An Optimised Implementation of Public-Key Cryptography for a Smart-Card Processor

Braden Jace Phillips

B.E. (Hons.), B.Sc.

A dissertation submitted in the
Department of Electrical and Electronic Engineering,
University of Adelaide, to meet the requirements for
the award of the degree of Doctor of Philosophy

7 August 2000

Abstract

Smart-cards for public-key cryptography are conventionally based around an 8-bit microprocessor core with a hardware co-processor to accelerate long wordlength arithmetic. While the co-processor can deliver excellent public-key performance, the 8-bit processor and available RAM limit other smart-card applications.

This thesis examines RSA public-key arithmetic with the objective of achieving adequate cryptographic performance without a large hardware co-processor. The chip area thus saved can be used to incorporate a more powerful 32-bit microprocessor as well as extra RAM. The resulting smart-card will better support desirable applications such as on-line media processing or biometric identification.

In the absence of a hardware co-processor, suitable cryptographic performance is achieved through the application of a variety of arithmetic techniques. Significantly, the arithmetic is optimised for average-case rather than worst-case delay. (The timing attack, which exploits variations in delay, is circumvented by a few simple blinding operations.)

Sliding window number representation has been applied to improve average case performance. Sliding windows are studied in detail and a new family of signed sliding window representations is developed. Improved algorithms for multiplication, Montgomery reduction and optimised squaring are presented that take advantage of the new signed representation.

The application of these techniques results in a smart-card design that is novel in many respects: dedicated arithmetic hardware is replaced by extra general-purpose RAM; constant worst-case timing is replaced by average-case execution and blinding; and high radix algorithms requiring a fast hardware multiplier are replaced with low radix algorithms that do not require a multiplier at all.

Contents

Abstract iii

Contents v

List of Figures vii

List of Tables xi

Notation xiii

Declaration xv

Acknowledgment xvii

1 Introduction 1

1 A New Approach 1

2 Thesis Outline 4

3 Original Contributions 5

2 Digit Set Conversion 9

1 Digit Set Conversion for an SRT Divider 10

2 Digit Set Conversion
Formalisation and Existing Results 15

3 Generalised Sliding Windows 25

4 Summary and Conclusions.....48

3 RSA Cryptography 51

1 The RSA Cryptosystem 52

2 Exponentiation 57

3 Attacking RSA Implementations 64

4 RSA Functional Specification 69

5 Summary and Conclusions.....75

4 Modular Reduction 79

1 Background 80

2 Recoded Montgomery Reduction.....95

3 Triangle Additions 102
4 Summary and Conclusions 104

5 Multiple-precision Multiplication and Squaring **107**

1 Background 108
2 Sliding Window Multiple-precision Multiplication 113
3 Optimised Squaring with Sliding Windows 121
4 Summary and Conclusions 134

6 A Smart-Card Implementation of RSA **137**

1 A Survey of Public-Key Smart-Cards 138
2 32-bit Smart-Cards 139
3 A New Smart-Card 142
4 ARM Implementation 145
5 Summary and Conclusions 159

7 Conclusion **163**

Publications **167**

A A Survey of Modular Multipliers **169**

B Analysis of Some Digit Set Conversions **173**

1 Hwang's Radix-r Canonical Conversion 173
2 Recoded m-ary Method 175

C Software for an Unmodified ARM **179**

3 Accumulation 179
4 Montgomery Reduction 183
5 Pre-computation 186
6 Multiplication 188
7 Optimised Squaring 191

D Software for a Modified ARM **197**

E Maple Script for Optimised Squaring with SSW **203**

Bibliography **207**

List of Figures

2.1 A Self-Timed SRT Divider . . . 12
2.2 SRT Quotient Digit Selection Logic . . . 14
2.3 Evaluation Times for SRT Division . . . 14
2.4 Example of Booth Recoding . . . 18
2.5 Example of Modified Booth Recoding . . . 19
2.6 An Example of Binary Canonic Conversion . . . 22
2.7 Hwang's Conversion . . . 23
2.8 Cohen's Window Conversion . . . 24
2.9 An Example of Sliding Window Conversion . . . 24
2.10 A Difficult Digit Set Conversion . . . 26
2.11 Another Difficult Digit Set Conversion . . . 26
2.12 An Example of USW Conversion . . . 28
2.13 State Diagram for USWr,m . . . 31
2.14 An Example of the Worst Case for USW . . . 32
2.15 Weight Distributions for USW . . . 34
2.16 An Example of SSW Conversion . . . 36
2.17 State Diagram for SSWr,m . . . 40
2.18 An Example of the Worst Case for SSWr,m with r > 2 . . . 41
2.19 An Example of the Worst Case for SSW2,m . . . 42
2.20 An Example SSW Window Arrangement . . . 42
2.21 Weight Distributions for SSW . . . 44
2.22 Selection of a Non-zero Digit . . . 44
2.23 Adaptive m-ary Segmentation Canonical Encoding and MSW . . . 45
2.24 Radix-2 Sliding Windows . . . 46
2.25 A Comparison of Sliding Window Conversions . . . 48

3.1 The First 5 Rows of the Power Tree 58
3.2 Average Modular Multiplications for Exponentiation with USW2,m 61
3.3 Bos and Coster's Digit-Set-Conversion 62
3.4 Yacobi's Compression Technique 63
3.5 A High Level Description of RSA Signature Generation 74
3.6 RSA Signature Generation 75

4.1 An Example of Montgomery Reduction 90
4.2 An Example of Montgomery Reduction using USW 97
4.3 An Example of Montgomery Reduction using MSW 98
4.4 Addition Corresponding to the Selection of a Positive Quotient Digit 99
4.5 Subtraction Corresponding to the Selection of a Negative Quotient Digit 99
4.6 An Example of Montgomery Reduction using SSW 100

5.1 Multiplication as a Sum of Partial Products 109
5.2 Average Total Accumulations for USW2,m Multiplication 116
5.3 Average Total Accumulations for SSW2,m Multiplication 117
5.4 Karatsuba Multiplication 119
5.5 Multiple-precision Square 121
5.6 Optimised Multiple-precision Square 122
5.7 An Example of Squaring a Converted Number 122
5.8 An Example of Squaring Combined with Sliding Window Conversion 123
5.9 A Problem with Pre-computed Partial Squares 124
5.10 A Second Problem with Pre-computed Partial Squares 125
5.11 Evaluating Partial Squares 126
5.12 The Generation of ci+m 126
5.13 Optimised Square using Unsigned Sliding Windows and Pre-computed Partial Squares. 129
5.14 A Problem with Pre-computed Partial Squares and Signed Sliding Windows 130
5.15 Optimised Square using Signed Sliding Windows and Pre-computed Partial Squares... 132

6.1 The Accumulation Function 149
6.2 A Simple Loop for Multiple-precision Accumulation 149
6.3 Accumulation Procedure 150
6.4 An Improved Loop for Multiple-Precision Accumulation 151

6.5 Schematic for the Extended Shift Left Mechanism 152
6.6 Multiple-precision Accumulation with the Hardware Enhancements 154

7.1 Bit Arrangements That Lead to Case 1 175
7.2 State Transitions Leading to Zero Digits 177

List of Tables

2.1 Improved Quotient Selection Logic 13
2.2 Booth Recoding 18
2.3 Modified Booth Recoding 18
2.4 An Overview of Digit Set Conversions 50

4.1 Number of Operations for 3 Modular Multiplication Algorithms on a 32-bit Processor .. 95

5.1 Design Trade-off for 512-bit USW2,m Multiplication 115
5.2 Design Trade-off for 512-bit SSW2,m Multiplication 117
5.3 The Influence of Karatsuba on SSW2,m Multiplication 120

6.1 A Summary of Commercial Smart-Cards 139
6.2 Comparing a Conventional Smart-Card and a 32-bit Smart-Card 142
6.3 Modular Multiplications versus Memory Requirement 146
6.4 Execution Times for the 512-bit Montgomery Reduction Function 146
6.5 Execution Times for the 512-bit Product Function 155
6.6 Execution Times for the 512-bit Square Function 156
6.7 Unmodified ARM Signature Generation Time 157
6.8 Improved ARM Signature Generation Time 157
6.9 Improved Execution Times with the ESR Shift Register 158
6.10 Further Improved ARM Signature Generation Time 159
6.11 Comparing the CASCADE Smart-Card with the Proposed Smart-Card 161

7.1 A Survey of Modular Multipliers 170

7.2 Recoded Binary Method Conversion Table 176

Notation

Number Representation

A	The multiple-precision representation of an integer.
$\|A\|$	The arithmetic value of the representation A.
$\|A\|$	The magnitude of the number $\|A\|$.
n	The number of digits in the representation of integers in a system and in particular the number of bits in the modulus of an RSA system.
r	The radix of number representation in a system.
a_i	The ith digit of the representation A such that $A = \sum_{i=0}^{n-1} a_i r^i$.
$c(A)$	The arithmetic weight of the representation A.
$m(\|A\|)$	The minimum possible arithmetic weight for a representation of $\|A\|$.
$\bar{c}(F)$	The average arithmetic weight resulting from the digit set conversion F.
m	The window size in digits of a sliding window digit set conversion.
$USW_{r,m}$	Unsigned sliding window digit set conversion with radix r and window size m-digits.
$SSW_{r,m}$	Signed sliding window digit set conversion with radix r and window size m-digits.
$MSW_{r,m,z}$	Modified signed sliding window digit set conversion with radix r, window size m-digits and maximum digit magnitude z.
$\bar{1}, \bar{2}, \ldots, \bar{x}$	Indicate negative digits in a representation i.e. $-1, -2, \ldots, -x$.
Æ	Indicates a 'don't care' state in a logic equation.

Set Notation

X	A set of scalars.
X^n	A set of n-bit vectors.
$\|X\|$	The cardinality (number of members) of the set X.
N	The set of all natural numbers.
Z	The set of all integer numbers.

Common Symbols

N	The n-bit modulus of an RSA implementation.
P and Q	The $n/2$-bit prime factors N.
M	An n-bit message to sign.
S	An n-bit RSA signature.

K An RSA secret exponent.

e An RSA public exponent.

w The wordlength of a processor in bits.

l The modulus length in processor words.

Functions and Operators

$\lfloor a \rfloor$ The floor of a. The largest integer less than or equal to a.

$\lceil a \rceil$ The top of a. The smallest integer greater than or equal to a.

a mod b Modular reduction of a modulo b (equal to $a - b\lfloor a/b \rfloor$).

a div b The integer quotient of a divided by b (equal to $\lfloor a/b \rfloor$).

$A \equiv B \pmod{N}$ A is congruent to B modulo N (implies that N divides $(A - B)$).

GCD(a, b) The greatest common divisor of a and b.

$MR_N(A)$ Montgomery reduction of A modulo N.

$MP_N(A, B)$ Montgomery product of A and B modulo N.

$ME_N(A, e)$ Montgomery exponentiation of A raised to the power of e modulo N.

$\bar{A}$ The Montgomery N-residue of A.

$a \leftarrow a + b$ The value of a becomes equal to the sum of b and the old value of a.

Logical Operators

$A \gg i$ Logically shift A right by i bits (equal to A div 2^i).

$A \ll i$ Logically shift A left by i bits (equal to $A \times 2^i$).

$a \oplus b$ The exclusive or of a and b.

$a \wedge b$ The logical and of a and b.

$a \vee b$ The inclusive or of a and b.

Abbreviations and Terminology

$n\%$ If system X is said to be $n\%$ faster than system Y, or X is an $n\%$ improvement over Y, then:

$$n = 100 \times \left(\frac{\text{Execution time of Y}}{\text{Execution time of X}} - 1 \right).$$

CRT Chinese Remainder Theorem.

GF Galois Field.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

7/12/00
Date

Acknowledgment

Thanks are due to my supervisors, Neil Burgess, who oversaw the lion's share of my 6½ year candidature, and Mike Liebelt who kindly stepped in for the critical final stages.

Throughout the course of my study I have received great support from, and become great friends with my colleagues at the University of Adelaide. Working with the 'HI Performance Computer Architecture Team' has been a highlight and I would like to thank Nick Betts, Sam Appleton, Shannon Morton and Andrew Johnson for their enthusiasm, humour and verve. I have also benefited from many fruitful (and also many wonderfully irrelevant) discussions with Andrew Blanksby, Richard Beare, Lama Chandrasena, Nasser Asgari, Said Al-Sarawi, Chris Howland, Ali Moini and Kiet To. Andrew Beaumont-Smith has been an endless source of help and information.

I would also like to acknowledge the careful review of this thesis by the two anonymous examiners and thank them for their constructive suggestions.

Throughout my candidature I have relied on support and encouragement from my family and friends. Thanks to Mum, Dad and Chad for believing I could get there. My thanks also to Denny Rosey, David and Liz Panton for adopting me. My friends outside of study have always been an important part of my life and I am especially fortunate that so many deserve thanks that I cannot possibly list them individually. Special mention should be made of my business partner Chris Schach who put up with me being distracted by research for so long. (The same is true of Neil Burgess who put up with me being distracted by business.) Thanks also to the congregation at St. Andrew's Walkerville and especially Malcolm Smith for their welcome and friendship over the past two years.

Jenny, you are my greatest support, inspiration and love. Thank you for all you have done for me.

Chapter 1

Introduction

PUBLIC-KEY CRYPTOGRAPHY AND SMART-CARDS appear to be ideal partners. Using a public-key cryptosystem it is possible to authenticate the identity of individuals, ensure the privacy of communications and validate the integrity of data. However, cryptography alone cannot guarantee a security. A secure system requires a trusted hardware element to store private keys and to perform critical arithmetic operations. Smart-cards meet this need in a convenient and portable form.

There is an unhappy aspect to this seemingly ideal marriage. The arithmetic operations required by public-key cryptosystems are notoriously complicated, and on the other hand, smart-cards are tightly constrained in terms of circuit size, power dissipation and memory capacity. Thus it is widely held that smart-cards which make use of public-key cryptography must contain a hardware co-processor if they are to perform cryptographic transactions in a timely fashion. In this work I challenge this assertion and demonstrate the viability of an alternative.

1 A New Approach

Commercial cryptographic smart-cards of the current generation are based on a short wordlength microprocessor with a hardware co-processor to perform the long wordlength arithmetic required by public-key cryptosystems. Typical examples use an 8-bit microprocessor with a co-processor consisting of 512-bit or 1024-bit registers, adders and shifters. This arrangement does achieve

excellent cryptographic performance; however, there are other applications for which an 8-bit processor is insufficient and for which the specialised co-processor is of no benefit. Examples include biometric processing for the identification of individuals, on-line processing of data for copyright protection or access control, and voice recognition for user-friendly smart-card interfaces. Most smart-card processors also lack support for high level languages and thereby impede the development of new applications; nor do they support protected mode operating systems capable of simultaneous execution of multiple applications from different vendors.

In recent years an alternative approach has emerged in which the small smart-card processor and specialised cryptographic hardware are replaced by a more powerful general purpose processor. A smart-card equipped with a general purpose 32-bit RISC processor provides a very flexible platform for a diverse set of applications. It will support a protected mode operating system and higher level languages and is more likely to be sufficient for new applications such as biometry. However, if this approach is to be viable, it must still deliver adequate cryptographic performance.

This thesis explores the arithmetic of public-key cryptography with a view to implementation on a 32-bit smart-card processor. Satisfactory cryptographic performance is achieved by combining new and existing algorithmic optimisations. In so doing, an extremely novel solution has emerged that challenges many of the assertions of conventional smart-card design.

1.1 The RAM Constraint

The development of algorithms for smart-card cryptography invariably begins with the assumption that the available RAM will be extremely limited. However, advances in semiconductor manufacture mean that this RAM constraint is changing. Surveys in 1996 and 1998 reported in Chapter 6 reveal that the RAM on a typical smart-card more than doubled over that two year period.

While it is true that the RAM on a smart-card will always be limited, it is now appropriate to consider what can be achieved when the RAM constraint is relaxed somewhat. The algorithms developed in this thesis take advantage of extra RAM to pre-compute and re-use intermediate results. As the power dissipation of a smart-card is still extremely limited, it is preferable to improve performance through the utilisation of RAM in this way than by increasing the clock rate.

Provision of extra RAM also conforms to the philosophy of providing general mechanisms suitable to the acceleration of a wide variety of applications. The technique of storing intermediate results for re-use is widely applicable to many algorithms. In addition, extra RAM can be used to store other temporary data such as communications buffers, session keys, certificates or file pointers.

1.2 Average Case Execution

In most instances this work explores the possibility of improving average case rather than worst case delay. This is appropriate for software and asynchronous hardware systems that are able to take advantage of variable evaluation time. That modern software exhibits variable completion time is a result of many factors such as multi-tasking and hierarchical memory. In these systems, the worst case delay is still a consideration, but average case is more indicative of the user's experience—particularly if the distribution of delays is such that the worst case occurs only infrequently.

The use of average case delay makes possible a number of optimisations that do nothing to improve the worst case. In particular, significant improvements are possible using the digit-set conversion techniques developed in Chapter 2.

One problem with cryptographic systems that exhibit variable delay is that they may be susceptible to the timing attack as described by Kocher [Koc96]. Since the publication of this attack many designers have been wary of average case execution and chosen instead to try and implement a constant worst case delay. This is not only costly in terms of performance and design effort but also susceptible to unforeseen sources of timing variation. The solution is the use of blinding operations as described in Chapter 3.

1.3 Improved Algorithms

To deliver adequate cryptographic performance without a hardware processor will require improved arithmetic algorithms. Where a quantum leap in algorithmic complexity is not forthcoming, improvements can still be made by reducing the coefficient terms in the complexity equation. In this instance timely cryptographic performance is achieved by combining an armoury of optimisations—especially those that improve the average case of execution.

Sliding window digit-set conversion has been previously applied to the operation of exponentiation for cryptography. In Chapter 2 sliding windows are studied thoroughly. The well-known binary unsigned sliding windows are shown to be a specific case of a general class of sliding window representations that extend to higher radices and may include signed digits. A number of useful general results are developed, and in subsequent chapters, sliding windows are applied to the operations of multiplication, Montgomery reduction and optimised squaring.

1.4 Fast Multipliers

Existing software implementations of public-key cryptography rely on a fast 32-by32-bit multiply instruction [Dhe98], [DK91], [KAK96]. However, a fast multiplier occupies a significant area of valuable smart-card real estate.

An interesting implication of the new sliding window algorithms is that arithmetic performance no longer relies on a fast multiplier. Instead, all arithmetic operations are reduced to additions, subtractions and shifts. While a hardware multiplier is likely to be useful for other applications, it may be sufficient to replace the fast multiplier with a smaller, slower unit. The chip area saved could be put to better use as extra RAM.

2 Thesis Outline

This thesis examines the arithmetic of public-key cryptography, working towards a software implementation in that is described in Chapter 6, and which demonstrates the viability of 1024-bit RSA on a 32-bit smart-card. Each of the chapters leading up to this point tackles a separate aspect of the design, beginning with a review of the current state of the art before moving on to the development of new ideas or techniques.

Chapter 2 deals with the idea of digit-set conversion, an algorithmic technique that is applied in every subsequent chapter. The chapter begins with the study of a circuit for arithmetic division. This design raises many of the critical issues surrounding digit-set conversion for improved average case execution. It was the starting point for the author's study of digit-set conversion and demonstrates the application of this technique to asynchronous hardware.

A notation and formalism is developed to describe digit-set conversion. This is used to study existing published applications of the technique, and to reveal some instances where published work is either sub-optimal or erroneous. Finally, a general family of sliding window digit-set conversions is developed, along with some useful general results and proofs. These sliding window conversions form the basis of improved arithmetic algorithms presented in subsequent chapters.

Chapter 3 presents the RSA cryptosystem and considers the requirements for an implementation of RSA. RSA signature generation is chosen as a benchmark operation representative of a wide variety of public-key cryptosystems. The critical operation in RSA signature generation, and a common feature of most public-key cryptosystems, is long wordlength modular exponentiation. A survey of algorithms follows.

Consideration is given to the security of smart-card implementations with special attention given to circumventing the timing attack and the fault attack. Chapter 3 concludes with the development of a high level specification of RSA signature generation. The Chinese Remainder Theorem, Montgomery arithmetic, blinding and verification are combined for a secure yet efficient implementation in which some of the operations of blinding and Montgomery transformation are combined to reduce computational overhead.

Chapter 4 is concerned with modular reduction. A substantial literature review is conducted. This succeeds in identifying 4 categories of modular reduction into which all surveyed implementations can be placed. A comparison of the 4 categories yields some interesting parallels, and while it is not clear that any one technique will ever be best for all circumstances, Montgomery's reduction is chosen for further investigation and a new algorithm is described that incorporates sliding window digit-set conversion.

Chapter 5 develops software implementations of multiplication and squaring. Once again surveys of existing implementations are followed by improved algorithms that incorporate sliding windows. The optimised squaring algorithms with sliding windows are of particular interest, firstly because of the importance of squaring to exponentiation and hence cryptography, but also because the ideas of optimised squaring and sliding windows have hitherto been considered to be mutually exclusive. Karatsuba's optimisation is applied to the sliding window algorithms and demonstrates the potential to reduce the memory required to achieve a given level of performance.

Chapter 6 begins with a survey of existing commercial smart-cards and discusses the need for a new type of 32-bit smart-card. The algorithmic techniques of previous chapters culminate in a software implementation of 1024-bit RSA for a 32-bit smart-card. Critical sections of the code are discussed in detail and the performance of the system is measured to demonstrate the viability of public-key cryptography on the new card. Some enhancements to the processor's architecture are also developed and their impact on the system's performance is evaluated.

Chapter 7 concludes the work and suggests some areas for further study.

3 Original Contributions

This thesis describes a novel and timely approach to the arithmetic implementation of cryptographic functions, which employs more RAM but no dedicated hardware co-processor or even a hardware multiplier.

A fundamental aspect of the implementation is the use of sliding window digit-set conversions. The author has analysed existing digit-set conversions for average arithmetic weight (Chapter 2 Section 2) and generalised an existing conversion, unsigned binary sliding windows, to account for higher radices as well as different digit lengths (Chapter 2 Section 3.1). The idea of SSW signed sliding windows, as defined in Chapter 2 Section 3.2, is to the best of my knowledge, entirely new (other than for the special binary case $SSW_{2,1}$). The new sliding windows have been formally studied to prove they achieve a minimal arithmetic weight for a given digit-set, to derive the theoretical average arithmetic weight at long wordlengths, and to find the distribution of arithmetic weights. A contribution of this work is to bring together many fragmented studies under a more general framework and to formalise some of the problems outstanding.

The study of a combined attack using both the fault and timing attacks (Chapter 3 Section 3) is the work of the author as is the high level description of RSA signature generation incorporating Chinese Remainders, Montgomery reduction, blinding and verification, with Montgomery transformations and blinding operations combined (Chapter 3 Section 4).

The author has developed new algorithms for Montgomery reduction (Chapter 4 Section 2), multiplication (Chapter 5 Section 2) and optimised squaring (Chapter 5 Section 3) that incorporate signed sliding windows, implemented these in assembly language for an ARM processor and tested the implementation (Chapter 6). Optimised squaring with sliding windows is a new idea as is the application of sliding windows to Montgomery reduction. A form of signed sliding windows and multiplication have appeared together before [KH92], but in that case the authors did not observe that the negative partial products need not be evaluated (a critical observation if any advantage is to be gained over unsigned sliding windows).

The estimation of RAM size in Chapter 6 Section 3.1 is due to Andrew Beaumont-Smith.

New enhancements to the ARM architecture have also been proposed to improve the accumulation of shifted multiple-precision numbers. The application of Karatsuba to sliding window multiplication and squaring to reduce memory for pre-computed results (Chapter 5 Section 2 and Section 3), and the re-casting of Takagi's triangle additions to Montgomery multiplication (Chapter 4 Section 3), are two more original (albeit supplementary) contributions.

The champion of the new 32-bit approach to smart-cards has been the European CASCADE project as described at http://www.dice.ucl.ac.be/crypto/cascade. The details of the cryptographic library for the CASCADE smart-card were published by Dhem [Dhe98]. The design described in the current thesis was developed simultaneously with the CASCADE research and the resulting implementation

differs significantly from the CASCADE version in many fundamental aspects as described in Chapter 6.

Chapter 2

Digit Set Conversion

EVALUATION OF ARITHMETIC functions can be simplified by choosing an appropriate number representation. Radix or digit set can be selected to suit the characteristics of an algorithm or implementation technology. Booth multiplication is an example of the latter—changing the digit set of the multiplier minimises the number of partial products—whereas modified Booth multiplication uses both techniques—the radix and digit set are changed so that the number of partial products is halved.

Changing the radix and digit set used to represent a number can achieve a number of benefits: the frequency of useful digits such as 0 or 1 can be increased and the total number of digits required to represent a number can be reduced. The cardinality of the digit set can be reduced and this in turn reduces the number of pre-computed values to evaluate and store. Reduced cardinality also simplifies digit encoding for hardware implementation, and increases the frequency of digits and hence the benefit available from pre-computation of partial results.

It is usually necessary to trade these benefits one against the other. For example, increasing the radix may reduce the number of digits required to represent a number but it may also increase the cardinality of the digit set.

Manipulation of number representation in this way is a fundamental technique in computer arithmetic. It provides an endless succession of publications as for almost every change in implementation or algorithm, a different number representation becomes optimal. In most papers the

digit set conversion is implicit in the algorithm and not studied directly. Publications that deal with number representation in a general fashion are more rare. The situation is expressed in [Kor94a]:

> Although there are notable exceptions of digit-level designs, very often arithmetic algorithms are described without a clear separation between the issues of the algorithm and its logic implementation. This is not to imply that the encoding is unimportant, but it cannot change the fundamental properties of the algorithms. In complexity terminology the encoding chosen can only change the time and area by constant factors, but is as important as layout and floor planning is in VLSI implementations.

This chapter starts with a specific instance of digit set conversion and moves towards a more general study. It begins with an example in which a digit set conversion is used to optimise a hardware divider. This gives form to a more general problem and the solutions that emerge are the foundation for the cryptographic algorithms presented in subsequent chapters.

1 Digit Set Conversion for an SRT Divider

The SRT algorithm for division has been extensively studied [Bur94] and in the absence of a leap in algorithmic complexity, research efforts have turned to incremental enhancements through optimised implementation. Some impressive hardware architectures have been the result [WH91].

This section concerns the author's attempt to improve the performance of a self-timed SRT divider [PB95]. The result was twofold: an enhanced quotient digit selection stage that improved the average case performance of the divider; and the groundwork for a subsequent study of digit set conversion. The quotient digit selection techniques developed here are re-applied to modular reduction in Chapter 4. This work also serves to demonstrate the application of digit set conversion to hardware systems as the remainder of this thesis concentrates on software implementation.

1.1 SRT Algorithm

The radix-2 SRT algorithm is expressed as an iteration by Equation 2.1 and Equation 2.2.

$$X'_i = X_i - q_i D \qquad (2.1)$$

$$X_{i+1} = 2X'_i \qquad (2.2)$$

D is the divisor expressed in unsigned binary ($D = d_0.d_{-1}\ldots$) and X_0 is the dividend expressed in 2's complement binary. After i iterations the partial remainder will be $X_i = x_2x_1x_0.x_{-1}\ldots$. The quotient $Q = q_0.q_1q_2\ldots q_m$ is generated digit by digit with each digit selected to be 1,-1 or 0.

The algorithm proceeds by choosing a quotient digit q_i at each iteration so that the resultant X_{i+1} remains within the bounds $|X_{i+1}| \le 2D$. Whenever $X_i > 0$ it is valid to select $q_i = 1$ and whenever $X_i < 0$ it is valid to select $q_i = -1$. It is also valid to select $q_i = 0$ whenever $|X_i| < D$. As D is normalised to be greater than 1, it is possible to choose $q_i = 0$ whenever $|X_i| < 1$. The procedure is shown in pseudo code as Algorithm 2.1.

Algorithm 2.1 SRT Division in Pseudo Code. Performs the division Q = X / D for $X \in [1,2)$ and $D \in [1,2)$.

$X_0 = X$

for i = 0 to m

q_i = quotient_digit_select(X_i) // q_i is 0, 1 or -1

$X_{i+1} = 2(X_i - q_i \times D)$ // $|X_{i+1}| \le 2D$

next i

$$Q = \sum_{i=0}^{m} q_{(-i)} \times 2^{(-i)}$$

1.2 Self-timed Implementation

In [WH91], the divider is implemented using a self-timed ring. Iterative stages are constructed using domino logic and connected in a ring (Figure 2.1). Evaluation cascades around the ring, with the stages behind the wavefront precharging ready for their next iteration. In this way evaluation time is determined by the switching time of the gates and not by the speed of a global clock.

Each stage of the ring must first examine the partial remainder X_i to select a quotient digit q_i. This is used to select either 0, D or $-D$ as input to the adder. Finally the adder must determine X_{i+1} before the next stage can begin quotient digit selection. A borrow save adder is used so that addition can be performed without carry propagation. Therefore, X_i is in signed-digit notation with $x_i \in \{-1, 0, 1\}$.

The switching time of each gate will vary with the inputs so that the time to perform a division will depend upon the inputs and the performance of the divider is characterised by a distribution of evaluation times. In an asynchronous system able to advantage of variable evaluation time, the average evaluation time may be more important than the worst case time.

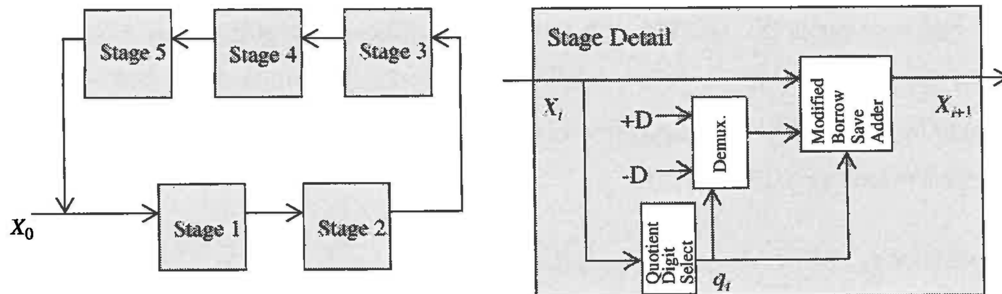

Figure 2.1 A Self-Timed SRT Divider.

1.3 Quotient Digit Selection

Quotient digit selection can improve the average evaluation time in two ways:

1. Whenever $q_i = 0$ is selected Equation 2.1 becomes trivial. The adder in Figure 2.1 can be designed to quickly generate $X_{i+1} = 2X_i$ without waiting for the demultiplexor.
2. Quotient digit selection is on the critical path. If q_i can be selected quickly under some circumstances then this will improve average evaluation time.

The rules for quotient digit selection in Section 1.1 show that it is sufficient to choose $q_i = 1$ when $X_i \geq 0$ and $q_i = -1$ when $X_i < 0$. However, to maximise the first benefit $q_i = 0$ should be chosen whenever possible – that is whenever $1 < X_i < 1$.

Put another way, the selection of q_i can be seen as a digit set conversion problem. If the redundant partial remainder is converted to non-redundant form then quotient selection becomes trivial. If the integer part of X_i is zero then it is possible to choose $q_i = 0$; if the integer part of X_i is greater then zero then $q_i = 1$; and if the integer part of X_i is less than zero then $q_i = -1$.

Conversion from redundant borrow-save form to non-redundant form requires full length carry propagation [Kor94a]. Clearly this is out of the question and a compromise must be met between the two benefits. The optimal quotient selection logic will depend upon their relative merits.

1.4 Improved Quotient Digit Selection

Consider examining the redundant partial remainder, starting with the most significant digit x_2. If this is 1 then $X_i > 0$ and it is possible to select $q_i = 1$ immediately. Alternatively, we may go on to examine x_1. If this is anything other than -1 then $X_i > 1$ and $q_i = 1$ must be selected; however, if $x_1 = -1$ then X_i may yet be less than 1 and further examination of digits may allow the selection of $q_i = 0$.

In domino hardware, the effect of examining more and more digits is to increase the complexity of the logic gates and hence slow down quotient digit selection. In the example above, the logic to select $q_i = 1$ whenever $x_2 = 1$ would be small and fast; extending it to consider more digits will slow it down, but may increase the frequency zero quotient digits.

In [PB95], the optimal quotient digit selection scheme was sought for a GaAs CCDL (capacitively coupled domino logic) implementation. To facilitate self-timing, each digit is encoded over three wires: x_i^p, x_i^n and x_i^z. The resulting logic is shown in Table 2.1 and one of the GaAs logic gates is shown in Figure 2.2. In CCDL the gate delay depends strongly upon the number of transistors in the pull-down path and hence upon the length of the product term that switches the gate.

x_2	x_1	x_0	x_{-1}	x_{-2}	q_i	q_i^p product term	q_i^n product term	q_i^z product term
1	∅	Æ	Æ	Æ	1	x_2^p		
–1	∅	Æ	Æ	Æ	–1		x_2^n	
0	1	∅	Æ	Æ	1	$x_2^z \cdot x_1^p$		
0	–1	∅	Æ	Æ	–1		$x_2^z \cdot x_1^n$	
0	0	1	0	0	1	$x_2^z \cdot x_1^z \cdot x_0^p \cdot x_{-1}^z \cdot x_{-2}^z$		
0	0	1	0	1	1	$x_2^z \cdot x_1^z \cdot x_0^p \cdot x_{-1}^z \cdot x_{-2}^p$		
0	0	1	1	∅	1	$x_2^z \cdot x_1^z \cdot x_0^p \cdot x_{-1}^p$		
0	0	1	0	–1	0			$x_2^z \cdot x_1^z \cdot x_0^p \cdot x_{-1}^z \cdot x_{-2}^n$
0	0	1	–1	∅	0			$x_2^z \cdot x_1^z \cdot x_0^z \cdot x_{-1}^n$
0	0	–1	0	0	–1		$x_2^z \cdot x_1^z \cdot x_0^n \cdot x_{-1}^z \cdot x_{-2}^z$	
0	0	–1	0	1	0			$x_2^z \cdot x_1^z \cdot x_0^n \cdot x_{-1}^z \cdot x_{-2}^p$
0	0	–1	1	∅	0			$x_2^z \cdot x_1^z \cdot x_0^n \cdot x_{-1}^p$
0	0	–1	0	–1	–1		$x_2^z \cdot x_1^z \cdot x_0^n \cdot x_{-1}^z \cdot x_{-2}^n$	
0	0	–1	–1	∅	–1		$x_2^z \cdot x_1^z \cdot x_0^n \cdot x_{-1}^n$	
0	0	0	∅	Æ	0			$x_2^z \cdot x_1^z \cdot x_0^z$

Table 2.1 Improved Quotient Selection Logic. Logical 'don't cares' are indicated as Æ.

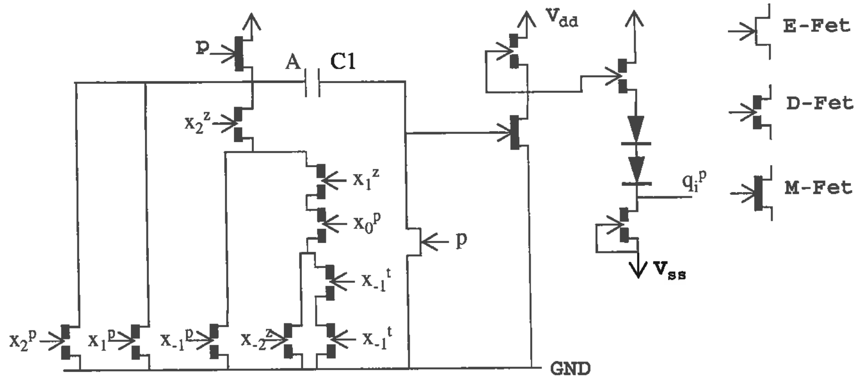

Figure 2.2 SRT Quotient Digit Selection Logic. The gate shown uses GaAs CCDL logic but is typical of domino logic gates used for many self-timed systems.

Two systems were simulated: one using the quotient selection scheme above and one in which $q_i = 0$ was chosen only for $(x_2, x_1, x_0) = (0, 0, 0)$. In both cases, the quotient selection logic was self-timed and the adder evaluated X_{i+1} quickly for $q_i = 0$. The distribution of evaluation times for 40000 divisions is shown in Figure 2.3. The quotient selection scheme of Table 2.1 gives a 4% improvement in average evaluation time.

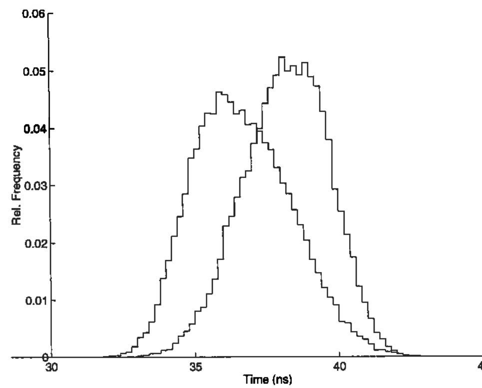

Figure 2.3 Evaluation Times for SRT Division. The diagram shows the distribution of evaluation times for 40000 divisions. The faster distribution uses the enhanced quotient selection scheme of Figure 2.1.

1.5 Summary

The problem of quotient digit selection demonstrates some of the benefits of digit set conversion, as well as some of the challenges. Converting the partial remainder to a non-redundant digit set would maximise the frequency of zero quotient digits; however, the effort required to perform this conversion would negate any benefit of doing so.

Throughout this thesis, digit set conversion is used to improve average execution time for software implementations. The example of the SRT divider demonstrates that these techniques are not only valid for software but also have a place in hardware design.

2 Digit Set Conversion Formalisation and Existing Results

The operation of digit set conversion between positional number representations can be expressed as follows. Let X be a positional *representation* of the integer $\|X\|$ using the digit set $\boldsymbol{X}$. We write:

$$X = (x_{n-1}, x_{n-2}, \ldots, x_0) \in \boldsymbol{X}^n$$

$$\text{with } x_i \in \boldsymbol{X} \qquad \forall 0 \le i < n$$

$$\text{such that } \|X\| = \sum_{i=0}^{n-1} x_i r_x^{\ i}.$$

Some additional terminology and notation will also be useful.

A *digit set conversion* is a function $F: \boldsymbol{X}^* \to \boldsymbol{Y}^*$ such that if $Y = F(X)$ then $\|Y\| = \|X\|$.

A *fixed radix conversion* is a digit set conversion $F: \boldsymbol{X}^* \to \boldsymbol{Y}^*$ with $r_x = r_y = r$.

Denote the set of representations of the integer i in set $\boldsymbol{X}^n$ as $V(\boldsymbol{X}^n(i))$:

$$V(\boldsymbol{X}^n(i)) = \{X \in \boldsymbol{X}^n \text{ s.t. } \|X\| = i\}$$

Denote the number of representations of the integer i in the set $\boldsymbol{X}^n$ as $|V(\boldsymbol{X}^n(i))|$.

Denote the set of all integers with representations in the set $\boldsymbol{X}^n$ as $D(\boldsymbol{X}^n(i)) = \{\|X\| \ \forall \ X \in \boldsymbol{X}^n\}$.

Call a digit set $\boldsymbol{X}^n$ *complete* for $\boldsymbol{D}$ if $\forall i \in \boldsymbol{D} \ \exists \ X \in \boldsymbol{X}^n$ s.t. $\|X\| = i$.

Call a digit set $\boldsymbol{X}^n$ *redundant* if $\exists \ i \in \boldsymbol{Z}$ s.t. $|V(\boldsymbol{X}^n(i))| > 1$.

Call a digit set $\boldsymbol{X}^n$ *non-redundant* if $\forall \ i \in \boldsymbol{Z}$, $|V(\boldsymbol{X}^n(i))| \le 1$.

For example, the familiar digit set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a complete, non-redundant digit set for radix 10 (decimal). If the extra digit $A = 11$ is added then the new digit set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A\}$ is still complete but is redundant: some arithmetic values have more than one representation. As an example, the number 301 can now also be written as 29A.

2.1 Arithmetic Weight

In Section 1 an improvement in average case execution time was achieved by taking advantage of arithmetic simplification possible with the occurrence of zero digits. This kind of average case enhancement is a major theme of this thesis. In subsequent sections, a particular emphasis is placed on digit set conversions that decrease the frequency of non-zero digits.

In a redundant number system there may be more than one representation of a given algebraic value and those representations with the minimum number of non-zero digits are of particular interest.

The number of non-zero digits in a number representation is variously called the Hamming weight, the arithmetic weight or just the weight of that representation. Hamming weight is most often applied to the number of non-zero bits in a binary representation so the term arithmetic weight will be used instead to emphasise that we may be dealing with the number of non-zero digits in a higher radix representation. Let $c(Y)$ be the arithmetic weight of the representation Y thus:

$$c : \mathbf{Y}^n \rightarrow N \text{ such that } c(Y) = \sum_{i=0}^{n-1} \delta(y_i)$$

where $\delta(i) = 1$ if $i \neq 0$ and $d(i) = 0$ otherwise.

Define the *minimum arithmetic weight* of an integer $\|Y\|$ as:

$$m(\|Y\|) = \min\{c(Y_i) \text{ s.t. } Y_i \in \mathbf{Y}^n, \|Y_i\| = \|Y\|\} .$$

Define the *average arithmetic weight* of a digit set conversion $F : \mathbf{X}^n \rightarrow \mathbf{Y}^n$ as:

$$\bar{c}(F) = \sum_{X \in \mathbf{X}^n} \frac{c(F(X))}{|\mathbf{X}^n|}$$

Call a representation Y a *minimal representation* if $c(Y) = m(\|Y\|)$.

Call the digit set conversion $F : \mathbf{X}^n \rightarrow \mathbf{Y}^n$ a *minimal digit set conversion* if for all $Y = F(X)$, Y is a minimal representation of $\|X\|$.

2.2 General Results on Digit Set Conversion

Redundant number representation for computer arithmetic was introduced by Avizienis in [Avi61]. In this paper *signed-digit representation* is defined as a class of redundant digit sets chosen such that: the radix is a positive integer; there is a unique representation of the algebraic value zero; the signed-digit set is complete within a specified range; and carry free addition and subtraction is possible between

two signed digit numbers. Avizienis determined the digit sets that satisfy these criteria for radices greater than 2 and demonstrated carry free addition with the radix-2 digit set $\{-1, 0, 1\}$ (calling this *modified signed-digit representation*). It was found that a fixed radix conversion from non-redundant to signed-digit representation is possible using a carry free addition. Conversion from signed-digit to non-redundant representation requires a carry propagate addition.

Parhami in [Par90] extended the theory of signed-digit representation and defined limited carry addition. This definition has the advantage that radix-2 is covered by the general theory and is no longer a separate case. Binary signed-digit, stored carry and stored borrow representations can all be studied as special cases of Parhami's signed-digit representation.

Carter and Robertson [CR90] also deal with addition using generalised digit sets at arbitrary radix. Their *Set Theory of Arithmetic Decomposition* is a method of describing adders and subtractors using a set notation. For example, $r<D^{W}>$ is the radix r digit set with $(D+1)$ elements, the smallest of which is W. An advantage of this notation is that it makes explicit the both radix and digit set being considered. Unfortunately the notation is limited to contiguous digit sets.

The computational complexity of digit set conversion was considered in a very general way by Kornerup ([Kor94a]) who found that, "For any [fixed radix] conversion from a digit set $\boldsymbol{D}$ to a non-redundant digit set E with $D \neq E$, there exist situations where the most significant digit of the result depends on the least significant digit of the number being converted, hence such conversions must take W(log n) ." Kornerup also concluded that a fixed radix conversion from any digit set with radix greater than 2, to a complete, redundant, contiguous digit set can take place in constant time (that is with bounded carry propagation). This result is extended in [Kor99] in which necessary and sufficient conditions for constant time digit-set conversion (and hence addition) are determined.

2.3 Multiplier and Exponent Recoding

Digit set conversion is a common practice for the implementation of multiplication and exponentiation. Three reasons to change the representation of the multiplier or exponent to a new digit set appear in the literature:

1. to reduce the total number of digits. Conversion to a higher radix will reduce the total number of digits in the multiplier or exponent and in turn reduce the number of iterations required. This is the usual objective of the Booth algorithms described below.
2. to convert between redundant forms. This is employed in systems that take an input in one redundant form and generate the result in another. A carry free conversion is performed to use the result as an input for a subsequent iteration.

3. to reduce the number of non-zero digits. As observed in Section 1, the occurrence of zero digits can accelerate the evaluation of a result. Schemes to reduce the number of non-zero digits are examined in Section 2.6.

2.4 Booth Recoding

Booth's objective in [Boo51] was a multiplication algorithm capable of dealing with positive and negative 2's complement numbers irrespective of their sign. His solution involves a digit set conversion. The multiplier is converted from 2's complement binary to a signed digit set $\{-1, 0, 1\}$. The recoded digits are selected from pairs of adjacent bits in the original multiplier according to Table 2.2.

x_i	x_{i-1}	y_i
0	0	0
0	1	1
1	0	-1
1	1	0

Table 2.2 Booth Recoding. The recoded digits y_0 to y_n are selected according to pairs of digits in the original representation. It is assumed that $x_{-1} = x_n = 0$.

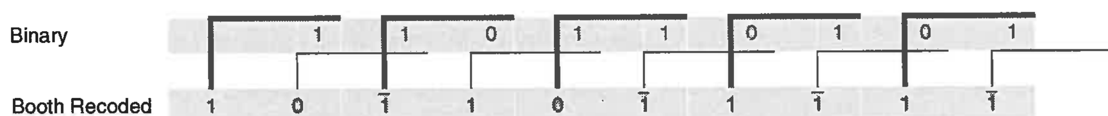

Figure 2.4 Example of Booth Recoding. Pairs of adjacent bits are used to determine the recoded bit. Note that in this example the recoded weight is worse than the original binary weight.

From the example in Figure 2.4 it can be seen that Booth conversion proceeds by considering pairs of adjacent bits, with each pair overlapped by 1 bit. The extension of this process to larger groups was described MacSorley in [MacS61]. His multiplication using *uniform shifts of* 2 uses 3-bit groups overlapped by one bit to convert the multiplier to the digit set $\{-4, -2, 0, 2, 4\}$. This has become one of the most familiar multiplication techniques, elsewhere known as *Modified Booth Recoding* and usually described with the digit set $\{-2, -1, 0, 1, 2\}$ as in Table 2.3. MacSorley also describes uniform shifts of 3 with a target digit set $\{-8, -6, -4, -2, 0, 2, 4, 6, 8\}$.

x_{2i+1}	x_{2i}	x_{2i-1}	y_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2

Table 2.3 Modified Booth Recoding.

x_{2i+1}	x_{2i}	x_{2i-1}	y_i
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 2.3 Modified Booth Recoding.

There are many other examples of Booth conversion with various length groups of which [AEGP67], [Atk70], [Ghe71], [Hwa79], [WF82], [Rod85], and [SD85] are the classical references. [Rub75] proves the correctness of MacSorley's 3-bit scanning.

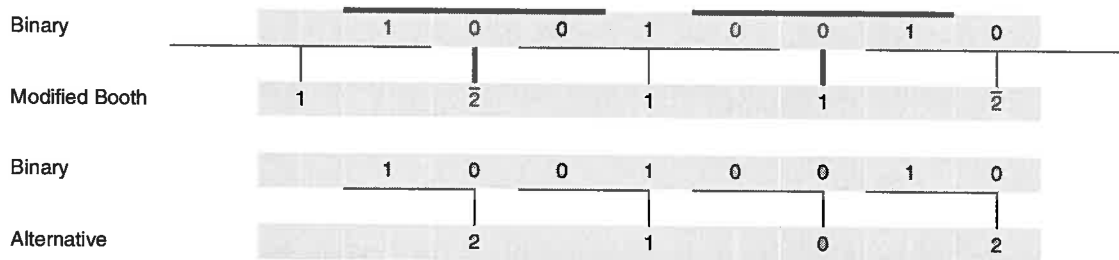

Figure 2.5 Example of Modified Booth Recoding. Groups of three adjacent bits are used to determine the recoded digit. Note that in this example the recoded weight is worse than could be achieved using the alternative recoding with only the unsigned digits {0, 1, 2}.

The authors of [Hwa79], [SD85] and others mention that the Booth technique (s-bit groups overlapped by 1 bit) can be extended to groups of any size. General treatments appear in [VSH89] and [SG90] in which the criteria to be met by all correct *uniform overlapped multiple-bit scanning* techniques or *generalized multibit recoding techniques* are derived.

These schemes have become almost ubiquitous in conventional hardware multiplier designs. The selection of each recoded digit is independent of the others so the entire multiplier conversion can take place in parallel: it is a constant time process. The size of the group scanned determines the number of partial products but the target digit set must be chosen so that the partial products themselves may be rapidly obtained. The remaining delay is due to partial product accumulation and this can be performed in a digit parallel manner using redundant number representation. A final carry propagate adder can be used to convert the sum to a non-redundant form, or alternatively another digit set conversion may be used with the result fed back as the recoded multiplier for the next multiplication (see for instance [LM95]).

2.5 Uniform and Variable Shift Methods

The conventional hardware multiplier described in the previous section has nothing to gain from an improvement in arithmetic weight – any benefit is dependent on the input data and there is no reduction of worst case delay. This is confirmed by Rubinfield in [Rub75]:

> These [multiplier recoding] algorithms may be divided into two categories: variable shift methods and uniform shift methods. The variable shift methods are disadvantageous for clocked systems since the time required for a multiply is data dependent.

Algorithms that take advantage of data dependent delays may not be of use to conventional clocked systems but are of great benefit to software and asynchronous hardware implementations.

The classification as variable and uniform shift methods is proposed in [MacS61] and maintained in [Rub75] and [VSH89]. The variable shift methods are understood to be those that take advantage of zero digits and therefore exhibit data dependent delay. However, this can be misleading as it is possible to always shift the multiplier by a uniform number of digits and still exploit groups of zeros.

Freeman [Fre67] gives an analysis of the mean shift length for a system taking groups of 2, 3 or 4 multiplier bits at a time. This is a very useful paper for two reasons. Firstly, his state table analysis demonstrates a simple and accurate method for evaluating the mean arithmetic weight for recoding schemes and is applied in Section 3.1 and Section 3.2 as well as Appendix B. Secondly, Freeman provides a clear description of what may be in the mind of other authors of the time when they refer to *variable shift methods*:

> Thus if multiplier bits are taken two at a time, the four possible outcomes are 00, 01, 10, and 11, and either 0, 1, 2, or 3 times the multiplicand is to be added to the partial product respectively. The first three of these can be handled directly by a combination of shifting and adding the multiplicand, as appropriate, to the existing partial product. However, the fourth combination, 11, requires that one either precalculate the triple of the multiplicand and add it to the partial product when this bit combination occurs, or that one merely take the first bit of the pair, shift only once, and then let the second bit become the first bit of the next combination. In the latter case, the *mean shift* (which clearly must lie somewhere between 1 and 2) can be obtained...

2.6 Reducing Arithmetic Weight

The variable shift method described by Freeman takes advantage of zero bits in the binary representation of a number. This can be taken a step further: one can actively seek to increase the number of zero digits in a representation using a digit set conversion.

An interesting starting point for representations with reduced arithmetic weight is presented in [Bri83]. Consider a binary representation X. If $c(X) \leq n/2$ then choose $Y = X$; otherwise choose $Y = 2^n - (2^n - X)$. To do this invert all the bits in X and then add 1, multiply each bit by -1 and finally append a 1 to the left of the leftmost bit. As a consequence we must have $c(Y) \leq \lceil n/2 \rceil + 1$.

2.7 Booth Conversion to Reduce Arithmetic Weight

Booth conversion can reduce the arithmetic weight of a representation [Ghe71]. This is due to the frequently cited observation that Booth conversion will replace strings of 1's by a string of zeros thus: $01111_2 = 1000\bar{1}_2$. However, Booth's is not a minimal recoding and this is evident from the example in Figure 2.4.

Consider the 2-bit scanning in Table 2.2 on page 18. For large n, as each row of the table is equally likely, $y_i \neq 0$ is chosen half of the time. Half of the time an extra bit is required for $y_n = 1$. Therefore the average arithmetic weight is $(n+1)/2$. Despite the elimination of strings of 1's, there is no improvement over non-redundant binary. However, Modified Booth conversion using 3-bit scanning converts 2-bits to a non-zero digit three quarters of the time. For even n an extra digit is required half the time. Thus time an average arithmetic weight of $(3n+4)/8$ is expected. Again Figure 2.5 demonstrates that the conversion is not minimal.

Booth's original 2-bit scanning will convert 0111_2 to $100\bar{1}_2$ but will also convert 0101_2 to $1\bar{1}1\bar{1}_2$. Modified Booth with 3-bit scanning fails to be minimal when confronted with sequences of three bits 001 as in Figure 2.4. By increasing the number of bits scanned, it is possible to improve the outcome. This is the idea behind the *Recoded Binary Method* of [Koç90] in which the signed bit y_i is determined by the four bits x_{i+1}, x_i, x_{i-1} and x_{i-2}. The resultant binary representation Y has an average arithmetic weight of $3n/8$ for large n. Unfortunately this is no improvement over the 3-bit scanning above.

Having performed a binary conversion to improve arithmetic weight, groups of adjacent bits can be combined to form higher radix digits. This leads to the *Radix-4 String Recoding* mentioned in [Hwa79] or the *Recoded M-ary Method* studied in [Koç90].

The *Recoded M-ary Method*, starts with the *Recoded Binary Method* to improve arithmetic weight. Then m-bit digits are then formed on regular m-bit boundaries. While the probability of a single bit being zero is $5/8$, the probability of finding m adjacent zero bits is not $(5/8)^m$ as claimed in [Koç90] but rather $5/(2^{m+2})$. This is because adjacent bits are not selected independently by the overlapped recoding. Thus, the approximate average arithmetic weight of n digits is not $n(1-(5/8)^m)$ but $n(1-5/(2^{m+2}))$.

The analysis in [Koç90] underestimates the average arithmetic weight but overestimates the resultant digit set. Following the binary conversion, not all strings of signed bits can actually appear. Hence the correct higher radix digit set is:

$$\{-2^{m-1}, \ldots, -2, -1, 0, \ldots, 1, 2, 2^{m-2} + 2^{m-1}\}$$

The analysis of this method is outlined in Appendix B.

2.8 Minimal Binary Conversion

Redundant binary representation with the digit set $\{-1, 0, 1\}$, sometimes called *modified signed digit* representation (and often just called *signed digit* representation), does not exhibit a unique minimal form. Algorithms to generate a minimal representation are widely reported for both multiplication (see for instance [MacS61], [Hwa79] and [KH92]) and exponentiation (including [JM89] and [Zha93]).

A particular representation was studied in [Rei60] in which it was proved that the representation with no two adjacent digits being both non-zero was both minimal and unique for a given algebraic value. This representation is variously called the canonic, sparse or nonadjacent form. An example of its generation is given in Figure 2.6.

Significance	7	6	5	4	3	2	1	0
$A =$	1	0	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	1	1

There are adjacent non-zero bits at significance 0 and 1.
Subtract 2 from bit 0 and compensate by adding 1 into bit 1.

$A =$	1	0	$\bar{1}$	1	$\bar{1}$	0	0	$\bar{1}$

There are adjacent non-zero bits at significance 3 and 4.
Add 2 to bit 3 and compensate by subtracting 1 from bit 4.

$A =$	1	0	$\bar{1}$	0	1	0	0	$\bar{1}$

There are no adjacent non-zero bits. The representation is now in canonic form.

Figure 2.6 An Example of Binary Canonic Conversion. The representation A is scanned from the right. Whenever two adjacent non-zero bits are found, the less significant is set to zero and a carry or borrow is propagated to the left to account for the change.

In the previous section Booth recoded binary representations were converted to higher radix by grouping bits. The same technique can be applied to a canonic binary representation: a number in canonic binary form is converted to radix $r = 2^m$ by forming m-bit groups from adjacent bits as in Figure 2.7. A specific instance of this appears in [Hwa79] in which the canonic binary form is

converted to radix-4 by forming digits from adjacent pairs of bits. The resultant representation has digits from the set $\{-2, -1, 0, 1, 2\}$. In general the target digit set is:

$$\{-(-2^{m+2}-(-1)^{m}-3)/6, \ldots, -1, 0, 1, \ldots, (-2^{m+2}-(-1)^{m}-3)/6\}.$$

For a large number of digits n, the average arithmetic weight is approximately $n(1-2^{2-m}/3)$. A summarised derivation of these results appears in Appendix B.

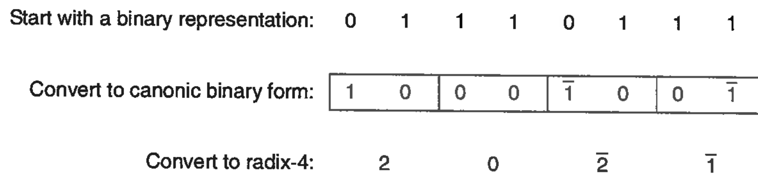

Figure 2.7 Hwang's Conversion. The binary number *A* is converted to canonic signed digit form. It is then converted to radix-4 by grouping adjacent bits.

2.9 Minimal Higher Radix Conversion

In [CL73] the authors seek to extend the concept of the canonic binary form (or the 'well known modified binary non-adjacent form' as they call it) to higher radices. They define a minimal form called the generalised non-adjacent form (GNAF) using the digit set $\{-r+1, \ldots, r-1\}$. The minimal representation of a number $\|Y\|$ using this digit set is not unique but there is only one representation, Y, which satisfies the extra conditions:

$$\text{if } y_i y_{i+1} < 0 \text{ then } |y_i| < |y_{i+1}|$$

$$|y_i + y_{i+1}| < r.$$

The authors provide an algorithm to convert from a representation using digits $\{-r+1, \ldots, r-1\}$ to the corresponding GNAF. This conversion involves propagation of information from the rightmost digit to the left.

The authors of [AW93] seek their own canonic form for the digit set $\{-r+1, \ldots, r-1\}$. They find another conversion algorithm (again propagating information to the left) and then use Markov Chain analysis to show that the expected value of the minimal arithmetic weight is $n(r-1)/(r+1)$ for large n. They then use combinatorial techniques to find a probability distribution of minimum arithmetic weights.

2.10 Sliding Window Algorithms

A conversion to the set of odd digits $Y = \{0, 1, 3, \ldots, 2^m - 1\}$ is implicit in the exponentiation scheme of [CL87]. To convert from binary, non overlapping groups of m-bits are considered. Each group forms a digit y_i and an offset z_i such that if the original group had a value x_i then $x_i = y_i 2^{z_i}$. Figure 2.8 demonstrates the process.

Binary Form	1 0 0 1	0 0 1 0	1 1 0 0	0 1 1 0
After Recoding	$y_3 = 9, z_3 = 0$	$y_2 = 1, z_2 = 1$	$y_1 = 3, z_1 = 2$	$y_0 = 3, z_0 = 1$

Figure 2.8 Cohen's Window Conversion. Groups of 4-bits form an odd digit y_i and an offset z_i.

It can be seen that this method forms digits from groups of adjacent bits or *windows*. The digits are separated by a strings of consecutive zeros. This method does not take advantage of strings of zeros that do not appear on m-bit boundaries. A more flexible method is demonstrated in [BC90] where the authors call it a window method and state that it is derived from [Knu69]. Similar conversions are presented in [Yac91], [KH92], [HL94] and [GHM96]. The conversion of [Yac91] groups bits in a right to left process; others use an analogous left to right process.

The right to left process to convert from the digit set $X = \{0, 1\}$ to $= \{0, 1, 3, \ldots, 2^m - 1\}$ can be simply expressed. Starting with the least significant bit, x_0, skip over all bits equal to 0 until a bit equal to 1 is found. This bit and the following m bits form the odd digit y_0. The process then returns to skipping zeros until another digit, y_i, is found and so on. Figure 2.9 shows an example.

Binary Form *X*	0	1	1	0	0	0	1	1	1	1	0	0	1	0	1
After Recoding *Y*	0	0	3	0	0	0	1	0	0	7	0	0	0	0	5

Figure 2.9 An Example of Sliding Window Conversion. The binary form is converted by grouping 3-bit windows, starting from the right and progressing to the left.

Hui and Lam call their conversion to odd m-bit digits SS(m). Koç and Hung call theirs an *adaptive m-ary segmentation*. In [LH94] it is shown to be a minimal conversion. The average arithmetic weight is found to be approximately $n/(m+1)$ with the approximation getting better for large n.

Gollmann, Han and Mitchell study the *string replacement algorithm k-SR* in which binary numbers are converted to a representation using the odd digits less than or equal to k. They define a *canonical k-SR* form and derive the average weight for this conversion (Equation 2.3), observing however, that

the canonical form is not always minimal. Note that SS(m) is a special case of *k-SR* for $k = 2^{m-1}$ and will always generate a minimal representation.

$$\bar{c}(k\text{-}SR) = \frac{n2^{n+k-2}}{2^k - 1} + \frac{(2^k - k - 1)\,2^{n+k-2} + (k - s - 1)\,2^{k+s-2} + (s+1)\,2^{s-2}}{(2^k - 1)^2}$$

$$\text{where } s \equiv n \ (\mathrm{mod}\ k) \text{ and } \le s \le k \tag{2.3}$$

In [KH92] sliding windows and canonical binary recoding are combined in the *adaptive m-ary segmentation canonical recoding*. A representation is first converted to binary canonic form and a sliding window is then used to group adjacent non-zero digits into odd digits. The digit set is the set of all odd digits with an m-bit canonical representation. This is the set of $(2/3)\,(2^m + (-1)^{m+1})$ odd digits $\pm\{1, 3, \ldots, (2/3)\,(2^m + (-1)^{m+1}) - 1\}$. The recoding achieves an average arithmetic weight for large of n of $3n/(3m+4)$. This result will be compared with other signed sliding window algorithms at the end of this chapter.

2.11 Mixed Radix Algorithms

There are similarities between the sliding window algorithms above and the hybrid number systems of [DC95] and [CCY96]. However, the former represent a number with redundant digits from a single radix whereas the hybrid methods use two radices, selecting zero digits from the higher radix whenever possible and non-zero digits from the lower radix otherwise.

A drawback with these systems is that the conversion requires repeated divisions by the odd numbers 3 or 5. The binary-ternary scheme [DC95] has an average arithmetic weight of $0.3381n$ where n is the length of the number to be converted in bits. The target digit set is $\{0, 1\}$. The ternary-quinary scheme [CCY96] has an average weight of $0.324n$ from a digit set of $\{0, 1, 2\}$. These can be compared with SS(2) from above that has an average weight of $0.333n$ using the digits $\{0, 1, 3\}$ and with the advantage of trivial conversion from binary.

This concludes the survey of existing digit-set conversions. The following section presents some new, generalised sliding window conversions and derives a number of results for these conversions.

3 Generalised Sliding Windows

Let us consider, in very general terms, the complexity of minimal digit set conversion. According to Kornerup a fixed radix conversion from any digit set with radix greater than 2, to a complete,

redundant, contiguous digit set can take place in constant time. Does this still hold if the conversion must also be minimal?

Figure 2.10 examines a fixed radix conversion to a complete, redundant but non-contiguous digit set. Consider conversion of the least significant digit. For a correct conversion, $\|Y\| \bmod r = \|X\| \bmod r$ and thus we must have $y_0 \bmod r = x_0 \bmod r$. This means that there are only two possible choices in Y for y_0 and Figure 2.10 traces the implications of each decision. Note that the optimal choice of y_0 depends upon the values of an arbitrary number of digits to the left. Similarly, an optimal choice of y_m cannot be made without examination of all of the digits to the right.

$$r = 8, X = \{0, 1, 2, 3, 4, 5, 6, 7\}, Y = \{\bar{6}, 0, 1, 2, 3, 4, 5, 6, 7\}$$

Consider $X = (2, 7, 7, 7, \ldots, 7, 2, 7, 7, 7, \ldots, 7, 2)$.

We can select $y_0 = \bar{6}$ in which case we must have $Y = (2, 7, 7, 7, \ldots, 7, 3, 0, 0, 0, \ldots, 0, \bar{6})$

or

we can select $y_0 = 2$ in which case we have $Y = (3, 0, 0, 0, \ldots, 0, \bar{6}, 7, 7, 7, \ldots, 7, 2)$

Figure 2.10 A Difficult Digit Set Conversion. The optimal choice of the least significant digit depends on which string of 7's is longer. This decision can only be based on knowledge of all the digits rather than just a finite subset.

The problem in Figure 2.10 arises because of carry propagation due to the introduction of a negative digit. A similar situation can be constructed using positive digits and borrow propagation.

$$r = 8, X = \{0, 1, 2, 3, 4, 5, 6, 7\}, Y = \{0, 1, 2, 3, 4, 5, 6, 7, 77\}$$

Consider $X = (1, 6, 1, 6, 1, 6, 0, \ldots, 0, 1, 5)$.

We can select $Y = X$

or

we can select $y_0 = 77$ in which case we have $Y = (1, 6, 1, 6, 1, 5, 7, \ldots, 7, 0, 77)$

and hence can generate $Y = (\ldots, 0, 77, 0, 77, 0, 77, 7, \ldots, 7, 0, 77)$

Figure 2.11 Another Difficult Digit Set Conversion. The optimal choice of the least significant digit depends on which is longer: the initial string of 0's or the recurring string of 1's and 6's.

A minimal representation in these digit sets can always be found using a 'brute force' approach, in which the complete set of possible representations is enumerated. Beginning at the least significant digit, x_0, one can record each of the possible values for y_0. For each value of y_0 there will be a set of possible values for y_1 and proceeding in this manner it is possible to find the set of representations $V(Y^n(\|X\|))$. If there is at most d possible values for each digit, then enumerating all representations is a process of complexity $O(d^n)$.

The awkwardness of these two examples is due to their non-contiguous digit sets. However, as discussed in the following sections, there are some useful non-contiguous digit sets for which less difficult minimal conversions can be found.

3.1 Unsigned Sliding Window Algorithm

The sliding window algorithms of Section 2.10 can be seen as specific instances of a more general family of sliding window digit set conversions that use only unsigned digits. Extending the SS(m) notation of [HL94], I will define a family of unsigned digit set conversions $USW_{r,m}$.

The digit set conversion $USW_{r,m}:X^n \to Y^n$ is a fixed radix-r conversion from digit set $X = \{0, 1, 2, \ldots, r-1\}$ to the set of digits $Y = \{y \text{ s.t. } 0 < y < r^m, y \neq 0 \bmod r\} \cup \{0\}$.

The pseudo-code of Algorithm 2.2 performs the conversion $Y = USW_{r,m}(X)$ such that $\|X\| = \|Y\|$.

Algorithm 2.2 USW in Pseudo Code. Performs the digit set conversion Y = $USW_{r,m}$ (X).

```
s = 0
for i = 0 to n - 1
    if s = 0 then
        if xi = 0 then
            yi = 0
        else
            yi = Σ_{j=0}^{m-1} x_{i+j} × r^j
            s = m - 1
        end if
    else
        yi = 0
        s = s - 1
    end if
next i
```

$USW_{r,m}$ can be understood as a higher radix expression of the binary sliding window algorithms of Section 2.10 with the original binary algorithms being the case $USW_{2,m}$. $USW_{r,m}$ starts with the least significant digit, x_0, and skips over zeros until a non-zero digit, x_i, is found. This and the following $(m-1)$ digits form the new digit y_i. The process returns to skipping zeros until another non-zero x_i is found and so on. Figure 2.12 shows an example.

Radix 4 Form X	3	1	0	0	0	3	0	0	3	0	2	2	1	0	2
After Recoding Y	0	13	0	0	0	3	0	0	3	0	2	0	9	0	2

Figure 2.12 An Example of USW Conversion. A radix-4 number is recoded according to $USW_{4,2}$.

Complexity of USW

In Figure 2.2 there is one iteration for each digit in X. From this it is clear that the computational complexity of USW is $O(\log\|X\|)$.

Minimality of USW

The minimality of $USW_{r,m}$ follows by induction on the optimal arithmetic weight $m(\|X\|)$.

Theorem: for all $X \in \mathbf{X}^n$ we have $c(USW_{r,m}(X)) = m(\|X\|)$.

Initial Case: $c(USW_{r,m}(X)) = 1 \Rightarrow m(\|X\|) = 1$.

This is trivially true. If $USW_{r,m}$ has one non-zero digit then $\|X\| > 0$ and all representations must have at least one non-zero digit.

Induction Case: If the hypothesis is true for all $c(USW_{r,m}(X)) < k$ then it is also true for $c(USW_{r,m}(X)) = k$.

Assume the contrary: that for some X with $Y = USW_{r,m}(X)$ and $c(Y) = k$ there exists another representation $Z \in \mathbf{Y}^n$ such that $\|Z\| = \|Y\|$ and $c(Z) < k$; however, for all X with $c(USW_{r,m}(X)) < k$ we have $c(USW_{r,m}(X)) = m(\|X\|)$.

Consider the following cases:

1. $y_0 = z_0 = 0$.

   Let $Y' = (0, y_{n-1}, \ldots, y_2, y_1) = R(Y)$ that is Y, shifted right by 1 digit. Let $Z' = R(Z)$.

   Now $\|Y'\| = \|Z'\|$, $c(Y') = c(Y)$ and $c(Z') = c(Z)$ so without loss of generality let us test the induction case on Y' and Z' instead.

2. $z_0 = 0$, $y_0 \neq 0$

   From $\|Y\| = \|Z\|$ we must have $\|Y\| \bmod r = \|Z\| \bmod r$.

   $\|Y\| \bmod r = y_0 \bmod r \neq 0$ for $y_0 \neq 0$ and $USW_{r,m}$.

   $\|Z\| \bmod r = z_0 \bmod r = 0$.

So $\|Y\| \bmod r \neq \|Z\| \bmod r$ and this contradicts the requirement that $\|Z\| = \|Y\|$.

3. $y_0 = z_0 \neq 0$.

Let $Y' = R(Y)$ and $Z' = R(Z)$ where R is the right shift function defined in case 1.

Now $\|Y'\| = \|Z'\|$, $c(Y') = k - 1$ and $c(Z') = c(Z) - 1 < k$

so $c(Z') < c(Y')$.

But Y' is an output of $USW_{r,m}$ with arithmetic weight less than k. Therefore, according to the induction hypothesis, Y' should be a minimal representation of $\|Y\|$. That Z' is better than minimal is a contradiction.

4. $y_0 \neq z_0$, $z_0 \neq 0$.

This case is resolved using a process that transforms Z into some Z' such that $\|Z\| = \|Z'\|$, $c(Z') \leq c(Z)$ and $z'_0 = y_0$. It is then possible to refer to case 1 or case 2 for a contradiction.

We must have $\|Y\| \bmod r^m = \|Z\| \bmod r^m$

$$\Rightarrow \sum_{i=0}^{m-1} z_i \times r^i \bmod r^m = y_0 \bmod r^m$$ because for $USW_{r,m}$ we have $y_i = 0$ for $< i < m$.

$$\Rightarrow z_0 - y_0 = ar^m - z_{m-1}r^{m-1} - z_{m-2}r^{m-2} \ldots - z_1 r$$ for some integer a.

Algorithm 2.3 shows the transformation from Z to Z'. This proceeds by examining z_i starting with $i = 1$. If this is not zero then $z_0 - y_0$ is a multiple of r^i. If $z_0 > y_0$ subtract r^i from z_0 and if $z_0 < y_0$ then add r^i to z_0. Keep adding or subtracting until $z_i = 0$.

To subtract r^i from z_0 and maintain $\|Z\| = \|Z'\|$ we must add 1 to z_i. If z_i exceeds $r^m - 1$ it is corrected modulo r^m and a carry is propagated into subsequent digits. In the original representation Z, $z_i \neq 0$ so this carry propagation can not increase c (Z) - it either creates more zero digits or leaves the total unchanged.

To add r^i to z_0 and maintain $\|Z\| = \|Z'\|$ we must subtract 1 from z_i. In this case there is no issue with borrow propagation: we stop when z_i falls to zero. Again, the number of zero digits in Z can only have increased.

When the transformation terminates, $z'_i = 0$ for $0 < i < m$ and hence $z'_0 - y_0 = ar^m$. Also $0 \le z'_0 \le r^m - 1$ and $0 \le y_0 \le r^m - 1$ so $-(r^m - 1) \le ar^m \le r^m - 1$ and hence $a = 0$ and $z'_0 = y_0$ which was the object of the exercise.

Algorithm 2.3 Transformation from Z to Z'.

```
for i = 1 to m - 1
    while z_i <> 0
        if z_0 > y_0 then
            z_0 = z_0 - r^i
            j = i
            /* carry propagate */
            do
                z_j = (z_j + 1) mod r^m
                j = j + m
            while z_{j-m} = 0
        else if z_0 < y_0 then
            z_0 = z_0 + r^i
            z_i = z_i - 1
        end if
    end while
next i
```

All four cases lead to a contradiction of the assumption that there exists a representation with an arithmetic weight less than that produced by $\mathrm{USW}_{r,m}$. Hence the induction hypothesis—that $\mathrm{USW}_{r,m}$ is minimal—holds. The sliding window conversion $\mathrm{USW}_{r,m}$ always produces a minimal representation for its target digit set. ■

Average Arithmetic Weight of USW

The average arithmetic weight of $\mathrm{USW}_{r,m}$ can be determined by Markov analysis of the state diagram in Figure 2.13. Each state represents the selection of a single digit.

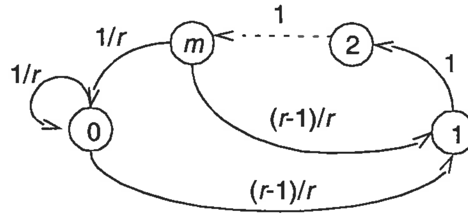

Figure 2.13 State Diagram for USW$_{r,m}$. State 0 corresponds to the selection of a zero such that the next digit may be non-zero. State 1 selects a non-zero digit. States 2 to m represent the selection of the $m-1$ zeros that must follow a non-zero digit.

Define $p_i(k)$ as the probability of being in state i after k digits. From Figure 2.13 we can write:

$$p_0(k+1) = \frac{1}{r}p_0(k) + \frac{1}{r}p_m(k)$$

$$p_1(k+1) = \frac{r-1}{r}p_0(k) + \frac{r-1}{r}p_m(k)$$

$$p_i(k+1) = p_{i-1}(k) \text{ for } 2 \le i \le m.$$

These equations can be combined into a single matrix equation:

$$P(k+1) = QP(k) \tag{2.4}$$

$$\text{where } Q = \begin{bmatrix} \frac{1}{r} & 0 & 0 & 0 & \dots & 0 & \frac{1}{r} \\ \frac{r-1}{r} & 0 & 0 & 0 & \dots & 0 & \frac{r-1}{r} \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix}.$$

Taking the Z-transform of Equation 2.4 and rearranging yields:

$$\boldsymbol{P}(z) = [zI - Q]^{-1} zP(0) \tag{2.5}$$

where $\boldsymbol{P}(z)$ is the Z-transform of $P(k)$.

Equation 2.5 can be solved using row-reduction to perform the matrix inversion. For $m \geq 1$ it is found that:

$$p_1(z) = \frac{z^{m-1}(r-1)}{z^m r - z^{m-1} - r + 1} \tag{2.6}$$

As $k \to \infty$ the frequency of state i approaches $kp_i(k)$ (from the strong law of large numbers for Markov chains [AW93]).

The final value theorem for Z-transforms asserts that $\lim_{k \to \infty} p(k) = \lim_{z \to 1} (z-1)p(z)$ and thus:

$$\lim_{k \to \infty} p_1(k) = \frac{1}{m + 1/(r-1)} . \tag{2.7}$$

There is a state transition for each of the n digits and the frequency of state 1 corresponds to the frequency of non zero digits. So the average arithmetic weight for large n is:

$$\bar{c}(USW_{r,m}) \sim \frac{n}{m + 1/(r-1)} \tag{2.8}$$

Cardinality of the USW Digit Set

The digit set Y must contain a digit for every window that can be formed with m digits from $X = \{0, 1, 2, \ldots, r-1\}$. The rightmost digit in a window must be non-zero and can therefore be one of $r-1$ possible values. The remaining digits in the window can take any of the r digit values. The number of different windows is therefore $r^{m-1}(r-1) = r^m - r^{m-1}$.

In addition, Y must also contain the zero digit. Therefore:

$$|Y| = r^m - r^{m-1} + 1 . \tag{2.9}$$

Worst Case Arithmetic Weight for USW

The worst case arithmetic weight for $USW_{r,m}$ occurs when every $(m+1)$-th digit in the original representation is non-zero as in Figure 2.14. The worst case arithmetic weight is therefore $\lceil n/m \rceil$.

Binary Form X	0	0	1	0	0	1	1	1	1	1	0	1	1	0	1
After Recoding Y	0	0	1	0	0	1	0	0	7	0	0	5	0	0	5

Figure 2.14 An Example of the Worst Case for USW. A worst-case situation is shown for $USW_{2,3}$. Every 3rd bit in the original representation is non-zero.

Arithmetic Weight Distribution of USW

This section considers the distribution of the arithmetic weights of representations resulting from $USW_{r,m}$ conversion.

Let us define $d(k) = |\boldsymbol{K}|$ where $\boldsymbol{K} = \{X \in \boldsymbol{X}^n \text{ s.t. } c(USW_{r,m}(X)) = k\}$. That is, $d(k)$ is the number of representations that have an arithmetic weight of k following $USW_{r,m}$ conversion.

To evaluate $d(k)$ count the number of representations in $\boldsymbol{X}$ that have k windows. The leftmost window requires special attention as it may contain fewer than m digits (if it begins to the right of x_{n-m}). Consider the following cases:

1. There are k windows of exactly m digits each.

   In each representation in $\boldsymbol{X}$ there are k windows interspersed with $n - km$ zero digits not grouped into a window. How many ways can these windows and zeros be arranged?

   This is the number of permutations two classes of things, taken all at a time. Hence:

$$\text{the number of window permutations} = \frac{(n-km+k)!}{k!\,(n-km)!} \,. \tag{2.10}$$

   For each window permutation, how many different ways can vales be assigned to the windows? There are $(r^m - r^{m-1})$ different window values and k windows so:

$$\text{the number of value permutations} = (r^m - r^{m-1})^k \,. \tag{2.11}$$

   The product of Equation 2.10 and Equation 2.11 is the number of representations of the form we seek.

2. There are $k-1$ windows of m digits and 1 window (the leftmost) of $m-i$ digits.

   Not including the leftmost window, there are $k-1$ windows interspersed with $n-km+i$ zero digits. For these windows:

$$\text{the number of window permutations} = \frac{(n-km+i+k-1)!}{(k-1)!\,(n-km+i)!} \tag{2.12}$$

$$\text{and the number of value permutations} = (r^m - r^{m-1})^{k-1} \,. \tag{2.13}$$

   In addition, for every arrangement of the windows to the right, the number of different values the leftmost window can take on is:

$$r^{m-i} - r^{m-i-1} \,. \tag{2.14}$$

The product of Equation 2.12, Equation 2.13 and Equation 2.14 is the number of representations of the form we seek.

The total number of representations that lead to an arithmetic weight of k is the number found in case 1 plus those found in case 2 for all values of i from 1 to $m-1$. The final result is shown in Equation 2.15 which has been re-arranged so that it is valid for all values of k from 0 to $\lceil n/m \rceil$.

$$\mathrm{d}(k) = \frac{(r^m - r^{m-1})^k}{k!}\left(\prod_{i=1}^{k}(n-km+i) + \frac{k}{(r^m - r^{m-1})}\sum_{i=1}^{m-1}(r^{m-i} - r^{m-i-1})\prod_{j=1}^{k-1}(n+i-km+j)\right) \tag{2.15}$$

Note that given Equation 2.15 it is possible to determine the exact average arithmetic weight for particular values of r, m and n from:

$$\bar{c}(\mathrm{USW}_{r,m}) = \frac{\sum_k (\mathrm{d}(k) \times k)}{r^n}.$$

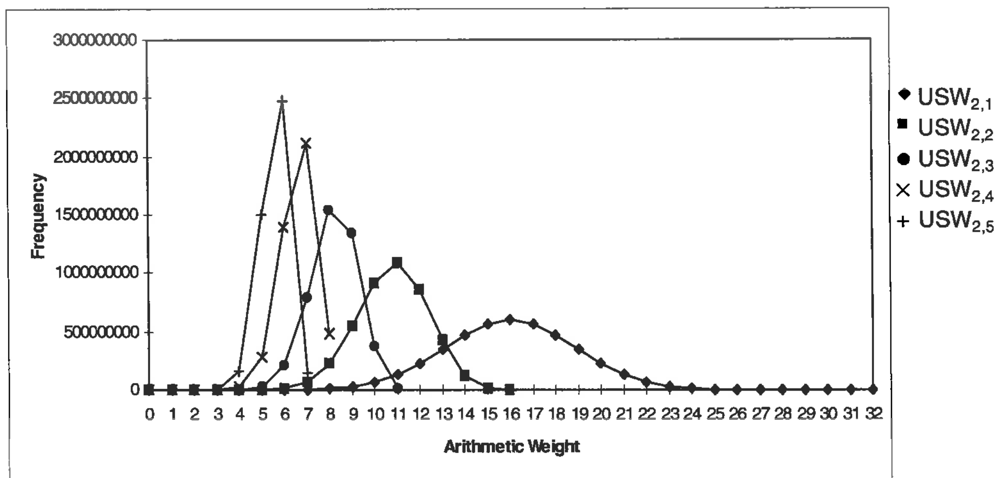

Figure 2.15 Weight Distributions for USW. The distributions are obtained using Equation 2.15 with $r = 2$ and $n = 32$.

We have now proved the minimality of and derived its average arithmetic weight, cardinality and weight distribution. In the following section we turn our attention to sliding window conversions in which the target digit set contains both negative and positive digits.

3.2 Signed Sliding Window Algorithm

A new minimal digit set conversion that uses both positive and negative digits is developed in this section. The new conversion will be called $SSW_{r,m}: X^n \to Y^n$ and is a fixed radix-r conversion from digit set $X = \{0, 1, 2, \ldots, r-1\}$ to the set of digits:

$$Y = \{y \text{ s.t. } -r^m < y < r^m, y \neq 0 \bmod r\} \cup \{0\} .$$

The pseudo-code of Algorithm 2.4 performs the conversion $Y = SSW_{r,m}(X)$ such that $\|Y\| = \|X\|$.

Algorithm 2.4 SSW in Pseudo Code. Performs the digit set conversion Y = $SSW_{r,m}$ (X).

```
s = 0, c = 0
x_n = 0, x_n+1 = 0, ... , x_n+m = 0
for i = 0 to n - 1
    if s = 0 then
        if (x_i + c) mod r = 0 then
            y_i = 0
        else
            y_i = c + sum_{j=0}^{m-1} x_{i+j} × r^j
            c = 0
            if x_i+m = r - 1 then
                y_i = y_i - r^m
                c = 1
            end if
            s = m - 1
        end if
    else
        y_i = 0
        s = s - 1
    end if
next i
y_n = c
```

Starting at the right, zeros are skipped until a non-zero digit, x_i, is found; this and the following $m-1$ digits form the digit y_i. Up until this point, $SSW_{r,m}$ is identical to $USW_{r,m}$; however, SSW

now checks the digit x_{i+m}. If $x_{i+m} = r-1$ then by setting y_i negative, a carry is generated that will set x_{i+m} and possibly subsequent digits to zero. Figure 2.16 shows an example.

Note that SSW may generate a carry out that will require one extra digit (or at least one extra bit) to store.

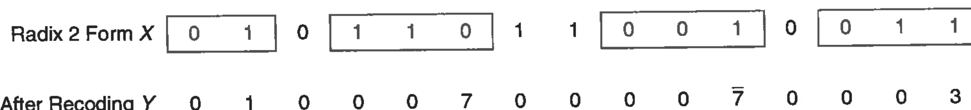

Figure 2.16 An Example of SSW Conversion. $SSW_{2,3}$ is applied to a binary representation.

Complexity of SSW

There is one iteration of the SSW algorithm for each digit of X. From this it is clear that the computational complexity of SSW is $O(\log\|X\|)$.

Minimality of SSW

The minimality of $SSW_{r,m}$ follows by induction on the optimal arithmetic weight $m(\|X\|)$.

Theorem: for all $X \in \mathbf{X}^n$ we have $c(SSW_{r,m}(X)) = m(\|X\|)$.

Initial Case: $c(SSW_{r,m}(X)) = 1 \Rightarrow m(\|X\|) = 1$.

This is trivially true. If $SSW_{r,m}$ has one non-zero digit then $\|X\| > 0$ and all representations must have at least one non-zero digit.

Induction Case: If the hypothesis is true for all $c(SSW_{r,m}(X)) < k$ then it is also true for $c(SSW_{r,m}(X)) = k$.

Assume the contrary: that for some X with $Y = SSW_{r,m}(X)$ and $c(Y) = k$ there exists another representation $Z \in \mathbf{Y}^n$ such that $\|Z\| = \|Y\|$ and $c(Z) < k$; however, for all X with $c(SSW_{r,m}(X)) < k$ we have $c(SSW_{r,m}(X)) = m(\|X\|)$.

Consider the following cases:

1. $y_0 = z_0 = 0$.

   Let $Y' = (0, y_{n-1}, \ldots, y_2, y_1) = R(Y)$ that is Y, shifted right by 1 digit. Let $Z' = R(Z)$.

   Now $\|Y'\| = \|Z'\|$, $c(Y') = c(Y)$ and $c(Z') = c(Z)$ so without loss of generality let us test the induction case on Y' and Z' instead.

2. $z_0 = 0$, $y_0 \neq 0$

From $\|Y\| = \|Z\|$ we must have $\|Y\| \bmod r = \|Z\| \bmod r$.

$\|Y\| \bmod r = y_0 \bmod r \neq 0$ for $y_0 \neq 0$ and $\mathrm{SSW}_{r,m}$.

$\|Z\| \bmod r = z_0 \bmod r = 0$.

So $\|Y\| \bmod r \neq \|Z\| \bmod r$ and this contradicts the requirement that $\|Z\| = \|Y\|$.

3. $y_0 = z_0 \neq 0$.

Let $Y' = \mathrm{R}(Y)$ and $Z' = \mathrm{R}(Z)$ where R is the right shift function defined in case 1.

Now $\|Y'\| = \|Z'\|$, $c(Y') = k-1$ and $c(Z') = c(Z) - 1 < k$

so $c(Z') < c(Y')$.

But Y' is an output of $\mathrm{SSW}_{r,m}$ with arithmetic weight less than k. Therefore, according to the induction hypothesis, Y' should be a minimal representation of $\|Y'\|$. That Z' is better than minimal is a contradiction.

4. $z_0 = y_0 - r^m$, $(z_1, \ldots, z_{m-1}) = (0, \ldots, 0)$.

From $\|Y\| \bmod r^{m+1} = \|Z\| \bmod r^{m+1}$ we have $z_m r^m + z_0 = y_m r^m + y_0 + a r^{m+1}$ for some integer a. Substituting $z_0 = y_0 - r^m$ gives

$$z_m = y_m + ar + 1 . \tag{2.16}$$

Now for $z_0 \in Y$ we must have $-r^m + 1 \leq z_0 = y_0 - r^m$ and hence $y_0 \geq 0$.

In converting x_0 to y_0 with SSW y_0 has not been set negative and we can therefore conclude that $x_m \neq r-1$. Proceeding with SSW, we generate y_m from x_m by adding multiples of r. We must have $y_m \neq -1 \bmod r$.

We can now return to Equation 2.16 and conclude that $z_m \neq 0$.

This means that it is possible to set $z'_0 = z_0 + r^m = y_0$ and $z'_m = z_m - 1 \bmod r^m$ and propagate any borrow through z'_{2m}, z'_{3m} and so on. This process can only reduce the arithmetic weight of Z' or leave it unchanged.

Finally $\|Z'\| = \|Z\|$, $c(Z') \leq c(Z)$ and $z'_0 = y_0$ so case 3 provides a contradiction.

5. $z_0 = y_0 + r^m$, $(z_1, \ldots, z_{m-1}) = (0, \ldots, 0)$.

From $\|Y\| \bmod r^{m+1} = \|Z\| \bmod r^{m+1}$ we have $z_m r^m + z_0 = y_m r^m + y_0 + ar^{m+1}$ for some integer a. Substituting $z_0 = y_0 + r^m$ gives

$$z_m = y_m + ar - 1 . \qquad (2.17)$$

Now for $z_0 \in Y$ we must have $r^m - 1 \geq z_0 = y_0 + r^m$ and hence $y_0 \leq 0$.

In converting x_0 to y_0 with SSW y_0 has been set negative and we can therefore conclude that $x_m = r - 1$. Proceeding with SSW, we will choose $y_m = 0$.

Equation 2.17 implies that $z_m \neq 0$ for $r \geq 2$.

This means that we can set $z'_0 = z_0 - r^m = y_0$ and $z'_m = z_m + 1 \bmod r^m$ and propagate any carry through z'_{2m}, z'_{3m} and so on. This process can only reduce the arithmetic weight of Z' or leave it unchanged.

Finally $\|Z'\| = \|Z\|$, $c(Z') \leq c(Z)$ and $z'_0 = y_0$ so case 3 provides a contradiction.

6. all remaining possibilities with $z_0 \neq 0$ and $y_0 \neq z_0$.

   We will use a process that transforms Z into some Z' such that $\|Z\| = \|Z'\|$, $c(Z') \leq c(Z)$, and either $z_0 = y_0$ or $z_0 = y_0 \pm r^m$ and $(z_1, \ldots, z_{m-1}) = (0, \ldots, 0)$. We can then refer to case 1, 2, 4 or 5 for a contradiction.

   We must have $\|Y\| \bmod r^m = \|Z\| \bmod r^m$

   $\Rightarrow \sum_{i=0}^{m-1} z_i \times r^i \bmod r^m = y_0 \bmod r^m$ because for $\mathrm{SSW}_{r,m}$ we have $y_i = 0$ for $0 < i < m$.

   $\Rightarrow z_0 - y_0 = ar^m - z_{m-1}r^{m-1} - z_{m-2}r^{m-2} \ldots - z_1 r$ for some integer a.

   Algorithm 2.5 shows the transformation from Z to Z'. This proceeds by examining z_i starting with $i = 1$. If this is not zero then $z_0 - y_0$ is a multiple of r^i. If $z_0 > y_0$ subtract r^i from z_0 and if $z_0 < y_0$ add r^i to z_0. Keep adding or subtracting until $z_i = 0$.

   To subtract r^i from z_0 and maintain $\|Z\| = \|Z'\|$ we must add 1 to z_i. If z_i exceeds $r^m - 1$ we correct it modulo r^m and propagate a carry into subsequent digits. In the original representation Z,

$z_i \neq 0$ so this carry propagation can not increase c(Z) - it either creates more zero digits or leaves the total unchanged.

To add r^i to z_0 and maintain $\|Z\| = \|Z'\|$ we must subtract 1 from z_i. If z_i falls below $-r^m + 1$ we correct it modulo r^m and propagate a borrow into subsequent digits. Again, we can only increase the number of zero digits in Z or leave the total unchanged.

When the transformation terminates, we have $z'_i = 0$ for $0 < i < m$ and hence $z_0 - y_0 = ar^m$. Also $-r^m + 1 \leq z'_0 \leq r^m - 1$ and $-r^m + 1 \leq y_0 \leq r^m - 1$ so $-(2r^m - 2) \leq ar^m \leq 2r^m - 2$ and hence $a \in \{-1, 0, 1\}$ and $z_0 = y_0$ or $z_0 = y_0 \pm r^m$ and $(z_1, \ldots, z_{m-1}) = (0, \ldots, 0)$ which was the object of the exercise.

Algorithm 2.5 Transformation from Z to Z'.

```
for i = 1 to m - 1
    while z_i <> 0
        if z_0 > y_0 then
            z_0 = z_0 - r^i
            j = i
            /* carry propagate */
            do
                c = (z_j == r - 1)
                z_j = (z_j + 1) mod r^m
                j = j + m
            while c
        else if z_0 < y_0 then
            z_0 = z_0 + r^i
            j = i
            /* borrow propagate */
            do
                b = (z_j == -r + 1)
                z_j = (z_j - 1) mod r^m
                j = j + m
            while b
        end if
```

```
        end while
    next i
```

All six cases lead to a contradiction of the assumption that there exists a representation with an arithmetic weight less than that produced by $SSW_{r,m}$. Hence the induction hypothesis—that $SSW_{r,m}$ is minimal—holds. The conversion $SSW_{r,m}$ always produces a minimal representation for its target digit set. ■

Average Weight of SSW

Following the procedure from Section 3.1, $SSW_{r,m}$ can be modelled with the state diagram in Figure 2.17.

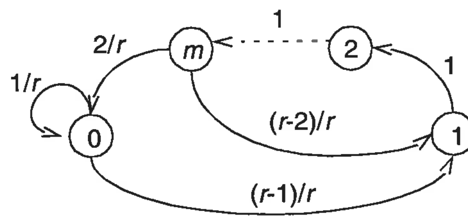

Figure 2.17 State Diagram for $SSW_{r,m}$. State 0 corresponds to the selection of a zero such that the next digit may be non-zero. State 1 selects a non-zero digit. States 2 to m represent the selection of the m - 1 zeros that must follow a non-zero digit. The mth digit following a non-zero digit can be selected as zero if it was originally 0 or r - 1.

The state transition equations can be written in matrix form as in Equation 2.4 with:

$$Q = \begin{bmatrix} \frac{1}{r} & 0 & 0 & 0 & \dots & 0 & \frac{2}{r} \\ \frac{r-1}{r} & 0 & 0 & 0 & \dots & 0 & \frac{r-2}{r} \\ 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix}.$$

Solving Equation 2.5 yields that for $m \geq 1$:

$$p_1(z) = \frac{z^m(r-1)}{z^{m+1}r + 2z - z^m - zr - 1}$$

Applying the final value theorem for Z-transforms gives:

$$\lim_{k \to \infty} p_1(k) = \frac{1}{m + 2/(r-1)} .$$

So the average arithmetic weight of $SSW_{r,m}$ for large n is:

$$\bar{c}(SSW_{r,m}) \sim \frac{n}{m + 2/(r-1)} \tag{2.18}$$

Cardinality of the SSW Digit Set

As with $USW_{r,m}$ the digit set Y for $SSW_{r,m}$ must contain a digit for every window that can be formed with m digits from $X = \{0, 1, 2, \ldots, r-1\}$. The rightmost digit in a window must be non-zero and can therefore be one of $r-1$ possible values. The remaining digits in the window can take on any of the r values. The number of different positive windows is therefore $r^{m-1}(r-1) = r^m - r^{m-1}$.

For each positive digit $y \in Y$, Y must also contain the negative digit $(y - r^m)$. Finally, Y must also contain the zero digit. Therefore:

$$|Y| = 2(r^m - r^{m-1}) + 1 . \tag{2.19}$$

Worst Case Arithmetic Weight for SSW

The worst case arithmetic weight for $SSW_{r,m}$ with $r > 2$ occurs when every m^{th} digit in the final representation is non-zero and the next most significant digit is not equal to $r-1$. Figure 2.18 demonstrates a trivial example. The worst case arithmetic weight in this case is $\lceil n/m \rceil$.

Binary Form X	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
After Recoding Y	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1

Figure 2.18 An Example of the Worst Case for $SSW_{r,m}$ with r > 2. The worst case arithmetic weight for $SSW_{r,3}$ occurs when every 3rd digit in the original representation is non zero, and in each case the next digit is not r - 1.

The case for $SSW_{2,m}$ is special as it is always possible to set the bit following a window to zero. The worst case arithmetic weight occurs when a window is formed every $m+1$ digits as shown in Figure 2.19. The worst case arithmetic weight is therefore $\lfloor n/(m+1) \rfloor + 1$.

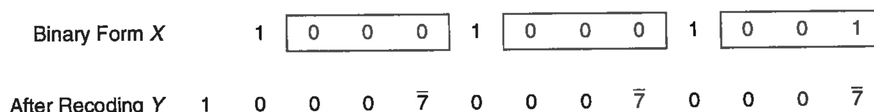

Figure 2.19 An Example of the Worst Case for SSW$_{2,m}$. The example shows the worst case for SSW$_{2,3}$.

Weight Distribution for Binary SSW

Let us now consider the distribution of arithmetic weights that result from $SSW_{2,m}$ conversion.

Define $d(k) = |K|$ where $K = \{X \in X^n \text{ s.t. } c(SSW_{2,m}(X)) = k\}$. That is, $d(k)$ is the number of representations that have an arithmetic weight of k following $SSW_{2,m}$ conversion.

To evaluate $d(k)$ we count the number of representations $X \in X$ that lead to a representation $Y \in Y$ with k non-zero digits. Each non-zero digit in Y will correspond to a window of m bits in X. In the binary case, SSW generates windows that are separated by strings of identical bits with windows always separated by at least one bit. An example window arrangement is shown in Figure 2.20.

Figure 2.20 An Example SSW Window Arrangement. Windows are formed for SSW$_{2,3}$.

The leftmost window requires special attention as it may contain fewer than m digits (if it begins to the right of x_{n-m}). Consider the following cases:

1. There are k windows of exactly m digits each (with $k \geq 1$).

   Each window except the leftmost will have a 'separating bit' to its left so there are km bits grouped into windows and $(k-1)$ bits immediately to the left of a window. This leaves $n - km - k + 1$ bits free to be arranged between and around the windows. How many ways these windows and free bits be arranged?

   This is the number of permutations two classes of things, taken all at a time. Hence:

$$\text{the number of window permutations} = \frac{(n-km+1)!}{k!\,(n-k(m+1)+1)!} \tag{2.20}$$

   Each window permutation corresponds to an arrangement of windows and separating bits like that shown in figure Figure 2.20.

   For each window permutation, how many different ways can values be assigned to the windows and separating bits? There are $(k-1)$ strings of separating bits and each can take on 2 values (all zeros or all ones). There are k windows of m bits, and the least significant bit in each window

can not be equal to the bit to its right. Therefore:

$$\text{the number of value permutations} = (2^{m-1})^k 2^{k-1} = 2^{km-1}. \tag{2.21}$$

The product of Equation 2.20 and Equation 2.21 is the number of representations of the form we seek.

2. There are $k-1$ windows of m digits and 1 window (the leftmost) of $m-i$ digits.

   Not including the leftmost window, there are $(k-1)$ windows each with a separating bit immediately to its left. This leaves $n-k(m+1)+1+i$ bits free to arrange around and between the windows. Therefore:

$$\text{the number of window permutations} = \frac{(n-km+i)!}{(k-1)!(n-k(m+1)+i+1)!} \tag{2.22}$$

$$\text{and the number of value permutations} = 2^{k-1}(2^{m-1})^{k-1}. \tag{2.23}$$

   In addition, for every arrangement of the windows to the right, the number of different values the leftmost window can take on is:

$$2^{m-i-1}. \tag{2.24}$$

   The product of Equation 2.22, Equation 2.23 and Equation 2.24 is the number of representations of the form we seek.

The total number of representations that lead to a arithmetic weight of k is the number found in the first case plus those found in the second case for all values of i from 1 to m. The final result is shown in Equation 2.25. Note that $k = 1$ is a special case as some arrangements are not possible with a single window.

$$d(k) = \frac{2^{km-1}}{k!}\left(\prod_{j=1}^{k}(n-k(m+1)+j+1) + k\sum_{i=1}^{q}2^{-i}\prod_{j=1}^{k-1}(n-km+i-j+1)\right) \tag{2.25}$$

$$\text{with } q = m \text{ if } k = 1$$

$$\text{or } q = m-1 \text{ if } k>1.$$

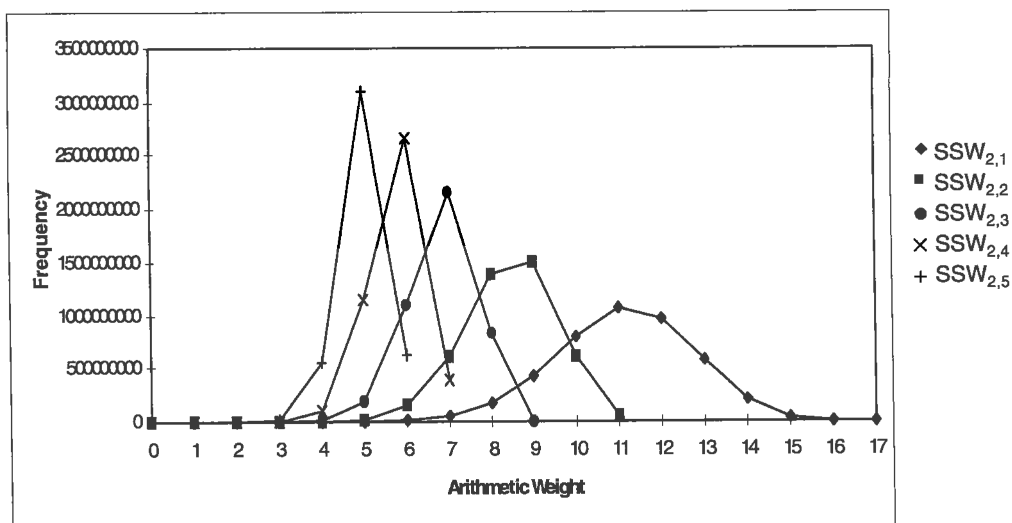

Figure 2.21 Weight Distributions for SSW. The distributions are obtained using Equation 2.25 with $r = 2$ and $n = 32$.

3.3 Other Signed Sliding Window Algorithms

By changing the rule that determines when negative digits are selected in , it is possible to define new conversions. The effect of changing the rule for is that is not set to zero in as many cases as before and hence the average arithmetic weight increases. However with a suitably designed rule, the cardinality of the resultant digit set can be decreased. This is the idea behind the modified sliding window algorithm discussed below.

Consider selecting a value for a non-zero digit in according to Algorithm 2.4 (the process is shown in Figure 2.22). The carry propagated from previous digit selections c and the current un-converted digit are such that the value of . The converted digit is originally set to the value of the window . Then may be set to the negative value if . This guarantees that the next recoded digit will be zero whenever possible.

Prior to conversion

Following Conversion

Figure 2.22 Selection of a Non-zero Digit. The digit y_i is selected from the window beginning at x_i.

It is possible to change the rule that determines when is selected to become negative. The effect will be that will not be set to zero in as many cases as before and hence the average arithmetic

weight will increase. However with a suitably designed rule, the cardinality of the resultant digit set Y can be decreased. This is the idea behind the modified sliding window algorithm discussed below.

Adaptive m-ary Segmentation Canonical Recoding

Koç and Hung's adaptive m-ary segmentation canonical recoding from [KH92] takes a binary canonic representation and forms odd signed digits using an m-bit sliding window. I will call this algorithm $\mathrm{MSW}_{2,m,z}$ where $z = \frac{2}{3}(2^m + (-1)^{m-1}) - 1$. (Although this is awkward it is less of a mouthful than 'adaptive m-ary segmentation canonical recoding'!). It can be understood as a modification of $\mathrm{SSW}_{2,m}$ with the following rule for selecting a non-zero y_i:

$$y_i = (x_{i+m-1}, \ldots, x_{i+1}, x_i + c)$$

if $y_i > z$ then $y_i \leftarrow y_i - 2^m$

else if $y_i \geq 2^m - z$ and $x_{i+m} = 1$ then $y_i \leftarrow y_i - 2^m$.

The equivalence of this recoding and that from [KH92] is demonstrated in Figure 2.23.

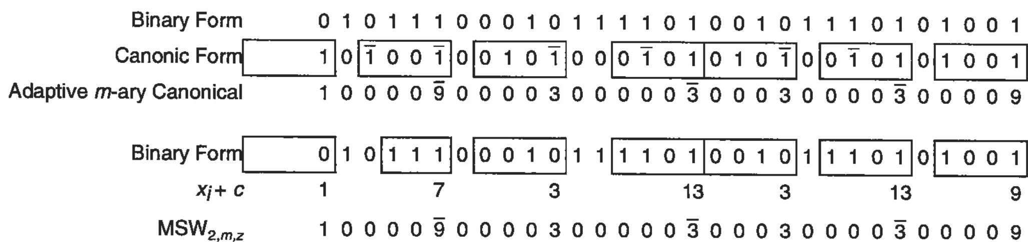

Figure 2.23 Adaptive m-ary Segmentation Canonical Encoding and MSW. A number is recoded according to these two rules with $m = 4$ to demonstrate their equivalence.

$\mathrm{MSW}_{2,m,z}$ results in the reduced target set and increased arithmetic weight described in Section 2.10. The modification is such that both the digit set cardinality and average arithmetic weight for $\mathrm{MSW}_{2,m,z}$ lie somewhere between those for $\mathrm{SSW}_{2,m-1}$ and $\mathrm{SSW}_{2,m}$. This can be seen from the graph in Figure 2.25 on page 48.

Higher Radix Modifications

As suggested by the notation adopted in the previous section $\mathrm{MSW}_{r,m,z}$ can be extended to a higher radix r. It is also possible to adjust the parameter z to any value $(r^m/2) - 1 \leq z \leq r^m - 1$.

The case $z = r^m - 1$ is identical to $\mathrm{SSW}_{r,m}$. Other values of z can be examined using the techniques developed in the previous sections.

If $z = (r^m/2) - 1$ then $\mathrm{MSW}_{r,m,z}$ achieves the same average arithmetic weight as $\mathrm{USW}_{r,m}$ but halves the number of distinct digit magnitudes in the target digit set. The new digit set is balanced around 0: $Y = \{y \text{ s.t. } (-r^m/2) < y < (r^m/2), y \neq 0 \bmod r\}$ (for r divisible by 2).

For z between these extremes the digit set cardinality and average arithmetic weight for $\mathrm{MSW}_{2,m,z}$ lie somewhere between those for $\mathrm{SSW}_{2,m-1}$ and $\mathrm{SSW}_{2,m}$.

3.4 Evaluation of Sliding Window Conversion

Having defined and studied a number of different sliding window conversions, it is now possible to try and compare their relative merits.

Sliding Windows in Radix 2

The binary signed conversion $\mathrm{SSW}_{2,m}$ has an average arithmetic weight of $n/(m+2)$ and a target digit set containing $2^m + 1$ digits. Compare this with the binary unsigned conversion $\mathrm{USW}_{2,m+1}$ which has an average arithmetic weight of $n/(m+2)$ and a target digit set containing $2^m + 1$ digits.

At first glance it is tempting to propose that $\mathrm{SSW}_{2,m}$ is identical to $\mathrm{USW}_{2,m+1}$ and nothing has been gained by the inclusion of negative digits.

However, the first thing to note is that the digit set conversions are not identical. Figure 2.24 demonstrates that there are representations for which $\mathrm{SSW}_{2,m}$ achieves a lower arithmetic weight than $\mathrm{USW}_{2,m+1}$. A situation is also shown where the converse is true.

An Example Where $SSW_{2,2}$ Beats $USW_{2,3}$

Binary	0	0	1	1	1	1	1	1	0	1	1	1	0	1	0	
$USW_{2,3}$	0	0	0	0	7	0	0	7	0	0	3	0	0	5	0	Weight = 4
$SSW_{2,2}$	0	1	0	0	0	0	0	0	$\bar{1}$	0	0	0	0	$\bar{3}$	0	Weight = 3

An Example Where $USW_{2,3}$ Beats $SSW_{2,2}$

Binary	0	0	1	1	1	0	1	1	0	1	0	1	0	1	1	
$USW_{2,3}$	0	0	0	0	7	0	0	3	0	0	0	5	0	0	3	Weight = 4
$SSW_{2,2}$	0	1	0	0	0	$\bar{1}$	0	0	$\bar{1}$	0	0	$\bar{3}$	0	0	3	Weight = 5

Figure 2.24 Radix-2 Sliding Windows.

A second important point is that $\mathrm{SSW}_{2,m}$ uses a balanced set of odd digits $Y = \pm\{0, 1, 3, \ldots, r^m - 1\}$ with only $2^{m-1} + 1$ distinct magnitudes. $\mathrm{USW}_{2,m+1}$ uses the digit set $Y = \{0, 1, 3, \ldots, r^{m+1} - 1\}$ containing $2^m + 1$ digit magnitudes. It is not uncommon that the sign

of a digit is of little consequence whereas the total number of digit magnitudes is an important consideration. This is the case for modular reduction as considered in Chapter 4 and multiplication and squaring in Chapter 5. For these operations the signed conversion has a distinct advantage.

Higher Radix Sliding Windows

Figure 2.25 compares the average arithmetic weight and digit set cardinality for a number of sliding window conversions. Note that binary sliding windows provide the lowest average arithmetic weight for a given number of digit magnitudes. This does not mean that the higher radix sliding windows are without applications. A drawback of the binary form is that digits can appear at any bit significance and hardware or software implementations must be designed to deal with single bit shifts. Compare this with radix 4 sliding windows in which digits can only occur at every 2nd bit significance. A system using this conversion may benefit from the reduced overhead of regular 2-bit shifts. An example is given in Section 2.3 of Chapter 2.

Naturally, if the radix is increased too far sliding windows will no longer be of any benefit. The number of possible digit values increases and the probability of skipping zero digits decreases.

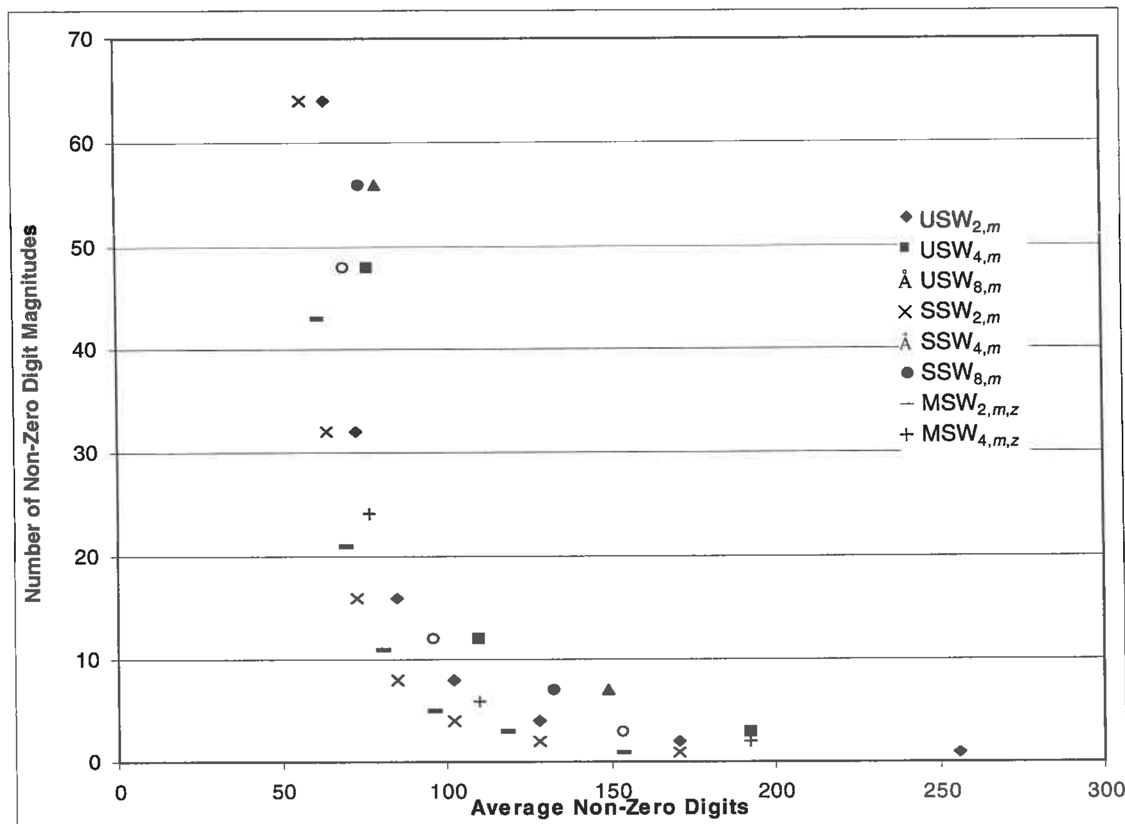

Figure 2.25 A Comparison of Sliding Window Conversions. The average arithmetic weight following conversion of a 512-bit number is shown against the number of distinct digit magnitudes in the digit set. $MSW_{2,m,z}$ is shown with z from [KH92]. $MSW_{4,m,z}$ is shown with minimum z.

4 Summary and Conclusions

This chapter began with a brief study of digit set conversion as applied to SRT quotient digit selection. This study revealed the potential benefit of conversion for systems capable of exploiting zero digits in a number representation. This specific instance of digit set conversion prompted a more general study of the field.

In Section 2 the idea of a digit set conversion was formalised and suitable terminology and notation was defined. This notation proved valuable in the understanding, classification and study of existing digit set conversions. It was found that digit set conversion is a commonly applied technique, but that it is only occasionally studied as a topic in its own right. Instead it usually considered to be an intrinsic feature of a particular algorithm. This approach has led to a confusion of different terms and

results in which it has not been uncommon for a particular digit set conversion to be re-invented a number of times. Erroneous results concerning conversions are also not unknown.

The latter part of this chapter was concerned with digit set conversion based on sliding windows. It was shown that sliding window algorithms can generate representations X with minimum arithmetic weight in $O(\log\|X\|)$ time. Two families of sliding window conversion were defined: $USW_{r,m}$ unsigned sliding windows and $SSW_{r,m}$ signed sliding windows. These families are quite general and include a number of existing digit set conversions as specific instances. It was found that both conversions are minimal. Other results were derived including digit set cardinality, worst case arithmetic weight and distribution of arithmetic weight. The techniques used to derive these results are possibly as valuable as some of the results themselves. An example of their application is provided in Appendix B in which some other digit set conversions are examined.

There is yet more scope to this work. The modified sliding windows $MSW_{r,m,z}$ include $SSW_{r,m}$ as a special case, but results for the general case have not been derived. Nor has a suitable minimal algorithm been found for the general unsigned case of *k-SR string replacement*. However, the result that $USW_{r,m}$ and $SSW_{r,m}$ are minimal may be sufficient for most circumstances. Digit set conversion to improve arithmetic weight is an attractive concept that frequently arises in the field of computer arithmetic. However, it has not always been easy to estimate the potential benefit of this approach. The average arithmetic weight for the minimal sliding window conversions now provide ready upper bounds.

Table 2.4 provides an overview of the digit set conversions covered in this chapter, with particular emphasis on the average arithmetic weight for a given digit set.

In the table, the average arithmetic weight is an approximation for large n. Many of these conversions require an extra digit under some circumstances and this is ignored for both the average and worst case weight. Some conversions behave differently for the most significant and least significant digits. These end conditions are also ignored in the analysis.

Name(s)	Ref.	r	Average Weight	Worst Case Weight	Digit Set	Min.
Non-Redundant Radix r $USW_{r,1}$		r	$\frac{n(r-1)}{r}$	n	$\{0, 1, 2, \ldots, r-1\}$	Yes
Booth Recoding	[Boo51]	2	$\frac{n}{2}$	n	$\{-1, 0, 1\}$	No
Modified Booth, Uniform Shifts of 2	[MacS61]	2	$\frac{3n}{8}$	$\frac{n}{2}$	$\{-2, -1, 0, 1, 2\}$	No
Recoded Binary Method	[Koç90]	2	$\frac{3n}{8}$	$\frac{2n}{3}$	$\{-1, 0, 1\}$	No
Recoded M-ary Method	[Koç90]	2^m	$n - \frac{5n}{2^{m+2}}$	$n+1$	$\{-2^{m-1}, \ldots, 2^{m-2} + 2^{m-1}\}$	No
Canonic Binary, $SSW_{2,1}$ Non-adjacent Binary	[Rei60]	2	$\frac{n}{3}$	$\frac{n}{2}$	$\{-1, 0, 1\}$	Yes
Generalised Non-adjacent, Minimal Signed Digit, $SSW_{r,1}$	[CL73] [AW93]	r	$\frac{n(r-1)}{(r+1)}$	n	$\{-r+1, \ldots, r+1\}$	Yes
Sliding Windows, SS(m), $USW_{2,m}$ Adaptive m-ary Segmentation	[Bar87] [HL94] [Yac91] [KH92]	2	$\frac{n}{m+1}$	n	$\{0, 1, 3, \ldots, 2^m - 1\}$	Yes
Cohen's Sliding Windows	[CL87]	2	$\left\lceil \frac{n}{m} \right\rceil \left(1 - 2^{-m}\right)$	n	$\{0, 1, 3, \ldots, 2^m - 1\}$	No
Hwang's Radix-4 Canonical	[Hwa79]	4	$\frac{2n}{3}$	n	$\{-2, -1, 0, 1, 2\}$	?
Hwang's Radix-r Canonical	[Hwa79]	2^m	$n\left(1 - \frac{2^{2-m}}{3}\right)$	n	$\pm\left\{0, \ldots, \frac{2^{m+2} - (-1)^m - 3}{6}\right\}$	?
Canonical k-SR	[GHM96]	2	Equation 2.3	$\frac{n}{\lceil \log_2 k + 1 \rceil}$	$\{0, 1, 3, \ldots, k\}$	No
Hybrid Binary -Ternary Number System	[DC95]	2, 3	$0.338n$	n	$\{0, 1\}$	?
Hybrid Ternary-Quinary Number System	[CCY96]	3, 5	$0.324n$	$\left(\log_3 2\right)n$	$\{0, 1, 2\}$	?
Adaptive m-ary Segmentation Canonical, $MSW_{2,m,z}$	[KH92]	2	$\frac{3n}{3m+4}$	$\frac{n}{m}$	$\pm\{0, 1, 3, \ldots, \left(\frac{2}{3}\right)\left(2^m + (-1)^{m+1}\right) - 1\}$	?
$USW_{r,m}$		r	$\frac{n}{m + \frac{1}{(r-1)}}$	$\frac{n}{m}$	$\{y \text{ s.t. } 0 \le y < r^m, y \ne 0 \bmod r\} \cup \{0\}$	Yes
$SSW_{r,m}$		r	$\frac{n}{m + \frac{2}{(r-1)}}$	$\frac{n}{m + \lfloor 2/r \rfloor}$	$\{y \text{ s.t. } -r^m < y < r^m, y \ne 0 \bmod r\} \cup \{0\}$	Yes
$MSW_{r,m,z}$		r	?	$\frac{n}{m}$	$\left\{y \text{ s.t. } \frac{-r^m}{2} < y < \frac{r^m}{2}, y \ne 0 \bmod r\right\} \cup \{0\}$	?

Table 2.4 An Overview of Digit Set Conversions. The average arithmetic weight shown is an approximation for large n. Many of these conversions require an extra digit under some circumstances and this is ignored for both the average and worst case weight.

Chapter 3

RSA Cryptography

PUBLIC-KEY CRYPTOGRAPHY is a mechanism for secret communication between parties who have never before exchanged a secret message. The consequences of this remarkable facility can be understood when we consider the alternative. With private-key cryptography both sender and receiver must have knowledge of a shared private key: their common 'code book'. To exchange secret messages they must have first secretly exchanged the private key. Public-key cryptography eliminates the need for this initial secret exchange. For example, using a public-key system, two people who have never met before could shout greetings across a crowded room and then proceed to shout a conversation that no-one else in the room could understand.

At the time of writing, cryptography, perhaps for the first time, has become the topic of extensive public debate. The information revolution has changed the way in which daily affairs are conducted. Transfer of money, commodities, corporate and personal information is effected through the transfer of data over digital networks. Information itself has become a commodity. In this environment, the importance of cryptography as a means of protecting the privacy of information is apparent. Moreover, new applications made possible by public-key cryptography have shifted the emphasis of cryptography from the transfer of secrets to the generation and transport of trust.

Key Exchange

The first published example of public-key cryptography is due to Diffie and Hellman in 1976 [DH76]. Their public-key distribution system allows two parties to exchange a private key over an insecure channel. This is still an important application of public-key cryptography: a public-key

system is used to initiate a private-key conversation. In this way the bulk of the data is encrypted using a private-key system—an advantage given that private-key systems are usually simpler and faster than public-key systems for a given level of security.

Digital Signatures

The generation of digital signatures is another very important application of public-key cryptography. Like a conventional signature, a digital signature is appended to a message to demonstrate the identity of its source. However, a public-key digital signature has many advantages. Firstly, it can only be generated (in a cost effective manner) by the holder of a private key and yet it can be verified by anyone with access to the corresponding public key. It is also unique for a particular document. This means that if the document is changed (by accident or design) the signature is no longer valid.

Aside from authentication of a message's source and demonstration of the message's integrity, digital signatures facilitate a raft of other services [Boy93] including: nonrepudiation (which prevents denial that a message was sent or that it was received), concurrence (which provides proof that two parties were in agreement or disagreement), and timeliness (which provides proof of the time of a message's transmission or the occurrence of another event). Clearly all of these applications are of great importance if transactions are to be performed electronically.

1 The RSA Cryptosystem

The RSA cryptosystem [RSA78] is simple and well known and has been described many times in both technical and popular literature. For almost as many times as the algorithm has appeared, it has been accompanied by a fictional couple, Alice and Bob. Rather than break this tradition, my description of RSA will take the form of an exchange between this secretive pair.

For Alice to receive secret messages she must first generate her private and public keys. This is done by selecting two prime numbers (P and Q) of approximately equal length such that their product $N = P \cdot Q$ is n-bits long. Alice must also choose an integer e which is less than N and relatively prime to $(P-1)(Q-1)$. Finally Alice must compute the modular inverse $K = e^{-1} \bmod (P-1)(Q-1)$.

Alice's public key is the pair of integers N and e known as the *modulus* and the *public exponent* respectively. Her private key is the number K which is also known as the *private exponent*. The prime factors P and Q must remain secret or be disposed of.

To send a secret message to Alice, Bob performs the following encryption:

$$C = M^e \bmod N \tag{3.1}$$

where M is the n-bit plaintext message and C is the cyphertext. Note that the encryption uses only publicly available parameters. To decode this message, Alice evaluates:

$$M = C^K \bmod N. \tag{3.2}$$

Decryption therefore requires knowledge of the private parameter K.

1.1 RSA Signature Generation

RSA can also be used to generate cryptographic signatures. To sign a message Alice performs an operation equivalent to decryption:

$$S = M^K \bmod N \tag{3.3}$$

where M is the n-bit message to sign and S is Alice's signature. Bob can verify that the signature is correct by 'encoding' it and comparing the result with the original message:

$$M = S^e \bmod N \tag{3.4}$$

Signatures are specific to both the signature's creator and the signed message. Anyone can verify that the signature is correct and that the message is unchanged since being signed, yet only the holder of the private key can generate a signature.

Signature generation is likely to be an important operation for smart-cards as it facilitates authentication of both messages and users.

1.2 Attacking RSA

There are two obvious ways of defeating RSA: factorise N and thereby deduce the private key (as in [AGLL95]); or compute e-th roots mod N and thereby perform decryption without the secret key. That RSA is still considered secure, 22 years after its introduction, is tesimony to the difficulty of these two problems. The computational effort of factorising N increases sharply with the wordlength n which is therefore chosen according to the level of security required. This leaves finding modular roots—a problem so difficult that no algorithm suitable for an attack on RSA is known.

It is important to note that RSA can never be 100% secure. It is always possible to factorise the public modulus, even if this involves searching through all possible prime factors. Therefore the security of RSA is like that of a locked door. Any door can be knocked down, but there a question of the effort

required and the benefit to be gained. Just as a door can be made thicker and stronger, the RSA modulus length can always be made longer.

Perhaps a greater threat to RSA systems than factoring are attacks that focus on details of the implementation rather than on the fundamental mathematical foundations of the cryptosystem. Section 3 discusses two implementation attacks—the fault attack and the timing attack—and examines how they can be circumvented.

1.3 RSA Operations

RSA uses modular exponentiation for both encryption and decryption (Equation 3.1 and Equation 3.2). For a fixed modulus N, the time required to perform a modular exponentiation is approximately proportional to the number of bits in the exponent. For this reason it is common to use a fixed small value such as 15 for the public exponent e. However, security dictates that the private exponent K must be approximately as long as the modulus [Wie90]. The result is that encryption and signature verification are usually much faster than decryption or signature generation.

It is unlikely that a smart-card would ever perform a significant data transfer using RSA. Instead, the usual scheme is to use a public-key system such as RSA to exchange secret keys for a more efficient private-key cryptosystem.

Therefore, we can expect a smart-card will perform only a very few RSA encryptions or decryptions per user transaction—much in the same way as a person will only sign a their name once when writing a cheque. For example, on requesting funds from an ATM, the smart-card may use a single RSA signature generation for authentication.

Most contemporary cryptographic smart-cards perform RSA using at least 512-bit integers and many support up to 1024-bit integers for higher security applications. At the time of writing, RSA Laboratories recommends key sizes of 768 bits for personal use, 1024 bits for corporate use, and 2048 bits for extremely valuable keys [RSA98].

From these considerations we can specify an appropriate goal for the performance of an RSA smart-card. I will take as a benchmark the time to perform a single 1024-bit RSA signature generation. With a signature generated for each user transaction it is reasonable to aim for a processing time of around 1 second. This is comparable, and probably faster, than the time it currently takes for an automatic teller machine to process a cash withdrawal using a magnetic stripe card.

1.4 Other Public-Key Cryptosystems

A review of the literature of cryptology will reveal that there are only a few known public-key cryptosystems that are considered to be secure [Sch96], [MOV97]. This is despite the fact that cryptographic research is very active, and that cryptography has become exceedingly important for many information systems. In fact most of today's cryptographic systems employ one of only two types of cryptosystem: those based on the problem of factorising large numbers and those based on the discrete logarithm problem.

The foremost example of a factoring based cryptosystem is RSA which has been both a de-facto and official security standard for many years [Riv92]. For example the Australian key management standard AS2805.6.5.3 specifies the use of RSA. A list of the very large number of products licensed to use RSA is available from RSA Laboratories [RSA98]. Other discrete logarithm based cryptosystems or signature schemes are usually very similar to RSA. Examples include the Pohlig-Hellman [PH78] and Rabin cryptosystems [Rab79], [Wil80].

Diffie Hellman key exchange is an example of a system based on the difficulty of calculating logarithms over a finite field. The most popular systems use the Galois Field $GF(p) = \{0, 1, 2, 3, \ldots, p\}$ where p is a prime number and all arithmetic is performed modulo p. For Alice and Bob to share a private key using Diffie Hellman key exchange:

1. Alice chooses A, a random non-zero member of $GF(p)$ and sends Bob the value $k_A = \alpha^A \bmod p$ where α is a publicly known (non-zero) member of $GF(p)$.
2. Similarly Bob chooses a non-zero element B and sends Alice the value $k_B = \alpha^B \bmod p$.
3. Both Alice and Bob can calculate the private key $k_{AB} = \alpha^{AB} \bmod p$. For Alice $k_{AB} = (k_B)^A \bmod p$ and for Bob $k_{AB} = (k_A)^B \bmod p$.

The public-key cryptosystem proposed by ElGamal in [ElG85] extends the Diffie Hellman key exchange scheme to provide privacy and digital signatures. The security of both systems depends upon the difficulty of calculating logarithms in finite fields. The use of various fields has been proposed but $GF(p)$ remains the most popular. It's security is believed to be comparable to that of RSA. Arithmetic based on the group of elliptic curves over finite fields can also be used and there are no known feasible attacks for such cryptosystems [Men93]. Elliptic curve cryptosystems have the potential to provide fast and secure systems with shorter key lengths; however they are not yet widely used.

There have been many modifications to the ElGamal scheme to provide alternative functions such as key exchange or password authentication. ElGamal based signature schemes have become particularly important with one variant, the Digital Signature Standard, being certified as the U.S. Federal Information Processing Standard FIPS 186. Other variants were proposed by Feige, Fiat and Shamir [FFS98] and Schnorr [Sch91].

The McEliece cryptosystem [McE78] is the best known of the remaining unbroken public-key cryptosystems. It is based on error correcting codes but suffers from two drawbacks: it uses an exceptionally long key and generates cyphertext twice the length of the plaintext. Although one of the oldest public-key cryptosystems, and faster than RSA, it has never been widely accepted.

Summary

All of the important public-key cryptosystems covered in the previous section are based on the arithmetic of large integers. The arithmetic algorithms and ideas developed in this thesis, although tested against the benchmark of RSA signature generation, can be applied to any of these public-key systems.

Finally, it is interesting to consider the position of public-key cryptography, given the existence of so few trusted cryptosystems. This is elegantly addressed by Diffie:

> Now that it has achieved acceptance, public-key cryptography seems indispensable. In some ways, however, its technological base is disturbingly narrow. With the exception of the McEliece scheme and a cumbersome knapsack system devised explicitly to resist the known attacks, virtually all surviving public-key cryptosystems and most of the more numerous signature systems, employ exponentiation over products of primes...
> From the standpoint of conventional cryptography, with its diversity of systems, the narrowness bespeaks a worrisome fragility. This worry, however is mitigated by two factors.
>
> - The operations on which public-key cryptography currently depends - multiplying, exponentiating, and factoring - are all fundamental arithmetic phenomena. They have been the subject of intense mathematical scrutiny for centuries and the increased attention that has resulted from their use in public-key cryptosystems has on balance enhanced rather than diminished our confidence.
>
> - Our ability to carry out large arithmetic computations has grown and now permits us to implement our systems with numbers sufficient in size to be vulnerable only to a dramatic breakthrough in factoring, logarithms, or root extraction.

> It is even possible that RSA and exponential key exchange will be with us indefinitely. The fundamental nature of exponentiation makes both good candidates for eventual proof of security... the position of public key cryptography should be seen not as a fragile, but as a strong one. [Dif88]

2 Exponentiation

Exponentiation, and especially modular exponentiation, is a fundamental operation in RSA and other important cryptosystems. Ronald Rivest, one of the originators of the RSA cryptosystem, observed that:

> Modular exponentiation is an interesting computational problem in that it seems intrinsically 'sequential': using extra hardware or extra parallelism doesn't seem to help beyond the amount it helps to speed up the underlying modular multiplications. To raise a k-bit number to a k-bit power modulo a k-bit modulus seems to take $O(k)$ multiplications. [Riv85]

This is indeed the case for ordinary (non-modular) exponentiation performed using multiplication. It is possible to prove that $\log_2 B$ is a lower bound on the number of multiplications required to evaluate A^B [Knu97]. The binary algorithms below require at most two multiplications for every bit in B and hence provide a ready upper bound of $2\log_2 B$ multiplications.

Modular exponentiation can be performed using repeated modular multiplications in the same way as exponentiation is performed with repeated multiplications. In published work it is usual to treat exponentiation and modular exponentiation as identical problems. To the best of my knowledge an algorithm for computing modular exponentials directly, without a direct application to ordinary exponentiation, has never been proposed.

Exponentiation is a well studied problem and recent surveys can be found in [Knu97], [MOV97] and [Dhe98]. This section provides an overview of the field.

Addition Chains

The problem of finding the minimum number of multiplications to evaluate a power is often considered from the perspective of addition chains. An addition chain for B is a list of numbers c_i with $c_0 = 1$ and $c_k = B$ such that any element c_i can be expressed as the sum of two elements earlier in the chain. That is, for all $i > 0$ one can find $j < i$ and $k < i$ such that $c_i = c_j + c_k$.

For example, the sequence 1, 2, 3, 5, 7 is an addition chain for 7. This chain also represents a means of raising to the power of 7 due to the additive property of powers: $A^2 = A^1A^1$, $A^3 = A^1A^2$, $A^5 = A^2A^3$, $A^7 = A^2A^5$. Thus the problem of finding a minimal sequence of additions to evaluate a

power is equivalent to finding a minimal length addition chain for that power. Unfortunately, finding minimal length addition chains is a hard problem.

As well as addition chains, one may consider addition/subtraction chains in which a term can be the difference of two earlier terms. For example, 1, 2, 4, 16, 32, 31 is an addition/subtraction chain for 31 and is shorter than the best possible addition chain for this number. A subtraction in a chain is equivalent to division (or multiplication by the inverse) in exponentiation.

A star chain is a special case of an addition chain in which each element c_i is the sum of the previous element c_{i-1} and one other element in the chain c_j. For exponentiation using star chains, one input to the multiplication is always the result of the previous multiplication. This may lead to more efficient implementations but it is interesting to note that there are numbers for which the minimum length addition chain is shorter than the minimum length star chain.

Other general results on addition chains can be found in [Knu97].

Power Tree and Factor Method Exponentiation

Good star chains can be found by building a data tree known as the power tree [Knu97]. The first 5 rows of the power tree are shown in Figure 3.1. The tree can be simply expanded to provide a star chain for any number. Exponentiation based a power tree star chain is guaranteed to use the same number or fewer multiplications than the binary algorithms described below. However, being a star chain method, the power tree will not always produce a minimum length addition chain. In fact, the power tree does not always produce a minimum length star chain.

The magnitude of the exponent used in RSA cryptography is such that the it will not be feasible to find a good star chain for this exponent using the power tree directly.

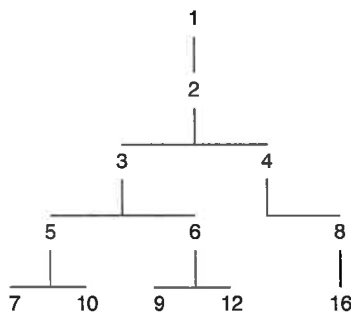

Figure 3.1 The First 5 Rows of the Power Tree.

A method of exponentiation based on recursive factoring of the exponent B is described in [Knu97]. According to Knuth, for values less than or equal to 100000, the power tree method uses fewer operations than the factor method 88 803 times, ties 11 191 times and loses 6 times. Even so, the factor method uses fewer total multiplication and squaring operations on average than the binary methods discussed below. However, for RSA decryption or signature generation, the exponent B is as long as the modulus and it is extremely unlikely that it can be factorised in a timely fashion.

Binary Right-to-left (Russian Peasant) Exponentiation

The binary right-to-left algorithm, a well known method for exponentiation in hardware and software, is described by the pseudo-code of in Algorithm 3.1.

Algorithm 3.1 Binary Right-to-left Exponentiation. The result A^B mod N is returned in C. The exponent B is n-bits long.

```
C = 1
for i = 0 to n - 1
    if b_i = 1 then
        C = C * A mod N
    end if
    A = A * A mod N
next i
```

Note that a modular square is required for every bit in B and a modular multiply is required for every non-zero bit in B. An advantage of this algorithm is that if two multipliers are available, the square and the multiply operations can be performed in parallel [Riv85], [OK91], [Chi93a].

One might consider extending the right-to-left algorithm to work in a higher radix so that more than one bit of the exponent is considered at a time; however the multiplication step then becomes $C = C + A^{b_i} \bmod N$ and as A changes at each iteration, we cannot pre-compute the values A^b which must themselves be evaluated using a series of multiplications.

Binary Left-to-right (Square and Multiply) Exponentiation

Binary left-to-right exponentiation can be described by the following pseudo-code algorithm:

Algorithm 3.2 Binary Left-to-right Exponentiation. The result A^B mod N is returned in C.

```
C = 1
for i = n - 1 to 0
    C = C * C mod N          // Square
    if b_i = 1 then
        C = C * A mod N      // Multiply
    end if
next i
```

Evaluation requires $n-1$ squares (the very first square is trivial). If it is assumed assume that $b_{n-1} = 1$ then there are $c(B)-1$ multiplications (where $c(B)$ is the number of non-zero bits in B).

Rather than consider B on a bit by bit basis, it is possible to express B in a higher radix r and consider it digit by digit. The resulting algorithm, shown as Algorithm 3.3, is sometimes known as *m*-ary exponentiation or *window* exponentiation. This can be implemented efficiently if the values $A^2 \bmod N$, $A^3 \bmod N$, ..., $A^{r-1} \bmod N$ are pre-computed.

Algorithm 3.3 Higher Radix Left-to-right Exponentiation. The result A^B mod N is returned in C. The exponent B is n digits long.

```
// Pre-computation Phase
A_1 = A
A_2 = A * A mod N
for i = 3 to r - 1
    A_i = A_(i-1) * A mod N       // A_i = A^i mod N
next i
// Evaluation Phase
C = 1
for i = n - 1 to 0
    C = C^r mod N                 // For r = 2^t this can be implemented as t squares
    if b_i <> 0 then
        C = C * A_(b_i) mod N
    end if
next i
```

Left-to-right Exponentiation with Unsigned Sliding Windows

In Algorithm 3.3 a multiplication can be skipped whenever a zero digit $b_i = 0$ occurs. It is possible, therefore, to improve the performance of this algorithm using the digit-set conversion techniques of Chapter 2. Note that this approach can reduce the number of multiplication steps but does nothing to reduce the number of squares which is fixed by the bit-length of the exponent.

Unsigned binary sliding windows (and the related *string replacement algorithms*) are used with left-to-right exponentiation in [CL87], [Yac91], [HL94] and [GHM96]. The exponent B is re-written with odd digits (using for example $USW_{2,m}$) to decrease the average number of non-zero digits for a given number of pre-computed powers. It is interesting to consider the total number of multiplications required for both pre-computation of the digit powers and evaluation of the exponential. For $USW_{2,m}$ the digit powers $\{A \bmod N, A^3 \bmod N, \ldots, A^{2^m-1} \bmod N\}$ are required. These can be determined by first squaring to find $A^2 \bmod N$ and then by successive multiplications to find $A^3 \bmod N$, $A^5 \bmod N$ and so on. In this way pre-computation takes 1 square and $2^{m-1}-1$ multiplications. Evaluation requires one multiplication for each non-zero digit in B, which for $USW_{2,m}$ is given by $n/(m+1)$ where n is the number of bits in B. The total number of multiplications is shown in Figure 3.2 for various values of m with $n = 512$. At this modulus length the system is optimal with $m = 5$.

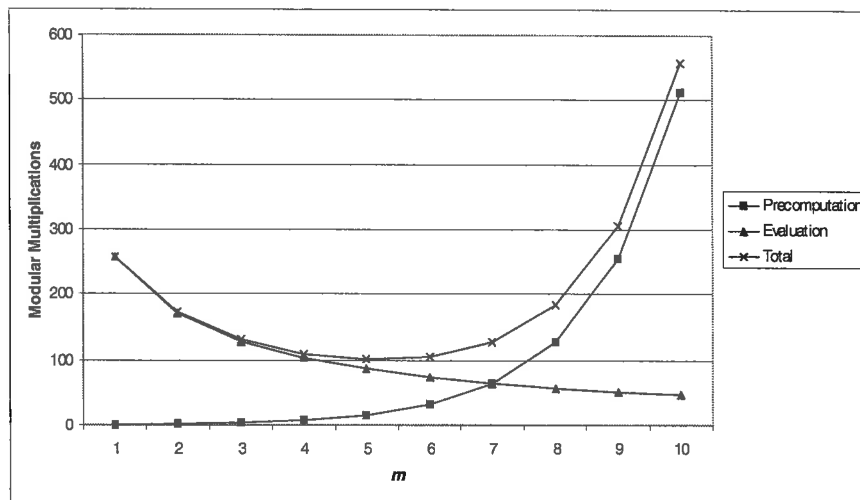

Figure 3.2 Average Modular Multiplications for Exponentiation with $USW_{2,m}$.

Left-to-right Exponentiation with Signed Sliding Windows

Using the modular inverse, it is possible to introduce negative digits into the representation of B. For example, whenever a digit $b_i = -1$ occurs in Algorithm 3.3, the multiplication step becomes $C = C \times A^{-1} \bmod N$. $A^{-1} \bmod N$ is the modular inverse of A, a positive integer with the property

$A \times A^{-1} \bmod N = 1$. The modular inverse always exists for an RSA modulus N and can be found with the extended Euclid algorithm as described in [Knu97].

The signed binary digit set $\{-1, 0, 1\}$ is used to recode B in [Bri83], [JM89], [Zha93] and [GHM96]. The first uses the sub-optimal recoding described in Section 2.6 of Chapter 2 and the latter use the binary canonic form.

Note that the binary canonic form $SSW_{2,1}$ only performs on average as well as the unsigned conversion $USW_{2,2}$. The former requires evaluation and storage of $A \bmod N$ and $A^{-1} \bmod N$; the latter $A \bmod N$ and $A^3 \bmod N$. In this case nothing has been gained by the introduction of negative digits and this is true of all of the binary sliding window forms. However, this does not apply to multiplication or reduction—Chapter 4 and Chapter 5 demonstrate that signed recoding is an advantage for these operations.

Other Exponent Recoding Techniques

Another approach to recoding the exponent for left-to-right exponentiation is taken in [BC90]. The exponent is examined and groups of adjacent bits are formed into odd digits as in Figure 3.3. Heuristic techniques are then applied to find an addition chain that contains all of the required odd digits. This addition chain is used to pre-compute $A^{b_i} \bmod N$ for each of the required digits b_i. Finally left-to-right exponentiation is used to compute $A^B \bmod N$. The authors claim a final addition sequence of average length 605 for a 512-bit exponent. From the results published in the paper it is possible to derive that to achieve this, an average of 30 different digit powers must be pre-computed. This is a small improvement over $USW_{2,5}$ which gives an average addition sequence of length 612 and requires storage of 32 powers. This comparison does not take into account additional temporary storage used by the heuristic approach to store powers that are required in the addition sequence but do not appear as digits of B.

```
1011000111001  000000 1110100101  00 1110101  00000 101111  0000 111110011  00 101010111
    5689                 933            117             47                499             343
```

An addition sequence for these digits is: 1, 2, 4, 8, 10, 11, 18, 36, **47**, 55, 91, 109, 117, 226, **343**, 434, 489, **499**, **933**, 1422, 2844, 5688, **5689**

Figure 3.3 Bos and Coster's Digit-Set-Conversion. Adjacent groups of non-zero bits form odd digits. An addition sequence containing all of these required digits is then sought.

In [Yac91] Yacobi applies data compression techniques to left-to-right exponentiation. Yacobi's algorithm re-writes the binary exponent B as odd digits by considering it least significant bit first. Digit b_i is chosen by first skipping any zeros to find a non-zero bit. This and the sequence of bits to

the right is followed until they form a string that has not yet appeared in any b_j for $j < i$. The new b_i will differ from some existing b_j by only the most significant bit. This process expresses B with a set of digits for which an efficient addition sequence can be found. Having calculated all of the required digit powers $A^{b_i} \bmod N$ using this sequence, the final exponentiation is evaluated using the left-to-right algorithm.

There are 'compressible' exponents for which the Yacobi algorithm performs very well. However, 5000 experiments on 1024-bit random exponents showed that on average the Yacobi method required 68 multiplications in pre-computation, 137 multiplications to evaluate the exponent, and 8.5 squares in pre-computation. These figures compare unfavourably with $USW_{2,6}$ which would require 31 multiplications in pre-computation, 146 multiplications to evaluate the exponent, and only 1 square in pre-computation.

```
11011 01011 000 1101 00 101 0 011 01 11 0 1
-----  -----    ----    ---   --- -- --   -
 27     11       13      5     3  1  3    1
```

An addition sequence for these digits is: 1, 2, 3, 4, 5, 8, 11, 13, 16, 27

Figure 3.4 Yacobi's Compression Technique. Beginning from the right, groups of bits are formed into digits such that the new digit differs from a previous digit by only the most significant bit. An addition sequence containing all of these required digits is trivially derived.

Fixed Base Exponentiation

If the exponent is split into halves such that $B = 2^{n/2}B_1 + B_0$ then $A^B = \left(A^{2^{n/2}}\right)^{B_1} \times A^{B_0}$ which can be more rapidly evaluated provided $A^{2^{n/2}}$ is computed in advance. Partitioning the exponent in this way into 2 or more parts provides a mechanism to shift most of the squaring operations from the evaluation phase into pre-computation. This is the key idea behind the exponentiation schemes of [BMGW93], [LL94] and [deR95]. However, all of these schemes are characterised by significant pre-computation effort and storage requirements. They are most appropriate for cryptosystems in which many powers A^B are calculated for a fixed base A (such as Elgamal or DSS).

In the case of RSA signatures (as described in Section 1.1) a new value A is submitted for each signature. Pre-computation of powers of A is a valid technique, but this pre-computation must be performed for every signature. Thus it is important to measure the total computation effort required for both pre-computation and evaluation. It is also important to consider the memory implications, particularly for a smart-card implementation.

Lim and Lee's algorithm [LL94] is an improvement over the method described in [BMGW93] for smaller pre-computed tables. The best case for 512-bit exponentiation occurs when 6 values are pre-

computed. The new algorithm requires on average 767 squares and 192 multiplications whereas $USW_{2,3}$ stores only 4 pre-computed values and performs on average 513 squares and 131 multiplications.

Paul de Rooij also improves on [BMGW93] for small pre-computed tables by using vector addition chains [deR95]. This time the algorithm is competitive with the sliding window technique when total pre-computation and evaluation effort is taken into account. For example, for 2 pre-computed values, 4 temporary n-bit values are required, 256 squares are performed in pre-computation and 411 multiplies are required in evaluation: a total of 667 operations. $USW_{2,3}$ has the same memory requirements and takes 644 operations. For large pre-computed tables, the de Rooij algorithm does offer an improvement over sliding windows. However, the algorithm is more difficult to implement than the sliding window form and requires a division at each iteration. This will be slow when implemented on smart-cards which do not usually have a hardware divider.

The mixed radix methods described in [DC95] and [CCY96] are also intended for a system where a large number of powers can be pre-computed and this pre-computation occurs off of the critical path of system performance. For a given number of digits, the mixed radix representation can achieve a lower average arithmetic weight than the sliding window techniques. However, conversion to mixed radix form requires repeated division operations that will significantly add to the cost of pre-compuatation.

3 Attacking RSA Implementations

The RSA cryptosystem has been under intense public scrutiny since it was proposed in 1978. That it has never succumbed to the best efforts of the public cryptology community gives some confidence in its security.

However, this is only true of the RSA cryptosystem as it is theoretically described. The timing attack and the fault attack are two significant attacks that exploit the characteristics of particular RSA implementations. They are described in more detail in the following sub-sections.

When dealing with this kind of attack it is important to note that the security of a system must not rely on the secrecy of the implementation (this is known as Kerckhoff's criteria). While it is common practice to keep the fine details of a commercial implementation secret, one should always work with the expectation that an attacker will have all of this information at their disposal. That this is reasonable is borne out by the successful attack on the Netscape Secure Socket Layer, a protocol for

secure Internet transactions. In the attack, the Netscape Web browser software was reverse-engineered to discover its mechanism for generating pseudo-random numbers and hence private keys [GW96].

3.1 The Timing Attack

An attack that exploits the variable timing of cryptographic operations was first publicly proposed by Kocher in [Koc96]. In this scheme the private key or other secret information is deduced from measurements of the time taken to perform a decryption or signature generation. A successful attack on an RSA implementation was reported in [Dhe98].

Algorithmic optimisation is a significant source of timing variability. As this thesis is largely concerned with exactly this kind of optimisation, it is necessary to consider the timing attack in some detail. Begin with the signature generation as in Algorithm 3.4.

Algorithm 3.4 RSA Signature Generation. The public message M is signed with the secret key K to form the signature S.

```
S = M
for i = n - 2 down to 0
      S = S^2 mod N
      if k_i = 1 then
            S = S * M mod N
      end if
next i
```

As an illustrative case, imagine that the multiplication step is usually fast, but can be significantly slower with some values of *S* and *M*. The attacker must know enough of the implementation to be able to determine if the multiplication of two operands will be fast or slow.

The attacker begins by attempting to discover bit k_{n-2} of the private key. If this bit is 1 then the multiplication will be $M \times M^2 \bmod N$. If $k_{n-2} = 0$ the multiplication is not performed. Measurements are taken of time required to sign a variety of messages chosen such that the multiplication would be slow if it was performed. A second set of results is obtained for messages in which the multiplication would be fast. To find k_{n-2} the attacker tests the statistical hypothesis that the mean of both sets of measurements is the same. If this is true it implies that the multiplication was never performed and hence k_{n-2} is zero. The inverse hypothesis reveals if this bit was one. (In both [Koc96] and [Dhe98] a test using the sample variance rather than the mean is found to reveal the key

with fewer timing measurements.) Given k_{n-2} the attack can proceed to examine k_{n-3} and so on until the entire private key has been disclosed.

Obfuscating Evaluation Times

A naive approach to preventing the timing attack would be to complicate the timing variations by introducing a random delay. At best such a delay can only increase the number of timing samples an attacker requires to recover the secret key. At worst an attacker may be able to detect when the processor is expending its random idle time. This could be achieved by observing changes in power consumption or some other indicator of processor activity.

Even performing the multiplication step and discarding the result if it is not required does not eliminate vulnerability. The time taken to perform the square in the following iteration can reveal whether the multiplication was really performed. Finally if one chooses to fix the time for the square, the timing of subsequent multiplication steps can still be used to recover the key.

Equalising Evaluation Times

One can seek to guard against the timing attack by ensuring the evaluation time is entirely independent of the operand M. This means that both the multiplication and the square step must run in fixed time. If this can be achieved, the most an attacker can reveal from timing observations is the arithmetic weight of the private key K.

However, as Kocher points out, creating a fixed time implementation can be difficult. Compiler optimizations, machine instruction timings, and even cache hits can all introduce unexpected timing variations. Moreover, systems with even the smallest variations are still vulnerable—they merely require more timing samples to reveal the key.

Where a fixed time implementation is possible—such as with carefully coded assembly routines on a simple microprocessor—all operations must take as long as their slowest case. This eliminates a wide variety of optimizations that improve the average case of execution rather than the worst case.

Blinding

Variable execution time can be maintained if the message input to the exponentiation is hidden from the attacker. To achieve this, Kocher suggested the use of a technique known a blinding [Koc96] which has subsequently been adopted by RSA Data Security for their B-safe software toolkit. They also recommend that their clients incorporate blinding into their implementations [RSA95].

To generate an RSA signature using blinding, start with a pair of random numbers u_1 and v_1 such that $u_1 = \left(v_1^{-1}\right)^e \bmod N$. To sign the i-th message M_i, transform the input message by multiplying by u_i:

$$M_i' = u_i \times M_i \bmod N \tag{3.5}$$

The exponentiation is then performed using M_i' to generate the signature S_i'. Multiplication by v_i gives the required signature:

$$S_i = v_i \times S_i' \bmod N \tag{3.6}$$

Generation of the modular inverse v_i^{-1} is a slow operation but the pairs (u_i, v_i) cannot be reused otherwise they may be disclosed by a timing attack on the blinding operations themselves (Equation 3.5 and Equation 3.6). The solution is to update u_i and v_i following each signature generation. A simple modular square is sufficient for this:

$$u_{i+1} = u_i^2 \bmod N,\ v_{i+1} = v_i^2 \bmod N \tag{3.7}$$

The cost of blinding is two modular squares, two modular multiplications and the storage of two n-bit numbers. The squares can be pre-computed between signature generation transactions.

Aside from insurance against unexpected timing variations, the benefit of blinding is that it permits the use of average case optimizations. That these costs can be justified in light of the optimizations is one of the issues to be addressed in the following sections.

3.2 The Fault Attack

A fault attack [BDeML97] relies on the occurrence of hardware or software errors that cause a cryptographic implementation to generate and publish erroneous results. In some circumstances it is possible for an attacker to analyse these incorrect results to extract secret information from the system.

Two kinds of fault are possible: transient or latent. The former are random hardware events such as a memory location suddenly becoming corrupt or gates spontaneously producing incorrect results. Errors of this kind occur only infrequently but they can be encouraged under some circumstances. Imagine, for example, that the system under attack is a tamper-resistant smart-card. A disturbing array of techniques for tampering with these devices is presented in [Bov98]. Sub-micron probes can be used to actively insert information onto bus lines or into registers, scanning electron microscopes

can be used to visualise voltages on very small tracks, and focused ion beam systems can etch out tiny holes or deposit metals onto the chip's surface.

Latent errors are permanent bugs in the hardware or software which cause a system to generate incorrect results under certain conditions. These faults may only manifest themselves very rarely but do so in a predictable manner.

A Fault Attack On RSA Blinding Parameters

A simple fault attack on the blinding parameters of an RSA signature generation system will serve as an example of fault attacks on transient errors.

Consider a smart-card system that uses blinded RSA signatures as described in Equation 3.5 to Equation 3.7. The attacker begins by obtaining the correct signature S_1 for any message M_1 such that $S_1 = M_1^K \bmod N$. While the smart-card is waiting idle for a new message to sign, the attacker can try and induce errors with the objective of corrupting the value of a single bit in the register holding v_1. If bit b is changed then v_1 becomes:

$$\hat{v}_1 = v_1 \pm 2^b \tag{3.8}$$

The original message M_1 is then signed again, but this time blinded with the erroneous value $\hat{v}_1$ to give an erroneous signature:

$$\tilde{S}_1 = \hat{v}_1 (u_1 M_1)^K \bmod N \tag{3.9}$$

Substituting Equation 3.8 into Equation 3.9 yields Equation 3.10 in which the attacker knows every term except the value of $\pm 2^b$. In n-bit RSA this unknown term can take on $2n$ possible values so that the attacker is left with $2n$ candidate values for the blinding parameter v_1.

$$v_1 = \frac{\pm 2^b}{\tilde{S}_1 / S_1 - 1} \bmod N \tag{3.10}$$

To determine the correct value of this blinding parameter, the attacker chooses a candidate at random and launches a timing attack on the secret key K. Useful correlations to reveal key bits will only emerge if the guess of v_1 was correct.

Fault Attacks On Latent Errors

A latent error is a "bug" in either hardware or the software that will cause erroneous results under special conditions. The infamous Intel division bug [CT95] is testimony to the fact that this kind of

fault can be difficult to detect. However, once discovered, it may be possible to exploit a latent fault to compromise secret information.

Consider an example that proceeds along similar lines to the timing attack on the exponentiation in Algorithm 3.4. Imagine that an attacker can choose M such that evaluation of $(M^3)^2 \bmod N$ produces an erroneous result and yet $(M^2)^2 \bmod N$ is evaluated correctly. If the correct signature is ever produced for such a message then the attacker knows that bit k_{n-2} of the secret key must be zero. Given this bit the attacker can proceed to determine the remaining bits of the secret key.

3.3 Preventing Timing and Fault Attacks

Blinding to hide the exact number exponentiated may protect a system against timing attacks and fault attacks on latent errors but in Section 3.2 it was demonstrated that blinding alone is not sufficient to defeat fault attacks on transient errors.

Fault attacks on transient errors can be circumvented by verifying results before they are released to the public. In an RSA signature scheme, this would mean that a new signature must be verified by the signature generator before it is published. This involves checking the signature using the public key, an operation that usually only requires a few multiplications.

Note that result verification alone can not prevent fault attacks on latent errors in which the existence of a correct result can be sufficient to disclose the key.

To guard against timing attacks as well as fault attacks on transient and latent errors one can employ both blinding and result verification.

4 RSA Functional Specification

Let us begin the design of an RSA signature generation function for a smart-card by developing a high level specification. The design will concentrate on the process of signature generation from the moment a message arrives to be signed, to the moment the signature is ready to be made public.

The critical step is the modular exponentiation given in Equation 3.3 but, as discussed in the preceding sections, blinding and verification operations are also required. This is not all—the following sections introduce more operations that must be performed before and after the exponentiation.

4.1 Montgomery Operations

Montgomery reduction is introduced in Chapter 4 and will be implemented in the smart-card design of Chapter 6. While Montgomery reduction can improve exponentiation performance, this comes at the cost of extra 'transformation' operations.

To reduce A modulo N using Montgomery reduction one must first find the N-residue of A. This will be given the symbol $\bar{A}$. The relationship between A and $\bar{A}$ is shown in Equation 3.11. Equation 3.12 defines the Montgomery reduction function.

$$\bar{A} = AR \bmod N \text{ where } R = 2^n. \tag{3.11}$$

$$\mathrm{MR}_N(A) = A/R \bmod N$$

$$\text{or equivalently } \mathrm{MR}_N(\bar{A}) = A \bmod N \tag{3.12}$$

The Montgomery product is defined in Equation 3.13 such that the Montgomery product of two N-residues, $\bar{A}$ and $\bar{B}$, is the N-residue of the product AB.

$$\mathrm{MP}_N(\bar{A}, \bar{B}) = \mathrm{MR}_N(\bar{A} \times \bar{B}) = ABR \bmod N \tag{3.13}$$

The Montgomery exponentiation function is defined similarly:

$$\mathrm{ME}_N(\bar{A}, e) = A^e R \bmod N. \tag{3.14}$$

An objective of the following sections is to perform all of the time-critical RSA signature calculations using only Montgomery operations.

4.2 Chinese Remainders

In its most general form, the Chinese Remainder Theorem or CRT, is concerned with the simultaneous solution of a series of congruence relations. Its application to RSA is due to Quisquater and Couvreur, and also Krishnamurthy and Ramachandran [QC82], [KR80]. Using the CRT it is possible to perform n-bit exponentiation modulo N as two $n/2$-bit exponentiations modulo P and Q as shown below. This result appears often in RSA literature but for a clear and complete derivation refer to Simmons [Sim92].

Given:

$$M_P = M \bmod P, M_Q = M \bmod Q,$$

$$Q_P^{-1} = 1/Q \bmod P, \; P_Q^{-1} = 1/P \bmod Q,$$

$$\text{and } K_P = K \bmod (P-1), \; K_Q = K \bmod (Q-1)$$

the $n/2$-bit exponents can be evaluated:

$$S_P = (M_P)^{K_P} \bmod P \text{ and } S_Q = (M_Q)^{K_Q} \bmod Q$$

and then:

$$S = M^K = \left(S_P Q Q_P^{-1} + S_Q P P_Q^{-1} \right) \bmod N \tag{3.15}$$

or for $P < Q$:

$$S = \left(\left((S_P - (S_Q \bmod P)) \times Q_P^{-1} \right) \bmod P \right) \times Q + S_Q \; . \tag{3.16}$$

Note that these equations are not in a form suitable for Montgomery operations and it is not immediately obvious how to combine this result and Montgomery reduction without converting back and forth between N-residues, P-residues and Q-residues.

4.3 Montgomery Reduction and Chinese Remainders

A special case of the CRT for two congruence equations can be written thus:

> For primes P and Q with $P < Q$ and arbitrary integers $A \in [0, P)$ and $B \in [0, Q)$ there exists a unique $X \in [0, PQ)$ with $X = A \bmod P$ and $X = B \bmod Q$. The value of X is given by $X = \left(\left((A - (B \bmod P)) \times Q_P^{-1} \right) \bmod P \right) \times Q + B$ for $Q_P^{-1} = 1/Q \bmod P$.

To combine Montgomery reduction with the CRT consider $\bar{S} = R \times M^K \bmod (PQ)$. Let this be the unique solution to $\overline{S_P} = \bar{S} \bmod P$ and $\overline{S_Q} = \bar{S} \bmod Q$ for some values of $\overline{S_P}$ and $\overline{S_Q}$.

What is the value of $\overline{S_P}$?

$$\overline{S_P} = \bar{S} \bmod P = (R \times M^K \bmod (PQ)) \bmod P = R \times M^K \bmod P .$$

$$\text{Let } K = t(P-1) + K_p \text{ i.e. } K_P = K \bmod (P-1) .$$

$$\text{Then } \overline{S_P} = R \times (M^{P-1})^t \times M^{K_P} \bmod P$$

$$\text{and via the Euler-Fermat[1] theorem } \overline{S_P} = R \times M^{K_P} \bmod P . \tag{3.17}$$

Similarly for $\overline{S_Q}$ we have:

$$\overline{S_Q} = R \times M^{K_Q} \bmod Q . \tag{3.18}$$

And finally the solution for $\bar{S}$ is:

$$\bar{S} = \left(\left(\left(\overline{S_P} - \left(\overline{S_Q} \bmod P\right)\right) \times Q_P^{-1}\right) \bmod P\right) \times Q + \overline{S_Q} \tag{3.19}$$

Note that $\overline{S_P}$ and $\overline{S_Q}$ are not the P- and Q-residues of S because the term R in Equation 3.17 and Equation 3.18 is 2^n rather than $2^{n/2}$.

4.4 Combining Montgomery, Chinese Remainders, Blinding and Verification

We now have all of the ingredients for the high level specification of the RSA signature generator using blinding, verification, Chinese remainders and conversion between various residue forms. The task at hand is to combine these in such a way that:

1. blinding by U occurs before any private key operations;
2. inverse-blinding by V occurs after all of the private key operations;
3. all operations from the moment the message arrives to be signed to the moment the signature is made public use Montgomery reduction; and
4. as many operations as possible are performed on $n/2$-bit operands rather than n-bit operands.

The first two conditions are essential to maintain security against timing attacks such as the attack directly on the Chinese remainder operations ($M_P = M \bmod P$ and $M_Q = M \bmod Q$) proposed by Kocher [Koc96].

A solution that meets these objectives is shown in Figure 3.5 and Figure 3.6.

1. The Euler-Fermat Theorem (Fermat's Little Theorem): $a^p \equiv a \bmod p$ and if $\mathrm{GCD}(a, p) = 1$ then $a^{p-1} \equiv 1 \bmod p$.

Note that it has been possible to combine some of the operations. Blinding by U has been combined with conversion to N-residues. In effect, some of the computational effort has been shifted to the pre-computation phase in which $\overline{U}$ is evaluated instead of just U. Similarly the blinding step by V can be combined with conversion from N-residues.

Thus the evaluation of signatures using both blinding and N-residues is little more expensive than using either alone.

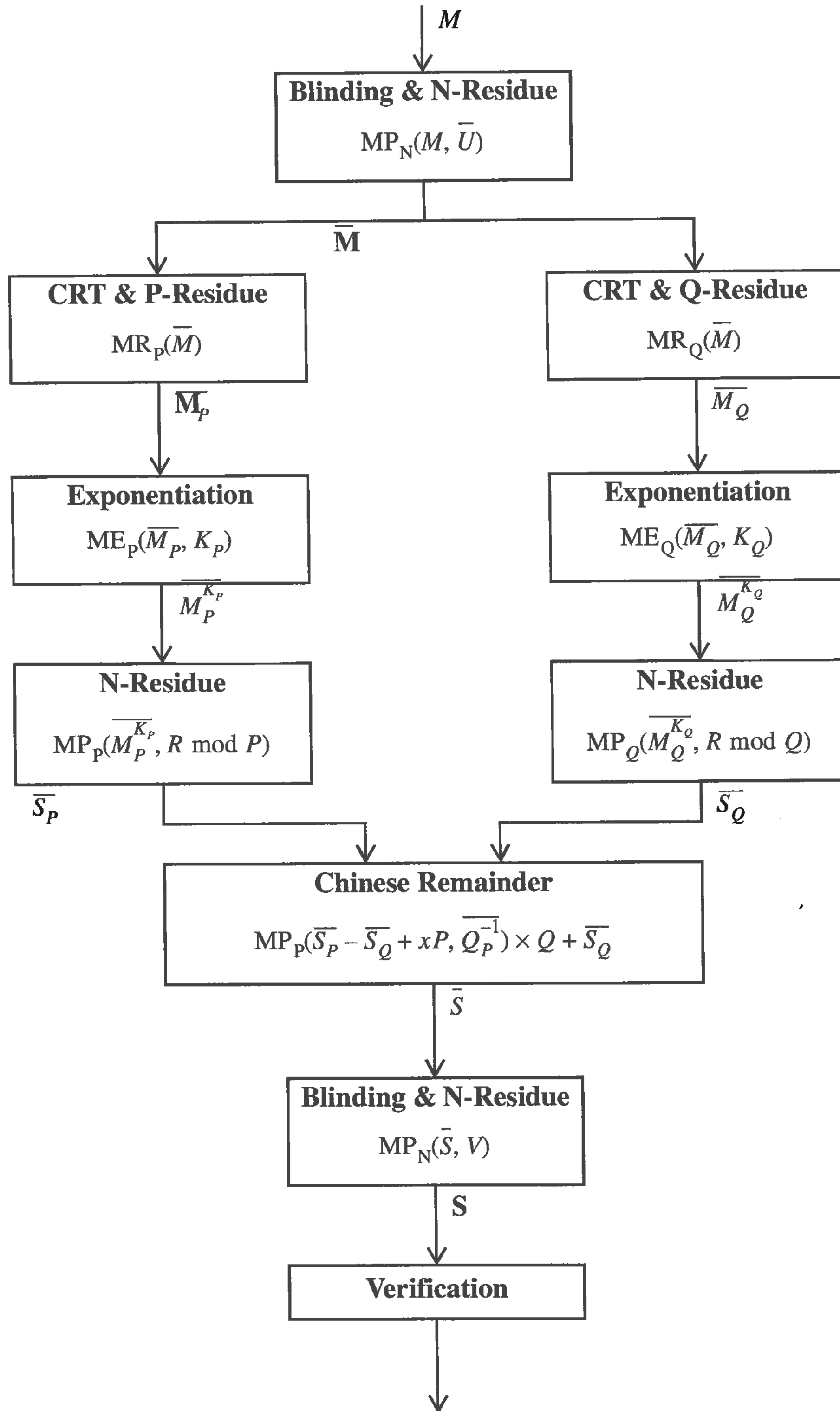

Figure 3.5 **A High Level Description of RSA Signature Generation.**

Pre-compute

$$V,\ U = (V^{-1})^{e} \bmod N,\ \bar{U} = U \times R^{2} \bmod N$$

$$R \bmod N = 2^{n} \bmod N,\ R \bmod P,\ R \bmod Q$$

$$K_P = K \bmod (P-1),\ K_Q = K \bmod (Q-1)$$

$$\overline{Q_P^{-1}} = Q^{-1} R' \bmod P \quad \text{(where } R' = 2^{n/2}\text{)}$$

Compute

$$\bar{M} = \mathrm{MP}_N(M, \bar{U}) = MUR \bmod N$$

$$\overline{M_P} = \mathrm{MR}_P(M) = MUR' \bmod P$$

$$\overline{M_P^{K_P}} = \mathrm{ME}_P\left(\overline{M_P}, K_P\right) = (MU)^{K_P} R' \bmod P$$

$$\overline{S_P} = \mathrm{MP}_P\left(\overline{M_P^{K_P}}, R \bmod P\right) = (MU)^{K_P} R \bmod P$$

$$\overline{M_Q} = \mathrm{MR}_Q(M) = MUR' \bmod Q$$

$$\overline{M_Q^{K_Q}} = ME_Q\left(\overline{M_Q}, K_Q\right) = (MU)^{K_Q} R' \bmod Q$$

$$\overline{S_Q} = \mathrm{MP}_Q\left(\overline{M_Q^{K_Q}}, R \bmod Q\right) = (MU)^{K_Q} R \bmod Q$$

$$\bar{S} = \mathrm{MP}_P\left(\left(\overline{S_P} - \overline{S_Q} + xP\right), \overline{Q_P^{-1}}\right) \times Q + \overline{S_Q} = (MU)^{K} R \bmod N$$

(with x chosen such that $\overline{S_P} - \overline{S_Q} + xP \geq 0$)

$$S = \mathrm{MP}_N(\bar{S}, V) = M^{K} \bmod N$$

Verification

$$\bar{T} = \mathrm{MP}_N(S, R^{2} \bmod N),\ \overline{T^{e}} = \mathrm{ME}_N(\bar{T}, e),\ T^{e} = \mathrm{MR}_N\left(\overline{T^{e}}\right), \text{ confirm that } T^{e} = M$$

Figure 3.6 RSA Signature Generation. The verification stage is required to confirm that the computed value of *S* is a correct signature for *M*

5 Summary and Conclusions

This chapter introduced public-key cryptography and some of the applications that make this technology invaluable to electronic commerce and communication. Subsequent chapters explore the

viability of public-key cryptography on 32-bit smart-cards through the implementation of 1024-bit RSA signature generation. This is a suitable benchmark as signature generation is a particularly useful operation for smart-cards and 1024-bit moduli represent a maximum level of security that will suffice for most applications into the next few years. It will also pose a significant challenge. A target time of 1 second was defined. This is reasonable given that there should only be a few 1024-bit operations per smart-card transaction.

RSA has been chosen as the benchmark cryptosystem but this does not mean that the algorithms developed in subsequent chapters are specific to RSA. Long integer operations such as multiplication, squaring and reduction are common to all trusted public-key cryptosystems. RSA is also one of the most computationally intensive of the signature schemes. If the target time can be met for RSA, then this bodes very well for the alternatives.

Algorithms for exponentiation were covered in Section 2. It was found that modular exponentiation is performed using a sequence of modular multiplications and squares. Recoding the exponent can reduce the total number of multiplications but does not usually reduce the number of squares. Various pre-computation and exponent recoding schemes have been proposed but for our purposes none perform much better on average than unsigned sliding window recoding ($USW_{2,m}$).

In Section 3 attacks on RSA implementations were considered. This is important as although RSA as theoretically specified is generally considered to be secure, it is possible to mount an attack based on the details of a specific implementation. The timing attack is of particular concern as it exploits the variations in evaluation time that result from many arithmetic optimisations. The fault attack, which exploits situations in which an implementation produces incorrect results, was also examined. It was found that combining two countermeasures—blinding and verification—it is possible to thwart the timing attack, fault attacks on latent or transient faults and a new combined attack. This approach is preferable to attempting to equalise evaluation times, a practice that rules out many algorithmic enhancements and which is beset with other difficulties.

In the last section, a high level specification of RSA signature generation was developed. This combined the accelerating techniques of Chinese reminders and Montgomery reduction with the security measures of blinding and verification. The combination of Chinese remainders with Montgomery operations, without converting back and forth between the various Montgomery residues forms, is not trivial and is not explained in any other publication of which I am aware. In the final procedure it has been possible to combine some of the residue transformations and some of the blinding operations, thereby reducing the total overhead.

From the results in this chapter we can conclude that the critical functions for our 1024-bit RSA signature implementation are 512-bit Montgomery multiplication and squaring.

Chapter 4

Modular Reduction

MODULAR REDUCTION is critically important for many public key cryptosystems. In fact, since its invention, public-key cryptography has been the prime motivation for the development of faster or more efficient mechanisms for long-wordlength modular arithmetic.

Few papers are published on modular reduction specifically. For cryptography, reduction is usually performed after each multiplication and hence the two operations are considered together as a modular multiplication. Indeed, most implementations interleave multiplication with reduction to reduce the size of intermediate results.

Even with interleaved schemes it is rarely difficult to isolate the reduction and multiplication algorithms. The latter is usually performed by a straightforward accumulation of partial products and hence it is the method of reduction that distinguishes one implementation from the next.

The aim of this chapter is to explore and develop algorithms for modular reduction. Algorithms for multiplication and squaring are examined in more detail in the next chapter.

Notation

This chapter considers the modular reduction $C = A \bmod N$ and often the modular multiplication $C = A \times B \bmod N$. The modulus N is assumed to be n digits long where each digit is in radix $r = 2^t$. When performing reduction, A is assumed to be up to $2n$ digits long. For a modular multiplication A and B are n digits long so that their product is $2n$ digits long.

1 Background

A sequence of surveys—Beth and Gollmann [BG89], Eldridge and Walter [EW93], Dhem [Dhe98] and Naccache and M'Raïhi [NM'R96]—have undertaken the classification of common reduction methods. The survey that follows owes much to these predecessors.

It is clear that modular reduction is closely related to division; however, division is usually performed to obtain the quotient whereas modular reduction is concerned only with the remainder. Of the reduction methods surveyed here, only those based on a sum of residues do not compute the useless quotient. The remaining methods proceed along the lines of a division:

$$Q = \left\lfloor \frac{A}{N} \right\rfloor \tag{4.1}$$

To find the remainder one of the following strategies can be employed:

1. Perform division to find the quotient Q or an estimate of it, then multiply by the modulus to find the remainder from $A \bmod N = A - QN$.
2. Use a division algorithm that generates the remainder as well as the quotient. These algorithms usually proceed by selecting the quotient digit-by-digit according to the value of a partial remainder. By the end of the division the partial-remainder has been reduced to the remainder.

1.1 An Overview of Modular Reduction

For the purposes of this survey, four categories of modular reduction have been identified:

1. reduction by sum of residues;
2. reduction using partial remainders;
3. reduction using multiplication by the inverse; and
4. reduction using Montgomery's method.

There is some correspondence between Montgomery's method and reduction using partial remainders – the significant difference being that the former operates from right to left and the latter from left to right. When interleaved with modular multiplication, the form of the three latter methods begins to look alike (with Montgomery again operating from the right). Note that these three categories apply equally well to division and reduction.

Enhancements

In 1993, Eldridge and Walter provided a list of ten techniques that could be used to accelerate modular multiplication [EW93]. Many publications since have applied these techniques in various combinations to the different categories of reduction. Paraphrased, the list is:

1. to deal with moduli of different lengths, shift them left so that the most significant digit of the implementation is always non-zero, and then adjust the operands and the result to correct for this change;
2. interleave multiplication and modular reduction so that the intermediate results remain approximately n digits long;
3. estimate the quotient from only the most significant bits of the partial remainder;
4. use a redundant representation to facilitate carry free addition and subtraction;
5. use a higher radix and hence reduce the number of iterations;
6. transform the operands in such a way that the quotient digit can be selected prior to accumulation of the current partial product;
7. pre-compute digit multiplies of the modulus and/or the multiplier or some linear combinations of these;
8. if factors of the modulus are know then apply the Chinese Remainder theorem so that calculation can be performed modulo the factors;
9. select quotient digits early enough that the generation (and possibly even accumulation) of the modulus multiple does not appear on the critical evaluation path; and
10. if A is not in non-redundant form then convert it 'on-the-fly' to simplify the formation of partial products.

To this list I may now add:

11. use the same redundant form for both operands and results so that conversion is not required during a modular exponentiation;
12. control quotient errors so that a single correction can be performed at the end of an exponentiation without overflow of intermediate results;
13. change the order in which operand digits are scanned;
14. transform the operands so that quotient selection becomes trivial;
15. change the degree of interleaving;
16. recode the operands using a digit set conversion to reduce the arithmetic weight;
17. work in residue number systems for fast multiplication and addition eg. [HP94]; and
18. use new methods of multiplication such as the discrete Fourier transform eg. [Knu97], [Zur93] (a

technique that is only likely to become effective for moduli of many thousands of bits).

A table summarising the algorithmic features of many published theoretical and fully implemented modular multiplication systems is presented in Appendix A. Unlike many other surveys, the details of an implementation's performance are ignored in favour of its algorithmic characteristics.

1.2 Reduction by Sum of Residues

Of the four categories of reduction, sum of residues has received the least attention in recent years; it does not even appear in some surveys although it is certainly a distinct method in its own right. Sum of residues reduction is based on Equation 4.2.

$$C \bmod N = \left(\sum_{i=0}^{2n-1} c_i (r^i \bmod N) \right) \bmod N \tag{4.2}$$

The *residues* are the n-digit values $r^i \bmod N$. A final reduction is required as the sum of the residues may exceed n-digits length. This simply involves a few corrective subtractions of the modulus.

The residues can be computed recursively using Equation 4.3 as in [FJ90] or pre-computed and stored in a table as in [KH88].

$$\text{If } r_i = r^i \bmod N \text{ then } r_{i+1} = (r \times r_i) \bmod N. \tag{4.3}$$

If C is the result of a multiplication then sum of residues reduction can be interleaved with evaluation of C from the right using the product scanning form of multiplication (Section 1.1 of Chapter 5). In the i-th iteration: c_i is generated and a few digits of carry information must be stored; the residue r_i is computed from r_{i-1} using Equation 4.3; and $c_n r_n$ is accumulated into the partial result. However, even with this interleaving, at least $2n$-digits of storage are still required for intermediate results: n for the residue r_i and n for the partial result.

Sum of Multiplicand Residues

Modular multiplication can be performed using a sum of pre-computed multiplicand residues as in Equation 4.4. Once again the sum may exceed N by a few digits and require a final modular reduction.

$$AB \bmod N = \left(\sum_{i=0}^{n-1} a_i (Br^i \bmod N) \right) \bmod N \tag{4.4}$$

This technique is used in [QC82], [KTS91] and [HOY96] with a table of pre-computed residues $Br^i \bmod N$. A higher radix version is used in [SM89] with a pre-computed table of residues $a_j Br^i \bmod N$ for all of the possible digits a_j at all significances r^i.

Tomlinson's Method

Interleaved modular multiplication using pre-computed residues is presented in [Tom89]. A pseudo-code algorithm is shown in Algorithm 4.1. During each iteration the intermediate result is allowed to grow by 2 bits. At the end of an iteration these two bits are set to zero. The residue corresponding to the reset bits is added at the next iteration. This interesting (and largely overlooked) method is applicable to higher radix implementation with significantly fewer stored residues than the previous methods in this section. In addition, the residues only depend on the modulus and can therefore be extensively re-used in an RSA system.

Algorithm 4.1 Interleaved Sum of Residues Modular Multiplication. The result A*B mod N is returned in C. Following the last iteration C may be 2-bits larger than N and require a corrective subtraction.

```
C = 0, q = 0
for i = n - 1 down to 0
    C = 2C + a_iB
    C = C + (q*2^n mod N)       /* the residue q*2^n mod N is pre-computed */
    q = C>>n                    /* q is the upper 2 bits of C */
    C = C and (2^n - 1)         /* set the upper 2 bits of C to zero */
next i
```

Right to Left Method

In [FDG90] multiplication occurs from right to left as in Algorithm 4.2. This algorithm can be understood as a sum of multiplicand residues with the residues $Br^i \bmod N$ being calculated recursively.

An interesting feature is that calculation of the next residue can be performed in parallel with the accumulation of the current residue (the two operations in the loop are executed in parallel). This will be of some benefit provided reduction is simple due to the smaller magnitude of the intermediate results $(C + a_i B)$ and $2B$. This benefit comes at the cost of two parallel execution units. Operand scaling (Section 1.3) and quotient pipelining (Section 1.5) provide greater benefit at reduced cost.

Algorithm 4.2 Right to Left Sum of Residues Modular Multiplication. The result A * B mod N is returned in C.

```
for i = 0 to n-1
    C := (C + a_iB) mod N
    B := 2*B mod N
next i
```

1.3 Reduction using Partial Remainders

Consider the form of long division as learned in school. The most significant digits of a partial remainder are examined to determine a digit of the quotient. The partial remainder is then updated by subtracting a multiple of the divisor. This is also the general form of the classical division algorithms: binary restoring division (sometimes called paper and pencil long division) and non-restoring division.

Early implementations of modular multiplication interleaved binary division with left to right multiplication along the lines of Algorithm 4.3 [Slo85].

Algorithm 4.3 Left to Right Modular Multiplication With Partial Remainders. The result A*B mod N is returned in C. Quotient digit selection may involve a full length subtraction to compare the magnitude of N and C.

```
C = 0
for i = n - 1 down to 0
    C = 2C
    q = quotient_digit_select(C, N)
    C = C - qN                          // modulus subtraction
    C = C + a_iB                        // partial product accumulation
    q = quotient_digit_select(C, N)
    C = C - qN
next i
```

There are two equivalent ways of thinking about this process. Either one long division is performed in which quotient digits are selected from the most significant bits of the product as they are generated, or many small divisions are performed to reduce the partial result at each iteration. Papers that take the latter approach do not always use the language of division and it is not immediately obvious that they belong to the category of 'reduction using partial remainders'. However, the form of the algorithm is ultimately the same, and converges on the same enhancements.

This binary algorithm can be improved with a variety of techniques: the number $(B-N)$ can be pre-computed so that modulus subtraction is combined with partial product accumulation; quotient digits -1, 0 or 1 can be selected with C temporarily allowed to become negative; Booth or higher radix multiplication can be used to decrease the number of partial product accumulation steps; and asynchronous adders can be used to reduce the average delay due to carry propagation. Papers that use these techniques in various combinations include [Bla83], [ORS+86], [ICHO89] and [TB91].

Brickell's Method

Brickell's method of reduction [Bri83] (corrected in [WE90]) contains three significant enhancements: parallel (redundant) adders are used, quotient digits are estimated from the most significant digits of the partial remainder, and quotient digits are selected such that there is only a single modulus subtraction per iteration.

A large number of papers that use these techniques in various combinations are surveyed in Appendix A. A significant number of implementations, even those published after [Bri83], use parallel addition yet require more than one modulus subtraction for each iteration. Quotient estimation usually accompanies parallel addition; however there are situations, such as in software, where parallel addition is not appropriate but quotient estimation is still of benefit (eg. [Tak93]).

Division using redundant partial remainders and quotient digit estimation is often called SRT division and attributed to Robertson [Rob58] and Atkins [Atk68]. Further development of Brickell's reduction has closely followed that of SRT division. Enhancements include the use of higher radices and simplified quotient digit selection. See, for example, [VVDJ90], [Mor90], [OK91] and [Tak92].

Algorithm 4.4 Improved Left to Right Modular Multiplication. Brickell's method allows for a single reduction per iteration with quotient digit selection independent of the current partial product a_iB. Addition is performed by carry free adders and the quotient digit is selected from only the most significant bits of the redundant partial result. Later implementations use a higher radix for the digits a_i and q_i to reduce the number of iterations.

```
C = 0
for i = n - 1 down to 0
    q_i = quotient_digit_select(C)
    C = C<<m + a_i B + q_i N
next i
```

Walter in [Wal92] applies the technique of operand scaling to achieve another substantial enhancement to the reduction algorithm. The modulus is scaled by an integer multiple such that its most significant digits are fixed to some desirable value. This provides a two benefits. Firstly,

quotient digit selection becomes independent of the modulus and this simplifies the selection logic. Secondly, the fixed digits of the scaled modulus can be chosen to further simplify the quotient digit selection function. For example, if quotient digit selection involves division by the most significant part of the modulus, then setting this part to a multiple of the radix will allow the division to be performed by a shift.

The cost of scaling the modulus in this way is a few extra subtractions at the end of a modular multiplication (or at the end of an exponentiation) to fully reduce the result to less than the unscaled modulus.

By shifting the operands to make the quotient selection independent of the current partial remainder and then scaling the modulus, Walter sufficiently simplifies quotient selection to shift it and the subsequent modulus multiple generation from the critical evaluation path of his implementation.

Sedlak's Method

Modular multiplication in Sedlak's cryptography processor [Sed88] fits the general interleaved scheme of Algorithm 4.4 but makes use of variable length shifts for both multiplication and reduction.

In each iteration, a multiplier digit $a_i \in \{-1, 0, 1\}$ and an offset s_a (from 0 to k-bits) are chosen according to a signed binary conversion of the multiplier. At the same time a quotient digit $q_i \in \{-1, 0, 1\}$ and an offset s_r (from 0 to k-bits) are also chosen. These are obtained by comparing the partial remainder with pre-computed values $N/3, N/6, \ldots, (2^{1-k}) N/3$.

The partial remainder is updated by accumulation of $a_i B$ at significance determined by s_a and of $q_i N$ at significance determined by s_r. The algorithm is such that the magnitude of the partial remainder stays within known bounds.

In the ideal case (with k very large), both the multiplication and the reduction take $n/3$ iterations to complete. Thus on average, the modular product should be ready in $n/3$ iterations. Two implementation issues reduce performance below this ideal: the shift at each iteration is limited to k-bits; and to reduce the cost of the comparisons, only the most significant bits of the partial remainder are examined.

Sedlak's implementation makes use of n-bit registers, a fast n-bit adder and an n-bit barrel shifter.

Of all of the papers surveyed, this is the only one to employ sliding window conversion for reduction ([KH92] use sliding windows for multiplication and others use sliding windows for exponentiation).

1.4 Reduction using Multiplication by the Inverse

If the inverse of N is pre-computed then the quotient of A/N can be found by multiplication:

$$Q = \lfloor A/N \rfloor = \lfloor A \times N^{-1} \rfloor .$$

The difficulty here is that N^{-1} is a real number. To perform this multiplication in integer arithmetic will require a special representation.

A first possibility is to scale N^{-1} by some power of two and then approximate the result by rounding off. Barrett, in [Bar87], observes that there is a clear trade-off. The more accurately N^{-1} is represented the longer it will take to perform the multiplication to find Q. Less accurate representation leads to a greater error in Q and hence to errors in that must be corrected at the end of the reduction.

Given Q, reduction can be performed using Equation 4.5.

$$A \bmod N = A - QN . \tag{4.5}$$

Reduction can be interleaved with multiplication by reducing the partial result at each iteration. The algorithm that results is of the same form as that for reduction using partial remainders (Algorithm 4.4) with the only difference being the rules for quotient digit selection.

The following sections discuss methods that use multiplication by a pre-computed integer $\lfloor 2^{\alpha}/N \rfloor$ to find the quotient. Similar methods appear in [NS81], [PP90], [AM91], and [Wal94].

Barrett's Method

The method introduced by Barrett in [Bar87] is generalised and extended in [Dhe98].

The quotient can be found exactly from:

$$Q = \left\lfloor \frac{A}{N} \right\rfloor = \left\lfloor \frac{\frac{A}{2^{n+\beta}} \frac{2^{n+\alpha}}{N}}{2^{\alpha-\beta}} \right\rfloor . \tag{4.6}$$

Instead of computing this, the quotient can be estimated as:

$$Q' = \left\lfloor \frac{\left\lfloor \frac{A}{2^{n+\beta}} \right\rfloor \left\lfloor \frac{2^{n+\alpha}}{N} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor . \tag{4.7}$$

The estimate of the quotient is therefore found by multiplying the most significant bits of A by the pre-computed constant $\lfloor 2^{n+\alpha}/N \rfloor$.

An analysis of the error in the estimate reveals that the parameters α and β can always be selected to reduce the error in the quotient to at most 1. With careful design only one correcting subtraction is required at the end of a reduction, or even at the end of a modular exponentiation.

Quisquater's Method

Quisquater's reduction algorithm follows a similar form to Barrett's except that the modulus N is normalised to N' in such a way that quotient digit selection becomes trivial.

It is reported [Dhe98] that this algorithm was first presented by Quisquater at the rump session of Eurocrypt 90; however, it does not appear in the published proceedings [Wal92]. The idea of scaling of the modulus is also used in the operand scaling technique for reduction using the partial remainder as discussed in Section 1.3.

Starting with Equation 4.7 and substituting $c = \alpha = \beta$ gives the following equation to reduce A:

$$A \bmod N \approx A - \left\lfloor \frac{A}{2^{n+c}} \right\rfloor \left\lfloor \frac{2^{n+c}}{N} \right\rfloor N.$$

However, if Q' is chosen according to Equation 4.8 then the reduction is performed with a new modulus N' given in Equation 4.9.

$$Q' = \left\lfloor \frac{A}{2^{n+c}} \right\rfloor \tag{4.8}$$

$$N' = \left\lfloor \frac{2^{n+c}}{N} \right\rfloor N \tag{4.9}$$

The advantage of this approach is that the new quotient is trivially obtained from the most significant bits of A. The disadvantage is that the final result is modulo N' and must be corrected according to Equation 4.10. This is no great cost for RSA as exponentiation can be performed modulo N' with a single correction occurring at the very end.

$$\text{If } C' = A \bmod N' \text{ then } C = \frac{\left\lfloor \frac{2^{n+c}}{N} \right\rfloor C' \bmod N'}{\lfloor 2^{n+c}/N \rfloor} = A \bmod N. \tag{4.10}$$

Further enhancements to this algorithm include simplified computation of the normalisation factor and the use of alternative normalisation factors to reduce the size of the normalised modulus. As was the case with Barrett's algorithm, the parameters can be designed so that the error in quotient selection is bound and hence intermediate results remain within a known length and only require a single corrective subtraction at the end of an exponentiation [Dhe98].

1.5 Montgomery Reduction

Montgomery's reduction method [Mon85] is a re-discovery of Hensel's odd division method [SV93]. It follows the general pattern of division using partial remainders (Section 1.3) with an important distinction: the partial remainder is reduced from the least significant digit first. Algorithm 4.5 gives the scheme in pseudo-code.

Algorithm 4.5 Montgomery Reduction. A 2n-digit number A is reduced to n digits. The result A / r^n mod N is returned in C.

```
C = A
for i = 0 to n - 1
    qi = quotient_digit_select(C, N)
    C = C + qi * N * r^i
next i
C = C / r^n                // shift right by n digits
if C >= N
    C = C - N
end if
```

The idea behind this algorithm is to choose a quotient digit q_i such that accumulation of $q_i N r^i$ will set i^{th} digit of C to zero. Then, prior to the shift step, the lower n digits of C are all zero.

Consider the least significant digits of the modulus multiple accumulation: $c_i \leftarrow c_i + q_i n_0 \bmod r$. Selection of $q_i = -c_i / n_0 \bmod r$ will therefore set c_i to zero. The constant $n'_0 = -1/n_0 \bmod r$ can be pre-computed so that quotient digit selection becomes a simple digit by digit multiplication [DK91]. An example of Montgomery reduction is given in Figure 4.1.

Note that the result of Montgomery reduction by N is not $A \bmod N$ but the n-bit integer given in Equation 4.11.

$$\mathrm{MR}_N(A) = A/R \bmod N \tag{4.11}$$

To obtain the correct value of $A \bmod N$ using Montgomery reduction, either the result must be corrected or the inputs must be transformed so that the correct result is produced. In Section 4.1 of Chapter 3 the N-residue of A was defined as $\bar{A} = AR \bmod N$. This has the useful property that the Montgomery reduced product of two N-residues is the N-residue of the product: $MR_N(\bar{A} \times \bar{B}) = \overline{ABR \bmod N}$. This leads to the definition of Montgomery multiplication and Montgomery exponentiation given in Equation 3.13 and Equation 3.14.

$n_0 = 7$, $n_0' = 7$	$N =$				2	4	7
	$A =$	3	6	6	4	2	3
$c_0 = 3$	$C =$	3	6	6	4	2	3
$q_0 = 3 \text{x} 7 \bmod 10 = 1$	$+q_0Nr$				2	4	7
$c_1 = 7$	$C =$	3	6	6	6	7	0
$q_1 = 7 \text{x} 7 \bmod 10 = 9$	$+q_1Nr^1$		2	2	2	3	0
$c_2 = 9$	$C =$	3	8	8	9	0	0
$q_2 = 9 \text{x} 7 \bmod 10 = 3$	$+q_2Nr^2$		7	4	1	0	0
	$C =$	4	6	3	0	0	0
	$-N =$	2	4	7			
	$MR_N(A) =$	2	1	6			

Figure 4.1 An Example of Montgomery Reduction. The 6-digit decimal number A is Montgomery reduced modulo the 3-digit number N.

A more thorough-going derivation of Montgomery reduction is provided in the following section.

Mathematical Foundation

The Montgomery reduction function is defined in Equation 4.11. Equation 4.12 to Equation 4.14 describe the implementation of this function.

$$Q = (A \bmod R) N' \bmod R \text{ where } N' = -N^{-1} \bmod R \tag{4.12}$$

$$C = (A + QN) / R \tag{4.13}$$

$$\text{if } C \geq N \text{ then } MR_N(A) = C - N \text{ otherwise } MR_N(A) = C \tag{4.14}$$

That C is an integer can be confirmed by observing that $QN \equiv ANN' \equiv -A \bmod R$ and hence $A + QN$ is a multiple of R.

That $C \equiv A/R \bmod N$ is clear given that Equation 4.13 implies $CR \equiv A \bmod N$.

Finally, if the reduction starts with $A < RN$ then $Q < R$ and $A + QN < RN + RN$ and the result $C < 2N$. That is, C can be reduced modulo N with at most one subtraction.

It is usual to choose R to be a binary power so that Equation 4.13 can be implemented with a binary shift to the right. Note that for N' to exist we must have $\mathrm{GCD}(N, R) = 1$ and this will always be the case when R is a binary power and N is odd.

Simplifying Quotient Digit Selection

The Montgomery quotient digit selection function is of the form $q_i = c_i n'_0 \bmod r$ where c_i is the digit of the partial remainder to be reduced. Note that whenever $n'_0 = 1$, quotient digit selection becomes trivial: $q_i = c_i$. Two methods have been proposed to ensure this condition is met.

In [Wal93], [Wal95] and [Oru95] the modulus N is transformed to $\tilde{N} = n'_0 N$ so that $\tilde{n}'_0 = 1$. Reduction is performed modulo $\tilde{N}$, an integer multiple of N. Both the modulus and the result may be $n+1$ digits long and hence an extra corrective subtraction may be required at the end of the reduction.

In [Wal95] it is observed that rather than transform the modulus, the modulus can be specifically chosen such that $n'_0 = 1$. For RSA, N is the product of two random primes, P and Q. Selection of N can proceed by first choosing a random P and then trying random primes Q until the condition for trivial quotient digit selection is met.

Quotient Pipelining for Montgomery Multiplication

An interleaved hardware Montgomery multiplier for $C = \mathrm{MR}_N(A, B)$ can be constructed according to the iteration described in Equation 4.15 and Equation 4.16. In iteration i the partial product $a_i B$ is accumulated along with a modulus multiple $q_i N$. The modulus multiple is selected such that the ith digit of the partial result is zero.

$$q_i = (C_i + a_i B \bmod r) \times n'_0 \bmod r \tag{4.15}$$

$$C_{i+1} = \frac{C_i + q_i N + a_i B}{r} \tag{4.16}$$

Quotient digit selection as described by Equation 4.15 can be a bottleneck on the critical path of the evaluation. This is because of the time required to perform the multiplication and addition in Equation 4.15 but especially because of the dependence of q_i upon the current partial result C_i.

Elimination of the multiplication from quotient digit selection can be achieved using the methods described in the previous section. In addition to this, if the multiplier B is replaced with $\tilde{B} = rB$ then the quotient selection function becomes trivial $q_i = C_i \bmod r$ [Wal93], [Wal95], [Oru95], [EW93], [Kor93b].

With these changes, Equation 4.16 becomes:

$$C_{i+1} = \frac{C_i + q_i\tilde{N} + a_i\tilde{B}}{r}.$$

If in this equation, the least significant digits of the new partial result are unchanged except by shifting—that is $C_{i+1} \bmod r^d = C_i/r \bmod r^d$—then the selection of the subsequent quotient digits $q_{i+1}\ldots q_{i+d}$ is independent of q_i. Thus generation and accumulation of $q_i\tilde{N}$ need not be complete for another d iterations.

This idea was introduced in [SV93] where it was called *quotient pipelining* and was achieved by selecting q_i to be $(d+1)$-digits long. The intermediate results require d extra bits of precision and d extra iterations must be performed. The implementation in [SV93] benefits from a four times improvement in speed due to quotient pipelining alone.

[Wal95] and [Oru95] achieve quotient pipelining without a longer quotient digit by transforming the modulus such that $\tilde{N} \bmod r^{d+1} = 1$.

Transformation Operations

The need to transform operands to and from N-residue form is an oft cited drawback of Montgomery multiplication. Certainly it is a significant issue for isolated modular multiplications, especially if full N-residue conversion of both operands is employed. This is the case in Equation 4.17, in which the product AB takes three Montgomery multiplications and a Montgomery reduction to evaluate.

$$\bar{A} = \mathrm{MP}_N(A, R^2),\ \bar{B} = \mathrm{MP}_N(B, R^2),\ \bar{C} = \mathrm{MP}_N(\bar{A}, \bar{B}),\ C = \mathrm{MR}_N(\bar{C}). \tag{4.17}$$

In Chapter 3 conversion to and from N-residues is combined with blinding such that the cost of the Montgomery transformations is almost entirely eliminated. The technique that was used to achieve this can also be used to reduce the cost of an isolated modular multiplication. Equation 4.18 shows that modular multiplication can be performed with two Montgomery multiplications provided $R^2 \bmod N$ is pre-computed.

$$\bar{A} = \mathrm{MP}_N(A, R^2),\ C = \mathrm{MP}_N(\bar{A}, B). \tag{4.18}$$

A similar method is described in [OM98]. The cost is reduced to that of a full multiplication and a Montgomery multiplication.

$$T = A \times B,\ T_H = \mathrm{MP}_N(T \operatorname{div} R, R^2),\ C = (T_H + T) \bmod R. \tag{4.19}$$

Other Developments

In [DK91] and [Eld91] the results of Montgomery multiplication are not fully reduced to n-bits until the very last multiplication of a modular exponentiation. It is shown that the intermediate results stay within known bounds.

In multiple-precision multiplication, the multiplier and multiplicand digits can be scanned in a number of different orders. For example in [DK91] the convolution-sum method of multiplication is used to reduce the number of shifts. It has already been observed that multiplication and reduction can be separated or interleaved. Extending this, the authors of [KAK96] examine both *coarsely* and *finely integrated* (interleaved) reduction. In the former the partial product is reduced after processing an array of words and in the latter reduction is performed after each word. The scanning order and the level of interleaving are independent and thus the authors are able to examine a variety of different combinations. Each of the methods requires a different number of additions, reads and writes and exhibits different loop overheads. The optimal choice for a particular implementation will depend on the relative cost of these operations.

Kaliski in [Kal95] develops an algorithm to find the Montgomery inverse $A^{-1}r^{n} \bmod N$ directly. This is found to be faster than the usual algorithm for computing a modular inverse and hence significantly faster than computing the modular inverse and then converting to N-residue form.

In Chapter 3 a method was developed to incorporate the N-residue transformations required by a Montgomery system with the security operation of blinding. An alternative method of removing the N-residue transformations is developed in [NM'RR95]. In this case the authors propose a change to the RSA cryptosystem in which Montgomery operations are used with right-to-left exponentiation directly on the plaintext to produce the modified cyphertext $C = M^{e}/R^{e-1} \bmod N$. The plaintext can be recovered as $M = C^{K}/R^{K-1} \bmod N$.

1.6 Comparing Reduction Algorithms

Comparing reduction methods is not a straight-forward exercise. That a case can be made for each of the methods is borne out by a survey of leading cryptographic smart-cards [NM'R96] in which the authors find at least 6 different classes of algorithm in a sample of 13 implementations.

There are similarities between all four categories of reduction: at each iteration they all accumulate an approximately n-digit multiple into the partial result; they can all be interleaved with multiplication to reduce intermediate storage to n-bits; and in all cases techniques exist to make quotient digit selection trivial.

The three most common categories of reduction—using partial remainders, multiplication by the inverse, and Montgomery reduction—have all evolved to the point where quotient digit selection and modulus multiple generation can be moved from the critical path of a hardware implementation. The critical path of the reduction becomes the accumulation of partial products and modulus multiples. From this it is clear that modular multiplication is roughly equivalent in complexity to ordinary multiplication. That the two are equivalent in time, memory and hardware complexity is reported to have been shown by Arazi [NM'RR95].

Comparisons between various modular reduction schemes are presented in, for example, [Kor93b], [Dhe98], [BGV93], [EW93], [HOY96] and [Wal93]. There is little by way of consensus in their conclusions. This is not too surprising as the algorithms are compared on a wide variety of different platforms, at different stages in their relative evolution and with different measures of performance.

In hardware, Montgomery reduction has one indisputable advantage over the other methods. At each iteration, the Montgomery method selects the quotient from the least significant digits of the partial result. This can be done before the carry propagation from previous iterations is complete. Thus a Montgomery multiplier can use small carry propagate adders without any detriment to overall performance. To achieve equivalent performance the left to right schemes must use larger redundant adders.

N-residue transformations are a disadvantage of Montgomery reduction. However for RSA, the cost of these conversions is relatively small and, as discussed in Chapter 3, the cost can be reduced further by combining N-residue transformations with blinding operations.

Comparisons are no more easy to make in software where the characteristics of a given processor may lend themselves more to one algorithm than another; the degree of interleaving and the order in which operands are processed have a significant influence on the number of memory accesses and loop overhead; pre-computation, instruction set and the style of programming—whether high level and portable, or low level and optimised—will also favour different algorithms. Nonetheless, the results of the comparison from [Dhe98] are worth repeating and are presented in Table 4.1.

Type of Operation	Barrett	Quisquater	Montgomery	$n = 16$ Barrett	Quisquater	Montgomery
Elementary	$5n+1$	$5n+6$	$5n+3$	81	86	83
Multiplication	$2n^2-n+2$	$2n^2+3n+2$	$n^2+\frac{5}{2}n$	498	562	296
Read	$3n^2-2n+4$	$3n^2+5n+6$	$3n^2+11n+1$	740	854	945
Store	n^2+3n+1	n^2+5n+4	n^2+4n	305	340	320
Loop	n^2+1	n^2+2n+2	n^2+3n+1	257	290	305

Table 4.1 Number of Operations for 3 Modular Multiplication Algorithms on a 32-bit Processor. Values are shown for $n = 16$ (i.e. 512-bits) as this is the most common operation for 1024-bit RSA using Chinese remainders [Dhe98].

The diversity of the factors that influence these figures mean that we should be careful in drawing conclusions. Along with the results from [BGV93], they confirm that none of the algorithms is superior in all circumstances. The figure for the number of multiplications is of some interest. In [Dhe98], the author constructs a smart-card software library using the Quisquater algorithm for a modified ARM7M processor. In order to achieve adequate performance with the large number of multiply instructions, a modified fast hardware multiplier is used. This multiplier produces a 64-bit result in only 4 cycles and operates in parallel with subsequent instructions.

An alternative approach is taken in Chapter 6 where it is assumed that no fast multiplier is available. From Table 4.1 it appears that Montgomery's algorithm is likely to be superior under these circumstances.

The following sections consider the implementation of Montgomery reduction in software. Many of the techniques discussed, in particular the use of sliding windows and pre-computation, are applicable to other reduction techniques as well. Reduction using multiplication by the inverse would benefit from the optimised multiplication routines of Chapter 5. Division using partial remainders could be enhanced using the techniques such as those applied to SRT in Section 1 of Chapter 2.

2 Recoded Montgomery Reduction

In this section the principles of sliding window digit set conversion from Chapter 3 are applied to Montgomery reduction. At this stage reduction will be kept separate from multiplication.

Algorithm 4.6 shows Montgomery reduction adjusted for quotient digit selection using sliding-windows. It proceeds from the right, considering the ith digit of the partial result c_i. If this digit is zero then it is skipped and the algorithm moves on to c_{i+1}. However, if c_i not zero then a multiple of the modulus is added to C such that all the digits c_i to c_{i+m-1} become zero and the algorithm can move on to c_{i+m}.

Algorithm 4.6 Sliding Window Montgomery Reduction. A 2n-digit number A is reduced to n-digits with the result A / r^n mod N returned in C.

```
C = A, i = 0
while i < n
    if c_i <> 0 then
        q_i = quotient_digit_select(C, N)
        C = C + q_i * N * r^i        // c_(i+m-1)...c_i are set to zero
        i = i + m
    else
        i = i + 1                    // skip zero digits
    end if
end while
C = C / r^n                          // shift right by n digits
if C > N then C = C - N end if
```

2.1 Montgomery Reduction with Unsigned Sliding Windows

The quotient digit selection function in Algorithm 4.6 must choose a multiple of the modulus that when accumulated will set digits c_i to c_{i+m-1} to zero. This condition is expressed in Equation 4.20.

$$\tilde{c}_i + q_i\tilde{n}_0 \bmod r^m = 0 \qquad (4.20)$$

$$\text{where } \tilde{c}_i = \sum_{j=0}^{m-1} c_{i+j}r^j \text{ and } \tilde{n}_0 = \sum_{j=0}^{m-1} n_j r^j .$$

This can be understood as a Montgomery reduction step in radix r^m with $\tilde{c}_i$, $\tilde{n}_0$ and q_i being digits in this higher radix. To satisfy Equation 4.20 the quotient digit can be selected as:

$$q_i = \tilde{c}_i\tilde{n}'_0 \bmod r^m . \qquad (4.21)$$

For the sliding window algorithm above, is possible to prove that there is a one-to-one relationship between $\tilde{c}_i$ and q_i with both taking on all values in the set $\{y \text{ s.t. } 0 < y < r^m, y \bmod r \neq 0\}$. This is the same digit set as $USW_{r,m}$.

Moreover, there is a direct connection between the selection of digits $\tilde{c}_i$ in Algorithm 4.6 and $USW_{r,m}$. Proceeding from the right zero digits in C are skipped until a non-zero digit c_i is found. This and the subsequent m digits are grouped together to form $\tilde{c}_i$. This corresponds to a non-zero digit q_i. The multiple q_iN is then accumulated into C scanning continues with c_{i+m}. If C was randomly selected at the beginning, then the digits to the left of c_{i+m-1} will still be random following the accumulation. The USW state diagram of Figure 2.13 in Chapter 2 applies equally well to this reduction algorithm. Hence the result concerning average number of non-zero digits for $USW_{r,m}$ is also true for the average number of non-zero quotient digits.

The average number of non-zero quotient digits for long wordlength n is:

$$\frac{n}{m + 1/(r-1)} . \tag{4.22}$$

An example of an iteration of Montgomery reduction using unsigned sliding windows is given in Figure 4.2.

$N =$	...	...	...	...	...	1	1	3	$\tilde{n}_0 = 13, \tilde{n}'_0 = 13$
		At iteration *i*...							
	...	c_{i+3}	c_{i+2}	c_{i+1}	c_i	c_{i-1}	...	c_0	
$C =$	...	1	0	2	2	0	...	0	$\tilde{c}_i = 22, q_i = \tilde{c}_i\tilde{n}'_0 \bmod r^m = 22$
$+q_iNr^i$	...	3	2	1	2	0	...	0	
$C' =$	...	0	3	0	0	0	...	0	

Figure 4.2 An Example of Montgomery Reduction using USW. The example shows a single iteration of Montgomery reduction using $USW_{4,2}$. The digits c_i and c_{i+1} are set to zero through the selection of the quotient digit q_i and accumulation of q_iNr^i. All digits are in radix 4.

2.2 Montgomery Reduction with Signed Sliding Windows

To apply a signed sliding window conversion to Montgomery reduction, start by considering a quotient digit q_i selected according to the unsigned sliding window technique of the previous section. Equation 4.23 considers the effect of selecting an alternative signed quotient digit $q'_i = q_i - r^m$.

$$\tilde{c}_i + q'_i\tilde{n}_0 \bmod r^m = \tilde{c}_i + \left(q_i - r^m\right)\tilde{n}_0 \bmod r^m = \tilde{c}_i + q_i\tilde{n}_0 \bmod r^m = 0 \tag{4.23}$$

This shows that whenever the unsigned digit q_i is selected, the signed digit q'_i could be used equally well.

Quotient digit selection corresponding to $MSW_{r, m, r^m/2}$ can be achieved if whenever $q_i > r^m/2$ the negative quotient digit q'_i is selected instead. This gives the average number of non-zero quotient digits as in Equation 4.22. However this time the quotient digits are from the balanced digit set:

$$\{q_i \text{ s.t. } (-r^m/2) < q_i < (r^m/2), q_i \neq 0 \bmod r\} .$$

An example of an iteration of Montgomery reduction using modified sliding windows $MSW_{r, m, r^m/2}$ is given in Figure 4.3.

Note that for $r = 2$, $MSW_{2, m, r^m/2}$ and $SSW_{2, m}$ are identical. For higher radices, $SSW_{r,m}$ achieves a lower average arithmetic weight than $MSW_{2, m, r^m/2}$ but at the cost of a larger digit-set. This conversion will be considered in the next section.

$N =$	..					1	1	3	$\tilde{n}_0 = 13$, $\tilde{n}'_0 = 13$
	..	c_{i+3}	c_{i+2}	c_{i+1}	c_i	c_{i-1}	..	c_0	
$C =$	..	1	0	2	2	0	..	0	$q_i = 22 > r^m/2$
$+q'_iNr^i$	..	2	0	2	2	0	..	0	Choose $q'_i = q_i - r^m = -12$.
$C' =$	..	3	0	0	0	0	..	0	

Figure 4.3 An Example of Montgomery Reduction using MSW. The example shows a single iteration of Montgomery reduction using $MSW_{4,2,8}$. The digits c_i and c_{i+1} are set to zero through the selection of the quotient digit q_i and accumulation of q_iNr^i. All digits are in radix 4.

Montgomery Reduction with SSW

In $SSW_{r,m}$ the digit $0 < y_i < r^m$ is selected such that the subsequent $(m-1)$ -digits $y_{i+1} \ldots y_{i+m-1}$ are all zero. On occasion it is possible to select $y_i < 0$ to ensure that y_{i+m} is also zero.

In the sliding window Montgomery reduction of Algorithm 4.6 the quotient digit $0 < q_i < r^m$ is selected from Equation 4.21 such that $q_{i+1} \ldots q_{i+m-1}$ are all zero. Under what circumstances should q_i be set negative ($q_i \leftarrow q_i - r^m$) to arrange that q_{i+m} is also zero whenever possible?

Given q_i one can either add $C' = C + q_iNr^i$, or subtract $C' = C - (r^m - q_i)Nr^i$. First consider the addition as in Figure 4.4 where $E = q_iNr^i$.

$$\begin{array}{cccccccccc} & c_{i+m} & c_{i+m-1} & .. & c_{i+1} & c_i & 0 & .. & 0 \\ + & e_m & e_{m-1} & .. & e_1 & e_0 & 0 & .. & 0 \\ \hline & c'_{i+m} & 0 & .. & 0 & 0 & 0 & .. & 0 \end{array}$$

Figure 4.4 Addition Corresponding to the Selection of a Positive Quotient Digit. Numbers are in non-redundant radix r such that each digit is positive or zero and less than r.

For a sliding window algorithm $c_i \neq 0$. At the ith significance the addition $c_i + e_i$ must generate a carry that is propagated all of the way to the mth significance. Therefore $c'_{i+m} = c_{i+m} + e_m + 1 \bmod r$ and to achieve $c'_{i+m} = 0$ we must have $c_{i+m} = r - 1 - e_m$. Whenever this condition is true, the positive quotient digit should be selected.

Now consider the subtraction in Figure 4.5.

$$\begin{array}{cccccccccc} & c_{i+m} & c_{i+m-1} & .. & c_{i+1} & c_i & 0 & .. & 0 \\ + & e_m & e_{m-1} & .. & e_1 & e_0 & 0 & .. & 0 \\ - & n_0 & 0 & .. & 0 & 0 & 0 & .. & 0 \\ \hline & c'_{i+m} & 0 & .. & 0 & 0 & 0 & .. & 0 \end{array}$$

Figure 4.5 Subtraction Corresponding to the Selection of a Negative Quotient Digit.

Again for sliding windows $c_i \neq 0$ so that i^{th} significance must generate a carry that is propagated all of the way to the mth significance. Thus $c'_{i+m} = c_{i+m} + e_m - n_0 + 1 \bmod r$. To achieve $c'_{i+m} = 0$ we must have $c_{i+m} = n_0 - 1 - e_m \bmod r$. Whenever this condition is true, the negative quotient digit should be selected.

Signed quotient digit selection begins by selecting q_i according to Equation 4.21. This gives rise to the digit e_m. If $c_{i+m} = r - 1 - e_m$ then the addition $C' = C + q_i N r^i$ will give $c'_{i+m} = 0$. However, if $c_{i+m} = n_0 - 1 - e_m \bmod r$ then the negative quotient digit should be selected and the subtraction $C' = C - \left(r^m - q_i\right) N r^i$ will also give $c'_{i+m} = 0$. As $n_0 \neq r$ there are two distinct values for c_{i+m} that will give $c'_{i+m} = 0$.

The algorithm follows the SSW state diagram of Figure 2.17 and therefore the average number of non-zero quotient digits for long wordlength n is:

$$\frac{n}{m + 2/(r-1)}. \tag{4.24}$$

An example of an iteration of Montgomery reduction using signed sliding windows is given in Figure 4.6.

$N =$	..			1	1	3			$\tilde{n}_0 = 13, \tilde{n}'_0 = 13$
	..	c_{i+3}	c_{i+2}	c_{i+1}	c_i	c_{i-1}	..	c_0	
$C =$	..	1	0	2	2	0	..	0	$q_i = 22, c_{i+m} = n_0 - 1 - e_m = 0$
$-q'_i N r^i$	..	2	0	2	2	0	..	0	Choose $q'_i = q_i - r^m = -12$
$C' =$	..	3	0	0	0	0	..	0	

Figure 4.6 An Example of Montgomery Reduction using SSW. The example shows a single iteration of Montgomery reduction using $SSW_{4,2}$. The digits c_i and c_{i+1} are set to zero through the selection of the quotient digit q_i and accumulation of $q_i N r^i$. All digits are in radix 4.

2.3 Applications of Sliding Window Montgomery Reduction

Sliding window quotient digit selection can reduce the average number of non-zero quotient digits per modular reduction. This reduces the number of modulus multiples to be generated and accumulated.

The benefit from sliding windows is greatest for low values of r where the chance of selecting zero quotient digits is greatest. However, high values of r increase the size of the quotient digits and thereby reduce the total number of iterations. Thus there is a trade-off between sliding windows and higher radices with the optimal radix determined by a compromise between the two.

Sliding windows will be of greatest use when other benefits can be derived from a low radix. The following sections consider systems that already use a lower radix and may benefit from sliding windows.

Pre-computed Modulus Multiples

In RSA the modulus is part of the public key. It will not change for the period of a modular exponentiation and may remain unchanged for a large number of exponentiations. Therefore multiples of the modulus can be pre-computed and re-used for a large number of RSA operations.

If a pre-computed multiple must be stored for each value in the quotient digit set then the memory requirements increase exponentially with radix. Clearly pre-computation is only feasible at lower radices.

For a sliding window Montgomery reduction where quotient digits q_i are m-digits long the modulus multiples q_iN will be $(m+n)$ -digits long. (Note, however, that the least significant m digits of each modulus multiple can be found from q_i and need not be stored. It is always true that addition of the m least significant digits of q_iN will set m digits of the partial result C to zero.)

Sliding windows will reduce the average number of non-zero quotient digits; signed sliding windows will also reduce the number of modulus multiples to be stored. Consider, for example, 2048-bit number reduced by a 1024-bit modulus. If quotient digits are selected from non-redundant radix 16 then on average there will be 240 non-zero quotient digits. $USW_{2,4}$ gives an average of 204.8 non-zero quotient digits using the pre-computed multiples $\{1N, 3N, 5N, \ldots, 15N\}$ – a 17% improvement with a smaller pre-computed table. $SSW_{2,3}$ produces the same average non-zero quotient digits by adding or subtracting the pre-computed multiples $\{1N, 3N, 5N, 7N\}$.

The smart-card implementation described in Chapter 6 makes use of pre-computed modulus multiples and sliding windows.

Rectangular Aspect Multipliers

A rectangular aspect multiplier capable of performing a low-wordlength by high-wordlength multiplication will occupy less hardware area or evaluate faster than a high-wordlength by high-wordlength multiplier. This is because there are fewer partial products to generate and accumulate.

For a hardware system we can choose to implement a rectangular aspect multiplier because of its reduced size, and then seek to improve performance with sliding windows.

Software systems can also benefit from this approach as demonstrated in the following example. The ARM6 processor has a multiply instruction capable of generating a 32-bit result in at most 17 cycles. The time taken to perform the multiplication depends on the size of the first operand and varies from 2 cycles for 1-bit operands to 17 cycles for operands of 29-bits or more.

Consider using the ARM6 multiplier to generate the 1040-bit modulus multiple q_iN for a 16-bit quotient q_i. This requires 64 16-by-16-bit multiplications. On average each multiplication takes 9.33 cycles for a total of 598 multiplication cycles. Without sliding windows, reducing a 2048-bit number will require an average of 64 non-zero quotient digits and thus 38272 multiplication cycles.

Now consider generating the 1032-bit modulus multiple q_iN for an 8-bit quotient. This time 43 8-by-24-bit multiplications are required. On average each multiplication takes 5.336 cycles for a total of 230 multiplication cycles. Using $USW_{16,2}$ to reduce a 2048-bit number leads to 124 non-zero quotient digits on average for a total of 28520 multiplication cycles. This 34% improvement in

multiplication cycles will be carried through to improve overall performance so long as modulus multiple accumulation is efficient.

Reducing the radix has helped in two ways. It has reduced the cost of each multiplication and it has increased the probability of zero quotient digits.

A greater saving in multiplication cycles is possible by choosing an even lower radix. However, this will come at the cost of greater loop overhead and will require accumulation at many different bit boundaries. The use of $r = 16$ in this example should reduce loop overhead and ensure that all accumulations take place on regular nibble (4-bit) boundaries. This may make accumulation more efficient for some processors, such as those that can implement 4-bit shifts efficiently using nibble-swap instructions.

Signed sliding windows provide further improvement by reducing the average magnitude of quotient digits. For example, with $SSW_{16,2}$ each quotient digit can be represented with at most 7-bits. This reduces the average multiplication cost to 4.672 cycles. The reduction now requires 24924 multiplication cycles.

Shift-and-Add Multiple Generation

Digit products of the modulus can be rapidly generated by adding (or subtracting) a few shifted versions of the modulus. For example, all 5-bit multiples of the modulus can be generated by adding at most 3 shifted versions of the modulus (as in, for example [OK91]).

A hardware system adopting this approach can derive further benefit from sliding window reduction. The situation is closely related to the rectangular aspect multiplier described in the previous section: sliding windows will reduce the number of non-zero quotient digits with signed sliding windows providing greater improvement for a given quotient digit size.

For example, 1024-bit Montgomery reduction with 5-bit quotient digits will give approximately 199 non-zero quotient digits on average. For the same multiplier, $USW_{2,5}$ gives approximately 171 non-zero quotient digits with only 147 for $SSW_{2,5}$.

3 Triangle Additions

Takagi in [Tak93] describes a method of (almost) separated multiplication and reduction that keeps intermediate results to around n-digits rather than $2n$-digits of conventional separated schemes. In this section, Takagi's method is adapted for Montgomery reduction.

3.1 Takagi's Triangle Additions

Equation 4.25 is at the heart of the method. It shows how the upper triangle of partial products D_H can be reduced prior to the accumulation of the lower triangle D_L.

$$\text{Let } D = A \times B \text{ with } D = 2^n D_H + D_L$$

$$\text{then } D \bmod N = \left(2^n D_H \bmod N + D_L\right) \bmod N. \tag{4.25}$$

The triangle additions proceed as follows:

1. Generate A_D from the upper triangle of partial products. This will require n-digits of storage (and a carry).
2. Compute $2^n D_H \bmod N$ using a left to right reduction scheme. The input number starts with n implicit zeros on the right. At each iteration the partial result loses an implicit zero from the least significant end but gains one at the most significant end. Thus intermediate results are kept to n-digits.
3. Accumulate the lower triangle of partial products D_L into the result.
4. Perform a final subtraction to reduce the result to less than N.

Takagi performs the reduction in step 2 using a series of shifts and subtractions. The total number of partial product and modulus multiple accumulation steps is the same as for interleaved reduction and intermediate results are also kept to around n-bits.

Montgomery Triangle Additions

In Takagi's triangle additions, reduction of the lower triangle is trivial with all of the reduction effort required in the upper triangle. The situation is reversed with Montgomery reduction. This time the upper triangle is trivially reduced with the lower triangle requiring all of the effort.

$$\text{Let } D = A \times B \text{ with } D = 2^n D_H + D_L$$

$$\text{then } \mathrm{MR}_N(D) = \left(\frac{2^n D_H + D_L}{2^n}\right) \bmod N = (D_H + \mathrm{MR}_N(D_L)) \bmod N. \tag{4.26}$$

Evaluation begins by generating D_L from the lower triangle of partial products. This will be n digits long (plus a carry). At each iteration of the Montgomery reduction $\mathrm{MR}_N(D_L)$ the least significant digit of the partial result is set to zero before the partial result is shifted one digit to the right. The intermediate result remains n-digits long.

Following the Montgomery reduction the upper triangle of partial results can be accumulated. A final subtraction is required to ensure the final result is less than N.

3.2 Summary

One of the most frequent objections to separated reduction and multiplication is the need for $2n$ digits of intermediate storage. Many papers eliminate separated modular multiplication from their considerations by citing this reason alone.

Takagi's triangle additions provide a method for separated left-to-right reduction with intermediate results kept to n-digits (and a few overflow bits). This technique has been adapted for Montgomery reduction to achieve the same result.

4 Summary and Conclusions

This chapter began with a survey of modular multiplication and reduction algorithms. It was found that all of the algorithms surveyed could be classified into 4 categories. Each of these categories was discussed and their evolution was traced over the history of publications. Further results of this survey appear in Appendix A.

There are a number of conclusions to draw from this survey. Firstly, the three most common categories of reduction have all evolved to the point where quotient digit selection and modulus multiple generation can be moved from the critical path of a hardware implementation. The critical path of the reduction has become the accumulation of partial products and modulus multiples. For reduction using partial remainders and reduction using multiplication by the inverse, these critical accumulations can be performed using parallel adders. For Montgomery reduction a carry propagate adder can be used without any loss of performance.

Comparisons between the performance of the different categories of reduction are difficult to make, especially in software. The best reduction algorithm for a given implementation will depend on the details of that implementation. The remainder of the chapter, and the rest of this thesis, concentrates on Montgomery reduction; however the techniques developed, particularly digit set conversion, can be applied to any of the other reduction algorithms.

In Section 2 sliding window digit set conversion was applied to Montgomery reduction. Using sliding windows it is possible to reduce the average number of non-zero quotient digits in modular reduction. This reduces the average number of modulus multiples that must be generated and accumulated.

In the case of reduction it was found that signed sliding windows have an advantage over unsigned sliding windows of the same length. When a negative quotient digit q_i emerges from signed recoding, it can be dealt with by subtracting the positive modulus multiple $|q_i|N$. In this way signed recoding gives a lower average arithmetic weight for a given number of quotient digit magnitudes. If modulus multiples are pre-computed, then signed recoding reduces the storage and pre-computation effort required. If modulus multiples are being generated then faster generation may be possible due to the reduced average magnitude of quotient digits. Montgomery reduction using sliding windows is therefore of benefit to both software systems (as in Chapter 6) or hardware systems (such as those similar to Sedlak's cryptography processor [Sed88]).

In the last section, Takagi's triangle addition technique of separated multiplication and reduction was applied to Montgomery multiplication. This provides a mechanism for separated multiplication and Montgomery reduction with intermediate results that remain approximately n digits long.

Chapter 5

Multiple-precision Multiplication and Squaring

TO ACCELERATE MULTIPLICATION OR SQUARING one or two ideas are usually applied: reduce the number of partial products or accelerate the accumulation of partial products. With a view ahead to a smart-card implementation of RSA, this chapter will concentrate on techniques applicable to multiple-precision software. This narrows the field somewhat as a significant proportion of multiplication literature is concerned with parallel schemes for the accumulation of partial products—a technique that is only applicable to hardware. In general, methods that reduce the number of partial products can be applied to both software and hardware and are of interest to the current study.

Multiplication and Squaring in RSA

Consider multiplication and squaring in the context of the modular exponentiation $S = M^K \bmod N$. If this is performed using the left-to-right algorithm, a multiplication $S = S \times M^{k_i} \bmod N$ is performed for each non-zero digit in the exponent, and a squaring $S = S^2 \bmod N$ is required for every bit in the exponent.

As the number M is constant throughout the exponentiation, the digit powers M^k can be pre-computed and re-used. With sufficient memory it is possible to go on and pre-compute digit multiples of these powers sM^k to re-use for every multiplication in the exponentiation.

Note that the value to square S changes for every iteration. If digit multiples sS are to be used then they must be re-computed for every square.

Therefore, with sufficient memory and pre-computation one can readily reduce the number of multiplications in an exponentiation and also the cost of each multiplication. The same does not hold true for squaring as the number of squares is fixed by the length of the exponent and the number to be squared changes at each iteration.

Multiplication or Modular Multiplication?

Much of the literature on public-key cryptography considers modular multiplication as a single operation in which multiplication and reduction are interleaved in one algorithm.

This chapter focuses on the operations of multiplication and squaring. It is important to note that this separated treatment of multiplication and reduction does not rule out the possibility of interleaved modular multiplication—on the contrary, with multiplication and reduction methods enumerated separately the possibility of combining them in new ways is made more explicit.

1 Background

The term *multiple-precision arithmetic* refers to arithmetic performed on numbers represented by more than one digit. It is usual to implement multiple-precision arithmetic using a series of digit by digit operations. This is commonly encountered in software implementations of long wordlength arithmetic where input operands exceed the wordlength of the processor.

This brief section considers existing algorithms for multiple-precision multiplication and squaring, especially as reported in the literature of public-key cryptography.

1.1 Multiple-precision Multiplication

Multiplication of long-wordlength integers on a w-bit processor often assumes the existence of a primitive w-by-w-bit multiplier capable of generating a $2w$-bit result. This facilitates the multiple-precision algorithm shown as Algorithm 5.1 (called *operand scanning* in [KAK96]).

Algorithm 5.1 Operand Scanning Multiplication. The product of two l-word numbers A and B, is returned in the 2l-word result *C*. The notation (c,s) represents the 2-word register composed of registers c and s.

```
for i = 0 to l -1
    c = 0
    for j = 0 to l - 1
        (c,s) = c_{i+j} + a_i * b_j
        c_{i+j} = s
    next j
next i
```

In this scheme partial products are generated and accumulated from the right to the left. This facilitates efficient carry propagation. Also, the final value of c_i is produced in the ith iteration. Montgomery reduction can be incorporated to reduce c_i to zero with a modulus accumulation in the inner (*j*) loop.

Re-ordered Iterations

The familiar form of multiplication as an accumulation of partial products is shown in Figure 5.1. Operand scanning multiplication (Figure 5.1) generates and accumulates one partial product row at a time. An alternative is to evaluate the product a digit at a time. This corresponds to generating and accumulating a column of the partial product digits before moving on to the next column. This is called *product scanning* in [KAK96] and *convolution sum multiplication* in [DK91]. An algorithm is given as Figure 5.2.

					b_3	b_2	b_1	b_0
				x	a_3	a_2	a_1	a_0
				$(a_0B)_4$	$(a_0B)_3$	$(a_0B)_2$	$(a_0B)_1$	$(a_0B)_0$
			$(a_1B)_4$	$(a_1B)_3$	$(a_1B)_2$	$(a_1B)_1$	$(a_1B)_0$	
		$(a_2B)_4$	$(a_2B)_3$	$(a_2B)_2$	$(a_2B)_1$	$(a_2B)_0$		
+	$(a_3B)_4$	$(a_3B)_3$	$(a_3B)_2$	$(a_3B)_1$	$(a_3B)_0$			
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

Figure 5.1 Multiplication as a Sum of Partial Products.

Algorithm 5.2 Product Scanning Multiplication. The product of two l-word numbers A and B, is returned in the 2l-word result C. The notation (c,s) represents the 2-word register composed of registers c and s. The length of the carry propagation steps (the maximum value of k) will be only a few registers and can be determined in advance

```
for i = 0 to l -1
    for j = 0 to i
        (c,c_i) = c_i + a_j b_(i-j)
        k = 1
        while c > 0                 // Carry propagation
            (c,c_(i+k) ) = c_(i+k) + c
            k = k + 1
        wend
    next j
next i
for i = l to 2l - 1
    for j = i - l + 1 to l - 1
        (c,c_i) = c_i + a_j b_(i-j)
        k = 1
        while c > 0                 // Carry propagation
            (c,c_(i+k)) = c_(i+k) + c
            k = k + 1
        wend
    next j
next i
```

Once again the final value of c_i is produced in the ith iteration. At this point a Montgomery quotient digit q_i can be selected and accumulation of $q_i N$ can be interleaved with subsequent iterations.

The most efficient form of multiplication depends on the instruction set of the target processor. In [Bar87] product scanning is used on a TMS32010 to reduce the number of fetches and carry propagation steps and to facilitate auto-incrementing of pointers. In [DK91] product scanning reduces the number of stores and right shifts for a DSP56000. Product and operand scanning with a variety of Montgomery interleaving schemes are compared in [KAK96] on a Pentium 60. For this machine operand scanning is found to be fastest in every case.

Changing the Multiplier Representation

To quickly compute $A \times B$ the representation of A can be changed to reduce the number of non-zero digits a_i and hence the number of partial products. One way to do this is to increase the radix of the digits. For the multiple-precision algorithms above the maximum radix is determined by the width of the primitive multiplier.

Digit set conversion to reduce the number of multiplier digits was discussed in Chapter 2. It was found that modified Booth conversion is commonly applied to hardware implementations. It provides a carry-free method of increasing the radix and also reduces the number of possible multiplier digit magnitudes by using a balanced set of signed and unsigned digits.

Software implementations of modified Booth are unable to take advantage of the digit parallel conversion. Instead, minimal sliding window conversion can be implemented just as efficiently to provide improved average case performance. This is discussed in more detail in Section 2 below.

For implementations based on a wordlength multiply instruction there is little to be gained by digit set conversion. With digits that are 16 or 32-bits long, the probability of finding zero-digits is extremely small. Pre-computation is also of no benefit: the probability of digit values recurring is small and complete pre-computed tables are so large as to be infeasible.

Karatsuba Multiplication

Karatsuba's multiplication method (Equation 5.2 and Equation 5.3) allows a $2w$-bit product to be evaluated from three w-bit products rather than the usual four suggested by Equation 5.1 [Knu97], [Zur93].

Given $A = A_H 2^w + A_L$ and $B = B_H 2^w + B_L$ we have

$$A \times B = (A_H B_H) 2^{2w} + A_H B_L 2^w + A_L B_H 2^w + A_L B_L \tag{5.1}$$

from Karatsuba $$A \times B = (2^{2w} - 2^w)(A_H B_H) + 2^w (A_H + A_L)(B_H + B_L) + (1 - 2^w) A_L B_L \tag{5.2}$$

or Knuth $$A \times B = (2^{2w} + 2^w)(A_H B_H) + 2^w (A_H - A_L)(B_L - B_H) + (2^w + 1) A_L B_L\ . \tag{5.3}$$

The benefit of reduced multiplication cycles in these equations must be weighed against the overhead of extra additions and shifts. In Knuth's form (Equation 5.3), the terms $(A_H - A_L)$ and $(B_L - B_H)$ may be negative. This problem of sign significantly exaggerates implementation overhead on most processors. In the first form (Equation 5.2), the product $(A_H + A_L)(B_H + B_L)$ may be $(2w + 2)$ -bits long and this again increases the overhead.

Shand and Vuillemin apply Karatsuba multiplication to improve the multiplication primitive on a MIPS R3000A processor [SV93]. The MIPS native instruction set provides a 32-by-32-bit integer multiplication in 16 cycles. With careful coding, 64-by-64-bit multiplication is possible in 51 cycles using Karatsuba. On this processor all but 3 cycles of the add and shift overhead can be overlapped with the execution of the integer multiply instruction.

It is possible to recursively apply Karatsuba multiplication to longer and longer wordlengths; however for the MIPS, Shand and Vuillemin find that deeper recursion is of no benefit as the processor runs out of registers for intermediate results and it is no longer possible to hide the Karatsuba overhead behind execution of the multiplier.

Karatsuba multiplication is considered for an ARM processor in [Dhe98]. The author finds Karatsuba of no benefit as the overhead of dealing with the signed results outweighs the benefit of reduced multiplication cycles. The memory required to implement a recursive algorithm is also a serious concern.

Other Multiplication Schemes

In Karatsuba multiplication, w-bit operands are split into 2 parts. Analogous techniques exist to split the operands into an arbitrary number of parts. Toom-Cook multiplication [Knu97] and Zuras' n-way multiplication [Zur93] involve recursive application of these techniques.

Zuras compares these recursive algorithms with multiple-precision product (or operand) scanning and also with multiplication using the Fast Fourier Transform. He finds that simple multiple-precision multiplication is fastest for small wordlengths (tens of words). Recursive Karatsuba (2-way) multiplication is best for up to thousands of words but the basic multiple-precision technique remains competitive even at these long wordlengths.

1.2 Optimised Squaring

In their discussion on modular squaring, Beth and Gollmann [BG89] observe that the product $A \times B$ can be computed by $A \times B = ((A+B)^2 - A^2 - B^2)/2$ using only additions and squaring. Karatsuba provides another method of computing the product using squares [Zur93]: $A \times B = ((A+B)^2 - (A-B)^2)/4$. Note that in this form the multiplication requires only two squares. One can postulate that squaring is about half as expensive as multiplication and indeed this is often the case.

Optimised squaring is a well known improvement over squaring using multiplication (as in [Che71]). The optimisation is due to the observation that for squaring most of the required digit products are

repeated twice. For example, in the evaluation of $A \times A$ the digit products $a_i a_j$ and $a_j a_i$ (where $i \neq j$) are both required but will be equal. The partial product parallelogram can be rearranged such the number of terms to be accumulated is approximately half that required for a full multiplication. In practice the speed-up is somewhat less than two: Shand and Vuillemin claim a 77% improvement for 512-bit squaring implemented software [SV93]. Dhem finds that the actual benefit due to squaring is much lower than the theoretical maximum due to difficulties with the accumulation of 'double products' $2a_i a_j$ that are 2 words and 1 bit long.

A pseudo-code algorithm for optimised squaring is given in Figure 5.3. Publications that consider optimised squaring for modular exponentiation include [KAK96], [Dhe98], [SV93], [DK91] and [HOY96]. In Section 3 a new algorithm is developed that applies sliding windows to optimised squaring.

Algorithm 5.3 Optimised Squaring. The square of an l-word number A is returned in the 2l-word result C. The notation (c,s) represents the 2-word register composed of registers c and s.

```
for i = 0 to l -1
    (c, c_2i) = c_2i + a_i a_i
    for j = i + 1 to l - 1
        (c, c_i+j) = c_i+j + 2a_j a_i + c
    next j
next i
```

2 Sliding Window Multiple-precision Multiplication

As discussed above, digit set conversion is a common practice for hardware multipliers. The conversion is usually performed to increase the radix and hence reduce the number of partial products to generate and accumulate. Parallel conversion schemes, such as modified Booth, decrease the total number of multiplier digits and may also reduce the number of possible digit magnitudes by employing a balanced signed digit set. Very occasionally, such as in [Sed88], a minimal binary conversion is used to reduce the average number of non-zero bits and improve average case performance.

Despite its prevalence in hardware implementations, digit set conversion is rarely applied to software multiplication. There are some good reasons for this. It would appear that the most promising method for performing a long multiple-precision multiplication would be to use word-by-word multiplications as in Figure 5.1—especially if the target processor has an efficient wordlength

multiply instruction. This has some advantages: intermediate results are always aligned on word boundaries, and the number of iterations and hence loop overhead is kept low. For these reasons the word-by-word approach now occupies the high-ground of good common sense and has become a standard solution for long multiplications in software. The usual methods suggested to improve performance are to provide a faster primitive multiply instruction, increase the processor's wordlength, provide special loop instructions to reduce the loop overhead, unroll software loops, provide a multiplier capable of executing in parallel with other instructions, or for very long multiplications, apply Karatsuba recursively. In the following sections I propose a different solution.

2.1 Multiple-precision Multiplication with Sliding Windows

Let us examine the multiple-precision multiplication $A \times B$ in which the digits a_i are smaller than a typical processor wordlength (16 or 32-bits). It has already been observed in Chapter 4 that some surprising results are possible when the radix is reduced in this way. The digit products $a_i b_j$ become easier to compute and pre-computation of partial products $a_i B$ becomes feasible. Although the worst case number of iterations is increased, digit-set conversion can be used to keep the average number of non-zero digits a_i well below the worst case.

In multiplication the effect of a reduced radix is that the cost of partial product generation is reduced at the expense of partial product accumulation. There will be more partial products and the partial products are no longer accumulated at regular word boundaries—they will need to be shifted for accumulation. However, it is possible that the benefits will outweigh these costs such that the new solution is worthwhile.

Pre-computed Partial Products with Unsigned Sliding Windows

Equation 5.4 expresses multiplication as a sum of partial products, a form suitable for use with unsigned sliding windows. It is assumed that the multiplier A has already been converted using $USW_{r,m}$ where $r = 2^t$.

$$A \times B = \sum_{i=0}^{l-1} a_i B 2^{i \times t} \tag{5.4}$$

The partial products $a_i B$ can be evaluated as required using a rectangular aspect multiplier. Alternatively, the partial products can be pre-computed, stored and re-used.

Note that one of the disadvantages of sliding windows is that partial products no longer occur on regular word boundaries. A fast implementation will require an efficient mechanism for accumulating

shifted multiple-precision numbers. One such solution is presented in Chapter 6 for a system based on an ARM processor.

For $USW_{r,m}$ with pre-computed partial products the digit multiples aB are pre-computed for all a such that $\le a < r^m$ and $a \ne 0 \bmod r$. For example, $USW_{2,m}$ can be implemented with the pre-computed table $\{B, 3B, 5B, \ldots, (2^m - 1)B\}$. The iteration $(i+2)B = iB + 2B$ provides a means of pre-computing the table using only adds and shifts. In this way the entire table takes $2^{m-1} - 1$ accumulations to generate.

Following digit-set conversion with $USW_{r,m}$ an average of $n/(m+1)$ non-zero partial products will require accumulation.

In the case where the table of partial products must be re-computed for each product, there is a trade-off between the accumulations required to pre-compute the table and the accumulations required to evaluate the product. In fact, the trade-off is identical to that observed for exponentiation using unsigned sliding windows (as discussed in Chapter 3). Figure 5.2 and Table 5.1 illustrate the situation for $n = 512$ and show that the total average accumulations is minimised for $m = 5$.

The size of the pre-computed table is also a consideration. The table must contain 2^{m-1} partial products (including B). Note that the partial products can be up to $(n + m)$ bits long, and in software an extra word would likely be required to store the extra m-bits. In Table 5.1 a word size of 32-bits is assumed.

m	Pre-compute Accumulations	Average Evaluate Accumulations	Total Average Accumulations	Table Size (Bytes)
m	$2^{m-1}-1$	$n/(m+1)$	$2^{m-1}-1+n/(m+1)$	$68 \times 2^{m-1}$
1	0	256	256	68
2	1	170.67	171.67	136
3	3	128	131	272
4	7	102.4	109.4	544
5	15	85.33	100.33	1088
6	31	73.14	104.14	2176
7	63	64	127	4352
8	127	56.89	183.89	8704

The table size is based on 512-bits and an extra 32-bits for each partial product.

Table 5.1 Design Trade-off for 512-bit $USW_{2,m}$ Multiplication.

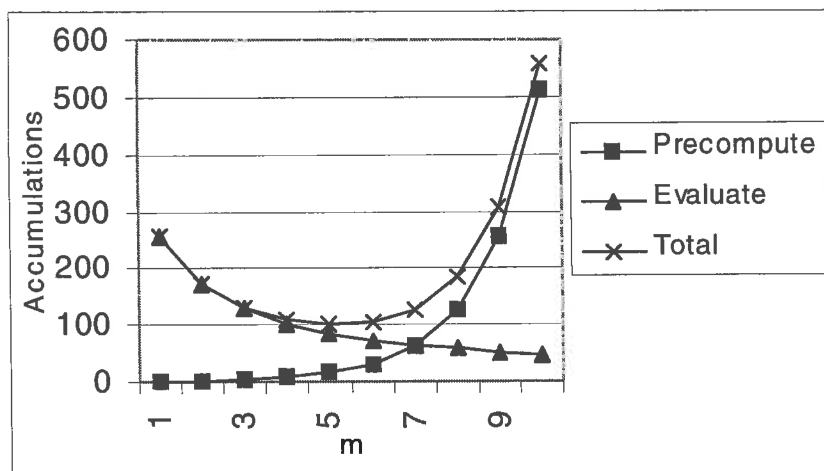

Figure 5.2 Average Total Accumulations for $USW_{2,m}$ Multiplication.

Pre-computed Digit Multiples with Signed Sliding Windows

The multiplication described by Equation 5.4 is equally applicable to representations where the digits a_i are signed. Therefore a signed digit recoding scheme can be used to further decrease the average number of non-zero digits.

In the signed conversion $SSW_{2,m}$ the digits a_i belong to the set $\{-2^m+1, \ldots, -3, -1, 0, 1, 3, \ldots, 2^m-1\}$. However, only the positive partial products $\{B, 3B, \ldots, (2^m-1)B\}$ need be pre-computed – whenever a negative digit occurs the corresponding positive partial product can be subtracted. Thus for the same pre-computation effort and storage, $SSW_{2,m}$ achieves a better average performance than $USW_{2,m}$: $n/(m+2)$ non-zero partial products compared with $n/(m+1)$.

The trade-off for $n = 512$ is illustrated in Figure 5.3 and Table 5.1. Once again the total average accumulations is minimised for $m = 5$, but this time the minimum is 88.15 accumulations compared with 100.33 for unsigned conversion – a 13.8% improvement.

The authors of [KH92] overlook the saving their signed *adaptive m-ary segmentation canonical recoding* delivers in terms of pre-computation effort and storage. They evaluate partial products for all quotient digits, including negative partial products for the negative digits. It is therefore little surprise that they conclude that their signed recoding offers no advantage over unsigned sliding windows for large *n*.

m	Pre-compute Accumulations	Average Evaluate Accumulations	Total Average Accumulations	Table Size (Bytes)
m	$2^{m-1}-1$	$n/(m+2)$	$2^{m-1}-1+n/(m+2)$	$68\times 2^{m-1}$
1	0	170.67	170.67	68
2	1	128	129	136
3	3	102.4	105.4	272
4	7	85.33	92.33	544
5	15	73.14	88.15	1088
6	31	64	95	2176
7	63	56.89	119.89	4352
8	127	51.2	178.2	8704

The table size is based on 512-bits and an extra 32-bits for each partial product.

Table 5.2 Design Trade-off for 512-bit $SSW_{2,m}$ Multiplication.

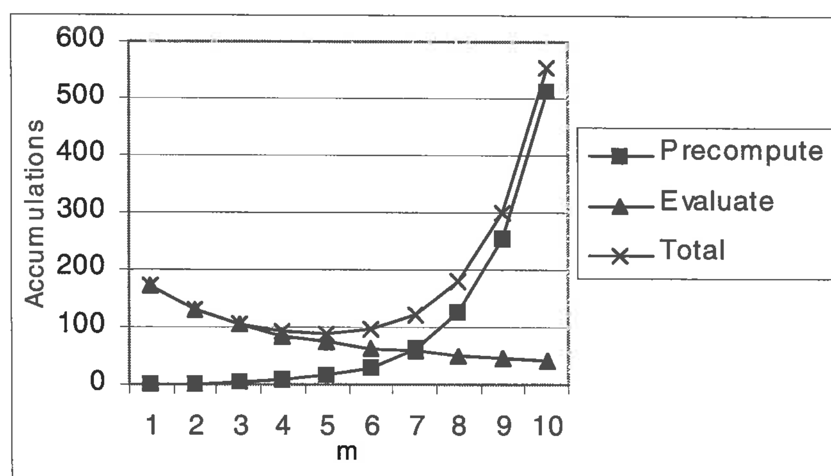

Figure 5.3 Average Total Accumulations for $SSW_{2,m}$ Multiplication.

One drawback of the signed scheme should be observed: as negative partial products are accumulated, the partial result can swing negative. If the partial result is kept in 2's complement form then a borrow or carry may propagate to the most significant bit whenever the partial result changes sign. However, if partial products are accumulated from the right to the left (from the least significant to the most) then it is possible to maintain a sign digit just to the left of the last partial product accumulated. This will still add some computational overhead but will avoid long carry propagation.

Sliding Window Conversion During Multiplication

In the previous sections it has been assumed that the multiplier A is recoded according to a sliding window conversion prior to multiplication. This need not be the case if both multiplication and

conversion proceed from the right to the left. As an example, Figure 5.4 shows the pseudo-code for multiplication performed simultaneously with $SSW_{2,m}$ conversion.

Algorithm 5.4 Simultaneous Multiplication and $SSW_{2,m}$ Conversion. The product of A and B is returned in C.

```
C = 0
i = 0
s = 0                               // Start skipping zeros
while i < l
    if A mod 2 = s then
        A = A >> 1                  // Skip a bit
        i = i + 1
    else
        a_i = A mod 2^m + s         // Choose a positive digit
        s = 0                       // Skip zeros
        if (A >> m) mod 2 = 1 then
            a_i = a_i - 2^m         // Make the digit negative
            s = 1                   // Skip ones
        end if
        A = A >> (m+1)
        C = C + a_i B 2^i           // Accumulate a partial product
        i = i + m + 1
    end if
wend
```

Karatsuba Multiplication with Sliding Window

In the literature of public-key cryptography it is most common for Karatsuba multiplication to be considered as a means for increasing the wordlength of the primitive multiply instruction. For example, on a 32-bit processor, Karatsuba may be used to define an efficient 64-by-64-bit multiplication routine that uses 3 32-by-32-bit multiply instructions. It may prove beneficial to apply Karatsuba again to define a 128-by-128-bit multiplication routine based on 3 applications of the 64-by-64-bit multiplication routine. Following this scheme, Karatsuba is recursively applied from small wordlengths up to longer wordlengths.

This is not applicable when sliding windows are used as in Algorithm 5.4—there are no short-wordlength multiplications to which to apply Karatsuba's algorithm! Instead it is possible to apply

Karatsuba from long wordlengths down, breaking a 1024-bit multiplication into 512-bit multiplications and then perhaps into 256-bit multiplications and so on.

Equation 5.5 and Figure 5.4 show Karatsuba applied to n-bit multiplication. For long n, most of the effort is required to evaluate the three $(n/2)$ -bit multiplications. The Karatsuba overhead due to signed terms or overflow bits is trivial when compared with the total cost of the multiplication.

$$A \times B = (2^n - 2^{n/2})\ (A_H B_H) + 2^{n/2}(A_H + A_L)\ (B_H + B_L) + (1 - 2^{n/2}) A_L B_L \tag{5.5}$$

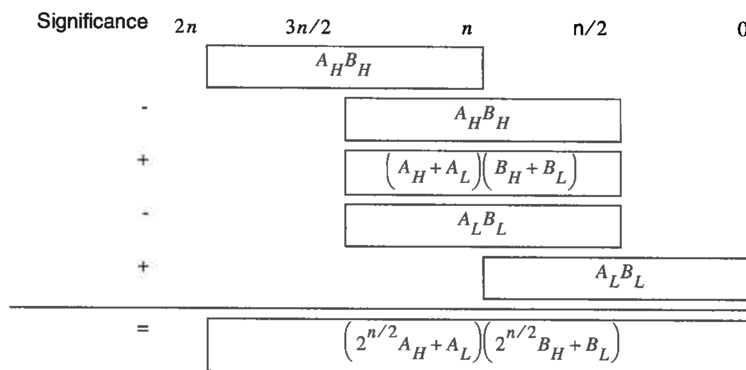

Figure 5.4 Karatsuba Multiplication. The diagram shows the 4 final accumulations required to perform n-bit multiplication using three ($n/2$)-bit multiplications.

Let us consider 512-bit multiplication using Karatsuba and $SSW_{2,m}$ with pre-computed partial products. For 256-bit multiplication, $SSW_{2,m}$ exhibits a minimum average number of accumulations when $m = 4$. In this case 49.67 accumulations are required per multiplication and hence 149 accumulations for the 3 multiplications required by Karatsuba. However, each one of these accumulations will be only 256-bits long. For the moment assume that a 256-bit accumulation takes half the time of a 512-bit accumulation. Thus the 3 multiplications require the equivalent of 74.5 512-bit accumulations. In addition, 4 more 512-bit accumulations are required to form the 1024-bit product. Therefore on average the equivalent of 78.5 512-bit accumulations are required per 512-bit multiplication. This compares favourably with the 88.15 accumulations required without Karatsuba.

The assumption that a 256-bit accumulation takes half the time of a 512-bit accumulation may be optimistic as each accumulation will exhibit an overhead that is independent of the wordlength. Nonetheless, the Karatsuba method is likely to be competitive with the non-Karatsuba multiplication in time with a significant advantage in terms of memory requirement.

Each of the three 256-bit multiplications requires pre-computation of 8 partial products. If each partial product is allocated nine 32-bit words of memory (288-bits) then the total table will occupy

288-bytes. Storage is also required for the three 512-bit intermediate results but this can be minimised using the following procedure where C is the 1024-bit destination buffer:

1. Form $A_H B_H 2^n$ directly into C.
2. Perform the subtraction $C \leftarrow A_H B_H 2^n - A_H B_H 2^{n/2}$. Note from Figure 5.4 that this can be performed within the buffer C without copying the operands to other buffers.
3. Accumulate the product $2^{n/2}(A_H + A_L)(B_H + B_L)$ into C as it is formed.
4. Form $A_L B_L$ in a separate buffer. It can then be accumulated twice into C to give the final result.

Therefore, only one additional 512-bit buffer is required. This increases the temporary memory requirement to 352-bytes – a significant improvement over $SSW_{2,5}$ which requires 1088-bytes.

Karatsuba may or may not provide some small improvement in average execution delay, depending on the implementation. However, as shown in Table 5.1 it is likely to achieve competitive performance with reduced memory requirements.

Note that Karatsuba multiplication and interleaved reduction can not be efficiently implemented together. Figure 5.4 shows that the intermediate products $A_H B_H$ and $A_L B_L$ must each be accumulated at 2 offsets. This is not possible if they have already been reduced.

m	Average SSW & Karatsuba 512-bit Accumulations	Average SSW 512-bit Accumulations	Karatsuba & SSW Memory (Bytes)	SSW Memory (Bytes)
m	$3 + \left(\frac{384}{m+2}\right) + 2^{m-1}$	$2^{m-1} - 1 + \frac{612}{(m+2)}$	$64 + 36 \times 2^{m-1}$	$68 \times 2^{m-1}$
1	132	170.67	100	68
2	101.5	129	136	136
3	85.3	105.4	208	272
4	78.5	92.33	352	544
5	81.36	88.15	640	1088
6	98.5	95	1216	2176
7	141.17	119.89	2368	4352
8	232.9	178.2	4672	8704

It is assumed that a 256-bit accumulation takes half the time of a 512-bit accumulation.

Table 5.3 The Influence of Karatsuba on $SSW_{2,m}$ Multiplication.

3 Optimised Squaring with Sliding Windows

It is not immediately obvious that sliding windows and optimised squaring are compatible. Optimised squaring seems to require digits at regular and repeated significances whereas following sliding window conversion, the non-zero digits appear at irregular bit significances. This section describes a new algorithm that combines these apparently incompatible techniques.

Figure 5.5 shows two forms of a multiple-precision square. The operand scanning form is shown on the left, and in the version on the right, the partial products have been re-ordered to demonstrate an optimised square.

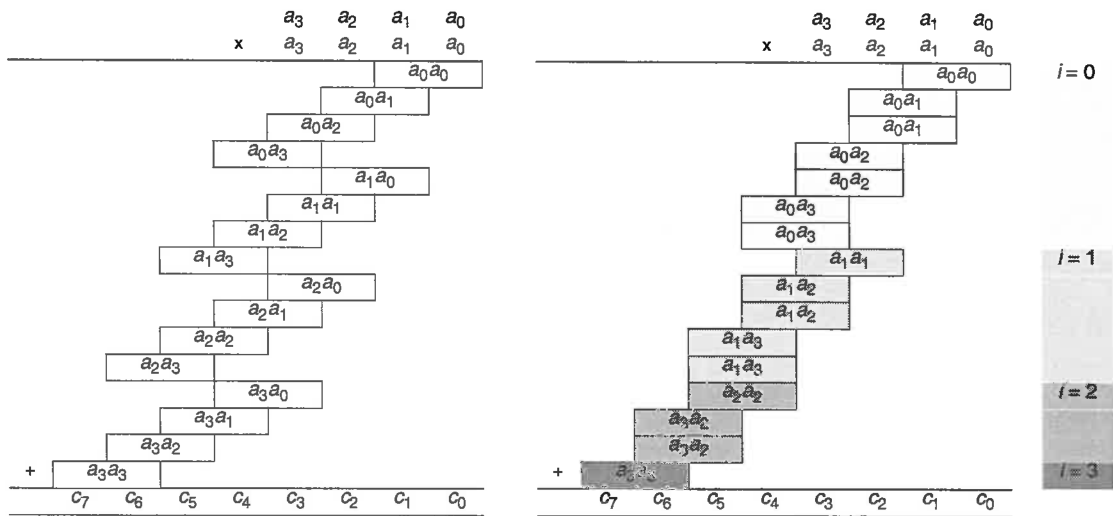

Figure 5.5 Multiple-precision Square. The 4-word number *A* is squared to give the 8-word result *C*. The operand scanning form of a square is shown on the left. An optimised arrangement is shown on the right.

The optimised version can be described by the following iteration:

$$A^2 = \sum_{i=0}^{l-1} \left(a_i^2 r^{2i} + 2a_i A_{i+1} r^{2i+1} \right)$$

$$\text{where } A_i = \sum_{k=i}^{l-1} a_k r^{k-i} . \tag{5.6}$$

The term A_i is a truncated version of A formed from the most significant $(l-i)$ digits: $A_i = (a_{l-1}, a_{l-2}, \ldots, a_i)$. The square from Figure 5.5 is repeated once more in Figure 5.6 to demonstrate the new iteration.

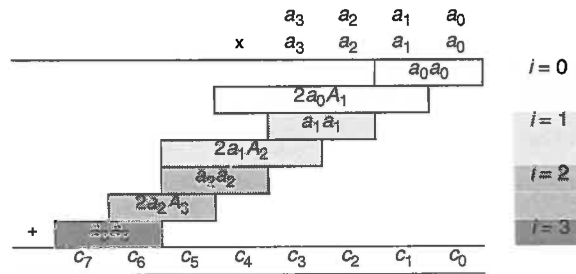

Figure 5.6 Optimised Multiple-precision Square. The 4-word number A is squared to give the 8-word result C.

Interleaving Optimised Squaring and Reduction

Note in Figure 5.6 that each iteration produces two digits of the final result: iteration i generates c_{2i} and c_{2i+1}. It is therefore possible to interleave Montgomery reduction with optimised squaring. At the end of iteration i one can add a multiple of the modulus that will set c_{2i} and c_{2i+1} to zero. In this way the intermediate results are kept to $(l+1)$ -digits in length.

Optimised Squaring Following Digit Set Conversion

Optimised squaring as described by Equation 5.6 is directly applicable to number representations that result from a sliding window digit-set conversion so that it is possible to first apply a digit set conversion and then perform an optimised square. Whenever there is a zero digit a_i, the ith iteration becomes trivial. An example is given in Figure 5.7.

Start with a radix 4 number	$A =$					1	2	3	1	1	$A = 437_{10}$
Following $SSW_{4,2}$	$A =$					0	7	0	0	-11	
Now square	x					0	7	0	0	-11	
							1	3	2	1	a_0^2
				-1	0	-3	-1				a_0a_3
				-1	0	-3	-1				a_0a_3
	+	0	3	0	1						a_3^2
		0	3	0	-1	0	-6	-1	3	2	1	$A^2 = 190969_{10}$

Figure 5.7 An Example of Squaring a Converted Number. The number A is converted using $SSW_{4,2}$. The new representation is then squared using the optimised algorithm. With a_1, a_2 and a_4 all equal to zero, the 1st, 2nd and 4th iterations all become trivial.

Digit Set Conversion During Square

Now consider performing a digit-set conversion from right to left during an optimised square. At iteration i the conversion may change the value of the digit a_i and correct for this change by adjusting the digits of A to the left. Thus the value of A_{i+1} also changes. However, in the new representation, the previous digits a_0 to a_{i-1} and the values A_0 to A_i remain unchanged. The existing partial result

from the previous iteration is valid for the new representation. A pseudo-code algorithm is given in Figure 5.5.

Algorithm 5.5 Optimised Squaring with Right to Left Digit Set Conversion. The square of A is returned in C. The convert_lsd(A_i) function performs one iteration of a right to left digit set conversion. It converts the least significant digit of A_i and propagates any carry or borrow into subsequent digits to correct for this change.

```
C = 0
A0 = A
for i = 0 to l -1
    convert_lsd(Ai)                  // the least significant digit of Ai is converted
    ai = Ai mod r                    // ai is the new least significant digit
    Ai+1 = Ai div r                  // shift right by one digit
    C = C + ai^2 r^2i + 2 ai Ai+1 r^(2i+1)
next i
```

Figure 5.8 demonstrates an optimised square with the operand converted according to $SSW_{2,1}$ during the square. Note that for this special case with the binary digit set $a_i \in \{-1, 0, 1\}$ the terms a_iA_{i+1} are trivially obtained.

Figure 5.8 An Example of Squaring Combined with Sliding Window Conversion. The left hand side of the figure shows the steps in converting *A* according to $SSW_{2,1}$. The right hand side shows the accumulation of partial results according to Equation 5.6.

Optimised Squaring with Unsigned Pre-computed Partial Squares

In Section 2 multiple-precision multiplication was implemented using pre-computed partial products. For example, in multiplication using $USW_{2,m}$ the partial products $\{B, 3B, 5B, (2^m - 1)B\}$ were evaluated once and re-used throughout the multiplication.

It is not at all clear that the same technique can be applied to optimised squaring. At each iteration, A is truncated and a multiple of the truncated result is accumulated. It is not obvious that this multiple can be calculated from a pre-computed, un-truncated value.

Before proceeding it will be helpful to define some new terminology. In the subsequent development the terms $a_i A_{i+1}$ will be referred to as *partial squares.* The partial square jA_i will be given the symbol $A_{i,j}$. This is formed by taking the top $(l-i)$ digits of A and multiplying the result by j:

$$A_{i,j} = jA_i = j \times (A \gg i) .$$

Where multiples of A are pre-computed, they will be called *pre-computed multiples* and are given the symbol $A_{0,j}$:

$$A_{0,j} = j \times A .$$

We will also need discern the original bits of A from the converted digits of A. Let α_i be the ith bit of A at the start of the square and let a_i be the ith digit of A after digit set conversion.

Figure 5.9 demonstrates a first problem with squaring with pre-computation. As the digit set conversion is applied to a_i, the value of A_{i+1} changes. If the partial square $a_i A_{i+1}$ was pre-computed based on the original value of A_{i+1} then it will not be correct following conversion of a_i.

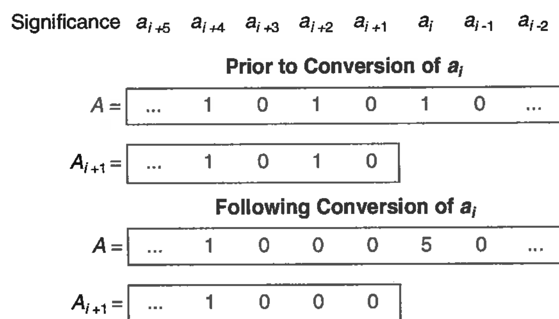

Figure 5.9 A Problem with Pre-computed Partial Squares. The value of A_{i+1} changes following conversion of a_i using $USW_{2,3}$.

If we constrain consideration to $USW_{2,m}$ then this problem is easily dealt with. When a non-zero digit a_i is selected, it is such that the subsequent digits $a_{i+1}, \ldots, a_{i+m-1}$ all become zero. None of

the other digits in A_{i+1} are changed. Following the selection of a_i we have $A_{i+1} = 2^{m-1} \times A_{i+m}$ and A_{i+m} is unchanged by the conversion.

The iteration of Equation 5.6 can be modified so that iteration i accumulates the partial square $a_i A_{i+m}$. This has the advantage that A_{i+m} is unchanged since the beginning of the square. The modified iteration is:

$$A^2 = \sum_{i=0}^{l-1} \left(a_i^2 2^{2i} + A_{i+m,\,a_i} 2^{2i+m+1} \right). \tag{5.7}$$

Now a second problem arises as illustrated in Figure 5.10. Iteration i of the square accumulates the partial square $A_{i+m,\,a_i}$ where this is formed from A shifted $(i+m)$ -bits to the right and then multiplied by a_i. However this is not the same as shifting the pre-computed multiple $A_{0,\,a_i}$ by $(i+m)$ -bits to the right. If pre-computation is to work then it must be possible to quickly determine the partial square $A_{i+m,\,a_i}$ from the pre-computed multiple $A_{0,\,a_i}$.

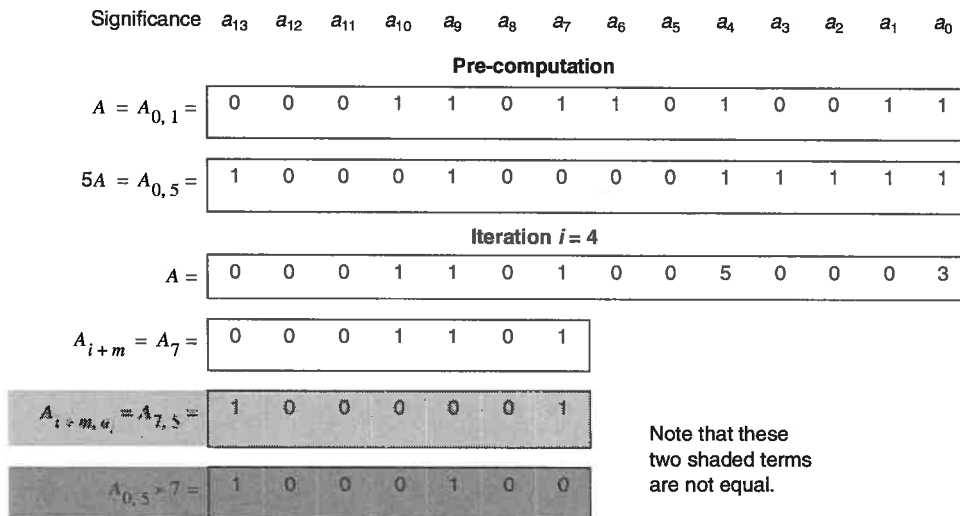

Figure 5.10 A Second Problem with Pre-computed Partial Squares. The diagram shows some of the steps in squaring with $USW_{2,3}$. Note that the value of the partial square $A_{7,5}$ is not a trivial truncation of the pre-computed multiple $5A$.

The problem is now: given a set of pre-computed multiples $A_{0,\,j}$, is it possible to quickly determine the partial squares $A_{i+m,\,a_i}$?

Consider the evaluation of the pre-computed multiple $A_{0,\,a_i}$ for an odd value of a_i from $A_{0,\,a_i} = a_i A_{0,1} = A_{0,1} + \left(\frac{a_i - 1}{2}\right) 2A_{0,1}$ using the binary addition shown in Figure 5.11.

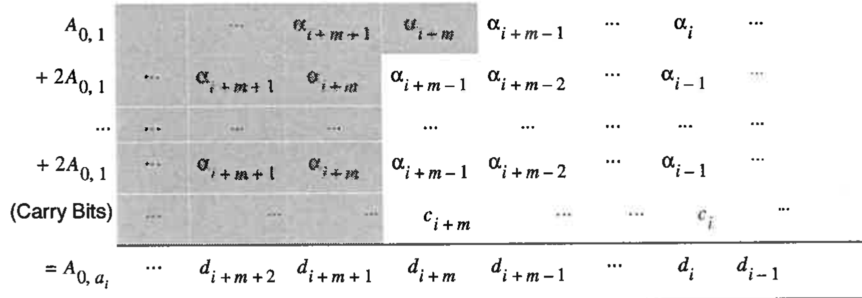

Figure 5.11 Evaluating Partial Squares. A binary addition is shown along with the carry bits that are generated. The shaded terms indicate the addition required to evaluate $A_{i+m,\,a_i}$ from $A_{i+m,\,1}$.

From the diagram, the shaded cells show the addition required to determine $A_{i+m,\,a_i}$. Observe that:

$$A_{i+m,\,a_i} = (A_{0,\,a_i} \gg (i+m)) - \left(\frac{a_i - 1}{2}\right)\alpha_{i+m-1} - c_{i+m}. \tag{5.8}$$

In this equation the term $A_{0,\,a_i} \gg (i+m)$ is readily available from the table of pre-computed partial squares and given that we are using USW $_{2,\,m}$, the bit α_{i+m-1} can be determined from a_i. It remains to compute the unknown carry term c_{i+m}. Figure 5.12 shows the generation of this term.

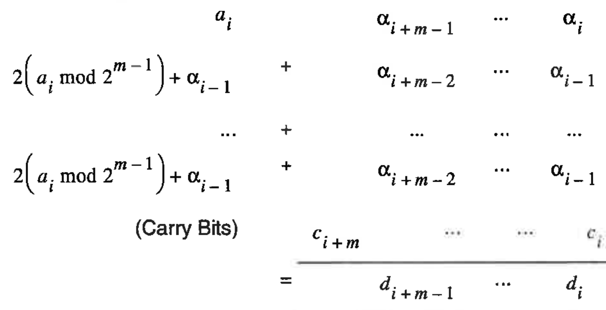

Figure 5.12 The Generation of c_{i+m}. A section of the addition from Figure 5.11 is examined to emphasise the generation of the unknown carry term.

Examination of the addition in Figure 5.12 gives the following equation for c_{i+m}:

$$c_{i+m} = \left(a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right) + \left(\frac{a_i - 1}{2}\right)\alpha_{i-1} + c_{i,\,a_i}\right) \gg m \quad . \tag{5.9}$$

In this equation, the term c_i is unknown and α_{i-1} is not readily available. Let us use $cmin_{i+m}$ to denote the minimum possible value for c_{i+m}. This occurs when $c_i = 0$ and $\alpha_{i-1} = 0$:

$$cmin_{i+m} = \left(a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right)\right) \gg m. \tag{5.10}$$

Similarly, the maximum $cmax_i$ occurs when $\alpha_{i-1} = 1$, and $c_i = (a_i - 1)/2$ where the latter represents the maximum possible carry out from the addition of $(a_i + 1)/2$ binary terms. Thus:

$$cmax_{i+m} = \left(2a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right) - 1\right) \gg m. \tag{5.11}$$

Consideration of Equation 5.10 and Equation 5.11 gives $cmax_{i+m} - cmin_{i+m} \leq 1$. Therefore, application of Equation 5.10 yields a value for c_{i+m} that is either correct or too small by 1.

Returning to Figure 5.11, observe that $d_{i+m} = (\alpha_{i+m} + ((a_i - 1)/2)\alpha_{i+m-1} + c_{i+m}) \bmod 2$ where d_{i+m} is the $(i+m)$ -th bit of A_{0,a_i}. Manipulation of this equation yields:

$$c_{i+m} \bmod 2 = d_{i+m} \oplus \alpha_{i+m} \oplus (\alpha_{i+1} \wedge \alpha_{i+m-1}). \tag{5.12}$$

So Equation 5.10 gives a value for c_{i+m} that is too small by at most 1 and Equation 5.12 reveals if the correct value of c_{i+m} is odd or even. Combining these two determines c_{i+m} exactly.

Although the analysis has been quite involved, it leads to a useful conclusion: given the correction terms c_{i+m} and α_{i+m-1}, the partial square A_{i+m,a_i} can be quickly determined from the pre-computed multiple A_{0,a_i}. The entire procedure for optimised squaring with unsigned sliding windows and pre-computed partial squares is summarised below. Note that the two correction terms of Equation 5.8 have been combined into a single term *t* thus: $t = ((a_i - 1)/2)\alpha_{i+m-1} + c_{i+m}$.

1. Pre-compute the table of partial squares $A_{0,j}$ for $= 1, 3, 5, \ldots, 2^m - 1$ where:

$$A_{0,1} = A \text{ and } A_{0,j+2} = A_{0,j} + 2A_{0,1}.$$

2. Pre-compute a correction table $t_{min}(j)$ for $= 1, 3, 5, \ldots, 2^m - 1$ from:

$$t_{min}(j) = ((j-1)/2)(j \operatorname{div} 2^{m-1}) + (j + (j-1)(j \bmod 2^{m-1})) \operatorname{div} 2^m.$$

3. Apply Algorithm 5.6. At iteration *i* determine the correction term:

$$t = t_{min}(a_i) + d_{i+m} \oplus \alpha_{i+m} \oplus (t_{min}(a_i) \bmod 2)$$

where d_{i+m} is the $(i+m)$ -th bit of A_{0,a_i}. This gives the partial square:

$$A_{i+m,a_i} = (A_{0,a_i} \gg i+m) - t.$$

Algorithm 5.6 Optimised Squaring with $USW_{2,m}$ and Pre-computed Partial Squares. The procedure to determine the partial square $A_{i+m, ai}$ is described above.

```
C = 0
i = 0
while i < l
      if A mod 2 = 0 then
            A = A >> 1
            i = i + 1
      else
            a_i = A mod 2^m
            A = A >> m
            C = C + a_i^2 2^(2i) + A_(i+m,ai) 2^(2i+m+1)
      end if
wend
```

In Chapter 6 this algorithm is successfully implemented on an ARM processor and achieves better execution time than the equivalent sliding window multiplication. In Figure 5.13 this algorithm is used to perform an optimised square using $USW_{2,3}$ with pre-computed partial squares.

Find A^2 using $USW_{2,3}$

Pre-compute Stage. A table of pre-computed multiples is evaluated and stored. The correction values $t_{min}(j)$ can also be determined in advance of the square.

$A = A_{0,1} =$	0	0	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1
$3A = A + 2A = A_{0,3} =$	0	0	1	1	1	0	1	1	0	1	1	0	0	1	0	1	1
$5A = 3A + 2A = A_{0,5} =$	0	1	1	0	0	0	1	0	1	1	1	1	1	1	1	0	1
$7A = 5A + 2A = A_{0,7} =$	1	0	0	0	1	0	1	0	1	0	0	1	0	1	1	1	1

$t_{min}(1) = 0,\ t_{min}(3) = 1,\ t_{min}(5) = 3,\ t_{min}(7) = 6$

Evaluate Stage. $USW_{2,3}$ is applied to determine the non-zero digits a_i. For each non-zero digit a partial square is evaluated by truncating one of the pre-computed multiples and applying a correction. The accumulation of partial squares is shown below.

Apply $USW_{2,3}$ gives $i = 0$, $a_i = 1$, $A =$	0	0	0	1	0	0	1	1	1	1	0	0	1	1	[0	0	1]
$t = t_{min}(1) + d_3 \oplus \alpha_3 \oplus (t_{min}(1) \bmod 2) = 0 + 1\oplus1\oplus0 = 0$																	
Find $A_{3,1} = (A_{0,1} >> 3) - t =$							1	0	0	1	1	1	1	0	0	1	1
Apply $USW_{2,3}$ gives $i = 3$, $a_i = 3$, $A =$	0	0	0	1	0	0	1	1	1	1	0	[0	0	3]	0	0	1
$t = t_{min}(3) + d_6 \oplus \alpha_6 \oplus (t_{min}(3) \bmod 2) = 1 + 1\oplus0\oplus1 = 1$																	
Find $A_{6,3} = (A_{0,3} >> 6) - t =$									1	1	1	0	1	1	0	1	0
Apply $USW_{2,3}$ gives $i = 7$, $a_i = 7$, $A =$	0	0	0	1	0	0	1	[0	0	7]	0	0	0	3	0	0	1
$t = t_{min}(7) + d_{10} \oplus \alpha_{10} \oplus (t_{min}(7) \bmod 2) = 6 + 1\oplus1\oplus0 = 6$																	
Find $A_{10,7} = (A_{0,7} >> 10) - t =$												1	1	1	1	1	1
Apply $USW_{2,3}$ gives $i = 10$, $a_i = 1$, $A =$	0	0	0	1	[0	0	1]	0	0	7	0	0	0	3	0	0	1
$t = t_{min}(1) + d_{13} \oplus \alpha_{13} \oplus (t_{min}(1) \bmod 2) = 0 + 1\oplus1\oplus0 = 0$																	
Find $A_{13,1} = (A_{0,1} >> 13) - t =$																	1
Apply $USW_{2,3}$ gives $i = 16$, $a_i = 1$, $A =$	0	[0	0	1]	0	0	1	0	0	7	0	0	0	3	0	0	1
$t = t_{min}(1) + d_{16} \oplus \alpha_{16} \oplus (t_{min}(1) \bmod 2) = 0 + 0\oplus0\oplus0 = 0$																	
Find $A_{16,1} = (A_{0,1} >> 16) - t =$																	0

Accumulation. The partial squares and digit squares (a_i^2) are accumulated to determine the final result.

$a_0^2 2^0$																											1
$+ A_{3,1} 2^4$													1	0	0	1	1	1	1	0	0	1	1				
$+ a_3^2 2^6$																		1	0	0	1						
$+ A_{6,3} 2^{10}$									1	1	1	0	1	1	0	1	0										
$+ a_7^2 2^{14}$								1	1	0	0	0	1														
$+ A_{10,7} 2^{18}$				1	1	1	1	1	1																		
$+ a_{10}^2 2^{20}$							1																				
$+ A_{13,1} 2^{24}$			1																								
$+ a_{13}^2 2^{26}$	1																										
(carry)		1	1	1	1	1	1	1				1			1	1	1										
$= A^2 =$	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	1

Figure 5.13 Optimised Square using Unsigned Sliding Windows and Pre-computed Partial Squares.

Pre-computed Signed Partial Squares

Yet another complication arises when we consider optimised squaring with signed sliding windows $SSW_{2,m}$. As in multiplication we would like to pre-compute only the positive multiples $\{A, 3A, 5A, (2^m - 1)A\}$. However, as the optimised square proceeds, the partial squares are not only truncated, but also change value as negative digits are introduced. The problem is illustrated in Figure 5.14.

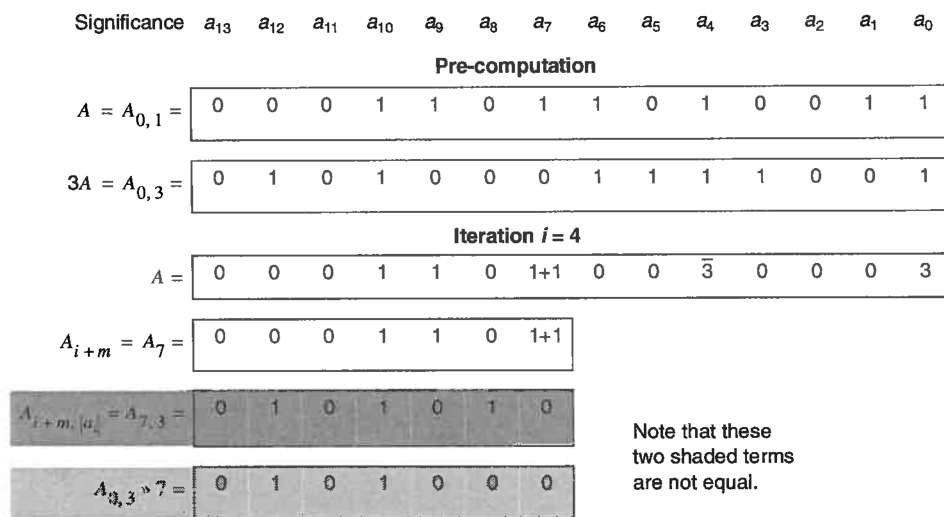

Figure 5.14 A Problem with Pre-computed Partial Squares and Signed Sliding Windows. The diagram shows some of the steps in squaring with $SSW_{2,3}$. Note that the value of the partial square $A_{7,3}$ is not a trivial truncation of the pre-computed multiple $3A$ and must take into account the carry due to selection of a negative digit.

Consider performing a digit set conversion according to $SSW_{2,m}$ during the optimised square. As is the case with $USW_{2,m}$ conversion, at iteration i the digit a_i is selected such that the subsequent digits $a_{i+1}, \ldots, a_{i+m-1}$ all become zero. However, with the signed conversion, a negative value for a_i can be selected, and to correct for this change, a carry is propagated into a_{i+m} and then to the left.

For $USW_{2,m}$ it was found that A_{i+m} was unchanged by the digit set conversion such that $A_{i+m,1} = A_{0,1} \gg (i+m)$. The same is true for $SSW_{2,m}$ with a_i positive. In this case, the partial square A_{i+m,a_i} can be found from Equation 5.8 (repeated below):

$$A_{i+m,a_i} = (A_{0,a_i} \gg (i+m)) - \left(\frac{a_i - 1}{2}\right)\alpha_{i+m-1} - c_{i+m}.$$

Note that in this case the assumptions used to derive the carry term c_{i+m} in the previous section may no longer be valid. A new analysis must be performed and the results are presented below.

In $SSW_{2,m}$ when a_i is negative, a carry is introduced that changes the value of A_{i+m} – let us call the new value A'_{i+m}. The required partial square becomes $A'_{i+m,|a_i|} = |a_i|A'_{i+m,1}$.

Note that $A'_{i+m,1} = (A_{0,1} \gg (i+m)) + 1$ and hence $A'_{i+m,|a_i|} = |a_i| \times (A_{0,1} \gg (i+m)) + |a_i|$. The term $|a_i| \times (A_{0,1} \gg (i+m))$ is the definition of $A_{|a_i|+m,j}$ from the unsigned case. Thus we can substitute Equation 5.8 and obtain:

$$A'_{i+m,|a_i|} = (A_{0,|a_i|} \gg (i+m)) - \left(\frac{|a_i|-1}{2}\right)\alpha_{i+m-1} - c_{i+m} + |a_i|$$

It now remains to derive equations for the carry terms c_{i+m}. This can be done by applying the derivation from $USW_{2,m}$ to the 4 cases outlined below. A summarised procedure for optimised squaring using signed sliding windows and pre-computed multiples is:

1. If $a_i > 0$ and $a_{i-1} > 0$ then:

$$t_{min} = \left(\frac{a_i-1}{2}\right)\alpha_{i+m-1} + \left(a_i + (a_i-1)\left(a_i \bmod 2^{m-1}\right)\right) \operatorname{div} 2^m$$

$t = t_{min} + (t_{min} \bmod 2) \oplus d_{i+m}$ where d_{i+m} is the $(i+m)$-th bit of A_{0,a_i}

$$A_{i+m,a_i} = (A_{0,a_i} \gg (i+m)) - t$$

2. If $a_i > 0$ and $a_{i-1} < 0$ then:

$$t_{min} = \left(\frac{a_i-1}{2}\right)\alpha_{i+m-1} + \left(\frac{3a_i-3}{2} + (a_i-1)\left((a_i-1) \bmod 2^{m-1}\right)\right) \operatorname{div} 2^m$$

$t = t_{min} + (t_{min} \bmod 2) \oplus d_{i+m}$ where d_{i+m} is the $(i+m)$-th bit of A_{0,a_i}

$$A_{i+m,a_i} = (A_{0,a_i} \gg (i+m)) - t$$

3. If $a_i < 0$ and $a_{i-1} > 0$ then:

$$t_{min} = \left(\frac{|a_i|-1}{2}\right)\alpha_{i+m-1} + \left(2^m - |a_i| + (|a_i|-1)\left(\left(2^m - |a_i|\right) \bmod 2^{m-1}\right)\right) \operatorname{div} 2^m$$

$t = t_{min} + (t_{min} \bmod 2) \oplus d_{i+m} \oplus 1$ where d_{i+m} is the $(i+m)$-th bit of $A_{0,|a_i|}$

$$A'_{i+m,|a_i|} = (A_{0,|a_i|} \gg (i+m)) - t + |a_i|$$

4. If $a_i < 0$ and $a_{i-1} < 0$ then:

$$t_{min} = \left(\frac{|a_i|-1}{2}\right)\alpha_{i+m-1} + \left(2^m - \frac{|a_i|+3}{2} + (|a_i|-1)\left(\left(2^m - |a_i| - 1\right) \bmod 2^{m-1}\right)\right) \operatorname{div} 2^m$$

$$t = t_{min} + (t_{min} \bmod 2) \oplus d_{i+m} \oplus 1 \text{ where } d_{i+m} \text{ is the } (i+m)\text{-th bit of } A_{0,|a_i|}$$

$$A'_{i+m,|a_i|} = (A_{0,|a_i|} \gg (i+m)) - t + |a_i|$$

This procedure has been verified by simulation with Maple (the Maple script is presented in Appendix E). In Figure 5.15 it is used to perform an optimised square using $\mathrm{SSW}_{2,3}$ with pre-computed partial squares.

Find A^2 using $\mathrm{SSW}_{2,3}$

Pre-compute Stage. A table of pre-computed multiples is evaluated and stored.

$A = A_{0,1} =$	0	0	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1
$3A = A + 2A = A_{0,3} =$	0	0	1	1	1	0	1	1	0	1	1	0	0	1	0	1	1
$5A = 3A + 2A = A_{0,5} =$	0	1	1	0	0	0	1	0	1	1	1	1	1	1	1	0	1
$7A = 5A + 2A = A_{0,7} =$	1	0	0	0	1	0	1	0	1	0	0	1	0	1	1	1	1

Evaluate Stage. $\mathrm{SSW}_{2,3}$ is applied to determine the non-zero digits a_i. For each non-zero digit a partial square is evaluated by truncating one of the pre-computed multiples and applying a correction. The accumulation of partial squares is shown below.

Apply $\mathrm{SSW}_{2,3}$ gives $i = 0$, $a_i = -7$, $A =$	0	0	0	1	0	0	1	1	1	1	0	1	0	0	[0	0	$\bar{7}$]
$t_{min} = 0$, $t = 0$																	
Find $A'_{3,7} = A_{0,7} >> 3 - t + 7 =$				1	0	0	0	1	0	1	0	1	0	1	1	0	0
Apply $\mathrm{SSW}_{2,3}$ gives $i = 5$, $a_i = -3$, $A =$	0	0	0	1	0	1	0	0	0	[0	0	$\bar{3}$]	0	0	0	0	$\bar{7}$
$t_{min} = 1$, $t = 1$																	
Find $A'_{8,3} = A_{0,3} >> 8 - t + 3 =$											1	1	1	1	0	0	0
Apply $\mathrm{SSW}_{2,3}$ gives $i = 11$, $a_i = 5$, $A =$	0	0	0	[0	0	5]	0	0	0	0	0	$\bar{3}$	0	0	0	0	$\bar{7}$
$t_{min} = 2$, $t = 3$																	
Find $A_{14,5} = A_{0,5} >> 14 - t =$																	0

Accumulation. The partial squares and digit squares (a_i^2) are accumulated to determine the final result.

$a_0^2 2^0$																					1	1	0	0	0	1
$- A_{3,7} 2^4$									$\bar{1}$	0	0	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	0				
$+ a_5^2 2^{10}$													1	0	0	1										
$- A_{8,3} 2^{14}$						$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0														
$+ a_{11}^2 2^{22}$	1	1	0	0	1																					
$+ A_{14,5} 2^{26}$	0																									
(carry)					$\bar{1}$	$\bar{2}$	$\bar{2}$	$\bar{2}$	$\bar{2}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$		$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$							
$= A^2 =$	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	1

Figure 5.15 Optimised Square using Signed Sliding Windows and Pre-computed Partial Squares.

Optimised Squaring and Karatsuba

In Section 2 Karatsuba multiplication was combined with multiplication using signed sliding windows. The result was a small reduction in the total average number of accumulations and a significant reduction in the memory required for pre-computed partial products.

To confirm that the same result carries over to optimised squaring it is sufficient to observe that Karatsuba's technique can be used to implement a n-bit square as three $(n/2)$ -bit squares as in Equation 5.13.

$$A^2 = (2^n - 2^{n/2})\left(A_H^{\ 2}\right) + 2^{n/2}(A_H + A_L)^2 + (1 - 2^{n/2})A_L^{\ 2} \tag{5.13}$$

Evaluating Optimised Squaring with Binary Sliding Windows

Optimised squaring leads to a reduction in the average length of the partial squares. The exact benefit to be derived from this depends strongly on the implementation. The approximate analysis that follows will serve only to indicate the maximum possible benefit that can be expected.

For ordinary binary non-redundant representation the average number of non-zero bits is $n/2$ and for multiplication each partial square is n bits long. Thus for a square, $n^2/2$ bits must be generated and accumulated.

Sliding window digit-set conversion reduces the average number of non-zero digits in a number representation and hence reduces the number of non-zero partial squares to be evaluated and accumulated. For $SSW_{2,m}$ and large n, the average number of non-zero partial squares is $n/(m+2)$. Optimised squaring reduces the average length of the partial squares. At iteration i the uncorrected partial square $(A_{0,|a_i|} \gg i+m)$ is at most $(n-i)$ bits long. For large n the average length of the uncorrected partial squares is approximately $(n/2)$ bits. Thus for the new optimised square $n^2/(2m+4)$ bits must be generated and accumulated.

For example, in the case of $SSW_{2,1}$, the number of bits to generate and accumulate is approximately one third of that required by the ordinary binary square. A suitable architecture could potentially operate up to almost 3-times faster by implementing this new optimised square.

In addition to the truncated multiple $A_{0,|a_i|} \gg (i+m)$, the optimised implementation must also accumulate the digit squares a_i^2 and the correction terms $-((|a_i|-1)/2)\alpha_{i+m-1}$, $-c_{i+m}$ and possibly $|a_i|$. Although these amount to only a few extra bits per non-zero digit, the efficiency with which they can be generated and injected into the accumulation array will be a critical influence on the performance of the optimised square. In particular, the negative terms represent a significant challenge. The implementation in Chapter 6 overcomes this by storing the negative correction terms

as they are generated into an extra $2n$-bit buffer. (Note that the negative correction terms from one iteration do not overlap with those from the next – they can be stored without carry propagation). The correction buffer is subtracted from the partial result at the end of the square. Although this is a time-efficient solution, it does consume extra temporary storage.

4 Summary and Conclusions

This chapter considered the operations of long integer multiplication and squaring on a software cryptography platform. A survey of software implementations of public-key cryptography revealed that contemporary implementations usually assume the existence of a wordlength-by-wordlength multiply instruction. Multiple-precision multiplication using a wordlength multiplier is a well understood problem with an almost standard solution. To optimise for performance many authors consider adjusting the order of partial product accumulation. The best approach of the two options—operand scanning and product scanning—is dependent on the target processor. Multipliers that execute in parallel with other instructions are used whenever available so that multiplications overlap with loop or memory overhead. Karatsuba multiplication is occasionally used to double or quadruple the length of the primitive multiply instruction. Further recursion of Karatsuba is rarely attempted as either memory requirements become a limiting factor or the Karatsuba overhead begins to exceed the benefit. Digit-set conversion, while common practice for hardware implementation, is of little use to an ordinary high-radix multiple-precision multiplication.

Squaring is a critical operation for long wordlength exponentiation. The number of multiplications required for exponentiation can be reduced by pre-computation, but the number of squares is fixed by the length of the exponent. Optimised squaring, which offers a potential 2-fold improvement in execution time over general multiplication, is widely applied. However in practice, the speed-up observed is usually much less than the theoretical maximum.

In the latter sections of this chapter, sliding window digit set conversion was applied to multiplication and optimised squaring. This represents a significant departure from the usual scheme of software implementations. Instead of relying on a fast wordlength-by-wordlength multiplier, the new algorithms decompose multiplication and reduction to a series of long accumulations. This new approach is therefore possible on processors without a hardware multiplier.

To reduce the computational effort, partial products are pre-computed and sliding window conversion is applied. It was found that when partial products must be pre-computed for each multiplication, an

optimal window size exists that minimises the average total accumulations. Karatsuba's multiplication technique also offers some potential benefit to the new scheme. By splitting an n-bit multiplication into three $(n/2)$ -bit multiplications, both the average number of total accumulations and the pre-computation memory requirement are reduced.

In the case of multiplication and squaring, signed sliding windows are a definite advantage over unsigned sliding windows. $\mathrm{SSW}_{2,m}$ exhibits the same number of non-zero digits as $\mathrm{USW}_{2,m+1}$ with half the number of pre-computed partial products. This advantage has been overlooked elsewhere, leading to the (incorrect) conclusion that signed recoding offers no advantage over unsigned sliding windows for large n [KH92].

Sliding window conversion was also applied to optimised squaring with pre-computed partial squares. This is a very novel approach that combines two apparently incompatible techniques. At each iteration, optimised squaring accumulates a partial square. This is correctly obtained by first truncating the number to be squared and then forming a multiple of the result. Incorrect results are obtained if the multiples are pre-computed and then truncated. However, the correct partial squares can be quickly determined from pre-computed multiples if a correction factor is applied.

In the following chapter these algorithms are put to the test when they are implemented in assembly language for a 32-bit smart-card processor.

Chapter 6

A Smart-Card Implementation of RSA

SMART CARDS are a remarkable culmination of the driving technological forces of the past 20 years: computation, miniaturisation and communication. There is no doubt of the important role they will come to play in a world of people who are increasingly networked and increasingly mobile. The importance of cryptography in this environment was discussed in Chapter 3. However, security of electronic information cannot be guaranteed by software and cryptography alone: software running on unsecured hardware cannot be protected from examination or modification. Somewhere in the system there must be a trusted hardware element capable of securely storing, and possibly processing data. Smart-cards meet this need in a portable form that allows individuals to carry their data, proof of identification and permissions with them. This facilitates applications such as:

- identification for access to bank accounts or buildings
- authentication of the right-to-use a resource such as pay-television
- electronic wallets or purses to hold currency credits, facilitate secure transactions and support applications such as customer discounts or customer loyalty schemes
- monitoring of usage or protection from copying of copyright material
- identification of individuals (possibly using biometric techniques)
- privacy of communications.

A smart-card is a special instance of a chip-card: in integrated circuit embedded in a small plastic card. The most basic chip-cards contain only non-volatile memory. More advanced versions provide security functions to protect areas of memory from unauthorised access and tamper-resistant

mechanisms to prevent direct access to the embedded electronics. A smart-card combines non-volatile memory, tamper-resistant hardware mechanisms and a programmable microprocessor. This allows a smart-card to implement a variety of applications or perform secure processing *on-card*.

Smart-Card Limitations

The design of smart-cards is constrained by a number of strict limitations:

- Standards: various standards define the acceptable parameters of smart-cards for different applications. A number of standards are referenced in [VW98].
- Dimensions: standard dimensions for smart-cards vary from the size of a credit card (54×85.6 mm) down to smaller versions (e.g. 15×25 mm). Cards must be only 0.76 ± 0.08 mm thick.
- Contacts: standards exist for cards with metal electrical contacts and also for contactless (radio-interface) cards.
- Supply Voltage: older standards specified a supply voltage of 5 V to the smart-card but newer standards have reduced this to 3 V or 1.8 V.
- Environment: cards must be able to sustain considerable torsion and bending. This limits the size of the chip that can be embedded to around 25 mm^2.
- Power Dissipation: being embedded in plastic limits the maximum thermal dissipation of a smart-card chip and hence the power and consequently maximum clock rate of the processor. The power that can be delivered to the card may also be limited, especially in contactless cards or cards for portable devices.

1 A Survey of Public-Key Smart-Cards

A smart-card capable of storing a user's private keys and performing public-key cryptographic operations in a timely manner opens the door to a wide variety of security applications. Consequently a significant number of commercial public-key smart-cards have been released in recent years. Surveys appear in [NM'R96] and [HP98] and some of the results are summarised in Table 6.1.

A number of trends can be observed from one survey to the next. RAM, ROM and EEPROM capacity, internal clock rate and cryptographic modulus length have all increased. It is reasonable to expect the trend in memory and clock rate to continue; however, demand for further increases in modulus length are likely to ease. For at least the next few years, 1024-bit moduli will be considered acceptable for most applications with 2048-bit recommended for exceptionally high security.

All of the smart-cards surveyed, except the CASCADE device, use an 8-bit microprocessor core coupled with a cryptographic co-processor to perform long wordlength modular arithmetic. This design certainly provides excellent public-key performance with 1024-bit RSA signature generation times falling well below 1 second in the latest devices.

Chip	µP Core	Max Modulus (bits)	RAM (Bytes)	ROM (kB)	NVM (kB)	Internal Clock (MHz)	Line Width (µm)	Chip Area (mm²)	ACP Area (mm²)	512 RSA (ms)	1024 RSA (ms)	Algorithm
Surveyed In [NM'R96]												
ST16CF54	68HC05	768	352	16	16	5	1.2	-	-	150		Montgomery
SLE44C200	80C51	540	256	9	9	5	1	24.5	5.7	60	456	Sedlak
SLE44CR80S	80C51	540	256	17	8	5	0.7	<25	-	60	450	Sedlak
P83C852	80C51	648	256	6	2	10	1.2	22.3	2.5	70		Quisquater
P83C858-FAME	80C51	1024	640	20	8	8	-	-	-	600	500	Quisquater
MC68HC05SC49	68HC05	1024	512	13.3	4	5	1.2	27	5	125	1499	Montgomery
Surveyed In [HP98]												
H8/3113	H8/300	1024	1536	32	16	14.32	0.5	-	-	68	480	
ST19KF16	8-bit	1088	960	32	16	10	0.6	-	-	20	110	
P83W8532	80C51	2048	2304	32	32	8	-	-	-	37	160	
SLE66CX160S	80C51	1100	1280	32	16	7.5	0.6	-	-	37	230	
uPD789828	78K0S	2048	1024	24	8	40	0.35	-	-	16	100	
[Pey95], [Dhe98]												
CASCADE	ARM7M	Software	512	8	32	20	0.8	-	N/A	72	488	Quisquater

RSA times are for signature generation using the CRT where possible.
NVM is Non Volatile Memory - usually EEPROM.

Table 6.1 A Summary of Commercial Smart-Cards.

2 32-bit Smart-Cards

In Chapter 3 a target time of 1 second was set for 1024-bit RSA signature generation with the justification that long public-key operations are only required once or twice for each user transaction. For example, RSA can be used to authenticate a smart-card to a host system and then to exchange session keys for a private-key cryptosystem. From this point on, security and privacy can be maintained using private-key cryptography which is, in general, much faster than its public-key counterpart.

The 1024-bit performance of the latest smart-cards can be seen as overkill for many applications. Perhaps a better approach would be provide a smart-card that is not so heavily optimised for public-key operations but instead provides more general mechanisms to support new and varied applications.

This is one of the arguments behind a push for smart-cards with a 16-bit or 32-bit microprocessor [Pey95], [Dhe98]. There are many strong reasons for this approach:

- Time to Market: processors with support for higher level languages are required to provide faster time to market for smart-card implementations. Existing 8-bit smart-card processors only provide very limited high-level language support and may even be best programmed in assembly language.
- Open Development Tools: secret implementation details and obscure development tools mean that existing smart-card software is usually developed *in-house* by the smart-card manufacturer. More open tools and architectures will allow smart-card vendors to customise software for their own applications or for new services.
- Multiple-application Cards: it is highly desirable that cards be able to support a number of different applications simultaneously. This will allow users to cut down on the number of cards they have to carry. A multiple-application card will require a protected-mode operating system to run software from different vendors at one time. It will also need to be able to run software downloaded into memory rather than hard-wired.
- Processing Cards: smart-cards capable of deciphering multimedia data on-line could be used for secure access to services such as pay-TV or for copyright protection. The 8-bit processor of existing smart-cards is not usually considered adequate for such applications.
- New Applications: voice control, biometrics and other new smart-card applications are computationally intensive and are not feasible with an 8-bit processor.
- New Cryptosystems: existing smart-cards are highly optimised for a certain set of cryptosystems. A smart-card with a general purpose processor will facilitate the implementation of new cryptosystems. It will also be more scalable, implementing cryptosystems at a variety of modulus lengths rather than relying on hardware optimised for particular length operands.

Along with a more powerful general-purpose processor, a smart-card capable of meeting these objectives will require more RAM than the existing generation of smart-cards. We have seen in previous chapters that RAM can have a significant influence on algorithm performance: with extra temporary storage, more intermediate results can be pre-computed. RAM is also useful as a buffer for communications between the smart-card and the host system. Increased buffer size increases the size of communications messages and reduces handshaking overhead. Desirable applications such as biometric algorithms and voice control are also RAM hungry. Finally, a multiple-application operating system will require RAM to store the status of applications, file pointers, session keys and so on.

2.1 The CASCADE Smart-Card

The CASCADE project (European Esprit project EP8670) sought to develop a smart-card based on the 32-bit Advanced Risc Machine ARM7M processor core [Dhe98].

The CASCADE card multiplies the external clock for an internal clock rate of 20 MHz. It has 512-bytes of static RAM, 8 kB of boot ROM, 16 kB of flash memory for data and anther 16 kB of flash memory for programs [Cas98]. The ARM7M core is implemented in a 0.8 µm, 2-metal process and occupies 5.9 mm^2. The 'M' in the 7M indicates that a fast (9 cycle) 64-bit result multiplier is included in the core. This usually occupies an extra 2 mm^2. ARM processors are optimised for power dissipation and the ARM7M on the CASCADE chip dissipates only 118 mW at 20 MHz and 3 V. The ARM core is also fully static allowing the internal clock to be stopped for a very low power stand-by mode.

The software library for the CASCADE chip is described in [Dhe98]. Quisquater's algorithm is used for 1024-bit modular multiplication. Left to right binary exponentiation is performed with unsigned sliding windows ($USW_{2,m}$). It is reported to perform 512-bit and 1024-bit RSA signatures using the CRT in 72 ms and 488 ms respectively.

To achieve adequate cryptographic performance, Dhem assumes a number modifications to the ARM architecture. Firstly:

> We have supposed that the memory accesses take only one cycle. This is not the case on an ARM7M and on most processors but is nevertheless possible with longer pipelines. It also implies a separated bus for data and code or a memory bus working at twice the speed of the CPU. All of this is not the case for ARM7M.

Dhem also assumes that the ARM7M's 32×32-bit multiplier is replaced with an 8×32-bit multiplier capable of performing a 32×32-bit operation in four cycles. The new multiplier is arranged such that the four multiplier cycles do not stall the ARM pipeline – the instructions following a multiply can be executed during the multiplication so long as they do not depend on the result of the multiplication.

Dhem augments the ARM7 with a dedicated register for counting loop iterations. A special branch instruction is added to decrement the loop counter and conditionally branch back into the loop. So that the branch does not stall the pipeline, the two instructions following the branch are always executed, even when the branch is taken.

Conventional Smart-Card	32-bit Smart-Card
8-bit Microprocessor	32-bit Microprocessor
Long Wordlength Co-processor	32-bit ALU
Excellent 1024-bit RSA	Adequate 1024-bit RSA
Poor High Level Language Support	Excellent High Level Language Support
Limited Operating System	Protected Mode Operating System
Obscure Development Tools	Accessible Development Tools
Limited Support for Multiple Applications	Simultaneous Multiple Applications Supported
Inadequate for New Applications e.g. Voice Control, Biometry, Multimedia Processing	Improved Facilities for New Applications
Very Limited RAM (≈1024 bytes)	CASCADE: Very Limited RAM (512 bytes) Proposed: Extended RAM (2 kB or more)
Significant Program ROM (≈ 32 kB)	Less Program ROM (8 kB)
Non Volatile RAM for Data (≈ 16 kB)	Non Volatile RAM for Programs & Data (32 kB)

Table 6.2 Comparing a Conventional Smart-Card and a 32-bit Smart-Card.

3 A New Smart-Card

This section investigates the feasibility of a new 32-bit smart-card. Like the CASCADE card, the new smart-card is based around an ARM processor core—for many of the same reasons. The ARM is optimised for portable applications, exhibits very low power dissipation and occupies a small chip area; it provides protected and user modes of operation suitable for a protected mode operating system; and the ARM instruction set is a good target for many higher level language compliers and achieves excellent code density.

The ARM's RISC instruction set also proves to be excellent for hand coding critical segments in assembly language. Every instruction is conditionally executed and this allows many small branches to be eliminated. The ARM's barrel shifter is very important for the efficient implementation of sliding window algorithms. Using the barrel shifter, one of the operands to a data processing instruction can be shifted by a register specified for the cost of a single internal cycle. Shifts by a fixed (immediate) value are performed with no extra delay.

3.1 The RAM Constraint

A first point of difference between the proposed card and CASCADE's solution will be the approach to RAM. Most algorithms for public-key cryptography on smart-cards, including Dhem's, have as their starting point the severely limited available RAM on a smart-card. This immediately excludes a number of interesting algorithmic alternatives.

It is true that smart-cards have only limited RAM. It has been estimated [VW98] that a cell of RAM takes 16 times the area of an equivalent cell of ROM, and that EEPROM requires 4 times as much area as ROM. However it can been seen in Table 6.1 that smart-card RAM is increasing and will increase further as process line widths decrease.

In the previous section it was argued that extra RAM is useful, not only for public-key cryptography but also for many new and desirable smart-card applications. Therefore, provision will be made for extra RAM in the new design. This may be at the cost of other components. Certainly, any large hardware dedicated to long-wordlength arithmetic will be removed—adequate cryptographic performance much be achieved in software without a co-processor. Also, a fast hardware multiplier will not be used. This means that the processor will be an ARM6 or ARM7 and not the ARM7M. The former two have a multipliers capable of generating a 32-bit result in up to 17 cycles but they do not have the fast, 9 cycle, 64-bit result multiplier of the 7M. This approach—swapping specialised hardware for extra RAM—is consistent with the objective of a smart-card that is useful for a wide variety of applications.

Let us estimate how much extra RAM can be fabricated in the area vacated by the fast multiplier. A standard 6T, dual ported SRAM cell occupies approximately 30 lambdas by 33 lambdas. In 0.8 μm technology this equates to 158.4 mm^2 per bit. An extra 20% area should be added to the entire RAM array to account for overhead for sense amps and decoding. In 0.8 μm technology the fast multiplier occupies 2 mm^2 which gives 1.67 mm^2 for the SRAM array: space enough for approximately 1300 bytes of RAM.

Note that this estimate was performed in obsolete 0.8 μm technology. A new smart-card would be fabricated with a line width of 0.35 μm or less. This will reduce the size of all the smart-card components, including the processor core and the multiplier. For a given chip size even more area will be available for SRAM. It is interesting to note that a 4kB SRAM was recently constructed at the University of Adelaide that occupied 2 mm^2 in 0.35 μm technology.

3.2 Modifications to the ARM Core

A second point of difference with the CASCADE chip will be in the extent to which I will be prepared to propose modifications to the ARM core.

In recent correspondence with the ARM engineers, it has become clear that the ARM processor is already highly optimised and that even apparently simple modifications are rarely possible without serious ramifications elsewhere in the architecture. This does not rule out all changes completely but it does, perhaps, exclude significant assumptions such as Dhem's single cycle memory access.

In subsequent sections cryptographic performance is evaluated on an unmodified ARM core. Improvements that may be possible with modified hardware are then explored.

3.3 Average Case Execution Time

Dhem proposed that the cryptographic functions in his CASCADE library be carefully written to always run at constant worst-case execution time and thereby circumvent the timing attack. There are a number of problems with this approach. Firstly it is an exceptionally difficult task to equalise evaluation times for all inputs: whenever a section of code is conditionally executed 'dummy' instructions must be provided for those cases when code is skipped. Also, some of the ARM instructions—particularly the multiply instructions—complete in a variable number of cycles depending on the inputs. Another source of timing variability is the ARM's conditionally executed instructions that take one internal cycle when skipped but one or more cycles when executed. All of these sources of variation are of significance given that even the smallest timing variations can be exploited by the timing attack. It is also clear that code designed for constant evaluation time will be larger and actually exhibit a longer worst case delay than code that executes in variable time.

A second issue is one of performance. The worst case for a complex operation such as a 1024-bit exponentiation is exceedingly unlikely to occur and will certainly be less than the accumulated worst case times for all of the constituent operations. Adopting worst-case delay severely degrades the performance of the system. It will also exclude many interesting optimisations such as sliding windows that improve average case execution without necessarily improving the worst case.

The new implementation will employ average case timing with blinding and verification to circumvent the timing and fault attacks. The cost of blinding is a few 1024-bit multiplications. Sliding window exponentiation (with $USW_{2,5}$) alone saves over 30 512-bit multiplications on average when compared with worst-case higher-radix left to right exponentiation (with $r = 2^5$). The total saving of average case execution will far outweigh the cost of blinding.

3.4 Algorithmic Differences

Dhem performs multiple-precision arithmetic in the conventional manner. Multiplication and reduction are performed word-by-word with multiplier digits and quotient digits being 32-bits long. The new design will be based on a low-radix approach. Smaller quotient and multiplier digits are chosen and this facilitates the pre-computation of digit multiples. Sliding window recoding techniques are used to reduce the number of non-zero digits. Another implication of the new approach is that performance no longer relies on a fast multiply instruction. Instead, arithmetic is decomposed down to additions and subtractions.

Of course there are costs to balance against the benefit of low radix operation. More iterations are required and this increases loop overheads. Sliding windows mean that partial results are not always aligned on word boundaries in memory and careful shifting must be performed.

4 ARM Implementation

Given the objectives for a new smart-card outlined in the previous section, an important question remains: can adequate 1024-bit RSA performance be achieved by such a smart-card? This question will be answered in this section.

Figure 3.5 in Chapter 3 illustrates the 'big picture' of the 1024-bit RSA signature generation. Functions for Montgomery multiplication and Montgomery reduction at 1024-bits and 512-bits and Montgomery exponentiation at 512-bits must be implemented.

The evaluation of two 512-bit Montgomery powers is critical to the system's performance. Each power will require at least 511 Montgomery squares as well as hundreds of Montgomery multiplications. Clearly these exponentiations account for the majority of the execution time and determine the feasibility of the whole system. The following sections consider the implementation of these exponentiations.

4.1 Exponentiation

Exponentiation is performed using left to right exponentiation with unsigned sliding windows $USW_{2,m}$ as described in Section 4.1 of Chapter 3. Each exponentiation begins with pre-computation of the odd digit powers $\{A^3 \bmod N, A^5 \bmod N, \ldots, A^{2^m-1} \bmod N\}$. This requires one modular square and $2^{m-1}-1$ modular multiplications. Evaluation then proceeds using the algorithm in Algorithm 3.3 which requires $n/(m+1)$ modular multiplications and $n-1$ modular squares.

As seen in Figure 3.2, for $n = 512$ the total number of modular multiplications is minimised when $m = 5$. Table 6.3 shows the number of modular multiplications for values of m up to this minimum.

The execution time of the exponentiation function will be dominated by the multiplication and squaring time with other delays being a very minor consideration at this outer level of the implementation.

Window Size m	Memory Required (Bytes)	Multiplications in Pre-computation	Multiplications in Evaluation	Total Multiplications
1	64	0	256	256
2	128	1	170.7	171.7
3	256	3	128	131
4	512	7	102.4	109.4
5	1024	15	85.3	100.3
6	2048	31	73.1	104.1

Table 6.3 Modular Multiplications versus Memory Requirement. The table shows the average number of modular multiplications required for 512-bit exponentiation at various window lengths *m*. Also shown is the memory required to store the table of 512-bit pre-computed powers including *A*.

4.2 Montgomery Reduction

Montgomery reduction is performed using pre-computed modulus multiples and binary sliding windows as developed in Chapter 4. A reduction function in ARM assembly language has been designed and in is included as Appendix C. This recodes the quotient digits using unsigned sliding windows $USW_{2,m}$. The code for signed sliding windows $SSW_{2,m}$ is easily derived from this, requiring only an extra function to perform subtraction (instead of accumulation) and some small modifications to the quotient selection instructions. The execution time of the unsigned version $USW_{2,m}$ provides an excellent indication of that of the signed version $SSW_{2,m-1}$.

Execution times for the MR function of Appendix C are presented in Table 6.4. Note that this and all the subsequent timing results were obtained using an ARM software simulator with experiments on random inputs. The ARM was configured with 40 ns (25 MHz) cycle times for all cycles (including internal cycles, sequential and non-sequential memory cycles).

The following sections describe important aspects of the ARM code in more detail.

USW Window Size m	SSW Window Size m	SSW Memory Required (Bytes)	Mean Execution Time (μs)	Improved Execution Time (μs)
4	3	256	1003	746
5	4	512	835	621
6	5	1024	718	542
7	6	2048	625	472

Table 6.4 Execution Times for the 512-bit Montgomery Reduction Function. The SSW memory required indicates the storage required for the pre-computed modulus multiples including *N*. The improved execution time is due to the architectural enhancements discussed below.

Separated Reduction

To simplify implementation and evaluation, and to facilitate Karatsuba multiplication, Montgomery reduction has not be interleaved with multiplication or squaring. Subsequent sections discuss the different software routines developed for multiplication and optimised squaring. Removing reduction from these routines and optimising it separately has reduced code size and complexity. It has also made measurement and optimisation of performance more straight-forward.

Separated reduction increases some temporary memory requirements slightly: for a 512-bit modular product a 1024-bit intermediate result is generated before reduction back to 512-bits. Separating reduction and multiplication has cost an extra 64 bytes of temporary storage. However, it is possible that Karatsuba multiplication will reduce the memory required for pre-computed results significantly (as in Table 5.1). For example, 512-bit multiplication performed using $SSW_{2,5}$ with pre-computed partial products and Karatsuba saves 448-bytes compared to the same scheme without Karatsuba.

It is possible that separated multiplication will also increase loop overhead as separate loops are used for multiplication and reduction. However, interleaved multiplication and reduction with sliding windows requires complex looping conditions (as in [Sed88]). The simpler separated implementation may well prove a performance benefit rather than a cost. Certainly, in the experiments conducted in [KAK96] the separated method was competitive in terms of speed. It also yielded the smallest code size.

Pre-computed Modulus Multiples

Reduction using $USW_{2,m}$ is performed with the pre-computed modulus multiples:

$$\{1N, 3N, 5N, \ldots, (2^m - 1)N\} .$$

Equivalent performance is achieved using $SSW_{2,m-1}$ with half the modulus multiples:

$$\{1N, 3N, \ldots, (2^{m-1} - 1)N\} .$$

Note that for an RSA signature scheme the modulus is part of the public key and is likely to be used for a large number of signatures before being changed. Therefore, modulus multiples can be pre-computed once for many signatures. They can also be stored in EEPROM as they are only rarely written.

In this way reduction is different to exponentiation – pre-computation occurs before the signature and is not counted as contributing towards the signature execution time. Exponentiation exhibits an optimal window length where the total evaluation and pre-computation effort is minimised.

Reduction has no such optimal length and execution time continues to improve with increased window length. The limit is the memory required for pre-computed values which grows as a binary power.

Quotient Digit Selection

At iteration i, the Montgomery reduction algorithm (Algorithm 4.6) must choose a quotient digit q_i from the least significant digits of the partial result c_i. The selection is such that accumulation of the modulus multiple q_iN into the partial result C sets c_i to zero. Equation 4.21 shows that q_i can be determined from a digit-by-digit multiplication. Alternatively the trivial quotient digit selection techniques of Section 1.5 in Chapter 4 can be applied.

In software another approach is possible. Having determined q_i the next step for a software implementation is to find the address of the pre-computed modulus multiple q_iN. This may require another calculation or possibly a table look-up. In the MR function of Appendix C a single table look-up is used to find the address of q_iN directly from the least significant digits of the partial result $\tilde{c}_i$. Quotient digit selection is implicit in this table look-up.

An equivalent technique is used for left to right reduction in [SM89].

Final Reduction

On termination, the result of the MR function can be $n + 1$ bits long due to a carry-out from the final modulus accumulation. In this case an extra modulus subtraction is required to reduce the result to n-bits. Alternatively, the software can be designed to handle intermediate results that are not always fully reduced to n bits. In [DK91] the intermediate results remain in the range $[-N,N-1]$ and in [Eld91] the range is $[0,2N-1]$. Either way, the subtraction at the end of each Montgomery reduction is postponed to a single subtraction at the very end of an exponentiation.

Efficient Loop Control

Loop overhead, the extra instructions required to count iterations and conditionally repeat execution or exit the loop, can contribute a significant delay to software implementations. This is especially true in the most deeply nested loops of an algorithm.

For the CASCADE design, Dhem proposes a modification to the ARM: an extra register specifically for loop counting and instructions to update the counter and branch based on the result. Loop unrolling, an alternative software based method of reducing loop overhead, is not used by Dhem. This is because of the extra program space required for unrolled loops, and because unrolling is difficult when the number of iterations it a parameter of the algorithm.

Let us explore loop control for the accumulate function. This function is the innermost loop of the exponentiation: its role is to add a pre-computed 512-bit number into a 1024-bit accumulator at variable significance. A diagram of the operation is shown in Figure 6.1.

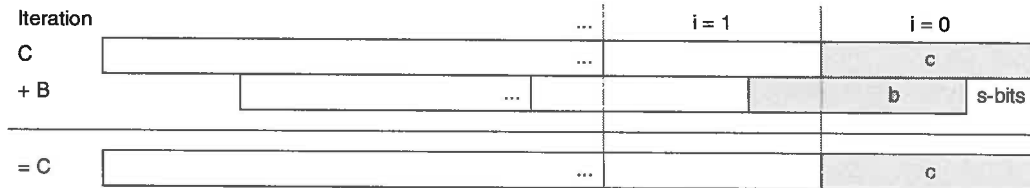

Figure 6.1 The Accumulation Function. The multiple-precision number B is shifted left by s-bits and added to the multiple-precision accumulator C.

This accumulate function is used to accumulate modulus multiples in Montgomery reduction and to accumulate partial products for multiplication. A simple version is given in ARM assembly code in Figure 6.2. The diagram in Figure 6.3 explains the general procedure. For more details on the ARM instruction set refer to [ARM94] and [ARM95].

```
; Accumulate Revision 1  - Unrolled Loop with Simple Control                Cycles
; Accumulate the 16-word number at pB into the number at pC
; at bit significance s.
        MOV     i, #0           ; initialise a loop counter
        MOV     o, #0           ; initialise a carry out register
        RSB     rs, s, #32      ; rs := 32 - s
loop
        LDR     b, [pB], #4     ; read a word of B and update the pointer        3
        LDR     c, [pC]         ; read a word of C                               3
        ADCS    c, c, o         ; accumulate carry out from last iteration       1
        MOV     o, b, LSR rs                                                     2
        ADC     o, o, #0        ; set the carry out                              1
        ADDS    c, b, LSL s     ; add the next word                              2
        STR     c, [pC], #4     ; write the result and update the pointer        2
        ADD     i, i, #1        ; increment the counter                          1
        TEQ     i, #16          ; use TEQ to preserve the carry flag             1
        BNE     loop                                                          3(1)
; Now the remaining bits in o and the c-flag must be accumulated into C
; with any subsequent carry propagated to the left...
```

Figure 6.2 A Simple Loop for Multiple-precision Accumulation.

This simple loop takes 302 cycles for each 32-bit accumulation. Loop overhead including branches and counter incrementing accounts for 78 cycles, or 26% of the total effort.

An apparently obvious enhancement would be to decrement the loop counter *i* and use the zero flag to check for the end condition. Unfortunately this cannot be done without disturbing the carry flag which is passed from one iteration to the next. An alternative is presented in Figure 6.4. In this case the explicit loop counter is replaced with a pointer to the end of the array. The iterations stop when the final word of the array has been accumulated. The memory access instructions automatically update the array pointers and hence implicitly update the loop counter.

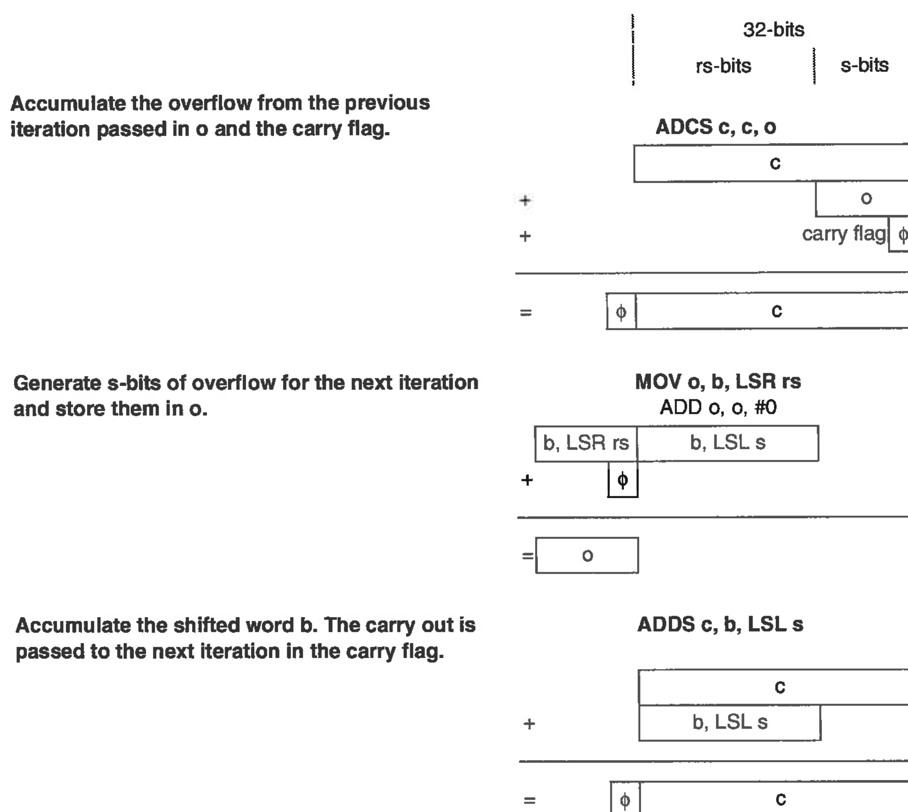

Figure 6.3 Accumulation Procedure. This diagram shows one iteration of the accumulation loop.

Figure 6.4 also includes a second significant enhancement. The loop has been partially unrolled so that each iteration accumulates 4 words instead of 1. Thus only 4 iterations are required to accumulate 16 words (512-bits) and the loop overhead is subsequently reduced. The unrolling also facilitates the use of the ARM's sequential memory access instructions. The *load multiple* instruction, LDM, takes 2 cycles and 1 extra cycle for each register loaded. This is more efficient than a series of *load register* instructions, LDR, that take 3 cycles for each register.

```
; Accumulate Revision 2 - Partial Loop Unrolling                              Cycles
; Accumulate the 16-word number at pB into the number at pC
; at bit significance s.
        MOV     o, #0                   ; initialise a carry out register
        ADDS    eB, pB,#(16*4)          ; point to the MSW of B & zero the c-flag
        RSB     rs, s, #32              ; rs := 32 - s
loop
        LDMIA   pB!, {b1, b2, b3, b4}   ; load 4 words of B, update pB          6
        LDMIA   pC, {c1, c2, c3, c4}    ; load 4 words of C                     6
        ADCS    c1, c1, o                                                       1
        ADCS    c2, c2, b1, LSR rs                                              2
        ADCS    c3, c3, b2, LSR rs                                              2
        ADCS    c4, c4, b3, LSR rs                                              2
        MOV     o, b4, LSR rs                                                   2
        ADC     o, o, #0                                                        1
        ADDS    c1, c1, b1, LSL s                                               2
        ADCS    c2, c2, b2, LSL s                                               2
        ADCS    c3, c3, b3, LSL s                                               2
        ADCS    c4, c4, b4, LSL s                                               2
        STMIA   pC!, {c1, c2, c3, c4}   ; store 4 words of C, update pC         5
        TEQ     pB, eB                  ; use TEQ to preserve the carry flag    1
        BNE     loop                                                         3(1)
; Now the remaining bits in o and the c-flag must be accumulated into C
; with any subsequent carry propagated to the left...
```

Figure 6.4 An Improved Loop for Multiple-Precision Accumulation. The critical loop is partially unrolled to reduce iterations and facilitate sequential memory access. A pointer is used to test for the end condition.

The improved loop takes only 154 cycles for a 32-bit accumulation. This is 96% faster than the simple loop and is possible with an unmodified ARM processor. It is true that the loop unrolling has increased code size: 15 instructions (60 bytes) versus 10 instructions (40 bytes). This cost is trivial when compared with the benefit.

Note that all 16 of the ARM's integer registers are used in this loop (including the program counter and a stack pointer). Further unrolling is not possible without stacking registers and incurring the cost of extra memory accesses and increased code size.

The accumulate routine ACC listed in Appendix C was used to obtain the timing figures that appear in Table 6.4. It is based on the improved loop above.

Hardware Enhancements for Accumulation of Shifted Multiple-Precision Numbers

The improvements in the critical accumulation loop described above were all achieved on an unmodified ARM processor. This section considers possible modifications the processor to improve performance further. The objective is to propose changes that augment the instruction set architecture of the ARM so that the modified device remains backwards compatible. Naturally, the circuits to implement these modifications must be as small as possible – the idea is not to produce a long wordlength co-processor but to 'fine tune' the existing processor for long wordlength arithmetic.

Note that in the accumulate function of Figure 6.4, each word of *B* must be shifted both left and right. This leads to the idea of an enhanced shift mechanism for the ARM. Let us add an extra 32-bit overflow register, *RE,* that will be used for a new extended shift left operation (ESL) but need not be directly accessible to the programmer. The ESL operation is described by the schematic diagram in Figure 6.5. The result is that the bits shifted out from the most significant end of one extended shift left operation are shifted in at the least significant end of the next extended shift left.

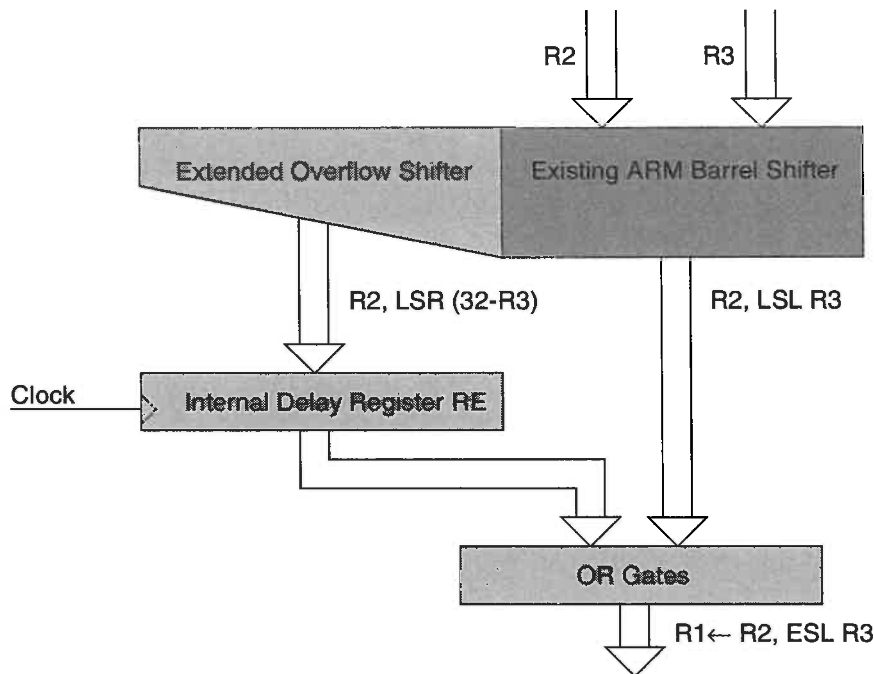

Figure 6.5 Schematic for the Extended Shift Left Mechanism.

Alternatively, the ESL operation can be described in register transfer notation [Man88] thus:

$$R1 \leftarrow (R2 \times 2^{R3}) \vee RE, \; RE \leftarrow \left\lfloor \frac{R2}{2^{R3}} \right\rfloor.$$

Note that the internal *RE* register, while not explicitly accessible, can be set to any value *R2* by performing an ESL operation with $R3 = 32$. For example, the following instructions copy the value of the $a1$ register into the *RE* register (corrupting $a1$ in the process).

```
ADD     a1, a1, a1, ESL #32
```

Similarly, the *RE* register can be moved into the register $a1$ using the following instructions:

```
MOV     a1, #0
ADD     a1, a1, a1, ESL #0
```

As is the case with the other ARM shift instructions, an ESL instruction with a register specified shift will consider only the lower byte of that register. For shifts of greater than 32-bits, the output of the barrel shifter will be zero and the *RE* register will be cleared.

The instruction coding in the ARM is such that it will not be possible to add the ESL shift to all arithmetic and logic operations without sacrificing one of the existing shift mechanisms. Instead, in the code of Appendix D, it is assumed that the ESL is only available on the addition and subtraction instructions (ADD, ADC, SUB, SBC). In this way these instructions can be implemented using some of the ARM's unused (co-processor) instruction space.

Along with the new hardware for extended left shifts, let us also consider the improved branch mechanism as proposed by Dhem. This may be of some benefit even though loops are already partially unrolled. The problem with an ordinary branch instruction is that once the branch condition is calculated, some of the instructions that follow the branch in the code will already be in the processor's pipeline. If the branch is taken, these instructions will have to be flushed from the pipeline before execution can re-commence. A well known improvement [PH96] is to always execute some of the instructions following a branch, irrespective of whether the branch is taken or not.

Dhem suggests that this mechanism could be incorporated into the ARM and proposes two new instructions. The 'MTCTR *n*' instruction moves the number *n* into an internal counter register. The 'BCTRGT *i*' instruction decrements the counter and branches to address *i* if the result is greater than zero. The two instructions that follow the BCTRGT instruction are always executed, irrespective of the value of the internal counter.

A final version of the accumulate loop incorporating these instructions as well as ESL operations is presented in Figure 6.6. The number of cycles per loop has been reduced. Further loop unrolling has also been made possible as the registers *o* and *rs* are no longer required. For a 32-bit accumulation, the new loop would require 3 iterations with a single word accumulated outside the loop. The loop now executes in 103 cycles compared with 154 for the previous revision – a 49.5% improvement in

execution time in the innermost loop of the exponentiation. A loop employing ESL operations without the modified branch instructions executes in 110 cycles, a 40% improvement. The routines presented in Appendix D use the ESL mechanism and the enhanced loop mechanism. Simulation results appear in Table 6.4.

```
; Accumulate Revision 3 - With ARM Hardware Enhancements                      Cycles
; Accumulate the 16-word number at pB into the number at pC
; at bit significance s.
        MTCTR  15                          ; set CTR = 15
loop
        LDMIA  pB!, {b1, b2, b3, b4, b5}   ; load 5 words of B, update pB          7
        LDMIA  pC, {c1, c2, c3, c4, c5}    ; load 5 words of C                     7
        ADCS   c1, c1, b1, ESL s           ; add [b1...b5] shifted left s-bits     2
        ADCS   c2, c2, b2, ESL s                                                   2
        ADCS   c3, c3, b3, ESL s                                                   2
        ADCS   c4, c4, b4, ESL s                                                   2
        BCTRGT      loop                   ; branch to loop in 2 instructions      1
        ADCS   c5, c5, b5, ESL s                                                   2
        STMIA  pC!, {c1, c2, c3, c4, c5}   ; store 5 words of C, update pC         6
        ; A single word remains to be accumulated outside the loop
        LDR    b1, pB                                                              3
        LDR    c1, pC                                                              3
        ADCS   c1, c1, b1, ESL s                                                   2
        STR    pC, c1                                                              2
```

Figure 6.6 Multiple-precision Accumulation with the Hardware Enhancements.

4.3 Multiplication

There is a strong similarity between the Montgomery reduction $MR_N(A)$ and the multiplication $A \times B$. The former accumulates shifted modulus multiples $q_i N$ whereas the latter accumulates shifted partial products $a_i B$. For the RSA signature generation, there is one significant difference between multiplication and reduction. For reduction, the modulus N is constant and the modulus multiples can be computed once and stored in EEPROM. This is not the case for the partial products which must be re-computed for each multiplication.

A product function in ARM assembly language is presented in Appendix C along with a routine to pre-compute the partial products $\{B, 3B, \ldots, (2^m - 1)B\}$. This function recodes the multiplier using unsigned sliding windows $USW_{2,m}$. As was the case with reduction, the code for

multiplication using signed sliding windows $SSW_{2,m}$ is easily derived from the unsigned version. The evaluation time of the unsigned version $USW_{2,m}$ also provides an excellent indication of that of the signed version $SSW_{2,m-1}$ – remembering that the signed version only pre-computes half as many partial products. Execution times are presented in Table 6.4 along with times for the improved versions from Appendix D that use the hardware modifications discussed above.

Window Size m	Memory Required (Bytes)	Mean Pre-compute Time (μs)	Improved Mean Pre-compute Time (μs)	$USW_{2,m}$ Mean Total Time (μs)	$USW_{2,m}$ Improved Mean Total Time (μs)	$SSW_{2,m}$ Mean Total Time (μs)	$SSW_{2,m}$ Improved Mean Total Time (μs)
512-bits							
4	544	37	32	1018	758	859	637
5	1088	73	65	895	670	780	591
6	2176	146	130	853	656	763	583
7	4352	290	260	907	713	838	663
Karatsuba (Three 256-bit Multiplications)							
4	352	63	56	1054	836	890	694
5	640	123	111	950	749	889	667
6	1216	245	222	961	777	872	708

Table 6.5 Execution Times for the 512-bit Product Function. The memory required indicates the storage for the pre-computed partial products including *B* and Karatsuba overhead. Note that an extra 32-bit word is allocated to each partial product to account for the overflow bits. The SSW results are derived from the simulated USW results.

4.4 Optimised Squaring

An optimised square function using $USW_{2,m}$ with pre-computed partial squares is presented in Appendix C. An improved version using extended left shifts and enhanced loop control is to be found in Appendix D. The execution time of the signed version $SSW_{2,m-1}$ is derived from the $USW_{2,m}$ results. Execution times for both appear in Table 6.4.

Window Size m	Memory Required (Bytes)	Mean Pre-compute Time (μs)	Improved Mean Pre-compute Time (μs)	$USW_{2,m}$ Mean Total Time (μs)	$USW_{2,m}$ Improved Mean Total Time (μs)	$SSW_{2,m}$ Mean Total Time (μs)	$SSW_{2,m}$ Improved Mean Total Time (μs)
			512-bits				
4	672	37	32	941	762	793	637
5	1216	73	65	829	670	723	587
6	2304	146	130	796	652	718	588
7	4480	290	260	862	718	798	667
			Karatsuba (Three 256-bit Squares)				
4	416	63	56	1205	976	994	826
5	704	123	111	1054	881	921	776
6	1344	245	222	1043	887	943	804

Table 6.6 Execution Times for the 512-bit Square Function. The memory required indicates the storage for the pre-computed partial squares including B and Karatsuba overhead. Note that an extra 32-bit word is allocated to each partial square to account for the overflow bits. An extra $2n$-bit temporary buffer is also required for negative correction terms. The SSW results are derived from the simulated USW results.

4.5 Signature Generation Time

Let us begin by determining the best signature time that can be achieved on an unmodified ARM, with little regard to memory. The critical operations are shown in Table 6.7. The overhead operations include the N-residue transformations, CRT operations and blinding shown in Figure 3.6 (but do not include verification or generation of a hash value for signing). The 1024-bit blinding operations are each estimated to be the equivalent of 4 512-bit modular products. The overhead memory is required for storage of the two blinding parameters. Note that the total time could be further reduced by choosing a longer window for the reduction algorithm. Also notice that the RAM used for pre-computed partial squares can be re-used for the pre-computed partial products.

Operation	Number of Times Executed	Recoding	Memory (Bytes)	Average Performance
Overhead	1		256 RAM	12 Multiplications 13 Reductions
Exponentiation	2	$USW_{2,5}$	1024 RAM	100.3 Multiplications 511 Squares 611.3 Reductions
Reduction	1235.6	$SSW_{2,5}$	1024 EEPROM	718 μs
Multiplication	212.6	$SSW_{2,6}$	2176 RAM	763 μs
Squaring	1022	$SSW_{2,6}$	2304 RAM	718 μs
1024-bit RSA Signature Generation			**3584 RAM 1024 EEPROM**	**1.78 seconds**

Table 6.7 Unmodified ARM Signature Generation Time. The multiplication, squaring and exponentiation windows are chosen to be optimal length.

This is a good result. Although the target time of 1 second has not been met, 0.78 seconds is not too much longer for a user to wait for a transaction to be processed. The RAM requirement is high but can be reduced by choosing some slightly less than optimal window lengths as in Table 6.7.

Operation	Number of Times Executed	Recoding	Memory (Bytes)	Average Performance (Unmodified ARM)	Improved Performance (Modified ARM)
Overhead	1		256 RAM	12 Multiplications 13 Reductions	
Exponentiation	2	$USW_{2,4}$	512 RAM	109.4 Multiplications 511 Squares 620.4 Reductions	
Reduction	1253.8	$SSW_{2,5}$	1024 EEPROM	718 μs	542 μs
Multiplication	230.8	$SSW_{2,5}$	1088 RAM	780 μs	591 μs
Squaring	1022	$SSW_{2,5}$	1216 RAM	723 μs	587 μs
1024-bit RSA Signature Generation			**1984 RAM 1024 EEPROM**	**1.82 seconds**	**1.42 seconds**

Table 6.8 Improved ARM Signature Generation Time. Sub-optimal window lengths have been selected to reduce RAM requirements.

A very small cost in terms of user time has resulted in a significant saving in memory requirement. This configuration is safely within the bounds of feasibility. In Section 3.1 it was calculated that elimination of the fast multiplier from the CASCADE smart-card would provide space for an extra 1300 bytes of RAM – and this does not account for improvements in fabrication line-widths.

Table 6.7 also shows the case where the ARM processor is modified to include the hardware mechanisms described in Section 4.2. In this case 1024-bit RSA signatures are generated in 1.42 seconds.

Further Modifications to the ARM

It is interesting to note that the optimal window lengths for squaring and multiplication are not $m = 5$ as predicted in Chapter 5. One reason for this is that in practice, the accumulations used for pre-computation are faster than those in evaluation. This is largely because during evaluation, accumulation occurs at a register specified offset. Shifted data processing instructions on the ARM take 1 cycle when the shift is by an immediate value but 2 cycles when the shift is register specified.

Presumably the extra cycle for register specified shifts is to allow for extra instruction decoding and register access time. One possible way of eliminating this extra cycle would be to incorporate a dedicated register to specify the shift amount. For example, the following code copies the value of register *s* to a new *ESR* register. An extended left shift operation is then performed in which the shift amount is contained in the ESR register.

```
MTESR    s                     // 1 cycle to copy s to ESR
ADD      a1, a2, a3, ESL ESR   // 1 cycle for a1 := a2 + (a3 ESL s)
```

Table 6.4 shows the simulation results for the case where this mechanism is available. This time a signature generation time of 1.2 seconds has been achieved using less that 2.5 kB of RAM and 2 kB of EEPROM.

	Reduction			Multiplication			Squaring		
Window Size m	Memory Required (Bytes)	$USW_{2,m}$ Mean Execution Time (μs)	$SSW_{2,m}$ Mean Execution Time (μs)	Memory Required (Bytes)	$USW_{2,m}$ Mean Total Time (μs)	$SSW_{2,m}$ Mean Total Time (μs)	Memory Required (Bytes)	$USW_{2,m}$ Mean Total Time (μs)	$SSW_{2,m}$ Mean Total Time (μs)
				512-bits					
4	512	680	569	544	685	587	672	730	618
5	1024	569	492	1088	620	542	1216	651	567
6	2048	492	432	2176	607	546	2304	632	563
7	4096	432		4352	676	630	4480	693	645
				Karatsuba (Three 256-bit operations)					
4				352	763	655	416	963	805
5				640	710	626	704	860	759
6				1216	737	673	1280	870	789

Table 6.9 Improved Execution Times with the ESR Shift Register.

Operation	Number of Times Executed	Recoding	Memory (Bytes)	Average Performance
Overhead	1		256 RAM	12 Multiplications 13 Reductions
Exponentiation	2	$USW_{2,5}$	1024 RAM	100.3 Multiplications 511 Squares 611.3 Reductions
Reduction	1235.6	$SSW_{2,6}$	2048 EEPROM	432 μs
Multiplication	212.6	$SSW_{2,5}$	1088 RAM	542 μs
Squaring	1022	$SSW_{2,5}$	1216 RAM	542 μs
1024-bit RSA Signature Generation			**2496 RAM 2048 EEPROM**	**1.2 seconds**

Table 6.10 Further Improved ARM Signature Generation Time.

Karatsuba Operations

Before proceeding, some attention should be given to the results of Karatsuba multiplication and squaring. In general, the Karatsuba implementations were competitive without ever offering an improvement in execution time for a given memory size. From this it is possible to conclude that the overhead involved in calling three 256-bit operations outweighs the benefit Karatsuba has over a single 512-bit operation. This will not necessarily always be the case. It is possible that Karatsuba implementations will be advantageous for longer moduli, where function overhead is negligible compared with the accumulation effort.

5 Summary and Conclusions

In this final chapter the arithmetic algorithms and techniques developed in previous chapters were applied to a smart-card implementation of 1024-bit RSA signature generation.

The chapter began with a brief examination of smart-card technology. Applications of smart-cards were discussed, as were the current limitations of the technology. A survey of existing commercial cryptographic smart-cards revealed some important trends: available memory is increasing as is internal clock rate.

All of the commercial cryptographic smart-cards surveyed were found to be based on an 8-bit microprocessor with a long wordlength co-processor to accelerate cryptographic functions. In Section 2 an alternative architecture was discussed. This alternative, championed by the CASCADE smart-card, employs a 32-bit general purpose RISC processor with no cryptographic co-processor.

This approach has many advantages, including the potential to support new smart-card applications such as biometry or multiple-application cards.

In Section 3 a new smart-card was proposed. Like the CASCADE, the proposed card will use a 32-bit ARM processor. However, the new card will not implement a fast multiplier, choosing instead to provide more RAM for temporary results. This approach is consistent with the aim of providing a smart-card suited to a wide range of applications. (Other differences with the CASCADE solution are summarised in Table 6.11.)

An implementation of the critical functions for 1024-bit RSA signature generation was described in Section 4. This implementation takes advantage of sliding window digit set conversion to achieve satisfactory performance without a cryptographic co-processor or a fast multiplier. It was found that signature generation can be performed in around 1.8 seconds on an unmodified ARM processor at 25 MHz. Modifications to the ARM were proposed that reduce signature time to 1.2 seconds. This is sufficient to claim that the target time of 1 second has been met – 200ms is hardly noticeable to a smart-card user. The memory required to achieve this level of performance, 2.5 kB of RAM and 2 kB of EEPROM, is well within the realms of possibility. It was estimated that elimination of the fast multiplier from the CASCADE card would make sufficient space available for 1300 bytes of memory – using obsolete 0.8 μm technology.

CASCADE Smart-Card	Proposed Smart-Card
Very Limited RAM (512 bytes)	Extended RAM (2 kB or more) R More pre-computed and temporary values R Larger communications buffers Q Extra chip area required
Heavily Modified ARM Processor (1 Cycle Memory Access, Pipelined Multiplier)	Unmodified or Lightly Modified ARM Processor
Fast 64-bit Result Multiplier	Small 32-bit Result Multiplier Q Slower 32 by 32-bit multiplication R Considerable saving in chip area R Sliding window algorithms do not use the multiplier
Modular Reduction with Quisquater's Algorithm	Modular Reduction with Montgomery's Algorithm
Constant Exponentiation Timing	Variable Exponentiation Timing Q Extra blinding and verification operations R Average case delay R Algorithmic optimisations improve average case R Blinding combined with Montgomery transform R Not susceptible to latent faults or unexpected timing variations
32-bit Multiple-precision Arithmetic	Low Radix Sliding Window Arithmetic Q More iterations Q Operand words require shifting R Recoding reduces iterations R Barrel shifter and ESL reduces shifting cost R Pre-computation of modulus multiples R Efficient re-computation of partial products R Improved average case without a fast multiplier
Interleaved Multiplication and Reduction	Separated Multiplication and Reduction Q Slightly larger intermediate results R Sliding window reduction with simple loop control R Facilitates Karatsuba multiplication
Loops Unrolled	Partial Loop Unrolling Q Slightly larger code R Sequential memory access R Reduced loop overhead

Table 6.11 Comparing the CASCADE Smart-Card with the Proposed Smart-Card.

Chapter 7

Conclusion

PUBLIC KEY TRANSACTIONS can be performed in a timely manner by a 32-bit smart-card without a cryptographic co-processor or a fast multiplier. This outcome was the foremost objective of this thesis and was demonstrated through the implementation of a benchmark operation on a viable 32-bit architecture.

The solution developed in this work challenges many of the assertions of conventional smart-card design. This is necessary if smart-card technology is to attain its full potential. An ideal smart-card will be useful for a wide variety of different applications, and support multiple applications from different vendors simultaneously. It will use biometry for secure authentication of user identity and it may use voice recognition to simplify the user interface. None of this is possible with an 8-bit processor and severely limited RAM.

Some of the existing components of a conventional smart-card will need to be omitted if space is to be found for a 32-bit processor and extra RAM. It is my proposal that the cryptographic co-processor and fast multiplier should give way. This raises the question of whether adequate cryptographic performance can be achieved without these components. However, when the constraint of severely limited RAM is relaxed somewhat, the door opens for an array of algorithmic enhancements.

Advances in semiconductor manufacture mean that the constraint of limited RAM in smart-cards is already changing. However power consumption still limits the maximum internal clock rate. Hence it is preferable to improve cryptographic performance through the utilisation of extra RAM and

enhanced arithmetic than rely on increased the clock rate. Improvements of this kind were sought and have been developed in this thesis.

Average case execution time is an important aspect of the solution that has emerged. In the face of the timing attack, other implementations have relied on constant worst-case delay. This is a problematic approach: inefficient, difficult to establish and prone to unexpected sources of timing variation. Blinding provides a better solution. For the cost of two extra stored parameters and a few extra modular multiplications, the exact operands values are hidden from an attacker. When combined with result verification this circumvents the timing attack, the fault attack, and a new combined attack. When Montgomery reduction is used with blinding, it is possible to combine the Montgomery N-residue transformations with blinding operations to reduce the total overhead.

One way to improve average case performance is to take advantage of the simplifications possible when zero digits arise in the representation of a number. Extending this idea, one can seek to change the representation of numbers to increase the probability of zero digits. This kind of enhancement does not provide a quantum leap in the algorithmic complexity of a problem yet still improves system performance by reducing the coefficient terms in the complexity equation. This is a valid approach for the arithmetic of public-key cryptography which is based on fundamental operations such a multiplication, exponentiation and modular reduction. A quantum leap in the implementation complexity of these operations is either unlikely or provably impossible.

It is also reasonable to optimise the average case execution time of a complicated operation such a public-key signature generation. The actual worst case of a variable-time implementation will almost certainly be less than the trivial worst case obtained when each iteration takes the longest time possible. With large operands the worst case is also very unlikely to occur. Therefore, average case execution time represents more closely a user's experience of the system.

Digit set conversion to improve average arithmetic weight has been proposed many times in the literature of computer arithmetic. A few digit set conversion schemes have been re-invented a number of times and occasionally even characterised incorrectly. The work in this thesis on digit set conversion, and sliding windows in particular, brings together many fragmented studies under a more general framework. Sliding windows have been extended to radices beyond 2 and to both signed and unsigned digits. Results have been derived for some minimal cases, including the average arithmetic weight, distribution of arithmetic weights and proof of minimality. Some questions remain concerning the completely general case of digit set conversion to an arbitrary, redundant,

non-contiguous, digit set. However, the terminology, notation and techniques developed in this thesis have made these problems more explicit and provide an excellent foundation for further work.

The use of digit set conversion for improved average case execution was demonstrated for hardware (an improved SRT divider) and also in software. Sliding window conversions were successfully applied to long integer reduction, multiplication and optimised squaring. It was found that signed sliding windows have an advantage for these operations, reducing the arithmetic weight for a given number of digit magnitudes.

Signed sliding windows accelerate multiplication by reducing the average number of partial products to accumulate, reducing the pre-computation memory, or simplifying the generation of partial products. These same advantages can be obtained when sliding windows are applied to optimised squaring. This is a surprising result as sliding windows and optimised squaring appear to be incompatible techniques. Karatsuba's optimisation can be used with the new multiplication and squaring algorithms and is likely to reduce the pre-computation memory required for a given level of performance—especially at very long wordlengths.

A thorough survey of algorithms for modular reduction identified the various algorithms already published and also succeeded in classifying them in such a way as to make their similarities and differences more apparent. Four classes of reduction were identified and their history traced. It was found that exact comparisons between the various methods is very difficult: they have all evolved to a point where the performance of each is comparable with the others. Montgomery reduction does seem to possess an advantage for hardware implementation (as it can be efficiently implemented without redundant adders). In software there does not yet appear to be an algorithm that is superior for all platforms.

Montgomery reduction with sliding window quotient selection can be used to accelerate interleaved modular multiplication. Both the sliding window multiplication and reduction will exhibit the same average number of iterations allowing them to be efficiently interleaved (as in Sedlak's processor). An alternative, separated scheme was also devised. Based on Takagi's triangle additions this eliminates the double-length intermediate product from separated Montgomery multiplication.

The simulation of the sliding window algorithms on an ARM processor demonstrated that benefits predicted in theory can be obtained by a real application. Potential difficulties such as shifted operands and loop overhead were all overcome on the ARM platform. It was shown that the performance of sliding window algorithms can be enhanced considerably with a few small modifications to the ARM.

Simulation of the arithmetic functions also demonstrated the viability of public-key cryptography on a 32-bit platform. At 25 MHz a 1024-bit signature time of between 1.2 and 1.8 seconds was achieved. This is acceptable given that this operation represents one of the slowest and most secure public-key transactions that a smart-card is likely to perform—only a few operations of this magnitude are likely per user transaction. The memory requirements were also acceptable with the blinded signature requiring less than 2.5 kB of temporary RAM and 2 kB of EEPROM.

The Future and the Present

The solution presented in this thesis is feasible with current technology and adequate for current demands. In the future there will be requirements for smart-card cryptography with moduli longer than 1024-bits and signature times less than 1 second. Increased clock rate may account for some of the required acceleration. Increased RAM will also contribute towards this end, allowing more pre-computed results and possibly recursive application of Karatsuba multiplication.

Cryptographic co-processors may also play a role in future smart-cards. As fabrication technology improves it not unreasonable to imagine that there will be enough space on a smart-card to support a long wordlength, general purpose processor, sufficient RAM, and a cryptographic co-processor. This does not spell an end to the algorithms developed in this thesis: they are as applicable to the design of a hardware co-processor as to a software library.

For the present, it will be interesting to establish with more certainty if a 32-bit smart-card can deliver some of the benefits it promises. Can the arithmetic of biometry be performed on such a platform? It will also be worthwhile considering further the architecture of processors designed for smart-cards. In what ways can they be optimised to best suit smart-card applications?

There are some loose ends to tie in the general study of digit set conversion. However further results may be of largely theoretical interest. The results concerning the minimal digit set conversions already provide an indication of the maximum benefit that can be obtained from this approach. For example, even though a minimal *k-SR* conversion algorithm is not known, it is certain that *k-SR* conversion can have an average weight no better than $\mathrm{USW}_{2,\lceil \log_2 k \rceil}$ and no worse than $\mathrm{USW}_{2,\lceil \log_2 k \rceil - 1}$.

Publications

B. J. Phillips and N. Burgess,
"A Self-Timed Approach to Radix 2 SRT Quotient Digit Selection for GaAs VLSI Technology",
13th Australian Microelectronics Conference,
Adelaide, Australia, July 1995.

B. J. Phillips and N. Burgess,
"Implementing 1024-bit RSA Exponentiation on a 32-bit Processor Core",
12th IEEE Conference on Application Specific Systems, Architectures and Processors,
Boston, Massachusetts, July 2000.

B. J. Phillips and N. Burgess,
"Optimised Squaring with Sliding Windows",
34th Asilomar Conference on Signals, Systems and Computers,
Monterey, California, October 2000.

B.J. Phillips and N. Burgess,
"Signed Sliding Window Algorithms for Modulo Multiplication",
Electronics Letters, Volume 36, Number 23, November 2000.

Appendix A

A Survey of Modular Multipliers

A SURVEY OF published implementations of modular reduction or modular multiplication has been conducted and a summary of the results appears in Table 7.1. The survey was primarily concerned with algorithmic details. A significant number of papers report the implementation of a cryptographic system but do not provide any details of the arithmetic algorithms. These do not appear in Table 7.1. Other surveys providing a more general view of implementation issues appear in [Riv85], [Bri90], [NM'R96] and [HP98].

The reduction algorithm has been classified according to the four categories described in Chapter 4.

The implementation platform has been categorised as:

- **Bit Serial Hardware:** operands are shifted into an evaluation array and the result emerges bit by bit.
- ***l*-bit Hardware:** iterative processing of n-bit operands by an arithmetic evaluation unit of wordlength less than n-bits.
- ***n*-bit Hardware:** iterative processing of n-bit operands by an n-bit evaluation unit.
- **Array Hardware:** parallel processing of n-bit operands by a array of n-bit evaluation units.
- **Software:** evaluation on a processor with an arithmetic datapath of wordlength less than n-bits.

The following techniques are also identified:

- **Interleaved:** multiplication and reduction are interleaved so that the intermediate results remain approximately n bits long.

- **Quotient Estimation:** the quotient is estimated from only the most significant bits of the partial remainder.
- **Carry Free:** a redundant representation is used to facilitate carry free addition and subtraction.
- **Higher Radix:** quotient digits or multiplier digits are chosen from a radix higher than 2 to reduce the number of iterations.
- **Single Reduction:** a single modulus multiple is subtracted per iteration.
- **Quotient Pipelining:** quotient digits are selected early enough that the generation (and possibly accumulation) of modulus multiples does not appear on the critical evaluation path.
- **R1:** if the multiplier is not in non-redundant form then convert it 'on-the-fly' to simplify the formation of partial products.
- **R2:** use the same redundant form for both operands and results so that conversion is not required during a modular exponentiation.
- **C:** considers a change the order in which operand digits are scanned.
- **D:** considers a change the degree of interleaving.
- **S:** use sliding window digit conversion.
- **P:** digit multiplies of the modulus or the multiplier are pre-computed and stored. (For reduction using the multiplication by the inverse, it is always assumed that the inverse is pre-computed and stored.)
- **Q1:** transform the operands so that quotient digit selection is independent of the current partial product.
- **Q2:** transform the operands so that quotient digit selection is independent of the modulus.

Reference	Reduction Category	Implementation	Int	Q Est	Par Add	Hi Rad	One Red	Q Pip		Notes
[QC82]	Residues	Theoretical	Yes		No	No				Pre-compute Multiplicand Residues
[KH88]	Residues	*n*-bit Hardware	No		No	Yes				Pre-compute Residues
[SM89]	Residues	Software	Yes		No	Yes				Pre-compute Multiplicand Residues
[FJ90]	Residues	Bit Serial	No		No	No				Recursively Evaluate Residues
		Software	No		No	Yes				
[Tom89]	Residues	Bit Serial	Yes		No	No				Pre-compute 4 Residues Only
[KTS91]	Residues	Software	Yes		No	No				Pre-compute Multiplicand Residues
[HOY96]	Residues	Software	Yes		No	Yes				Pre-compute Multiplicand Residues
[QC82]	Remainder	Theoretical	Yes	No	No	No	No	No		
[Bla83]	Remainder	Software	Yes	No	No	No	No	No		Restoring Division
[Bri83]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	No	Q1	
[ORS+86]	Remainder	*n*-bit Hardware	Yes	No	No	Yes	No	No		Binary Reduction, Synchronous & Async.
[Bak87]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	No		

Table 7.1 A Survey of Modular Multipliers.

Reference	Reduction Category	Implementation	Int	Q Est	Par Add	Hi Rad	One Red	Q Pip		Notes
[HDVG88]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	No	P	
[Sed88]	Remainder	*n*-bit Hardware	Yes	Yes	No	No	Yes	No	S	
[SM89]	Remainder	Software	No	No	No	Yes	N/A	No	P	Pre-compute Modulus Multiples
[ICHO89]	Remainder	*n*-bit Hardware	Yes	No	No	No	No	No		Carry Completion Adder
[KH90b]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	No	No		
[KWH90]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	No	No	P	
[FDG90]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	No	No	R2	Right to Left Multiplication
[VVDJ90]	Remainder	*n*-bit Hardware	No	Yes	Yes	No	N/A	No		
[Mor90]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	Yes	No	R2	Radix 4 Division & Booth Multiply
[TB91]	Remainder	*n*-bit Hardware	Yes	No	No	Yes	No	No		High Radix Multiplication, Binary Reduction
[OK91]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	Yes	No	Q1	Radix 32, Fast Digit Multiple Generation
[Wal91]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	Yes	No		
[Wal92]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	Yes	Q1, Q2	Operand Scaling
[TY92]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	No	No	R2	Reduction for Every Shift or Add
[Tak92]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	Yes	Yes	No	R2	
[IWSD92]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	No	No		Asynchronous Datapath
[Chi93b]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	No		Intermediate Results In $(0,2^n)$
[EW93]	Remainder	*n*-bit Hardware	Yes	Yes	Yes	No	Yes	No		
[Tak93]	Remainder	Software	No	Yes	No	Yes	N/A	No		Triangle Add, Wordlength Divider
[AM95]	Remainder	*l*-bit Hardware	Yes	Yes	No	Yes	Yes	No		
[KH98]	Remainder	Theoretical	Yes	Yes	Yes	No	Yes	No	Q1	
[NS81]	Inverse	*n*-bit Hardware	No		No	No				Two Cascaded Multipliers
[Bar87]	Inverse	Software	No		No	Yes			C	Convolution Sum
[PP90]	Inverse	*n*-bit Hardware	No		Yes	Yes				Partial Array Multiplication / High Radix
[AM91]	Inverse	Array (Small *n*)	No							Uses Multipliers as a Primitive Operation
[Wal94]	Inverse	Array	No		Yes	No				
[Dhe98]	Inverse	Software	Yes		No	Yes				
[DK91]	Montgomery	Software	Yes		No	Yes		No	C	Convolution Sum, Opt. Square, Control Errors
[Eve91]	Montgomery	Bit Serial	No		No	No		No		Systolic Array
[Wal93]	Montgomery	Bit Serial	Yes		No	Yes		No	Q1, Q2	Systolic Array
[EW93]	Montgomery	*n*-bit Hardware	Yes		Yes	No		No	R1	
[Kor93b]	Montgomery	*n*-bit Hardware	Yes		Yes	Yes		No	Q1, R2	
[Kor94b]	Montgomery	Bit Serial	Yes		No	No		No		Systolic Array
[Oru95]	Montgomery	*n*-bit Hardware	Yes		Yes	Yes		Yes	Q1, Q2	
[Wal95]	Montgomery	*n*-bit Hardware	Yes		Yes	No		Yes	Q1,Q2,R1	
[KAK96]	Montgomery	Software	Y/N		No	Yes		No	C,D	
[SV93]	Montgomery	*n*-bit Hardware	Yes		No	Yes		Yes		Carry Completion Adder
[SV93]	Montgomery	Software	Yes		No	Yes		No		Karatsuba 64x64 & Opt. Square

Table 7.1 **A Survey of Modular Multipliers.**

Appendix B

Analysis of Some Digit Set Conversions

NEW RESULTS concerning the digit set and average arithmetic weight for some existing digit set conversions appear in Chapter 2. This appendix provides an overview of the derivation of these results.

1 Hwang's Radix-r Canonical Conversion

A digit set conversion mentioned in [Hwa79] involves conversion to the binary canonic form followed by conversion to radix $r = 2^m$ by grouping digits on m-bit boundaries. The digit set and average arithmetic weight for this conversion were not published.

1.1 Digit Set

The digit set resulting from this conversion will be the set of all integers that can be represented with an m-bit canonic binary representation.

The maximum m-bit canonic binary representation $i(m)$ is $1010\ldots1$ for m odd and $1010\ldots0$ for m even (recall that in the canonic binary form no two adjacent digits can be non-zero).

The minimum m-bit canonic binary representation is $\bar{1}0\bar{1}0\ldots\bar{1}$ for m odd and $\bar{1}0\bar{1}0\ldots0$ for m even. That these digits can appear in a final representation is demonstrated by the binary canonic conversion of $\ldots10101011$ to $\ldots0\bar{1}0\bar{1}0\bar{1}0\bar{1}$.

For m odd $(m+1)$ is even and the maximum value for a $(m+1)$ -bit representation is $2i(m)$. For m even $(m+1)$ is odd and the maximum value for a $(m+1)$ -bit representation is $2i(m)+1$. Therefore:

$$i(m+1) = 2i(m) + \frac{1-(-1)^{m+1}}{2}.$$

The closed form of this recursive equation can be found by taking z-transforms:

$$i(m) = \frac{2^{m+2}-(-1)^{m}-3}{6}.$$

The minimum value can be found similarly and is $-i(m)$. It now remains to show that all values $\pm\{0, 1, 2, \ldots, i(m)\}$ can be represented in m-bit binary canonic form. That this is true can be established by considering the addition of 1 to any positive m-bit binary canonic representation such that the result is also in binary canonic form. The first case in which a carry is propagated beyond the m-th bit is when the original representation is $1010\ldots1$. A similar argument proves the case for the negative values.

The digit set resulting from Hwang's radix-r conversion is therefore $\pm\{0, \ldots, \frac{2^{m+2}-(-1)^{m}-3}{6}\}$.

1.2 Average Arithmetic Weight

Consider a group of m adjacent bits prior to conversion and label these bits $(x_{i+m-1}, \ldots, x_{i+1}, x_i)$. Choose i as a multiple of m to ensure these bits are grouped together following conversion to binary canonic form.

After conversion this group will form a zero digit if:

1. $(x_{i+m-1}, \ldots, x_{i+1}, x_i) = (0, \ldots, 0, 0)$ prior to conversion and there is no carry in to x_i during conversion; or
2. $(x_{i+m-1}, \ldots, x_{i+1}, x_i) = (1, \ldots, 1, 1)$ prior to conversion and there is a carry in to x_i during conversion.

The possible bit arrangements that can lead to case 1 are shown in Figure 7.1. This probability of this case can be found from:

$$P(\text{no carry in to } x_i) = \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} + \ldots = \frac{2}{3}$$

$$P((x_{i+m-1}, \ldots, x_{i+1}, x_i) = (0, \ldots, 0, 0)) = \frac{1}{2^m}$$

$$P(\text{case 1}) = P((x_{i+m-1}, \ldots, x_{i+1}, x_i) = (0, \ldots, 0, 0)) \wedge P(\text{no carry in to } x_i) = \frac{1}{2^m} \times \frac{2}{3}$$

	x_{i+m-1}	...	x_{i+1}	x_i	x_{i-1}	x_{i-2}	...							
	0	...	0	0	0	∅	...							
or	0	...	0	0	1	0	0	∅	...					
or	0	...	0	0	1	0	1	0	0	∅	...			
or	0	...	0	0	1	0	1	0	1	0	0	∅	...	
or	0	...	0	0					etc...					

Figure 7.1 Bit Arrangements That Lead to Case 1.

A similar argument gives the probability for case 2:

$$P(\text{case 2}) = \frac{1}{2^m} \times \frac{2}{3}.$$

Therefore the probability that a digit in the final representation is:

$$P(\text{zero digit}) = \frac{2^{2-m}}{3}.$$

For n radix r digits and n large the average arithmetic weight is therefore $n\left(1 - \frac{2^{2-m}}{3}\right)$.

2 Recoded m-ary Method

The *Recoded m-ary Method* appears in [Koç90] but some of the results in this publication are incorrect. The conversion begins with application of the *Recoded Binary Method:* a parallel conversion to the digit set $\{-1, 0, 1\}$ similar to modified Booth. Following this conversion, the probability that a randomly selected bit will be zero is $5/8$. The original publication spends some

effort deriving this result which can be trivially obtained by counting the rows in the conversion table: 6 of the 16 possible cases in the conversion table lead to a zero digit.

The next step is not so straightforward. When m-bit digits are formed on regular m-bit boundaries the probability of finding m adjacent zero bits is not $(5/8)^m$ as claimed in [Koç90]. This is because adjacent bits are not selected independently by the overlapped recoding.

2.1 Average Arithmetic Weight

The conversion table for the *Recoded Binary Method* is shown in Table 7.2.

State	x_{i+1}	x_i	x_{i-1}	x_{i-2}	y_i
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	$\bar{1}$
13	1	1	0	1	$\bar{1}$
14	1	1	1	0	0
15	1	1	1	1	0

Table 7.2 Recoded Binary Method Conversion Table. The table shows the output digit y_i obtained from examination of input bits x_{i-2} to x_{i+1}.

Figure 7.2 shows all the states that correspond to $y_i = 0$ and all of the ways these can lead to $y_{i+1} = 0$. For example, states $\{0, 1, 2, 6, 7, 8, 9, 10, 14, 15\}$ correspond to $y_i = 0$. Of these, only states $\{0, 1, 2, 14, 15\}$ can lead to a situation where $y_{i+1} = 0$. The figure also shows the ways in which the situation can be extended so that $y_{i+2} = 0$ as well.

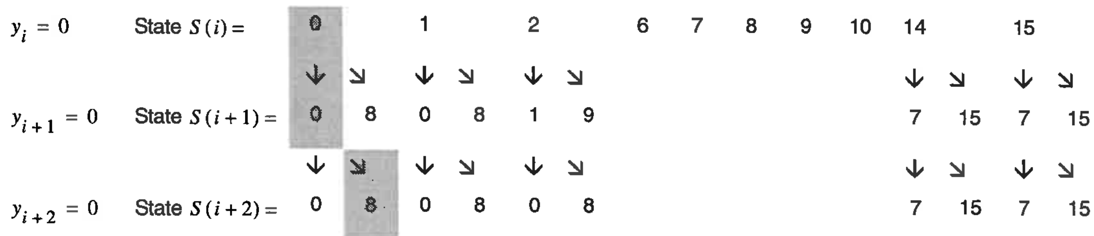

Figure 7.2 State Transitions Leading to Zero Digits.

A sequence of states is shaded in Figure 7.2. If all initial states are equally likely then the probability of $S(i) = 0$ is $1/16$. The probability of a transition to $S(i+1) = 0$ is $1/2$, and the probability of a subsequent transition to $S(i+2) = 8$ is $1/2$. Therefore the probability of the shaded sequence is $1/64$.

All possible sequences of consecutive zeros can be considered in this way. This gives the probability of m consecutive zeros as $5/(2^{m+2})$. From this the average arithmetic weight can been seen to be $n\left(1-\frac{5}{2^{m+2}}\right)$ where n is the number of digits following conversion and n is large.

2.2 Digit Set

Following conversion with the *Recoded Binary Method* not all m-bit strings of signed bits can actually occur.

Consider a group of bits from the original representation $(x_{i+m-1}, \ldots, x_{i+1}, x_i)$ and label the value of this group x'. Following recoded binary conversion these bits will be converted to the signed bits $(y_{i+m-1}, \ldots, y_{i+1}, y_i)$ which will be grouped together to form a digit of the result y'.

The exact value of y' is determined by the extended group $(x_{i+m}, \ldots, x_{i+1}, x_i, x_{i-1}, x_{i-2})$. One way to find the possible values of y' for arbitrary m is to enumerate the conversions for all values of $(x_{i+m}, x_{i+m-1}, x_{i+m-2}, x_i, x_{i-1}, x_i)$, leaving the rest of the bits of x' unspecified. This produces a table of 64 entries from which the following relationships between y' and x' can be determined:

1. For $0 \le x' \le 2^{m-1} + 2^{m-2} - 1$ it is possible for conversion to yield $y' = x'$.

2. For $0 \le x' \le 2^{m-1} + 2^{m-2} - 1$ it is possible for conversion to yield $y' = x' + 1$.

3. For $2^{m-1} \le x' \le 2^m - 1$ it is possible for conversion to yield $y' = x' - 2^m$.

4. For $2^{m-1} \le x' \le 2^m - 1$ it is possible for conversion to yield $y' = x' - 2^m + 1$.

From this we have the resulting digit set $y' \in \{-2^{m-1}, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{m-2} + 2^{m-1}\}$.

Appendix C

Software for an Unmodified ARM

CRITICAL ARITHMETIC functions for the generation of RSA signatures were hand-coded in ARM assembly language. They were simulated with the 'ARMulator' simulation tool to verify that the arithmetic was correct and to measure execution time. The software tool Maple was used to generate 1024-bit test inputs and correct results.

The functions implemented were: *Precom* to pre-compute odd partial products and partial squares, *Acc* to shift and accumulate multiple-precision numbers, *MR* for Montgomery reduction, *Product* for multiplication and *Square* for optimised squaring. Software listings appear below. For more information on the ARM instruction set refer to [ARM94] and [ARM95].

3 Accumulation

```
; ACC.S
;
; M U L T I P L E - P R E C I S I O N   A C C U M U L A T I O N
;
; ON ENTRY:
; a1 is a pointer to an array B (call it pB)
; a2 is an integer less than or equal to 32 (call it s)
; a3 is a pointer to an array C (call it pC)
; a4 is an integer (call it o)
; v1 is the length of B in words (call it lB)
; v2 is a pointer to the word after the end of the array C (call it eC)
;
; ON EXIT:
```

```
; The array B and the word o have been accumulated into the array C thus:
; C := C + o + B<<s.
;
; MORE DETAILS:
; Call Acc0 to set parameter o to zero.
; C must be at least as long as B.
; Any carry beyond the end of B will be propagated to the end of C.
; Any carry beyond the end of C will be returned in the carry flag.
;
; CALLING CONVENTIONS:
; All registers are used and few are stacked.
; This has been done to simplify calls from assembly routines.
; Calls directly from C are not possible.
; Registers s and eC are unchanged.
; The remaining registers are corrupted.
;
        AREA |ACC$$code|, CODE, READONLY
|x$codeseg| DATA
        EXPORT  Acc
        EXPORT  Acc0
        EXPORT  DeAcc
        EXPORT  DeAcc0

;
; Register Allocation                       Previous Use
pB      RN 0                                ;a1
s       RN 1                                ;a2
pC      RN 2                                ;a3
o       RN 3                                ;a4
eB      RN 4                                ;v1 (lB)
b1      RN 5                                ;v2 (eC)
b2      RN 6                                ;v3
b3      RN 7                                ;v4
b4      RN 8                                ;v5
t1      RN 9                                ;r9
t2      RN 10                               ;r10
t3      RN 11                               ;fp
t4      RN 12                               ;ip
;sp     RN 13                               ;sp
rs      RN 14                               ;lr
;pc     RN 15                               ;pc

Acc0
        MOV     o, #0                       ; initialise o to zero

Acc
        STMDB   sp!, {v2,lr}                ; push lr, eC
        RSB     rs, s, #32                  ; rs := 32 - s

;       Start accumulating 1 word at a time
        MSR     CPSR_flg, #0                ; clear the carry flag
        ANDS    b3, eB, #3                  ; b3 := lB mod 4
        BEQ     Acc_4                       ; lB is a multiple of 4
        ADD     b3, pB, b3, LSL #2          ; b3 := pB + 4(lB mod 4)
```

```
Acc_1loop
        LDMIA   pB!, {b2}               ; load 1 word of B, update pB
        LDMIA   pC, {t2}                ; load 1 word of C
        ADCS    t2, t2, o
        MOV     o, b2, LSR rs
        ADC     o, o, #0
        ADDS    t2, t2, b2, LSL s
        STMIA   pC!, {t2}
        TEQ     pB, b3
        BNE     Acc_1loop

;       Accumulate 4 words at a time
Acc_4
        ANDS    eB, eB, #0xfffffffc     ; eB := lB - lB mod 4
        BEQ     Acc_carry
        ADD     eB, pB, eB, LSL #2      ; eB := pB + 4*eB
Acc_4loop
        LDMIA   pB!, {b1, b2, b3, b4}   ; load 4 words of B, update pB
        LDMIA   pC, {t1, t2, t3, t4}    ; load 4 words of C
        ADCS    t1, t1, o
        ADCS    t2, t2, b1, LSR rs
        ADCS    t3, t3, b2, LSR rs
        ADCS    t4, t4, b3, LSR rs
        MOV     o, b4, LSR rs
        ADC     o, o, #0
        ADDS    t1, t1, b1, LSL s
        ADCS    t2, t2, b2, LSL s
        ADCS    t3, t3, b3, LSL s
        ADCS    t4, t4, b4, LSL s
        STMIA   pC!, {t1, t2, t3, t4}   ; store 4 words of C, update pC
        TEQ     pB, eB                  ; use TEQ to preserve the carry flag
        BNE     Acc_4loop

;       Accumulate the remaining bits in o and the c-flag
Acc_carry
        LDMIA   sp!, {v2, lr}           ; v2 := eC
Acc_cloop
        TEQ     pC, v2                  ; use TEQ to preserve the c-flag
        BEQ     Acc_exit
        LDMIA   pC, {t1}
        ADCS    t1, t1, o
        STMIA   pC!, {t1}
        MOV     o, #0
        BCS     Acc_cloop

Acc_exit
        MOV     pc, lr                  ; return

;
; M U L T I P R E C I S I O N   D E - A C C U M U L A T I O N
;
; ON ENTRY:
; a1 is a pointer to an array B (call it pB)
; a2 is an integer less than or equal to 32 (call it s)
```

```
; a3 is a pointer to an array C (call it pC)
; a4 is an integer (call it o)
; v1 is the length of B in words (call it lB)
; v2 is a pointer to the word after the end of the array C (call it eC)
;
; ON EXIT:
; The array B and the word o have been accumulated into the array C thus:
; C := C - o - B<<s.
;
; MORE DETAILS:
; Call DeAcc0 to set parameter o to zero.
; C must be at least as long as B.
; Any borrow beyond the end of B will be propagated to the end of C.
; Any borrow beyond the end of C will be returned in the carry flag.
;
; CALLING CONVENTIONS:
; All registers are used and few are stacked.
; This has been done to simplify calls from assembly routines.
; Calls directly from C are not possible.
; Registers s and eC are unchanged.
; The remaining registers are corrupted.
;

DeAcc0
        MOV     o, #0                   ; initialise o to zero

DeAcc
        STMDB   sp!, {v2,lr}            ; push lr, eC
        RSB     rs, s, #32              ; rs := 32 - s

;       Start de-accumulating 1 word at a time
        MSR     CPSR_flg, #0xf0000000   ; set the carry flag
        ANDS    b3, eB, #3              ; b3 := lB mod 4
        BEQ     DeAcc_4                 ; lB is a multiple of 4
        ADD     b3, pB, b3, LSL #2      ; b3 := pB + 4(lB mod 4)
DeAcc_1loop
        LDMIA   pB!, {b2}               ; load 1 word of B, update pB
        LDMIA   pC, {t2}                ; load 1 word of C
        SBCS    t2, t2, o
        MOV     o, b2, LSR rs
        SBC     o, o, #0
        SUBS    t2, t2, b2, LSL s
        STMIA   pC!, {t2}
        TEQ     pB, b3
        BNE     DeAcc_1loop

;       De-accumulate 4 words at a time
DeAcc_4
        ANDS    eB, eB, #0xfffffffc     ; eB := lB - lB mod 4
        BEQ     DeAcc_borrow
        ADD     eB, pB, eB, LSL #2      ; eB := pB + 4*eB
DeAcc_4loop
        LDMIA   pB!, {b1, b2, b3, b4}   ; load 4 words of B, update pB
        LDMIA   pC, {t1, t2, t3, t4}    ; load 4 words of C
```

```
        SBCS    t1, t1, o
        SBCS    t2, t2, b1, LSR rs
        SBCS    t3, t3, b2, LSR rs
        SBCS    t4, t4, b3, LSR rs
        MOV     o, b4, LSR rs
        SBC     o, o, #0
        SUBS    t1, t1, b1, LSL s
        SBCS    t2, t2, b2, LSL s
        SBCS    t3, t3, b3, LSL s
        SBCS    t4, t4, b4, LSL s
        STMIA   pC!, {t1, t2, t3, t4}       ; store 4 words of C, update pC
        TEQ     pB, eB                      ; use TEQ to preserve the borrow flag
        BNE     DeAcc_4loop

;       Deaccumulate the remaining bits in o and the c-flag
DeAcc_borrow
        LDMIA   sp!, {v2, lr}               ; v2 := eC
DeAcc_bloop
        TEQ     pC, v2                      ; use TEQ to preserve the c-flag
        BEQ     DeAcc_exit
        LDMIA   pC, {t1}
        SBCS    t1, t1, o
        STMIA   pC!, {t1}
        MOV     o, #0
        BCC     DeAcc_bloop

DeAcc_exit
        MOV     pc, lr                      ; return
END
```

4 Montgomery Reduction

```
; MR.S
;
; M O N T G O M E R Y   R E D U C T I O N
;
; Version 2.0
; 12/7/00
;
; Unsigned Sliding Window Version
; Modulus Multiples Precomputed
; Address Table Quotient Digit Selection
;
; ON ENTRY:
; a1 is a pointer to a table of modulus multiple pointers (ppB)
; a2 is a pointer to a (2l)-word array to be reduced (call it A)
; a3 is a pointer to a (2l)-word destination buffer (call it C)
;
; ON EXIT:
; C may be l words and 1 bit long with the upper bit returned in a1.
; The lower l words of C should be zero.
; The upper part of C should be congruent with C/(2^l) mod B.
```

```
;
; MORE DETAILS:
;
; The quotient digits (q) are selected to be m-bits long.
; They are radix r = 2^m.
;
; The modulus multiple address table is at address ppB:
; Address          Contents
; a2               p(b')B
; a2+4             p(3b')B
; ..a2+4*(r/2-1)   p((r-1)b')B
;
; where b' = (-1)/B mod r.
;
; The modulus mutiples themselves are (l+1)-word arrays at
; addresses p(b')B, p(3b')B...
;
; The modulus length l must be divisible by four.
;
; CALLING CONVENTIONS:
; I stack all important APCS registers. Calls from C are possible.
;

        AREA |MR$$codel, CODE, READONLY

lx$codesegl DATA

        EXPORT  MR
        IMPORT  Acc0

        m       EQU 7                   ; quotient digit length in bits
        r       EQU 1:SHL:m             ; quotient digit radix
        l       EQU 16                  ; modulus word-length

; Register Allocation                   Previous Use
ppB     RN  0                           ;a1
s       RN  1                           ;a2 (pA)
iC      RN  2                           ;a3 (pC)
;a4     RN  3                           ;a4
lB      RN  4                           ;v1
eC      RN  5                           ;v2
eCl     RN  6                           ;v3
q       RN  7                           ;v4
C       RN  8                           ;v5
t       RN  9                           ;r9
cC      RN  10                          ;r10
;fp     RN  11                          ;fp
;ip     RN  12                          ;ip
;sp     RN  13                          ;sp
;lr     RN  14                          ;lr
;pc     RN  15                          ;pc

MR
        STMDB   sp!, {v1-v5,r9-fp,lr}           ; store all important c registers
```

```
;       If pA <> pC then copy A to C four words at a time.
        TEQ     a2, a3
        BEQ     MR_init
        ADD     v2, a3, #(4*2*l)        ; v2 points to the end of C
        ADD     a2, a2, #(4*2*l)        ; a2 points to the end of A
MR_copy
        LDMDA   a2!, {v5-fp}            ; post decrement
        STMDA   v2!, {v5-fp}
        CMP     v2, a3
        BGE     MR_copy

;       Initialise the loops
MR_init
        ADD     eCl, iC, #(4*(l-1))     ; point to the last word to reduce
        ADD     eC, iC, #(4*2*l)        ; parameter eC for Acc
        SUB     C, iC, #4               ; point before the first word to reduce
        MOV     s, #0
        MOV     cC, #0

;       An outer loop over the words of C
MR_cloop
        LDR     C, [iC, #4]!            ; load C and pre-increment iC

;       An inner loop over the bits in a word
MR_sloop
        MOVS    C, C, LSR #1            ; shift the next bit into c-flag
        BCS     MR_digit                ; found the first bit of a digit
        ADD     s, s, #1                ; skip zeros
        ANDS    s, s, #31
        BNE     MR_sloop
        TEQ     iC, eCl
        BNE     MR_cloop
        B       MR_exit

;       Found a non-zero digit
;       NOTE: extract the m-1 upper bits of q as the LSB is always 1
MR_digit
        MOV     lB, #(l+1)              ; parameter lB for Acc
        AND     q, C, #(r/2-1)          ; the top m-1 bits of q
        CMP     s, #(32-m)
        BGE     MR_overlap
        STMDB   sp!, {ppB,iC,eCl,cC}    ; save registers
        LDR     a1, [ppB, q, LSL #2]    ; load the address of the modulus multiple
        BL      Acc0
        LDMIA   sp!, {ppB,iC,eCl,cC}    ; restore registers
        ADC     cC, cC, #0              ; keep any carry out
        ADD     s, s, #m
        LDR     C, [iC]                 ; pre-increment & write back
        MOV     C, C, LSR s
        B       MR_sloop

;       The quotient digit straddles two words of C or abuts the end of a word
MR_overlap
```

```
        TEQ     iC, eCl
        BEQ     MR_lastdigit
        LDR     C, [iC, #4]             ; pre-increment
        RSB     t, s, #31
        ORR     q, q, C, LSL t          ; form q from the overlap
        AND     q, q, #(r/2-1)
        STMDB   sp!, {ppB,iC,eCl,cC}    ; save registers
        LDR     a1, [ppB, q, LSL #2]    ; load the address of the modulus multiple
        BL      Acc0
        LDMIA   sp!, {ppB,iC,eCl,cC}    ; restore registers
        ADC     cC, cC, #0              ; keep any carry out
        SUB     s, s, #(32-m)
        LDR     C, [iC, #4]!            ; pre-increment & write back
        MOV     C, C, LSR s
        B       MR_sloop

;       The last digit overlaps beyond eCl
MR_lastdigit
        STMDB   sp!, {cC}               ; save registers
        LDR     a1, [ppB, q, LSL #2]    ; load the address of the modulus multiple
        BL      Acc0
        LDMIA   sp!, {cC}               ; restore registers
        ADC     cC, cC, #0              ; keep any carry out

;       All modulus multiples have been accumulated.
MR_exit
        MOV     a1, cC                  ; return any carry out bit
        LDMIA   sp!, {v1-v5,r9-fp,pc}   ; restore all important c registers

END
```

5 Pre-computation

```
; PRECOM.S
;
;ODD MULTIPLE PRECOMPUTATION
;
; ON ENTRY:
; a1    is a pointer to a table of 2^(m-1) arrays of (l+1)-words (call it pB)
;       the first array is the l-word number B
;
; ON EXIT:
; The table of partial products (B, 3B, ... (2^m - 1)B) at pB is complete.
; All registers are restored to their original values.
;

        AREA |PRECOM$$code|, CODE, READONLY

|x$codeseg| DATA

        EXPORT  Precom
```

```
m       EQU 7                          ; digit length in bits
r       EQU 1:SHL:m                    ; digit radix
l       EQU 16                         ; operand word-length

; Register Allocation                  Previous Use
pB      RN  0                          ;a1
pA      RN  1                          ;a2
eB      RN  2                          ;a3
o       RN  3                          ;a4
pC      RN  4                          ;v1
b1      RN  5                          ;v2
b2      RN  6                          ;v3
b3      RN  7                          ;v4
b4      RN  8                          ;v5
t1      RN  9                          ;r9
t2      RN  10                         ;r10
t3      RN  11                         ;fp
t4      RN  12                         ;ip
;sp     RN  13                         ;sp
j       RN  14                         ;lr
;pc     RN  15                         ;pc

Precom
        STMDB   sp!, {a1-ip,lr}        ; store registers

;       Start by copying 2B into the array position for (2^m-1)B
        MOV     pC, pB                 ; pC points to B
        LDR     pA, = (r/2-1)*(l+1)*4
        ADD     pA, pB, pA             ; pA points to 2B
        ADD     eB, pB, #(4*l)
        MOV     o, #0
Precom_cloop
        LDMIA   pC!, {b1, b2, b3, b4}
        MOV     t1, b1, LSL #1
        MOV     t2, b2, LSL #1
        MOV     t3, b3, LSL #1
        MOV     t4, b4, LSL #1
        ORR     t1, t1, o
        ORR     t2, t2, b1, LSR #31
        ORR     t3, t3, b2, LSR #31
        ORR     t4, t4, b3, LSR #31
        MOV     o, b4, LSR #31
        STMIA   pA!, {t1, t2, t3, t4}
        TEQ     pC, eB
        BNE     Precom_cloop
        STMIA   pA!, {o}

;       An outer loop over the multiples 3B, 5B, ... (2^m -1)B
        MOV     j, #(r/2-1)            ; pB points to xB where x = 1,3,5...
        ADD     pC, pB, #(4*(l+1))     ; pC points to the destination

Precom_jloop
        SUB     pA, pA, #(4*(l+1))     ; pA points to 2B
```

```
;       An inner loop over l words
        ADDS    eB, pB, #(4*l)          ; point to the (l+1)-th word (clear the c-flag)
Precom_iloop
        LDMIA   pA!, {t1, t2, t3, t4}   ; load 4 words of 2B
        LDMIA   pB!, {b1, b2, b3, b4}   ; load 4 words of xB
        ADCS    t1, t1, b1
        ADCS    t2, t2, b2
        ADCS    t3, t3, b3
        ADCS    t4, t4, b4
        STMIA   pC!, {t1, t2, t3, t4}   ; store 4 words of C
        TEQ     pB, eB                  ; use TEQ to preserve the carry flag
        BNE     Precom_iloop

;       A single iteration for the (l+1)-th word
        LDMIA   pA!, {t1}
        LDMIA   pB!, {b1}
        ADC     t1, t1, b1
        STMIA   pC!, {t1}

        SUBS    j, j, #1
        BNE     Precom_jloop

Precom_exit
        LDMIA   sp!, {a1-ip,pc}         ; restore registers, and return

END
```

6 Multiplication

```
; PRODUCT.S
;
; M U L T I P L I C A T I O N
;
; Version 2.0
; 12/7/00
;
; Unsigned Sliding Window Version
; Precomputes Partial Products
;
; ON ENTRY:
; a1    is a pointer to a table of 2^(m-1) arrays of (l+1)-words (call it pB)
;       the first array is the multiplicand B
; a2    is a pointer to an l-word array: the multiplier A (call it pA)
; a3    is a pointer to a (2l)-word destination buffer C (call it pC)
;
; ON EXIT:
; The table of partial products (B, 3B, ... (2^m - 1)B) at pB is complete.
; C is the product of A and B. C = A * B.
;
; MORE DETAILS:
;
; The multiplier digits (a) are selected to be m-bits long.
```

```
; They are radix r = 2^m.
;
; CALLING CONVENTIONS:
; I stack all important APCS registers. Calls from C are possible.
;

        AREA |PRODUCT$$code|, CODE, READONLY

|x$codeseg| DATA

        EXPORT Product
        IMPORT Acc0
        IMPORT Precom

m       EQU 7                          ; multiplier digit length in bits
r       EQU 1:SHL:m                    ; multiplier digit radix
lgl     EQU 4                          ; binary log of multiplier word length
l       EQU 1:SHL:lgl                  ; multiplier word-length

; Register Allocation                  Previous Use
pB      RN  0                          ;a1
s       RN  1                          ;a2 (pA)
pC      RN  2                          ;a3 (pC)
pA      RN  3                          ;a4
lB      RN  4                          ;v1
eC      RN  5                          ;v2
a       RN  6                          ;v3
A       RN  7                          ;v4
eA      RN  8                          ;v5
t       RN  9                          ;r9
;r10    RN  10                         ;r10
;fp     RN  11                         ;fp
;ip     RN  12                         ;ip
;sp     RN  13                         ;sp
;lr     RN  14                         ;lr
;pc     RN  15                         ;pc

Product
        STMDB   sp!, {v1-v5,r9-fp,lr}  ; store all important c registers

;       Precompute 3B, 5B, ... (2^m -1)B
        MOV     pA, a2
        BL      Precom

;       Initialise
        ADD     eA, pA, #(4*(l-1))     ; point to the last multiplier word
        SUB     pA, pA, #4             ; point before the first multiplier word
        ADD     eC, pC, #(4*2*l)       ; point after the MSW in C
        SUB     pC, pC, #4             ; point before the first destination word
        MOV     s, #0

;       An outer loop over the multiplier words
Product_aloop
        LDR     A, [pA, #4]!           ; load A and pre-increment pA
```

```
        ADD     pC, pC, #4              ; accumulate at higher significance

;       An inner loop over the bits in a word
Product_sloop
        MOVS    A, A, LSR #1            ; shift the next bit into c-flag
        BCS     Product_digit           ; found the first bit of a digit
        ADD     s, s, #1                ; skip zeros
        ANDS    s, s, #31
        BNE     Product_sloop
        TEQ     pA, eA
        BNE     Product_aloop
        B       Product_exit

;       Found a non-zero digit
;       NOTE: extract the m-1 upper bits of a as the LSB is always 1
Product_digit
        MOV     lB, #(l+1)              ; parameter lB for Acc
        AND     a, A, #(r/2-1)          ; the top m-1 bits of a
        CMP     s, #(32-m)
        BGE     Product_overlap
        STMDB   sp!, {pB,pC,pA,A,eA}    ; save registers
        ADD     a, a, a, LSL #lgl
        ADD     pB, pB, a, LSL #2       ; calculate the address of (2a+1)B
        BL      Acc0
        LDMIA   sp!, {pB,pC,pA,A,eA}    ; restore registers
        ADD     s, s, #m
        MOV     A, A, LSR #(m-1)
        B       Product_sloop

;       The multipler digit straddles two words of A or abuts the end of a word
Product_overlap
        TEQ     pA, eA
        BEQ     Product_lastdigit
        LDR     A, [pA, #4]             ; pre-increment
        RSB     t, s, #31
        ORR     a, a, A, LSL t          ; form a from the overlap
        AND     a, a, #(r/2-1)
        STMDB   sp!, {pB,pC,pA,eA}      ; save registers
        ADD     a, a, a, LSL #lgl
        ADD     pB, pB, a, LSL #2       ; calculate the address of (2a+1)B
        BL      Acc0
        LDMIA   sp!, {pB,pC,pA,eA}      ; restore registers
        SUB     s, s, #(32-m)
        LDR     A, [pA, #4]!            ; pre-increment & write back
        ADD     pC, pC, #4              ; accumulate at higher significance
        MOV     A, A, LSR s
        B       Product_sloop

;       The last digit overlaps beyond eA
Product_lastdigit
        ADD     a, a, a, LSL #lgl
        ADD     pB, pB, a, LSL #2       ; calculate the address of (2a+1)B
        BL      Acc0
```

```
;       All partial products have been accumulated.
Product_exit
        LDMIA   sp!, {v1-v5,r9-fp,pc}           ; restore all important c registers

    END
```

7 Optimised Squaring

```
; SQUARE.S
;
;O P T I M I S E D   S Q U A R E
;
; Version 2.0
; 12/7/00
;
; Unsigned Sliding Window Version
; Precomputes Partial Products
;
; ON ENTRY:
; a1    is a pointer to a table of 2^(m-1) arrays of (l+1)-words (call it pA)
;       the first array is the number to square A
; a2    is pointer to a 2^m word array
;       the first 2^(m-1) words are the minimum correction factors tmin[a]
;       the second 2^(m-1) words are the integer squares 1,9,25,49 etc...
; a3    is a pointer to a (4l+2)-word destination buffer C (call it pC)
;       the result is returned in the lower 2l+1 words
;       the upper 2l+1 words are used as temporary working space
;       the buffer should be cleared to zero initially
;
; ON EXIT:
; The table of partial products (A, 3A, ... (2^m - 1)A) at pA is complete.
; C is the square of A. C = A * A.
;
; MORE DETAILS:
;
; The multiplier digits (a) are selected to be m-bits long.
; They are radix r = 2^m.
; The min correction terms: for a = 1,3,...,(2^m-1)
; tmin[a] = (a + (a-1)(a mod 2^(m-1)))>> m + ((a-1)/2)*((a mod m)>>(m-1))
;
; CALLING CONVENTIONS:
; I stack all important APCS registers. Calls from C are possible.
;

        AREA |SQUARE$$code|, CODE, READONLY

|x$codeseg| DATA

        EXPORT Square
        IMPORT Acc
        IMPORT Precom
        IMPORT DeAcc0
```

```
m       EQU 7                               ; multiplier digit length in bits
r       EQU 1:SHL:m                         ; multiplier digit radix
lgl     EQU 4                               ; binary log of multiplier word length
l       EQU 1:SHL:lgl                       ; multiplier word-length

; Register Allocation                       Previous Use
pA      RN 0                                ;a1
s       RN 1                                ;a2 (pA2)
pC      RN 2                                ;a3 (pC)
o       RN 3                                ;a4
lA      RN 4                                ;v1
eC      RN 5                                ;v2
t       RN 6                                ;v3
eA      RN 7                                ;v4
a       RN 8                                ;v5
pA2     RN 9                                ;r9
A1      RN 10                               ;r10
A2      RN 11                               ;fp
t1      RN 12                               ;ip
;sp     RN 13                               ;sp
t2      RN 14                               ;lr
;pc     RN 15                               ;pc

Square
        STMDB   sp!, {v1-v5,r9-fp,lr}       ; store all important c registers

;       Precompute 3A, 5A, ... (2^m -1)A
        MOV     pA2, a2
        BL      Precom

;       Initialise
        STMDB   sp!, {pC}                   ; store pC for the final correction
        ADD     eA, pA, #(4*(l-1))          ; point to the last multiplier word
        SUB     pA, pA, #4                  ; point before the first multiplier word
        ADD     eC, pC, #(4*2*l)            ; point after the MSW in C
        SUB     pC, pC, #(4*2)              ; point before the destination word
        MOV     s, #0
        MOV     lA, #((l+1)-2+1)            ; paramter lB for Acc
                                            ; (-2 as A1 & A2 are acced outside Acc)
                                            ; (+1 as we decrement in the 1st iteration)

;       An outer loop over the multiplier words
Square_aloop
        LDR     A1, [pA, #4]!               ; pre-increment pA
        ADD     pC, pC, #(4*2)              ; accumulate at higher significance
        SUB     lA, lA, #1                  ; partial squares are shorter

;       An inner loop over the bits in a word
Square_sloop
        MOVS    A1, A1, LSR #1              ; shift the next bit into c-flag
        BCS     Square_digit                ; found the first bit of a digit
        ADD     s, s, #1                    ; skip zeros
        ANDS    s, s, #31
```

```
        BNE     Square_sloop
        TEQ     pA, eA
        BNE     Square_aloop
        B       Square_exit

;       Found a non-zero digit
;       NOTE: extract the m-1 upper bits of a as the LSB is always 1
Square_digit
        MOV     o, #0
        AND     a, A1, #(r/2-1)             ; the top m-1 bits of a
        CMP     s, #(32-m)
        BGE     Square_overlap

        STMDB   sp!, {pA,pC,lA,eA,pA2}

        MOV     v4, A1, ROR #m              ; v4 is used to correct tmin
        ADD     t, a, a, LSL #lgl           ; calculate the address of (2a+1)A
        ADD     pA, pA, t, LSL #2
        LDMIA   pA!, {A1, A2}
        ADD     t, s, #m                    ; set the lower s+m bits to zero
        MOV     A1, A1, LSR t
        EOR     v4, v4, A1, ROR #1
        MOV     A1, A1, LSL t

        LDR     t, [pA2, a, LSL #2]         ; look up the min correction tmin
        EORS    v4, v4, t, ROR #1
        ADDMI   t, t, #1                    ; correct the correction term
        ADD     pA2, pA2, #(4*(r/2))        ; look up the precomputed square
        LDR     a, [pA2, a, LSL #2]         ; a := (2a+1)^2

        ADD     r9, pC, #((2*l+1)*4)
        LDMIA   r9, {t1, t2}
        MOV     v4, s, LSL #1               ; v4 := 2s+m+1
        ADD     v4, v4, #(m+1)
        ORR     t1, t1, t, LSL v4           ; [t2,t1] |= t << (2s+m+1)
        SUBS    v4, v4, #32
        ORRPL   t2, t2, t, LSL v4
        RSBS    v4, v4, #0
        ORRNE   t2, t2, t, LSR v4
        STMIA   r9, {t1, t2}

        LDMIA   pC, {t1, t2}
        MOV     t, s, LSL #1                ; t := 2s
        ADDS    t1, t1, a, LSL t            ; [t2,t1] += (2a+1)^2 << (2s)
        SUB     t, t, #32
        ADCS    t2, t2, a, LSL t
        ADC     o, o, #0
        RSBS    t, t, #0
        BEQ     Square_digit_skip0          ; to deal with the case 2s = 32
        ADD     t2, t2, a, LSR t
        ADC     o, o, #0
Square_digit_skip0

        ADD     s, s, #1                    ; [t2,t1] += [A2,A1] << (s+1)
```

```
        RSB     t, s, #32
        ADDS    t2, t2, A1, LSR t
        ADC     o, o, A2, LSR t
        ADDS    t1, t1, A1, LSL s
        ADCS    t2, t2, A2, LSL s
        ADC     o, o, #0
        STMIA   pC!, {t1, t2}

        CMP     lA, #0
        BLGT    Acc
        LDMIA   sp!, {pA,pC,lA,eA,pA2}      ; restore registers
        ADD     s, s, #(m-1)
        LDR     A1, [pA]
        MOV     A1, A1, LSR s
        B       Square_sloop

;       The digit straddles two words of A or abuts the end of a word
Square_overlap
        ADD     pA, pA, #4
        ADD     pC, pC, #4                  ; accumulate at higher significance
        SUB     A, lA, #1                   ; partial squares are shorter
        CMP     pA, eA
        BGT     Square_lastdigit

        STMDB   sp!, {pA,pC,lA,eA,pA2}      ; save registers

        LDR     A1, [pA]                    ; pre-increment
        RSB     t, s, #31
        ORR     a, a, A1, LSL t             ; form a from the overlap
        AND     a, a, #(r/2-1)
        RSB     t, t, #m
        MOV     v4, A1, ROR t
        ADD     t, a, a, LSL #lgl           ; calculate the address of (2a+1)A
        ADD     pA, pA, t, LSL #2
        LDMIA   pA!, {A1, A2}
        SUB     t, s, #(32-m)               ; set the lower s+m-32 bits to zero
        MOV     A1, A1, LSR t
        EOR     v4, v4, A1, ROR #1
        MOV     A1, A1, LSL t

        LDR     t, [pA2, a, LSL #2]         ; look up the min correction tmin
        EORS    v4, v4, t, ROR #1
        ADDMI   t, t, #1                    ; correct the correction term
        ADD     pA2, pA2, #(4*(r/2))        ; look up the precomputed square
        LDR     a, [pA2, a, LSL #2]         ; a := (2a+1)^2

        ADD     r9, pC, #((2*l+1)*4)
        LDMIA   r9, {t1, t2}
        MOV     v4, s, LSL #1               ; v4 := 2s+m+1-32
        SUB     v4, v4, #(32-m-1)
        ORR     t1, t1, t, LSL v4           ; [t2,t1] |= t << (2s+m+1-32)
        SUBS    v4, v4, #32
        ORRPL   t2, t2, t, LSL v4
        RSBS    v4, v4, #0
```

```
        ORRNE  t2, t2, t, LSR v4
        STMIA  r9, {t1, t2}

        LDMIA  pC, {t1, t2}
        MOV    t, s, LSL #1             ; [t2,t1] += (2a+1)^2 << (2*s-32)
        SUB    t, t, #32
        ADDS   t1, t1, a, LSL t
        SUB    t, t, #32
        ADCS   t2, t2, a, LSL t
        ADC    o, o, #0
        RSBS   t, t, #0
        ADD    t2, t2, a, LSR t
        ADC    o, o, #0

        ADD    s, s, #1                 ; [t2,t1] += [A2,A1] << (s+1)
        RSB    t, s, #32
        ADDS   t2, t2, A1, LSR t
        ADC    o, o, A2, LSR t
        ADDS   t1, t1, A1, LSL s
        ADCS   t2, t2, A2, LSL s
        ADC    o, o, #0
        STMIA  pC!, {t1, t2}

        CMP    lA, #0
        BLGT   Acc
        LDMIA  sp!, {pA,pC,lA,eA,pA2}   ; restore registers

        SUB    s, s, #(32-(m-1))        ; s := s + (m-1) - 32
        LDR    A1, [pA]
        MOV    A1, A1, LSR s
        ADD    pC, pC, #4               ; accumulate at higher significance
        B      Square_sloop

;       The last digit overlaps beyond eA
Square_lastdigit
        ADD    t, a, a, LSL #lgl        ; calculate the address of (2a+1)A
        ADD    pA, pA, t, LSL #2
        LDMIA  pA!, {A1}
        SUB    t, s, #(32-m)            ; set the lower s+m-32 bits to zero
        MOV    A1, A1, LSR t
        MOV    v4, A1, ROR #1
        MOV    A1, A1, LSL t

        LDR    t, [pA2, a, LSL #2]      ; look up the min correction tmin
        EORS   v4, v4, t, ROR #1
        ADDMI  t, t, #1
        ADD    pA2, pA2, #(4*(r/2))     ; look up the precomputed square
        LDR    a, [pA2, a, LSL #2]      ; a := (2a+1)^2

        ADD    r9, pC, #((2*l+1)*4)
        LDMIA  r9, {t1, t2}
        MOV    v4, s, LSL #1            ; v4 := 2s+m+1-32
        SUB    v4, v4, #(32-m-1)
        ORR    t1, t1, t, LSL v4        ; [t2,t1] |= t << (2s+m+1-32)
```

```
        SUB     v4, v4, #32
        ORR     t2, t2, t, LSL v4
        RSBS    v4, v4, #0
        ORRNE   t2, t2, t, LSR v4
        STMIA   r9, {t1, t2}

        LDMIA   pC, {t1, t2}
        MOV     t, s, LSL #1            ; [t2,t1] += (2a+1)^2 << (2*s-32)
        SUB     t, t, #32
        ADDS    t1, t1, a, LSL t
        RSB     t, t, #32
        ADCS    t2, t2, a, LSR t

        ADD     s, s, #1                ; [t2,t1] += A1 << (s+1)
        RSB     t, s, #32
        ADD     t1, t1, A1, LSL s
        ADC     t2, t2, A1, LSR t

        STMIA   pC!, {t1, t2}

;       All partial products have been accumulated.
;       Subtract the correction array
Square_exit

        LDMIA   sp!, {pC}
        ADD     a1, pC, #((2*l+1)*4)    ; point to the correction array
        MOV     v1, #(2*l)
        MOV     s, #0
        BL      DeAcc0

        LDMIA   sp!, {v1-v5,r9-fp,pc}   ; restore all important c registers

    END
```

Appendix D

Software for a Modified ARM

ARCHITECTURAL modifications to the ARM were proposed in Chapter 6 with the goal of improving the average case performance of the sliding window functions presented in Appendix C. The changes included an extended left shift feature for add and subtract instructions, a dedicated loop counting register and delayed branch instruction.

Improved software routines were developed to take advantage of this modified hardware and these were simulated to obtain timing measurements. The hardware changes had the greatest impact on the accumulate function: the remaining functions were largely unchanged. The new accumulate function is shown below.

```
; ACC.S
;
; M U L T I P R E C I S I O N   A C C U M U L A T I O N
;
; Version 2.1ESL
; 7/12/00
;
; Modified to include the extended shift left (ESL) mechanism
; and the enhanced loop counter.
;
; ON ENTRY:
; a1 is a pointer to an array B (call it pB)
; a2 is an integer less than or equal to 32 (call it s)
; a3 is a pointer to an array C (call it pC)
; a4 is an integer (call it o)
; v1 is the length of B in words (call it lB)
; v2 is a pointer to the word after the end of the array C (call it eC)
;
```

```
; ON EXIT:
; The array B and the word o have been accumulated into the array C thus:
; C := C + o + B<<s.
;
; MORE DETAILS:
; Call Acc0 to set parameter o to zero.
; C must be at least as long as B.
; Any carry beyond the end of B will be propagated to the end of C.
; Any carry beyond the end of C will be returned in the carry flag.
;
; CALLING CONVENTIONS:
; All registers are used and few are stacked.
; This has been done to simplify calls from assembly routines.
; Calls directly from C are not possible.
; Registers s, lB, eC are unchanged.
; The remaining registers are corrupted.
; Note that the loop counter register is used. This routine cannot be called
; from within a loop controlled by the enhanced loop counter.
;

        AREA |ACC$$code|, CODE, READONLY

|x$codeseg| DATA

        EXPORT Acc
        EXPORT Acc0
        EXPORT DeAcc
        EXPORT DeAcc0

; Register Allocation                       Previous Use
pB      RN 0                                ;a1
s       RN 1                                ;a2
pC      RN 2                                ;a3
b1      RN 3                                ;a4 (o)
lB      RN 4                                ;v1
eC      RN 5                                ;v2
b2      RN 6                                ;v3
b3      RN 7                                ;v4
b4      RN 8                                ;v5
t1      RN 9                                ;r9
t2      RN 10                               ;r10
t3      RN 11                               ;fp
t4      RN 12                               ;ip
;sp     RN 13                               ;sp
;lr     RN 14                               ;lr
;pc     RN 15                               ;pc

Acc0
        MOV     a4, #0                      ; initialise o to zero

Acc
        ADD     a4, a4, a4, ESL #32         ; initialise RE to o

;       Start accumulating 1 word at a time
```

```
        MSR     CPSR_flg, #0            ; clear the carry flag
        ANDS    b1, lB, #3              ; b3 := lB mod 4
        BEQ     Acc_4                   ; lB is a multiple of 4
        MTESR   s
        MTCTR   b1
Acc_1loop
        LDMIA   pB!, {b1}               ; load 1 word of B, update pB
        LDMIA   pC, {t1}                ; load 1 word of C
        BCTRGT  Acc_1loop
        ADCS    t1, t1, b1, ESL ESR
        STMIA   pC!, {t1}

;       Accumulate 4 words at a time
Acc_4
        MOVS    b1, lB, LSR #2          ; b1 := lB div 4
        BEQ     Acc_carry
        MTCTR   b1
Acc_4loop
        LDMIA   pB!, {b1, b2, b3, b4}   ; load 4 words of B, update pB
        LDMIA   pC, {t1, t2, t3, t4}    ; load 4 words of C
        ADCS    t1, t1, b1, ESL ESR
        ADCS    t2, t2, b2, ESL ESR
        ADCS    t3, t3, b3, ESL ESR
        BCTRGT
Acc_4loop
        ADCS    t4, t4, b4, ESL ESR
        STMIA   pC!, {t1, t2, t3, t4}   ; store 4 words of C, update pC

;       Accumulate the remaining bits in RE and the c-flag
Acc_carry
        MOV     b1, #0
        ADD     b1, b1, b1, ESL #32     ; b1 := RE
Acc_cloop
        TEQ     pC, eC                  ; use TEQ to preserve the c-flag
        BEQ     Acc_exit
        LDMIA   pC, {t1}
        ADCS    t1, t1, b1
        STMIA   pC!, {t1}
        MOV     b1, #0
        BCS     Acc_cloop

Acc_exit
        MOV     pc, lr                  ; return

;
; M U L T I P R E C I S I O N   D E - A C C U M U L A T I O N
;
; ON ENTRY:
; a1 is a pointer to an array B (call it pB)
; a2 is an integer less than or equal to 32 (call it s)
; a3 is a pointer to an array C (call it pC)
; a4 is an integer (call it o)
; v1 is the length of B in words (call it lB)
; v2 is a pointer to the word after the end of the array C (call it eC)
```

```
;
; ON EXIT:
; The array B and the word o have been accumulated into the array C thus:
; C := C - o - B<<s.
;
; MORE DETAILS:
; Call DeAcc0 to set parameter o to zero.
; C must be at least as long as B.
; Any borrow beyond the end of B will be propagated to the end of C.
; Any borrow beyond the end of C will be returned in the carry flag.
;
; CALLING CONVENTIONS:
; All registers are used and few are stacked.
; This has been done to simplify calls from assembly routines.
; Calls directly from C are not possible.
; Registers s, lB, eC are unchanged.
; The remaining registers are corrupted.
;

DeAcc0
        MOV     a4, #0                      ; initialise o to zero

DeAcc
        ADD     a4, a4, a4, ESL #32         ; initialise RE to o

;       Start de-accumulating 1 word at a time
        MSR     CPSR_flg, #0xf0000000       ; set the carry flag
        ANDS    b1, lB, #3                  ; b1 := lB mod 4
        BEQ     DeAcc_4                     ; lB is a multiple of 4
        MTESR   s
        MTCTR   b1
DeAcc_1loop
        LDMIA   pB!, {b1}                   ; load 1 word of B, update pB
        LDMIA   pC, {t1}                    ; load 1 word of C
        BCTGT   DeAcc_1loop
        SBCS    t1, t1, b1, ESL ESR
        STMIA   pC!, {t1}

;       De-accumulate 4 words at a time
DeAcc_4
        MOVS    b1, lB, LSR #2              ; b1 := lB div 4
        BEQ     DeAcc_borrow
        MTCTR   b1
DeAcc_4loop
        LDMIA   pB!, {b1, b2, b3, b4}       ; load 4 words of B, update pB
        LDMIA   pC, {t1, t2, t3, t4}        ; load 4 words of C
        SBCS    t1, t1, b1, ESL ESR
        SBCS    t2, t2, b2, ESL ESR
        SBCS    t3, t3, b3, ESL ESR
        BCTRGT DeAcc_4loop
        SBCS    t4, t4, b4, ESL ESR
        STMIA   pC!, {t1, t2, t3, t4}       ; store 4 words of C, update pC

;       Deaccumulate the remaining bits in RE and the c-flag
```

```
DeAcc_borrow
        MOV     b1, #0
        ADD     b1, b1, b1, ESL #32          ; b1 := RE
DeAcc_bloop
        TEQ     pC, eC                       ; use TEQ to preserve the c-flag
        BEQ     DeAcc_exit
        LDMIA   pC, {t1}
        SBCS    t1, t1, b1
        STMIA   pC!, {t1}
        MOV     b1, #0
        BCC     DeAcc_bloop

DeAcc_exit
        MOV     pc, lr                       ; return

END
```

Appendix E

Maple Script for Optimised Squaring with SSW

MAPLE was used to implement the algorithm for optimised squaring using $SSW_{2,m}$ as described in Section 3 of Chapter 5. The results were compared with the correct square generated directly by Maple. While this does this does not provide proof of correctness, repeated simulations do give some confidence in the correctness of the algorithm.

```
#	A Maple script to test the correctness of the equations for SSW
#	optimised square. Generates results in the file sstest.in.
#	Usage: from within within maple type: read "sstestin.map"
#	Signed Sliding Square
#
with (numtheory):

#
#	C O N S T A N T S
#
#	n-word operands
n := 16:
R := 2^(n*32):
#	m-bit multiplier digits
m := 5:
r := 2^m:

#
#	F U N C T I O N S
#

#	A function to find n-word random numbers.
randfunn := rand(R/2..R-1):

#	A function to return bit i of a number.
```

```
bit := proc(A,i)
        floor((A mod (2^(i+1)))/2^i):
end:

#       A div function
div := proc(A,B)
        floor(A/B);
end:

#       A function to return the xor of the least significant bits of two numbers
xor := proc(A,B)
        (A mod 2 + B mod 2) mod 2:
end:

#       A function to print long hexadecimal numbers
#       Maple's printf %X format specifier is 'subject
#       to the restrictions of the machine architecture'
#       and won't print out very long integers.
#       At least w digits are printed.
fprinthex := proc(f,D,w)
        local h,j:
        h := convert(D,hex):
        # print leading zeros as required
        for j from length(h) to w-1 do
            fprintf(f,"0"):
        od:
  fprintf(f,"%s\n",h):
end:

#
#       M A I N   P R O C E D U R E
#

#       Open an output file
f := fopen("sstest.in",WRITE):

#       Choose and print the operand
A := randfunn():
fprinthex(f,A,n*8):

#       Perform the square
C := A*A:
fprinthex(f,C,2*n*8):

#       Initialise variables
P := 0:
T := A:
X := 0:
i := 0:
s := 0:

#       Main iteration
while i < 32*n do
```

```
        if T mod 2 = s then
            i := i + 1:
            T := div(T,2):
        else
            alpha := T mod r:
            a := s + alpha:
            T := div(T,r):

            # a is positive
            if T mod 2 = 0 then
                Ai := div((a*A),2^(i+m)):
                # Case A
                if s = 0 then
                    tmin := (a-1)*bit(alpha,m-1)/2 + div((a + (a-1)*(a mod 2^(m-1))),2^m):
                    t := tmin + xor(tmin,Ai):
                    Ai := Ai - t:
                # Case B
                else
                    tmin := (a-1)*bit(alpha,m-1)/2 + div(( (3*a-3)/2 + (a-1)*((a-1) mod 2^(m-
1))),2^m):
                    t := tmin + xor(tmin,Ai):
                    Ai := Ai - t:
                fi:
                s := 0:

            # a is negitive
            else
                a := a - 2^m:
                Ai := div(abs(a)*A,2^(i+m)):
                # Case C
                if s = 0 then
                    tmin := (abs(a)-1)*bit(alpha,m-1)/2 + div(2^m - abs(a) + (abs(a)-
1)*((2^m - abs(a)) mod 2^(m-1)),2^m):
                    t := tmin + xor(xor(tmin,Ai),1):
                    Ai := Ai + abs(a) - t:
                # Case D
                else
                    tmin := (abs(a)-1)*bit(alpha,m-1)/2 + div(2^m - ((abs(a)+3)/2) + (abs(a)-
1)*((2^m - abs(a) - 1) mod 2^(m-1)),2^m):
                    t := tmin + xor(xor(tmin,Ai),1):
                    Ai := Ai + abs(a) - t:
                fi:
                s := 1:
            fi:

            P := P + (a^2)*(2^(2*i)) + (a/abs(a)) * Ai * (2^(2*i+m+1)):
            fprintf(f,"i = %d, a = %d, tmin = %d, t = %d\n",i,a,t,tmin):
            fprinthex(f,abs(P),2*n*8):
            i := i + m:
        fi:
od:

#       Print the results
fprintf(f,"The correct answer is\n"):
```

```
fprinthex(f,C,2*n*8):
fprintf(f,"The difference is\n"):
fprinthex(f,abs(P-C),2*n*8):

#       Close the output file
fclose(f):
```

Bibliography

[AEGP67] S. F. Anderson, J. G. Earle, R. E. Goldschmidt and D. M. Powers. The IBM System/360 Model 91: Floating Point Execution Unit. *IBM Journal of Research and Development,* 11:34-53, 1967.

[AGLL95] Derek Atkins, Michael Graff, Arjen K. Lenstra and Paul C. Leyland. The Magic Words are Squeamish Ossifrage. *Lecture Notes in Computer Science: Advances in Cryptology - Asiacrypt 94*, 917:263-277, 1995.

[AM91] Giuseppe Alia and Enrico Martinelli. A VLSI Modulo *m* Multiplier. *IEEE Transactions on Computers*, 40(7):873-878, July 1991.

[AM95] Masayuki Abe and Hikaru Morita. Higher Radix Nonrestoring Modular Multiplication Algorithm and Public-key LSI Architecture with Limited Hardware Resources. *Lecture Notes in Computer Science: Advances in Cryptology - Asiacrypt 94*, 917:365-375, 1995.

[ARM94] Advanced Risc Machines. *ARM 7 Data Sheet.* Document Number ARM DDI 0020C, Available from http://www.arm.com, December 1994.

[ARM95] Advanced Risc Machines. *ARM Software Development Toolkit V2.0: Programming Techniques*. Document Number ARM DUI 0021A, Available from http://www.arm.com, June 1995.

[Atk68] D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers,* 17:925-934, October 1968.

[Atk70] Daniel E. Atkins. Design of the Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods. *IEEE Transactions on Computers.* 19(8):720-733, August 1970.

[Avi61] Algirdas Avizienis. Signed-Digit Number Representation for Fast Parallel Arithmetic. *IRE Transactions on Electronic Computers,* 10:389-400, 1961.

[AW93] Steven Arno and Ferrell S. Wheeler. Signed Digit Representations of Minimal Hamming Weight. *IEEE Transactions on Computers.* 42(8):1007-1011, August 1993.

[Bak87] P. W. Baker. Fast computation of $A * B$ modulo N. *Electronics Letters*, 23(15):794-795, July 1987.

[Bar87] Paul Barrett. Implementing the Rivest, Shamir and Adleman Public-Key Encryption Algorithm on a Standard Digital Signal Processor. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 86.* 263:311-323, 1987.

[BC90] J. Bos and M. Coster. Addition Chain Heuristics. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 89.* 435:400-407, 1990.

[BDeML97] Dan Boneh, Richard A. DeMillo and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 97*, 1233:37-51, 1997.

[BG89] Thomas Beth and Dieter Gollmann. Algorithm Engineering for Public Key Algorithms. *IEEE Journal on Selected Areas in Communications,* 7(4):458-466, May 1989.

[BGV93] Antoon Bosselaers, René Govaerts and Joos Vandewalle. Comparisons of three modular reduction functions. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 93.* 773:175-186, 1993.

[Bla83] G. R. Blakley. A Computer Algorithm for Calculating the Product AB Modulo M. *IEEE Transactions on Computers.* 32(5):497-500, May 1983.

[BMGW93] Ernest F. Brickell, Kevin S. McCurley, Daniel M. Gordon and David B. Wilson. Fast Exponentiation with Precomputation. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 92,* 658:200-207, 1993.

[Boo51] Andrew D. Booth. A Signed Binary Multiplication Technique. *Quarterly Journal of Mechanics and Applied Mathematics.* 4:236-240, 1951.

[Bov98] Ernst Bovelander. Smart Card Security, 'How Can We Be So Sure?'. *Lecture Notes in Computer Science: COSIC'97 Course*, 1528:332-337, 1998.

[Boy93] C. Boyd. Modern data encryption. *Electronics and Communication Engineering Journal,* pages 271-278, October 1993.

[Bri83] Ernest F. Brickell. A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography. In *Advances in Cryptology, Proceedings of Crypto 82,* Plenum Press, 1983.

[Bri90] Ernest F. Brickell. A Survey of Hardware Implementations of RSA. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 89,* 435:368-370, 1990.

[Bur94] N. Burgess. A Pre-Scaled Maximally-Redundant Radix-4 SRT Divider. *Electronics Letters.* 30(23):1926-1928, November 1994.

[Cas98] Cascade. *A Smarter Chip for Smart Cards.* Press release available from http://www.dice.ucl.ac.be/crypto/cascade, May 1998.

[Che71] Tien Chi Chen. A Binary Multiplication Scheme Based on Squaring. *IEEE Transactions on Computers,* 20:678-680, 1971.

[Chi93a] Che Wun Chiou. Parallel Implementation of the RSA Public-Key Cryptosystem. *International Journal of Computer Mathematics*, 48:153-155, 1993.

[Chi93b] Che Wun Chiou. A fast logic for modular multiplication. *International Journal of Electronics*, 74(6):851-855, 1993.

[CCY96] C.-Y. Chen, C.-C. Chang and W.-P. Yang. Hybrid method for modular exponentiation with precomputation. *Electronics Letters*, 32(6):540-541, March 1996.

[CL73] W. Edwin Clark and J. J. Liang. On Arithmetic Weight for a General Radix Representation of Integers. *IEEE Transactions on Information Theory,* 19:823-826, November 1973.

[CL87] H. Cohen and A. Lenstra. Implementation of a New Primality Test. *Mathematics of Computation.* 48(177):103-121, January 1987.

[CR90] Tony M. Carter and James E. Robertson. The Set Theory of Arithmetic Decomposition. *IEEE Transactions on Computers.* 39(8):993-1005, August 1990.

[CT95] T. Coe and P. T. P. Tang. It takes six ones to reach a flaw. In *Proceeding of the 12th Symposium on Computer Arithmetic*, pages 140-146, IEEE Computer Society Press, 1995.

[DC95] V. Dimitrov and T. Cooklev. Two Algorithms for Modular Exponentiation Using Non-Standard Arithmetics. *IEICE Transactions of Fundamentals*, E78-A(1):82-87, 1995.

[deR95] Peter de Rooij. Efficient Exponentiation using Precomputation and Vector Addition Chains. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 94,* 950:389-399, 1995.

[deWQ91] Dominique de Waleffe and Jean-Jacques Quisquater. CORSAIR: A Smart Card for Public Key Cryptosystems. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 90,* 537:503-513, 1991.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory,* 22:644-654, November 1976.

[Dhe98] Jean-François Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards.* PhD Thesis, Université Catholique de Louvain, May 1998.

[Dif88] W. Diffie. The First Ten Years of Public-Key Cryptography. *Proceedings of the IEEE,* 76(5):560-577, May 1988.

[DK91] Stephen R. Dussé and Burton S. Kaliski Jr. A Cryptographics Library for the Motorola DSP56000. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 90,* 473:230-244, 1991.

[Eld91] Stephen E. Eldridge. A Faster Modular Multiplication Algorithm. *International Journal of Computer Mathematics,* 40:63-68, 1991.

[ElG85] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory,* 31(4):469-481, 1985.

[Eve91] Shimon Even. Systolic Modular Multiplication. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 90,* 537:619-624, 1991.

[EW93] Stephen E. Eldridge and Colin D. Walter. Hardware Implementation of Montgomery's Modular Multiplication Algorithm. *IEEE Transactions on Computers,* 42(6):693-699, June 1993.

[FDG90] C. H. N. Forster, S. S. Dlay, R. N. Gorgui-Naguib. Carry Delayed Save Adders for Computing the Product $A.B$ Modulo N in $\log_2 N$ Steps. *Electronics Letters,* 26(18):1544-1545, August 1990.

[FFS98] U. Feige, A. Fiat and A. Shamir. Zero knowledge proofs of identity. *Journal of Cryptology,* 1:77-94, 1998.

[FJ90] P. A. Findlay and B. A. Johnson. Modular Exponentiation Using Recursive Sums of Residues. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 89,* 435:371-386, 1990.

[Fre67] Herbert Freeman. Calculation of Mean Shift for a Binary Multiplier Using 2, 3, or 4 Bits at a Time. *IEEE Transactions on Electronic Computers.* 16(6):864-866, December 1967.

[Ghe71] Clive Ghest. Multiplication made easy for digital assemblies. *Electronics,* 44:56-61, November 1971.

[GHM96] Dieter Gollmann, Yongfei Han and Chris J. Mitchell. Redundant Integer Representations and Fast Exponentiation. *Designs, Codes and Cryptography,* 7:135-151, 1996.

[GW96] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal,* pages 66-70, January 1996.

[HDVG88] Frank Hoornaert, Marc Decroos, Joos Vandewalle and René Govaerts. Fast RSA-Hardware: Dream or Reality? *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 88,* 330:257-264, 1988.

[Hen60] Herbert C. Hendrickson. Fast High-Accuracy Binary Parallel Addition. *IRE Transactions on Electronic Computers,* EC9:465-459, December 1960.

[HOY96] Seong-Min Hong, Sang-Yeop Oh and Hyunsoo Yoon. New Modular Multiplication Algorithms for Fast Modular Exponentiation. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 96,* 1070:166-177, 1996.

[HL94] L. C. K. Hui and K. -Y. Lam. Fast square-and-multiply exponentiation for RSA. *Electronics Letters,* 30(17):1396-1397, August 1994.

[HP94] C. Y. Hung and B. Parhami. Fast RNS division algorithms for fixed divisors with application to RSA encryption. *Information Processing Letters,* 51:163-169, 1994.

[HP98] Helena Handschuh and Pascal Paillier. Smart Card Crypto-Coprocessors for Public-Key Cryptography. *Cryptobytes Summer 1998 - The Technical Newsletter of RSA Laboratories,* 4(1):6-10, 1998.

[Hwa79] K. Hwang. *Computer Arithmetic Principles, Architecture and Design.* Wiley, 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,* 24(4):1071-1075, August 1989.

[IWSD92] Peter A. Ivey, Simon N Walker, Jon M Stern and Simon Davidson. An Ultra-High Speed Public Key Encryption Processor. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, 1992.

[JM89] J. Jedwab and C.J. Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronics Letters,* 25(17):1171-1172, August 1989.

[KAK96] Çetin Kaya Koç, Tolga Acar and Burton S. Kaliski, Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro,* 16(2): 26-33, June 1996.

[Kal95] Burton S. Kaliski Jr. The Montgomery Inverse and Its Applications. *IEEE Transactions on Computers,* 44(8):1064-1065, August 1995.

[KH88] Shin-ichi Kawamura and Kyoko Hirano. A Fast Modular Arithmetic Algorithm Using a Residue Table. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 88,* 330:245-250, 1988.

[KH90a] Ç. K. Koç and C. Y. Hung. Multi-Operand Modulo Addition Using Carry Save Adders. *Electronics Letters,* 26(6):361-363, March 1990.

[KH90b] Ç. K. Koç and C. Y. Hung. Carry-Save Adders for Computing the Product AB Modulo N. *Electronics Letters,* 26(13):899-900, June 1990.

[KH92] Çetin K. Koç and Ching-Yu Hung. Adaptive m-ary Segmentation and Canonical Recoding Algorithms for Multiplication of Large Binary Numbers. *Computers and Mathematics with Applications.*24(3):3-12, 1992.

[KH98] Ç. K. Koç and C. Y. Hung. Fast algorithm for modular reduction. *IEE Proceedings on Computers and Digital Techniques,* 145(4):265-271, July 1998.

[Knu69] Donald E. Knuth. *The Art of Computer Programming.* Volume 2, Seminumerical Algorithms, Addison-Wesley, pages 398-419, 1969.

[Knu97] Donald E. Knuth. *The Art of Computer Programming.* Volume 2, Seminumerical Algorithms, Third Edition, Addison-Wesley, 1997.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 96,* 109:104-113, 1996.

[Koç90] Çetin K. Koç. High-Radix and Bit Recoding Techniques for Modular Exponentiation. *International Journal of Computer Mathematics,* 40:139-156, November 1990.

[Kor93a] Israel Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[Kor93b] Peter Kornerup. High Radix Modular Multiplication for Cryptosystems. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic,* pages 277-283, IEEE Computer Society Press, 1993.

[Kor94a] Peter Kornerup. Digit-Set Conversions: Generalizations and Applications. *IEEE Transactions on Computers.* 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*, 43(8):892-898, August 1994.

[Kor99] Peter Kornerup. Necessary and Sufficient Conditions for Parallel, Constant Time Conversion and Addition. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic,* pages 152-156, IEEE Computer Society Press, 1999.

[KR80] E. V. Krishnamurthy and V. Ramachandran. A cryptographic system based on finite field transforms. *Proceedings of the Indian Academy of Science (Math. Sci.),* 89:75-93, 1980.

[KTS91] Shin-ichi Kawamura, Koyoko Takabayashi and Atsushi Shimbo. A Fast Modular Exponentiation Algorithm. *IEICE Transactions,* E-74(8):2136-2142, August 1991.

[KWH90] Michitaka Kameyama, Shugang Wei and Tatsuo Higuchi. Design of an RSA Encryption Processor Based on Signed-Digit Multivalued Arithmetic Circuits. *Systems and Computers in Japan*, 21(6):21-31, 1990.

[LH94] K.-Y. Lam and L. C. K. Hui. Efficiency of SS(l) square-and-multiply exponentiation algorithms. *Electronics Letters,* 30(25):2115-2116, December 1994.

[LL94] Chae Hoon Lim and Pil Joong Lee. More Flexible Exponentiation with Precomputation. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 94,* 839:95-107, 1994.

[LM95] Chung Nan Lyu and David W. Matula. Redundant Binary Booth Recoding. In *Proceedings of the 12th Symposium on Computer Arithmetic,* pages 193-199, IEEE Computer Society Press, 1995.

[MacS61] O. L. MacSorley. High-Speed Arithmetic in Binary Computers. *Proceedings of the IRE.* 49:91-103, January 1961.

[Man88] M. Morris Mano. *Computer engineering: hardware design.* Prentice-Hall International, 1988.

[McE78] R. J. McEliece. A public key cryptosystem based on algebraic coding theory. DNS Report 42-44, Jet Propulsion Laboratory, Pasadena, 1978.

[Men93] A. Menezes. *Elliptic curve public key cryptosystems*. Kluwer Academic Publishers, 1993.

[MOV97] A. J. Menezes, P. von Oorschot and S. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.

[Mon85] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519-521, April 1985.

[Mor90] Hikaru Morita. A Fast Modular-multiplication Algorithm based on a Higher Radix. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 89*, 435:387-399, 1990.

[NM'RR95] D. Naccache, D. M'Raihi and D. Raphaeli. Can Montgomery Parasites Be Avoided? A Design Methodology Based on Key and Cryptosystem Modifications. *Designs, Codes and Cryptography*, 5(1):73-80, January 1995.

[NM'R96] David Naccache and David M'Raïhi. Cryptographic Smart Cards. *IEEE Micro*, 16(3):14-23, June 1996.

[NS81] Michael J. Norris and Gustavus J. Simmons. Algorithms for High Speed Modular Arithmetic. *Congressus Numerantium*, 31:153-163, 1981.

[OK91] Holger Orup and Peter Kornerup. A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N. In *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 51-57, IEEE Computer Society Press, 1991.

[OM98] J.-H. Oh and S.-J. Moon. Modular multiplication method. *IEE Proceeding of Computers and Digital Techniques*. 145(4)317-318, July 1998.

[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. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 86*, 263:277-301, 1986.

[Oru95] Holger Orup. Simplifying Quotient Determination in High-Radix Modular Multiplication. In *Proceedings of the 12th Symposium on Computer Arithmetic*, pages 193-199, IEEE Computer Society Press, 1995.

[OSA91] Holger Orup, Erik Svendsen and Erik Andreasen. VICTOR: an efficient RSA hardware implementation. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 90*, 473:246-252, 1991.

[Par90] Behrooz Parhami. Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations. *IEEE Transactions Computers,* 39(1):89-98, January 1990.

[PB95] B. J. Phillips and N. Burgess. A Self-Timed Approach to Radix 2 SRT Quotient Digit Selection for GaAs VLSI Technology. In *Proceedings of the 13th Australian Microelectronics Conference*, pages 80-85, IREE, July 1995.

[Pey95] Patrice Peyret. Which Smart Card Technologies Will You Need to Ride the Information Highway Safely? A presentation from Smart Card 95. Available from http://www.dice.ucl.ac.be/crypto/cascade.

[PH78] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over *GF(p)* and its cryptographic significance. *IEEE Transactions on Information Theory,* 24:106-110, 1978.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, 1996.

[PP90] K. C. Posch and R. Posch. Approaching encryption at ISDN speed using partial parallel modulus multiplication. *Microprocessing and Microprogramming,* 29:177-184, 1990.

[QC82] J-J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters,* 18(21):905-906, October 1982.

[Rab79] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. MIT/LCS/TR-212, MIT Laboratory for Computer Science, 1979.

[Rei60] G. W. Reitwiener. Binary Arithmetic. *Advances in Computers,* 1:261-265, 1960.

[Riv85] Ronald L. Rivest. RSA Chips (Past / Present / Future). *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 84,* 209:159-165, 1985.

[Riv92] Ronald L. Rivest. Responses to NIST's Proposal. *Communications of the ACM,* 35:41-47, July 1992.

[Rob58] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, 7:218-222, September 1958.

[Rod85] J. R. Rodriguez. Improved Approach to the Use of Booth's Multiplication Algorithm. *IBM Technical Disclosure Bulletin,* 27(11):6624-6632, April 1985.

[RSA78] 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.

[RSA95] RSA Data Security. *Question & Answers Regarding the Paul Kocher Timing Attack on Public Key Cryptosystems.* Available from http://www.rsa.com, December 1995.

[RSA98] RSA Laboratories. *Frequently Asked Questions About Today's Cryptography.*Version 4, Available from http://www.rsa.com/rsalabs/faq, 1998.

[Rub75] Louis P. Rubinfield. A Proof of the Modified Booth's Algorithm for Multiplication. *IEEE Transactions on Computers,* 24(10):1014-1015, October 1975.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology,* 4:161-174, 1991.

[Sch96] Bruce Schneier. *Applied Cryptography.* Second Edition, John Wiley and Sons, 1996.

[Sed88] Holger Sedlak. The RSA Cryptography Processor. *Lecture Notes in Computer Science - Eurocrypt 87*, 304:95-105, 1988.

[SG90] Homayoon Sam and Arupraran Gupta. A Generalized Multibit Recoding of Two's Complement Binary Numbers and Its Proof with Applications in Multiplier Implementations. *IEEE Transactions on Computers*, 39(8):1006-1015, August 1990.

[Sim92] Gustavus J. Simmons. *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,* 34(3):290-292, March 1985.

[SM89] A. Selby and C. Mitchell. Algorithms for software implementations of RSA. *IEE Proceedings,* 136(E-3):166-170, May 1989.

[SD85] J. A. Starzyk and S. R. Dandu. Overlapped multi-bit scanning multiplier. In *Proceedings of the International Conference on Computer Design*, pages 363-366, October 1985.

[SV93] M. Shand and J. Vuillemin. Fast Implementations of RSA Cryptography. In *Proceedings 11th IEEE Symposium on Computer Arithmetic,* pages 252-259, IEEE Computer Society Press, 1993.

[Tak92] Naofumi Takagi. A Radix-4 Modular Multiplication Hardware Algorithm for Modular Exponentiation. *IEEE Transactions on Computers,* 41(8):949-956, August 1992.

[Tak93] Naofumi Takagi. A Modular Multiplication Algorithm with Triangle Additions. In *Proceedings 11th IEEE Symposium on Computer Arithmetic,* pages 210-214, IEEE Computer Society Press, 1993.

[TB91] F. A. Al-Tuwaijry and S. K. Barton. A High Speed RSA Processor. In *Proceedings of the 6th International Conference on Digital Processing of Signals in Communications,* pages 210-214, IEE, London, 1991.

[Tom89] A. Tomlinson. Bit-Serial Modular Multiplier. *Electronics Letters,* 25(24):1664, November 1989.

[TY92] Naofumi Takagi and Shuzo Yajima. Modular Multiplication Hardware Algorithms with a Redundant Representation and Their Application to RSA Cryptosystem. *IEEE Transactions on Computers,* 41(7):887-890, July 1992.

[VSH89] Stamatis Vassiliadis, Eric M. Schwarz and Don J. Hanrahan. A General Proof for Overlapped Multiple-Bit Scanning Multiplications. *IEEE Transactions on Computers,* 38(2):172-183, February 1989.

[VVDJ90] André Vandemeulebroecke, Etinne Vanzieleghem, Tony Denayer and Paul G. A. Jespers. A Single-Chip 1024-b RSA Processor. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 89*, 434:219-236, 1990.

[VVDJ90] André Vandemeulebroecke, Etinne 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,* 25(3):748-755, June 1990.

[VW98] Klaus Vedder and Franz Weikmann. Smart Cards - Requirements, Properties, and Applications. *Lecture Notes in Computer Science: State of the Art in Applied Cryptography, Course on Computer Security and Industrial Cryptography,* 1528:307-331, 1998.

[Wal91] Colin D. Walter. Fast Modular Multiplication Using 2-Power Radix. *International Journal of Computer Mathematics*, 39:21-28, 1991.

[Wal92] Colin D. Walter. Faster Multiplication by Operand Scaling. *Lecture Notes in Computer Science: Advances in Cryptology - Crypto 91,* 576:313-323, 1992.

[Wal93] Colin D. Walter. Systolic Modular Multiplication. *IEEE Transactions on Computers,* 42(3):376-378, March 1993.

[Wal94] C.D. Walter. Logarithmic speed modular multiplication. *Electronics Letters,* 30(17):1397-1398, August 1994.

[Wal95] C. D. Walter. Still faster modular multiplication. *Electronics Letters*, 31(4):263-264, February 1995.

[WE90] C. D. Walter and S. E. Eldridge. A verification of Brickell's fast modular multiplication algorithm. *International Journal of Computer Mathematics,* 33:153-169, 1990.

[WF82] Shlomo Waser and Michael J. Flynn. *Introduction to arithmetic for digital systems designers*. Holt, Rinehart and Winston, 1982.

[WH91] T. E. Williams and M. A. Horowitz. A Zero-Overhead Self-Timed 160ns 54-b CMOS Divider. *IEEE Journal of Solid-State Circuits.* 26(11):1651-1661, November 1991.

[Wie90] Michael J. Wiener. Cryptanalysis of Short RSA Secret Exponents. *IEEE Transactions on Information Theory,* 36(3):553-558, May 1990.

[Wil80] H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory,* 26:726-729, 1980.

[Yac91] Y. Yacobi. Exponentiating Faster with Addition Chains. *Lecture Notes in Computer Science: Advances in Cryptology - Eurocrypt 90*, 473:222-229, 1991

[Zha93] C. N. Zhang. An Improved Binary Algorithm for RSA. *Computers and Mathematics with Applications.* 25(6):15-24, 1993.

[Zur93] Dan Zuras. On Squaring and Multiplying Large Integers. In *Proceedings 11th IEEE Symposium on Computer Arithmetic,* pages 260-271, IEEE Computer Society Press, 1993.