain restricted values of modulus. Consequently, Arazi proposes to restrict the modulus values used in crypto systems to these favourable modulus values. It turns out that the adjustment of the modulus values, presented in this chapter, is equivalent to a transformation into the domain of such favourable moduli. Hence, it is not necessary to impose the restriction suggested by Arazi to the crypto systems.

Chapter 6

Conclusion

One of the aims of the work presented in this thesis was to implement a single-chip processor in the form of a VLSI circuit. The processor should be able to perform modular exponentiation at a rate of at least 64 Kbit/sec using 561 bit operands. As described in Chapter 4 the processor was successfully implemented. The performance measurements on a sample of chips show that they are able to compute the exponentials at a rate of more than 100 Kbit/sec. This is the fastest known implementation for computation of modular exponentials when no assumptions about the operand values are made.

Apart from a presentation of the results of implementing the processor, the thesis contains a throughout discussion of techniques for achieving efficient methods for computation of exponentials and modular products. Hereby, an insight into the problems associated to these computations have been provided. Furthermore, through a discussion of the related literature, the implications of the results presented in this thesis have been illuminated.

For a conclusion on each of the subtopics covered by this thesis, the reader is referred to the summarising sections included in each chapter. It is, however, appropriate to mention the main strategies and techniques used in the research for efficient computation methods:

Identify and utilise the properties of the problem under consideration. Both the properties of the *application* of an arithmetic operation and the properties of the *operation* itself can be utilised to improve the performance of a computation. Consider the application of RSA cryptography. If the prime factorisation of the modulus is known this can be utilised to improve the speed of modular exponentiation by a factor of four. Then consider the modular exponentiation operation. This operation is defined in terms of modular multiplication which is an associative algebraic composition. This algebraic property can be utilised to obtain a dramatic reduction in the required number of modular multiplications and, hence, the computing time. Finally, consider modular exponentiation as an application that uses the modular multiplication operation. The fact that the modulus value is fixed over several modular multiplications can be utilised to improve the speed of modular multiplication.

Choose a proper representation of the intermediate operands. Usually the input and output to an arithmetic operations is represented as binary numbers. However, the freedom of choosing the best suited representation of the intermediate operands can be utilised to obtain faster computations. Indeed, the issue of representation is one of the most important in the research on efficient methods for computation of arithmetic operations. Especially the concept of redundancy is useful: A *redundant digit* set can be utilised to obtain an addition operation where the computing time is independent of the length of the operands, and it can be used to reduce the required number of additions in multiplication operations. Furthermore, a *redundant residue range* can be used to improve the time for performing division and hence, modular reduction. The results presented in Chapter 5 show that it useful to represent residues by means of *M-residues* (Montgomery-residues). Finally, the effect of applying the *Chinese Remainder Theorem* for improving the speed of modular exponentiation can be viewed as the result of using a particular representation. It should be emphasised that none of these representations does exclude the use of one of the other representations. Indeed, it turns out that the fastest performing methods for modular exponentiation are based on a utilisation of *all* these representations.

Parallel computation techniques are very attractive when special-purpose hardware architectures can be applied. The possibility of using this technique is one of the main reasons that special-purpose hardware implementations have a higher performance than the standard micro-processors. Parallel computation techniques are utilised at all levels of computation: At the lowest level it is used in the architecture of redundant addition units, and at the highest level it used to speed up a modular exponentiation by means of the parallel binary method and/or by means of the Chinese Remainder Theorem. Furthermore, by *pipelining* the architecture, the parallel computations can be implemented with a minimum of additional circuitry.

Precomputation techniques are utilised in various forms. First, the technique of precomputing a table of often used values reduces the total computing time at the cost of additional circuitry for the table. Second, the precomputation technique can be used for transforming the operands of a computation. As discussed in the thesis it is advantageous to transform the value of modulus prior to a modular multiplication operation.

As discussed in Chapter 2 it seems difficult to achieve further progress in the methods for computation of exponentials. However, for applications that differ from RSA cryptography, it may be possible to find properties that can be utilised. The prospect of achieving improvements of the modular multiplication methods is more promising. In this thesis the possibilities of using *high-radix* modular multiplication has been investigated. Judging from the presented results, it is likely that computation rates of several Mbit/set soon will be achievable for modular exponentiation using 512 bit operands.

Bibliography

[ABMV93] G. B. Agnew, T. Beth, R. C. Mullin, and S. A. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6:3–13, 1993.

[AGLL94] Derek Atkins, Michael Graff, Arjen K. Lenstra, and Paul C. Leyland. The magic words are squeamish ossifrage. In Joseph Pieprzyk and Reihanah Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT '94. Proceedings*, volume 917 of *Lecture Notes in Computer Science*, pages 265–277, Wollongong, Australia, November 28–December 1 1994. Springer-Verlag, Berlin, 1995.

[AMOV91] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, 3:63–79, 1991.

[AMV88] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. Fast exponentiation in GF(2^n). In Christoph G. Günther, editor, *Advances in Cryptology – EURO-CRYPT '88. Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 251–255, Davos, Switzerland, May 1988. Springer-Verlag, Berlin, 1988. The contents of this article is included in [ABMV93].

[Ana92] Analogy, Inc., Aldwych House, Aldwych, London WC2B 4JP. *Saber, User's Guide*, 1992.

[Ara94] Benjamin Arazi. On primality testing using purely divisionless operations. *The Computer Journal*, 37(3):219–222, 1994.

[Atk68] Daniel E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968. This article is reprinted in [Swa90a], pages 173–182.

[Atk70] Daniel E. Atkins. Design of the arithmetic units of ILLIAC III: Use of redundancy and higher radix methods. *IEEE Transactions on Computers*, C-19(8):720–733, August 1970.

[Avi61] Algirdas Avižienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, EC-10(3):389–400, September 1961. This article is reprinted in [Swa90b], pages 54–65.

[Bak87] P. W. Baker. Fast computation of $A * B$ modulo N. *Electronics Letter*, 23(15):794–795, July 1987.

[Bak90] H. B. Bakoglu. *Circuits, interconnects, and Packaging for VLSI*. The VLSI Systems Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.

[Bar86] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86. Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323, Santa Bar-bara, California, August 11–15 1986. Springer-Verlag, Berlin, 1987.

[BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89. Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 400–407, Santa Barbara, California, August 20–24 1989. Springer-Verlag, Berlin, 1990.

[BFS91] Th. Beth, M. Frisch, and G. J. Simmons, editors. *Public-Key Cryptography: State of the Art and Future Directions*, volume 578 of *Lecture notes in Computer Science*, E.I.S.S. Workshop, Oberwolfach, Germany, July 3–6 1991. Springer-Verlag, Berlin, 1992.

[BG89] Thomas Beth and Dieter Gollman. Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communication.* 7(4):458–466, May 1989.

[BGMW92] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. Fast exponentiation with precomputation (extended abstract). In Rainer A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92. Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207, Balatonfüred, Hungary, May 24–28 1992. Springer-Verlag, Berlin, 1993.

[BGV93] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Comparison of three modular reduction functions. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPT0 '93. Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 175–186, Santa Barbara, California, August 22–26 1993. Springer-Verlag, Berlin, 1994.

[Bla83] G. R. Blakley. A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, C-32(5):497–500, May 1983.

[BO92] E. F. Brickell and A. M. Odlyzko. Cryptanalysis. A survey of recent results. In Simmons [Sim92b], chapter 10, pages 501–540.

[Boo51] Andrew D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(pt. 2):236–240, June 1951. This article is reprinted in [Swa90a], pages 100–104.

[Bra39] Alfred Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45:736–739, October 1939.

[Bra88] Gilles Brassard. *Modern Cryptology: A Tutorial*, volume 325 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

[Bri82] Ernest F. Brickell. A fast modular multiplication algorithm with application to two key cryptography. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology – Proceedings of Crypto 82*, pages 51–60, Santa Barbara, California, August 23–25 1982. Plenum Press, New York, 1983.

[Bri89] Ernest F. Brickell. A survey of hardware implementations of RSA (abstract). In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO '89. Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 368–370, Santa Barbara, California, August 20–24 1989. Springer-Verlag, Berlin, 1990.

[BRV89] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. Research report 3, Digital Equipment Corporation, Paris Research Laboratory, 85, Avenue Victor-Hugo, 92563 Rueil-Malmaison Cedex, France, June 1989.

[BRV93] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Programmable active memories: a performance assessment. Research report 24, Digital Equipment Corporation, Paris Research Laboratory, 85, Avenue Victor-Hugo, 92563 Rueil-Malmaison Cedex, France, March 1993.

[Cas91a] Cascade Design Automation Corporation, 3650 131st Avenue SE, Bellevue, WA 98006. *Cascade Design Automation Databook*, 1991.

[Cas91b] Cascade Design Automation Corporation, 3650 131st Avenue SE, Bellevue, WA 98006. *ChipCrafter, Designer's Handbook*, 1991.

[Cas91c] Cascade Design Automation Corporation, 3650 131st Avenue SE, Bellevue, WA 98006. *FINESSE Reference Manual*, 1991.

[CBK91] Andreas V. Curiger, Heinz Bonnenberg, and Hubert Kaeslin. Regular VLSI architectures for multiplication modulo $(2^n + 1)$. *IEEE Journal of Solid-State Circuits*, SC-26(7):990–994, July 1991.

[Che71] Tien Chi Chen. A binary multiplication scheme based on squaring. *IEEE Transactions on Computers*, C-20(6):678–680, June 1971. This article is reprinted in [Swa90a], pages 111–113.

[DDLM93] T. Denny, B. Dodson, A. K. Lenstra, and M. S. Manasse. On the factorization of RSA-120. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO '93. Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 166–174, Santa Barbara, California, August 22–26 1993. Springer-Verlag, Berlin, 1994.

[Den82] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1982.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[Dif88] Whitfield Diffie. The first ten years of public key cryptology. *Proceedings of the IEEE*, 76(5):560–577, May 1988. This article is also available as [Dif92].

[Dif92] Whitfield Diffie. The first ten years of public key cryptology. In Simmons [Sim92b], chapter 3, pages 135–175. This article is also available as [Dif88].

[DK90] Stephen R. Dussé and Burton S. Kaliski Jr. A cryptograhpic library for the Motorola DSP56000. In Ivan B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 230–244, Aarhus, Denmark, May 21–24 1990. Springer-Verlag, Berlin, 1991.

[DLS81] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, August 1981.

[DMW94] Diana M. D'Angelo, Bruce McNair, and Joseph E. Wilkes. Security in electronic messaging systems. *AT&T Technical Journal*, pages 7–13, May/June 1994.

[dR94] Peter de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Pre-proceedings of EUROCRYPT '94*, pages 403–415, 1994.

[dWQ90] Dominique de Waleffe and Jean-Jacques Quisquater. CORSAIR: A smart card for public key cryptosystems. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology – CRYPT0 '90. Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 502–513, Santa Barbara, California, August 11–15 1990. Springer-Verlag, Berlin, 1991.

[EL85] Miloš D. Ercegovac and Tomás Lang. A division algorithm with prediction of quotient digits. In Kai Hwang, editor, *Proceedings. 7th IEEE Symposium on Computer Arithmetic*, pages 51–56, Urbana, Illinois, June 4–6 1985. IEEE Computer Society Press, Los Alamitos, California, 1985.

[EL90] Miloš D. Ercegovac and Tomás Lang. Simple radix-4 division with operand scaling. *IEEE Transactions on Computers*, C-39(9):1204–1208, September 1990.

[Eld91] Stephen E. Eldridge. A faster modular multiplication algorithm. *International Journal of Computer Mathematics*, 40:63–68, 1991.

[Erc83] Miloš D. Ercegovac. A higher-radix division with simple selection of quotient digits. In *Proceedings. 6th IEEE Symposium on Computer Arithmetic*, pages 94–98, Aarhus, Denmark, June 20–22 1983. IEEE Computer Society Press, Los Alamitos, California, 1983.

[Erd60] P. Erdős. Remarks on number theory III. On addition chains. *Acta Informatica*, 6:77–81, 1960.

[ES293] ES2, European Silicon Structures, Mount Lane, Bracknell, Bergshire RG12 3DY, United Kingdom. ES2, *Techno1ogy & Services*, 1993.

[Eve90] Shimon Even. Systolic modular multiplication. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology – CRYPT0 '90. Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 619–624, Santa Barbara, California, August 11–15 1990. Springer-Verlag, Berlin, 1991.

[EW93] Stephen E. Eldridge and Colin D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, C-42(6):693–699, June 1993.

[FDG90] C. H. N. Forster, S. S. Dlay, and R. N. Gorgui-Naguib. Carry delayed save adders for computing the product $A \cdot B$ modulo N in $\log_2 N$ steps. *Electronics Letter*, 26(18):1544–1545, August 1990.

[Fum91] Walter Fumy. (Local area) network security. In Preneel et al. [PGV91], pages 211–226.

[Gar59] Harvey L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, June 1959.

[Gar77] Martin Gardner. Mathematical games. A new kind of cipher that would take millions of years to break. *Scientific American*, 237(2):120–124, August 1977.

[GD85] Lance A. Glasser and Daniel W. Dopperpuhl. *The Design and Analysis of VLSI Circuits*. The VLSI Systems Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

[GD88] Phillip Gallay and Eric Depret. A cryptography processor. In *35th IEEE International Solid-State Circuits Conference, ISSCC '88. Digest of Technical Papers*, pages 148–149, San Francisco, California, 1988. IEEE Press, New York, N. Y., 1988.

[GG90] W. Geiselmann and D. Gollmann. VLSI design for exponentiation in $GF(2^n)$. In Jennifer Seberry and Josef Pieprzyk, editors, *Advances in Cryptology – AUSCRYPT '90. Proceedings*, volume 453 of *Lecture Notes in Computer Science*, pages 398–405, Sydney, Australia, January 8–11 1990. Springer-Verlag, Berlin, 1990.

[Gib88] J. K. Gibson. A generalization of Brickell's algorithm for fast modular multiplication. *BIT*, 28:755–763, 1988.

[Has85] Johan Hastad. On using RSA with low exponent in a public key network. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPTO '85. Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 403–408, Santa Barbara, California, August 18–22 1985. Springer-Verlag, Berlin, 1986.

[Has88] Johan Hastad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, April 1988. An early version of this article is [Has85].

[HDVG88] Frank Hoornaert, Marc Decroos, Joos Vandevalle, and René Govaerts. Fast RSA-hardware: Dream or reality? In Christoph G. Gunther, editor, *Advances in Cryptology – EUROCRYPT '88. Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 257–264, Davos, Switzerland, May 1988. Springer-Verlag, Berlin, 1988.

[Hen08] K. Hensel. *Theorie der Algebraischen Zablen*. B. G. Teubner, Leipzig, 1908.

[HNN$^+$87] Yoshihisa Harata, Yoshio Nakamura, Hiroshi Nagase, Mitsuharu Takagawa, and Naofumi Takagi. A high-speed multiplier using a redundant binary adder tree. *IEEE Journal of Solid-State Circuits*, SC-22(1):28–34, February 1987. This article is reprinted in [Swa90b], pages 237–243.

[HPS71] Paul G. Hoel, Sidney C. Port, and Charles J. Stone. *Introduction to Probability Theory*. The Houghton Mifflin Series in Statistics. Houghton Mifflin Company, Boston, 1971.

[Hwa79] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, Inc., New York, 1979.

[ICHO89] Peter A. Ivey, Alan L. Cox, John R. Harbridge, and John K. Oldfield. A single-chip public key encryption subsystem. *IEEE Journal of Solid-State Circuits*, SC-24(4):1071–1075, August 1989.

[IMI92a] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. High-speed implementation methods for RSA scheme. In Rainer A. Rueppel, editor, *Advances in Cryptology – EUROCRYPL '92. Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 221–237, Balatonfüred, Hungary, May 24–28 1992. Springer-Verlag, Berlin, 1993.

[IMI92b] Keiichi Iwamura, Tsutomu Matsumoto, and Hideki Imai. Systolic-arrays for modular exponentiation using Montgomery method (extended abstract). In Rainer A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92. Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 477–481, Balatonfüred, Hungary, May 24–28 1992. Springer-Verlag, Berlin, 1993.

[IWSD92] Peter A. Ivey, Simon N. Walker, Jon M. Stern, and Simon Davidson. A high performance RSA encryption processor in SOI and bulk CMOS technologies. In *Proceedings. 18th European Solid-State Circuits Conference, ESSCLRC '92*, pages 238-241, Copenhagen, Denmark, September 21–23 1992. Technical University of Denmark, 1992.

[JM89] J. Jedwab and C. J. Mitchell. Minimum weight modified signed-digit representation and fast exponentiation. *Electronics Letter*, 25(17):1171–1172, August 1989.

[Kat94] Rajendra Katti. A modified Booth algorithm for high radix fixed-point multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):522–524, December 1994.

[KH88] Scin-ichi Kawamura and Kyoko Hirano. A fast modular arithmetic algorithm using a residue table (extended abstract). In Christoph G. Günther, editor, *Advances in Cryptology – EUROCRYPT '88. Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 245–250, Davos, Switzerland, May 1988. Springer-Verlag, Berlin, 1988.

[KH90a] Ç. K. Koç and C. Y. Hung. Carry-save adders for computing the product AB modulo N. *Electronics Letter*, 26(13):899–900, June 1990.

[KH90b] Ç. K. Koç and C. Y. Hung. Multi-operand modulo addition using carry save adders. *Electronics Letter*, 26(6):361–363, March 1990.

[KH91] Ç. K. Koç and C. Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Processing*, 3:215–223, 1991.

[Kno88] Hans-Joachim Knobloch. A smart card implementation of the Fiat-Shamir identification scheme. In Christoph G. Günther, editor, *Advances in Cryptology – EUROCRYPT '88. Proceedings*, volume 330 of *Lecture Notes in Computer Science*, pages 87–95, Davos, Switzerland, May 1988. Springer-Verlag, Berlin, 1988.

[Knu68] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1968.

[Knu81] Donald E. Knuth. *Seminumerica1 Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1981.

[Koc85] Martin Kochanski. Developing an RSA chip. In Hugh C. Williams, editor, *Advances in Cryptology – CRYPT0 '85. Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 350–357, Santa Barbara, California, August 18–22 1985. Springer-Verlag, Berlin, 1986.

[Kor93a] Israel Koren. *Computer Arithmetic Algorithms*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1993.

[Kor93b] Peter Kornerup. High-radix modular multiplication for cryptosystems. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic*, pages 277–283, Windsor, Ontario, June 29–July 2 1993. IEEE Computer Society Press, Los Alamitos, California, 1993.

[Kor94a] Peter Kornerup. Digit-set conversions: Generalizations and applications. *IEEE Transactions on Computers*, C-43(5):622–629, May 1994.

[Kor94b] Peter Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, C-43(8):892–898, August 1994.

[Kri70] E. V. Krishnamurthy. On optimal iterative schemes for high-speed division. *IEEE Transactions on Computers*, C-19(3):227–231, March 1970.

[Kun88] S. Y. Kung. *VLSI Array Processors*. Prentice Hall Information and System Sciences Series. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1988.

[LHLH88] Erl-Huei Lu, Lein Harn, Jau-Yien Lee, and Wen-Yih Hwang. A programmable VLSI architecture for computing multiplication and polynomial evaluation modulo a positive integer. *IEEE Journal of Solid-State Circuits*, SC-23(1):204–207, February 1988.

[Lin] Lintel. Cryptech PQR 512 RSA chip. Data Sheet Rev. 1.0 DOC 750512, Lintel nv/sa, Av. de Jettelaan 32, 1080 Brussels, Belgium.

[LKB$^+$94] Susan Landau, Stephen Kent, Clint Brooks, Scott Charney, Dorothy Denning, Whitfield Diffie, Anthony Lauck, Douglas Miller, Peter Neumann, and David Sobel. Crypto policy perspectives. *Communications of the ACM*, 37(8):115–121, 1994.

[LL94] Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputing. In Yvo G. Desmedt, editor, *Advances in Cryptology – CRYPTO '94. Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107, Santa Barbara, California, August 21–25 1994. Springer-Verlag, Berlin, 1994.

[LM89] Arjen K. Lenstra and Mark S. Manasse. Factoring by electronic mail. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT '89. Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 355–371, Houthalen, Belgium, April 10–13 1989. Springer-Verlag, Berlin, 1990.

[MA85] S. B. Mohan and B. S. Adiga. Fast algorithms for implementing RSA public key cryptosystem. *EIectronics Letters*, 21(17):761, August 1985.

[Mac61] O. L. MacSorley. High-speed arithmetic in binary computers. *IRE Proceedings*, 49(1):67–91, January 1961. This article is reprinted in [Swa90a], pages 14–38.

[McC86] D. P. McCarthy. Effect of improved multiplication efficiency on exponentiation algorithms derived from addition chains. *Mathematics of Computation*, 46(174):603–608, April 1986.

[Miy82] Shoji Miyaguchi. Fast encryption algorithm for the RSA cryptographic system. In *Proceedings. 24th IEEE Computer Society International Conference, COMPCON fall '82*, pages 672–678, Washington, D.C., 1982. IEEE Press, New York, N.Y., 1982.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Moo88] J. H. Moore. Protocol failures in cryptosystems. *Proceedings of the IEEE*, 76(5), May 1988. This article is also available as [Moo92].

[Moo92] J. H. Moore. Protocol failures in cryptosystems. In Simmons [Sim92b], chapter 11, pages 541–558. This article is also available as [Moo88].

[Mor89] Hikaru Morita. A fast modular-multiplication algorithm based on higher radix. In Gilles Brassard, editor, *Advances in Cryptology – CRYPT0 '89. Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 387–399, Santa Barbara, California, August 20–24 1989. Springer-Verlag, Berlin, 1990.

[Mor90] Hikaru Morita. A fast modular-multiplication module for smart cards. In Jennifer Seberry and Josef Pieprzyk, editors, *Advances in Cryptology – AUSCRYPT '90. Proceedings*, volume 453 of *Lecture Notes in Computes Science*, pages 406–409, Sydney, Australia, January 8–11 1990. Springer-Verlag, Berlin, 1990.

[MP89] Rajeev Madhaven and Lloyd E. Peppard. A multiprocessor GaAs RSA cryptosystem. In *Proceedings. Canadian Conference on VLSI, CCVLSI'89*, pages 115–122, Vancouver, Canada, 1989.

[Nec92] James Nechvatal. Public key cryptography. In Simmons [Sim92b], chapter 4, pages 177–288.

[NS81] Michael J. Norris and Gustavus J. Simmons. Algorithms for high-speed modular arithmetic. *Congressus Numerantium*, 31:153–163, April 1981.

[Obe79] R. M. M. Oberman. *Digital Circzuits for Binary Arithmetic*. The MacMillan Press Ltd., London, 1979.

[OK91] Holger Orup and Peter Kornerup. A high-radix hardware algorithm for calculating the exponential M^E module N. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 51–56, Grenoble, France, June 26–28 1991. IEEE Computer Society Press, Los Alamitos, California, 1991.

[OPT93] Glenn Orton, Lloyd Peppard, and Stafford Tavares. A design of a fast pipelined modular multiplier based on a diminished-radix algorithm. *Journal of Cryptology*, 6:183–208, 1993.

[ORS$^+$86] G. A. Orton, M. P. Roy, P. A. Scott, L. E. Peppard, and S. E. Tavares. VLSI implementation of public-key encryption algorithms. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86. Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 277–301, Santa Barbara, California, August 11–15 1986. Springer-Verlag, Berlin, 1987.

[Oru94] Holger Orup. A 100Kbit/s single chip modular exponentiation processor. In *HOT Chips VI, Symposium Record*, pages 53–59, Stanford University, Stanford, California, August 14–16 1994. Only the slides from the presentation at HOT Chips VI are printed in the Symposium Record. An abstract is available from the author.

[Oru95] Holger Orup. Simplifying quotient determination in high-radix modular multiplication. In Simon Knowles and William H. McAllister, editors, *Proceedings. 12th 1EEE Symposium on Computer Arithmetic*, pages 193–199, Bath, England, July 19–21 1995. IEEE Computer Society Press, Los Alamitos, California, 1995.

[OS90] Holger Orup and Erik Svendsen. VICTOR. Forbedringer og videreudviklinger af VICTOR—en integreret kreds til understøttelse af RSA-kryptosystemer. Internal report, Department of Computer Science, University of Aarhus, Denmark, June 1990. In Danish.

[OSA90a] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an efficient RSA hardware implementation. In Ivan B. Damgård, editor, *Advances in crypto1ogy – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture notes in Computer Science*, pages 245–252, Aarhus, Denmark, May 21–24 1990. Springer-Verlag, Berlin, 1991.

[OSA90b] Holger Orup, Erik S vendsen, and Erik Andreasen. VICTOR. Teoretiske og eksperimentelle undersøgelser af algoritmer til understøttelse af RSA-krypto-systemer med henblik på VLSI design. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, January 1990. In Danish.

[Par93] Behrooz Parhami. On the implementation of arithmetic support functions for generalized signed-digit number systems. *IEEE Transactions on Computers*, C-42(3):379–384, March 1993.

[PGV91] Bart Preneel, René Govaerts, and Joos Vandevalle, editors. *Computer Security and Industrial Cryptography: State of the Art and Evolution*, volume 741 of *Lecture Notes in Computer Science*, ESAT Course, Leuven, Belgium, May 21–23 1991. Springer-Verlag, Berlin, 1993.

[PH94] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[Pij91] Pijnenburg. PCC100 DES encryption device. Product data sheet version 1.2, Pijnenburg micro-electronics & software b.v., Boxtelseweg 26, 5261 NE Vught. PO Box 330, 5260 AH Vught, The Netherlands, 1991.

[Pij92] Pijnenburg. PCC200 RSA encryption device. Product data sheet version 1.7, Pijnenburg micro-electronics & software b.v., Boxtelseweg 26, 5261 NE Vught. PO Box 330, 5260 AH Vught, The Netherlands, 1992.

[QC82] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letter*, 18(21):905–907, October 1982.

[Rei60] George W. Reitwiesner. Binary arithmetic. In Franz L. Alt, editor, *Advances in Computers*, volume 1, pages 231–308. Academic Press, Inc., New York, 1960.

[Riv80] Ronald L. Rivest. A description of a single-chip implementation of the RSA cipher. *Lambda Magazine*, 1(3):14–18, Fourth Quarter 1980.

[Riv82] Ronald L. Rivest. A short report on the RSA chip. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology – Proceedings of Crypto 82*, page 327, Santa Barbara, California, August 23–25 1982. Plenum Press, New York, 1983.

[Riv84] Ronald L. Rivest. RSA chips (past/present/future) – (extended abstract). In Thomas Beth, Norbert Cot, and Ingemar Ingemarsson, editors, *Advances in Cryptology. Proceedings of EUROCRYPT 84*, volume 209 of *Lecture notes in Computer Science*, pages 160–165, Paris, France, April 9–11 1984. Springer-Verlag, Berlin, 1985.

[Rob85] James E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7(3):218–222, September 1958. This article is reprinted in [Swa90a], pages 159–163.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digi-tal signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, feb 1978.

[Rub75] Louis P. Rubinfield. A proof of the modified Booth's algorithm for multiplication. *IEEE Transactions on Computers*, C-24(10):1014–1015, October 1975.

[Sau92] Jörg Sauerbrey. A modular exponentiation unit based on systolic arrays. In Jennifer Seberry and Yuliang Zheng, editors, *Advances in Cryptology – AUSCRYPT '92. Proceedings*, volume 718 of *Lecture Notes in Computer Science*, pages 505–516, Gold Coast, Queensland, Australia, December 13–16 1992. Springer-Verlag, Berlin, 1993.

[SB88] Miles E. Smid and Dennis K. Branstad. The Data Encryption Standard. Past and future. *Proceedings of the IEEE*, 76(5):550–559, May 1988. An updated version of this article is [SB92].

[SB92] Miles E. Smid and Dennis K. Branstad. The Data Encryption Standard. Past and future. In Simmons [Sim92b], chapter 1, pages 43–64. This article is an updated version of [SB88].

[SBV90] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. In *Proceedings. 2nd Annual ACM Symposium on Parallel Algoritms and Architectures. SPAA '90*, pages 138–145, Island of Crete, Greece, July 2–6 1990. Association for Computing Machinery, New York, 1990. This article also appears as [SVB91].

[SVB91] M. Shand, P. Bertin, and J. Vuillemin. Hardware speedups in long integer multiplication. *Computer Architecture News, SIGARCH – ACM*, 19(1):106–113, March 1991. This article also appears as [SBV90].

[Sch75] Arnold Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1(1):1–12, June 1975.

[Sch93] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code* in C. John Wiley & Sons, New York, 1993.

[Sco85] Norman R. Scott. *Computer Number Systems and Arithmetic*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[SD92] Jörg Sauerbrey and Andreas Dietel. Resource requirements for the application of addition chains in modulo exponentiation. In Rainer A. Rueppel, editor, *Advances in Cryptology – EUROCRYPT '92. Proceedings*, volume 658 of *Lecture Notes in Computer Science*, pages 174–182, Balatonfüred, Hungary, May 24–28 1992. Springer-Verlag, Berlin, 1993.

[Sed87] Holger Sedlak. The RSA cryptography processor. In David Chaum and Wyn L. Price, editors, *Advances in Cryptology – EUROCRYPT '87. Proceedings*, volume 304 of *Lecture Notes in Computer Science*, pages 95–105, Amsterdam, The Netherlands, April 13–15 1987. Springer-Verlag, Berlin, 1988.

[SG86] H. Sedlak and U. Golze. An RSA cryptography processor. *Microprocessing and Miccroprogramming*, 18:583–590, 1986.

[SG90] Homayoon Sam and Arupratan Gupta. A generalized multibit recoding of two's complement binary numbers and its proof with application in multiplier implementations. *IEEE Transactions on Computers*, C-39(8):1006–1015, August 1990.

[Sim92a] Gustavus J. Simmons. Contemporary cryptology, a foreword. In *Contemporary Cryptology: The Science of Information Integrity* [Sim92b], pages vii–xv.

[Sim92b] Gustavus J. Simmons, editor. *Contemporary Cryptology: The Science of Information Integrity*. IEEE Press, New York, 1992.

[Slo85] K. R. Sloan, Jr. Comments on "a computer algorithm for calculating the product AB modulo M". *IEEE Transactions on Computers*, C-34(3):290–292, March 1985.

[Spa81] Otto Spaniol. *Computer Arithmetic: Logic and Design*. Wiley Series in Computing. John Wiley & Sons Ltd., Chichester, 1981.

[ST67] Nicolas S. Szabo and R. I. Tanaka. *Residue Arithmetic and Its Applications to Computer Technology*. McGraw-Hill, New York, 1967.

[ST83] D. Simmons and S. E. Tavares. An NMOS implementation of a large number modulo multiplier for data encryption systems. In *Proceedings. Custom Integrated Circuits Conference, CICC 1983*, pages 262–266. IEEE Press, New York, 1983.

[Sti90] D. R. Stinson. Some observations on parallel algorithms for fast exponentiation in GF(2^n). *SIAM Journal on Computing*, 19(4):711–717, August 1990.

[SV93] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Windsor, Ontario, June 29–July 2 1993. IEEE Computer Society Press, Los Alamitos, California, 1993.

[Svo63] Antonin Svoboda. An algorithm for division. *Information Processing Machines*, 9:25–32, 1963. This article is reprinted in [Swa90a], pages 183–190.

[Swa80] Earl E. Swartzlander, Jr., editor. *Computer Arithmetic, volume 21 of Benchmark Papers in Electrical Engineering and Computer Science.*

Dowden Hutchinson & Ross, Inc., Stroudsburg, Pennsylvania, 1980. This volume is reprinted as [Swa90a].

[Swa90a] Earl E. Swartzlander, Jr., editor. *Computer Arithmetic*, volume 1. IEEE Computer Society Press, Los Alamitos, California, 1990. This volume is a reprint of [Swa80].

[Swa90b] Earl E. Swartzlander, Jr., editor. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, Los Alamitos, California, 1990.

[Tak91] Naofumi Takagi. A radix-4 modular multiplication hardware algorithm efficient for iterative modular multiplications. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 35–42, Grenoble, France, June 26–28 1991. IEEE Computer Society Press, Los Alamitos, California, 1991. This article also appears as [Tak92].

[Tak92] Naofumi Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, C-41(8):949–956, August 1992. This article is based on [Tak91].

[Tay84] Fred J. Taylor. Residue arithmetic: A tutorial with examples. *IEEE Computer Magazine*, 17(5):50–62, May 1984.

[Tay85] George S. Taylor. Radix 16 SRT dividers with overlapped quotient selection stages. In Kai Hwang, editor, *Proceedings. 7th IEEE Symposium on Computer Arithmetic*, pages 64–71, Urbana, Illinois, June 4–6 1985. IEEE Computer Society Press, Los Alamitos, California, 1985.

[Tho88] Thorn EMI. RSA evaluation board. Data Sheet Rev. 1.0, 10/88, Thorn EMI Central Research Laboratories, Dawley Road, Hayes, Middlesex, England, 1988.

[Thu73] Edward G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.

[Toc58] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quarterly Journal of Mechanics and Applied Mathematics*, 11 (pt.3):364–384, 1958.

[TY92] Naofumi Takagi and Shuzo Yajima. Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem. *IEEE Transactions on Computers*, C-41(7):887–891, July 1992.

[TYY85] Naofumi Takagi, Hiroto Yasuura, and Shuzo Yajima. High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers*, C-34(9):789–796, September 1985.

[VBR$^+$94] J. Vuillemin, P. Bertin, D. Rochin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: The coming of age. Research article, Digital Equipment Corporation, Paris Research Laboratory, 85, Avenue Victor-Hugo, 92563 Rueil-Malmaison Cedex, France, July 1994. Draft.

[vO92] Paul C. van Oorschot. A comparison of practical public key cryptosystems based on integer factorization and discrete logarithms. In Simmons [Sim92b], chapter 5, pages 289–322.

[VSH89] Stamatis Vassiliadis, Eric M. Schwarz, and Don J. Hanrahan. A general proof for overlapped multiple-bit scanning multiplications. *IEEE Transactions on Computers*, C-38(2):172–183, February 1989.

[VVDJ89] André Vandemeulebroeck, Etienne Vanzieleghem, Tony Denayer, and Paul G. A. Jespers. A single chip 1024 bits RSA processor. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT '89. Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 219–236, Houthalen, Belgium, April 10–13 1989. Springer-Verlag, Berlin, 1990. This article also appears as [VVDJ90].

[VVDJ90] André Vandemeulebroeck, Etienne Vanzieleghem, Tony Denayer, and Paul G. A. Jespers. A new carry-free division algorithm and its application to a single-chip 1024-b RSA processor. *IEEE Journal of Solid-State Circuits*, SC-25(3):748–756, June 1990. This article also appears as [VVDJ89].

[Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, February 1964. This article is reprinted in [Swa90a], pages 114–117.

[Wal91a] Colin D. Walter. Fast modular multiplication using 2-power radix. *International Journal of Computer Mathematics*, 39:21–28, 1991.

[Wal91b] Colin D. Walter. Faster modular multiplication by operand scaling. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91. Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 313–323, Santa Barbara, California, August 11–15 1991. Springer-Verlag, Berlin, 1992.

[Wal93] Colin D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, C-42(3):376–378, March 1993.

[WC81] Robert Willoner and I-Ngo Chen. An algorithm for modular exponentiation. In *Proceedings. 5th IEEE Symposium on Computer Arithmetic*, pages l35–138, Ann Arbor, Michigan, May 18–19 1981. IEEE Computer Society Press, New York, N.Y., 1981.

[WE90] Colin D. Walter and Stephen E. Eldridge. A verification of Brickell's fast modular multiplication algorithm. *Inernational Journal of Computer Mathematics*, 33:153–169, 1990.

[WE92] Neil H. E. Weste and Kamran Eshragian. *Principles of CMOS VLSI Design: A Systems Perspective*. The VLSI Systems Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1992.

[WH91] Ted E. Williams and Mark A. Horowitz. A 160ns 54bit CMOS division implementation using self-timing and symmetrically overlapped SRT stages. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic*, pages 210–217, Grenoble, France, June 26–28 1991. IEEE Computer Society Press, Los Alamitos, California, 1991.

[Wil93] Ted E. Williams. Performance of iterative computation in self-timed rings. *Journal of VLSl Signal Processing*, January 1993.

[Yac90] Y. Yacobi. Exponentiating faster with addition chains. In Ivan B. Damgård, editor, *Advances in Cryptology – EUROCRYPT '90. Proceedings*, volume 473 of *Lecture Notes in Computer Science*, pages 222–229, Aarhus, Denmark, May 21–24 1990. Springer-Verlag, Berlin, 1991.

[Yao76] Andrew Chi-Chih Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100-103, March 1976.

[YS89] Jiren Yuan and Christer Svensson. High-speed CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, SC-24(1):62–70, February 1989.

[Zha93] C. N. Zhang. An improved binary algorithm for RSA. *Computers and Mathematics with Applications*, 25(6):15–24, 1993.

[ZMY88] Chang N. Zhang, Herold L. Martin, and David Y. Y. Yun. Parallel algorithms and systolic array designs for RSA cryptosystem. In Keith Bromley, Sun-Yuan Kung, and Earl Swartzlander, editors, *Proceedings. 1988 IEEE International Conference on Systolic Arrays*, pages 341–350, San Diego, California, 1988. IEEE Computer Society Press, Washington, D.C., 1988.

Appendix A

VICTOR, an Efficient RSA Hardware Implementation

VICTOR
an Efficient RSA Hardware Implementation

Holger Orup
Erik Svendsen
Erik Andreasen

Department of Computer Science, Aarhus University
Ny Munkegade 116
DK-8000 Aarhus C, DENMARK
e-mail: orup@daimi.aau.dk

March 19, 2001

Abstract

The latest improvements of RSA chips are based on progress in implementation technology and strategy i.e. smaller circuits and higher clock frequencies. There has been no improvements in efficiency of the algorithms. The efficiency is here defined as the number of bits produced pr. 1000 clock cycles.

We present algorithms which improve the efficiency by 300%–400%. The main strategy is multiple bit scan and parallel execution of two multiplications. Using these algorithms and the presented hardware architecture a bit rate greater than 90 Kbit/sec can be achieved encrypting 512 bit blocks.

	Ω	Bit rate pr. 512 bits	efficiency
Thorn EMI	24 MHz	29.0 K	1.21
AT & T	15 MHz	19.0 K	1.27
Cryptech	14 MHz	17.0 K	1.21

Table 1: *The three justest RSA chips to date*

1 Introduction

Several implementations or suggestions of how to implement the RSA protocol in hardware have been presented in the past. Brickell made an overview of existing RSA chips. The three implementations with the highest bit rate, when the length of an encryption block is 512 bits, are shown in table 1 [?] [?].

We have defined the efficiency as the number of bits produced pr. 1000 clock cycles. Note that the efficiency of the three implementations are approximately the same, and the difference in bit rate is due to the difference in clock speed. The efficiency as defined here is a performance measure of the algorithm used. On the other hand, the clock rate is a rough performance measure of the technology and methods used for realizing the algorithm in hardware.

Apparently there has been no development of more efficient algorithms suited for hardware implementation. In the following, we will present algorithms for exponentiation and multiplication which result in a higher efficiency than the above mentioned.

2 Exponentiation

The main operation in the RSA protocol is $M^E \bmod N$, where the length of each operand is at least 500 bits. Therefore it is essential to have an efficient exponentiation algorithm. The most commonly used algorithm is named *Russian Peasant* [?]. Below is shown a variant in which E is read from the least significant bit. The i'th bit of E is denoted e_i.

```
Algorithm:    Modulo exponentiation.
Stimulation:  $E$, $M$, $N$, where $E \geq 0$ and $0 \leq M < N$.
Response:     $X = M^E \bmod N$
Method:       $i := 0$
              $X := 1$;
              WHILE $i < n$ DO
                  IF $e_i = 1$ THEN $X := (X \cdot M) \bmod N$ END;
                  $M := (M \cdot M) \bmod N$;
                  $i := i + 1$;
              END;
```

Algorithm 1: *Variant of Russian Peasant for module exponentiation*

If we denote the length of M, E and N by n the time complexity is:

$$T[\mathrm{Exp}, n] = \frac{2}{3} n T[\mathrm{Mult}, n]$$

Assuming it is possible to perform *two* multiplications in parallel, this is indeed possible because the three statements in the loop do not depend on each other, the complexity is:

$$T[\mathrm{Exp}, n] = n T[\mathrm{Mult}, n]$$

This gives a 33% time reduction compared with the variant with one multiplication unit.

3 Multiplication

To implement the exponentiation algorithm mentioned above, we need an efficient way to perform modulo multiplication. Several algorithms have been presented [?] [?] [?] [?]. None of them are able to carry out a multiplication with fewer than n full additions of n-bit words.

```
$S := 0$; $i := n' - 1$;
WHILE $i \geq 0$ DO
        $S := (2^k S + a_i B) \bmod N$;
        $i := i - 1$;
END;
```

Algorithm 2: *Serial-parallel multiplication with integrated modulo reduction.*

The usual way of multiplying is by scanning the multiplier one bit at a time and conditionally accumulating the multiplicand parallelly. Assume we scan the multiplier k bits at a time, corresponding to base 2^k we can express the serial-parallel multiplication scheme as in algorithm 2. In this algorithm S is the accumulator, n' is the number of digits in base 2^k, a_i is digit number i of the multiplier, B the multiplicand and N the modulus. The multiplier is scanned from the most significant digit.

```
$S := 0$; $i := n' - 1$;
WHILE $i \geq 0$ DO
        $q :=$ estimate($S$ div $N$);
        $S := 2^k S + a_i B - 2^k q N$
        $i := i - 1$;
END;
Correction of S;
```

Algorithm 3: *Module multiplication with quotient estimation.*

The modulo reduction can be carried out by subtracting N from S until $S \in [0; N[$. The maximal number of subtractions will be $2^k + 2^k - 2$, because $a_i \in [0; 2^k - 1]$ and $B \in [0; N]$. Even though the number of subtractions is limited, this method is rather slow. Instead we can estimate the quotient, S div N, belonging to $[0; 2^{k+1} - 2]$ and carry out the reduction in *one* subtraction. This is shown in algorithm 3. Note that this method is only feasible if we are able to generate the products $a_i B$ and qN rapidly. We could for example precalculate all the possible values of $a_i B$ and qN and save them in a table. According to Barrett [**?**], the quotient estimate can be found by multiplying a few of the most significant bits of the dividend S and the reciprocal of the divisor N. We assume that the necessary amount of bits of $\frac{1}{N}$ is part of the input to the chip. If q is to have an accuracy of x bits, then by using $x + 2$ bits from S and $\frac{1}{N}$ we get an estimate which at the most is one less than the exact quotient.

In algorithm 3 the accumulator S does not necessarily belong to the interval $[0; N[$ after each iteration. This does not matter, as long as S belongs to the same residue as $S \bmod N$, and S does not diverge. But after the loop S has to be corrected by subtracting N until S belongs to the correct interval. It is proven in [**?**] that $S \in [0; (3 \cdot 2^k - 1)N]$ after each iteration. The range of q is therefore $[0; 3 \cdot 2^k - 1]$. As we shall see later, we can construct hardware that generates qN efficiently if the range of q belongs to $[0; 42]$, this means the scan factor k is limited to 3.

We are able to reduce the range of q, the idea is as follows: if we estimate $\frac{S}{2}$ div N instead of S div N, the range of the quotient is apparently halved. The modulo reduction is performed by subtracting $2 \cdot 2^k qN$ instead of $2^k qN$. However, the accuracy of the quotient estimate is hereby reduced, implying an increase of the range of S. A closer analysis [**?**] shows that if we estimate $\frac{S}{2^r}$ div N we get a minimal range of q when $k \leq r : [0; 2^{k+1}]$. This means that the scan factor can be increased to 4. The final algorithm for module multiplication is shown as algorithm 4. Note that the final corrections can be made by iterating two extra times while setting $a_i = 0$ and further more assuming $r = k$.

The final result is read from S discarding the $2k$ least significant bits. Note that this result belongs to the interval $[0; 2N[$. A further reduction is not necessary. When the exponentiation algorithm terminates, the result will also belong to $[0; 2N[$, here a reduction is necessary. This reduction is easily carried out while outputting the result serially. The correctness of the algorithm is proven in [**?**]. The time complexity is:

$$T[\text{Mult}, n] = (\lceil \frac{n+1}{k} \rceil + 2) T[\text{loop}]$$

In the rest of this paper we will describe how to perform the central operation of the loop: $S := 2^k S + a_i B - 2^{k+r} qN$. To do this we have to take a closer look at the hardware architecture of the multiplication unit.

```
Algorithm:     Module multiplication.

Stimulation:   $A = a_{n'-1}a_{n'-2}\cdots a_0a_{-1}a_{-2}$,
               where $a_i \in [0; 2^k - 1], a_{-1} = a_{-2} = 0$ and $n' = \lceil \frac{n+1}{k} \rceil$;
               $B$, where $B \in [0; 2N[$;
               $N$, where $N \in ]2^{n-1}; 2^n[$;
               $k$, where $k \geq 3$;
               $r$, where $r = k$.

Response:      $S$ div $2^{2k} \equiv_N AB$ and
               $S$ div $2^{2k} \in [0; .2N[$.

Method:        $S := 0$; $i := n' - 1$
               WHILE $i \geq -2$ DO
                   $q = \text{estimate}(S \text{ div } 2^r, N)$;
                   $S := 2^k S + a_i B - 2^{k+r} qN$;
                   $i := i - 1$;
               END;
```

Algorithm 4: *Module multiplication.*

3.1 Hardware architecture

The multiplication unit consists of circuits for generating the values $-qN$ and a_iB. Each circuit returns the result represented in two words, i.e. the value a_iB is represented as a_iB_s and a_iB_c, where $a_iB_s + a_iB_c = a_iB$. Similarly the accumulator S is represented in two words S_s and S_c. The main task of the loop is now to add six words together and represent the sum as two words S'_s and S'_c Using the carry save addition technique this is easily done with four rows of fulladders as shown in figure 5. The critical path of the multiplication unit is calculating the quotient estimate, generating $-qN$ followed by the delay of two fulladders. To be able to use the parallel version of the exponentiation algorithm we have to perform two multiplications i parallel. This can be achieved by pipelining the circuit and adding an extra A register.

It is not necessary to duplicate the B register since the two multiplications always have a common operand. See algorithm 1.

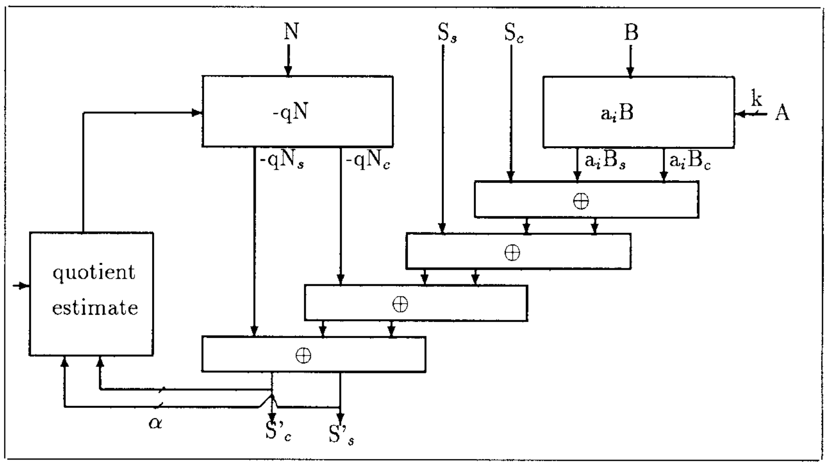


Figure 5: Hardware architecture of the multiplication unit.

3.2 Generating $\mathbf{a_iB}$ and $-\mathbf{qN}$

To compute $-qN$ we again use the carry save technique. Observe that all numbers in $[0;42]$ can be expressed as a sum of three powers of two. Table 2 shows which values, α, β and γ, are needed to compute $-qN = \alpha N + \beta N + \gamma N$. The values αN, βN and γN are generated through a selection network, and added through a row of fulladders as shown in figure 6. Here again the result is renresented in two words $-qN_c$ and $-qN_s$.

The computation of a_iB is performed following the same principle.

3.3 Quotient estimation

The quotient estimate q is calculated by adding the δ most significant bits of S'_c and S'_s and then multiplying the sum with the ε most significant bits of $\frac{1}{N}$. The quotient can then be found by discarding the $\delta + \varepsilon - (k+2)$ least significant bits of the product, where k is the scan factor.

q	α	β	γ	q	α	β	γ	q	α	β	γ	q	α	β	γ	q	α	β	γ
0	0	0	0	10	-16	4	2	20	-16	-4	0	30	-32	0	2	40	-32	-8	0
1	0	0	-1	11	-16	4	1	21	-16	-4	-1	31	-32	0	1	41	-32	-8	-1
2	0	-4	2	12	-16	4	0	22	-32	8	2	32	-32	0	0	42	-32	-8	-2
3	0	-4	1	13	-16	4	-1	23	-32	8	1	33	-32	0	-1				
4	0	-4	0	14	-16	0	2	24	-32	8	0	34	-32	0	-2				
5	0	-4	-1	15	-16	0	1	25	-32	8	-1	35	-32	-4	1				
6	-8	0	2	16	-16	0	0	26	-32	4	2	36	-32	-4	0				
7	-8	0	1	17	-16	0	-1	27	-32	4	1	37	-32	-4	-1				
8	-8	0	0	18	-16	-4	2	28	-32	4	0	39	-32	-4	-2				
9	-8	0	-1	19	-16	-4	1	29	-32	4	-1	39	-32	-8	1				

Table 2: q expressed as the sum of α, β and γ

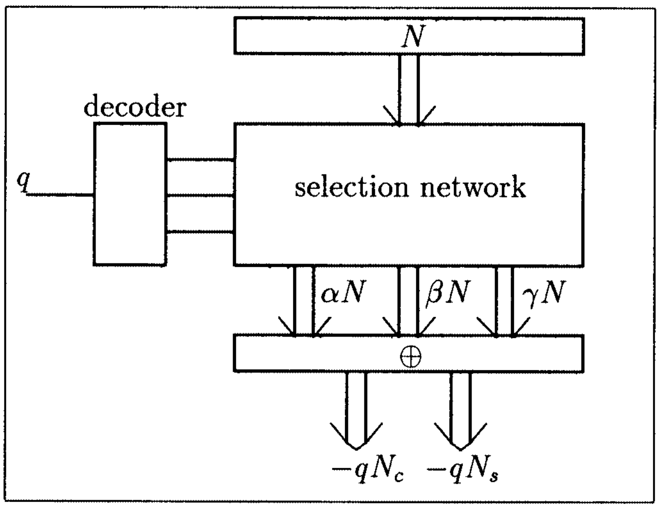


Figure 6: Unit for generating $-qN$

Earlier we have given an upper bound of $q = 2^{k+1}$, and this restricted $k \leq 4$. In [?] we have investigated the interdependency of $q, \delta, \varepsilon, k$ and found an expression that gives a smaller upper bound for q:

$$q_{max} = \frac{2^k(2^{k+3-\delta} + 1 + 2^{1-k})}{1 - 2^{k-\varepsilon}}$$

Table 3 shows that we can achieve an upper bound equal to 42 with $k = 5$ by selecting $\delta = 10$ and $\varepsilon = 11$. This scan factor is optimal for the presented hardware architecture because the maximal value of a_i will exceed 42 if k is greater than 5. Simulations indicate that an even lesser upper bound for q can be found, which means that δ and ε can be reduced, giving a simpler circuit.

k	δ	ε	$\lfloor q_{max} \rfloor$
4	8	8	27
4	8	9	26
4	9	9	22
4	9	10	22
5	9	9	51
5	9	10	50
5	10	10	43
5	10	11	42

Table 3: Upper bounds for the quotient estimate.

4 Performance

The critical path of the multiplication unit has been designed in a 2μ process. Simulations show that a loop in the multiplication unit takes less than 85 ns. In a pipelined version each loop takes two clock cycles, thereby giving a clock period less than 50 ns., corresponding to a clock frequency of more than 20 MHz. The layout shows high regularity and the area is estimated at approximately 100 mm^2.

The efficiency of the algorithms is:

$$\frac{n \cdot 1000\text{bits}}{2 \cdot n(\frac{n+1}{k} + 2)\text{cycles}}$$

For $n = 512$ we achieve an efficiency of 3.8 for $k = 4$ and 4.8 for $k = 5$. The bit rates for a clock frequency of 20 MHz are 78 Kbit/sec and 97 Kbit/sec respectively.

5 Conclusion

We have presented a way to speed up a well known exponentiation algorithm by performing two multiplications in parallel, and we have shown how these multiplications can be performed efficiently using multiple bit scan. Further more we have developed a highly regular hardware architecture, based on the carry save addition technique, implementing the multiplication algorithm with a scan factor of up to 5.

Currently a prototype is being developed at the Computer Science Department, Aarhus University.

References

[Bar86] Paul Barrett. Implementing the Riverst Shamir and Adleman public key encryption system on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86*, pages 311–323, 1986.

[Bla83] G.R. Blakely. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Trans. Computers*, C-32:497–500, 1983.

[Bri89] Ernest F. Brickell. A Survey of Hardware Implementations of RSA. In *CRYPTO '89*, 1989.

[EMI88] THORN EMI. RSA Evaluation Board. Technical Report 10, Thorn EMI Central Reasearch Laboratories, 1988.

[HDVG88] Frank Hoornaert, Marc Decroos, Joos Vandewalle, and René Govaerts. Fast RSA-Hardware: Dream or Reality ? In *Advances in Cryptology - EURO-CRYPT '88*, pages 257–264, 1988.

[Knu69] Donald E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, volume 2. Addison-Westley, 1969.

[ORSP86] G.A. Orton, M.P. Roy, P.A. Scott, and L.E. Peppard. VLSI implementation of public-key encryption algorithms. In *Advances in Cryptology - CRYPTO '86*, pages 277–301, 1986.

[OS90] Holger Orup and Erik Svendsen. VICTOR. Forbedringer og videreudviklinger af VICTOR - en integreret kreds til understøttelse af RSA-kryptosystemer. Computer Science Department of Aarhus University - Internal report, 1990.

[OSA90] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR - Teoretiske og eksperimentelle undersøgelser af algoritmer til understøttelse af RSA-kryptosystemer med henblik på VLSI design. Master's thesis, Computer Science Department of Aarhus University, 1990.

Appendix B

A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N

A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N

Holger Orup
Computer Science Department
Aarhus University
DK-8000 Aarhus C, DENMARK
e-mail: orup@daimi.aau.dk

Peter Kornerup
Dept. of Math. and Computer Science
Odense University
DK-5230 Odense M, DENMARK
e-mail: kornerup@imada.ou.dk

March 19, 2001

Abstract

In a class of crypto systems fast computation of modulo exponentials is essential. The popular RSA protocol uses operands of more than 500 bits to achieve a suficient security. We present a parallel version of a well known exponentiation algorithm that halves the worst case computing time. It is described how a high radix module multiplication can be implemented by interleaving a serial-parallel multiplication scheme with an SRT division scheme. The problems associated with high radices are eficiently solved by the use of a redundant representation of intermediate operands. We show how the algorithms can be realized as a highly regular VLSI circuit. Simulations indicate that a radix 32 implementation of the algorithms is able of computing 512 bit operand exponentials in 3.2 msec. This is more than 5 times faster compared to other known implementations.

1 Introduction

The concept of two-key crypto systems was introduced by Diffie and Hellman [5] in 1976. In 1978 Riverst, Shamir and Adleman [13] published an encryption scheme based on computing exponentials. The RSA scheme realizes encryption as put forth by Diffie and Hellman. Both encryption of a message and decryption of an encrypted message are done by computing an exponential $M^E \bmod N$, $M \in [0, N[$. The message is denoted M and the key (E, N). Modulus N is a product of two very large primes and the security of the system depends on the length of the keys. To achieve a sufficient security key lengths of 500-600 bits are necessary.

For an implementation of the RSA protocol it is crucial to calculate modulo exponentials in a rate corresponding to the transmission rate between transmitter and receiver to avoid the encryption to be a bottle neck in the communication system. Brickell [4] has made a survey of hardware implementations of RSA. In his paper chips from Cryptech [7] are the fastest, with a computing time of 512 bit messages corresponding to a rate of $17\frac{\text{Kbit}}{\text{sec}}$. Thorn EMI [15] has made chips that encrypt 512 bit messages at $29\frac{\text{Kbit}}{\text{sec}}$.

In this paper we will present a method for obtaining rates of more than $150\frac{\text{Kbit}}{\text{sec}}$. Section 2 describes how we construct a new and faster algorithm for performing exponentials by splitting the computation into parallel multiplications. The implementation of a high radix module multiplication is elaborated in sections 3, 4 and 5, where it is shown how to interleave a multiplication with a SRT division scheme and how the algorithm can be realized in VLSI design. Section 6 discusses the performance of a VLSI implementation. The summary indicates how the methods can be generalized to even higher radices.

2 Exponentiation

A commonly known algorithm for performing an expo-nentiation is named *Russian Peasant* [8]. It performs the computation as a number of multiplications and squarings proportional to the bit length of the exponent. The algorithm can be modified to perform a modulo exponentiation by substituting the multiplications and squarings for modulo multiplications and module squarings [13]. Below is shown a variant in which the exponent E is read from the least significant bit. The i'th bit of E is denoted e_i. In the curly brackets is an invariant for the loop.

If we denote the bit length of M,E and N by n the worst case time is

$$T[\text{Exp}, n] = 2nT[\text{Mult}, n]$$

and the average time is

$$T[\text{Exp}, n] = \frac{3}{2}nT[\text{Mult}, n],$$

where it is assumed that the computing time for squaring and multiplication is identical.

Observing that the three statements in the loop are independent of each other it is possible to speed up the algorithm by performing *two* multiplications in *parallel*. In this way the computing time is reduced to

$$T[\text{Exp}, n] = nT[\text{Mult}, n], \qquad (1)$$

which is independent on the number of 1-bits in E. It is hard to imagine how an exponentiation can be performed with less than n squarings.

```
Algorithm:
    Module exponentiation.
    Stimulation: $E, M, N$, where $E \leq 0$
                 and $0 \leq M < N$,
Response: $X = M^E \bmod N$
Method: $i := 0$
    $X := 1; Y := M$;
    WHILE $i < n$ DO
        $\{*X \cdot Y^{E \text{ div } 2^i} \equiv_N M^E*\}$
        IF $e_i = 1$ THEN
            $X := (X \cdot Y) \bmod X$
        END;
        $Y := (Y \cdot Y) \bmod N$;
        $i := i + 1$;
    END;
```

Algorithm 1: *Variant of Russian Peasant for module exponentiation.*

Note that by distributing the modulo reduction to squaring and multiplication the bitlength of intermediate operands is bounded. This makes the exponentiation algorithm feasible to implement for large values of n.

3 Multiplication

To implement the exponentiation algorithm mentioned above we need an efficient way to perform modulo multilication. Several algorithms have been presented [3] [2] [1] [10] [7]. All of them use radix 2. We will follow the approach in e.g. [3], where the modulo reduction is further distributed in the algorithm for multiplication. A similar approach is followed in [9] where a radix 4 algorithm is elaborated.

The usual way of multiplying is by scanning the multiplier serially from the *least* significant digit and parallelly adding a multiple of the multiplicand followed by a *right* shift of the partial product [14]. This gives a maximal carry ripple length of the parallel additions corresponding to the length of the multiplicand.

In the algorithm shown below we scan the multiplier from the *most* significant digit and the partial product is *left* shifted. In an ordinary multiplication this would result in a maximal carry ripple length equal to the sum of the multiplier length and the multiplicand length. But since we perform a modulo reduction on the partial product in every iteration, the length of the intermediate operands will be limited to n plus a few digits. Assume we scan the multiplier k bits at a time, corresponding to processing a digit in radix 2^k, we can express the serial-parallel multiplication scheme as in Algorithm 2.

```
$S := 0; i := n' - 1;$
WHILE $i \geq 0$ DO
        $S := (2^k S + a_i B)$ mod $N$
        $i := i + 1;$
END;
```

Algorithm 2: *Serial-parallel multiplication with integrated modulo reduction.*

In this algorithm S is the accumulator, n' the number of radix 2^k digits, a_i is digit number i of the multiplier, B the multiplicand and N the modulus.

The module reduction can be carried out by *interleaving the multiplication with a division.* Division is usually performed by inspecting the partial remainder and subtracting a multiple of the divisor, followed by a *left* shift of the resulting partial remainder. In Algorithm 3 the variable S has the dual role of a partial remainder in a SRT division scheme [14] and of a partial product in a serial-parallel multiplication. This is the reason why we want to scan the multiplier from the most significant digit and left shift the partial product. Note that this method is only feasible if we are able to generate the multiples $a_i B$ and qN rapidly.

```
$S := 0; i := n' - 1;$
WHILE $i \geq 0$ DO
        $q :=$ Estimate($S$ div $N$);
        $S := 2^k S + a_i B - 2^k aN$
        $i := i - 1;$
END;
Correction of $S$;
```

Algorithm 3: *Module multiplication with quotient estimation.*

Instead of calculating the *exact* value of the quotient digit, which is a tedious task for long operands, we *estimate* a value of q. This, of course, implies that the final result is not necessarily completely module reduced, and a correction must be performed after the loop by subtracting N until S belongs to the correct interval. It is then required that the precision of the estimate is chosen such that the range of S does not diverge during a computation. Assuming that

$$\begin{array}{lcl} a & \in & \{0, 1, \ldots, 2^k - 1\} \\ B & \in & [0; 2N[\\ q & \in & \{0, 1, \ldots, q_{max}\} \end{array}$$

we can derive a range restriction, which must be satisfied by the estimated q, in the following way

$$\begin{array}{rcl} 2^k S + aB - 2^k qN & \leq & q_{max} N \\ S - qN & \leq & \dfrac{q_{max} N - aB}{2^k} \\ S - qN & \leq & \dfrac{q_{max} - 2(2^k - 1)}{2^k} N. \end{array} \quad (2)$$

As we shall see later we can construct hardware that generates multiples qN efficiently if the range of q belongs to $[0; 42]$. Since q is non negative it is required that S is non negative. We achieve this by restricting $S - qN$ to be non negative. The restriction then implies that the maximal radix 2^k is 16, i.e. the scan factor is limited to 4.

However, we are able to increase the scan factor. The idea is to reduce the contribution of the multiple aB in (2) by choosing a larger divisor. Recall that we just want to modulo reduce the partial product, so we can choose an easily generated multiple of N, e.g. $2^r N$. This gives the following range restriction

$$\begin{array}{rcl} 2^k S + aB - 2^k q 2^r N & \leq & q_{max} 2^r N \\ S - q2^r N & \leq & \dfrac{q_{max} 2^r N - aB}{2^k} \\ S - q2^r N & \leq & \dfrac{q_{max} 2^r - 2(2^k - 1)}{2^k} N. \end{array} \quad (3)$$

Since we want to generate multiples aB with the same hardware as qN, a is limited to 42 and the maximal radix is therefore 32, corresponding to a scan factor of 5. To achieve this, restriction (3) implies that r must be greater than or equal to 1.

We are now ready to present the final algorithm for modulo multiplication. Note that the final corrections can be made by iterating two extra cycles, while setting $a_i = 0$ and furthermore assuming $r = k$.

The final result is read from S discarding the $2k$ least significant bits, and belongs to the interval $[0; 2N[$. A further reduction is not necessary since, according to the stimulation conditions, we can directly start up a new multiplication in the exponentiation algorithm with inputs in the interval $[0; 2N[$. When the exponentiation algorithm terminates the result will also belong to $[0; 2N[$, here a reduction is necessary. This reduction is easily carried out while outputting the result serially. The correctness of Algorithm 4 is proven in [11]. The time complexity is:

$$T[\text{Mult}, n] = (\left\lceil \frac{n+1}{k} \right\rceil + 2) T\ [\text{iteration}] \quad (4)$$

In the rest of this paper we will describe how to perform the central operations of the loop, and take a closer look at the hardware architecture of the multiplication unit.

Algorithm: Modulo multiplication.

Stimulation: $A = a_{n'-1}a_{n'-2}\cdots a_0a_{-1}a_{-2}$,
where $a_i \in [0; 2^k - 1]$,
$a_{-1} = a_{-2} = 0$,
$n' = \lceil \frac{n+1}{k} \rceil$;

B, where $B \in [0; 2N[$;
N, where $N \in]2^{n-1}; 2^n[$;
k, where $k \geq 3$;
r, where $r = k$.

Response: S div $2^{2k} \equiv_N AB$ and
S div $2^{2k} \in [0; 2N[$

Method:

```
$S := 0$; $i := n' - 1$
WHILE $i \geq 2$ DO
   {*$S \equiv_N (A$ div $2^{k(i+1)})$
   $\cdot B$*}
   $q :=$ Estimate($S$ div $2^r N$);
   $S := 2^k S + a_i B - 2^{k+r} qN$
   $i := i - 1$;
END;
```

Algorithm 4: *Modulo multiplication.*

4 Calculation of $2^kS + aB - q2^{k+r}N$

Because we are dealing with very long operands we use redundant carry save adders. This implies that the result of a multiplication is represented in *two* words. To avoid an area or time consuming carry-completing adder in the circuit we represent the multipliers and multiplicands in carry save form during the *complete* computation of an exponential. The modulus is represented in 2'complement form and in one word. We get the expression

$$2^k(S_s + S_c) + a(B_s + B_c) - q2^{k+r}N \quad (5)$$

The multiplier digit a in radix 32 is recoded from the carry-save representation of A through two levels. The 5 digit positions of A is interpreted as three digits in radix 4 : d_2, d_1, d_0 where $d_2 \in [0; 2]$ and $d_1, d_0 \in [0; 6]$. These are first recoded into digit sets $[-1; 1]$ repectively $[-1; 3]$, possibly generating and absorbing carries. Secondly these digits are recoded into $\alpha_2, \alpha_1, \alpha_0$ with $\alpha_i \in [-1; 2]$, where α_2 may absorb a carry without generating a carry out. Hence the radix 32 digit a is represented as:

$$a = 4^2\alpha_2 + 4\alpha_1 + \alpha_0, \; a_i \in \{-1, 0, 1, 2\}, \quad (6)$$

thus aB can be computed as the sum of three shifted versions of B in a multiplexing network as shown in Figure 5. The result is again represented in carry save form $(aB)_s$ and $(aB)_c$. The computation of $-qN$ is performe the same way, noting that $q \in [0; 42]$ can be represented in radix 4 as in (6).

Expression (5) is now expanded to the form

$$\begin{aligned} &2^k(S_s + S_c) \\ &+((aB_s)_s + (aBH_s)_c) + ((aB_c)_s + (aB_c)_c)(7) \\ &+2^{k+r}((-qN)_s + (-qN)_c) \end{aligned}$$

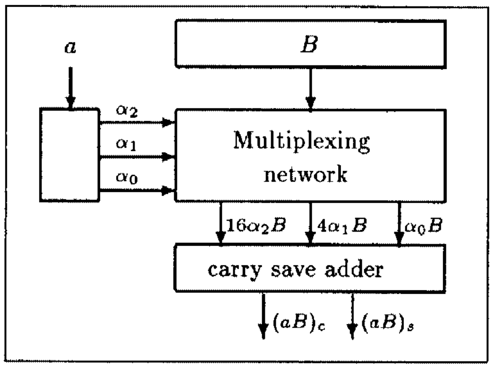


Figure 5: *Unit for generating a multiple.*

The cost of keeping the operands in carry save form is that we get an expression of eight terms instead of three terms. To reduce the hardware we perform the computation in a pipelined fashion, and share a single unit for generating multiples and a single 4-2 adder for summing terms. The adder is constructed of two carry save adders. In Figure 6 the timing of the computation of expression (7) is illustrated. Each row shows the activity of a hardware component by enclosing the activity in a box. In the left column the components are described.

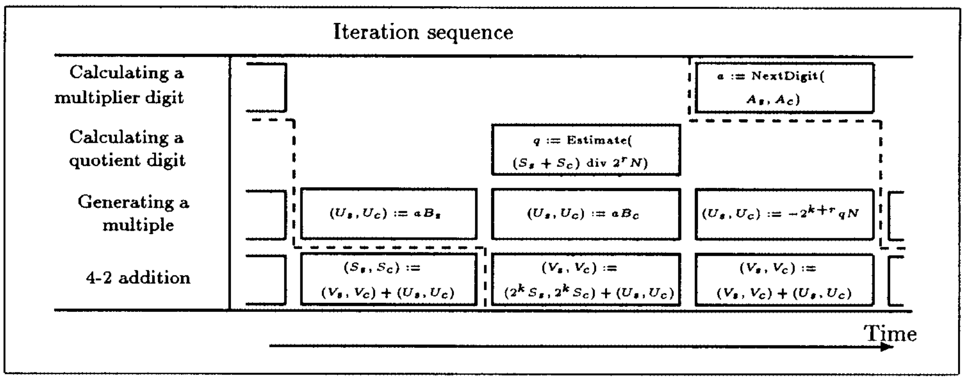

Iteration sequence
Calculating a multiplier digit
Calculating a quotient digit
Generating a multiple
4-2 addition
a := NextDigit(A_s, A_c)
q := Estimate($(S_s + S_c)$ div 2^rN)
$(U_s, U_c) := aB_s$
$(U_s, U_c) := aB_c$
$(U_s, U_c) := -2^{k+r}qN$
$(S_s, S_c) := (V_s, V_c) + (U_s, U_c)$
$(V_s, V_c) := (2^kS_s, 2^kS_c) + (U_s, U_c)$
$(V_s, V_c) := (V_s, V_c) + (U_s, U_c)$
Time

Figure 6: *Timing diagram for an iteration in the multiplication algorithm. The iterations are overlapped.*

A single iteration of the loop in Algorithm 4 is computed by six cycles of the hardware. The iterations are overlapped, as illustrated by the dashed lines, resulting in a throughput of one iteration per 3 hardware cycles. (U_s, U_c), (V_s, V_c) and (S_s, S_c) denote registers in the pipeline for saving results in carry save form. The computation is performed from left to right.

5 Estimation of S div 2^rN

Several implementations or suggestions of how to implement the quotient estimation have been presented in the past. According to [1] the estimate can be found by multiplying a few of tbe most significant digits of the partial remainder S and the reciprocal of the divisor 2^rN. This assumes that the necessary amount of digits of $\frac{1}{2^rN}$ is part of input to the chip or that it is computed on the chip. The standard approach for SRT division, or as extended in [6], is to use table lookup, implemented as a PLA circuit. This method seems to be infeasible for radices as high as in this paper. We will follow the approach in [3] which is based on what we identify as a "parallel exhaustive search" for the quotient digit. In parallel we compute the sign of resulting partial remainders when the quotient digit assumes all possible values, i.e. we perform in parallel

$$S - q2^rN,\ q \in \{0, 1, \ldots, q_{max}\}$$

where S is in carry save form. The sign is detected as the carry out of the computation. A high carry out indicates a non negative partial remainder. Then we determine the quotient digit as the smallest value of q which results in a positive remainder. Since the sign computation involves a carry ripple we only use a few of the most significant digits of S and $-q2^rN$, and con-sequently the quotient digit will just be an *estimate.* The necessary number of digits is determined by the range restriction (3). In Figure 7 is shown a single cell in the quotient estimation unit for calculating the sign of $S - q2^rN$. There is one cell for each possible value of q. In the figure we denote by p the position of the most significant bit, the sign bit, in S_s, S_c and in $-q2^rN$.

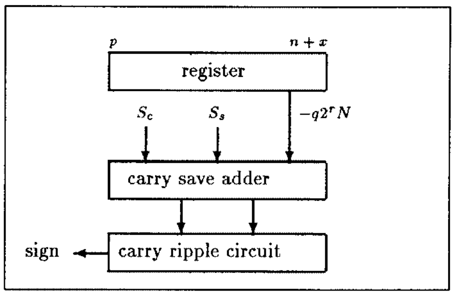


Figure 7: *A cell in the quotient estimation unit.*

According to restriction (3) $p = n + r + \lceil \log_2 q_{max} \rceil$ and for $q_{max} = 42$ this gives $p = n+r+k+1$. n is the bit length of N, and $n+x$ denotes the position of the least significant bit in the quotient estimation. Since the weights of the discarded parts of S_s S_c and $-q2^rN$ are all positive and belongs to $[0; 2^{n+x}[$, this computation gives the following range for the estimated value of q:

$$\begin{aligned} S - q2^rN &= S' + (-q2^rN)' + \varepsilon \\ &\in [0; 2^rN + 3 \cdot 2^{n+x}[, \end{aligned}$$

where S' and $(-q2^rN)'$ denotes S and $-q2^rN$ with the bits from 0 to $n+z-1$ set to zero. Now x can be determined from the range restriction (3), where the redundant digit set $\{0, 1, \ldots, q_{max}\}$ is assumed for the multiplier:

$$\begin{aligned} 2^rN + 3 \cdot 2^{n+x} &< \frac{q_{max}2^r - 2q_{max}}{2^k}N \\ 2^x &< \frac{1}{6}(\frac{2^r - 2}{2^k}q_{max} - 2^r), \end{aligned}$$

where the smallest value 2^{n-1} of N has been inserted. Substituting 5 for r and k, and 42 for q_{max} we get $x \leq 0$. Thus the necessary number of bits in the quotient estimation is $p - (n+x) + 1 = 12$ in the case of radix 25.

This method for quotient estimation seems very area consuming but compared to the size of the multiple generating unit and the 4-2 adder for operands longer than 500 bits this area is reasonable. As usual in VLSI design a high degree of concurrency results in faster circuits at the cost of area.

6 Performance

Equation (1) and (4) gives an expression for the computing time of an exponentiation

$$T[\mathrm{Exp}, n] = n(\left\lceil \frac{n+1}{k} \right\rceil + 2)T \text{ [iteration]}.$$

As explained in Figure 6 an iteration in the multiplication loop is performed in three cycles through the hard-ware. Anticipating that the quotient estimation unit is the critical path in the circuit we get

$$T[\mathrm{Exp}, n] = n(\left\lceil \frac{n+1}{k} \right\rceil + 2)3T$$

$$\text{[quotient estimation]}.$$

A cell of the quotient estimate unit has been designed in a 2μ CMOS process and simulations shows a delay less than 20 ns. For $n = 512$ and $k = 5$ we achieve a computing time of 3.2 ms, corresponding to a bit rate of

$$\frac{n}{T[\mathrm{Exp}, n]} = 159\frac{\mathrm{Kbit}}{\mathrm{sec}}.$$

Compared to the hardware implementation from Thorn EMI [15] with a bit rate of 29 $\frac{\mathrm{Kbit}}{\mathrm{sec}}$ this design improves the speed by a factor of more than 5.

The calculation on computing time assumes that we have implemented the parallel version of the exponentiation algorithm. This can be done in two ways: By replicating the multiplication unit or by pipelining a single multiplication unit. With respect to area the last approach is preferable. Observing that the two parallel multiplications have the modulus N and the multiplicand B in common we only need to add extra registers for a multiplier in carry save form and latches to implement the pipeline. An iteration in the multiplication loop now consist of six clock cycles at approximately 10ns. Clock frequencies as high as this can be hard to achieve. A way to avoid the use of a clock to synchronize the circuit is to use self-timed circuit schemes [16].

The VLSI design of the hardware components shows high regularity and the area for wiring is minimized through the use of carry save adders. At the expense of regularity and area, the speed of quotient estimation can be increased by replacing the carry ripple circuit in Figure 7 by carry look-ahead circuit.

We can obtain a rough estimate of the area by comparing the proposed architecture to the one described in [12], which has been laid out using a silicon compiler: The area is approximately 200 mm^2 in a 1.2μ CMOS process technology for a chip capable of modulo exponentiating 561 bit operands. The architecture presented here include *one* unit for generating multiples and one 4-2 adder where [12] include two of each and additionally a 561 bit ripple adder. The adder is used to convert the result of a multiplication from carry save form to a non redundant representation. Taking into account the extra latches for pipelining a multiplication unit we believe that the area will be less than 200 mm^2 in a 1.2μ process if we use the silicon compiler and its library cells. We can reduce the area significantly by making a full custom layout of the design, since the library cells are designed in a conservative manner using static registers and static logic gates. Another way to reduce the area (and the computing time) is by choosing a smaller process technology, e.g. 0.8μ which approximately halves the area.

All of the fastest implementations in Brickells survey [4] include more than one chip. Cryptechs 712 bit solution [7] comprise 6 chips, where each chip contains a datapath for 120 bit. Thorn EMIs 768 bit solution [15] comprise a controller chip and 3 datapath chips for 256 bit each.

7 Summary

We have presented a way to speed up a well known exponentiation algorithm by performing two multiplications in parallel, and we have shown how these multiplications can be performed efficiently using high radices. Further more we have developed a highly regular hard-ware architecture, based on the redundant carry save addition technique, implementing the multiplication algorithm with a radix of 32. Simulations indicates a resulting speed improvement of more than 500% compared to other known implementations.

Currently we are working on generalizing the multiplication to even higher radices. By expressing the quotient and multiplier in a symmetric redundant digit set it seems simple to modify the hardware architecture to radix 64.

References

[1] Paul Barrett. Implementing the Riverst Shamir and Adleman public key encryption system on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86*, pages 311–323, 1986.

[2] G.R. Blakely. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Trans. Computers,* C-32:497–500, 1983.

[3] Ernest F. Brickell. A fast modular multiplication algorithm with applications to two key cryptography. In D. Schaum, R.L. Riverst, and A.T. Sherman, editors, *Advances in Cryptology, Proceed-*

ings of Crypto '82, pages 51–60, New York, 1982. Plenum Press.

[4] Ernest F. Brickell. A Survey of Hardware Implementations of RSA. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89*, pages 368–370. Springer-Verlag, 1990.

[5] W. Diffie and M.E. Hellman. New directions in cryptography. In *IEEE Trans. on Info. Theory*, volume IT-22(6), pages 644–654, Nov. 1976.

[6] Jan Fandrianto. Algorithm for high speed shared radix 8 division and radix 8 square root. In *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 68–75. IEEE, 1989.

[7] Frank Hoornaert, Marc Decroos, Joos Vandewalle, and René Govaerts. Fast RSA-Hardware: Dream or Reality ? In *Advances in Cryptology - EUROCRYPT '88*, pages 257–264, 1988.

[8] Donald E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, volume 2. Addison-Wesley, 2. edition, 1981.

[9] Hikaru Morita. A Fast Modular-multiplication Algorithm based on a Higher Radix. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89*, pages 387–399. Springer-Verlag, 1990.

[10] G.A. Orton, M.P. Roy, P.A. Scott, and L.E. Peppard. VLSI implementation of public-key encryption algorithms. In *Advances in Cryptology - CRYPTO '86*, pages 277–301, 1986.

[11] Holger Orup and Erik Svendsen. VICTOR. Forbedringer og videreudviklinger af VICTOR - en integreret kreds til understøttelse af RSA-kryptosystemer. Computer Science Department of Aarhus University - Internal report, 1990.

[12] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an Efficient RSA Hardware Implementation. In I.B. Damgård, editor, *Advances in Cryptology - EUROCRYPT '90*, pages 245–252. Springer-Verlag, 1991.

[13] Ronald L. Riverst, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, volume 21, pages 120–126, Feb. 1978.

[14] Norman R. Scott. *Computer Number Systems & Arithmetic.* Prentice-Hall, 1985.

[15] THORN EMI. RSA Evaluation Board. Technical Report 10, Thorn EMI Central Reasearch Laboratories, 1988.

[16] T.E. Williams, M. Horowitz, R.L. Alverson, and T.S Yang. A self-timed chip for division. In Paul Losleben, editor, *Advanced Research in VLSI. Proceedings of the 1987 Stanford Conference*, pages 75–95. The MIT Press, 1987.

Appendix C

Area Reduction for Bit-Sliced Layouts using a Commercial Development System

Area Reduction for Bit-Sliced Layouts using a Commercial Development System

Abstract

A large ASIC prototype has been developed by means of a commercial development system. The ASIC has a very regular bit-sliced architecture consisting of many instances of a few cell types. The first version consumed 340 mm^2, *too large to fabricate. Through area reduction the second version was reduced to* 210 mm^2 *and fabricated. The techniques were cell modification and routing area minimization. This paper describes the experiences gained from this area reduction process under the constraints of a commercial development system.*

INTRODUCTION

The ASIC prototype is a RSA processor developed to investigate the possibilities of making hardware support for cryptographic applications. The applications are based on the so-called RSA scheme [RSA78], where a fast computation of modular exponentials, $M^E \bmod N$, is crucial. The operands, M, E and N, have more than 500 bits. Modular exponentiation is implemented by modular multiplications, $AB \bmod N$. The architecture for computing a modular multiplication consists of two regular bit-sliced parts: One that adds and subtracts words with a length corresponding to the operands in the modular exponentiation, the other determines quotients for modular reduction. The first part dominates with respect to area consumption.

The project was initiated in 1989, where efficient algorithms suitable for VLSI implementation were developed [OSA91, OK91]. In cooperation with the Department of Research and Development at Jydsk Telefon in Denmark we began an VLSI implementation in September 1990. The requirements for the RSA processor, set up by Jydsk Telefon, was a single chip implementation, capable of handling 576 bit operands and supporting a special self-synchronizing i/o protocol that enables the processor to be embedded into telecommunication equipment.

A commercial chip development system, Chipcrafter from Cascade Design Automation [Cas91], was used in the implementation of the RSA processor. The RSA processor was laid out in a 1.2 μm CMOS process technology. The first version of the RSA processor, with an area of 340 mm^2, was completed in December 1991, but the layout consumed too much area, and it was never fabricated. During the next 13 months the layout was analyzed, and it went through an area reduction process that resulted in a 38% reduction of the area. This second version of the RSA processor, with an area of 210 mm^2, was fabricated. The test of the processor showed that it was functionally correct, and the speed was approximately as predicted by the development system. It is more than twice as fast as other known RSA processors [Oru94].

Since we wanted to retain the extensive support offered by Chipcrafter, the area reduction process was carried out as much as possible within the scope of Chipcrafter, i.e. the manual interventions were kept to a minimum. The main area reduction was obtained by modifying cells in bit-slices and minimizing the area for routing within the bit-slices. Since the largest part of the chip layout is 576 identical bit-slices an effort in reducing the area of a single bit-slice yielded the vast area reduction of 38%.

DEVELOPMENT SYSTEM

The silicon compiler Chipcrafter was the environment for the implementation of the RSA processor. It was used throughout the process of specification, construction and analysis ending with a layout of the complete RSA processor, ready for design rule check and fabrication. In order to preserve the support offered by the tools in Chipcrafter our policy was to *stay* in this environment during the area reduction process and perform the task under the constraints of chipcrafter.

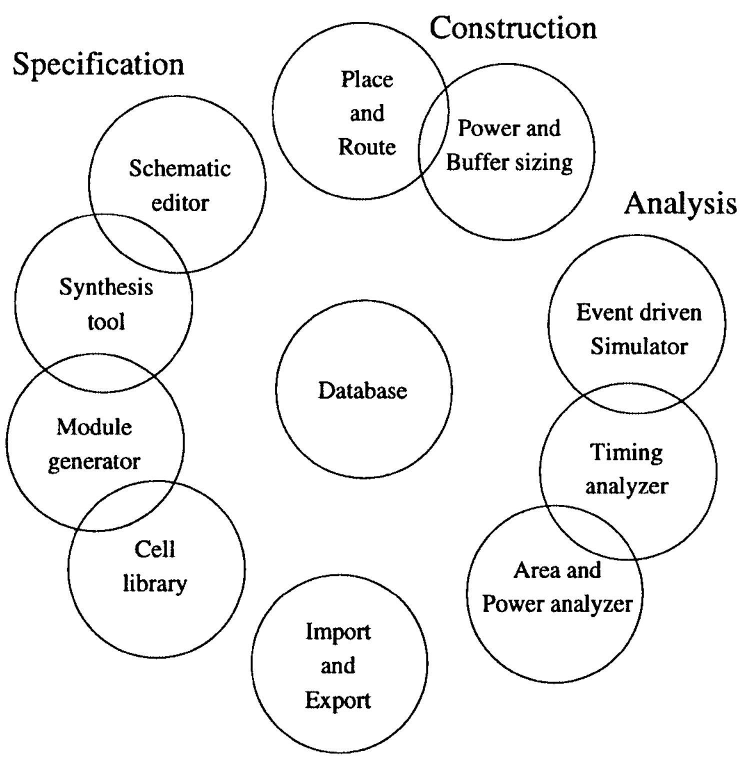


Figure 1: Main tools in Chipcrafter.

Chipcrafter is a silicon compiler in the sense that it is possible to achieve a complete chip layout by means of high-level tools without detailed manual interference. Figure 1 shows the main tools around the Chipcrafter database. A chip is specified in the schematic editor as a set of hierarchal structured schematic drawings, where the building blocks are cells from the cell library or the outcome from invoking the module generator or the synthesis tool.

The cell library consists of a rich set of pad cells, standard cells and datapath cells. Standard cells and datapath cells are functionally equivalent, but differ in the layout style and in the way they are handled by the placement and routing tools. A *datapath* is an array of datapath dells, as illustrated by figure 2, where each row forms a so-called *bit-slice* and each column consists of identical datapath cells. Datapaths are well suited for implementation of circuitry, where each data bit in a bus is treated similarly. Power rails and control signals are routed vertically in a datapath, and data lines are routed horizontally. Chipcrafter lays out datapath cells with a minimal use of second layer metal in order to reserve this layer for data line routing *over* the cells.

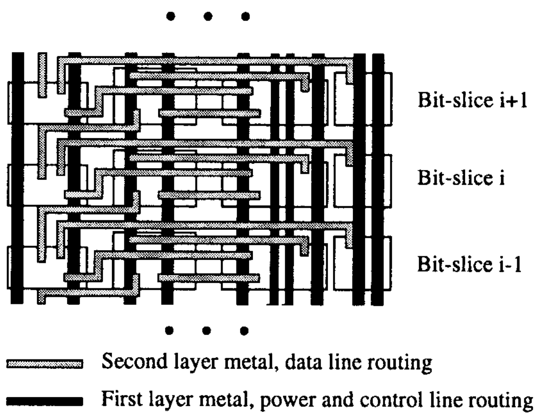

Bit-slice i+1
Bit-slice i
Bit-slice i-1
Second layer metal, data line routing
First layer metal, power and control line routing

Figure 2: A datapath is an array of datapath cells.

Module generators are available for generating a variety of sub circuits. We used these for specifying a 576 bit high-speed adder, realized as a carry select adder with Manchester carry chains, ready to be inserted into our datapaths, and for the generation of counters in the control logic. The control logic and i/o protocols were expressed in terms of finite state machines and implemented by the synthesis tool as standard cell groups. We also applied the synthesis tool for implementing functions specified by tables.

After the schematic specifications have been completed, Chipcrafter is invoked in order to construct a layout of the chip. It separates the design in datapaths, groups of standard cells and a pad frame. Then it performs the placement and routing automatically by using special purpose tools on each part. After that, the core, composed of all parts inside the pad frame, is placed and routed. Finally, the pad cells are attached to the core. It is possible to guide the relative placement of parts in the core through an interactive floorplanning tool and to influence the placement of cells in datapaths and standard cell groups by assigning different weights to routing nets. This implies, that cells will be placed in a way, that tries to reduce the length of nets with high weights.

The cells in the cell library are parameterized with respect to driving capability of the outputs, and the datapath cells are parameterized with respect to the width of power supply rails as well. The tool for power and buffer sizing is used for assigning proper widths to supply lines in order to prevent high voltage drops and metal migration phenomena, and it is also used for sizing the output driving stages of cells selected by the user. We used the tool for sizing the buffers, that drive control and clock signals, and for sizing all power supply lines.

The functionality of the design is analyzed by an event driven simulator. Informations, regarding critical paths, setup and hold time violations, delays etc., are extracted by the timing analyzer. The area and power analyzer reports the area and power consumption for parts of the layout.

In the Chipcrafter environment it is impossible to edit the layout directly. Furthermore, it is not feasible to export a datapath, make changes by means of a layout editor and import the layout back into the environment. The reason is that in order to insert a new layout into the database considerable additional information have to be included: The schematic editor needs a symbolic representation, the place and route tool needs information about cell type, port types and port positions, the power and buffer sizing tool needs information about capacitance and driving capability, the simulator needs a model, the timing analyzer must know delay times, setup and hold times etc., and a tool, that extract transistor netlists of the design, needs a SPICE netlist. Hence, the Chipcrafter export/import supports the inclusion of new layouts, but it is only feasible for a fairly small number of small layouts, such as alternative leaf cells for the library.

RSA PROCESSOR

The RSA processor includes circuitry for calculating modular exponentials, control logic and i/o protocols. The largest parts, with respect to area consumption, are the circuitry implementing 576 bit operand computations and the circuitry implementing quotient determination. These parts are implemented as large datapath structures. Control logic and i/o protocols are implemented as standard cell groups. The 576 bit-slices have been distributed in a number of datapaths because Chipcrafter has an upper limit on the number of bit-slices per datapath, and furthermore a nearly quadratic floorplan is obtained. The quotient determination unit is a single datapath.

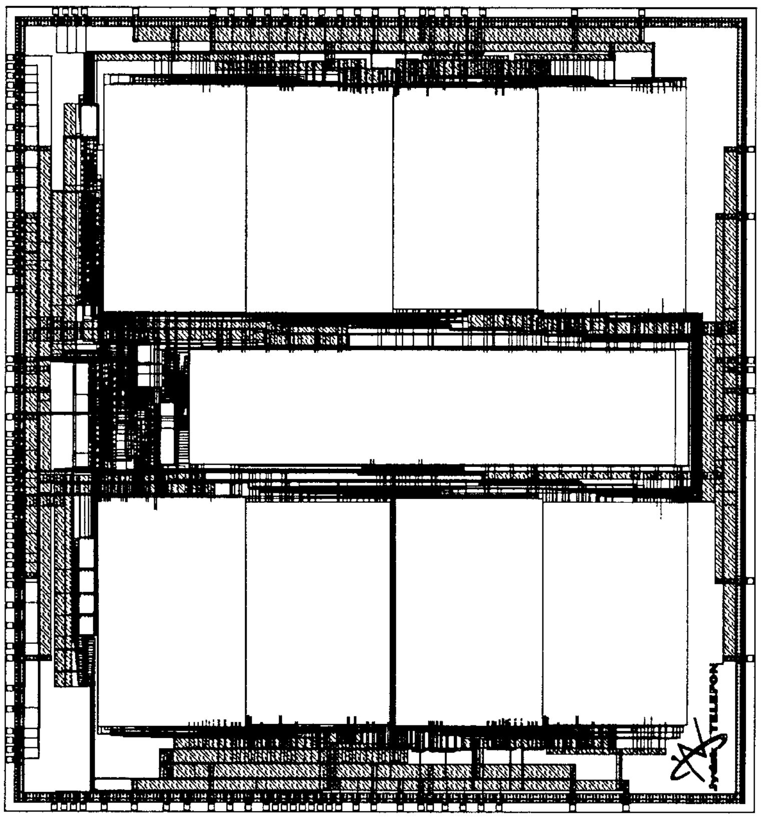


Figure 3: Layout of the RSA processor.

Figure 3 shows a plot of the chip, that was sent to fabrication. The border consists of the pad frame and of very wide power supply lines. The eight rectangles of equal sizes are datapaths including a total of 576 bit-slices, and the middle rectangle is the quotient determination unit. To the left, between the datapaths and the border, control logic and i/o protocol circuitry are located.

The very first layout of the RSA processor, laid out by Chipcrafter without any user interference, comprised 550,000 transistors on an area of more than 500 mm^2. The core parts were badly placed, and the datapaths had a high area consumption. We reduced the area to 340 mm^2 by applying the floor-planning tool to the core and by assigning different weights to routing nets inside the datapaths.

Since a 340 mm^2 chip is infeasible to fabricate, even for a prototype, we considered the possibilities for further reduction of the area within the development system. We wanted to avoid a complete redesign, i.e. a new algorithm and a new architecture. Instead we decided to keep the original algorithm and architecture and focus on the physics layout only. In spite of a large area consumption the core showed a good placement of the parts, so we concentrated on the parts of the core.

The layout is dominated by 576 identical bit-slices, and an effort in reducing the area contribution of a single bit-slice would yield a 576-fold improvement. The area consumption of a bit-slice has two sources: The datapath cell area and the routing area between datapath cells. In the following we will identify the main contributions to the final area reduction from 340 mm^2 to 210 mm^2.

Datapath Routing

Inside a datapath power rails and control signals are routed different from data signals. Power rails and control signals are routed vertically and feed through all the identical cells in a column, as sketched in figure 2.

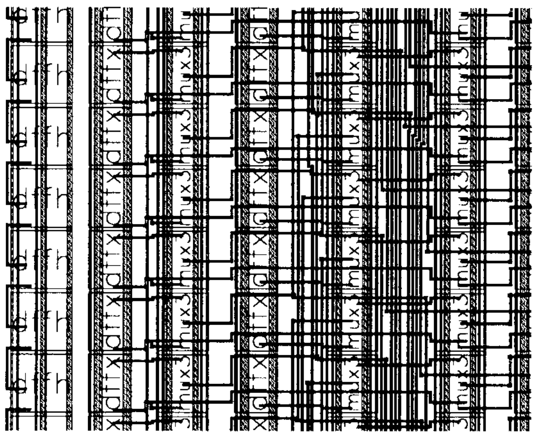


Figure 4: Fraction of a datapath in the fabricated chip.

Data signals are mainly routed horizontally, but when a connection to another bit-slice is required, a vertical routing channel is allocated. The area for data routing is influenced by the relative placement of datapath cells in a bit-slice.

The bit-slices in figure 4, that shows a fraction of a datapath in the fabricated chip, are stacked densely without routing channels between the slices, i.e. the horizontal data routing is exclusively *over* the cells. This was not the case for the 340 mm^2 layout, where the router allocated channels *between* the slices as well. In order to obtain this elimination of routing area between bit-slices, we changed the relative placement of cells in the bit-slice. We tried to eliminate the routing area by assigning different weights to routing nets but did not succeed. Instead we wrote a program that enforced a certain cell placement by changing placement attributes in Chipcrafters internal representation of a datapath.

If routing channels between bit-slices can be avoided, the height per bit-slice in a datapath corresponds to the highest cell in the slice. In the 340 mm^2 layout the height per bit-slice was 72.5 μm, and the highest cell was 58 μm. Hence, a 20% area reduction of datapaths is obtained by avoiding routing channels between bit-slices.

Datapath Cells

The 576 bit-slices are composed of many instances of a small set of different datapath cell types. By changing the layout of a few cells, the effort will

Cell Type	Trans	$x \times y$	Area	Instances	Trans.	Area	Area	Area
	per Cell		per Cell	per Slice	per Slice	per Slice	per Trans.	per Slice
		$\mu m \times \mu m$	μm^2			μm^2	μm^2	%
DFF	28	76 × 58	4408	6	168	26448	157	18.2
DFF inv	30	90 × 58	5220	2	60	10440	174	7.2
DFF cir	34	93 × 52	4836	2	68	9672	142	6.7
MUX 2-1	12	36 × 44	1584	5	60	7920	132	24.7
MUX 3-1	22	83 × 54	4482	8	176	35856	204	5.5
MUX 4-1	30	99 × 50	4950	4	120	19800	165	13.8
Fulladder	28	81 × 50	4050	6	168	24300	145	16.7
CSadder	35-48	245 × 44	10780	1	~42	10780	225-308	7.4
Total				34	862	145216	168	100.0
Bit-slice		2793 × 58	161994					

Table 1: Cells in a bit-slice in the 340 mm^2 layout.

pay back several times. In order to illustrate this, we have included table 1 that lists characteristics of the different cell types in a bit-slice. The first three cells are D flip-flops, where the second cell has both inverting and non-inverting outputs and the third cell has an asynchronous clear option.

The next three cells are multiplexors having two, three and four data inputs. These are followed by a full-adder and a component of a 576 bit high-speed adder. For each cell type the transistor count, dimension and area are listed. The right part of the table lists total contributions per bit-slice originating from the cell type: Number of instances, transistor count, area, mean area per transistor and the area in percentage of the total area per slice. At the bottom line is the bit-slice dimension and area expressed as the sum of all instance widths times the height of the highest cell. This represents the area consumption per bit-slice when all routing area has been stripped off. The line above the bottom line represents totals of the upper part. Table 2

Cell Type	Trans per Cell	$x \times y$ $\mu m \times \mu m$	Area per Cell μm^2	Instances per Slice	Trans. per Slice	Area per Slice μm^2	Area per Trans. μm^2	Area per Slice %
DFF	11	29 × 49	1421	5	55	7105	129	9.4
DFFH	15	48 × 49	2352	1	15	2352	157	3.1
DFFH inv	17	54 × 49	2646	2	34	5292	156	7.0
DFF clr	17	53 × 49	2597	2	34	5194	153	6.8
MUX 3-1	10	40 × 49	1960	6	60	11760	196	15.5
MUXZ 3-1	10	41 × 49	2009	2	20	4018	201	5.3
MUXZ 4-1	13	41 × 50	2050	4	52	8200	158	10.8
Fulladder	20	72 × 49	3528	6	120	21168	176	27.9
CSadder	35-48	245 × 44	10780	1	~42	10780	225-308	14.2
Total				29	432	75869	176	100.0
Bit-slice		1570 × 50	78500					

Table 2: Cells in a bit-slice in the 340 mm^2 layout.

shows the same information for the final 210 mm^2 layout. In the following we describe the main contributions to the reduction of the area of a bit-slice by changing cell layouts:

Aspect Ratio. According to table 1 the cell heights varies from 44 μm to 58 μm. Because the height of a bit-slice is determined by the highest cell in the slice, this variation implies wasted area. Figure 5 illustrates a bit-slice, where all routing area is stripped off. The shaded area represents wasted area due to varying cell heights. For the 340 mm^2 layout the wasted area per slice is $(161,994 - 145,216)\mu\text{m}^2 = 16,778\mu\text{m}^2$, 10% of the bit-slice area. This can be reduced by changing some of the cells aspect ratio. The re-implemented cells in table 2 were laid out with

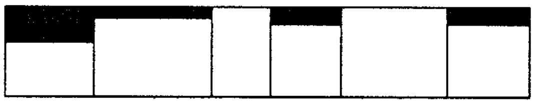


Figure 5: Area is wasted because of varying cell heights.

a smaller height variations and the wasted area per slice is now only $(78500 - 75869)\mu\text{m}^{2} = 2,631\mu\text{m}^{2}$, or 3%.

Transistor Density. Some of the cells have a large area per transistor and an inspection of the layouts shows a bad area utilization within a cell. This is caused by the way cell layouts are represented in Chipcrafter: They have a generic form that takes technology parameters as input, hence, a layout cannot be optimal for all technologies. The area per bit-slice can be reduced by taking the actual technology into account. A comparison of table 1 and table 2 apparently shows that we did not perform well on this. Even though the transistor density has increased in most cell types, the mean density for a bit-slice has decreased slightly. The mean area per transistor in a bit-slice has increased from 168 μm^2 to 176 μm^2, or 5%. The reason for this is, that the two adder cell types with bad transistor densities make up a larger percentage of the total bit-slice area in table 2.

Other Cell Implementations. The cells in Chipcrafters library are designed to be used in all kinds of circuits. This means that a conservative style has been chosen, where reliability has high priority. All combinatorial cells are based on static logic and the flip+flops have static outputs. A significant area reduction can be obtained by using *fewer* transistors in the implementations of flip-flops (32.1% of the bit-slice area is flip-flops) and of multiplexors (43.8%).

In the final layout we used a nine transistor dynamic D flip-flop [YS89] plus an inverter as basis for all flip-flop types. The cell types DFF in both tables are immediate comparable and display a 68% reduction of cell area. Similar results are achieved for the other flip-flop types. All 3–1 and 4–1 multiplexors were replaced by pass transistor implementations. The MUXZ cell types in table 2 differs slightly from a MUX type: One of the data inputs is connected directly to a logical zero in order to reduce the area allocated by the routing tool. The cell area reduction is more than 55% for all types of 3–1 and 4–1 multiplexors.

Merging Cells. All 2–1 multiplexor instances in table 1 are used to implement flip-flops with a hold option. By merging dynamic flip-flops and pass gate multiplexors into a single cell the area for flip-flops with hold option is reduced by about 60% and there is no longer an additional routing area to connect these cells. The types DFFH in table 2 represent flip-flops with hold option.

Total Effect of Bit-slice Area Reduction

Until now we have described the area reduction techniques applied to the 576 bit-slices and we have seen the isolated effects of these. This section will report the total effect on the area of datapath structures containing the bit-slices and the effect on the total chip area.

If we disregard the routing area in the datapaths, the modification of datapath ceils reduced the bit-slice dimension from $2793\mu\text{m}\times 58\mu\text{m}$ to $1570\mu\text{m}\times 50\mu\text{m}$, corresponding to a total reduction from $(576 \cdot 2793 \cdot 58)$ mm^2 = 93 mm^2 to $(576 \cdot 1570 \cdot 50)$ mm^2 = 45 mm^2. The total area of the datapaths were 159 mm^2 in the 340 mm^2 layout and 82 mm^2 in the final layout. This gives a reduction in routing area from $(159-93)$ mm^2 = 66 mm^2 to $(82-45)$ mm^2 = 37 mm^2. Thus, the 576 bit datapaths were reduced by 48%: A 44% reduction in routing area and a 52% reduction in datapath cell area. The main part of the reduction in datapath cell area is due to a reduction in the transistor count by 52%, the reduction in wasted area sets off the decreased transistor density. The same techniques were applied to the quotient determination datapath. The total area was reduced from 32 mm^2 to 20 mm^2, a reduction of 38%.

If we focus on the layout of the chip, the area went from 340 mm^2 down to 210 mm^2, a reduction of 130 mm^2 or of 38%. The transistor count was reduced from 550,000 to 300,000. $(159 - 82) + (32 - 20)$ mm^2 = 89 mm^2 of the area reduction is a direct result of the datapath area reduction. The remaining 41 mm^2 is mainly due to a reduction of the routing area *outside* the datapaths and a consequence of the datapath area reduction: When the datapath area decreases, the length of routing nets along the datapath edges decreases. As illustrated in figure 6 a simplified explanation of this is that the total chip area

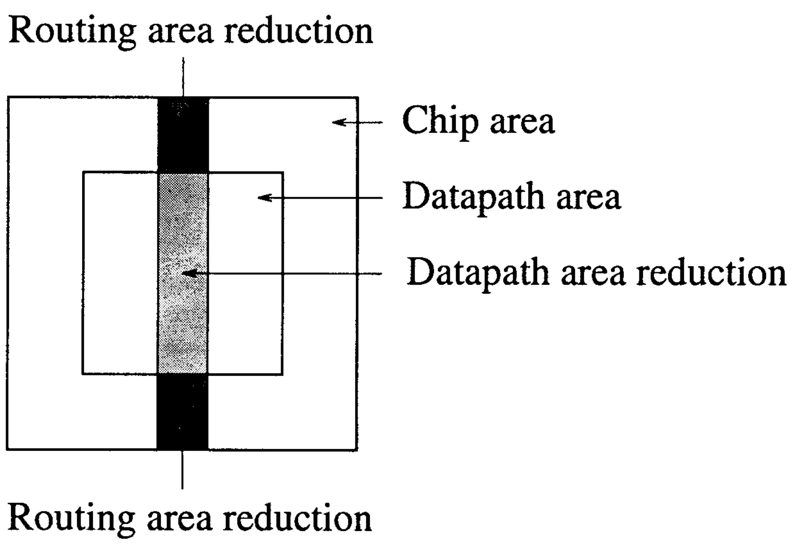


Figure 6: Simplified explanation of reduction outside datapaths.

reduction is proportional to the datapath area reduction, where the reduction factor is the ratio of chip edge length to datapath edge length. Applied to the above mentioned areas, and assuming a quadratic floorplan of the chip area and the datapath area, we get $\sqrt{340}/\sqrt{159+32} = 1.33$, corresponding to 33% further reduction, or 29 mm^2, i.e. the main contribution to the 41 mm^2 reduction outside the datapaths.

CONCLUSION

The project of developing the RSA processor shows that it is possible, even though tedious, to reduce the area of the RSA processor from 340 mm^2 to 210 mm^2 under the constraints of Chipcrafter. This made it possible to fabricate the RSA processor prototype. In the area reduction process the bit-sliced architecture played an essential role. By modifying cells in a single bit-slice and reducing the area for routing within the slice, an area reduction of 38% was obtained. We are convinced, that a further area reduction is infeasible within the scope of the Chipcrafter.

The experience from this area reduction process shows that it is important that a development systems not only offers possibilities but *supports* the user in the process of modifying the placement of cells and the routing of signals. Moreover it must also support the inclusion of user-defined cells into the development system. In later versions of Chipcrafter the possibilities for modifying the placement of cells and routing of signals have been improved.

References

[Cas91] Cascade Design Automation Corporation. *ChipCrafter Designer's Handbook,* 1991.

[OK91] Holger Orup and Peter Komerup. A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N. In Peter Kornerup and David W. Matula, editors, *10th Symposium on Computer Arithmetic,* pages 51–56. IEEE Computer Society Press, 1991.

[Oru94] Holger Orup. A 100 Kbit/s Single Chip Modular Exponentiation Processor. In *HOT Chips VI, Symposium Record,* pages 53–59. Stanford University, 1994.

[OSA91] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an Efficient RSA Hardware Implementation. In I.B. Damgård, editor, *Advances in Cryptology - EUROCRYPT'90*, pages 245–252. Springer-Verlag, 1991.

[RSA78] Ronald L. Riverst, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM,* volume 21, pages 120–126, Feb. 1978.

[YS89] Jiren Yuan and Christer Svensson. High-speed CMOS Circuit Technique. *IEEE Journal of Solid-state Circuits,* 24(1), 1989.

Appendix D

Simplifying Quotient Determination in High-Radix Modular Multiplication

Simplifying Quotient Determination in High-Radix Modular Multiplication[*]

Holger Orup
Computer Science Department
Aarhus University
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, DENMARK
e-mail: orup@daimi.aau.dk

June 1993

Abstract

Until now the use of high radices to implement modular multiplication has been questioned, because it involves complex determination of quotient digits for the module reduction. This paper presents algorithms that are obtained through rewriting of Montgomery's algorithm. The determination of quotients becomes trivial and the cycle time becomes independent of the choice of radix. It is discussed how the critical path in the loop can be reduced to a single ship-and-add operation. This implies that a true speed up is achieved by choosing higher radices.

1 Introduction

Since the introduction of public key crypto systems [1, 12] considerable effort has been directed toward fast hardware implementation of modular multiplication of very large integer operands. A review of techniques for speeding up modular multiplication is included in [4]. It is recognized that *quotient determination*, i.e. determination of the multiple of modulus to subtract at each reduction stage, is the critical operation [4, 13]. This is the reason why, during the last five years, Montgomery's modular multiplication method [7] has been considered the best candidate for faster implementations. Compared to traditional SRT division, the method requires additional pre- and post-processing, but the time for this additional processing becomes negligible when several modular multiplications have to be performed on intermediate results, as is the case when calculating modular exponentials.

In our previous work [11, 10], we have studied the possibilities of speeding up modular multiplication by using higher radices. A

[*]This work has been supported search by the Danish Natural Science Research Council, grant no. 5.21.08.02.

single chip modular exponentiation processor using radix 32 multiplication has been successfully implemented [9]. It is based on a traditional division method and is capable of exponentiating 560 bit operands in less than 5.5 ms, corresponding to a throughput of more than 100 Kbit/s, at a clocking frequency of 25 MHz. According to our knowledge, this is the fastest *single chip* implementation for performing modular exponentials. Only one implementation [13] has been reported to be faster. The high radix approach has been criticized [4, 13] for a large hardware depth (meaning a slow clocking frequency), for a large hardware consumption and for having a non-trivial determination of quotient digits. This is also our experience, but the high radix approach gives potential for substantial speed improvements of modular multiplication. In the rest of this paper we will rewrite the original algorithm of Montgomery and show how this leads to a high-radix algorithm, suited for hardware implementation, where the quotient determination becomes trivial, and the obtainable clocking frequency is independent of the choice of radix.

2 High-radix modular multiplication algorithm

In this section we will rewrite Montgomery's algorithm through a series of development steps. We will show how this leads to an algorithm, where the quotient determination is trivial. Each of the presented algorithms is supplied with an invariant, stating the algebraic relation between the stimulus and the intermediate result, and an upper bound for the range of the intermediate result. The range condition of stimulus has been matched such that the response of the modular multiplication algorithm can be used as stimulus for the same algorithm without additional processing. This has implications on the usefulness of the algorithms for e.g. modular exponentiation. All algorithms are expressed in terms of non-redundant radix 2^k digit sets. This is done in order to limit the description, but the ideas apply as well to other digit sets. See [5] for a high-radix version of Montgomery's algorithm using a symmetric redundant digit set. Additional processing, that may have be performed when Montgomery's method for modular multiplication is used, is discussed in [7, 4, 5]. The first algorithm is a radix 2^k version of the algorithm proposed by Peter L. Montgomery [7] for multiplying two integers modulo M.

Algorithm 1
(Radix 2^k Montgomery Modular Multiplication)

Stimulus:
A modulus $M > 2$ *with* $\gcd(M, 2) = 1$ *and positive integers k, n such that* $4M < 2^k n$.
The integers R^{-1} and M' *are given such that* $(2^{kn}R^{-1}) \bmod M = 1$ *and* $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, where $0 \le A \le 2M$*, and integer multiplier* $B = \sum_{i=0}^{n-1}(2^k)^i b_i$*, where digit* $b_i \in \{0, 1, \ldots, 2^k - 1\}$ *and* $0 \le B \le 2M$.

Response:
An integer S_n *such that* $S_n \equiv ABR^{-1} \pmod M$ *and* $0 \le S_n < 2M$.

Method:
```
$S_0 := 0$;
for $i := 0$ to $n - 1$ do
  $L : q_i := (((S_i + b_iA) \bmod 2^k)M') \bmod 2^k$;
       $S_{i+1} := (S_i + q_iM + b_iA) \text{ div } 2^k$;
end
```

Where:
The following invariant holds at label L:

$$\begin{array}{rcl} 0 & \le & S_i \\ & < & A + M. \end{array}$$

$$2^{ki}S_i \equiv A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} + M \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \quad \bigwedge$$

Correctness: The condition $\gcd(M, 2) = 1$ is sufficient to ensure the existence of R^{-1} and M'. To establish the invariant, note that $q_i \equiv (S_i + b_iA)M'$ (mod 2^k) so $q_iM \equiv -(S_i+b_iA)$ (mod 2^k) and, hence, that 2^k divides $S_i + q_iM + b_iA$ in the updating of S_{i+1}. The invariant holds trivially for $i = 0$. Assuming it holds for $i = \ell$, from the updating of $S_{\ell+1}$, $2^kS_{\ell+1} = S_\ell + q_\ell M + b_\ell A$, we then obtain:

$$\begin{aligned} 2^{k(\ell+1)}S_{\ell+1} &= A \cdot \textstyle\sum_{j=0}^{\ell-1} b_j 2^{kj} \\ &\quad + M \cdot \textstyle\sum_{j=0}^{\ell-1} q_j 2^{kj} \\ &\quad + 2^{k\ell} q_\ell M + 2^{k\ell} b_\ell A \\ &= A \cdot \textstyle\sum_{j=0}^{\ell} b_j 2^{kj} \\ &\quad + M \cdot \textstyle\sum_{j=0}^{\ell} q_j 2^{kj}. \end{aligned}$$

Hence the first part of the invariant holds for $i = \ell+1$. The last part follows from $0 \le q_i \le 2^k - 1$ and $0 \le b_i \le 2^k - 1$:

$$\begin{aligned} 2^{k(\ell+1)}S_{\ell+1} &\le A \cdot (2^k-1)\tfrac{2^{k(\ell+1)}-1}{2^k-1} \\ &\quad + M \cdot (2^k-1)\tfrac{2^{k(\ell+1)}-1}{2^k-1} \\ S_{\ell+1} &< A + M. \end{aligned}$$

By inserting conditions of stimulus we find upon exit from the loop:

$$\begin{aligned} 2^{kn}S_n &= AB + M \cdot \textstyle\sum_{j=0}^{n-1} q_j 2^{kj} \text{ and} \\ 0 &\le 2^{kn}S_n \\ &< 2M \cdot 2M + M \cdot 2^{kn} \\ &< M \cdot 2^{kn} + M \cdot 2^{kn}. \end{aligned}$$

So $R^{-1}2^{kn}S_n \equiv S_n \pmod{M} \equiv ABR^{-1} \pmod{M}$ and $0 \le S_n < 2M$ which proves the correctness of Algorithm 1. □

Montgomery's method for modular multiplication has been implemented for a standard DSP processor [2] and for a board of field programmable gate arrays [13]. Further, in [3, 17, 4, 6] some suggestions for hardware implementations are described. Apart from the DSP implementation, where the radix is determined from the available instruction set, all proposals for a hardware implementation end up with choosing radix 2 or radix 4.

However, for a given operand size, the number of iterations in Algorithm 1 can be reduced by choosing a larger radix. But it is not obvious that this leads to a smaller computation time. The time for an iteration increases for higher radices. This is mainly due to the quotient determination which requires a k bit addition and a $k \times k$ bit multiplication. In the updating of S_{i+1}, the calculation of multiples and addition can be efficiently performed by constant time adders, e.g. carry save adders, [11, 10]. Still the carry-out from the k least significant bits of $S_i + q_iM + b_iA$ must be computed in each iteration. Because the two statements of the loop are strictly sequential, we are not able to reduce the computation time by overlapping the execution of the statements.

2.1 Avoiding multiplication in quotient determination

In the case of radix 2 , i.e. $k = 1$, the multiplication operation in the quotient dete nation in Algorithm 1 is avoided. Because M' mod $2 = 1$, the statement reduces to $q_i := (S_i + b_iA)$ mod 2. For all values of modulus, having the property M' mod $2^k = 1$, the multiplication operation is avoided in the general case of radix 2^k. This observation leads us to transform modulus M to a new value $\widetilde{M}$ that possesses the wanted property. The transformation is simple, $\widetilde{M} = (M' \bmod 2^k)M$, and only has to be performed once. The resulting algorithm is:

Algorithm 2
(Avoiding Multiplication in Quotient Determination)

Stimulus:
A modulus $M > 2$ *with* $\gcd(M,2) = 1$ *and positive integers k, n such that* $4\widetilde{M} < 2^kn$, *where* $\widetilde{M}$ *is given by* $\widetilde{M} = (M' \bmod 2^k)M$.
The integers R^{-1} and M' *are given such that* $(2^{kn}R^{-1}) \bmod M = 1$ *and* $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, where $0 \le A \le 2\widetilde{M}$, *and integer multiplier* $B = \sum_{i=0}^{n-1}(2^k)^ib_i$, *where digit* $b_i \in \{0, 1, \ldots, 2^k - 1\}$ *and* $0 \le B \le 2\widetilde{M}$.

Response:
An integer S_n *such that* $S_n \equiv ABR^{-1} \pmod{M}$ *and* $0 \le S_n < 2\widetilde{M}$.

Method:
```
$S_0 := 0$;
for $i := 0$ to $n-1$ do
  $L : q_i := (((S_i + b_iA) \bmod 2^k$;
       $S_{i+1} := (S_i + q_i\widetilde{M} + b_iA) \text{ div } 2^k$;
end
```

Where:
The following invariant holds at label L:

$$\begin{aligned} 2^{ki}S_i &= A \cdot \textstyle\sum_{j=0}^{i-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \bigwedge \\ 0 &\le S_i \\ &< A + \widetilde{M}. \end{aligned}$$

Correctness: Algorithm 2 is verified by using $\widetilde{M} = (M' \bmod 2^k)M$ and $\widetilde{M} \equiv -1 \pmod{2^k}$. □

Transforming M into $\widetilde{M}$ corresponds to moving the common factor M' (mod 2^k) from the quotient determination to the updating of S_{i+1}. Hereby, a single initial multiplication replaces a multiplication in each iteration. The penalty of using this algorithm is a larger range of the resulting S_n and a value of n that has increased by at most one.

2.2 Avoiding addition in quotient determination

We can further reduce the quotient determination complexity by replacing A by $2^k A$. This technique is also used in [4] and [5]. Since the expression $(S_i + b_i A)$ mod 2^k in Algorithm 2 then reduces to S_i mod 2^k, we have avoided the addition operation. In the update of S_{i+1} we replace $(S_i + q_i\widetilde{M} + b_i A)$ div 2^k by $(S_i + q_i\widetilde{M})$ div $2^k + b_i A$. Comared to Algorithm 2 the number of iterations is increased by one to compensate for the extra factor 2^k:

Algorithm 3
(Avoiding Multiplication in Quotient Determination)

Stimulus:
A modulus $M > 2$ *with* $\gcd(M, 2) = 1$ *and positive integers* k, n *such that* $4\widetilde{M} < 2^k n$, *where* $\widetilde{M}$ *is given by* $\widetilde{M} = (M' \bmod 2^k)M$
The integers R^{-1} *and* M' *are given such that* $(2^{kn}R^{-1}) \bmod M = 1$ *and* $(-MM') \bmod 2^k = 1$.
Integer multiplicand A, *where* $0 \le A \le 2\widetilde{M}$, *and integer multiplier* $B = \sum_{i=0}^{n}(2^k)^i b_i$, *where digit* $b_n = 0, b_i \in \{0, 1, \ldots, 2^k - 1\}$ *and* $0 \le i < n$ *and* $0 \le B \le 2\widetilde{M}$.

Response:
An integer S_{n+1} *where* $S_{n+1} \equiv ABR^{-1}$ (mod M) *and* $0 \le S_{n+1} < 2\widetilde{M}$.

Method:
```
  $S_0 := 0$;
  for $i := 0$ to $n$ do
    $L_1 : q_i := S_i \bmod 2^k$;
    $L_2 : S_{i+1} := (S_i + q_i\widetilde{M}) \text{ div } 2^k + b_i A$;
  end
```

Where:
The following invariant holds at label L_1*:*

$$\begin{array}{rcl} 2^{ki}S_i & = & 2^k A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{i-1} q_j 2^{kj} \quad \bigwedge \\ 0 & \le & S_i \\ & < & 2^k A + \widetilde{M}. \end{array}$$

Correctness: To verified the response, we note that $q_0 = 0$ and $b_n = 0$, hence upon exit from the loop we get:

$$\begin{array}{rcl} 2^{k(n+1)}S_{n+1} & = & 2^k A \cdot \sum_{j=0}^{n} b_j 2^{kj} + \\ & & \widetilde{M} \cdot \sum_{j=0}^{n} q_j 2^{kj} \\ 2^{k(n)}S_{in+1} & = & A \cdot \sum_{j=0}^{n-1} b_j 2^{kj} + \\ & & \widetilde{M} \cdot \sum_{j=0}^{n-1} q_{j+1} 2^{kj} \end{array}$$

So $S_{n+1} \equiv ABR^{-1}$ (mod M) and $0 \le S_{n+1} < 2\widetilde{M}$. □

Noting that $q_i = S_i$ mod 2^k and that $\widetilde{M} + 1$ is divisible by 2^k, we can rewrite the statement at label L_2 in the loop:

$$\begin{array}{l} (S_i + q_i\widetilde{M}) \text{ div } 2^k + b_i A \\ \quad = S_i \text{ div } 2^k + (q_i\widetilde{M} + S_i \bmod 2^k) \\ \qquad \text{div } 2^k + b_i A \\ \quad = S_i \text{ div } 2^k + (q_i(\widetilde{M} + 1)) \text{ div } 2^k + b_i A \\ \quad = S_i \text{ div } 2^k + q_i((\widetilde{M} + 1) \text{ div } 2^k) + b_i A \end{array}$$

The same approach is used for a radix 2 version of Montgomery modular multiplication in [6]. By calculating $(\widetilde{M} + 1)$ div 2^k once for each new value of M, this is a simplification of the stiatement at label L_2. Now we not have to calculate the carry-out from the k least sign cant bits of $S_i + q_i\widetilde{M}$ in the updating statement.

2.3 Utilizing quotient pipelining in modular multiplication

In [13] Montgomery's modular multiplication algorithm has been modified by applying *quotient pipelining*. The idea is to delay the use of quotient digit q_{i-d}, determined from information available in iteration $i - d$ by d iterations. The effect is that d iterations are now available for determining a quotient. In [13] the penalty is d extra iterations and an increased quotient digit range, $q_{i-d} \in \{0, 1, \ldots, 2^{k(d+1)-1}\}$. In Algorithm 4 we have pipelined the quotient determination of Algorithm 3 without increasing the quotient digit range increasing the range of the result:

Algorithm 4
(Modular Multiplication with Quotient Pipelining)

Stimulus:
A modulus $M > 2$ with $\gcd(M,2) = 1$ and positive positive integers k, n such that $4\widetilde{M} < 2^{kn}$, where $\widetilde{M}$ is given by $\widetilde{M} = (M' \bmod 2^{k(d+1)})M$ and integer $d \geq 0$ is adelay parameter.
Integer R^{-1}, where $(2^{kn}R^{-1}) \bmod M = 1$ and integer M', where $(-MM') \bmod 2^{k(d+1)} = 1$, are given.
Integer multiplicand A, where $0 \leq A \leq 2\widetilde{M}$, and integer multiplier $B = \sum_{i=0}^{n+d}(2^k)^i b_i$, where digit $b_i \in \{0,1,\ldots,2^k-1\}$ for $0 \leq i < n, b_i = 0$ for $i \geq n$ and $0 \leq B \leq 2\widetilde{M}$.

Response:
Integer S_{n+d+2} where $S_{n+d+2} \equiv ABR^{-1} \pmod M$ and $0 \leq S_{n+d+2} < 2\widetilde{M}$.

Method:
```
$S_0 := 0; q_{-d} := 0; q_{-d+1} := 0; \ldots; q_{-1} := 0;$
**for** $i := 0$ **to** $n + d$ **do**
  $L_1 : q_i := S_i \bmod 2^k;$
  $L_2 : S_{i+1} := S_i \text{ div } 2^k +$
          $q_{i-d}((\widetilde{M}+1) \text{ div } 2^{k(d+1)}) + b_i A;$
**end**
```

$$S_{n+d+2} := 2^{kd}S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1}2^k$$

Where:
The following invariant holds at label L_1:

$$\begin{array}{rcl} 2^{ki}S_i & + & \sum_{j=0}^{j=i-d} q_j 2^{kj} \\ & = & 2^k A \cdot \sum_{j=0}^{i-1} b_j 2^{kj} \\ & & +\widetilde{M} \cdot \sum_{j=0}^{i-d-1} q_j 2^{kj} \ \bigwedge \\ 0 & \leq & S_i \\ & < & 2^k A + \widetilde{M}. \end{array}$$

Correctness: To establish the invariant, note that $\widetilde{M} \equiv -1 \pmod{2^{k(d+1)}}$ so $2^{k(d+1)}$ divides $\widetilde{M}+1$ and note that $2^k(S_i \text{ div } 2^k) = S_i - q_i$. The invariant holds trivially for $i = 0$. Assuming it holds for $i = \ell$, from the updating of $S_{\ell+1}$ at label L_2, we then obtain:

$$\begin{array}{rcl} 2^{k(\ell+1)}S_{\ell+1} & = & 2^{k(\ell+1)}(S_\ell \text{ div } 2^k) \\ & & +\ 2^{k(\ell+1)}q_{\ell-d}((\widetilde{M}+1) \text{ div } \\ & & 2^{k(d+1)}) +\ 2^{k(\ell+1)}b_\ell A \\ & = & 2^{k\ell}(S_\ell - q_\ell) + 2^{k(\ell-d)}q_{\ell-d} \\ & & (\widetilde{M}+1) + 2^{k(\ell+1)}b_\ell A \\ & = & 2^k A \cdot \sum_{j=0}^{t-1} b_j 2^{kj} + \\ & & \widetilde{M} \cdot \sum_{j=0}^{t-d-1} q_j 2^{kj} \\ & & -\sum_{j=\ell-d}^{\ell-1} q_j 2^{kj} - 2^{k\ell}q_\ell \\ & & +2^{k(\ell-d)}q_{\ell-d} \\ & & +2^{k(\ell-d)}q_{\ell-d}\widetilde{M} \\ & & +2^{k(\ell+1)}b_\ell A \\ & = & 2^k A \cdot \sum_{j=0}^{t} b_j 2^{kj} + \\ & & \widetilde{M} \cdot \sum_{j=0}^{t-d} q_j 2^{kj} \\ & & -\sum_{j=\ell+1-d}^{\ell} q_j 2^{kj} \end{array}$$

Hence the first part of the invariant holds for $i = \ell+1$. The last part is established by noting that,

$$\widetilde{M} + 1 \leq (2^{k(d+1)} - 1)M < 2^{k(d+1)}M.$$

So $(\widetilde{M}+1) \text{ div } 2^{k(d+1)} < M$. The updating of $S_{\ell+1}$ at label L_2 then gives:

$$\begin{array}{rcl} S_{\ell+1} & < & (2^k(A+M))\text{div}2^k \\ & & +(2^k-1)M + (2^k-1)A \\ & \leq & 2^k(A+M) \end{array}$$

Hence the last part of the invariant holds for $i = \ell+1$. To verify the response, we note that $q_0 = 0$ and $b_j = 0$ for $j \geq n$, hence upon exit of the loop we get:

$$\begin{array}{rl} & 2^{k(n+d+1)}S_{n+d+1} + \sum_{j=n+1}^{n+d} q_j 2^{kj} \\ = & 2^k A \cdot \sum_{j=0}^{n+d} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n} q_j 2^{kj} \\ & 2^{kn}(2^{kd}S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1}2^{kj} \\ = & A \cdot \sum_{j=0}^{n-1} b_j 2^{kj} + \widetilde{M} \cdot \sum_{j=0}^{n-1} q_{j+1}2^{kj} \end{array}$$

So after the last statement of the algorithm we obtain $S_{n+d+2} \equiv ABR^{-1} \pmod M$ and $0 \leq S_{n+d+2} < 2\widetilde{M}$. $\square$

The last statement in Algorithm 4 is just a left shift of S_{n+d+1} where the d last quotient digits are shifted in from the right. Algorithm 4 is clearly an improvement of the quotient pipelined version in [13]: There is *no* calculation involved in the quotient determination. The quotient digit range has *not* increased by a factor of 2^d, otherwise implying an increased complexity in the calculation of multiples $q_{i-d}((\widetilde{M}+1) \text{ div } 2^{k(d+1)})$, where $(\widetilde{M}+1)$ div $2^{k(d+1)}$ is a pre-calculated integer. However, compared to the quotient pipelined algorithm in [13], Algorithm 4 has an increased range of the result. In [13] the pipelining technique was applied in order to perform overlapping calculations of quotient digits, hereby reducing the hardware depth for a radix 4 modular multiplication algorithm. In Algorithm 4 it could seem meaningless to apply pipelining because of the trivial quotient determination. The calculation of multiples becomes more time consuming for higher radices, but we use the pipeline technique for performing overlapping calculations of the multiples $q_{i-d}((\widetilde{M}+1) \text{ div } 2^{k(d+1)})$ and b_iA. All of these operands are available after iteration $i-d$, and the resulting multiples are not needed before iteration i. If convenient, we could even perform an addition of the multiples before iteration i. Then we have reduced the time for an iteration to the time for a shift and an addition of two words, $S_{i+1} := S_i \text{ div } 2^k + T_{i-d}$. Because of the increased range of the result and the increased number of iterations, we should choose d to be as small as possible. The calculation of the multiples can be performed in about $\log_2 k$ steps by two pipelined adder-trees if a redundant representation of the resulting multiples is allowed [16].

3 Example hardware architecture

To get an impression of the speed potential of Algorithm 4 we will discuss a hardware architecture for executing the algorithm. Figure 1 shows an example architecture where $k = 8$ and $d = 3$, i.e. the radix is 2^8 and the architecture has 3 pipeline stages. The architecture includes registers for the operands A, B and $(\widetilde{M}+1) \text{ div } 2^{8(3+1)}$.

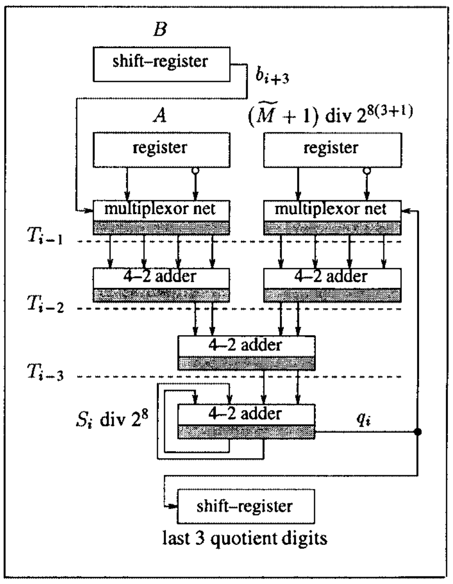


Figure 1: Hardware architecture using radix 2^8 and pipelined into 3 stages.

A small register for holding the last 3 quotient digit (q_{i-1}, q_{i-2} and q_{i-3}) is also depicted. This register is used in the last statement of Algorithm 4. All intermediate results are redundant represented and an 4–2 adder is used for adding two redundant represented integers. As described in [10], a multiple of A or $(\widetilde{M}+1) \text{ div } 2^{8(3+1)}$ can be calculated by feeding the binary representation through a multiplexor network. Then the multiple is represented as the sum of a number of integers. In this case, where the radix is 2^8, the number of integers that represents a multiple is four. These four integers can then be compressed into two integers by using an 4–2 adder. Hence, a multiple is redundant represented as two integers. In the figure, the upper pairs of multiplexor networks and 4–2 adders are used for calculating the multiples of A and of $(\widetilde{M}+1) \text{ div } 2^{8(3+1)}$. After this calculation, the multiples are added by a third 4–2 adder giving $T_{i-3} = b_iA + q_{i-3}(\widetilde{M}+1) \text{ div } 2^{8(3+1)}$ Finally, T_{i-3} is added to $S_i \text{ div } 2^8$ by the 4–2 adder shown in the lowest part of the figure. All of the multiplexor networks and 4–2 adders are latched. This implements the pipeline.

The cycle time of Algorithm 4 (the time for computing a single iteration of the loop) is determined by the critical path, i.e. the circuitry with the longest delay between two latches. Since the delay of the multiplexor network is approximately the delay of a single 4–1 multiplexor plus the set-up delay of a latch, the cycle time of Algorithm 4 is seen to be equal to the (longer) delay of an 4–2 adder. In modern CMOS technologies the delay of the 4–2 adders (including the latch set-up delay) is, by a conservative estimate, less than 5 ns. As an example, we will perform modular multiplication with 512 bit operands. If n is set to 69 the stimulus conditions of Algorithm 4 are fulfilled. Then the number of cycles in the loop is $69+3+1 = 73$. Since the first multiple of A, b_0A, is delayed by 3 stages in the pipeline, the first cycle of the loop can begin after 3 clock periods. So, from the input are available to the result is present 76 clock periods are elapsed. With a 5 ns clock period this is 380 ns. Note that the result is redundant represented and that A and B, in this example, are assumed to be non-redundant represented. This means that the result has to be converted to non-redundant representation before it can be used as input for a new multiplication. The conversion must be performed by a carry completion adder. According to Algorithm 4 the result can be up to $8 \cdot 69 - 1 = 551$ bit wide. The fastest carry completion adders for these very large operands have a delay proportional to $\log_2 551$ but they are quit large in comparison with a carry ripple adder. In [13] a carry ripple adder with an asynchronous carry completion detection circuit is proposed. It is utilized that the average carry propagation length is the logarithm of the operand bit length, i.e. $\log_2 551 < 10$ for our example. If we estimate the average time for a conversion from redundant to non-redundant representation to be 10 clock periods a total of 83 clock periods, or 415 ns, is used for the multiplication.

In the above estimate for the computing time it turns out that about 15% of the time is used for conversion into non-redundant representation. As described in [10] and in [14] it is also possible to perform multiplications where the inputs are in redundant representation. Regarding the multiplier B, this does not imply serious trouble: The multiplier is scanned digit by digit from the least significant end, so a conversion into non-redundant representation may be done on-the-fly. The required circuitry for a register capable of holding a redundant represented operand will be about double the circuitry for holding a non-redundant represented operand. Regarding the multiplicand A, the penalty for using a redundant representation is larger: The required circuitry for computation of the multiples b_iA will expand and the depth of the adder-tree will increase. This means that the delay parameter d must be increased. So, it is possible to obtain a further improvement of the computation time at the cost of additional circuitry.

In the computation of modular exponentials, also with 512 bit operands, the average number of required multiplications is 768 for a sequential algorithm. This can be done in $768 \cdot 415ns \approx 319\mu s$, which corresponds to a throughput of more than 1.6 Mbit/s. If a parallel algorithm is applied, see [10], the computing time is equal to the time for performing 512 multiplications, 212 μ s, and a throughput of more than 2.4 Mbit/s is achieved. This is four times faster than the fastest known implementation [13]. Furthermore, if the modulus is a composite, and the prime factorization is known, it possible to speed up the computation by using the Chinese Remainder Theorem. This technique is applied in [13] to improve the time by about a factor of four.

The hardware consumption for the example architecture in Figure 1 is two multiplexor networks, four latched 4–2 adders and registers for the input operands. Each multiplexor network consists of four rows of latched 4–1 multiplexors and each 4–2 adder consists of two rows of fulladders where one of these is latched. Compared to the exponentiation processor in [9] this is an increase in circuitry of two rows of latched fulladders, two rows of latched 4–1 multiplexors plus the cost for modifying six rows of 4–1 multiplexors into latched 4–1 multiplexors. In [9] a large quotient determination unit and a carry completion adder are also included. There is no longer a need for the quotient determination unit. The circuitry cost of the primitive hardware components used in the exponentiation processor is analyzed in [8]. The transistor count for the complete exponentiation processor is 304,000. We estimate that the architecture in Figure 4, capable of multiplying 512 bit operands, can be implemented by not more than 300,000 transistors.

The above example architecture illustrates the potential of Algorithm 4. We could achieve even higher speeds by choosing higher radices and adding more circuitry to the architecture. A radix 2^{16} version could be implemented by adding a level in the trees for calculating multiples. This would increase the number of pipeline stages by one, increase the number of latched 4–2 adders by four and double the circuitry for the multiplexor networks. The number of clock periods for a 512 bit operand multiplication would then be $4 + 38 + 5 = 47$ for producing a redundant represented result and 57 for producing a non-redundant represented result This is about 31% reduction of the total computing time for the radix 2^8 version. A 512 bit exponentiation would have a throughput of 2.3 Mbit/s if the sequential exponentiation algorithm is applied or 3.5 Mbit/s if the parallel algorithm is applied.

4 Summary

In this paper we have rewritten a high-radix version of Montgomery's modular multiplication algorithm in order to obtain a trivial quotient determination, where the multiplication and addition operation is avoided by a simple transformation of modulus. The result is a quotient determination that is reduced to a trivial extraction of the least significant digit of the partial modular product, $S_i \bmod 2^k$. By applying a pipeline technique we have enabled overlapping computations. This implies that the critical computation path can be reduced to a shift-and-add operation that is efficiently implemented by a constant time adder. We have achieved a modular multiplication algorithm, where *the critical hadware path is independent of the choice of radix.* For a fixed high radix, the cost of the proposed algorithms, over Montgomery's algorithm, is a few extra iteration cycles, additional pre-processing for the transformation of modulus and a wider range of the final result. When several modular multiplications have to be performed on intermediate results, this cost is more than compensated by the faster time for an iteration and the possibility to reduce the number of iterations through the choice of radix. By pipelining the formations of the multiples, the only limitation to the choice of radix is the size of the circuitry, not the cycle time.

References

[1] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory,* IT-22(6):644–654, November 1976.

[2] Stephen R. Dussé and Burton S. Kaliski Jr. A Cryptograhpic Library for the Motorola DSP56000. In Ivan B. Damgård, editor, *Advances in Cryptology - EUROCRYPT '90. Proceedings,* volume 473 of *Lecture Notes in Computer Science,* pages 230–244. Springer-Verlag, Berlin, 1991.

[3] Stephen E. Eldridge. A Faster Modular Multiplication Algorithm. *International Journal of Computer Mathematics,* 40:63–68,1991.

[4] Stephen E. Eldridge and Colin D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers,* C-42(6):693–699, June 1993.

[5] Peter Komerup. High-Radix Modular Multiplication for Cryptosystems. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic,* pages 277–283. IEEE Computer Society Press, Los Alamitos, California, 1993.

[6] Peter Komerup. A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms. *IEEE Transactions on Computers,* C-43(8):892–898, August 1994.

[7] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation,* 44(170):519–521, April 1985.

[8] Holger Orup. Area Reduction for Bit–Sliced Layouts using a Commercial Development System. This article is not published. It is available from the author.

[9] Holger Orup. A 100Kbit/s Single Chip Modular Exponentiation Processor. In *HOT Chips VI, Symposium Record,* pages 53–59. Stanford University, 1994. Only the slides from the presentation at HOT Chips VI are printed in the Symposium Record. An abstract is available from the author.

[10] Holger Orup and Peter Kornerup. A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic,* pages 51–56. IEEE Computer Society Press, Los Alamitos, California, 1991.

[11] Holger Orup, Erik Svendsen, and Erik Andreasen. VICTOR an Efficient RSA Hardware Implementation. In Ivan B. Damgård, editor, *Advances in Cryptology - EUROCRYPT '90. Proceedings,* volume 473 of *Lecture Notes in Computer Science,* pages 245–252. Springer-Verlag, Berlin, 1991.

[12] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM,* 21(2):120–126, February 1978.

[13] M. Shand and J. Vuillemin. Fast Implementations of RSA Cryptography. In Earl Swartzlander, Jr., Mary Jane Irwin, and Graham Jullien, editors, *Proceedings. 11th IEEE Symposium on Computer Arithmetic,* pages 252–259. IEEE Computer Society Press, Los Alamitos, California, 1993.

[14] Naofumi Takagi. A Radix-4 Modular Multiplication Hardware Algorithm Eficient for Iterative Modular Multiplications. In Peter Kornerup and David W. Matula, editors, *Proceedings. 10th IEEE Symposium on Computer Arithmetic,* pages 35–42. IEEE Computer Society Press, Los Alamitos, California, 1991. This article also appears as [15].

[15] Naofumi Takagi. A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation. *IEEE Trans. actions on Computers,* C-41(8):949–956, August 1992.

[16] C. S. Wallace. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers,* EC-13(1):14–17, February 1964.

[17] Colin D. Walter. Systolic Modular Multiplication. *IEEE Transactions on Computers,* C-42(3):376–378, March 1993.

Appendix E

RSA Processor, Preliminary Engineering Data

RSA Processor, Preliminary Engineering Data

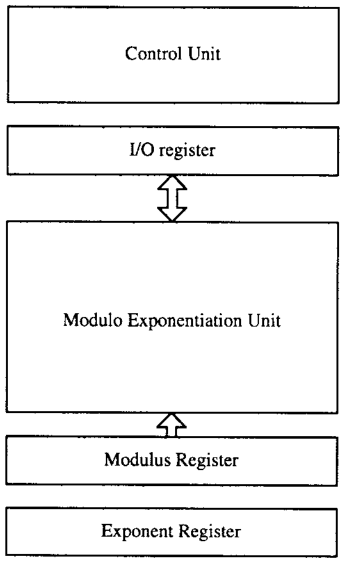
Holger Orup
e-mail: orup@daimi.aau.dk

Features

Calculates $M^E \bmod N$
561 bit operand size
Self–synchronizing SLD interface
TTL input compatible
TTL and CMOS output compatible

Applications

Crypto systems
Primality testing



1 Introduction

The RSA processor is a special purpose processor for calculating modular exponentials. It supports two interfaces. A general purpose interface with a hand shake protocol, and a self-synchronizing SLD interface that enables the processor to be embedded into telecommunication equipment. Figure 1 shows the basic blocks of the RSA processor.

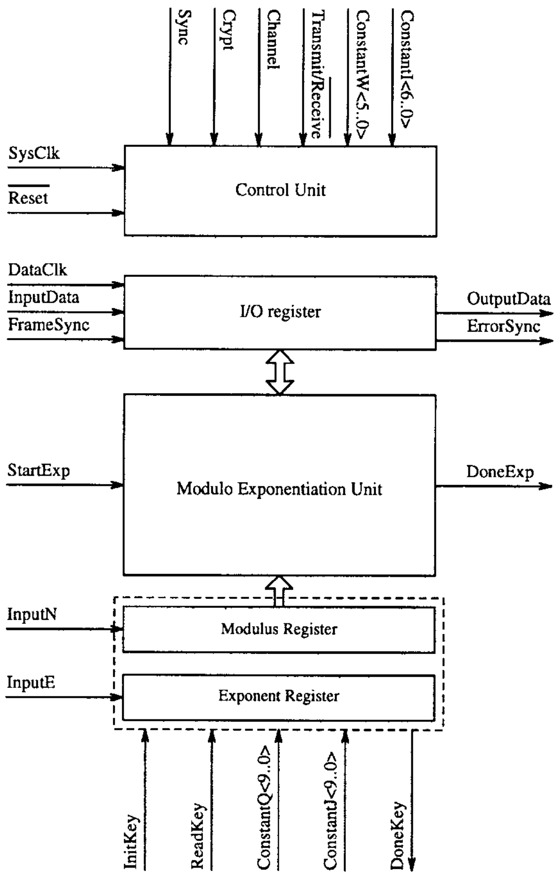


Figure 1: Basic blocks in RSA processor.

The I/O register is a shift-register. Simultaneously with reading in a new block of data, the result of the previous modular exponentiation is written to the output. The Modular Exponentiation Unit can compute a modular exponential in parallel with the I/O register is collecting data for the next computation. Therefore, the user may take full advantage of the computing power of the RSA processor.

The processor speed is determined by the system clock. At 20 MHz the throughput for computations using 561 bit operands is more than 64 Kbit/sec. The RSA processor is designed to support the RSA public key crypto system [RSA78] with adequate computing power.

For the purpose of testing and diagnosing of errors, virtually all the internal registers are linked together in scan chains [WE85]. The scan chains are not shown in Figure 1.

2 Pin designations

The tables showen below list the pins of the RSA processor. If a pin name is overlined it is active at the low voltage level. Otherwise it is active at the high level. A bus notation is used to group a number of pins. E.g. *ConstantW*<5..0> denotes a 6 bit bus with the name *ConstantW*. The least significant bit is denoted *ConstantW*<0> and most significant bit is denoted *ConstantW*<5>. The pins marked with '(SLD)' in their functional description are not used in the general purpose interface. Input pins should never be left unconnected. Unused pins must be tied to a high or a low voltage level.

Pin	**In/Out**	**Function**
Vdd, GND		Power supply and return
SysClk	in	System clock
$\overline{\text{Reset}}$	in	System reset
Sync	in	Use self-synchronizing SLD interface
Crypt	in	(SLD) Perform crypt operations on I/O register
Channel	in	(SLD) Use channel channel B1
Transmit/$\overline{\text{Receive}}$	in	(SLD) Add block synchronization pattern
ConstantW<5..0>	in	Number of inserted wait states
ConstantI<6..0>	in	Number of radix 32 digits in multipliers

Table 1: Pins for system control and mode configuration.

Table 1 lists the pins used for configuring the RSA processor, for the power supply, and for controlling the basic system. *SysClk* is the input from an external generated system clock. The RSA processor does not contain an internal clock generating circuit. The speed of the Modulo Exponentiation Unit is set by the system clock frequency. None of the pins used in normal operation mode need to be synchronized to the *SysClk*.

By pulling the pin $\overline{\textit{Reset}}$ to low, the processor is reset. This must be done after power up and are sometimes required to effectuate changes in the configuration settings. A reset initializes some of the internal registers. The contents of the I/O, the Modulus, and the Exponent registers are unchanged.

Pin *Sync* configures the processor to the self-synchronizing SLD interface or to the general purpose interface. *Crypt, Channel* and *Transmit/*$\overline{\textit{Receive}}$ are configuration pins for the SLD interface. *Crypt* selects between the crypt and the transparent mode, *Channel* selects between the two B channels of the ISDN connection, and $Transmit/\overline{Receive}$ selects the direction of communication. When the RSA processor is in transmit *and* crypt mode it adds a synchronization pattern to the output. If it is configured to the receive *and* crypt mode, it checks the synchronization pattern of the input.

The pins *ConstantW*<5..0> select the number of wait states that is inserted by the processor when an internal carry completion addition is performed on the very long words. A wait state has a duration of one system clock period. By adjusting the value of the pins *ConstantW*<5..0>, the processor can be tuned for a wide range of system clock frequencies. The pins *ConstantI*<6..0> denote the number of radix 32 digits, i.e. 5 bit groups, in the internal multiplier registers. Currently this is 115 and should not be changed by the user.

Pin	**In/Out**	**Function**
ErrorSync	out	(SLD) A block synchronization error is detected
FrameSync	in	(SLD) Frame synchronization clock
DataClk	in	Data synchronization clock
InputData	in	Serial data input
OutputData	tri-state out	Serial data output
StartExp	in	Start the exponentiation process
DoneExp	out	The exponentiation process is terminated

Table 2: Pins used for data I/O.

Table 2 lists the pins used for reading and writing data in the I/O register and for controlling the moments of computation. Pin *DataClk* synchronizes the reading and writing of each bit on the serial input pin *InputData* and the serial output pin *OutputData*. The processor flags when it has completed a modular exponentiation on pin *DoneExp*.

In the general purpose interface the pin *StartExp* is used to initiate a modular exponentiation. Before starting this operation the I/O register must be initialized with an operand. We denote the contents of the I/O register a *block*. The usual bit length of a block is 561 bit.

In the SLD interface a *frame* is a group of 64 bits: 32 bits are communicated in each direction of a bidirectional serial data line. Just 8 of the bits in a frame are collected by the RSA processor. The *Outputdata* pin is a tri-state output pin. This makes it possible to connect directly to the SLD data line. Pin *FrameSync* is used to signal the start of a frame on the serial data line. Pin *ErrorSync* only has a meaning when the processor uses the SLD interface and is configured to the receive and crypt mode. Then the pin flags that an error in the synchronization pattern of a block has been detected.

The pins used for the initialization of the Modulus and the Exponent registers are listed in Table 3. Pin *InitKey* signals that a process of writing new values to the Modulus and the Exponent registers is in progress. The serial input pins for these registers are pin *InputN* and pin *InputE*. Pin *ReadKey* is used

Pin	**In/Out**	**Function**
InitKey	in	Start key initialization process
ReadKey	in	Key synchronization clock
InputN	in	Data input, Modulus
InputE	in	Data input, Exponent
DoneKey	out	Key initialization process terminated
ConstantQ<9..0>	in	Constant value for internal counter
ConstantJ<9..0>	in	Number of bits in exponent

Table 3: Pins used for key initialization.

for synchronizing the moment of writing to the Modulus and the Exponent registers. The registers must be simultaneously initialized, i.e. the processor reads values from both serial inputs in the same period of *ReadKey*. When the initialization of the Modulus and the Exponent registers are completed, and the processor is ready for a new series of modular exponentiations, the flag *DoneKey* is activated by the processor.

The pins *ConstantQ*<9..0> are used to inform the processor of the number of bits that are input to the Modulus register and the Exponent register. At present, this number is equal to the bit length of the registers plus eight additional bits, a total of 569 bit. This value should not be changed by the user. The pins *ConstantJ*<9..0> determine the number of bits in the Exponent register that should be used in a modular exponentiation. If *ConstantJ*<9..0> is less than 561 the processor will discard bits from the most significant end of the Exponent register. See Section 3.2 for further details on the assignment of values to *ConstantJ*<9..0>.

The pins in Table 4 are only used for test purposes. So, they are not used in normal operation modes. However, all of the input pins must be connected to certain voltage levels. This is described in Section 3. None of the pins in Table 4 are shown in Figure 1. The pins *TestSysClk*, *TestDataClk* and *TestFrameSync* are control pins that choose between the normal operation modes and the test operation modes.

The *TestSysClk* pin is controlling the three scan chains that must be synchronized to the *SysClk*: The chain with the serial input *ScanInSysClk* and the serial output *ScanOutSysClk*, the chain with the 5 bit input *LSBreg<4..0>* and the 5 bit output *ScanOutRegX<4..0>* and, finally, the chain with the 5 bit input *LSBregM<4..0>* and the 5 bit output *ScanOutRegM<4..0>*. All input to, and output from, the pins controlled by *TestSysClk* must be synchronized to the *SysClk*.

The *TestDataClk* is controlling the scan chain that must be synchronized to the *DataClk* The chain with the serial input *ScanInDataClk* and the serial output *ScanOutDataClk* All communication with these pins must be synchronized to the *DataClk*.

Finally, *TestFrameSync* is controlling the scan chain that must be synchronized to the *FrameSync*: The input is pin *ScanInFrameSync* and the output is *ScanOutFrameSync*. All communications with these pins must synchronized to *FrameSync*.

Pin	In/Out	Function	Note
TestSysClk	in	Activate system clock scan chain	
ScanInSysClk	in	Input for system clock scan chain	
ScanOutSysClk	out	Output from system clock scan chain	1
LSBregX<4..0>	in	Input for shift register X	
ScanOutRegX<4..0>	out	Output from shift register X	
LSBregM<4..0>	in	Input for shift register M	
ScanOutRegM<4..0>	out	Output from shift register M	
TestSysClk	in	Activate data clock scan chain	
ScanInSysClk	in	Input for data clock scan chain	
ScanOutSysClk	out	Output from data clock scan chain	
TestFrameSynk	in	Activate frame clock scan chain	
ScanInFrameSynk	in	Input for frame clock scan chain	
ScanOutFrameSynk	out	Output from frame clock scan chain	

Table 4: Pins for scan chain inspection in test operations.

Notes

1. This pin is physical identical to pin *ErrorSync* in Table 2. In normal operation mode the pin is *ErrorTSync*, and in test operation mode, the pin is *ScanOutSysClk.*

3 Processor

In normal operation mode the architecture of the RSA processor, from the users point of view, is as illustrated in Figure 1. The Modulus and the Exponent register holds the operands for the modular exponentiation. The registers must be properly initialized before the start of the exponentiation. The I/O register partly holds the new data to be exponentiated and partly holds the result of the previous computation. The computation of the modular exponentials is performed by the Modular Exponentiation Unit and it may be done in parallel with the users access to the I/O register.

To obtain a correct functionality of the RSA processor, it is required that the pins in Table 5 are properly connected. The values in the table ensures a correct functionality for 561 bit modular exponentiation.

Pin	In/Out	Function	Note
TestSysClk	0	0	
TestDataClk	0	0	
TestFrameSync	0	0	
LSBregX<4..0>	1	00001	
LSBregM<4..0>	0	00000	
ConstantW<5..0>	6	000110	1
ConstantI<6..0>	115	1110011	
ConstantJ<9..0>	561	1000110001	2
ConstantQ<9..0>	569	1000111001	

Table 5: Pin values in normal operation mode.

Notes

1. The necessary number of wait states depends on the System Clock frequency. In this table a frequency of 20 MHz is assumed. At higher frequencies a higher value may be necessary.
2. As explained in section 3.2 this value may be decreased in order to decrease the computing time for smaller exponent values.

3.1 I/O register and Modulo Exponentiation Unit

The interface of the RSA processor is entirely serial. Data is input, least significant bit first, to pin *InputData* and shifted into the I/O shift register. After each shift a new bit of the previous computation is ready to be read from pin *OutputData*. When the I/O register is filled with a new 561 bit data block, and the previous result is shifted completely out, the processor performs a swap of the I/O register contents and the contents of the result register in the Module Exponentiation Unit. Then, the next computation is initiated. If the I/O register contents before the swap is denoted M, the resulting output from the Modulo Exponentiation Unit can be expressed as $M^E \bmod N$, where E is the contents of the Exponent register and N is the contents of the Modulus register. In the SLD interface this swap is performed automaticly when the I/O register has been filled. In the general purpose interface the user have to control the moment of swapping. This is signalled by making a positive transition on pin *StartExp*. The processor responds by pulling pin *DoneExp* to high as soon as the computation is terminated. In the general purpose interface the user also have to control the I/O register by means of pin *DataClk*: A new result bit is written to *OutputData* at a positive transition, and a new data bit is read from *InputData* at the next negative transition.

3.2 Modulus and Exponent registers

In both interfaces the initialization of the Modulus and the Exponent registers is controlled completely by the user. Pin *InputN* is the serial input to the Modulus register, and pin *InputE* is the serial input to the Exponent register. Both registers expect the least significant bit first. The initialization process is controlled by pin *InitKey* and *ReadKey*. By pulling *InitKey* to high the user signals that a new initialization process is started. When a new set of 561 bit values has been shifted into the registers the user pulls *InitKey* back to low, and the RSA processor proceeds with some preprocessing in order to initialize the internal circuitry. The processor responds by pulling pin *DoneKey* to high upon completion, and is ready for new computations. Before an initialization is started, the user must make sure that previous initializations and computations are terminated, i.e. *DoneKey* and *DoneExp* are high. The moment of reading pin *InputN* and *InputE* are at a negative transition of pin *RedKey*, and the processor reads values from *both* pins. This means that the registers are initialized in parallel. Even though the bit length of the modulus and the exponent is 561,

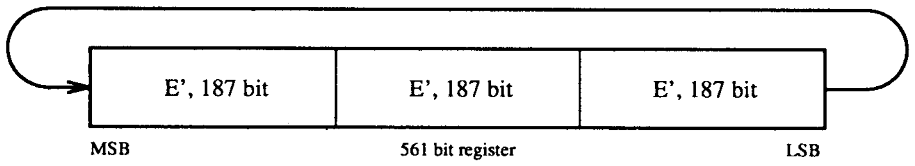


Figure 2: The Exponent register initialized to 187 bit exponents.

it is required that 569 bits are shifted into the registers: 8 zero bits must be padded to the values. The format of the input to the registers are

$$\begin{aligned} InputE &= e_{560}e_{559}\cdots e_0 00000000 \\ InputN &= 00000000 n_{560}n_{559}\cdots n_0 \end{aligned}$$

where e_0 is the least significant bit of the exponent, and n_0 is the least significant bit of the modulus.

The Exponent register is a cyclic shift register. The pins *ConstantJ*<9..0> express the number of cyclic shift operations performed during the calculation of modular exponentials. It expresses the bit length of the exponent. If it less than 561, some of the most significant bits will not be used in the calculation. In some cases this can be used to decrease the computing time for an exponential, which is roughly proportional to the value of *ConstantJ*. E.g. if an exponent value is less than 187 bit ($3 * 187 = 561$) the register can be loaded with a 561 bit value that is constructed by repeating the 187 bit exponent value three times. When *ConstantJ* is set to 187 only the 187 least significant bits of the Exponent register is used in a modulo exponentiation, and only 187 cyclic shifts of the Exponent register are done. However, because of the repeated 187 bit pattern the register contents after a modular exponentiation will be identical to the contents prior to the exponentiation. This is shown in Figure 2. This will reduce the computing time to one third of the computing time for *ConstantJ* set to 561.

4 Interfaces

The RSA processor supports a general purpose interface and a self-synchronizing SLD interface [Sie92c]. In both interfaces the processor is reading and writing data in a bit serial format, least significant bit first. Initialization of the Modulus and the Exponent registers, and reset of the processor is performed the same way in both interfaces. The interfaces differs in how to access the I/O register and how to start a modular exponentiation. Pin *Sync* determines which interface to choose, a high value selects the self-synchronizing interface and a low value selects the general purpose interface. The value of pin *Sync* must be well defined when the RSA processor is reset, and a change of value will have an effect after a new reset operation.

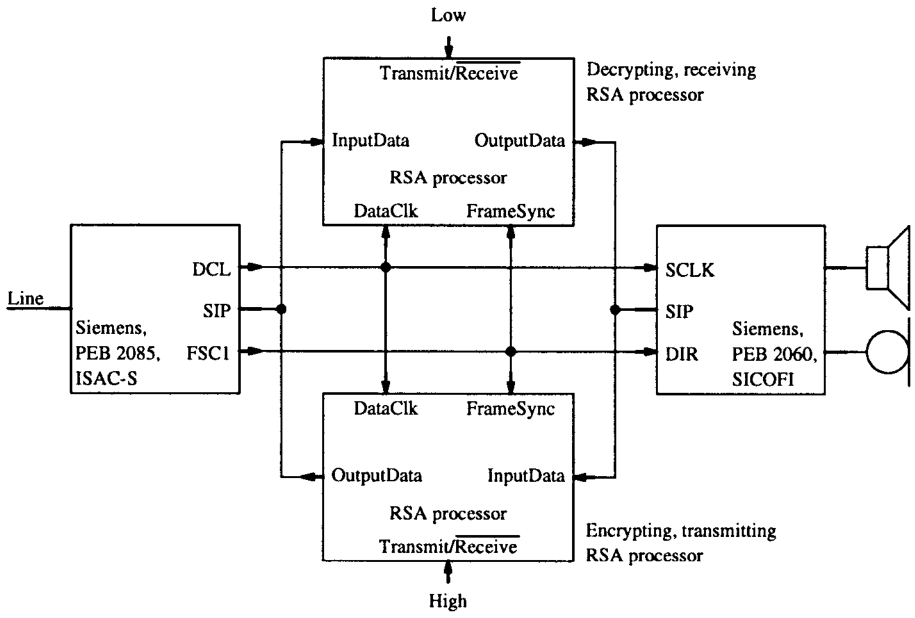


Figure 3: RSA processors placed at SLD bus interface.

4.1 General purpose interface

Most of the control is handed to the user in this interface. When a new 561 bit data block has been shifted into the I/O register and a 561 bit result shifted out, the user must control that the Modular Exponentiation Unit has terminated its previous computation, i.e. pin *DoneExp* is high. Then a positive transition on pin *StartExp* can be done in order to perform a swap of the I/O register and the Modular Exponentiation Unit and to start a computation with the new data.

4.2 SLD interface

The SLD interface is included in the processor in order to ease the task of embedding the processor in a telecommunication application. In the application is an ISDN telephone modified to encrypt/decrypt digital voice, where the RSA protocol is used. The ISDN telephone is of type Ascotel Crystal from Ascom, Schwitzerland, and uses components from Siemens. A SLD bus is used to exchange data between a Codec [Sie92b] and a Line Subscriber Circuit [Sie92a]. The bus has a bidirectional bit serial data connection and is controlled by a Data Clock signal and a Frame Synchronization signal. Two RSA processors are placed at the SLD bus interface, one for encrypting transmitted data and one for decrypting received data. The configuration is shown in Figure 3.

Pin *OutputData* at the processor is a tri-state output pin, and the periods

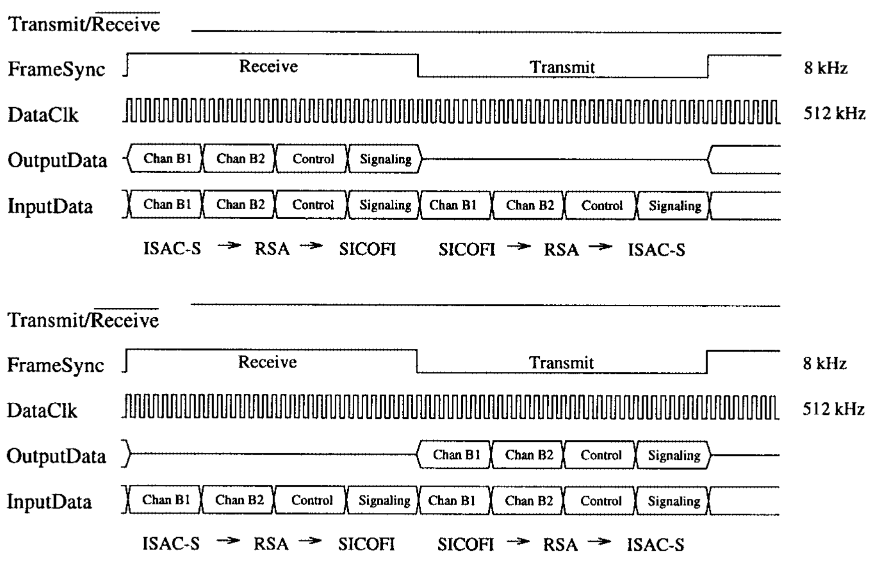


Figure 4: Byte sequence and timing in SLD interface.

with high impedance state are controlled by the configuration pins and the Frame Synchronization signal pin *FrameSync*.

The protocol in the SLD interface is illustrated in Figure 4. In each frame, starting at a positive transition of *FrameSync*, are four bytes transmitted in each direction. The first half frame is denoted the receive direction, where data sent on channel B1 or B2 are decrypted on its way from the Line Subscriber Circuit to the Codec. The second half frame is denoted the transmit direction, where data sent on the selected B channel are encrypted on its way from the Codec to the Line Subscriber Circuit. The data byte associated with the unselected B channel, the control byte, and the signalling byte are sent directly through the RSA processor from pin *InputData* to pin *OutputData*. It is only the byte associated to the selected B channel that is processed by the RSA processors. Note that pin *OutputData* is brought into a high impedance state in one of the half frames in order to be able to connect the RSA processors directly to the bidirectional bit serial data line in the SLD interface.

Pin $Transmit/\overline{Receive}$ must be high when the processor is used for encryption. In the transmit mode parts of the incoming data bits are substituted with a synchronization pattern. This is done in order to enable the receiving processor to recognize the start of a block of encrypted data. When the processor is used for decryption, pin $Transmit/\overline{Receive}$ must be low. In this mode the processor is checking the synchronization pattern, and if an error is detected, it discards the actual data block and search for the start of the next data block. $Transmit/\overline{Receive}$ must be at a high or low logic level when resetting the RSA

Frame	**Input block,** M	**Output block,** C
1	$x\ m_{006}\ m_{005} \dots m_{000}$	$1\ c_{006}\ c_{005} \dots c_{000}$
2	$x\ m_{013}\ m_{012} \dots m_{007}$	$0\ c_{013}\ c_{012} \dots c_{007}$
3	$x\ m_{020}\ m_{019} \dots m_{014}$	$0\ c_{020}\ c_{019} \dots c_{014}$
$\vdots$		
78	$x\ m_{545}\ m_{544} \dots m_{539}$	$0\ c_{545}\ c_{544} \dots c_{539}$
79	$x\ m_{552}\ m_{551} \dots m_{546}$	$c_{553}\ c_{552} \dots c_{547}\ c_{546}$
80	$x\ m_{559}\ m_{558} \dots m_{553}$	$0\ c_{560}\ c_{559} \dots c_{554}$

Table 6: Format of input and output block of RSA processor in transmit, crypt mode.

processor, and a change of value will only have an effect upon a new reset operation. Pin *Channel* is used to select the B channel used for the communication. Channel B1 is selected by a high input and channel B2 by a low input. A change of value will have effect on the next positive transition of *FrameSync*.

The RSA processor can be configured to be completely transparent, i.e. all data at *InputData* is sent directly to *OutputData* without modifying the values or adding a synchronization pattern. In transparent mode the only effect is that the values on *OutputData* are delayed a few nano seconds with respect to the values on *InputData* and that pin *OutputData* is brought into a high impedance state in the half frames determined by pin $\textit{Transmit}/\overline{\textit{Receive}}$. This is controlled by pin *Crypt*, a high value selects the crypt mode and a low value the transparent mode. A change of value will have effect on the next positive transition of *FrameSync*.

When the RSA processor is in transmit and crypt mode a 560 bit data block, $M = m_{559}m_{558} \dots m_0$, is read from the selected B channel in 80 frames. As shown in Table 6 is the least significant bit, denoted x, of the 8 data bit in a frame discarded and used for the synchronization pattern. The remaining 7 bits in 80 frames is modular exponentiated and forms a 561 bit result, $C = c_{560}c_{559} \dots c_0$. This is one bit longer than the input block. In the table 80 bits are discarded per block but only 79 synchronization bits are added because the result block is one bit longer than the input block. Note that the least significant bit in the selected B channel is lost in this synchronization scheme.

When the RSA processor is in receive and crypt mode a 561 bit data block, $C = c_{560}c_{559} \dots c_0$, is read from the selected B channel in 80 frames. As shown in Table 7 the least significant bit of the 8 bit in a frame is mainly used for the synchronization pattern. One of the them is used for data. After discarding 79 synchronization bits the remaining 561 bit are decrypted by performing a modular exponentiation, and the result is a 560 bit block, $M = m_{559}m_{558} \dots m_0$. This block is written in 80 frames where the least significant bit in each frame is set to a high value in order to compensate for the discarded bits in the encryption process of the transmitting processor.

Frame	Input block, M	Output block, C
1	1 c_{006} $c_{005} \ldots c_{000}$	1 m_{006} $m_{005} \ldots m_{000}$
2	0 c_{013} $c_{012} \ldots c_{007}$	1 m_{013} $m_{012} \ldots m_{007}$
3	0 c_{020} $c_{019} \ldots c_{014}$	1 m_{020} $m_{019} \ldots m_{014}$
⋮		
78	0 c_{545} $c_{544} \ldots c_{539}$	1 m_{545} $m_{544} \ldots m_{539}$
79	c_{552} $c_{551} \ldots c_{547} c_{546}$	1 m_{552} $m_{551} \ldots m_{547}$
80	0 c_{560} $c_{559} \ldots c_{554}$	1 m_{559} $m_{558} \ldots m_{553}$

Table 7: Format of input and output block of RSA processor in receive, crypt mode.

The processor checks the synchronization pattern to find the start of a input block. If noise on the transmission line has changed the value of a synchronization bit, the processor is capable of finding the next block. This is what is meant by a *self-synchronizing* interface. Pin *ErrorSync* is raised as soon as an error is found and lowered again when the start of a new block is found.

5 Timing Specifications

The specifications in this section are based on estimates. The time unit "SysClk" denotes the clock period for the system clock.

Symbol	Parameter	Min	Max	Units
TREpl	Reset, widh of low pulse	1		SysClk

Table 8: Timing of signals in reset operation.

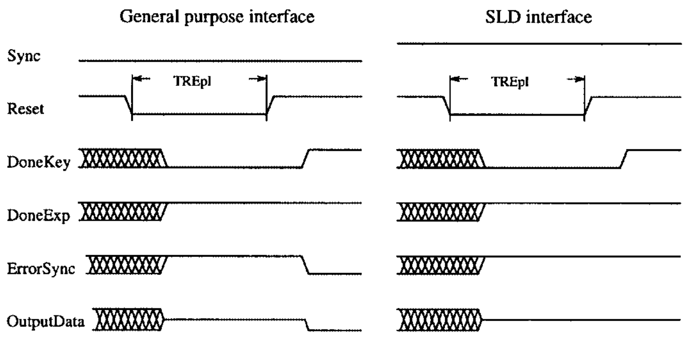


Figure 5: Timing of signals in reset operation.

Symbol	Parameter	Min	Max	Units
TDKdl	DoneKey, delay of negative transition	1	3	SysClk
TDKdh	DoneKey, delay of positive transition	571	573	SysClk
TIKs	InitKey, setup	0		*ns*
TIKh	InitKey, hold	1		SysClk
TIs	InputN and InputE, setup	10		*ns*
TIh	InputN and InputE, hold	10		*ns*
TRKpl	ReadKey, width of low pulse	1		SysClk
TRKph	ReadKey, width of high pulse	1		SysClk
TRKw	ReadKey, width from low to low transition	4		SysClk

Table 9: Timing of signals used for key initialization.

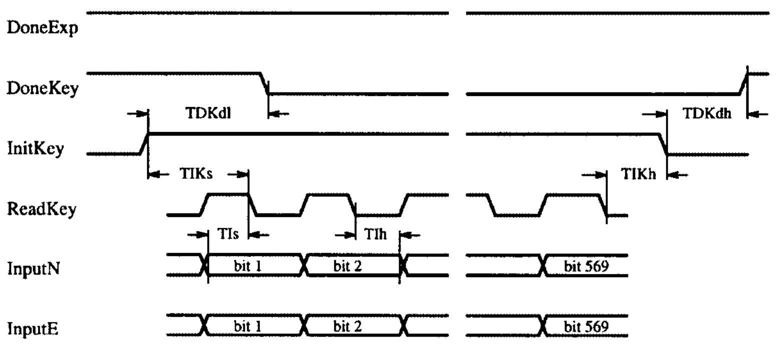


Figure 6: Timing of signals used for key initialization.

Symbol	Parameter	Min	Max	Units
TIDs	InputData, setup	10		*ns*
TIDh	InputData, hold	10		*ns*
TODd	OutputData, delay	0	2	SysClk
TDCpl	DataClk, width of low pulse	1		SysClk
TDCph	DataClk, width of high pulse	1		SysClk
TDCw	DataClk, width from low to next low transition	4		SysClk
TSEs	StartExp, setup	5		SysClk
TSEph	StartExp, width of high pulse	1		SysClk
TSEpl	StartExp, width of low pulse	1		SysClk
TDEd	DoneExp, delay	2	4	SysClk

Table 10: Timing of signals used in general purpose interface.

Symbol	Parameter	Min	Max	Units
TDCd	DataClock, delay	-20	20	*ns*
TDOz	OutputData, delay to/from high impedance state	1	3	SysClk
TFSpl	FrameSync, width of low pulse	1		SysClk
TFSph	FrameSync, width of high pulse	1		SysClk

Table 11: Timing of signals used in SLD interface.

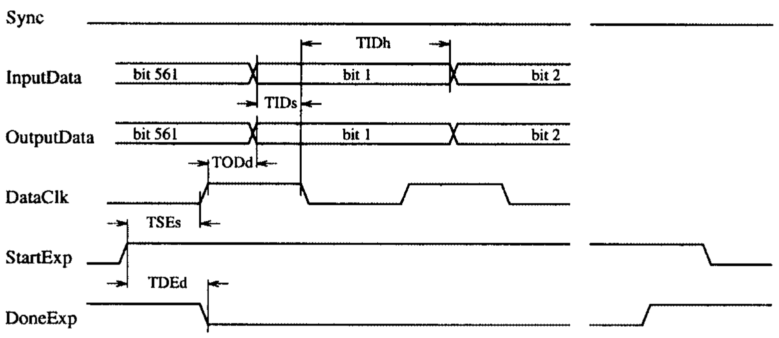


Figure 7: Timing of signals used in general purpose interface.

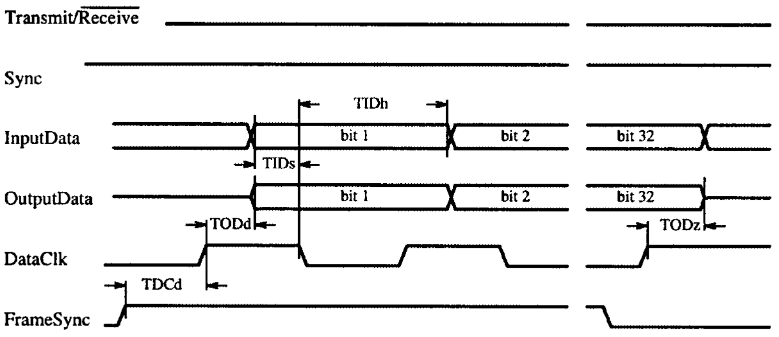


Figure 8: Timing of signals used in SLD interface, receive mode.

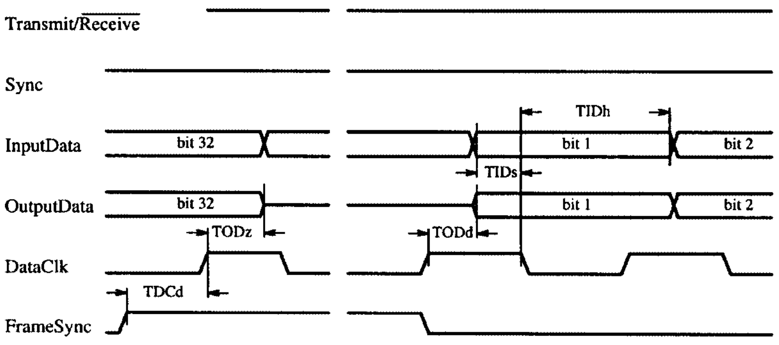
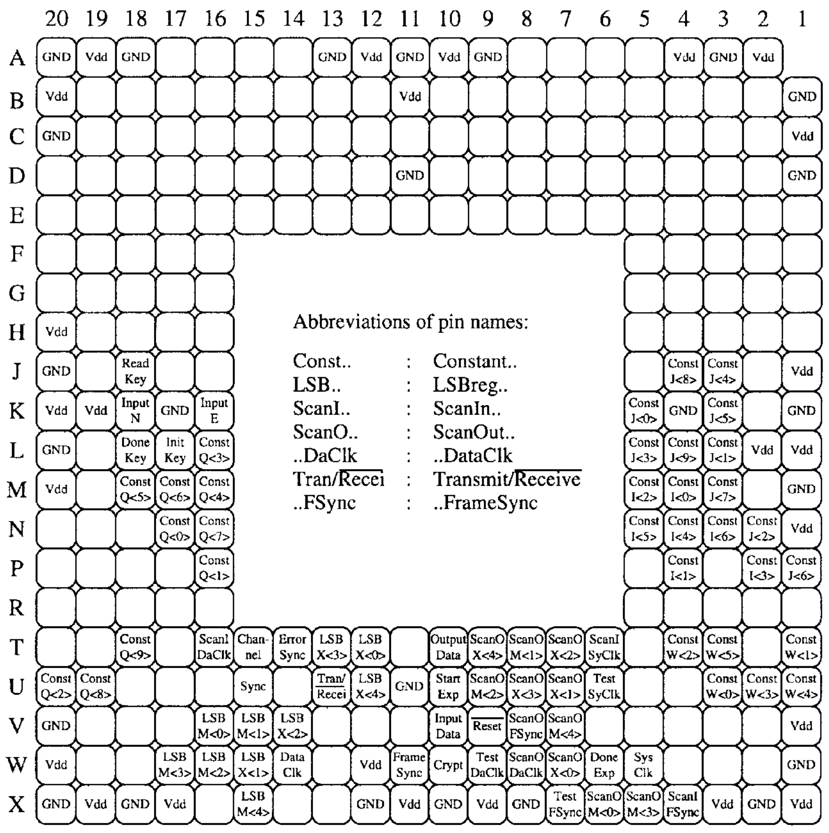


Figure 9: Timing of signals used in SLD interface, transmit mode.

6 Package specification



Abbreviations of pin names:

Const..	:	Constant..
LSB..	:	LSBreg..
ScanI..	:	ScanIn..
ScanO..	:	ScanOut..
..DaClk	:	..DataClk
Tran/$\overline{\text{Recei}}$	:	Transmit/$\overline{\text{Receive}}$
..FSync	:	..FrameSync

Figure 10: 299 ceramic pin grid array package pinout (bottom view).

Pin	Position
Vdd	A2, A4, A10, A12, A19, B11, B20, C1, H20, J1, K19, K20, L1, L2, M20, N1, V1, W12, W20, X1, X3, X9, X11, X17, X19
GND	A3, A9, A11, A13, A18, A20, B1, C20, D1, D11, J20, K1, K4, K17, L20, M1, U11, V20, W1, X2, X8, X10, X12, X18, X20

Table 12: Positions of power pins.

Pin	**Position**	**Pin**	**Position**	**Pin**	**Position**
SysClk	W5	ConstantI<6>	N3	ConstantW<5>	T3
$\overline{\text{Reset}}$	V9	ConstantI<5>	N5	ConstantW<4>	U1
Sync	U15	ConstantI<4>	N4	ConstantW<3>	U3
Crypt	W10	ConstantI<3>	P2	ConstantW<2>	T4
Channel	T15	ConstantI<2>	M5	ConstantW<1>	T1
Transmit/$\overline{\text{Receive}}$	U13	ConstantI<1>	P4	ConstantW<0>	U3
		ConstantI<0>	M4		

Table 13: Positions of pins for system control and mode configuration.

Pin	**Position**
ErrorSync	T14
FrameSync	W11
DataClk	W14
InputData	V10
OutputData	T10
StartExp	U10
DoneExp	W6

Table 14: Positions of pins used for data I/O.

Pin	**Position**	**Pin**	**Position**	**Pin**	**Position**
InitKey	L17	ConstantQ<9>	T18	ConstantJ<9>	L4
ReadKey	J18	ConstantQ<8>	U19	ConstantJ<8>	J4
InputN	K18	ConstantQ<7>	N16	ConstantJ<7>	M3
InputE	K16	ConstantQ<6>	M17	ConstantJ<6>	P1
DoneKey	L18	ConstantQ<5>	M18	ConstantJ<5>	K3
		ConstantQ<4>	M16	ConstantJ<4>	J3
		ConstantQ<3>	L16	ConstantJ<3>	L5
		ConstantQ<2>	U20	ConstantJ<2>	N2
		ConstantQ<1>	P16	ConstantJ<1>	L3
		ConstantQ<0>	N17	ConstantJ<0>	K5

Table 15: Positions of pins used for key initialization.

Pin	Position	Pin	Position
TestSysClk	U6		
ScanInSysClk	T6	ScanOutSysClk	T14
LSBregX<4>	U12	ScanOutRegX<4>	T9
LSBregX<3>	T13	ScanOutRegX<3>	U8
LSBregX<2>	V14	ScanOutRegX<2>	T7
LSBregX<1>	W15	ScanOutRegX<1>	U7
LSBregX<0>	T12	ScanOutRegX<0>	W7
LSBregM<4>	X15	ScanOutRegM<4>	V7
LSBregM<3>	W17	ScanOutRegM<3>	X5
LSBregM<2>	W16	ScanOutRegM<2>	U9
LSBregM<1>	V15	ScanOutRegM<1>	T8
LSBregM<0>	V16	ScanOutRegM<0>	X6
TestDataClk	W9		
ScanInDataClk	T16	ScanOutDataClk	W8
TestFrameSync	X7		
ScanInFrameSync	X4	ScanOutFrameSync	V8

Table 16: Positions of pins for scan chain inspection in test operations.

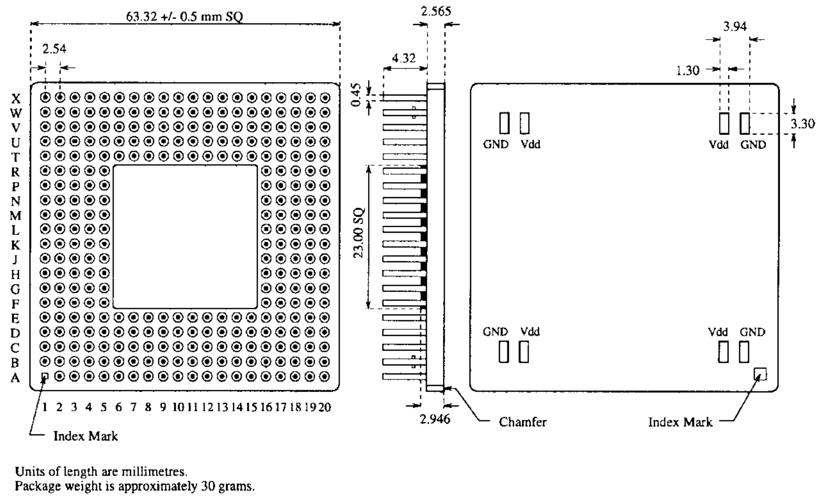


Figure 11: 299 ceramic pin grid array package dimension.

Symbol	Parameter	Min	Nom	Max	Units
$\theta_{J\ A}$	At ?? linear ft/min transverse airflow		23		$°C/W$

Table 17: 299 ceramic pin grid array package junction to ambient thermal resistance.

7 Scan chains

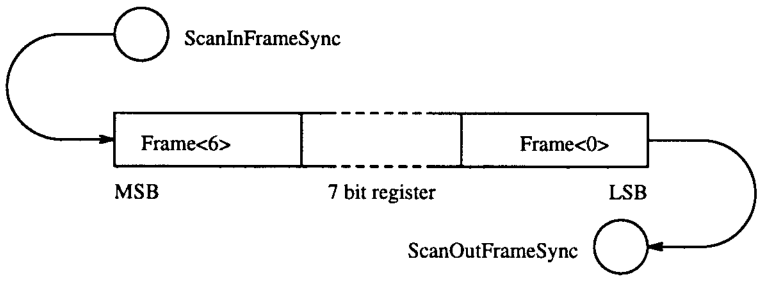


Figure 12: Scan chain controlled by *TestFrameSync*.

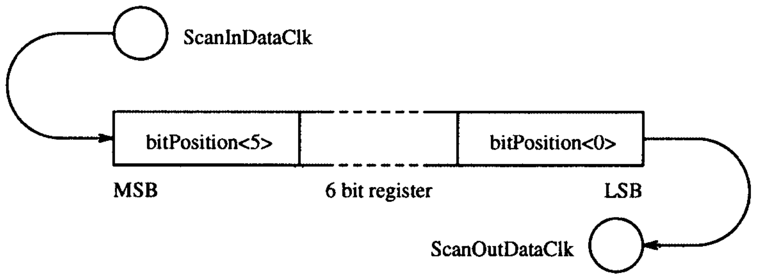


Figure 13: Scan chain controlled by *TestDataClk*.

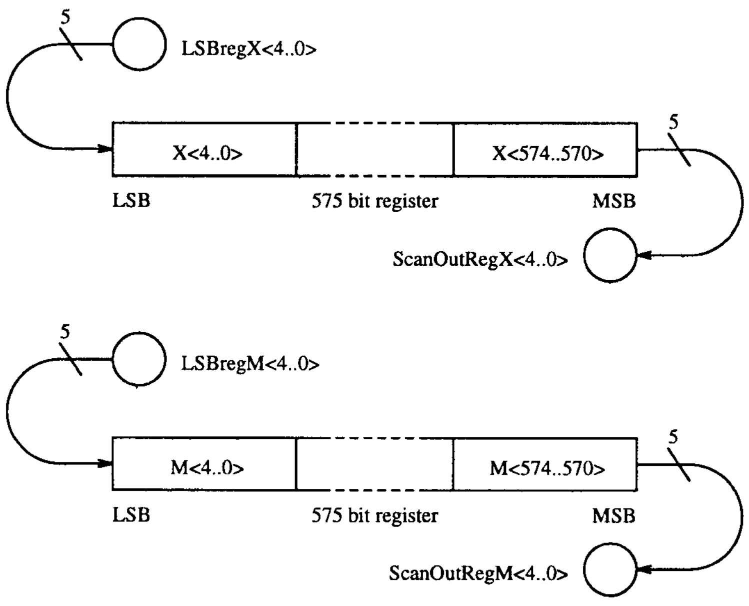


Figure 14: 5 bit scan chains controlled by *TestSysClk*.

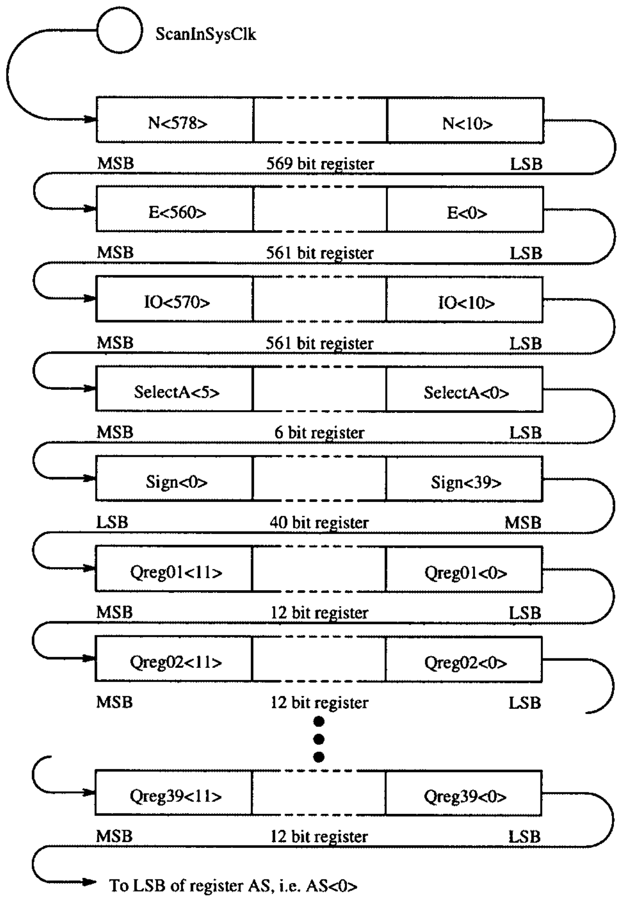


Figure 15: First part of single bit scan chain controlled by *TestSysClk*.

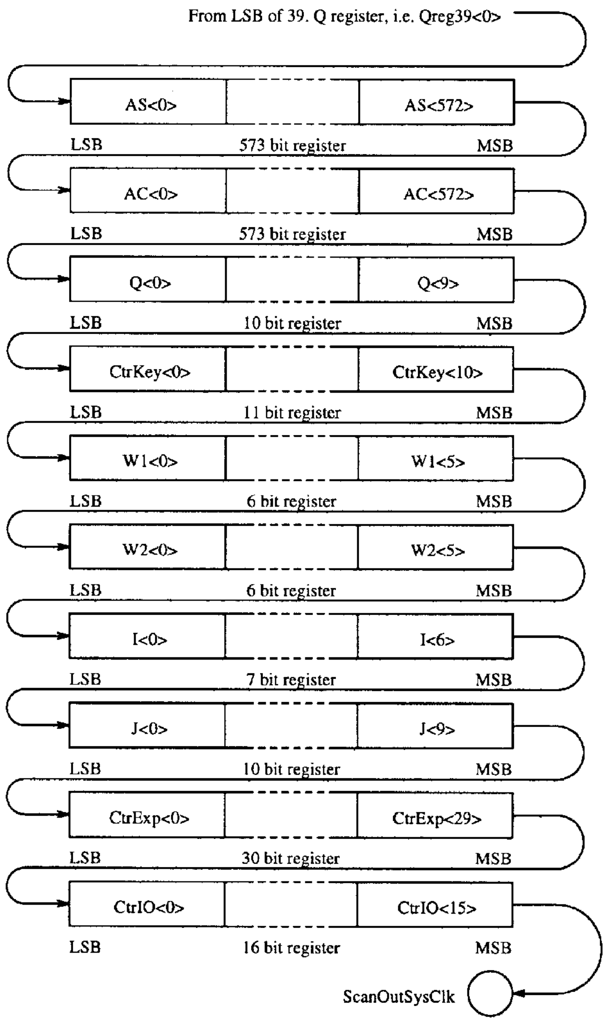


Figure 16: Last part of single bit scan chain controlled by *TestSysClk*.

References

[RSA78] Ronald L. Riverst, A. Shamir, and L. Adlernan. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, volume 21, pages 120–126, Feb. 1978.

[Sie92a] Siemens AG. *ICs for Communications. ISDN Subscriber Access Controller, ISAC-S PEB 2085. User's Manual 02.92*, 1992.

[Sie92b] Siemens AG. *ICs for Communications. Signal Processing Codec Filter, SICOFI PEB 2060, SICOFI-2 PEB 2260. User's Manual 03.92*, 1992.

[Sie92c] Siemens AG. *ICs for Communications. Telecom Handbook 06.92*, 1992.

[WE85] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 1985.