Software and Hardware Implementation of the RSA Public Key Cipher

Volume 1(1)

Paul Brady
B.Sc(Eng), Hons. Dip. E.E.

Submitted to the National Institute for Higher Education, Dublin for the degree of Master of Engineering

This research was carried out under the supervision of Dr. T. Curran in the School of Electronic Engineering at the National Institute for Higher Education, Dublin and at the National Microelectronics Research Centre, Cork.

This thesis is based on the candidates own work

September 1988

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to

Dr. Tommy Curran for his expert advice, guidance and for his infinite patience.

Dr. Sverre Lidholm for his invaluable assistance.

Ericssons and the N.B.S.T for financial support of this project

The staff and postgraduate students at the N.M.R.C and N.I.H.E Dublin.

I would also like to thank all the people who helped me in this endeavour either directly or indirectly.

Contents

Abstract

Title : Software and Hardware Implementation of the RSA Public Key Cipher.

Author : Paul Brady

Cryptographic systems and their use in communications are presented. The advantages obtained by the use of a public key cipher and the importance of this in a commercial environment are stressed. Two two main public key ciphers are considered.

The RSA public key cipher is introduced and various methods for implementing this cipher on a standard, non-dedicated, 8 bit microprocessor are investigated. The performance of the different algorithms are evaluated and compared. Various ways of increasing the performance are considered. The limitations imposed by the performance on the practical use of the cipher are discussed.

The importance of the key to the security of the cipher is assessed. Different forms of attack are mentioned and a procedure for generating keys, which minimise the probability of a sucessful attack is presented. This procedure is implemented on a minicomputer. Use of the method on personal computers or microprocessors is examined.

Methods for performing multiplication in hardware, with particular emphasis on the use of these methods in modular multiplication, are detailed. An algorithm for performing part of the encryption function in hardware and the hardware necessary for it is described. Different methods for implementing the hardware are discussed and one is choosen. A description of the hardware unit is given.

The design and development of an application specific integrated circuit (ASIC) to perform key elements of the encryption function is described. The various stages of the design process are detailed. The results expected from this device and its integration into the overall encryption scheme are presented.

1.INTRODUCTION

1.1 GENERAL

The use of cryptography to protect confidential information has a long history. The Romans are certainly known to have used simple ciphers and one exists today which is attributed to Julius Caesar. Leonardo DaVinci is said to have used mirror writing to protect his ideas. Further details of the users of encryption throughout history and the methods that they used can be found in KAHN(1).

Until recently the main users of cryptography were the military. Commercial institutions did not, in general, consider it necessary to resort to encryption to protect information. Any use of cryptography by commercial institutions did not usually consider the security of the code or cipher being used. The main objective was to deter the casual eavesdropper, and not to secure against a determined attack by experience cryptanalysts.

This attitude of commercial organisations towards cryptography has undergone a sudden and dramatic change in recent years. There are two major factors which have influenced this change. The first is the advent of a worldwide telecommunications network, and the second is the increased use of electronic media to transfer and store information. Electronic information transmitted over the public telecommunications network must travel by cable, radio or microwave links, all of which are vulnerable to eavesdropping. Access to computers can now be obtained by virtually anyone with a terminal and modem, thus introducing the risk that confidential data may be exposed even when it is not in transit. The increasing use of electronic funds transfer has highlighted the problem not only of security but of authentication. Security in this case is insufficient, it is not enough to know that the data has not been altered between sender and receiver, it is necessary to verify that the data originated where it claims. Equally important is that the sender of the data cannot later disown it.

Traditional means of protecting data usually involve physical protection of the data itself, for example in a safe or with a trusted courier. The increased use of electronic storage renders this approach less effective. Information stored on magnetic disk or tapes can be physically protected by restricting access to the devices, however, when these devices are on line, or when information is in transit to or from such devices, it is vulnerable. Ways of protecting data both in storage and in transit are many but the method that is most widespread is the use of cryptography.

1.2 CRYPTOGRAPHY

The problem that cryptography seeks to address is that of a sender A who wishes to send a plaintext message P to a recipient B over an insecure channel. To do this A first encrypts the plaintext, using a transformation E, to produce ciphertext C. This ciphertext is now transmitted over the insecure channel. The intended recipient B then decrypts the ciphertext, using a transformation D, to produce the plaintext P again. This process of encryption and decryption is shown in Fig.(1.1) and can be described by the equations :

$$C = E(P)$$
$$P = D(C)$$

The transformations E and D must therefore have the property that D(E(P)) = P to allow B to decipher the message. In most cipher systems the general transformations are known but the specific transformations, which are determined by the encryption and decryption keys are not.

The function of the eavesdropper, shown in Fig.(1.1) is to recover the plaintext without knowing the decryption key D. Eavesdroppers can be classified into two categories passive and active. A passive eavesdropper can listen to and record, but he cannot alter, the data passing through the insecure channel. An active eavesdropper can dynamically alter data passing through the channel. He can therefore prevent data from reaching the intended recipient or he can record messages and send them to B at some later time. The active form of eavesdropping is particularly dangerous in electronic funds transfer where messages crediting an account could be recorded and replayed many times while messages debiting an account could be prevented from reaching their destination. this illustrates the need for verification, not only of the sender, but of the message itself. Message authentication can be achieved by inserting some non-repeating information, such as the date or a message number, into the message prior to encryption.

To recover the plaintext without knowing the decryption key D the eavesdropper, or cryptanalyst, must use the information available to him. The worst situation for the cryptanalyst is if he has a large amount of ciphertext, a general knowledge of the encryption method used and some information regarding the message. This knowledge may only extend as far as knowing the language in which the message was written or the probability of occurrence of certain words and phrases. This form of attack is the weakest and is known as a ciphertext only attack. If the cryptanalyst knows some plaintext and the corresponding ciphertext, e.g press releases which are intercepted in encrypted form which are released later, he has more information with which to work. This form of attack is described as a known plaintext attack. The most favourable situation for the cryptanalyst is when he can submit messages for encryption and obtain the corresponding ciphertext. This is called a chosen plaintext attack. To be considered secure an encryption system must withstand each of these attacks.

1.3 SECURITY OF CRYPTOGRAPHIC SYSTEMS

The security of a cipher can best be described in terms of the resources required by the cryptanalyst to determine the decryption key. Cryptographic systems can be either absolutely (or unconditionally) secure or they can be computationally secure. Absolute security implies that, regardless of the resources available to the cryptanalyst, he has insufficient information to obtain the decryption key and hence the plaintext. With computationally secure systems, however, it is possible to obtain the decryption key provided that sufficient resources are available. The designer of a computationally secure system must ensure that encryption and decryption are fast and inexpensive but that cryptanalysis would require more computational resources than would be possible to obtain. Computers are becoming more powerful and less expensive all the time, so it is necessary to include a very large margin in the system to allow for future developments. Even with this large margin computationally secure systems eventually become insecure, thus information which must remain secure for a long period of time e.g government archives should not be encrypted using such a system. All practical cryptographic systems are computationally secure.

With few exceptions it is not possible to prove that a cryptographic system is secure. A system is considered secure if it has withstood a concerted attack by experienced cryptanalysts for an extended period of time. This attack would be undertaken under the most favourable conditions possible. This process of testing the security of a system is known as certification. The only cipher that can be proven to be absolutely secure is the one time pad. In this cipher the plaintext is encrypted using a random key which is the same length as the plaintext. This system is absolutely secure as the key is random and never repeats thus concealing totally the statistical properties of the plaintext. This cipher is not in widespread use as the management of keys makes it impractical. To exchange a plaintext message the sender and receiver must first agree on a key which is the same length as the plaintext. As this key exchange requires a secure channel this channel could be used to transfer the plaintext instead. The use of such a system is therefore limited to high security diplomatic links where total security is vital and cost is not a factor.

1.4 CRYPTOGRAPHIC SYSTEMS

Cryptographic systems fall into two main categories, conventional or single key systems, and public or two key systems. In a conventional system the encipherment and the decipherment key are the same, or one can be easily obtained from the other. Two people who wish to exchange information thus share a common key. Each pair of users therefore requires a different key and the number of keys required for large numbers of user-pairs quickly becomes unmanageable, i.e for N user pairs the number of keys is $(N^2 - N)/2$. Before contact is made between two users it is necessary that they both receive a key and this can be time consuming for once off or infrequent communication.

1.4.1 Public Key Systems

In public key systems the encipherment key E and the decipherment key D are different and D cannot easily be obtained from E. It is therefore possible to make E public while ensuring that D remains secret. The public keys of many users could be published regularly as in a telephone directory. Any person who wishes to send secure information to a user A need only look up A's public key in the directory and use it to encrypt the message. The only person who can now read the message is A as only he knows the secret decryption key. Contact between user- pairs can now be established without the need for a secure key exchange and, if there are N users the number of key pairs required is also N. The reduced key management problem of public key ciphers make them attractive for use in a commercial environment where the number of contacts is large and constantly changing.

The idea of public key ciphers was introduced by DIFFIE and HELLMAN(2) in 1976. They did not however give a practical example of such a system. This paper was soon followed by two papers, one by MERKLE and HELLMAN(3) and another by RIVEST,SHAMIR and ADLEMAN(4), both of which described practical public key cryptosystems. These two systems were based on a mathematical concept known as a trapdoor function. A trapdoor function is a transformation that is essentially one way unless some secret information, the trapdoor, is known. Consider the use of such a transformation in a cryptographic system. The trapdoor function is the encryption key which converts the plaintext into ciphertext. Now to anyone, without the secret information, this transformation is irreversible. The secret information is thus the decryption key. The method proposed by Merkle and Hellman is based on the knapsack problem while that proposed by Rivest, Shamir and Addleman, and hence known as the RSA method, is based on the difficulty of factoring large numbers. These two methods are the basis of public key cryptography.

1.4.2 Conventional Systems

Unlike public key cryptography, which is of recent origin conventional encryption systems have a much longer history. This has led to the development of a large number of cipher systems being developed. However, along with this development there has been a parallel increase in methods of cryptanalysis of such ciphers. The two main groups into which most ciphers fall are, substitution and transposition ciphers.

1.4.2.1 Substitution Ciphers

In a substitution cipher each character in the plaintext is replaced by a character from an alternative alphabet. The simplest substitution cipher is the Caesar cipher, named after its reputed inventor Julius Caesar. In this cipher, shown in Fig.(1.2), the alternative alphabet is simply the regular alphabet shifted by three places e.g A becomes D etc. The security of such a system is very low as once the method of encryption is known it is a simple matter to cryptanalyse the ciphertext. Variations of this in which the alternative alphabet differs are similarly prone to cryptanalysis. One method of solution is to simply try all possible keys, of which there are only twenty six, the second method is to attempt a statistical attack. A statistical attack relies on the fact that certain letters in English appear more often than others e.g E,A. With sufficient ciphertext it is possible to create a table of probabilities and match them with standard English. The use of a large alphabet in a substitution cipher increases its security as the number of possible keys increases.A substitution cipher can also be easily broken by a known plaintext attack.

To increase the security of a substitution cipher more than one alternative alphabet can be used, under the control of a key. This is known as a polyalphabetic cipher, Fig.(1.4) and is much more difficult to solve, as the probability distribution of letters is now much flatter and does not resemble standard English. Given sufficient ciphertext, however, it is possible to solve such a cipher using a method developed by KASISKI(5). This method is based on determining the length of key by looking for repetitions in the ciphertext. The longer the key, therefore, the more difficult it is to solve. To increase the apparent keylength multiple encryption can be used e.g if the plaintext is encrypted twice, using keys whose lengths are relatively prime to each other the apparent key length is the product of the individual keylengths. As the key length increases the amount of ciphertext required and the time needed to solve the cipher increase, thus affording an increasing level of security.

1.4.2.2 Transposition Ciphers

In a transposition cipher the plaintext is split into fixed length blocks and the letters are rearranged under the control of a key, as shown in Fig.(1.3). The size of the block must be large to prevent the cryptanalyst from trying all possible keys to obtain meaningful plaintext. A transposition cipher can also be defeated by a statistical attack. In this method the frequency of occurrence of common letter pairs e.g TH, QU, IN is used to obtain transformations which will reunite such pairs. A transposition cipher is completely vulnerable to a known plaintext attack.

1.4.2.3 Product Cipher

Transposition and substitution ciphers do not possess adequate security for practical use. They can however be used together to produce a much stronger cipher known as a product cipher. A product cipher is of the form ST where S is a simple substitution cipher on a large alphabet and T is the transposition of bits within a fixed length block. Repeated encipherment using a different substitution key each time produces a strong cipher. One such cipher is the DES, or Data Encryption Standard (6).

1.4.2.3.1 The Data Encryption Standard (DES)

The National Bureau of Standards in the U.S.A issued a requirement for an encryption scheme which could be used as an encryption standard by the Federal authorities. The result was a product cipher developed by IBM and it is this which is now the Data Encryption Standard. This cipher uses a combination of transposition and substitution to achieve a very high level of security. This cipher, uses a 56 bit key to perform 16 rounds of encryption on a 64 bit block. Each round of encryption is a combination of substitution and transposition. The DES cipher is considered computationally secure as the only known way of solving it is to search all possible keys. With 2^{56} , or approximately 10^{17} possible keys such a search is computationally infeasible at present. However with the speed and power of computers increasing all the time the life of the DES will be short lived, by 1990 it is believed that the DES will be insecure.

The selection of a standard by the U.S government and the subsequent availability of inexpensive and secure encryption methods has helped the use of encryption to become more widespread. The DES algorithm is available in the form of a single integrated circuit which can be easily added to new or existing products(18)The setting up of a standard also ensures widespread acceptance of the method by commercial organisations and thus enables a large number of users to communicate with each other.

1.5 KEY MANAGEMENT

The DES is a secure and easily implementable cipher but it is prone to the same problem of key management that is common to all conventional cryptosystems. The problem of key management is two fold. The first is caused by the need for each user to store secretly all the keys it uses. This is a major problem for large commercial organisations with many contacts as the number of keys that must be securely stored is large. The second problem is one of contact initiation. A user A wishing to communicate securely with a user B has first to establish a secret key. This requires a secure channel, e.g a courier. The setting up of this key introduces a large time delay before the two users can communicate. Also if the requirement was only for a single secure transaction this could be accomplished using the secure key channel and the cipher need not be used.

Public key ciphers, which will be discussed in more detail in chapter 2 do not suffer from this key distribution problem. The public key of all users can be published or held in a public database. This removes the need for each user to maintain a large collection of secret keys, the only key that must be stored is the users secret decryption key. A user who now wishes to establish a secure communication channel with another user need only look up that users public key. This abolishes the need for an expensive and time consuming secure key channel.

Despite this lack of a key distribution problem public key ciphers have not yet gained widespread acceptance. There are several reasons for this. Public key ciphers, in general involve a more complex process than conventional ciphers and they are therefore slower. This reduces the transmission rate that can be obtained over a channel and rules them out for applications in which high speed communication is essential. Due to the complexity of the public key encryption process no single chip solution is commercially available at present. This makes the inclusion of a such a cipher into a new or existing product much more difficult than with DES.

Another problem with public key ciphers is that due to their relative youth there are still doubts regarding their security. Conventional ciphers such as DES have undergone extensive attacks by cryptanalysts for an extended period of time. This is the only sure way of testing the security of a system. Public key ciphers being relatively new have not undergone such a period of extensive testing. Conventional ciphers, such as DES are based on well understood principles so the prospect of finding a hidden flaw in such a system is quite small. One public key cipher, the RSA method, has received extensive attention since it was proposed in 1978 and has resisted attempts to render it insecure. This system is now being proposed as a federal standard (7). If it is accepted semiconductor manufacturers may produce single chip implementations of the cipher. This would enable it to be introduced into products as easily as DES. The problem of key management would thus be greatly reduced.

1.6 IMPLEMENTING CIPHERS

Methods of implementing ciphers vary according to the application. Where speed is not a problem encryption and decryption algorithms can be written in a high level programming language. This could be used, for example, before and after long term storage of computer files. This would remove the need for secure storage of computer backups etc. For higher speed applications a microprocessor, programmed in a low level language, can be used. For higher speeds hardware assisted microprocessors or dedicated hardware can be used. Cipher systems implemented in software can be rendered insecure if the programme itself is violated. The ideal encryption device is therefore a single chip VLSI device on which the secret key is stored, perhaps on EPROM, so that is is never available outside the chip.

1.7 OBJECTIVES

The objectives of this thesis are two fold. The first is an investigation into methods of implementing the RSA public key cipher on a standard microprocessor. The results of this investigation will determine whether such a system is practically viable. The second part is an investigation into the possible uses of hardware to improve the speed and security of the cipher.
Various methods of hardware encryption are considered and a solution is chosen. The means of implementing this system are described and ways of testing it are discussed.

Chapter 2 will discuss public key ciphers. Methods for producing public key ciphers are described with particular attention being paid to the RSA method. Different encryption schemes are compared and the advantages and disadvantages of each method is discussed. Ways of determining the security of each and the possible means of attacking the systems are examined. The importance of the key to the security is described and the different levels of security obtainable with different key lengths is tabulated for the RSA method. Implementation of the RSA method, in general, and of modular exponentiation, in particular is considered.

Chapter 3 investigates ways of performing modular multiplication, which is a vital part of the RSA scheme, using a standard eight bit microprocessor. Several schemes are described and flowcharts for each are given. A detailed timing analysis of each method is performed and graphs of the relative performance of each are produced. From this timing analysis the time to encrypt blocks of data of different lengths is calculated and graphed. The practical uses of such a system are discussed and ways of improving the performance of the system are considered.

Chapter 4 considers the problem of key generation in detail. The importance of the proper choice of key, and the implications of key choice for security are discussed. A practical key generation scheme, for use on a personal computer or a mainframe, is detailed with the aid of flowcharts. The expected execution time of the programme, and the effect of this on the overall encryption scheme are considered. Ways of implementing the key generation algorithm in a low level language, for use on a microprocessor are discussed.

Chapter 5 details methods for performing multiplication in hardware with particular emphasis on the use of these methods in modular multiplication. These methods are compared and the advantages and disadvantages of each noted. The tradeoffs inherent in each choice are considered. A method is chosen for implementation and the reasons for the choice are given. Various ways of implementing this choice are discussed and one is chosen.

Chapter 6 describes the design and development of the hardware unit. The various stages of the design process are described and the results expected are mentioned. A description of the hardware unit is given and the test performed to estimate its performance are given. Testing of the completed unit is examined. The effect of the hardware unit on the encryption process, in terms of the time taken to encrypt various block sizes is discussed.

Chapter 7 is a summary of the result achieved by this research. The implications of these results are considered and possible commercial applications are suggested. Further improvements and refinements are considered and the effect of continuing advances is speculated upon.

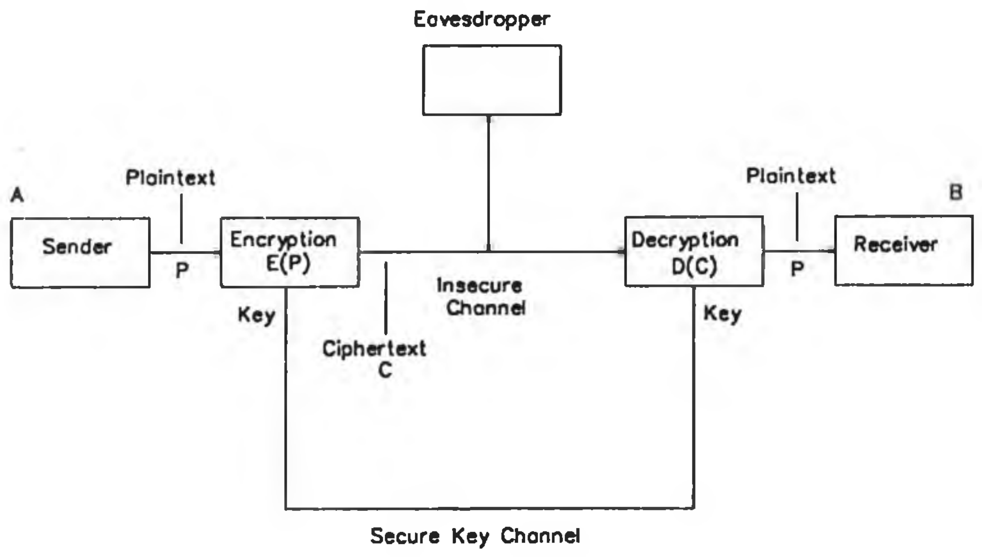


Fig. 1.1 Cryptography in communication

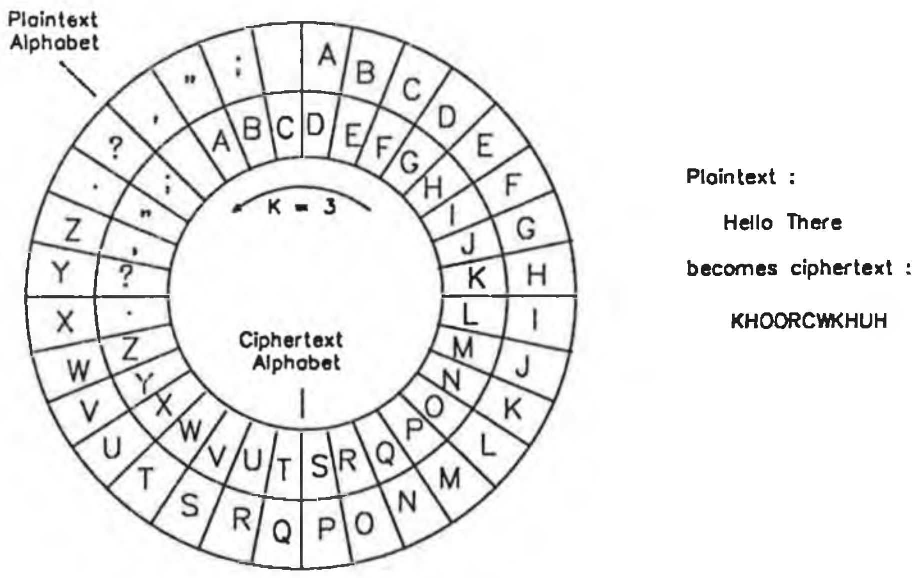
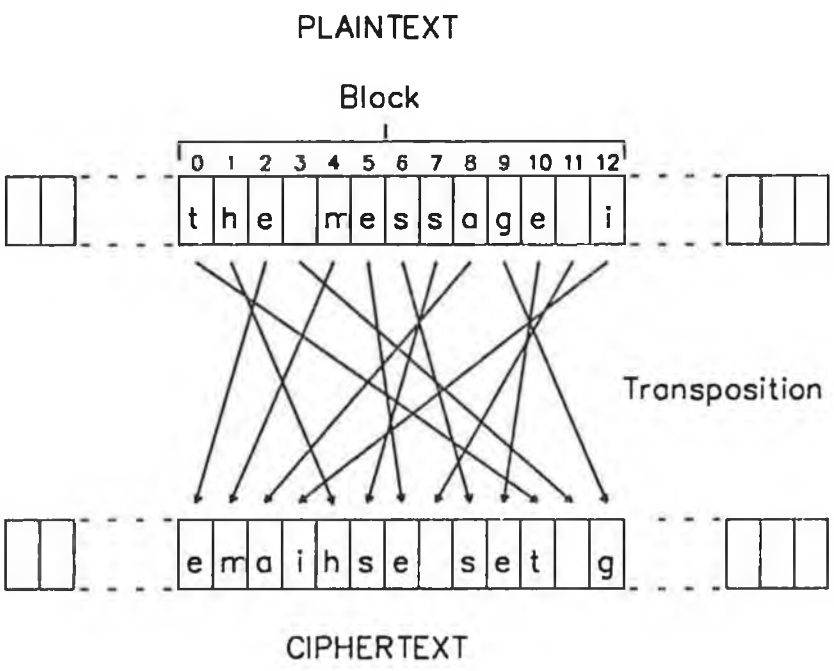


Fig. 1.2 Caesar cipher



Key = (2 4 8 12 1 7 5 11 6 10 0 3 9)

Fig. (1.3) Transposition Cipher

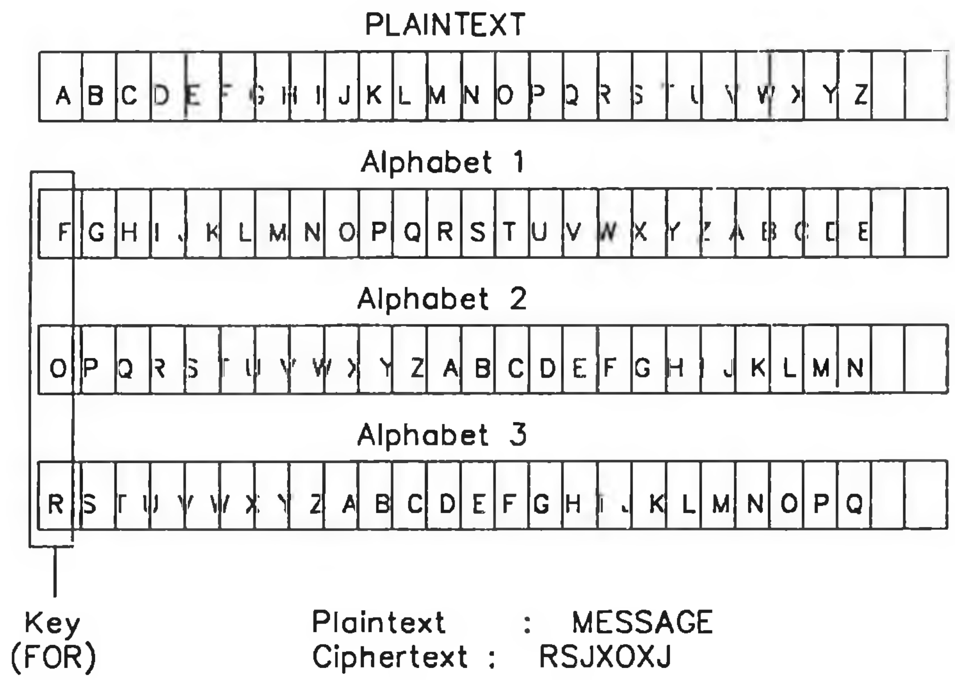

PLAINTEXT
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Alphabet 1
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
Alphabet 2
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
Alphabet 3
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
Key
(FOR)
Plaintext : MESSAGE
Ciphertext : RSJXOXJ

Fig. (1.4) Polyalphabetic Substitution Cipher

2. PUBLIC KEY SYSTEMS

2.1 INTRODUCTION

The concept of a public key system was first introduced by DIFFIE and HELLMAN (2) in 1976. They considered the problem of key distribution, which is a major disadvantage of conventional cryptosystems and limits their widespread use in commercial environments. Their approach was to find cryptosystems which would remove the need for a secure channel to exchange keys. This would eliminate the time consuming and expensive process of establishing a secure channel between two parties. Two alternatives were suggested, public key distribution and public key ciphers.

Public key distribution allows two users A and B to establish a key, which can then be used in a conventional encryption algorithm, by communicating over an insecure channel. This insecure channel can be the channel over which the conventional cryptosystem will be used. An eavesdropper listening in on the key establishment procedure must not be able to obtain sufficient information to make it computationally feasible to find the key. DIFFIE and HELLMAN give an example of a public key distribution system which will be repeated here to clarify the procedure.

A user, i, who wishes to be included in the public key distribution scheme must first generate a random number, r_i from a finite field GF(N) i.e a positive integer less than N. This number is kept secret but the user publishes the key, K_i, which is obtained from r_i by the following relationship:

$$K_i = a^{r_i} \bmod N$$

where a is a fixed primitive element of GF(N).
Two users, i and j, who wish to initiate secure communication arrive at a common key, K_{ij} using the

information available in the public directory. For user i,

$$K_{ij} = a^{r_i r_j} \bmod N$$

$$= (a^{r_j} \bmod N)^{r_i} \bmod N$$

$$= (K_j)^{r_i} \bmod N$$

where r_i is known only to user i and K_j is obtained from the public directory. User j arrives at the same key by calculating,

$$K_{ij} = a^{r_i r_j} \bmod N$$

$$= (K_i)^{r_j} \bmod N$$

whereas in this case r_j is known only to user j and K_i is obtained from the public key. Users i and j have thus arrived at a common key without the need for a secure key channel.

The security of this method relies on the difficulty of calculating logarithms in a finite field GF(N). For a cryptanalyst to determine K_{ij} it is necessary that he know r_i or r_j. These are not available directly as each user has kept their r number secret. The eavesdropper must therefore obtain r_i (or r_j from the corresponding K_i. Now

$$K_i = a^{r_i} \bmod N$$

therefore

$$r_i = \log_a(K_i) \bmod N$$

The system will be insecure if logarithms are computationally easy to calculate. The fastest algorithm, which is applicable for any value of N, is attributed to KNUTH(8) and has a run time which is $O(N^{0.5})$. Thus for large N ($>10^{50}$) this is computationally infeasible. Faster algorithms are available; POHLIG and HELLMAN(10), ADDLEMAN(11) but these are generally dependent on some property of the generator a or the modulus N. The choice of a and N should be so as to defeat or slow down these algorithms.

2.2 PUBLIC KEY CIPHERS

A public key cipher, fig. 2.1, differs from conventional ciphers in that the encryption and decryption keys are different and it is not computationally feasible to calculate the decryption key if the encryption key is known. In a system of this kind the encryption key can be made public. A user, A, who wishes to communicate privately with another user B, obtains B's encryption key from the public directory and uses it to encrypt the message. B can reply by obtaining A's encryption key from the public directory. As each user keeps his decryption key secret only he can decrypt the message, however anyone with access to the public directory can send secure information to another party without the need for a prior agreement on keys.

In a public key system the encryption algorithm, E, and the decryption algorithm, D, must have the following properties:

. D(E(P)) = P
. E, D form a distinct pair.
. It is computationally infeasible to calculate D from E
. It is easy to encrypt and decrypt data if E an D are known.

The first property ensures that each encryption key has an inverse, thus allowing the user to decrypt the ciphertext. The second property ensures that each encryption key has a unique decryption key. The third property makes it possible to publish E without compromising the security of the system, while the fourth minimises the work necessary for encryption and decryption.

Diffie and Hellman in their original paper did not propose a practical scheme for implementing a public key cipher. However soon after MERKLE and HELLMAN(3), and RIVEST, SHAMIR and ADLEMAN(4) proposed practical public key ciphers. The cipher proposed by Merkle and Hellman is based on the knapsack problem while that proposed by Rivest, Shamir and Adleman, and hence known as the RSA method is based on modular exponentiation and the difficulty of factoring large numbers.

2.2.1 The Knapsack Problem

The knapsack problem, see Fig.(2.2), is an old arithmetic problem and can be described as follows. Given a positive integer C and a vector of elements A = $\{a_1, a_2, \ldots\ldots a_n\}$ the knapsack problem is to find a subset of A that sums to C i.e.

$$C = A.M = \sum_{i=1}^{n} a_i m_i$$

where M = $\{m_1 \ldots\ldots m_n\}$ is a binary vector. The knapsack problem appears in many commercial applications. For example a freight company wishing to maximise the value of cargo carried by boat, truck or air, and given a finite storage space and a collection of cargo, is faced with solving the knapsack problem.

The difficulty of solving a given knapsack varies depending on the elements of A. If the elements of A are superincreasing i.e if each element a_i is greater by one unit than the sum of all previous a_i's, the solution is not difficult and such a knapsack is called an easy knapsack. The simplest example of an easy knapsack is one based on the binary sequence A = {1,2,4,8,.....} and in this instance the solution is trivial. The binary vector M is simply the binary representation of C. Not all knapsacks are easy to solve, the generalised knapsack problem is known to belong to the class of problems known as NP-complete which can not be solved in polynomial time on a deterministic machine. The best known algorithm for solving the knapsack problem is attributed to

SCHROEPPEL(12) and takes $O(2^{n/2})$ in time and $O(2^{n/4})$ in memory. This information does not help greatly in determining how the knapsack problem can be used in developing a public key cipher as it tells us that at best a solution to the knapsack problem is trivial while at worst it is computationally infeasible, for large n.

To make use of the knapsack problem as a public key cipher Merkle and Hellman constructed a trapdoor by taking an easy knapsack and transforming it into a hard knapsack. Anyone not possessing the secret trapdoor information would find it computationally infeasible to solve the knapsack, however, with the trapdoor information the problem can be converted to an easy knapsack and quickly solved.

To construct a trapdoor knapsack it is first necessary to form an easy knapsack by choosing a set of elements $A = \{a_1 ... a_n\}$ such that A is superincreasing i.e

$$a_i > \sum_{j=1}^{i-1} a_j$$

Then choose two large numbers, a modulus, r and, a multiplier, s that are relatively prime i.e gcd(r,s) = 1. Use these numbers to calculate the hard knapsack $B = \{b_1 ..... b_n\}$ where b_i is :

$$b_i = s.a_i \bmod r$$

The elements of b are not superincreasing and thus B is a hard knapsack. This hard knapsack $B = \{b_1 .... b_n\}$ constitutes the users public key. All other information, particularly s,r,A should remain secret.The decryption key d, which is the multiplicative inverse modulo r of s i.e ds 1 (mod r), can be calculated and should also be kept secret.

A user X who wishes to send a secure message $M = \{m_1 ... m_k\}$, where m_i is a binary digit, to user Y first breaks the message into n bit blocks. These blocks are then encrypted using Y's public key producing the ciphertext C where

$$C = M.B = \sum_{i=1}^{n} m_i b_i$$

where B = $\{b_1.....b_n\}$ is the public encryption key of Y. The ciphertext can now be transmitted to Y over an insecure channel. On receipt of the ciphertext, C, user Y retrieves the plaintext by applying his secret decryption key d :

$$dC = d \sum_{i=1}^{n} m_i b_i = \sum_{i=1}^{n} d m_i b_i$$

However,

$$db_i \equiv dsa_i \equiv a_i \pmod r$$

as,

$$ds \equiv 1 \pmod r$$

Thus

$$dC = \sum_{i=1}^{n} m_i a_i \pmod r$$

and the problem is to recover m_i. This is now an easy knapsack problem as the set A = $\{a_1......a_n\}$ is superincreasing.

The following simple example illustrates the method :

A = { 1,3,5,10 }
r = 23
s = 9

The easy knapsack A = { 1,3,5,10 } is transformed into the hard knapsack B .

B = A.s mod r
= (9 mod 23,3*9 mod 23,5*9 mod 23,10*9 mod 23)
= (9, 4, 22, 21)

this is the users encryption key which can be published in a directory or a public database. The decryption key can be calculated by solving

$$ds \equiv 1 \pmod{r}$$
$$d*9 \equiv 1 \pmod{23}$$

using Euclids algorithm, resulting in $d = 18$. This key should not be disclosed. To encrypt a message $M = 7 = (0111)_2$ to form ciphertext C use

$$C = B.M = 9 + 4 + 22 = 35 = (100011)_2$$

The ciphertext can be decrypted by using the secret key d to form the simple knapsack E.

$$E = dC = 18*35 \pmod{23} = 9$$

This simple knapsack (E,A) can be solved to give M. Such a knapsack can be solved in linear time.

The security of the knapsack cipher system relies on the difficulty of solving hard knapsacks. As mentioned previously the fastest known algorithm can solve an nth order knapsack in time $O(2^{n/2})$. For this to be computationally infeasible it is recommended that n be greater than 100. If n is 100 then the time taken to solve the hard knapsack, assuming that a single operation takes one microsecond, is approximately 10^9 seconds or 32 years. However a large number of processors working in parallel would reduce this time considerably and this would have to be considered when choosing n.

The choice of A, r and s is also important to the security of the cipher. These should be chosen from a sufficiently large set so that a direct search is not computationally feasible. For $n = 100$ Merkle and Hellman recommend that the numbers be chosen from the following ranges :

$$2^{201} + 1 \leqslant r \leqslant 2^{202} - 1$$
$$2 \leqslant d \leqslant r - 2$$
$$1 \leqslant a_1 \leqslant 2^{100}$$
$$2^{100} + 1 \leqslant a_2 \leqslant 2*2^{100}$$

$$3*2^{100}+1 \leqslant a_3 \leqslant 4*2^{100}$$

$$\vdots \qquad \vdots \qquad \vdots$$

$$(2^{i-1}-1)2^{100}+1 \leqslant a_i \leqslant 2^{i-1}.2^{100}$$

This ensures that each parameter is can have one of 2^{100} possible values thus defeating a direct search. Note that as each element of the public key B is less than r they will require 202 bits each for their representation. Thus the public key of each user will require approximately 20k bits of storage. An n bit message M when encrypted will require 202 + $\log_2(100)$ or 209 bits to represent it. The message is therefore expanded by a factor of 2.09. If the ciphertext is to be transmitted over a bandlimited channel this will result in a halving of transmission rate when compared with unencrypted transmission.

2.2.2 The RSA Cipher

The RSA public key cipher is based on modular exponentiation and the difficulty of factoring large numbers. To encrypt a plaintext message, M, it is first divided into blocks and then raised to the e^th power modulo N, where e and N are the encryption keys of the user i.e

$$C = M^e \text{ Mod } N$$

The size of the plaintext block should be such that M is always less than N. To recover the plaintext from the ciphertext, the ciphertext C is raised to the d^th power modulo N where d is the users decryption key i.e

$$M = C^d \text{ Mod } N$$

Note that as all arithmetic is performed modulo N there is no data expansion from plaintext to ciphertext.

To make use of this method in a public key cipher the key parameters e,d,N have to be carefully chosen. Rivest, Shamir and Adleman recommend the following procedure for their selection.

First choose two large prime numbers p and q. There are restrictions necessary when choosing such primes to ensure that the cipher is resistant to cryptanalysis. These restrictions and the procedures for choosing large primes will be described in a later chapter. These primes must be chosen from a large enough set that a direct search is computationally infeasible. The modulus N is the product of these two primes:

$$N = pq$$

The encryption key, e, can now be chosen to be a large random number that is relatively prime to $\emptyset(N)$ where $\emptyset(N)$ is the Euler totient function of N :

$$\gcd(e, \emptyset(N)) = 1$$

where gcd is the greatest common divisor. The Euler totient function of an integer, N, is the number of integers less than N and relatively prime to N. For a prime number the Euler totient function is one less than the prime number itself so

$$\emptyset(N) = \emptyset(pq) = (p-1)(q-1)$$

This is the users encryption key and it can be published, along with N , in a public directory or database. The decryption key, d, can be calculated as the multiplicative inverse of e modulo $\emptyset(N)$. Such a number exists and is unique as e is relatively prime to $\emptyset(N)$. the decryption key is therefore given by:

$$ed \equiv 1 \pmod{\emptyset(N)}$$

This equation can be solved using Euclids algorithm to find d. This is the decryption key of the user and it should not be made public. Note that due to the nature of the cipher e and d are interchangeable and either one can be made public retaining the other to use as a decryption key.

To test the validity of the RSA cipher it is necessary to prove that decryption of an encrypted message will yield the original plaintext i.e

$$D(E(M)) = M$$

where D denotes the decryption function and E the encryption function. For the RSA cipher

$$E(M) = C = M^e \text{ Mod } N$$

and $D(E(M)) = D(C) = C^d \text{ Mod } N = (M^e \text{ Mod } N)^d \text{ Mod } N$
$= M^{ed} \text{ Mod } N$

but $ed = 1 \; (\text{ Mod } \varnothing(N))$
$\Rightarrow$ $ed = k\varnothing(N)+1$, for some integer k.

thus

$$D(E(M)) = M.M^{k\varnothing(N)} \text{ Mod } N$$

Now for any integer, a, relatively prime to O(N) Eulers generalisation of Fermats theorem states that :

$$a^{\varnothing(N)} \equiv 1 \; (\text{ Mod } N) \quad , \text{for any integer a.}$$

therefore

$$D(E(M)) \equiv M \; (\text{Mod } N)$$

and as M is less than N :

$$D(E(M)) = M$$

Thus it is seen that the functions described in the RSA method do perform encryption and decryption.

As an example of the use of the RSA cipher consider the following example, note that the numbers involved here are too small for a practical cipher but the method is the same.
Choose p and q , the random primes to be:

$$p = 53, \; q = 61$$

Hence the modulus N is 53 * 61 = 3233 and $\emptyset(N)$ the Euler totient function of N, is 52 * 61 = 3172. To choose e we select a number in the required range and check to see if it is relatively prime to $\emptyset(N)$. If it is not another number must be used. let d = 279, which is relatively prime to 3172, this can be proven using Euclids algorithm. To find the encryption key e we need to solve the equation :

$$d * 279 = 1 \ (\mathrm{Mod}\ 3172)$$

this can again be solved using Euclids algorithm. This results in a value for e of 71. To encrypt the message RENAISSANCE we first assign numbers to each letter say A = 00.....Z = 25 and then divide the message into 4 digit blocks. This gives

$$M = (1704\ 1300\ 0818\ 1800\ 1302\ 0426)$$

To encrypt this message each block is raised to the power of 71 modulo 3172 giving the ciphertext C

$$C = (3106\ 0100\ 0931\ 2691\ 1984\ 2927)$$

To recover the plaintext message M the same procedure is carried out with d = 279 as the exponent.

The security of the RSA cipher relies on the difficulty of finding d when e and N are known. As d is the multiplicative inverse of e modulo $\emptyset(N)$ it is necessary to determine $\emptyset(N)$. $\emptyset(N)$ is formed by subtracting one from each of the prime divisors of N and forming their product i.e

$$\emptyset(N) = (p-1)(q-1) = 1 + N - p - q$$

If p and q are known $\emptyset(N)$ can be calculated and the decryption key d can also be found. The problem therefore reduces to finding the prime factors p, q of N. Factoring a number into its prime factors is easy when N is small but becomes increasingly more difficult as N increases.

Various factoring algorithms have been suggested and KNUTH(8) describes some of these in detail. The fastest known algorithm for factorisation attributed to SCHROEPPEL and detailed in KNUTH(8 , pp. 380-384) has a runtime of the order of n^r where r is :

$$(\mathrm{Ln}(N) * \mathrm{Ln}(\mathrm{Ln}(N)))^{1/2}$$

Even with this algorithm the time required to factor N, for large N, is excessive. A table of the time required to factor N for various sizes of N and assuming that one operation takes one microsecond is given in Fig.(2.3). From this table it can be seen that for N greater than 10^{100} calculating the prime factors is computationally infeasible. To protect against future increases in the speed and power of computers Rivest, Shamir and Adleman recommend choosing N to be $O(10^{200})$ i.e 200 digits. This would require approximately $3.8*10^9$ years to factor. This should provide a secure cipher for a long time regardless of progress in computer technology. Other methods of attack are possible (Rivest,Shamir,Adleman(4)) but these are all believed to be equivalent to factoring N and thus would require the same computational resources.

2.3 COMPARISON OF PUBLIC KEY CIPHERS

The knapsack method and the RSA method are the main contenders in the field of public key cryptography. The knapsack cipher has the advantage that encryption and decryption are fast as only modular multiplication and addition are required. The RSA method is more complex requiring modular exponentiation and this is time consuming. The public key size required for a knapsack cipher is approximately 20k bits whereas for the RSA the key memory required is about 1.2k bits. The RSA cipher does not expand the data from plaintext to ciphertext as the plaintext and ciphertext space are the same. This allows the RSA to be used to authenticate messages by forming digital signatures. The knapsack cipher does expand the data and thus it can be used for secrecy or authentication but not both. The ability of the user of an RSA system to choose the level of security is also an advantage. There has been doubts cast on the security of the knapsack cipher,(19)which undermines confidence in the method. The RSA method has, so far, remained secure, providing the keys are chosen properly. This confidence in the RSA method and the advantages that a public key cipher provide when compared to a conventional cipher has led to it being proposed as a national standard in the U.S.A (ZIMMERMANN(7)). The main difficulty with the RSA cipher is the complexity of the encryption and decryption process. Using dedicated hardware the DES cipher, a conventional cipher, has a throughput in the megabits/sec range. To be successful in a commercial environment the RSA method must attain a speed comparable to or at least the same order as the DES. The objective of this report is to investigate ways of implementing the RSA cipher.

2.4 MICROPROCESSOR IMPLEMENTATION OF RSA CIPHER

The first objective is to investigate the implementation of the RSA cipher on a microprocessor. This would be an inexpensive and flexible method of implementation that would have many commercial applications. If it is possible to produce a programme that will enable a reasonable throughput rate it could be included in many devices already utilising microprocessors providing a high level of security for a small cost.

The choice of an 8 bit microprocessor for use in implementing the RSA cipher is based mainly on power, cost and the availability of a development system. The microprocessor in common use in the N.I.H.E is the MC6809 from Motorola. This is considered one of the most powerful 8 bit microprocessors due to its flexible addressing methods and its use of a hardware multiplier. Development systems for this microprocessor were readily available in the college, consisting of VAX based assemblers and simulators, and real time emulators. The suitability of this processor and the time delay incurred in the purchase of development systems should an alternative be used, motivated the use of the MC6809 in the implementation of the RSA cipher. As all 8 bit microprocessors are basically similar the results obtained using this microprocessor should be immediately applicable to other microprocessors.

2.4.1 Modular Exponentiation

The basis of the RSA cipher is modular exponentiation. As the encryption and decryption algorithms are similar we need only consider one or the other. The objective therefore is to find ways of calculating C where

$$C = M^e \text{ Mod } N$$

where N,e,M are of the order of 200 decimal digits or 600 bits. This calculation must be performed on an 8 bit microprocessor.

To calculate M^e, M could simply be multiplied by itself e times. This requires e multiplications. However not all of the partial products are necessary. Consider the case when e is a power of two say e = 16. This would require 16 multiplications if the simple method above is used. The number of multiplications can be dramatically reduced by squaring each partial product i.e forming M, M^2, M^4, M^8, M^{16}. This has reduced the number of multiplications to 4 and in general only $\text{Log}_2(e)$ multiplications are required if e is a power of 2. This method can be expanded to include any arbitrary e by using the following algorithm, known as exponentiating by repeated squaring and multiplying. This method is an extension of the binary method described by KNUTH(8, pp.441-443) to numbers of arbitrary radix. This method is described below:

First Multiplication :

Let the result C be equal to M. This carries out the first multiplication so reduce e by one. A temporary variable T is needed to hold the partial product and this should be initialised to M also.

```
: If e is odd  : Let C = C * T and reduce e by one.
: If e is even : Let T = T^2 and divide e by two.
: Repeat until e is zero.
```

To demonstrate the method consider the example M^{37}.

```
First  C = M,          T   = M,          e = 36
Then e is even :  T = T^2   = M^2,       e = 18
     e is even :  T = T^2   = M^4,       e = 9
     e is odd  :  C = C*T = M.M^4,       e = 8
     e is even :  T = T^2   = M^8,       e = 4
     e is even :  T = T^2   = M^16,      e = 2
     e is even :  T = T^2   = M^32,      e = 1
     e is odd  :  C = C*T = M^5.M^32,e = 0
```

The number of multiplications necessary to calculate M^{37} is thus 7 which is a large reduction on the 37 required by the direct method.

To calculate the number of multiplications required for arbitrary e consider the binary representation of e, $(e_0 \ldots e_k)$. To test if e is odd or even it is only necessary to determine whether e_0, the least significant bit, is one or zero. A zero would imply that e is even, a one that it is odd. In the squaring and multiplying method described above if e is even we calculate T^2 and divide e by 2 whereas if e is odd we calculate C*T and decrease e by one. Reducing an odd number by one results in an even number. Therefore an odd number is equivalent to an even number followed by an extra multiplication (C*T) in the above method. The number of times that e can be divided by two is $\log_2(e)$. The total number of multiplications required is $\log_2(e)+D$ where D is the number of extra multiplications required because e is odd. Dividing by two, in binary, is equivalent to shifting right by one bit, thus the number of times that e can be odd is equal to the number of ones in the binary representation of e or the Hamming weight of e. Therefore the total number of multiplications is given by :

$$NM = \log_2(e) + w(e)$$

where w(e) is the Hamming weight of e. The worst case condition is when e contains all ones in which case w(e) = log2(e) and the number of multiplications required is :

$$NM_{max} = 2*\log_2(e)$$

A flowchart describing this exponentiation method is shown in Fig(2.4).

2.4.1.1 Results

A programme was written in MC6809 assembly language to implement the algorithm described in Fig.(2.4). While it may be possible to compress the programme further the times given below represent an approximate run time which would not be significantly altered by variations in programming techniques. The MC6809 is an 8 bit microprocessor therefore all calculations are performed on 8 bit bytes. The worst case time to encrypt a message M using a key (e,N) in which e and N can be represented by N_B bytes is(20) :

$$T_e = 53 + 2583(N_B) + 2432(N_B)^2 + 32(N_B)T_{Mod}[N_B] \text{ cycles}$$

where $T_{Mod}[N_B]$ is the number of cycles required to calculate the product of two N_B byte numbers modulo a third N_B byte number.
A cycle is one microsecond on the standard 1MHz microprocessor. Various methods for performing modular multiplication and the resulting overall encryption will now be discussed.

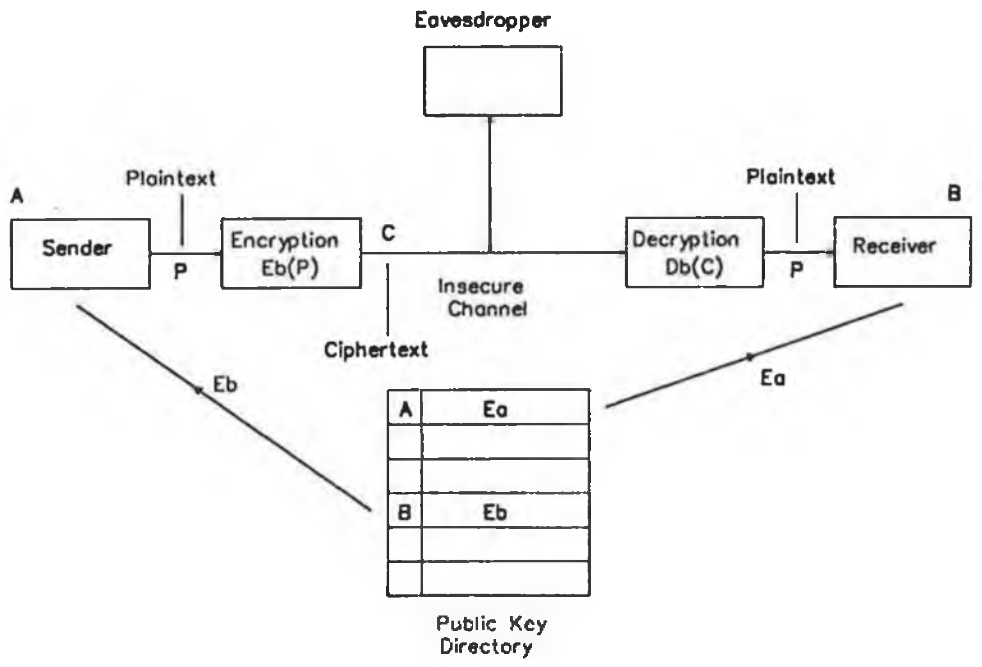


Fig. 2.1 Public Key Cipher

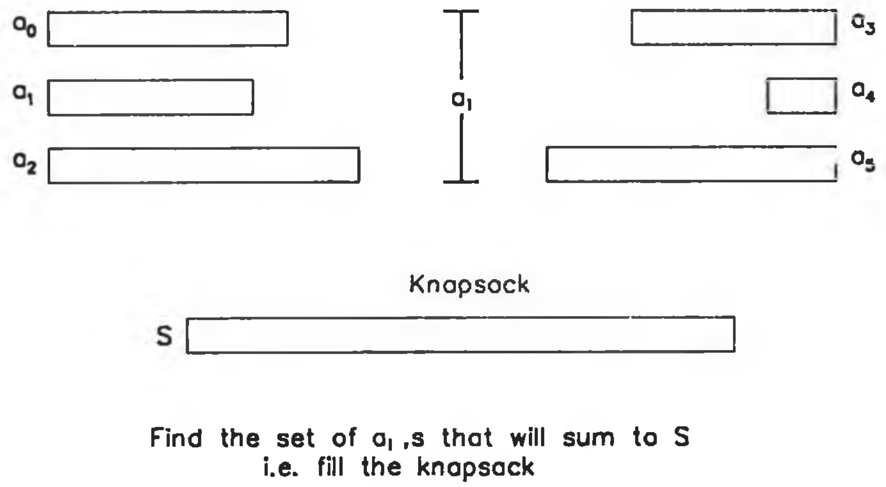


Fig. 2.2 The Knapsack problem

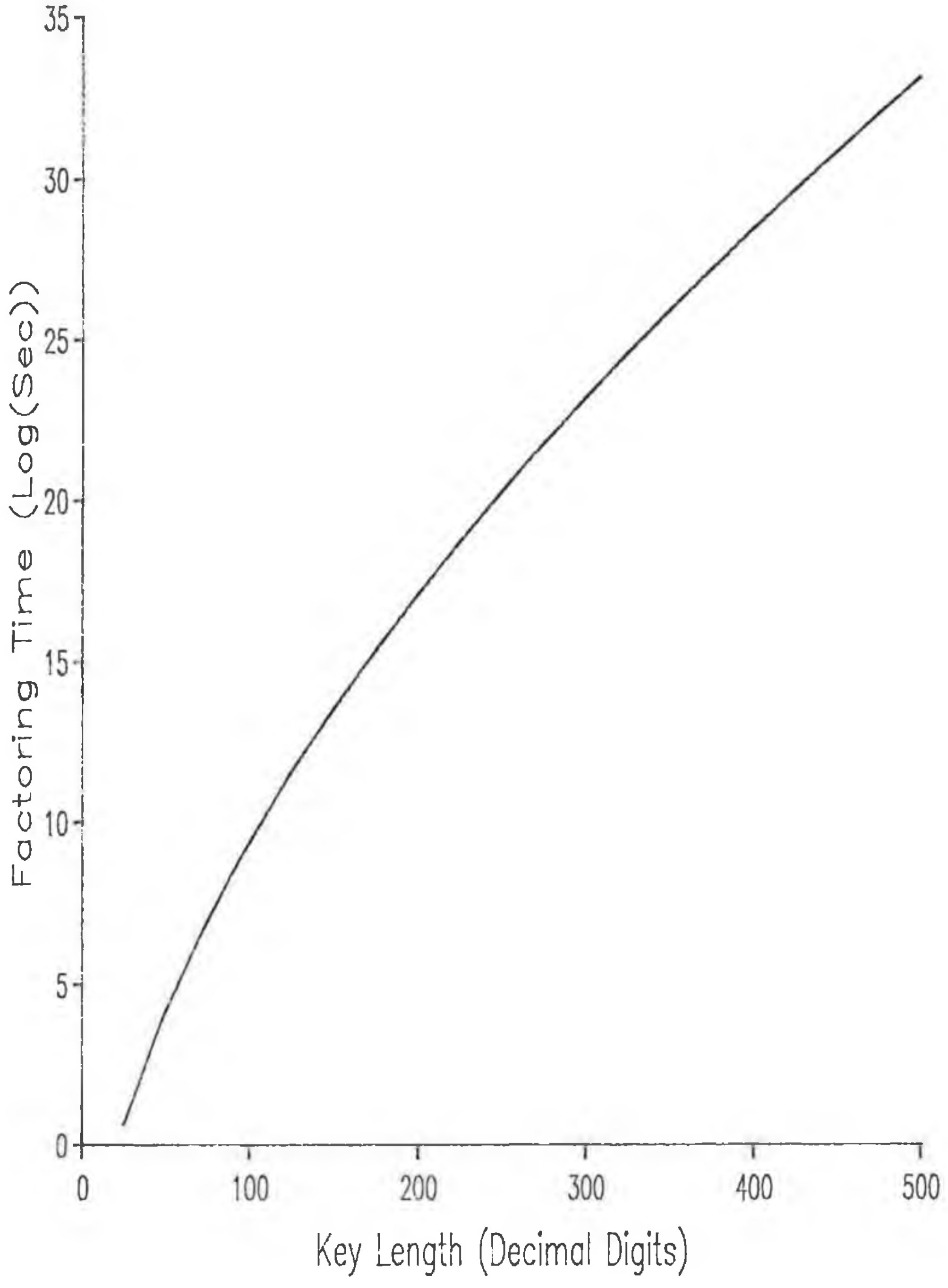
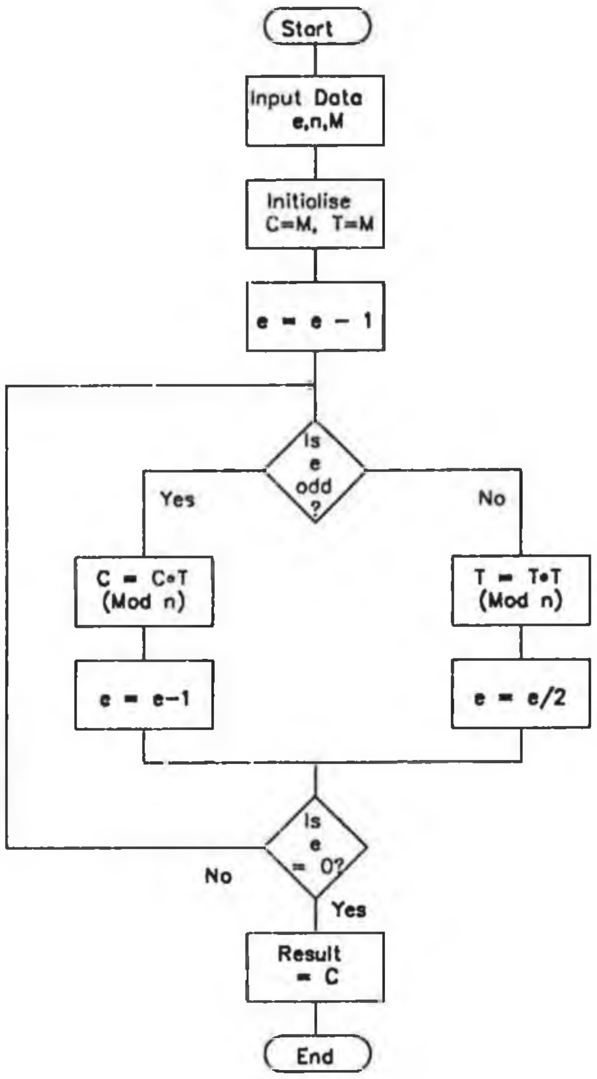


Fig (23) Factoring time for various key lengths



$$C = M^e \text{ (Mod } n)$$

Variables :

e:	Exponent	T:	Temporary variable
n:	Modulus	C:	Result
M:	Operand		

Fig. 2.4 Modular Exponentiation

3. Modular Multiplication Methods

3.1 INTRODUCTION

Modular exponentiation requires that modular multiplications be performed. In the RSA public key cipher it is necessary to calculate the product AB (mod N) where A, B, N are positive integers of the order of 100 decimal digits or 330 bits. Calculations on numbers of this size cannot be performed directly on a standard microprocessor whose word size is much smaller than this, typically 8 or 16 bits. It is thus necessary to consider such calculations as multiple precision calculations. It is convenient to use the word size of the computer as the radix for all calculations thus a 336 bit number becomes a 42 digit number on an 8 bit microprocessor such as the MC6809. It is considered that the multiplication of two single digit numbers to form a double precision result is a primitive operation of the microprocessor.

To form the product C = AB (Mod N) two approaches are possible. The first forms the product C = AB and then reduces this modulo N to complete the calculation. This entails the use of memory to store the double length result prior to its reduction. The second method achieves the reduction modulo N simultaneously with the formation of the product AB. This removes the need to store a double length result as the product is a single length number, however such methods are generally slower than the former methods.

The reduction of an m digit number C modulo N where N is an N_B digit number is the remainder when C is divided by N and is therefore less than N. Multiple precision division and subtraction are thus needed for reduction. The process of performing modular multiplication is one of multiple precision multiplication of two single length numbers to provide a double length result followed by multiple precision division and subtraction to reduce this number modulo N.

3.2 MULTIPLICATION

The conventional 'pen and paper' method for forming multiple precision products is known as the sum of products method. In this method the first operand is multiplied by each digit of the second operand forming a number of partial products. These partial products are then shifted and added to obtain the result. As an example consider multiplying 814639 by 462115 :

```
Multiplicand:       814639
Multiplier:       x 462115
                   4073195
                  814639
                 814639
                ------

Result:      376456301485
```

When implementing this method on a computer it is more efficient in both time and memory to add the partial products as they are produced instead of storing each result and adding when all partial products are obtained.

In the general case for numbers of arbitrary radix consider the product of $A = (a_0, \ldots a_{t-1})_r$ and $B = (b_0, \ldots b_{s-1})_r$ where the a_i's and b_i's are digits in radix r. The radix r is usually chosen as the word length of the computer being used e.g $r = 256$ for an 8 bit microprocessor such as the MC6809. The product of A and B is therefore :

$$A*B = \sum_{i=0}^{t-1} (a_i r^i) \quad \sum_{j=0}^{s-1} (b_j r^j)$$

This can be written as :

$$A*B = \sum_{i=0}^{t-1} \quad P_i + r^i \sum_{j=0}^{s-1} (a_i b_j r^j)$$

where P_i is the partial product. It can be seen from this that $s*t$ single precision multiplications are required along with multiple precision additions.

This algorithm was implemented in assembly language on the MC6809 microprocessor and a flowchart describing the programme is shown in Fig.(3.1a). The digits of the multiple digit numbers A and B were stored in successive memory locations. A pointer to the operands and the number of digits contained in each was also kept. This facilitates the addressing and manipulation of the operands. The time taken to calculate the product is given below.

$$\begin{aligned} T_{mult} &= 62 + 11(s+t) + s(\ 45 + \ 49t\ ) \text{ cycles} \\ &= 62 + 56s + 11t + 49st \text{ cycles} \end{aligned}$$

In the RSA public key cipher s and t will tend to be the same as all multiplication is done modulo N, thus the maximum value for either operand is N. If N_B is the number of digits in N then and if $s = t = N_B$ then the multiplication time is :

$$T_{mult}[N_B] = 62 + 67(N_B) + 49(N_B)^2 \text{ cycles}$$

The time taken to multiply numbers of differing precision is shown in Fig.(3.1b). In this case a clock cycle is taken as being equal to 1 microsecond as in the standard MC6809. The area of particular interest, to the RSA cipher, in this figure is the multiplication of numbers in the 40 to 80 byte range as this provides a significant level of security. It can be seen that a multiplication of this size takes from 80 to 320 milliseconds.

3.3 REDUCTION MODULO N

To represent a number C in modular form it is necessary to find an integer R such that :

$$C \equiv R \pmod N$$

Therefore C = Q.N + R where Q is some integer. This is equivalent to dividing C by N to obtain a quotient Q and a remainder. Reduction modulo N is thus very similar to division except that in this case it is the remainder and not the quotient that is of interest. It is therefore necessary to consider ways to implement multiple precision division. Three methods are considered :

1. Division by repeated shift and subtract.
2. Division using KNUTHs algorithm.
3. The calculation of reciprocals and its use in division.

3.3.1 Division by repeated shift and subtract

The shift and subtract method of division is similar to the familiar 'pen and paper' division in common use. In this the quotient is obtained one digit at a time and the divisor is subtracted from the divisor at decreasing levels of significance until the calculation is complete. Two types of division using this method can be defined, restoring or non-restoring division. In restoring division, if a trial subtraction is unsuccessful i.e the partial remainder is less than zero the partial remainder is restored to its previous value, the divisor is shifted right and another trial subtraction is made. In non-restoring division, however, if a trial subtraction is unsuccessful the partial remainder is not restored instead the next trial becomes an addition. This has the advantage that it is not necessary to retain an unmodified version of the partial remainder or to perform an additional addition to restore the remainder. Non-restoring division can easily be implemented when twos complement numbers are being used, however, when dealing with unsigned integers the restoring method is preferred. This can be described as shown below:

Let the dividend C be $(c_{m-1}c_{m-2}\ldots c_0)_2$ and let the divisor N be $(d_{n-1}d_{n-2}\ldots d_0)_2$ where c_i and d_i are binary digits then the algorithm is :

1. From the divisor N form an m bit number by adding m-n trailing zeros.

$$N = (d_{n-1}d_{n-2}\cdot\cdot d_0 0_{m-n}0_{m-n-1}\cdot\cdot\cdot 0_1)_2$$

2. Subtract the divisor N from the dividend C to form the partial remainder.

   $$C = C - N$$

   If the partial remainder is negative the quotient bit is zero otherwise it is one. As the quotient is not required in this application it is not necessary to store it.

3. If the partial remainder is negative then restore it by adding N.

   $$C = C + N$$

4. The divisor is now shifted one bit to the right and steps (2) to (4) are repeated.

5. The algorithm ends when the divisor has been shifted m-n times.

As an example of this method consider dividing $(11001001)_2$ by $(111)_2$.

```
          __________
111      | 11001001
           111
          1111         : remainder is negative => Q5=0

           1100        : restore partial remainder
            111
            101        : remainder is positive => Q4=1

            1011
             111
             100       : remainder is positive => Q3=1

             1000
              111
                1      : remainder is positive => Q2=1

                10
               111
              1011     : remainder is negative => Q1=0
```

```
  101    : restore partial remainder
  111
 1110    : remainder is negative => Q0=0

   101   : restore partial remainder
```

Therefore $(11001001)_2/(111)_2 = (11100)_2$ plus a remainder of $(101)_2$ or :

$$(11001001)_2 \equiv (101)_2 \pmod{(111)_2}$$

This algorithm was implemented in assembly language on an MC6809 microprocessor. A flow chart of this algorithm is given in Fig.(3.2a). The run time of the algorithm was :

$$T_{D1} = 638 + 486(N_C-N_B) + 208(N_C-N_B)^2 + 8(N_C-N_B+1)[105 + 123(N_C-N_B)]$$

$$= 1478 + 2310(N_C-N_B) + 1192(N_C-N_B)^2$$

where N_C is the number of bytes in the dividend C and N_B is the number of bytes in the divisor or modulus N. The dividend will in the case of the RSA cipher contain a maximum of twice as many digits as the divisor (it is the product of two N_B digit numbers) thus $N_C = 2N_B$ therefore,

$$T_{D1} = 1478 + 2310(N_B) + 1192(N_B)^2$$

A table and graph showing the variation of the division time T_{D1} for differing values of N_B is shown in Fig.(3.2b). As in the case of multiplication we are particularly interested in the range from 40 digits to 80 digits and it can be seen that reduction of an integer of this size takes from 2 to 8 seconds. It is again noted that one clock cycle is of 1 microsecond duration.

3.3.2 Division using KNUTHs algorithm

The disadvantage of the previous method is that it does not utilise the power of the microprocessor efficiently. The algorithm deals with bits at a time whereas the microprocessor can more readily deal with bytes. An algorithm which can be used with computers of arbitrary radix is KNUTHs(8) algorithm. The purpose of this algorithm is to approximate the 'pen and paper' method of division. Consider the case of an N_C digit dividend $C = (c_{Nc-1} \ldots c_0)_r$ divided by an N_B digit divisor $N = (d_{NB-1} \ldots d_0)_r$. The first step is to divide the divisor into the most significant (N_B+1) digits of the dividend to obtain a quotient digit q and a partial remainder P.

$$\begin{array}{r|c} & q \\ \cline{2-2} (d_{NB-1} \ldots d_0)_r & (c_{Nc-1} \ldots c_{Nc-NB-1}) \\ & qD \\ \cline{2-2} & P \end{array}$$

P is always less than N so the next step uses $P*r + c_{Nc-NB-2}$, where r is the radix, as the dividend. The result of this is to shift P left by one digit and to add the next most significant digit of the dividend to it. This is similar to the shift in the previous shift and subtract method.

The difficulty arises in finding the quotient digit q. This can not be obtained directly as it involves a multiple precision division. An approximation to q, call it q_a, can be obtained from the leading digits of the divisor and the dividend. Thus

$$q_a = \text{int}\left[\frac{c_{Nc-1}b + c_{Nc-2}}{d_{NB-1}}\right]$$

where int denotes the integer part of. If q_a is larger than r it can be replaced by (r-1), the largest single digit number. The accuracy of this approximation to q will determine how useful the method is. KNUTH(x pp. 255-260) shows that if the dividend is first normalised so that its most significant digit is greater than half the radix then

q_a will always be greater than q but never more than two greater i.e

$$q \leqslant q_a \leqslant q+2 \qquad (\text{ if } c_{Nc-1} > \text{int}(r/2))$$

The trial quotient can thus be obtained by dividing a double digit number by a single digit number to obtain a single digit approximation q_a. Many microprocessors include such an instruction in their instruction set but unfortunately the MC6809 does not. This is not a great disadvantage as such a division can be carried out rapidly using the shift and subtract method described previously. When q_a is calculated it is multiplied by the divisor N and subtracted from the dividend.
The complete algorithm is thus :

1. Normalise: Ensure that the most significant digit of the divisor is greater than half the radix. Multiply the dividend by the same factor. This can be accomplished by:

   $$N = \text{int}\left[r/(d_{NB-1}+1)\right] *N, \quad C = \text{int}\left[r/(d_{NB-1}+1)\right] *C$$

   Note that although the normalisation will not affect the quotient it will affect the remainder and as it is the remainder which is important in this application the result will have to be denormalised.

2. Initialise : set a pointer $i=N_C-1$, the most significant digit of the dividend.

3. Calculate trial quotient q_a . This is obtained from the most significant digits of the dividend and the divisor.

   $$q_a = \text{int}\left[\frac{c_i b + c_{i-1}}{d_{NB-1}}\right]$$

4. Test q_a. Determine if q_a is too large. The following test is a quick method of determining if q_a is too large. It will determine all cases when q_a is two

greater than q and most of the cases when q_a is one greater than q.

if $q_a d_{NB-2} > (c_i b + c_{i-1}) - q_a d_{NB-1}\, r + c_{i-3}$

then

$$q_a = q_a - 1$$

5. Calculate partial remainder. Multiply q_a by N and subtract from most significant digits of C.

$$P_i = (c_i \ldots c_{i-NB}) - q_a N$$

The partial remainder can be stored in the most significant digits of C i.e.

$$(c_i \ldots c_{i-NB}) = P_i$$

6. Restore. If q_a is one greater than q and is not detected by step (4) P_i will be negative. Reduce q_a by one and restore the partial remainder by adding N to it.

$$P_i = P_i + N$$

7. Decrement pointer i. Repeat steps (3) to (7) until $i = N_B$

8. Denormalise the remainder to obtain the required result. The remainder is stored in the N_B least significant digits of C.

A flow chart describing this algorithm is shown in Fig.(3.3a). The algorithm was implemented on the MC6809 microprocessor and the run time was as follows:

$$T_{D2} = 827 + 561N_B + (N_C - N_B + 1)\left[701 + 103(N_C - N_B)\right]$$

Again as in the previous case if $N_C = 2N_B$ the total worst case run time is:

$$T_{D2} = 1528 + 1365(N_B) + 103(N_B)^2 \text{ cycles}$$

A table and graph showing the division time for varying block lengths is shown in Fig.(3.3b). The time taken to divide numbers in the 40 to 80 digit range is from 200ms to 800ms.

3.3.3 Division using reciprocals.

If the reciprocal of a number can be found then a quotient can be formed by taking the reciprocal of the divisor and multiplying the result by the dividend.

$$\text{quotient} = \frac{\text{Dividend}}{\text{Divisor}} = \left[\text{Dividend} * \frac{1}{\text{Divisor}} \right]$$

This method is especially useful when the divisor is common to many multiplications. In this case the divisor can be precalculated thus saving time. In the RSA method the divisor which is the modulus N will remain constant throughout the exponentiation process which will require many modular multiplications and thus divisions by N. Two methods of obtaining an approximation to the reciprocal of a number were considered. These are :

1. Divide 2^K by the number N, whose reciprocal is required, to get a K place approximation to 1/N. The division in this case can be carried out by any means available such as the shift and subtract method described previously or KNUTHs algorithm. The binary point is assumed to be to the left of the most significant bit.

2. Newtons Method.

   This requires some initial approximation X_0 to the reciprocal 1/N and then to use the iterative algorithm:

$$X_{j+1} = 2X_j - DX_j^2$$

This algorithm converges rapidly, at a quadratic rate, to 1/N if the initial approximation to 1/N was 'good'. If X_0 is not sufficiently close to 1/N the method may converge only slowly, if at all.

The speed of the algorithm chosen to calculate the reciprocal is not a limiting factor as it will only be calculated once per exponentiation. The first method was chosen using the shift and subtract division algorithm as it avoids the problem of convergence and it had already been coded.

The implementation of the shift and subtract reciprocal algorithm on the MC6809 is essentially the same as that of the division programme with the exception that in this case it is the quotient and not the remainder that is of interest and this must be stored. The flowchart for this algorithm is thus the same as for the division programme and can be seen in Fig.(3.2a). The time required to calculate the reciprocal is:

$$T_R = 1447 + 174(N_C) + 424(N_C)^2 \text{ cycles.}$$

where N_C is the number of bytes in the dividend. Having calculated the reciprocal it is necessary to find the quotient and the remainder. The product $C*N^{-1}$ needs to be calculated. As C is typically $2N_B$ bytes long and N^{-1} is N_B bytes long this would involve a long multiplication yielding a result of $3(N_B)$ bytes in length. This level of accuracy is not required, only integer accuracy is necessary, thus the multiplication can be considered as the multiplication of two (N_B) byte numbers. This product should then be multiplied by the divisor N and subtracted from C to obtain the remainder. Two multiple precision multiplications are performed.

This algorithm, a flow chart of which is shown in Fig.(3.4a), was implemented on the MC6809 microprocessor. The run time of the algorithm is :

$$T_{D3} = 240 + 101(N_B) + 2(T_{Mult}[N_B]) \text{ cycles.}$$

where N_B is the number of bytes in the divisor, N, and $T_{Mult}[N_B]$ is the time taken to multiply two N_B byte numbers to form a $2(N_B)$ byte product using the algorithm described previously. A table showing the run time for various lengths of operands is shown in Fig.(3.4b).

3.4 BLAKELYS ALGORITHM.

The methods discussed so far have performed the modular multiplication in two steps. The first step multiplied the two operands together forming a double length product. This product was then reduced by dividing it by the modulus and keeping the remainder. It is possible to perform the modular multiplication directly and an algorithm to do this was described by BLAKELY(9).

Blakelys algorithm is similar to the conventional bitwise integer multiplication algorithm. In this method the following steps are carried out:

1. Initialise the result to zero.

2. Test the most significant bit of the first operand. If it is one add the second operand to the result.

3. Shift the result one bit to the left.

4. Repeat for successively less significant bits of the first operand. Stop when all the bits have been tested.

The modular multiplication algorithm developed by Blakely modifies this basic algorithm to provide modular multiplication. To do this it is necessary to test the result after every addition or left shift. If the result is greater than the required modulus subtract the modulus from the result and continue. The reduction is thus performed along with the multiplication. The algorithm is therefore:

1. Initialise the result to zero.

2. Test the most significant bit of the first operand. If it is one add the second operand to the result.

3. If the result is greater than the modulus N subtract N from the result.

4. Shift the result to the left by one bit.

5. If the result is greater than N, subtract N from the result.

6. Repeat steps 2 to 6 for successively less significant bits of the first operand. Stop when all the bits have been tested.

A flowchart of this algorithm appears in Fig.(3.5a). A programme utilising this algorithm was written for the MC6809 and the run time is given as follows:

$$T_B = 66 + 197(N_B) + 768(N_B)^2 \text{ cycles.}$$

where N_B is the number of bytes in the modulus. A table showing the run time of this programme for various values of N_B is given in Fig.(3.5b).

3.5 ENCRYPTION TIME.

Various methods for performing modular multiplication have been described. It is now necessary to determine overall encryption times using each of these methods. The encryption algorithm is of the form :

$$C = M^e \text{ Mod } N$$

C is the required ciphertext, M is the plaintext, e is the encryption key and N is the modulus. The length of C,M,e,N will be N_B bytes where N_B is of the order of 40 bytes or more. The exponentiation algorithm has a run time of :

$$T_e = 53 + 2583(N_B) + 2432(N_B)^2 + 32(N_B)T_{Mod}[N_B] \text{ cycles}$$

where $T_{Mod}[N_B]$ is the time required to form the product of two N_B byte numbers modulo a third N_B byte number. Four methods of calculating such a product have been discussed:

A. Methods which form the product and then reduce.

These all multiply using the same algorithm but reduce modulo N in different ways.

1. Reduction using division by repeated shifting and subtracting.

2. Reduction using division by KNUTHs algorithm.

3. Reduction using reciprocals.

B. Methods which reduce as the product is being calculated.

4. BLAKELYS Algorithm

For the first three methods the time required for a modular multiplication is the combination of the time required for a conventional multiplication plus the time for the reduction plus any overheads that are necessary. For the methods described above the total time for a modular multiplication is:

1. Shift and subtract

Overheads	: 96
$T_{mult}[N_B]$	: $62 + 67(N_B) + 49(N_B)^2$
T_{D1}	: $2462 + 1326(N_B) + 1192(N_B)^2$

$$T_{mod1} = 2620 + 1393(N_B) + 1241(N_B)^2$$

2. Reciprocal

   Overheads : 96

   $T_{mult}[N_B]$: $62 + 67(N_B) + 49(N_B)^2$

   T_{D2} : $1528 + 1365(N_B) + 103(N_B)^2$

   $T_{mod2} = 1686 + 1432(N_B) + 152(N_B)^2$

3. KNUTHs algorithm

   Overheads : 96

   $T_{mult}[N_B]$: $62 + 67(N_B) + 49(N_B)^2$

   T_{D3} : $240 + 101(N_B) + 2*T_{mult}[N_B]$

   $T_{mod3} = 522 + 302(N_B) + 147(N_B)2$

4. BLAKELYS algorithm.

   T_{mod4} $(=T_B) = 66 + 197(N_B) + 768(N_B)^2$

Using these results in the equation for exponentiation will give the overall encryption time for each of the methods outlined. The results of this are shown in Fig.(3.6).

3.6 OBSERVATIONS

With reference to the table in Fig.(3.6) it can be seen that the fastest way of encrypting is to use the reciprocal method when performing modular multiplications. This is followed closely by Knuths algorithm and then by the shift and subtract and Blakelys algorithm. In their original paper Rivest, Shamir and Adleman recommend using key lengths of 200 decimal digits or approximately 80 bytes. From the table it can be seen that the time taken to encrypt a message of this length is 40 minutes. This implies a bit rate of 1/4 bps. If keys of lengths of 100 decimal digits or 40 bytes are used the encryption time reduces to 321 seconds or a bit rate of about 1 bps. This may be of use in a limited number of applications but as a general purpose encryption device it is of no value.

3.7 POSSIBLE SOLUTIONS.

There are three possible solutions to the speed problem of the microprocessor based encryption unit which will be considered.

(1) Reduce the key length even further.

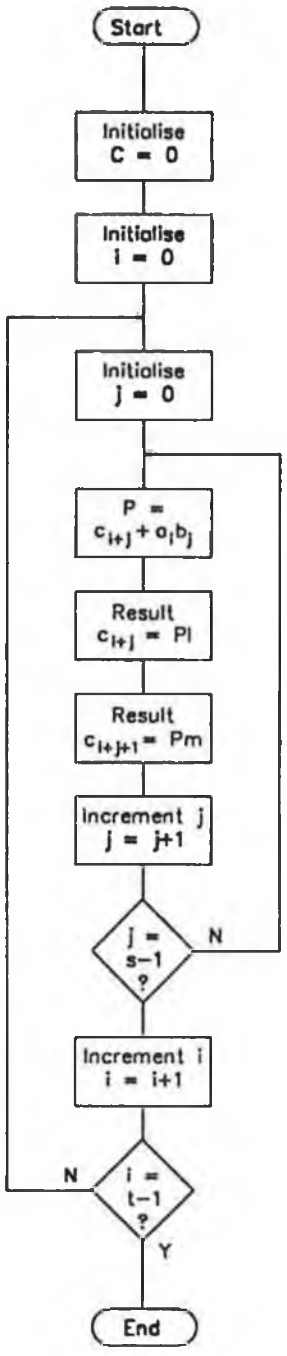
This may be possible in certain instances but as discussed earlier the security of the RSA method relies heavily on long key lengths. For medium to high security applications the key cannot be reduced below 100 decimal digits without compromising the security of the system.

(2) Find a more efficient algorithm.

More efficient algorithms have been suggested most of which have been based on the Chinese Remainder Theorem. These methods e.g (13), will decrease the encryption time by a factor of four at best. These algorithm could be implemented on a fast 16 or 32 microprocessor thus increasing the speed further. However the size of the necessary reduction in encryption time, a bit rate of 1000 bps at least is required, suggests that a software solution on its own is not possible.

(3) Implementing the system in hardware.

Several options are available under this heading. The addition of hardware multipliers and/or accumulators to the existing microprocessor is one alternative. The implementation of the complete system in hardware is another alternative. This approach offers the best possibility of considerably reducing the encryption time of the system. The parallelism inherent in the enciphering algorithm can best be taken advantage of in hardware. The use of semicustom or full custom integrated circuits could also reduce the physical size of the system. This approach will be dealt with in more detail later.



$A = (a_{t-1} \ldots a_0)$

$B = (b_{s-1} \ldots b_0)$

$C = (c_{s+t-1} \ldots c_0)$

$P = (Pm, Pl)$

Variables :

A	:	Multiplier
B	:	Multiplicand
C	:	Result
P	:	Partial result
t	:	No. of bytes in A
s	:	No. of bytes in B
Pm	:	MS byte of P
Pl	:	LS byte of P
i,j	:	Pointers

Fig. (3.1a) Multiple precision binary multiplication

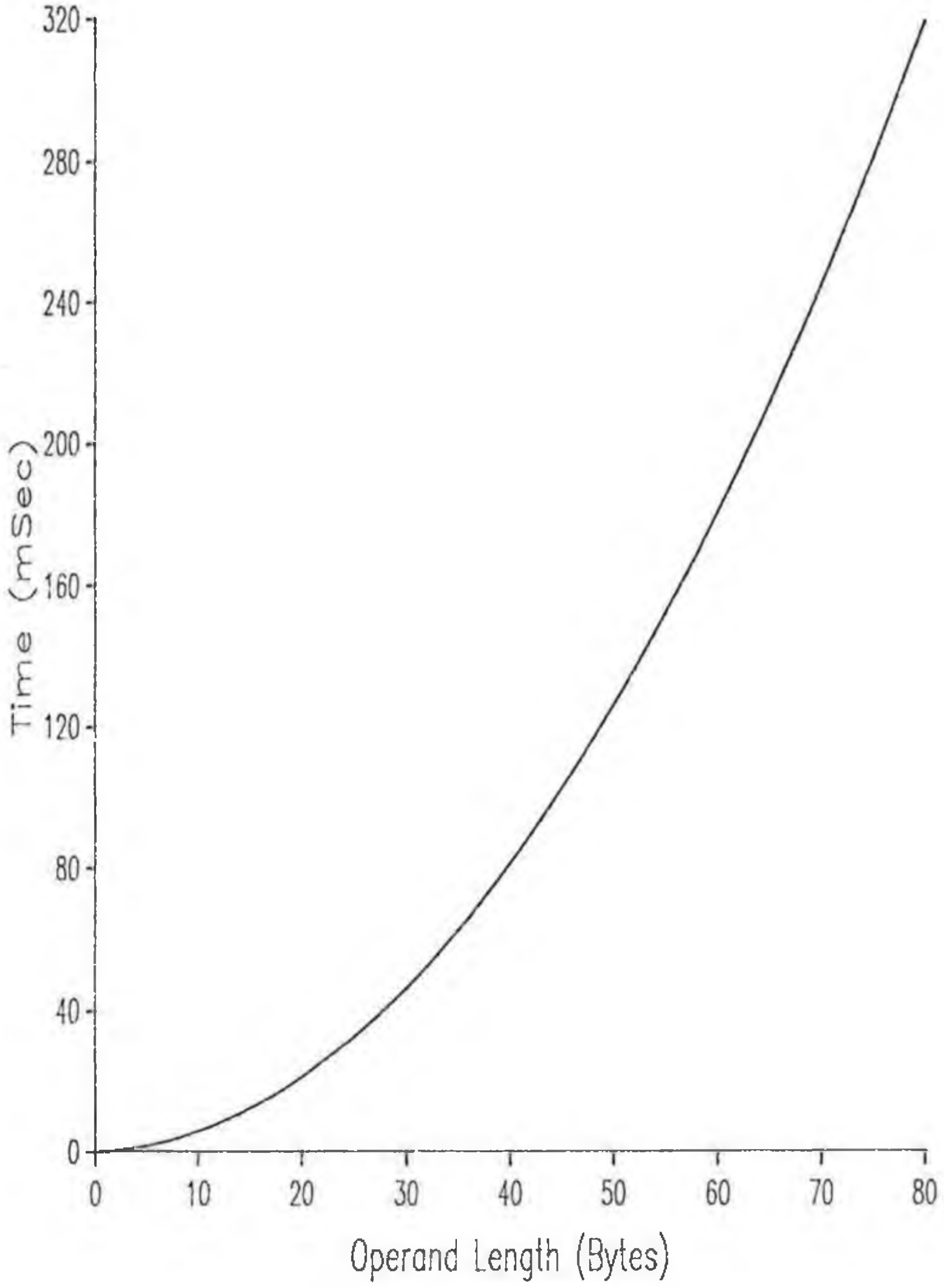
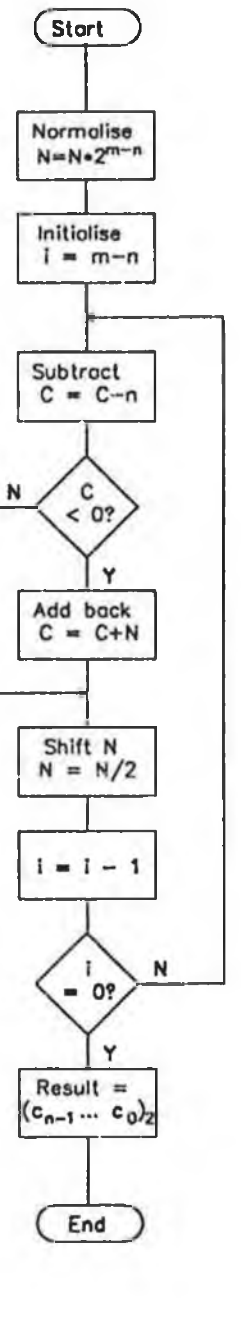


Fig (3.1b) Multiple percision binary multiplication - runtime

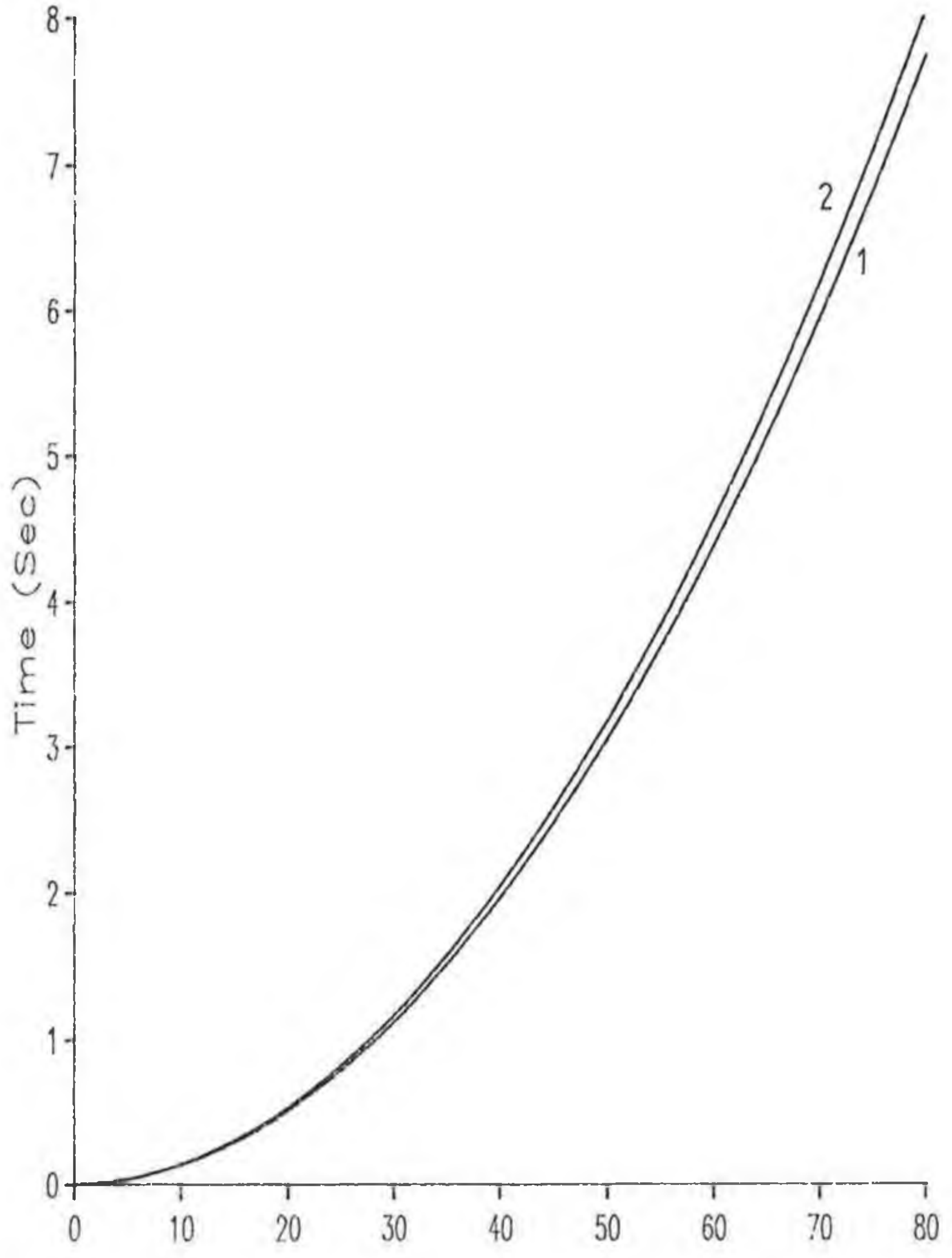


Variables :

C : Dividend = $(c_{m-1} \ldots c_0)_2$
N : Divisor = $(d_{n-1} \ldots d_0)_2$
m : No. of bits in dividend C
n : No. of bits in divisor N
i : pointer

C = C Mod N

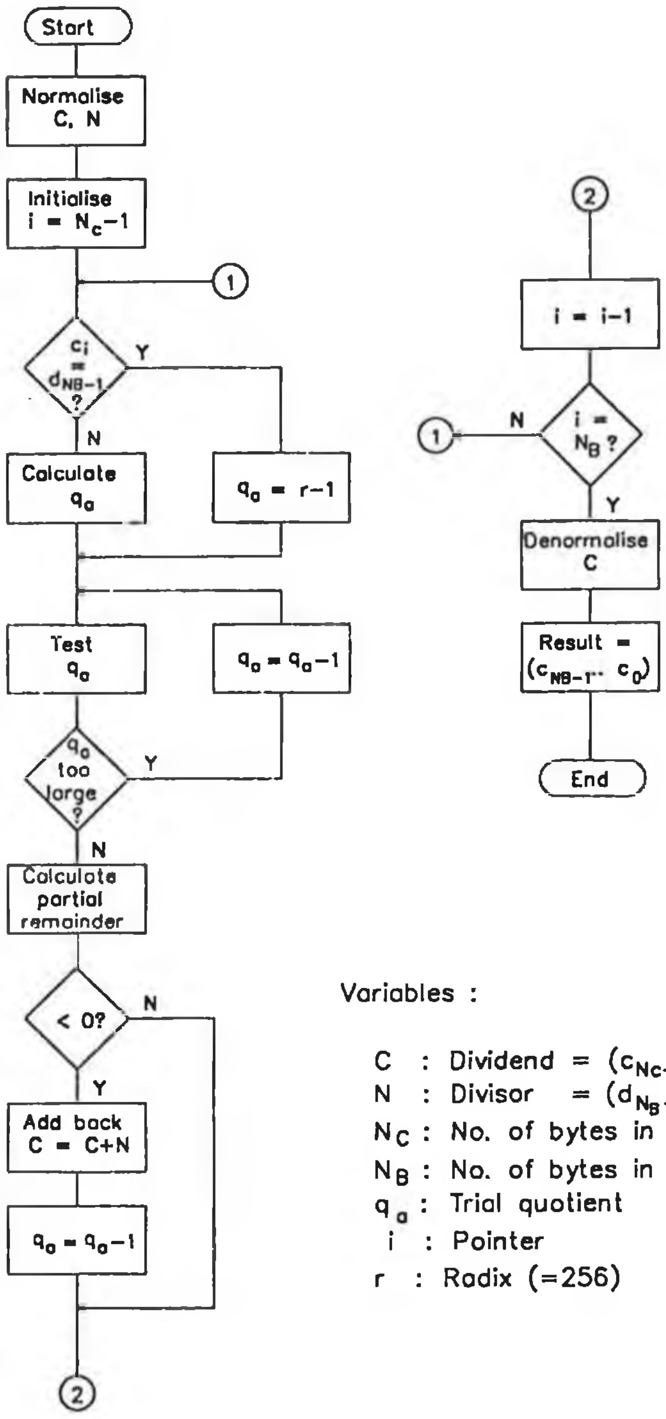
Fig. (3.2a) Shift and subtract algorithm



1: Division using shift and subtract

2: Modular multiplication
(multiplication + division)

Fig. (3.2b) Shift and subtract algorithm
- runtime

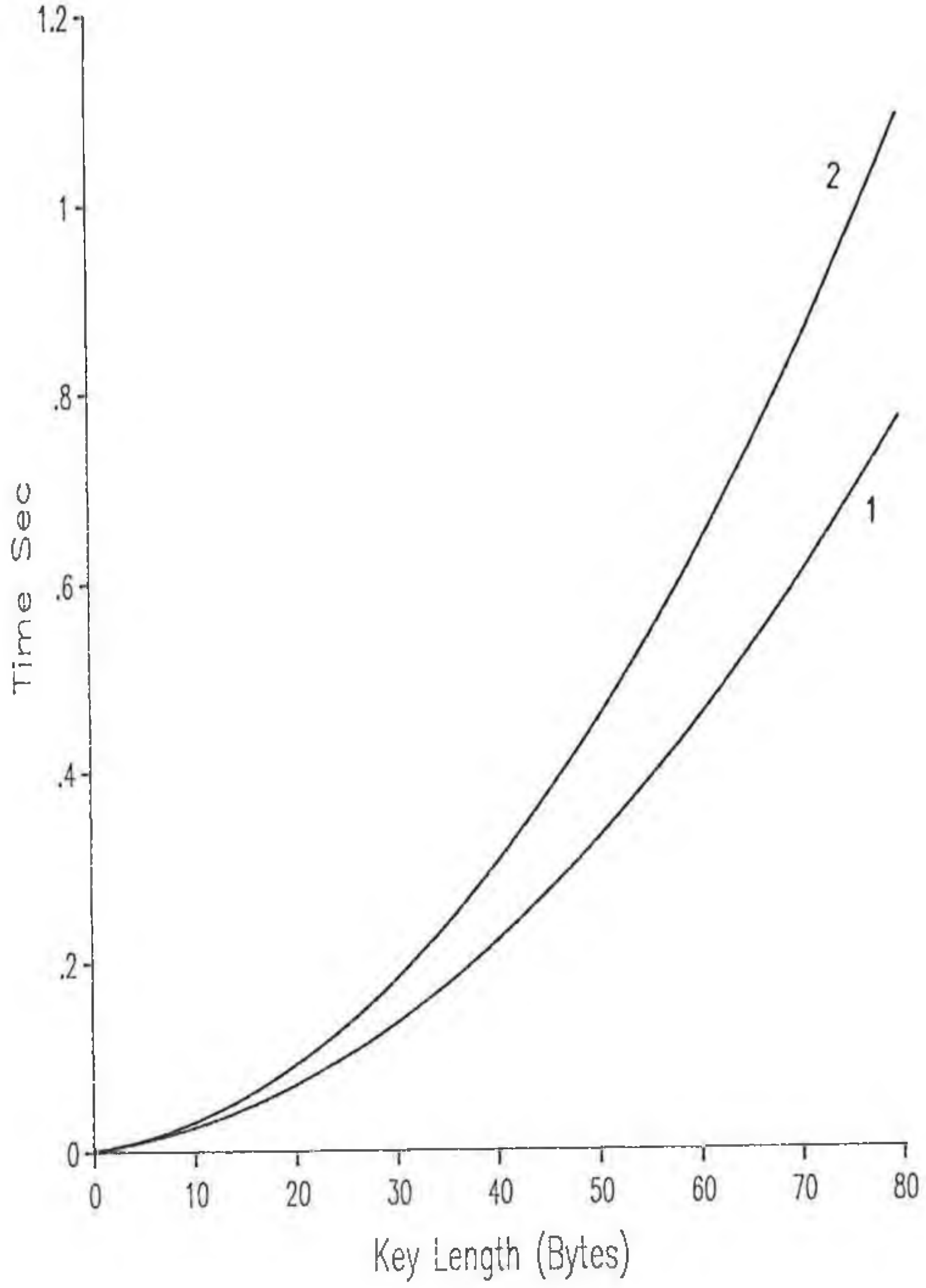


Variables :

C	:	Dividend = $(c_{Nc-1} .. c_0)$
N	:	Divisor = $(d_{N_B-1} .. d_0)$
N_C	:	No. of bytes in C
N_B	:	No. of bytes in N
q_a	:	Trial quotient
i	:	Pointer
r	:	Radix (=256)

C = C Mod N

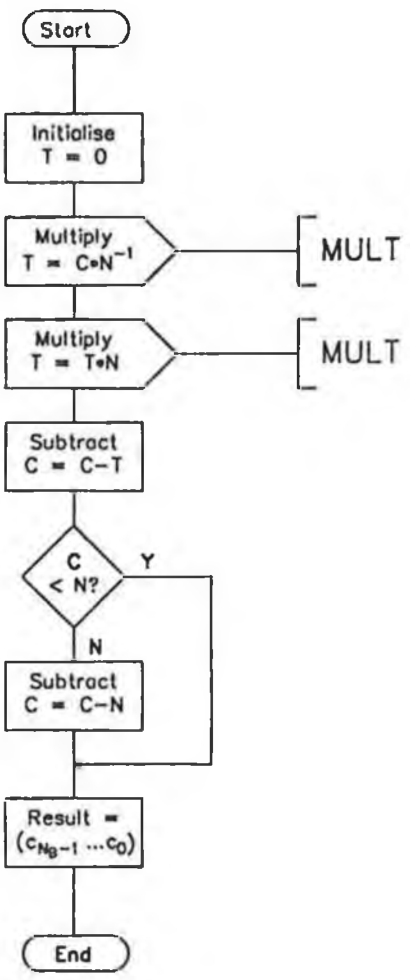
Fig. (3.3a) Knuths division algorithm



1: Reduction (Division

2: Modular Multiplication

Fig. (3.3b) Runtime of Knuths algorithm

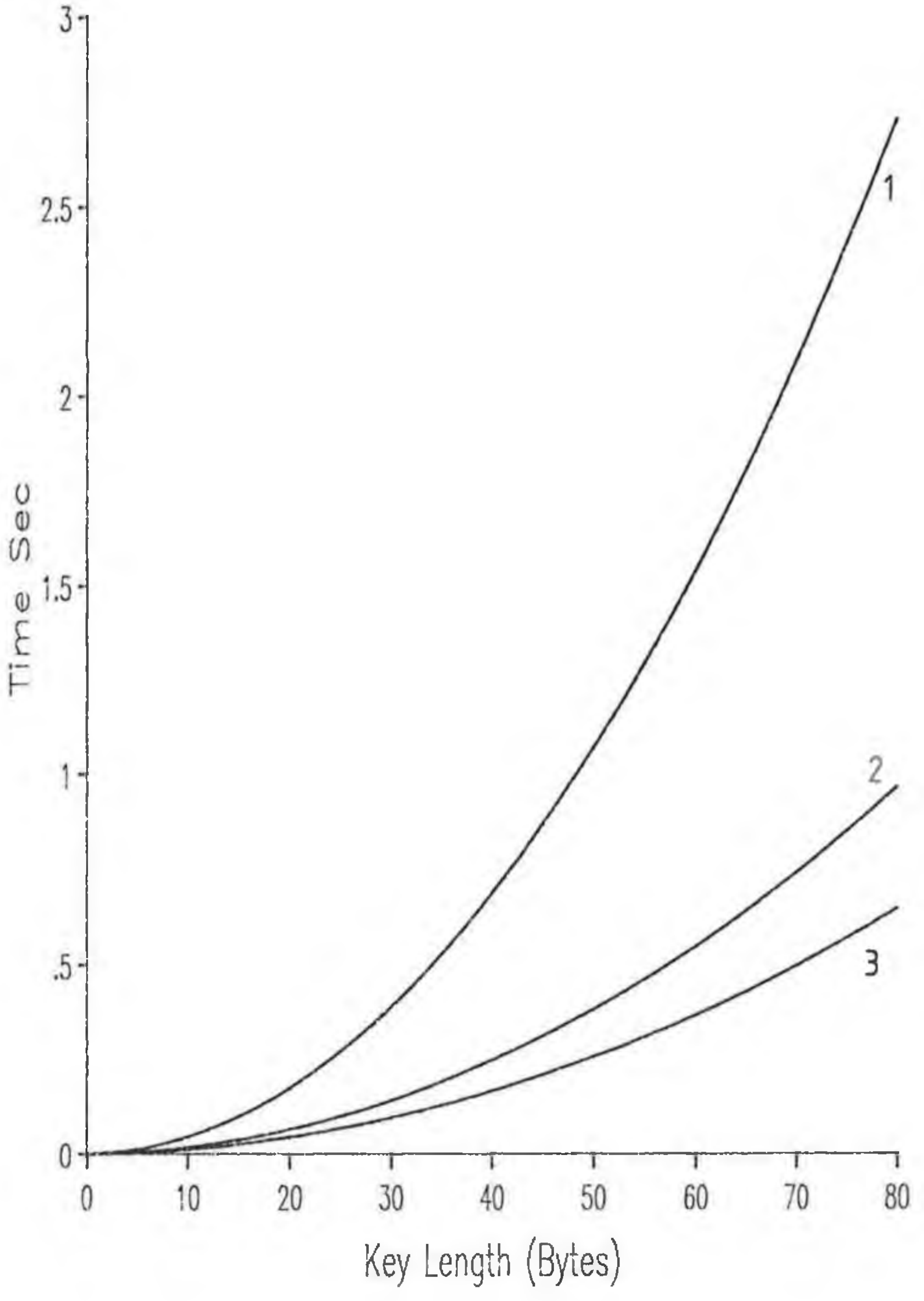


$$C = C \text{ Mod } N = C - \text{int}\left[\frac{C}{N}\right] * N$$

Variables :

N : Divisor = $(d_{N_B-1} \, ..d_0)$	N_B : No. of bytes in divisor
C : Dividend = $(c_{N_C-1} \, ..c_0)$	N_C : No. of bytes in dividend
N^{-1} : Reciprocal of divisor	T : Temporary variable

Fig. (3.4a) Division by reciprocals

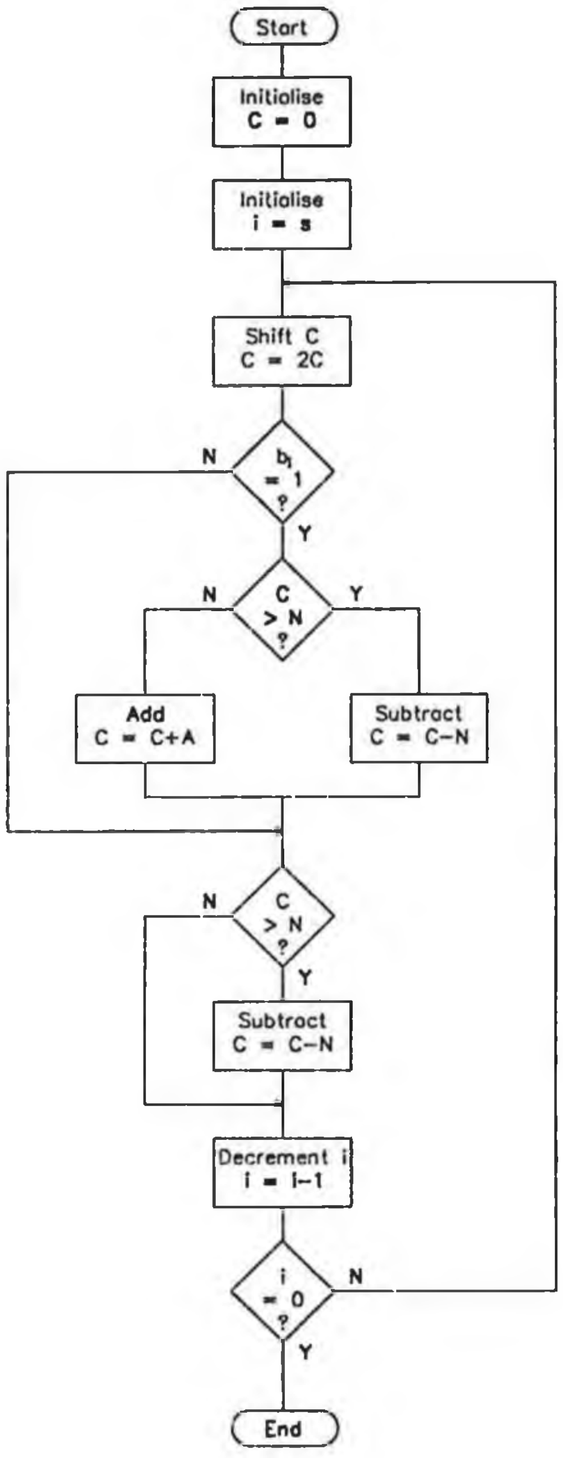


1: Calculation of reciprocal

2: Multiplication + Division

3: Division

Fig. (3.4b) Division by reciprocals



$A = (a_{t-1} \ldots a_0)$

$B = (b_{s-1} \ldots b_0)$

$C = (c_{s+t-1} \ldots c_0)$

A	:	Operand 1
B	:	Operand 2
C	:	Result
t	:	No. of bits in A
s	:	No. of bits in B
i	:	Pointer

$$C = AB \text{ Mod } N$$

Fig.(3.5a) Blakelys algorithm

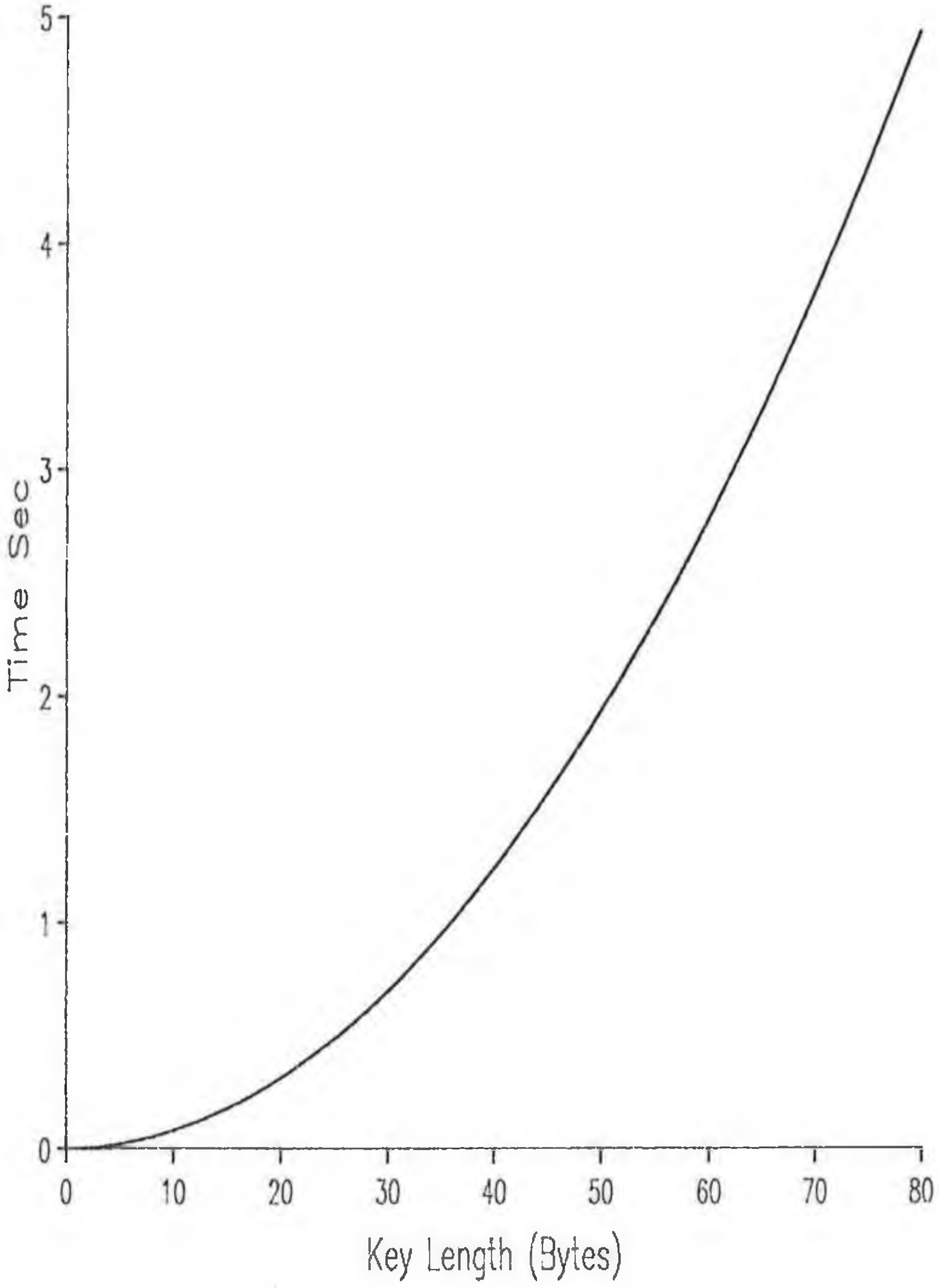
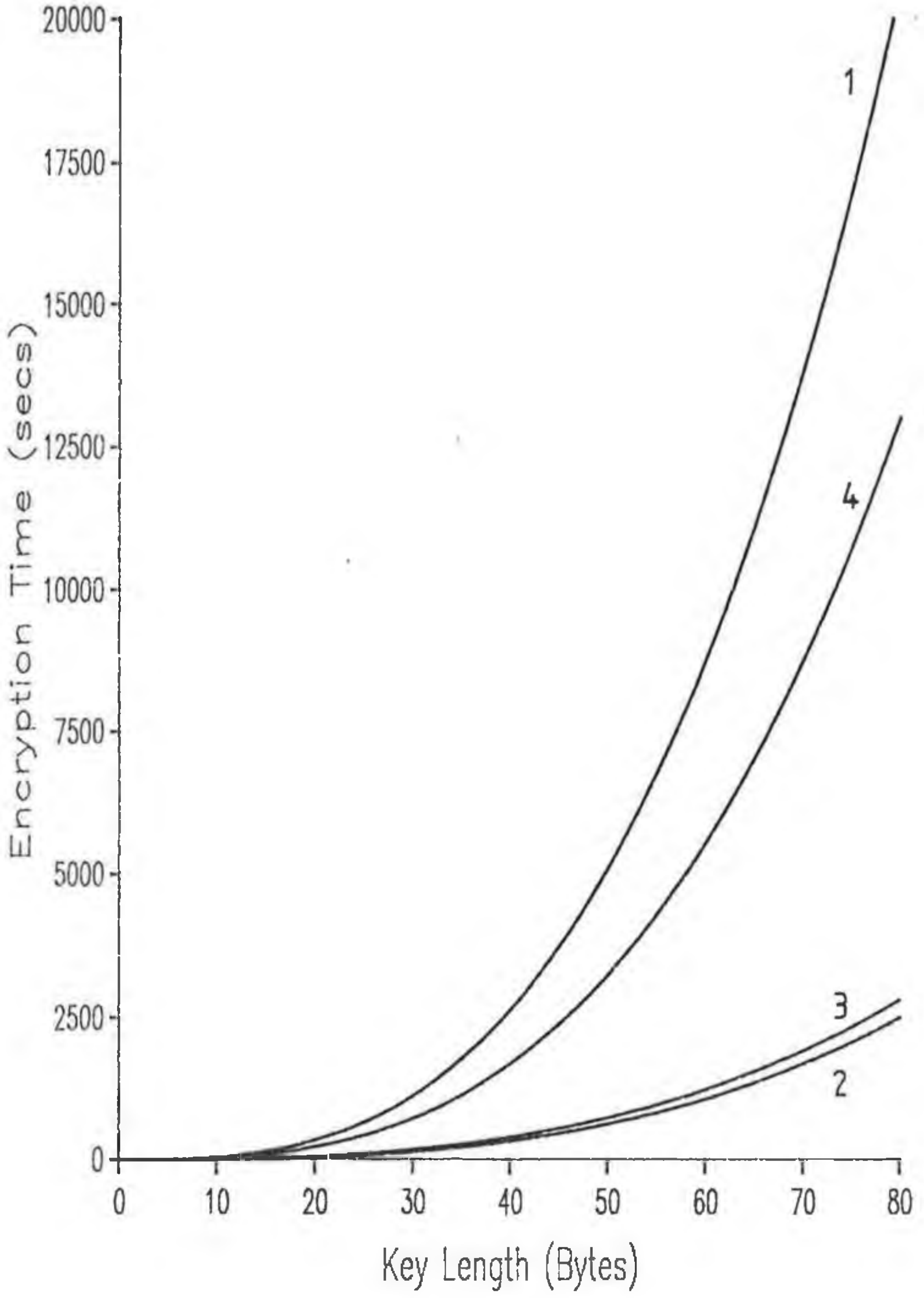


Fig. (3.5b) Runtime of Blakelys algorithm



1: Shift and subtract
2: Reciprocal
3: Knuths algorithm
4: Blakelys algorithm

Fig. (3.6b) Encryption times

Key Length (Bytes)	Method 1	Method 2	Method 3	Method 4
8	24	3	6	17
10	45	6	10	31
12	76	10	16	52
14	119	16	24	80
16	176	23	33	118
18	248	32	45	165
20	338	43	59	223
22	448	57	76	293
24	578	73	96	378
26	732	92	120	476
28	911	114	146	591
30	1117	139	176	723
32	1352	168	211	873
34	1618	200	249	1042
36	1917	236	292	1232
38	2250	277	339	1443
40	2620	322	391	1678
42	3029	371	448	1937
44	3478	426	510	2221
46	3969	485	578	2531
48	4504	550	652	2869
50	5086	620	731	3236
52	5715	696	817	3633
54	6395	778	910	4061
56	7126	867	1009	4522
58	7911	961	1115	5016
60	8752	1062	1228	5545
62	9651	1171	1348	6109
64	10608	1286	1476	6711
66	11628	1408	1612	7351
68	12710	1538	1756	8031
70	13858	1676	1909	8751
72	15072	1822	2070	9513
74	16356	1976	2239	10318
76	17711	2139	2418	11167
78	19138	2310	2606	12061
80	20640	2490	2804	13003

Method 1	:	Shift and subtract
Method 2	:	Reciprocal
Method 3	:	Knuths algorithm
Method 4	:	Blakelys algorithm

Fig. (3.6b) Encryption Times

4. Key Generation

4.1 INTRODUCTION

Like many cipher systems the security of the RSA public key cipher is not absolute. The security of the system relies on the necessity for large computational resources to break the cipher. This is called computational security. To evaluate the level of security provided by the system it is necessary to estimate the computational resources required to defeat it. If these resources are excessive then the system can be considered secure. As will be seen later the RSA cipher allows the user to define the level of security required for a particular application .

4.2 SECURITY OF THE RSA CIPHER

The security of the RSA cipher is entirely dependent on the decryption key. An eavesdropper who obtains this key will be able to decrypt any messages sent to the keys owner. It is also possible for the eavesdropper to pose as the owner in any transactions with a third party. In the RSA cipher the encryption key (e,N) is made public and the decryption key (d,N) is kept secret. The eavesdropper thus has access to all the key parameters except d. The decryption key d is related to the encryption key e by the following relationship:

$$ed \equiv 1 \pmod{\emptyset(N)}$$

where $\emptyset(N)$ is the Euler totient function of the encryption modulus N. To calculate d it is thus necessary to know $\emptyset(N)$. Other methods of obtaining d e.g by a direct search of the keyspace, can be prevented by choosing d uniformly from a large enough set. The Euler totient function of N is the number of integers less than N and relatively prime to N. In this case as discussed previously N is the product of two prime numbers p and q and thus $\emptyset(N)$ is equal to (p-1)(q-1) i.e

$$N = pq \Rightarrow \varnothing(N) = (p-1)(q-1) \quad \text{for } p,q \text{ prime.}$$

The cryptanalyst must therefore find the prime factors p,q of N to obtain Ø(N). Other methods of calculating the Euler totient function without knowing p and q may be possible but RIVEST, SHAMIR and ADLEMAN show that any such method is equivalent in complexity to factoring N. An attack on the cipher can thus be reduced to the problem of finding the prime factors of N. Note that while it is believed that this is the case this hypothesis has not been proven and a method may be found which has a complexity less than that of factoring. The RSA cipher has however withstood attempts to find such a method since its proposal and thus the possibility of finding such a method is remote.

4.3 CHOICE OF KEY SIZE

A large number of algorithms exist for finding the prime factors of a number, n, . A detailed discussion of many of these algorithms is given in KNUTH(8). The fastest known algorithm, is attributed to SCHROEPPEL(12) and has a run time of order :

$$n^{(\ln(\ln(n))/\ln(n))^{1/2}}$$

If $n = 10^K$, where K is an integer, then the run time is of the order of :

$$10^{(K(\ln(2.3K))/2.3)^{1/2}}$$

where n is the number to be factored. The size of n can thus determine the computational resources necessary to defeat the cipher. In their original paper Rivest, Shamir and Adleman recommend choosing N to be of the order of 200 decimal digits. The time required to factor numbers of this size is 3.8×10^9 years, at 1 microsecond per instruction. This provides a large safety margin for increases in the speed and efficiency of the algorithms.

The choice of the key size also affects the time required for encryption and decryption. Keys of 200 decimal digits may not be required in many cases and smaller keys would allow faster encryption times. For example if the key length was reduced to 100 digits, the factoring time is reduced to 74 years and the encryption time is reduced from 46 minutes to 6 minutes (see Fig.3.11). The level of security provided by the cipher and hence the transmission speed can be chosen to suit the application.

4.4 CHOICE OF PRIMES.

Although finding the prime factors of a number is very time consuming in the worst case, a bad selection of p and q may undermine the security of the cipher by allowing algorithms to exploit properties of the primes. Certain choices of primes will allow the cipher to be broken by repeated enciphering (NORRIS and SIMMONS), i.e :

$$E(E(E(.......E(M))....)=M$$

where the number of encryptions required is relatively small ($O(10^6)$). This form of attack can be countered by choosing p and q to be safe primes, i.e

$$p = 2p_1 + 1$$
$$q = 2q_1 + 1$$

where p, p_1, q, q_1 are odd primes. Other forms of attack are also feasible if p and q are not carefully chosen (BLAKELY and BLAKELY).

To protect against these and other forms of attack KNUTH recommends that the following conditions should be met by the primes :

1. p and q should not be the same length but should differ by a few digits.
2. The numbers (p-1) and (q-1) should both have large prime factors.
3. The greatest common divisor of (p-1) and (q-1) should be small.

In addition to these three guidelines it is also recommended that p+1 and q+1 should not consist of small prime factors. KNUTH suggests the following procedure for choosing primes :

1. Generate a random number r_1, with 80 or 81 digits.
2. Search for the first prime number p_1 greater than r_1.
3. Choose another random number r_2 with approximately 40 digits.
4. Search for the first prime number, p, greater than r_2 which is of the form:

   $$kp_1 + 1$$

   where $k \gg p_2$, k is even and $k \equiv p_1$ (Mod 3). This prime, which will have around 120 digits, can be used in the key. A similar process can be used to find q.

Before implementing the above algorithm it is necessary to discuss ways of generating large random numbers and procedures for testing numbers for primality.

4.5 GENERATION OF RANDOM NUMBERS.

To generate truly random numbers it would be necessary to monitor a random event, such as the tossing of a coin or the throwing of a dice, for a fixed period of time. This approach is not practical, nor is it really necessary, for most applications. The requirement in most case is to produce a sequence of numbers which appear random. These sequences, called psuedo-random sequences, must fulfill two criteria. The first is that the sequence should not repeat itself for a long time and the second is that the terms of the sequence should be uniformally distributed over a given interval.

The algorithm used for the generation of a sequence of random numbers is known as the linear congruential method. In this method the next random number R_{i+1} is determined from the present number R_i using the relationship :

$$R_{i+1} = (aR_i + b) \text{ Mod } m$$

where the numbers a, b and m, known as the multiplier, increment and modulus respectively are carefully chosen. The initial seed number R_0 has also to be chosen. The maximum sequence length obtainable is the modulus, m, but this will only be obtained if the following criteria (KNUTH) are adhered to :

1. The increment, b, should be relatively prime to the modulus, m, i.e gcd(b,m) = 1.
2. The number (a-1) should be a multiple of every prime factor of m i.e if
   $$m = p_1^x p_2^y p_3^z$$
   where p_1, p_2, p_3 are prime x,y,z are integers then
   $$a = kp_1p_2p_3$$
   where k is an integer. In the simplest case where $m = p_1p_2p_3$ then a = km.
3. If the modulus, m, is a multiple of 4 then the increment b must also be a multiple of 4 i.e if $m \equiv 0 \pmod 4$ then $b \equiv 0 \pmod 4$.

Once the initial choice of parameters have been chosen the linear congruential generator is readily implemented in a high level language. A random number generator of this form was implemented in FORTRAN on the VAX. A flowchart showing the programme used to generate the parameters a,b,m is shown in Fig. (4.1).

4.6 PRIMALITY TESTING

To determine if a number is prime it is necessary to show that it has no factors except itself and one. This is equivalent to the factoring problem on which the security of the RSA cipher lies and is thus impractical. However there are methods for determining, with high probability, if a number is prime or not. These algorithms will, in some instances indicate that a number is prime when it is composite. The probability of this occuring can be reduced by sucessive iterations . A detailed discussion of primality testing is given in KNUTH(8).

The basis of many of the 'probabilistic' primality tests resides in Fermats theorem which states that :

$$a^{p-1} \text{ (Mod p)} = 1$$

where p is prime and a is any integer that is not a multiple of p. If the number, p, is composite then the above relationship may still hold for various values of a. To determine, with high probability, that p is prime it is necessary to ensure that the above equation hold for many different values of a.

Calculation of the above test requires modular exponentiation and its complexity is thus of the same order as encryption and decryption using the RSA cipher. To increase the speed of the test, by reducing the size of the exponent, SOLOVAY and STRASSEN (15) proposed the following modification. To test a number p for primality, an integer, a,is chosen randomly from the range $1 < a < p-1$. If p is prime then it will satisy the following conditions :

1. a and p are relatively prime i.e $\gcd(a,p) = 1$
2. $J(a/p) \text{ Mod } p = a^{(p-1)/2} \text{ Mod } p$

wher J(a/p) is the Jacobi symbol(5). The propability of a composite number passing the above tests is 1/2, the test must therefore be repeated a number of times to reduce the probability that p is composite to acceptable limits. If p passes the test H times then the probability that p is composite is $1/2^H$. Repeating the test 100 times (H=100), as recommended by RIVEST, SHAMIR and ADLEMAN, would reduce the probability that p is composite to negligible levels ($1/2^{100}$).

This algorithm for testing prime numbers was implemented in FORTRAN on a VAX, according to the flowchart shown in Fig. (4.2)

4.7 GENERATING KEYS.

Suitable procedures for generating random numbers and for testing numbers for primality have been discussed. A key generation scheme based on these procedures, and using KNUTHs recomendations will now be given. It is first necessary to decide on the key length. This will affect the security of the cipher and the time required for encryption. A value in the range 100 to 200 decimal digits is recommended (4). The following steps should then be taken.

1. Chose two prime numbers p and q using the procedure described in section 4.4.
2. The modulus N is the product of p and q i.e $N = pq$.
3. Choose an encryption (or decryption) key e that is relatively prime to $\varnothing(N)$ i.e $\gcd(e, \varnothing(N)) = 1$. This can be done by generating a random number using the method of section 4.5 and testing that it is relatively prime to $O(N)$ using Euclids(8) algorithm. If it is not relatively prime to $\varnothing(N)$ another random number can be generated. This is the users public key and it can be inserted in the public directory.
4. Calculate the decryption key, d, as the multiplicative inverse of e modulo $\varnothing(N)$ i.e

$$ed = 1 \ (\text{Mod } \varnothing(N))$$

This equation can be solved for d using Euclids algorithm. This number is the secret key of the user and steps should be taken to ensure that it remains secret.

A flowchart for the above key generation scheme is shown in Fig. 4.3. This flowchart was implemented in FORTRAN on a Vax.

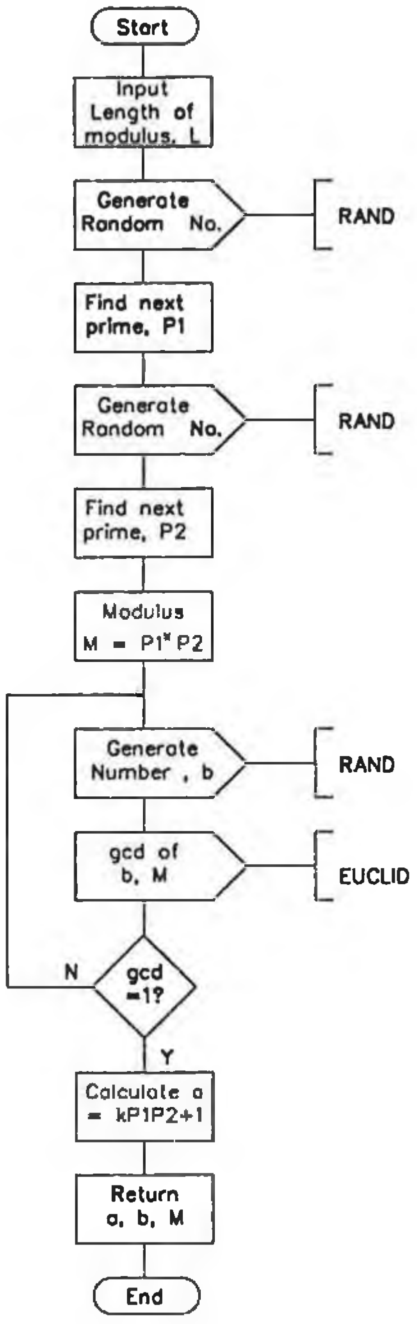
4.9 EFFECT OF KEY GENERATION ON ENCRYPTION.

The runtime of the key generation algorithm described above was of the order of 10 to 20 minutes for keys in the

100 digit range on a VAX 11/780. The time required on a personal computer would be larger. The effect of this time on the overall encryption/decryption process is not significant as the key is only calculated once per user. However in some applications short keys (<100 digits) are used with frequent changing of keys. In this case the time required to calculate the keys becomes significant and will affect the overall efficiency of the unit. To overcome this problem a number of keys could be generated by a powerful computer, such as a VAX, and downloaded to the encryption device (a personal computer or dedicated device) on a regular basis. This would remove the need for the encryption device to calculate the keys but would increase the problem of key management.

4.10 MICROPROCESSOR IMPLEMENTATION.

Implementaion of the key generation scheme on a microprocessor is feasible but suffers from the same limitations as the use of a personal computer i.e that of speed. Frequent changing of the keys unless carried out during some idle period (e.g at night) would be too time consuming. The use of a dedicated microprocessor to generate the keys and to perform encryption and decryption offers security advantages in that the secret key need never leave the microprocessor and would thus be unknown to everyone, including the user.

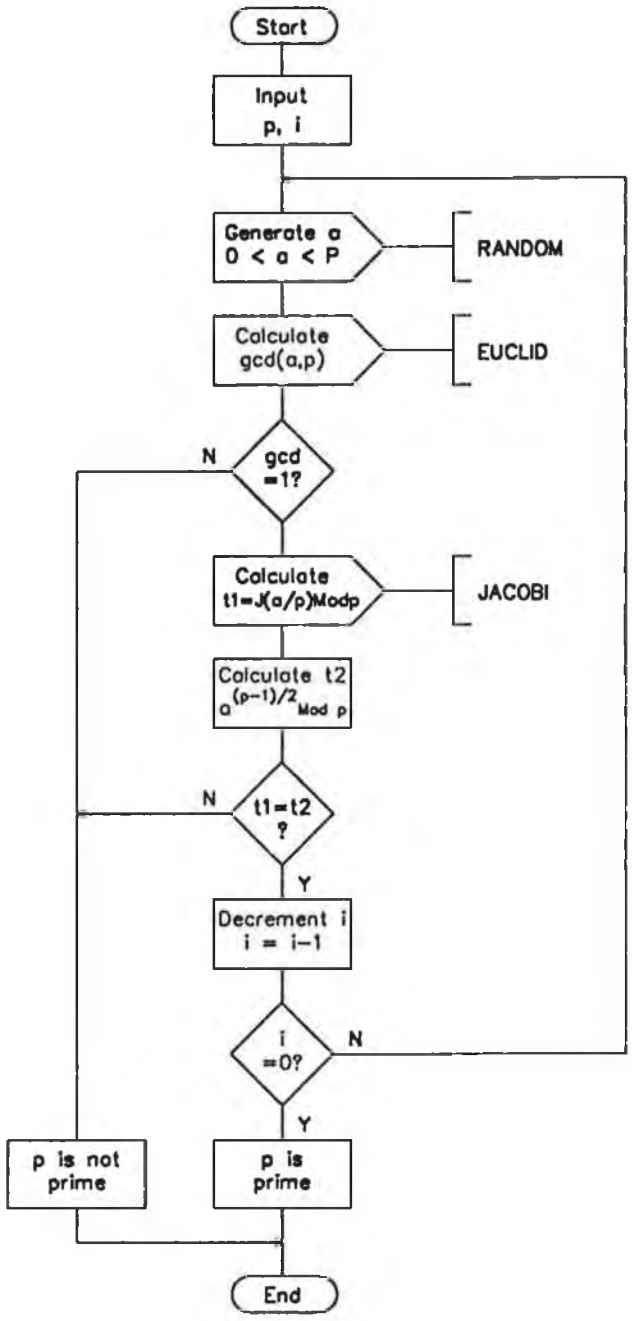


a	:	Multiplier
b	:	Increment
M	:	Modulus
x	:	Arbitrary integer > 2
k	:	Integer chosen so that kP1P2+1 (Mod 4) = M (Mod 4) if M (Mod 4) = 0

$$R_{i+1} = (aR_i + b) \text{ Mod } M$$

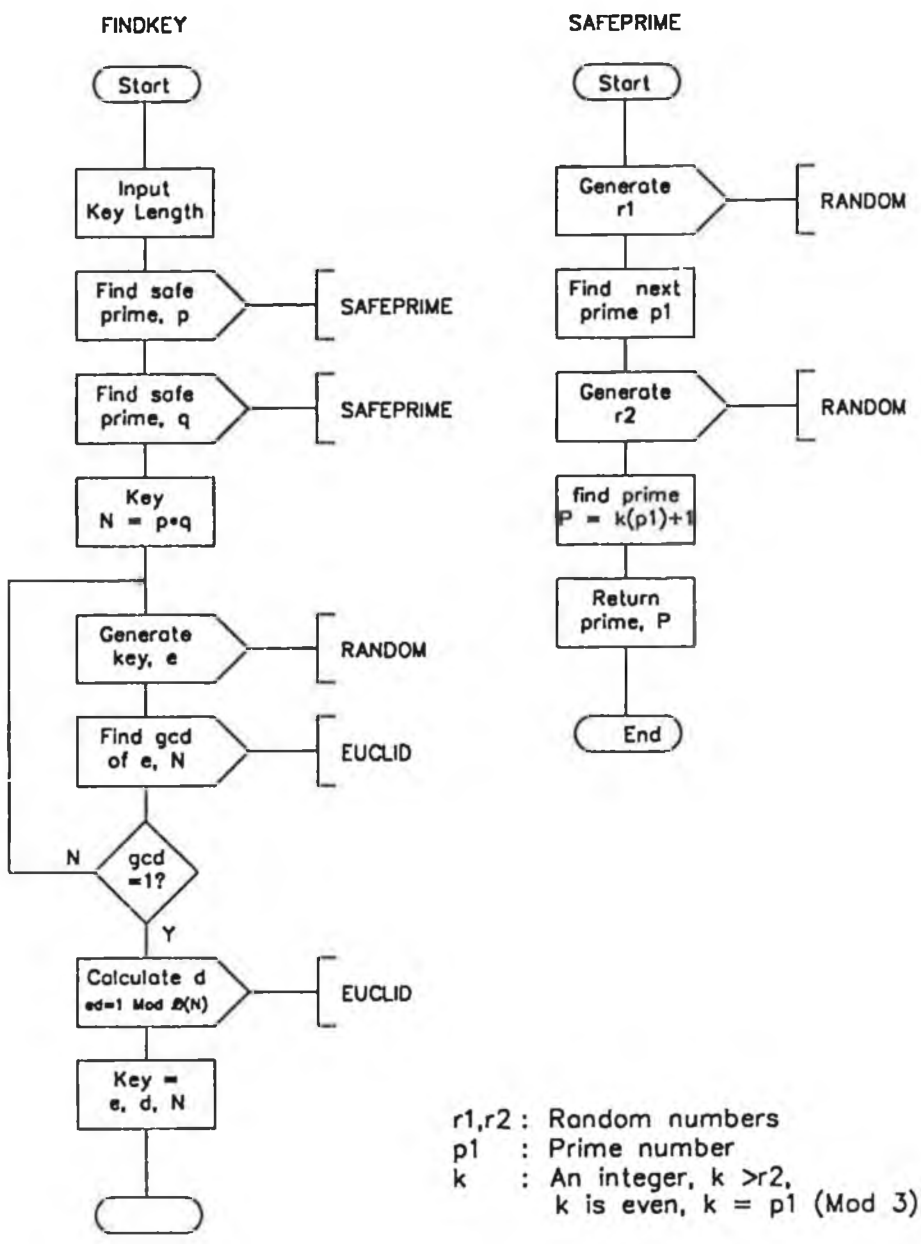
Fig. (4.1) Random Number generator –Parameters

PRIMETEST



p	: Number which is to be tested for primality
i	: No. of iterations (probability of p being composite)
a	: Random number in the range [0, p−1]
t1, t2	Temporary variables
J(a/p)	Jacobi symbol (see text)

Fig. (4.2) Primality Testing



r1,r2	:	Random numbers
p1	:	Prime number
k	:	An integer, k >r2, k is even, k = p1 (Mod 3)

p,q	:	Safe primes generated using Knuths procedure
N	:	Encryption modulus.
e	:	Users public encryption key
d	:	Users secret decryption key
O(N)	:	Euler totient function of N, O(N) = (p-1)(q-1)

Fig. (4.3) Key Generation Scheme

5. Hardware Multiplication

5.1 INTRODUCTION

The implementation of the RSA public key cipher using a standard unassisted 8 bit microprocessor and several software algorithms has been discussed in chapters 2 and 3. It has been shown that the encryption time i.e the time required to perform modular exponentiation using these algorithms is too slow by several orders of magnitude for a practical general purpose encryption device. To achieve the necessary increase in speed it is necessary to consider ways of implementating part or all of the encryption algorithm in hardware. Several ways of doing this were considered :

1. Addition of arithmetic co-processor to the microprocessor.
2. Use of dedicated hardware multipliers to assist microprocessor.

Arithmetic co-processors are available for most standard microprocessors. These co-processors perform arithmetic functions and pass the results to the main processor. These processors are usually optimised for floating point operations. The use of a coprocessor in modular exponentiation causes the same problems as the use of the unassisted microprocessor namely that of word length. The word length recommended for the RSA public key cipher is of the order of 400 bits. Commercially available arithmetic coprocessors can not deal directly with word lengths of this size and they cannot be cascaded. The cost of these coprocessors rules them out for many applications.

A dedicated hardware multiplier is a device which performs only multiplication, unlike the arithmetic co-processor which can perform many other arithmetic functions. These multipliers deal mainly with signed or unsigned binary integers. The use of these devices in many aspects of digital signal processing has produced a wide variety of such devices.

5.2 HARDWARE MULTIPLIERS

Hardware multipliers perform multiplication using the 'shift and add' algorithm that was discussed previously. Consider two operands :

$$A = (a_{n-1}, a_{n-2}, \ldots\ldots a_1, a_0)_2$$
$$B = (b_{m-1}, b_{m-2}, \ldots\ldots b_1, b_0)_2.$$

where a_i and b_i are binary digits. The product A * B is given by :

$$C = A*B = (a_{n-1}, \ldots\ldots a_1, a_0)_2 * (b_{m-1}, \ldots\ldots b_1, b_0)_2.$$

$$= \left(\sum_{i=0}^{n-1} a_i 2^i\right)\left(\sum_{j=0}^{m-1} b_j 2^j\right)$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} (a_i b_j)\, 2^{i+j}$$

thus C is the shifted sum of the partial products $a_i b_j$. The bitwise multiplication of binary digits is equivalent to logical AND. Multipliers can be divided into three categories which describe the way in which they implement the above expression :

1. Serial Multipliers.
2. Serial/Parallel Multipliers.
3. Parallel Multipliers.

5.2.1 Serial Multipliers

In a serial multiplier the operands are input to the device serially i.e one bit at a time. A diagram for a typical serial adder is shown in Fig. (5.1). It consists of a logical AND gate, a full adder with carry feedback and a temporary n bit storage register, $R = (r_{n-1} \ldots\ldots r_0)$. The AND gate forms the bitwise product of $a_i b_j$. This is added to least significant bit of the temporary storage register, r1, and shifted to form the new partial product.

Thus

$$r_k = r_{k+1} \qquad 0 \leqslant K \leqslant n-1$$

and

$$r_n = (a_i b_j) + r_1 + c_i$$

where c_i is the input carry bit. The result is obtained by clocking out the least significant bit of register R every mth clock cycle.

$$C = \sum_{i=0}^{n-1} 2^i \left(a_i \sum_{j=0}^{m-1} b_j 2^j\right)$$

Now the contents of register R at time i are:

$$R_i = a_i \sum_{j=0}^{m-1} b_j 2^j$$

therefore

$$C = \sum_{i=0}^{n-1} 2^i R_i$$

This is most easily seen in an example. Consider A = (10110) and B = $(1011)_2$. The steps required to multiply these integers and the contents of the partial result register at each step is shown in Fig.(5.2). As can be seen the time required to multiply an n bit number by an m bit number is m*n clock cycles. The maximum clock speed is determined by the carry propagation delay T_c.

5.3.2 Serial/Parallel Multiplier

In a serial/parallel multiplier one of the operands is presented to the device in serial form, the other is presented in parallel form. There are two possible ways of implementing such a multiplier :

1. Horners method.
2. Right to left factorisation method.

5.3.2.1 Horners Method

In Horners method the product C = A*B, can be written as:

$$C = A*B = \sum_{i=0}^{n-1} 2^i a_i B$$

This can be expanded in the form :

$$C = ((..(\ a_{n-1}B)2 + a_{n-2}B)2 + \ldots\ )2 + a_1B)2 + a_0B$$

The partial product P_i can be defined by the recurrence :

$$P_n = 0,$$
$$P_{i-1} = 2P_i + a_{i-1}B \qquad i=n,\ n-1,\ \ldots.3,\ 2,\ 1.$$

It can be seen from the above expression that the result C is equal to P_0. A diagram of a circuit to implement this recurrence is shown in Fig. (5.3). In this circuit an m bit full adder is required. Also as there are n additions, there is a possibility of obtaining n carry bits. To accomadate this it is necessary to include an n bit half adder. Alternatively a m+n bit full adder could be used.

5.3.2.2 Right to Left Factorisation

The right to left factorisation method removes the need for a double length i.e (m+n) bit adder. Consider the product C = A*B which gives :

$$C = \sum_{i=0}^{n-1} 2^i a_i B$$

Expand this as in Horners method and divide both sides by 2^{n-1}

$$C/2^{n-1} = a_{n-1} + 2^{-1}(a_{n-1}B + 2^{-1}(\ldots+2^{-1}(x_1B + 2^{-1}(a_0B)..)$$

Again as in Horners method if we define a partial product P_i by the recurrence :

$$P_{-1} = 0,$$

$$P_{i+1} = 2^{-1}P_i + a_iB , \qquad i = -1, 0, 1, \ldots.., n-1$$

This result is similar to the Horner method except for the order in which it accesses the bits of operand A, from most significant (MSB) to least significant (LSB) for Horner and from LSB to MSB for this method. However this method has a significant advantage in that at any time i, the i least significant bits of the result are known. This differs from the Horner in which a carry can propagate throughout all the bit positions at any time. This allows the use of a single length (m bits) adder in the circuit. A diagram of such a circuit appears in Fig.(5.4). This consists of m and gates, an m bit full adder and a (m+n) bit shift register. The operand B is presented to the multiplier in parallel form while A is presented one bit at a time. Both of these algorithms shift the partial result one bit if a_i is 0 and add B to the partial result if a_i is 1.

5.3.3 Parallel Multiplier

In all of the multipliers described so far at least one of the operands is input to the multiplier in serial form. However a consideration of the original expression for the product C = A*B shows that this is not necessary :

$$C = A*B = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} 2^{i+j}(a_ib_j)$$

The partial products (a_ib_j) can be computed independently and summed. From the above expression it can be seen that there are (m*n) partial products. It will thus be necessary to have (m*n) AND gates and (m*n) adders. A diagram of a typical parallel multiplier is shown in Fig.(5.5). It can be seen from this that the shift required between partial products is produce by physical wiring between adders. As both operands are available to the multiplier in parallel form the product can be calculated in a single clock cycle.

5.3 SPEED AND SPACE COMPARISON OF MULTIPLIERS.

The three types of multipliers described above can be compared in terms of speed and physical size. The speed of the multiplier will be determined in terms of n the number of bits in the operands. For the purpose ofthis comparison it will be assumed that both operands contain the same number of bits.

The serial multiplier is the slowest of the multipliers as it takes $O(n^2)$ clock cycles to calculate the product. In terms of physical size it is however the smallest as it only rquires a single bit adder. The temporary register can be implemented using one of the operand registers. The size of the multiplier is thus independent of n i.e O(1) as it is assumed that operand registers are always available. The maximum clock speed is limited by propagation delays within the adder.

The serial/parallel adder can calculate the product in O(n) clock cycles. It is thus significantly faster than the serial adder. In terms of space however, it requires an n bit full adder, n AND gates and a 2n bit shift register. The amount of space required is thus O(n). The maximum clock speed usable with this type of multiplier is determined by the speed of the adder. The speed of the adder is limited by the carry propagation time. To illustrate this consider the addition of $A=(0111)_2$ and $B=(0001)_2$:

```
 0111
+0001
 1000
```

Addition of the least significant bits produces a carry into the next bit. This carry when added produces a further carry into the next bit, and so on until the most signifcant bit changes. The value of the most significant is thus dependent on the carry produced in the least significant bit. If the propagation delay is T_p then the time required before a valid result is obtained is nT_p. There are ways of reducing this delay, some of which will be described later, however the delay is still a function of the number of bits in the adder. The maximum clock speed possible with a serial/parallel adder is thus less than that of a serial adder and decreases as n increases i.e $f_{c(max)}$ 1/n.

The parallel multiplier is the fastest of three types as it can calculate the product in a single clock cycle. The multiplier however requires n^2 AND gates and n^2 adder elements. The space requirements are therefore $O(n^2)$ while the time is limited only by delays within the circuit itself. As can be seen from the diagram in Fig.(5.5) the maximum propagation delay is $O(nT_p)$, where T_p is the worst case propagation of the adder.

The three type of multipliers therefore represented time/space tradeoffs. If speed is of prime importance and space is not a problem then the parallel multiplier would be chosen. Conversely if speed is not a problem but space is the serial multiplier would be the best choice. The serial/parallel adder is a compromise. To summarise therefore the multipliers can be compared as follows :

Type	Delay	Time	Space
Serial	O(1)	$O(n^2)$	O(1)
Serial/Parallel	O(n)	O(n)	O(n)
Parallel	O(n)	O(1)	$O(n^2)$

where time is in clock cycles and delay is in term of worst case adder propagation delay.

5.4 CHOICE OF MULTIPLIER.

To decide on a multiplier it is necessary to consider the size of operands that will be required. The use of a keylength of 200 decimal digits, or approximately 600 bits, is recommended for high security applications. For lower levels of security, and consequently higher transmission rates, shorter keys i.e 300 bits could be used. The choice of multiplier should enable differing levels of security to be offered by allowing multipliers to be cascaded to produce the bit length, and hence the security level, required.

The serial multiplier can easily handle the various bit lengths required. As the cost of hardware is independent of bit length the cost of such a device would be low. The speed of the serial multiplier is $O(n^2)$ and thus woud take O(360000) clock cycles to multiply 600 bit numbers. the maximum clock speed is limited by the propagation delay of the adder. The use of very high speed (>10MHz) clocks, however, causes other problems e.g circuit layout becomes critical. Thus the clock should be restricted to 10MHz. This would result in a time of O(36mS) for the multiplication. The time required to multiply two 600 bit numbers using subroutine MULT, described in chapter 3, is approximately 300mS. The serial multiplier therefore provides a tenfold increase in speed. This is not however considered sufficient.

A 600 bit parallel multiplier could produce the result in one clock cycle, the propagation delay being O(1200) adder propagation delays. The space requirement for such a multiplier is O(360000) which is excessive. The space, cost and power requirements of so many elements renders such a multiplier impractical. The parallel multiplier is not easily cascaded, to illustrate this consider implementing an 8x8 bit parallel multiplier using 4x4 bit multipliers. A diagram of the circuit required is shown in FIG.(5.6). As can be seen from this figure 4 4x4 bit multipliers, along with adders are required. It would therefore not be feasible to implement a parallel multiplier for use with small key lengths and cascade it for larger keys.

The serial/parallel multiplier offers a compromise in terms of speed and size and for this reason it was decided to use this type of multiplier. The time required is O(n) i.e O(600) clock cycles. The hardware cost is O(600) which is significantly less than that of a parallel multiplier. Serial/parallel multipliers are also easily cascaded to produce different word lengths thus allowing the level of security to be easily chosen. The delay is greater than that of a serial multiplier, thus a slower clock must be used. If fast adders were implemented using 'look ahead carry' techniques, as shown in Fig. (5.7) a clock of 1MHz could be used. This would result in a multiplication time of 600uS which is a factor of 500 times greater than subroutine MULT.

5.5 MODULAR MULTIPLIERS.

The use of a multiplier to assist in the implementation of the RSA public key encryption algorithm would be of greater benefit if the multiplier could perform modular multiplication directly. The most time consuming element of the encryption algorithmis the calculation of the modular multiplication A*B mod M. the algorithms described in chapter 3, with one exception, calculate this producvt in two steps. First the integer multiplication A*B is perrformed and then the result is reduced modulo M. Of the algorithms described in chapter three only BLAKELYs algorithm performed modular multiplication directly.

As mentioned in chapter 2 BLAKELYs algorithm is a modification of the 'shift and add' multiplication method. Consider the expression for the product of two binary integers A and B to form the product C :

$$C = A*B = \sum_{i=0}^{n-1} 2^i a_i B$$

Modification of this expression to produce the modular product A*B mod M, where $M = (m_{k-1}, \ldots, m_0)$ is the required modulus and $0 < A,B < M$, yields :

$$C \text{ (mod M)} = A*B \text{ (mod M)} = (\sum_{i=0}^{n-1} (2^i a_i B \text{ mod M})) \text{ mod M}$$

Comparing this to the 'shift and add' algorithm it can be seen that it is necessary to reduce each partial product modulo M and to add the partial products modulo M. To reduce an integer modulo M it is necessary to compare the integer to M and if it is greater than M subtract M from it i.e if P is the integer then :

$$\begin{array}{llll} P \bmod M & = P, & 0 & P < M \\ P \bmod M & = P-M, & M & P < 2M \end{array}$$

Note that if P is reduced after every addition then P is always less than 2*M. A flowchart detailing BLAKELYs algorithm, with particular reference to hardware implementation is shown in FIG. (5.8). From this flowchart it can be seen that the following hardware is required :

1. An adder
2. Registers to hold, the operands A,B, the modulus M and the result R.
3. A control unit.

The adder is the main computational unit. It has two input operands. One of the operands is the result R, the other is either the operand A or the modulus M, the choice being governed by the control unit. The modulus register M can contain -M, in twos complement form, instead of M. The adder will thus be able to perform the calculation R+A or R-M, which is all that is required as can be verified by the flowchart.

The registers are required to hold the operands and intermediate results throughout the calculation. Again it can be seen from the flowchart that a shift register is required for operand B and result R, whereas a standard register will suffice fo operand A, and modulus M.

The control unit implements the sequence of instructions described in the flowchart. It requires as inputs the operand B and a decision on whether R is greater or less than M. It controls the adder and the outputs and clocking of the registers.

A block diagram of the hardware described above is shown in FIG. (5.9).

5.6 IMPLEMENTATION.

There are several ways in which the modular multiplication device can be implemented. The method chosen will depend on cost, ease of implementation and expansion, physical size and its ease of integration into the overall encryption algrithm. the methods of implementation considered were :

1. Discrete hardware using standard SSI and MSI devices.
2. Implementation using bit-slice computer techniques.
3. Use of application specific integrated circuits (ASICS)

5.6.1 Discrete Hardware

The modular multiplication device can be readily implemented using standard integrated circuit building blocks. The TTL and CMOS series of logic circuits can supply all of the required function blocks e.g adders, shift registers, registers etc. The control unit can be implemented as a finite state machine using either hardwired logic i.e NAND, NOR gates and FLIP-FLOPs, or by using programmable devices such as PROMS, PALS etc. The use of programmable logic allows the flexibility of changing the sequence of operations carried out by the control unit should this be required. The disadvantage of this method is the space required for its implementation. All of the building blocks are seperate devices and thus a large number would be necessary to realise the device. The bit lengths available with this form of construction are small generally being 4 or 8 bits and while the structure is cascadable the size would quickly become excessive.

As discussed earlier the major constraint when using microprocessors to perform multiplication on large (>100) bit numbers is the limited data path, usually 8 or 16 bit available to the microprocessor. This necessitates that the operands be divided into more manageable size with a subsequent reduction in the speed of operation. The core of a microprocessor is an arithmetic and logic unit (ALU) which performs calculations and input and output registers for the storage and shifting of operands and results. On any given microprocessor the width of the ALU is predetermined and cannot be changed.

5.6.2 Bit Slice Devices

Bit-slice devices however seek to implement a single or multiple bit ALU with the possibility for cascading to produce any desired bit length. Other bit-slice devices are available for implementing control sequences. With bit-slice devices it would be possible to implement an ALU of sufficient length to perform Blakelys algorithm. The disadvantage with this approach is similar to that when using standard logic building blocks namely that of size. At present most bit-slice dvices deal with 4 bit, although 8 and even 16 bit devices are becoming available but at high cost. There are also a number of peripheral devices, such as sequencers which are also required. Therefore a large number of devices would be required and the cost of the completed unit would be large. The bit-slice approach is intended mainly for the design of high speed general purpose CPUs and as such are generally implemented in TTL or ECL technology. The power requirements of a large number of such devices would be excessive.

5.6.3 Application Specific Integrated Circuits (ASICS)

The use of semi or full custom integrated circuits (ASICS) has a higher initial cost both in terms of design time and actual production costs. The advantage however is that a circuit can be tailored to suit the needs of the encryption device andthe space required should be greatly reduced. At present there are three major techniques for integrated circuit design:

1. Gate Arrays Design.
2. Standard Cell Design.
3. Full Custom Design.

The use of Gate Array or Standard Cell design is referred to as 'semicustom' design as part of the design work has already been carried out by the manufacturer.

5.6.3.1 Gate Array

The Gate Array, also known under various trade names such as uncommitted logic array (ULA), consist of a two dimensional array of cells. Each cell consist of a number of circuit elements e.g. N and P channel transistors etc. The design process entails connecting these cells up so as to perform the required function. This connection is done on the metal layer(s) which are not predetermined. As only one (or two) layer needs to be designed and the rest of the i.c. is standard this form of semi-custom circuit is the quickest and least expensive to implement. Many software design tools are available to aid in this process, many of which can take a circuit diagram and generate the masks required for the metal layers automatically.

5.6.3.2 Standard Cell

The standard cell approach is essentially similar to design using TTL or CMOS building blocks. With this method a large number of building blocks are predefined e.g logic gates, flip-flops etc. the designer must lay these out on the wafer and interconnect them. This is a much more difficult process than gate array design as there are many variables. The positioning of the cells and the interconnection paths must be determined by the designer. The level of software tools available for standard cell layout is not as great as for gate array design. The advantage of the standard cell over the gate array is that denser layout, and hence more functions per wafer, is possible using standard cells.

5.6.3.3 Full Custom

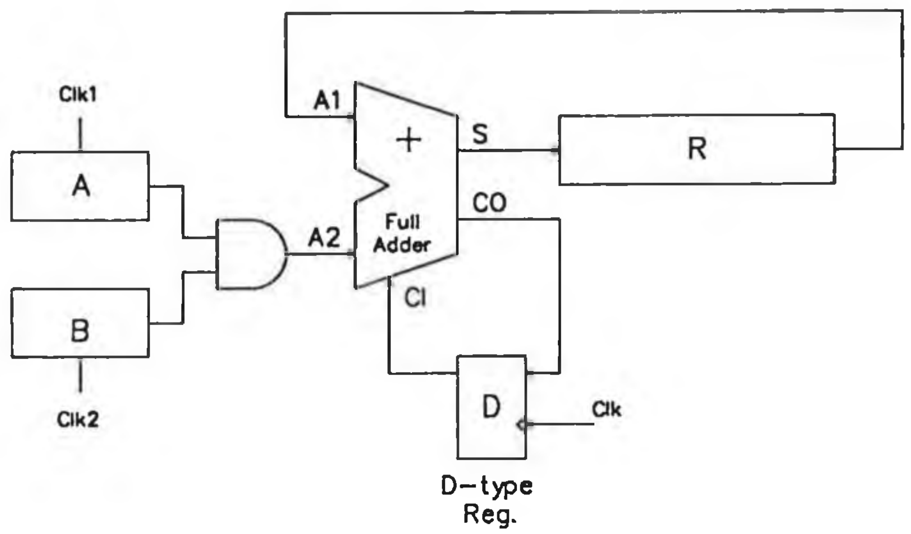
Full custom design requires that the designer designs the complete mask set for the wafer using no predefined cells. This is a much more difficult approach than semi-custom as it requires a detailed knowledge of i.c design and a large amount of time. This method however produces the densest layout.

Of the three methods the Gate Array is the least expensive and easiest to implement, however it is also the least dense of the methods. The standard cell is more expensive to design and produce but offers higher density. The full custom method is the most expensive to produce but offers the highest density.

5.7 CHOICE OF IMPLEMENTATION METHOD.

The choice of implementation method is determined by size, cost, and expected volume. The discrete hardware approach is the least expensive but requires a large amount of space. A bit slice approach is more expensive and the amount of space required would compare with the discrete approach. The use of dedicated integrated circuits offers the best possibility for space reduction but the cost, especially in the low volume required for a research project, is prohibitive.

When this decision was being considered, the National Board of Science and Technology (N.B.S.T.) were offering grants to enable researchers avail of the facitilities of the National Microelectronics Centre (N.M.R.C.), where full custom and semi-custom integrate circuit design is carried out. A grant was applied for and granted for the design of the modular multiplication device at the N.M.R.C. For this reason it wass decided to design a dedicated integrated circuit. This process will be described in the next chapter.



A : Multiplier (n bits)
B : Multiplicand (m bits)
R : Shift Register

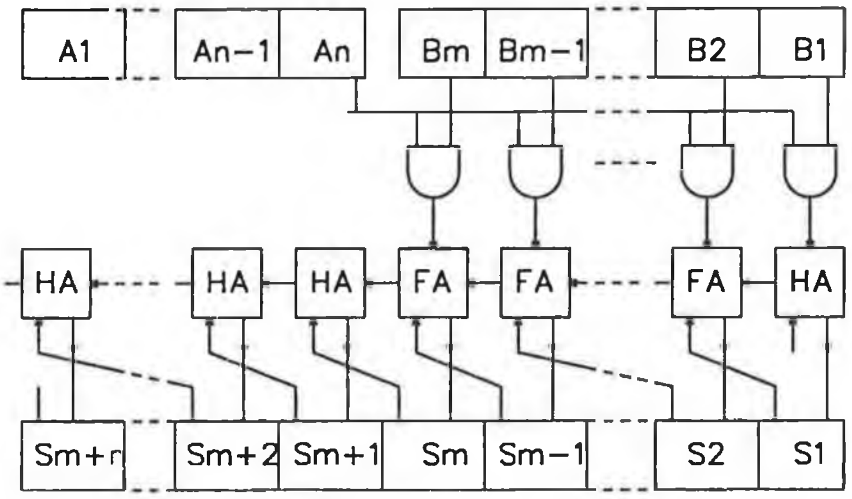
CI : Carry in
CO: Carry out

Fig. (5.1) Serial Multiplier

A = (10110)
B = (1011)

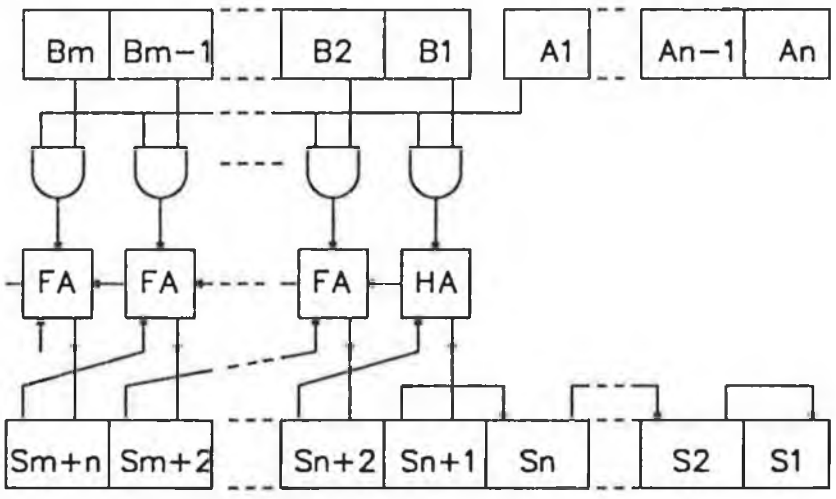
Clock Cycle	B	R								
	1									
1		0	0	0	0	0				
2		1	0	0	0	0				
3		1	1	0	0	0				
4		0	1	1	0	0				
5	1	1	0	1	1	0	0			
6										
7										
8										
9										
10	0	1	0	0	0	0	1	0		
11										
12										
13										
14										
15	1	0	1	0	0	0	0	1	0	
16										
17										
18										
19										
20		1	1	1	1	0	0	1	0	Result

Fig. (5.2) Serial Multiplier Example



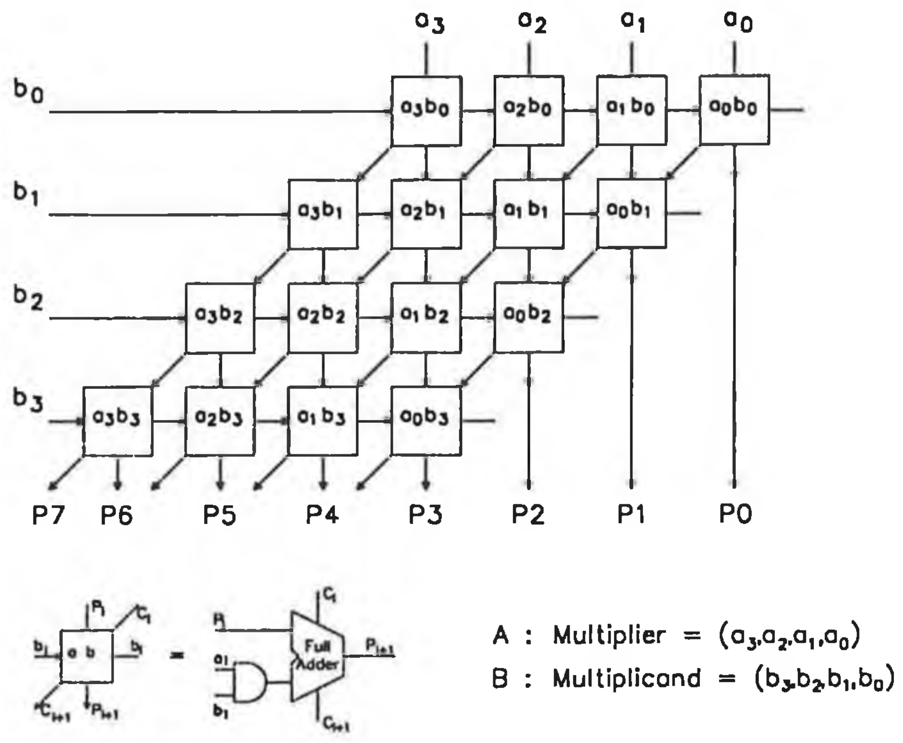
A: Multiplier (n bits)
S: Result (m+n bits)
B: Multiplicand (m bits)
FA : Full Adder
HA: Half Adder

Fig. (5.3) Horners multiplication method



A: Multiplier (n bits)
S: Result (m+n bits)
B: Multiplicand (m bits)
FA : Full Adder
HA: Half Adder

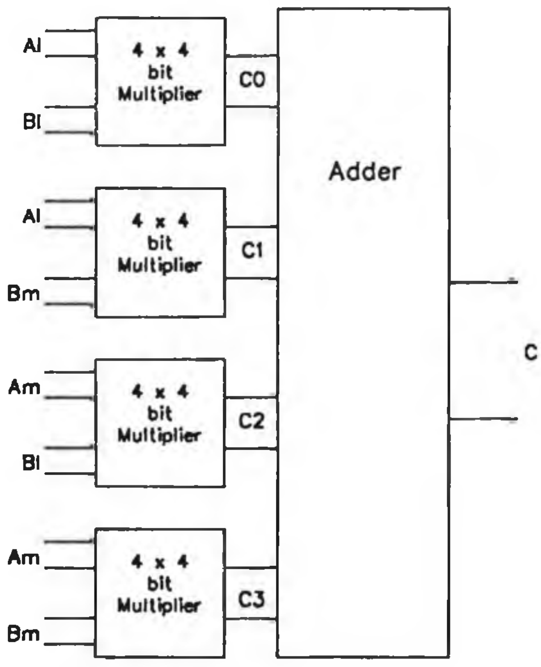
Fig. (5.4) Right to left factorisation



A : Multiplier = (a_3, a_2, a_1, a_0)

B : Multiplicand = (b_3, b_2, b_1, b_0)

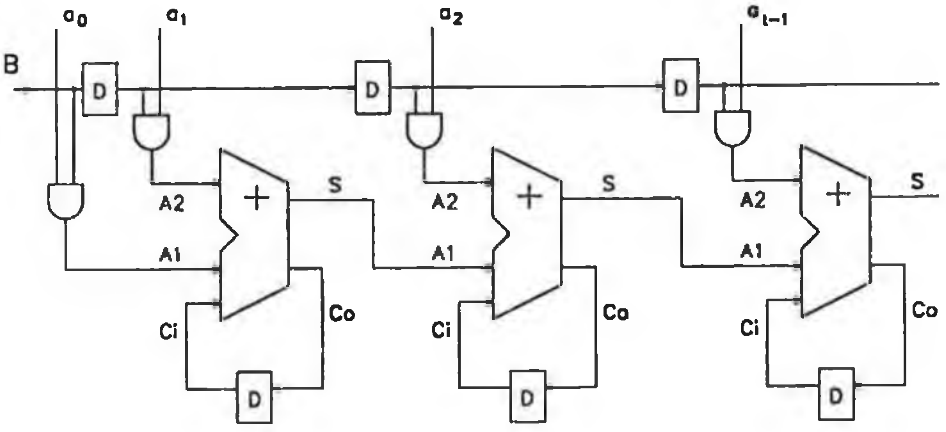
Fig.(5.5) Parallel Multiplier



C = A•B

A = (Al, Am)
B = (Bl, Bm)
Al : LS nibble of A
Am : MS nibble of A
Bl : LS nibble of B
Bm : MS nibble of B

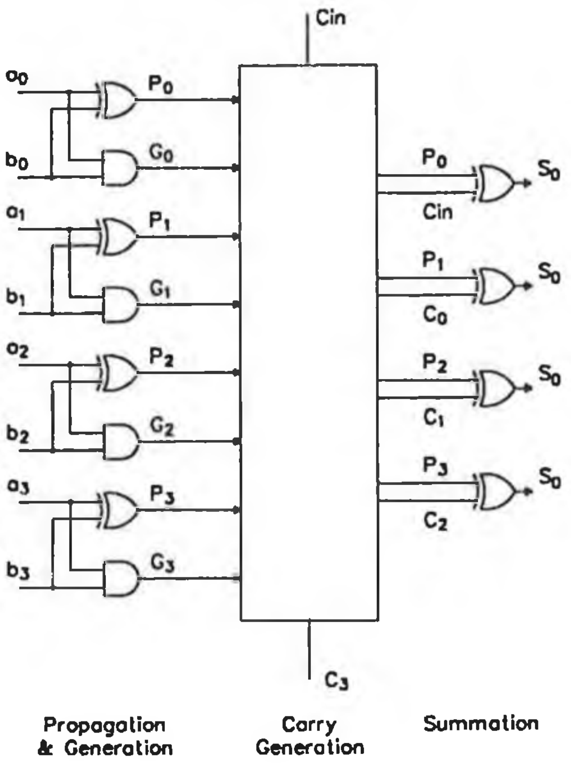
Fig. (5.6) 8x8 bit multiplier using 4x4 bit devices



A : Multiplier = $(a_{l-1} \ldots a_0)$
B : Multiplicand = $(b_{s-1} \ldots b_0)$

+ : Adder
D : D-type register

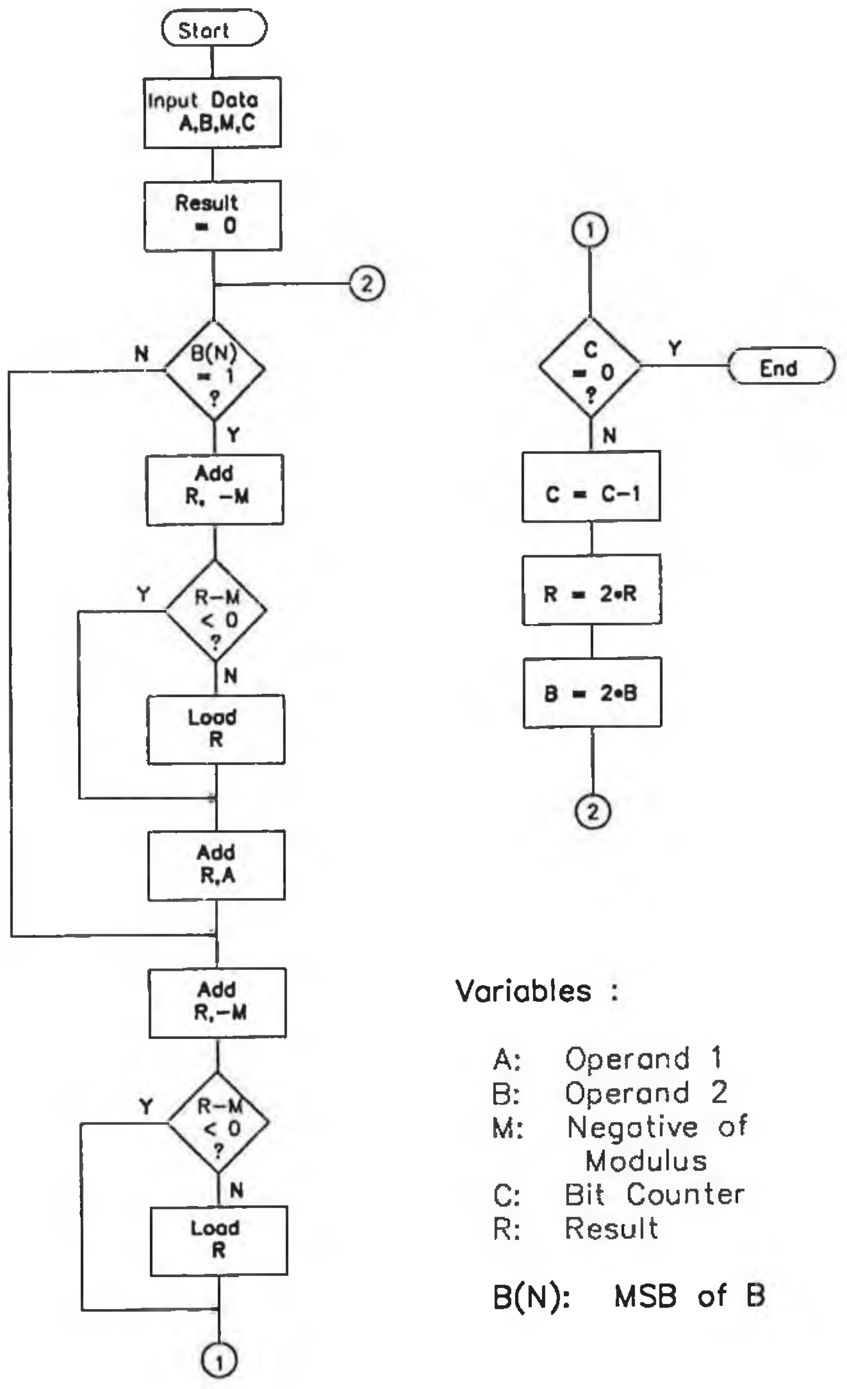
Serial/Parallel Multiplier



$$
\begin{aligned}
C_0 &= P_0 Cin + G_0 \\
C_1 &= P_1 P_0 Cin + P_1 G_0 + G_1 \\
C_2 &= P_2 P_1 P_0 Cin + P_2 P_1 G_0 + P_2 G_1 + G_2 \\
C_3 &= P_3 P_2 P_1 P_0 Cin + P_3 P_2 P_1 G_0 + P_3 P_2 G_1 + P_3 G_2 + G_3
\end{aligned}
$$

Look-ahead carry adder (4 bits)

fIG.(5.7) Serial/Parallel Multiplier (Carry look-ahead adder)

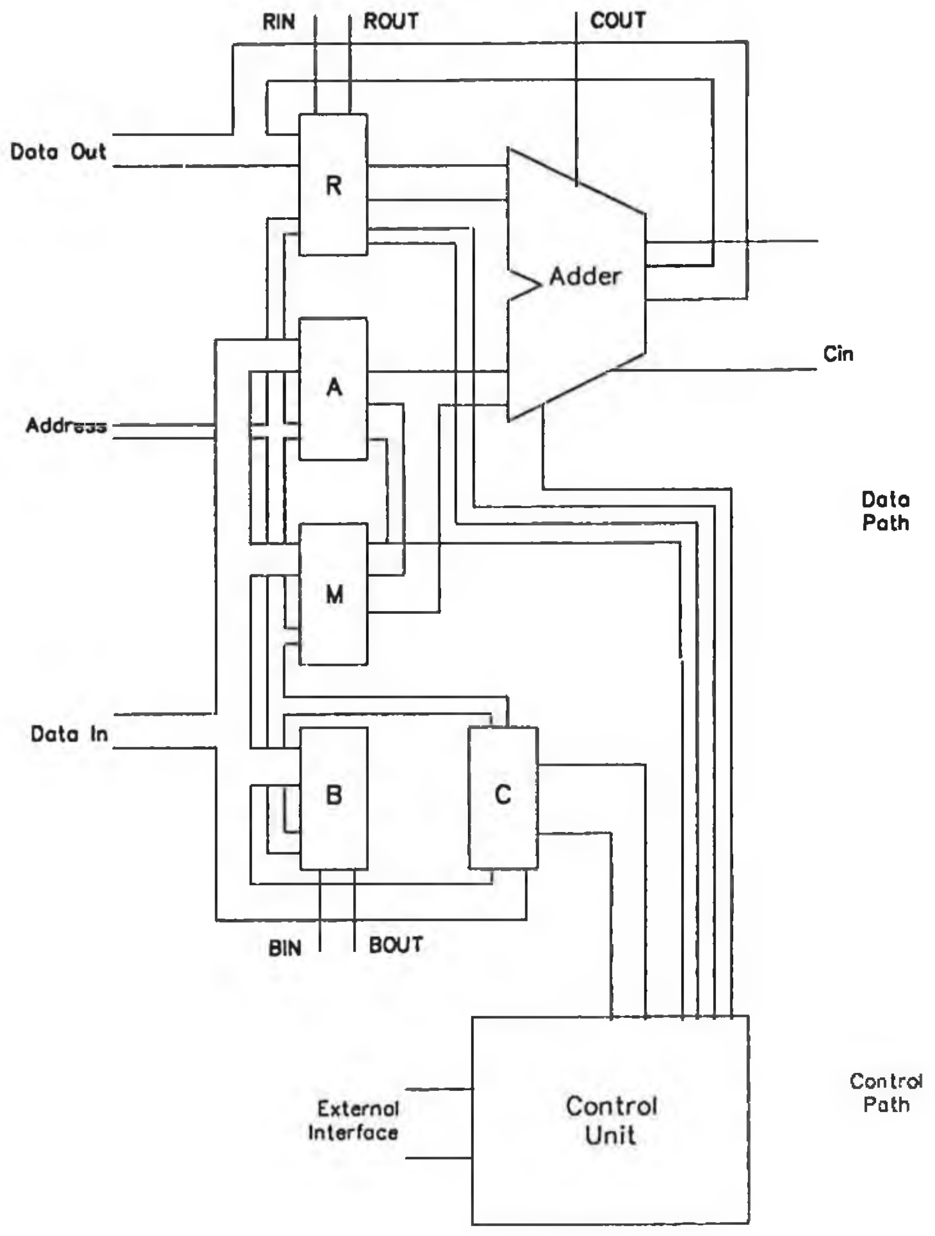


Variables :

A:	Operand 1
B:	Operand 2
M:	Negative of Modulus
C:	Bit Counter
R:	Result

B(N): MSB of B

Fig. 5.8 Modified Blakely algorithm



A : Operand 1
B : Operand 2 (Control)
M : Modulus
R : Result = AB Mod M
C : Bit counter

Fig. (5.9) Simplified block diagram

6. Semi-Custom Implementation

6.1 DESIGN CRITERIA

The function of the modular multiplication device (MMD) is to perform the modular multiplication required for the RSA public key cipher using Blakelys algorithm. The device was designed with regard to certain criteria. The first of these was that it should interact with the rest of the encryption algorithm and to this end an interface to standard 8 bit microprocessors was required. the second criteria was that the use of different key lengths to provide different levels of security was also required. It had to be possible, therefore, to easily change the bit length that could be used with the MMD.

6.2 DESIGN PROCESS

The design of semi-custom circuits at the NMRC involves a number of design stages as shown in Fig. 6.1. The major design stages are :

. Concept
. Block Design
. Logic Design
. Circuit Design
. Layout
. Fabrication

These design stages will be discussed in greater detail below.

6.2.1 Concept

The first stage of any design work is the concept or idea stage. In most cases where semi-custom implementation is considered a circuit has already been designed and implemented using standard logic and thus the idea has been proven. In other cases where the circuit is to be implemented directly on silicon, without a prototype having been constructed using standard logic, the idea is not proven.

Both of these cases require that at each stage of the design process the design is simulated and tested to ensure that it meets the requirements placed on it. If a circuit has already been prototyped this can be used to predict and compare responses obtained from simulation. Circuits which are to be implemented without a prototype require that results from higher level simulations be used for comparison and prediction of how the final circuit will behave.

6.2.2 Block Design

The block design requires that the system to be designed is broken into a number of functional units, e.g storage, interface, computation and control , and that the interface between and function of these units is defined. At this stage it is necessary to consider how the system can be divided if it is too large to fit on a single die.

Testing of the block design is carried out using a register transfer simulator. This deals with units at a functional or logic level. The input to the register transfer simulator is a description of the circuit written in a hardware description language (HDL). These simulators are event driven and can therefore not be used for detailed timing analysis. They are used to test the validity of the design and can also provide test vectors for use in later simulation and testing.

6.2.3 Logic Design

The logical design is carried out for each of the functional units described in the block design. This involves designing the function units using existing logical functions e.g AND, OR and Flip-Flops. There are many tradeoffs which can be made at this stage. The major tradeoff is between size and speed. Implementation of functions such as adders or counters using serial techniques will result in a large saving in space but will affect the speed. Similarly if these devices are implemented using parallel techniques the speed will increase but so too will the physical size of the device. A compromise must be reached between the two techniques. The result of this stage should be a complete circuit diagram at a logic level of each of the function units.

In the design of the modular multiplication device the width of the data path, i.e the number of bits that the device can handle, was not known at this stage. Because of this all function units that were dependent on the data path had to be designed on a modular basis to allow easy expansion when the size of the data path became known.(A3)

Testing of the logical design is carried out using the register transfer simulator. This is carried out at two levels, the testing of each of the function units individually and the testing of the complete device.

6.2.4 Circuit Design

Circuit design is the creation of cells which are necessary for the function units. These cells are created using elements from the standard cell library available at the NMRC. Elements that are not present in the standard cell library have to be designed. The cells are designed on a bit basis, e.g a one bit shift register , and the inputs and outputs to each cell placed so as to allow abutment of cells to produce longer word lengths. Functions which are independent of the data path can be designed completely at this stage.

The result of this stage is a series of modules, at a higher level than the standard cell elements e.g a shift register cell or a counter cell, which can be used to produce the function units necessary. The size of the data path can also be decided at this stage, this will be discussed later.

Before testing of the cells can be carried out they must first be extracted. The cells consist physical shapes at different levels e.g polysilicon, metal and silicon dioxide. These shapes must be converted back to electrical devices such as resistors and transistors. A programme which carries out this function is called an extractor. The extractor uses relationships between the different layers to deduce the presence of transistors etc. The electrical description output by the extractor , which will include parisitic capacitances and diodes, will be in a format which can then be used as input to a circuit simulator.

Testing of the circuit design is performed using a circuit simulator. The two used at the NMRC are SPICE, a public domain programme developed at the University of California at Berkely, and SIMON, developed and supplied by ECAD. Both of these simulators allow detailed timing analysis to be carried out. SIMON and SPICE are compatible in many ways, they can both accept the same input file. However SIMON is optimised for digital logic circuits and is considerably faster than SPICE when used on these circuits. These simulators provide accurate predictions of time delay and waveforms accepted from circuits but they are very CPU intensive. The time delay involved in simulating circuits of reasonable size is considerable.

Simulation of the circuit design is carried out at two levels. The first level is a test of the cell design. The second level involves connecting cells to obtain function units and simulation of these units

Circuit design of cells can be carried out using an interactive layout editor. An example of such an editor is KIC2, a public domain product, which is available at the NMRC and at N.I.H.E Dublin.

6.2.5 Layout

Layout is the actual physical design of the integrated circuit using the cells developed in the previous stage. The width of the data path is known at this stage and thus the physical size of all the function units can be estimated. The first step in the layout process is to develop a floor plan. This will show the location of all of the function units in the device and the space occupied by each of them. Sufficient space must also be left for routing of interconnections between the function units. The floor plan will be hierarchical in nature in that function units will consist of cells which in turn will consist of elements from the standard cell library._This provides for flexibility in the design of the floor plan as the shape of the function units can be changed, by rearranging the cells within it, and thus there are many possible floor plans.

When the floor plan has been completed each of the function units is constructed from its cells so as to fit the space allocated to it. The function units are then placed in their appropriate places and interconnections between them are made. It is good practice to have all inputs and outputs to the function units named as this will enable connections to be automatically checked. Interconnection is carried out using metal and polysilicon layers. Layout is performed on an APPLICON CAD system at the NMRC.

Testing of the completed layout on a physical and an electrical basis is required. On the physical level it is necessary to ensure that the layers and shapes conform to the design rules at the NMRC. These design rules specify the minimum size of shapes on different layers and the minimum distances between shapes. The circuit is tested, to ensure that there are no design rule violations, using programmes developed at the NMRC.

Electrical testing of the layout is in two forms. The first is an electrical rule check which ensures that there are no electrical rule violations, e.g power and ground rails connected together or signals with different names connected to one another. The second is the use of a circuit simulator to test the final circuit. Before this can be done the circuit must be extracted. The use of circuit simulators at this level is very time consuming. For this reason emphasis is placed on testing at the function unit level and on the electrical rule checker.

The result of this stage is geometrical representation of the integrated circuit at different levels.

6.2.6 Fabrication

After the layout has been completed masks are made of the different layers. This involves sizing the layout to its correct size and making positive or negative masks, as appropriate, of each layer. These masks are used in the fabrication process. The process used at the NMRC was a 5 micron CMOS p-well process.

6.3 SYSTEM OVERVIEW

The device can be divided into two main sections, the Data path and the Control path. The Data path contains the registers for storing the variables , the adder and the interface to the microprocessor. The Control path contains the control sequence for implementing Blakelys algorithm. it should be noticed that the width of the Data path depends on n, the keylength while the control path is essentially independent of this. (A2)

It was not expected that a modular multiplication device of sufficient size (300 bits) would fit on a single integrated circuit. It was thus necessary to divide the device into sections which could be cascaded to produce the complete device. Two options were considered :

1. Separation of Control and Data paths.
2. Master/Slave device.

The first option involved the design of two separate integrated circuits, a control circuit and a data circuit. The control circuit would contain the finite state machine to implement Blakelys algorithm while the Data circuit would contain the various registers and adders required. Again it was expected that the complete data path could not be contained in a single device, so several identical data circuits would be required to produce the necessary wordlength.

The Master/slave approach differs from the first option in that the control and data paths would not be separated. Each device would be autonomous, containing its own control and data path. As a single device would not be sufficient for the keylengths being used these devices could be cascaded. When the devices where cascaded the most significant device, i.e the device which contained the most significant bits of the Data path would be the 'Master', all other devices being 'Slaves'. The 'Master' device would make all the decisions e.g whether the result was greater than the modulus, and pass these to the Slaves. There would be no physical difference between Master and Slave devices, each would be selected by the level on an external pin. A diagram showing several of these devices cascaded is shown in FIG.(6.3). The labels used in this diagram will be explained later.

This method was chosen as it does not require that different devices be produced. The time and resources necessary to produce two different integrated circuits could not be justified.

6.3 SYSTEM DESIGN

The proposed device, shown in FIG.(6.4), consists of several subunits. These subunits are:

1. Interface unit.
2. Storage unit.
3. Computation unit.
4. Cascade and Master/Slave unit.
5. Control Unit.

The interface unit provides the interface between the modular multiplication device and external circuitry. The storage unit contains registers to hold the input variables required by the device and storage for the result obtained. The computation unit performs all the calculations necessary to implement Blakelys algorithm. The cascade and master/slave unit allows for cascading of modular multiplication devices to produce longer wordlengths. The control unit is a finite state machine which controls all the other hardware in the device to perform Blakelys algorithm.

6.3.1 Interface Unit

The purpose of the interface unit is to allow data to be transferred between the modular multiplication device and external devices. This interface had to be compatible with standard 8 bit microprocessors. The interface consists of an address bus, a data bus and a control bus.

The data bus is an 8 bit wide bi-directional tri-state bus. Data is loaded into the internal 16 bit registers 8 bits at a time.

The address bus is an input only port to the device which selects the appropriate register and byte within that register. The size of the address bus is 3 bits. This allows the 3 sixteen bit input registers, the sixteen bit output register and the 8 bit counter to be addressed. The input registers are write-only and the output register is read only. The modular multiplication device appears to the external microprocessor or other circuitry as a block of contiguous memory locations.

The control bus is a collection of control signals necessary for the synchronisation of data transfer and for proper operation of the device. This bus contains the clock, the read/write signal, the chip select signal and reset.

The diagram of the interface unit is shown in Fig. 6.5. It contains an input data selector, an output data selector and address decoding. The input and output data selectors are provided by means of multiplexors. Address decoding is implemented using a 3 to 8 line decoder.

Connections between the device and the external world are performed by means of input and output buffers. These differ from internal buffers in their ability to drive large capacitances. The input and output buffers were placed around the edge of the integrated circuit.

6.3.2 Storage Unit

The storage unit contains four registers to hold the operands A and B, the modulus M and the result R and also the bit counter C.

Registers A and M are simple storage registers with tri-state output. The outputs of these registers are connected to one of the adder input busses.

Register B, also called the control register, is a parallel in serial out shift register. The only bit of interest in register B is the most significant bit which is used by the control algorithm to select between two alternative actions.

Registers A,B and M are loaded from the external circuitry via the interface unit.

The result register R is a parallel in parallel out shift register. Register R is loaded from the output of the adder. The output of the R register provides the second operand for the adder. It can also be read by external circuitry via the interface unit. It must also be possible to clear register R when required.

The bit counter C is an 8 bit down counter. It is loaded by the microprocessor with the keylength. This is then used by the control algorithm to determine the number of iterations of the control algorithm necessary. The counter is decremented by the control unit at each iteration until it is zero. The output from the counter is a signal which indicates to the control unit that it has reached zero.

Registers A, B and M and counter C appear as write only memory location to the external microprocessor. Register R appears as read only. The microprocessor can not, therefore, read the contents of A, B or M and cannot write to R.

Logic Design

As the width of the data path was not known the design of the storage unit was accomplished on a modular basis. Single bit registers and counters were designed so that they could be linked together to form longer wordlengths.

A logic diagram of a simple register cell used to implement registers A and M is shown in Fig. 6.6a. The register used is a standard level triggered D-type.

The circuit design of the A and M registers was complicated by the lack of a tri-state buffer in the cell library at the NMRC. This was developed by the NMRC for this application. The circuit design of registers A and M is shown in Fig. 6.6b. The basic cell consists of three standard cells, a D-type register, a tri-state buffer and a via. The via is included to allow the data bus to pass through the register and onto other registers. There are control signals for reseting and loading the register and and for enabling the output buffers. The register cells were stacked together to form the complete registers.

The shift register cell used for registers R and B is shown in Fig. 6.7a. These cells are fully synchronous in operation. Connection of these cells to form longer registers requires that an extra circuit, which provides the control signals, is added for every 8 cells. A logic diagram of this circuit is shown in Fig. 6.8a.

The shift register cell is shown in FIg. 6.7b. the use of multiplexors in this design reduces the area when compared to standard logic. Again vias are included in each cell to allow the data bus to pass through the register. The circuit required for each octet of shift register cell is shown in Fig. 6.8b. This provides the control signals for the eight cells.

Several options for the implementation of the 8 bit down counter, C, were considered. These include :

. PLA Implementation
. Parallel feedforward
. Serial feedforward
. Asynchronous Implementation

The use of a PLA for implementing the counter offered advantages in terms of ease of layout. A parallel feedforward counter would provide the fastest operation at the expense of area. The asynchronous implementation is the slowest of the counters considered but occupies the smallest area. The serial feedforward counter is a compromise in terms of speed and size and for this reason it was decided to use this type of counter. A logic diagram of a single down counter cell is shown in Fig. 6.9a.

The implementation of the counter cell is shown in Fig. 6.9b. Again the extensive use of multiplexors reduces the physical size of the devices. These cells can be cascaded to produce a down counter of the required length. A circuit to determine when the contents of the counter are zero must also be included.

6.3.3 Computation Unit.

The computation unit contains the adder. This is the only computation required for Blakelys algorithm. The inputs to the adder come from register operand register A or modulus register M and from the result register R. In addition to this there is a carry input which comes from off chip and an enable input. The carry input enables the cascading of devices to produce longer wordlengths and is obtained from previous stages. The enable signal comes from the control unit and informs the adder that the two inputs are to be added. The output of the adder is a 16 bit result, which can be loaded into register R, and a carry out which is passed to subsequent stages.

The design of the adder was carried out by the NMRC as it did not already exist in the cell library. There are many possible ways of implementing adders and all methods provide different advantages and disadvantages. The method chosen is called a Manchester carry chain adder and offers a compromise in terms of speed and size.

The principle of the Manchester carry chain type adder is shown in Fig. 6.10. When the adder enable signal is low the carry out signal is pulled high by the P-transistor, Q1. When the adder enable signal goes high n-transistor, Q2, is turned on. If the carry generate signal is high, ($A_i.B_i$, where A_i and B_i are input bits to the adder), then transistor Q3 will turn on discharging the carry output node. If the carry propagate signal (A_i+B_i) is high then transistor Q4 is turned on enabling the carry from the previous stage to be propagated through to the next stage.

The adder was designed as a 4 bit cascadable unit. The logic diagram and circuit layout of a 4 bit module are shown in Fig. 6.11a and 6.11b respectively.

6.3.4 Cascade and Master/Slave Unit.

The Cascade and Master/Slave consists of a number of small circuits. The signals needed for cascading of devices are the inputs and outputs of the shift registers R and B i.e Rin, Bin, Rout, Bout and the carry in, Cin, and carry out, Cout, of the adder. The adder poses a problem when cascading as the time required for addition increases as devices are cascaded. Sufficient time must be allowed for addition and as there is no way of knowing how many devices will be cascaded the total addition time is not known. Several solutions to this problem of addition time were considered:

1. Set a maximum limit on the number of devices which can be cascaded. This will determine a maximum addition time which can be used in all cases.
2. Determine the worst case addition time for a single device. Multiply this by the number of devices in cascade and use this to determine the maximum clock speed that can be used with the device.
3. Construct an extra register on chip which can be programmed with the number of devices to be cascaded. This programming can be done at manufacture or preferably by the end user. The required addition time is then some function of the number stored in the register. Alternatively the bit count register C could be used to determine the addition time. C contains the number of bits in the keylength which is N by the number of devices, where N is the data path width per device.
4. Generate an Addition Complete signal on chip. This signal is then passed to the next chip where it is ANDed with its Addition complete signal and passed to

the next chip and so on. The Addition Complete signal from the most significant chip is then used as an Addition Finished signal which is passed to all chips.

The first option while easiest to implement has many disadvantages. The total addition time would need to be the worst case for the largest possible number of cascaded devices. Use of the devices for smaller keylengths would not produce any increase in speed.

The second option would allow the user to increase the speed of addition for short keylengths. However the use of the clock as a controlling device poses some problems. The first is that in most microprocessor systems the clock is crystal controlled and as such cannot be varied. Fractions of the clock could be obtained using dividers but this requires extra circuitry. Another problem is that there are clock cycles within the device during which addition is not performed e.g shifting of registers etc. It is not necessary to slow these down fo long keylengths and choosing the clock speed to suit the adder delay would do this, resulting in a significant increase in multiplication time.

The use of an extra register to hold the adder delay or the derivation of that delay from the bit counter overcomes many of the disadvantages of the previous options. The register would have to be user programmable to maintain flexibility. The use of the bit counter would make the control sequence more complex and hence it would require more space.

The generation of an Addition Complete signal on chip differs from previous options in that previous knowledge of the number of cascaded devices is not required. the total addition time is the time taken for the Addition Complete signal to propagate from the least to the most significant device. The main advantage of this is that the add time is automatically suited to the number of devices. Its advantage over the previous method of using the bit counter is that the control sequence is not as complicated and also that the Addition complete signal does not have to be based on the worst case addition time and should thus be faster.

It was therefore decided to chose the final option to determine the addition time. This resulted in an extra three pins being required by the device, Addition Complete In, AIN, Addition Complete Out, AOUT, and Addition Finished, ADFIN. The addition finished signal, ADFIN, is used by the control unit to determine when addition is complete.

The logic diagram of the circuit used and the circuit layout are shown in Figs. 6.12a and 6.12b.

The Master/Slave unit consists of a number of multiplexers controlled by the Master/Slave signal. The first multiplexer selects between the most significant bit of register B, BOUT, and the signal DIN. The control unit needs the most significant bit of register B on which to base a decision. If the device is in Master mode then BOUT is the most significant bit and this is passed to the Master control unit and to all other devices via the DOUT signal. If the device is in Slave mode then the most significant bit of B is obtained from DIN, which is connected to DOUT of the Master device. In Slave mode the device ignores the most significant bit of its own B register.

The second multiplexer is used in the determination of sign. The sign of the result is determined by the most significant bit of the Adder. Therefore if the device is in Master mode it uses this bit as the sign bit. The master device also sends the sign bit to all Slave devices via the Sign Out, SGNO, signal. If the device is in Slave mode it ignores the most significant bit of its own adder and obtains the sign bit from the Sign In, SGNI,signal. The cascading of of Master and Slave devices to produce long wordlengths is shown in FIG. (6.3).

6.3.5 Control Unit

The Control unit is a finite state machine which controls all the other hardware in the device. The state machine implements Blakelys algorithm as can be seen fom the state diagram of FIG. 6.13. The number of internal states is 16 , hence requiring 4 state variables. The control unit also has 5 inputs and 8 outputs which are :

Inputs :

ADFIN	The Addition Complete signal from the Master device. This is used to determine when addition is finished so that the control sequence can continue.
SIGN	The Sign Bit which is also obtained from the Master device.
CZ	An signal produced by the bit counter, C, to indicate that its contents are zero.
BI	The most significant bit of the control register, B. This is again obtained from the Master device and passed to all Slave devices.
RUN	The signal which starts the state machine and hence Blakelys algorithm.

Outputs :

ADEN	Adder enable.
SAM	Select A or M. Selects the output of either the operand register A or the modulus register M as input to the adder.
LDR	Loads the result register from the output of the adder.
SR	Shifts the contents of the result register R left by one bit.
SB	As above but for the B register.
CLRR	Clear the contents of the result register R to zero.
DEC	Decrement the bit counter C by one.
BUSY	Signal which indicates that the algorithm is in progress and that the device is busy.

The state diagram described above implements Blakelys algorithm on a serial/parallel multiplier as described in the previous chapter, however several points are worth noting. There are two major sequences that are carried out. These are the addition of A and R, and the addition of R and -M. The test to check if the result is greater than the modulus is performed by subtracting M from R. If the output of the adder is negative then R was less than M and no further action is taken. If however the result was positive then R was greater than M and so it is replaced by the output of the adder.

The control unit was implemented using a programmable logic array(PLA). This produced two advantages over a dedicated control unit, size and the ease of modifying the control algorithm should that be necessary. (A1)

A block diagram of the PLA used is shown in Fig. 6.14. This consists of an AND plane, an OR plane and product terms. Inputs are made to the AND plane and outputs are taken from the OR plane. Programming of the device is carried out by placing transistors at the junction of the AND/OR plane and the product terms. Feedback from the output to the input is used to implement a state machine. The size of the PLA is determined by the number of inputs, the number of outputs and the number of product terms. The logic equations resulting from the state diagram were reduced both manually and automatically to minimise the size of the PLA.

The PLA used is a dynamic type requiring a two phase clock. Programming of the device requires that transistor cells be placed in the appropriate location.

6.4 DEVICE PERFORMANCE

The speed of the modular multiplication device is limited by the carry propagation delay of the adder. Let the carry propagation delay of a single device be T_A. As can be seen from FIG. (6.13) the control algorithm has 16 possible internal states, of which 14 lie within the main programme loop. Three additions are performed per main loop, thus the loop time is :

$$14T_A+(3N_B/2)T_A$$

where T_A is the minimum clock period, determined by the carry propagation delay and N_d is the keylength in bytes. As each device can accomodate 16 bits $N_B/2$ is the number of devices required for a given keylength.

The main programme loop is carried out $8N_B$ times. Initialisation of the device requires that 7 bytes be loaded into each device, two each for the operand A, the control register B, the modulus register M and one for the bit counter C. The total time for the modular multiplication is therefore :

$$[\ (7N_B/2) + 8N_B(14 + (3N_B/2))\ ]T_A$$
$$= (12N_B^2 + 116N_B)T_A$$

Simulaion results from the adder show a maximum carry propagation delay of several hundred nanoseconds. As the device is designed to work with standard 8 bit microprocessors a 1MHz clock could be used. This would produce a multiplication time of :

$$(12N_B^2 + 116N_B) \text{ microseconds}$$

A table showing the multiplication time for various values of N_B is shown in FIG. (6.15).

Fabrication of the device by the NMRC has just been completed and testing to verify the above figures has now to be started.

6.5 ENCRYPTION USING THE MMD

The modular multiplication device can be interfaced to standard 8 bit microprocessors in the same way as memory. The only additional requirement is that two bits of i/o are necessary , one for starting the device (RUN), the other for determining that the result is ready (BUSY). The devices can be cascaded to produce the desired wordlength, the ultimate limitation being the carry propagation delay of the adder and the size of the bit counter C. The time required for exponentiation, as shown in chapter 2 is given by :

$$T_e = 53 + 2583N_B + 2432N_B^2 + 32N_B[T_{Mod}[N_B]] \text{ cycles}$$

where $T_{Mod}[N_B]$ is the number of clock cycles for multiplication of two N_B byte numbers modulo a third N_B byte number. Substituting for the multiplication time of the modular multiplication time into the above equation , and assuming a clock frequency of 1MHz, yields :

$$T_e = 53 + 2583N_B + 2432N_B^2 + 32N_B[12N_B^2 + 116N_B]]$$
$$= 53 + 2583N_B + 6144N_B^2 + 384N_B^3$$
$$\text{microseconds}$$

A table showing the exponentiation time for various values of N_B is shown in Fig. (6.16).

For an 80 byte (approx. 200 decimal digits) the use of the hardware device results in an exponentiation time of approximately 240 seconds. The fastest software algorithm (reciprocal), described in chapter 3 has an exponentiation time of 2500 seconds. An order of magnitude increase in encryption speed can therefore be expected.

6.6 FURTHER ENHANCEMENTS

The modular multiplication device described here was implemented in 5 micron CMOS technology at the NMRC. Since then the NMRC has introduced a 3 micron process. This would provide a 2.5 fold increase in the space available on chip with a subsequent increase in the number of bits per device. There are some commercial processes which are at 1.2 micron densities. This would provide a 17 fold increase in area.

The design of the circuit could also be changed to eliminate the need for loading the operands into the device prior to each multiplication. In the modular exponentiation algorithm described in chapter 2, only two type of modular multiplications are performed , T = T*T and C = C*T. The storage of the partial results C and T on-chip may be possible and this would increase the exponentiation time.

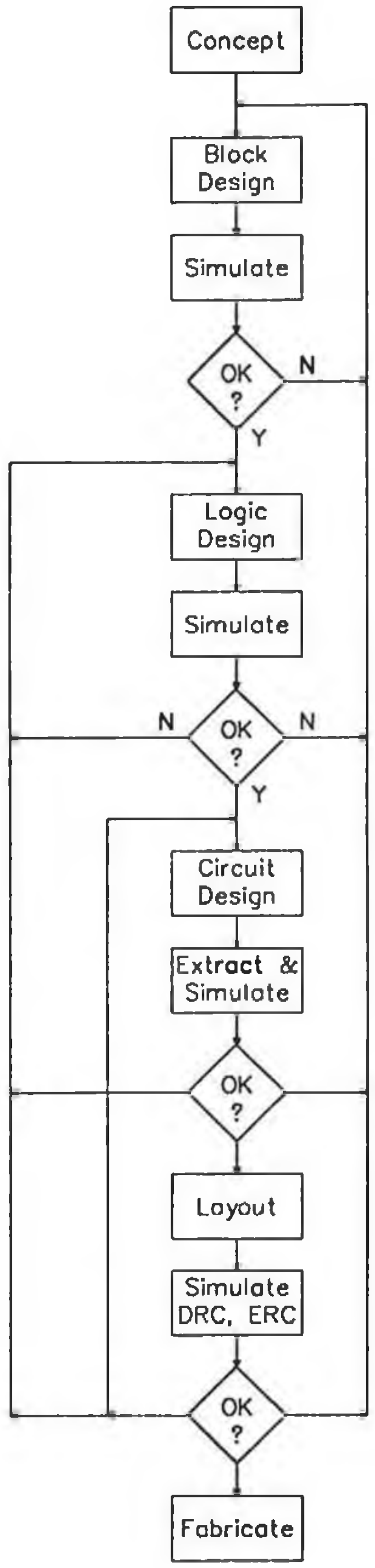
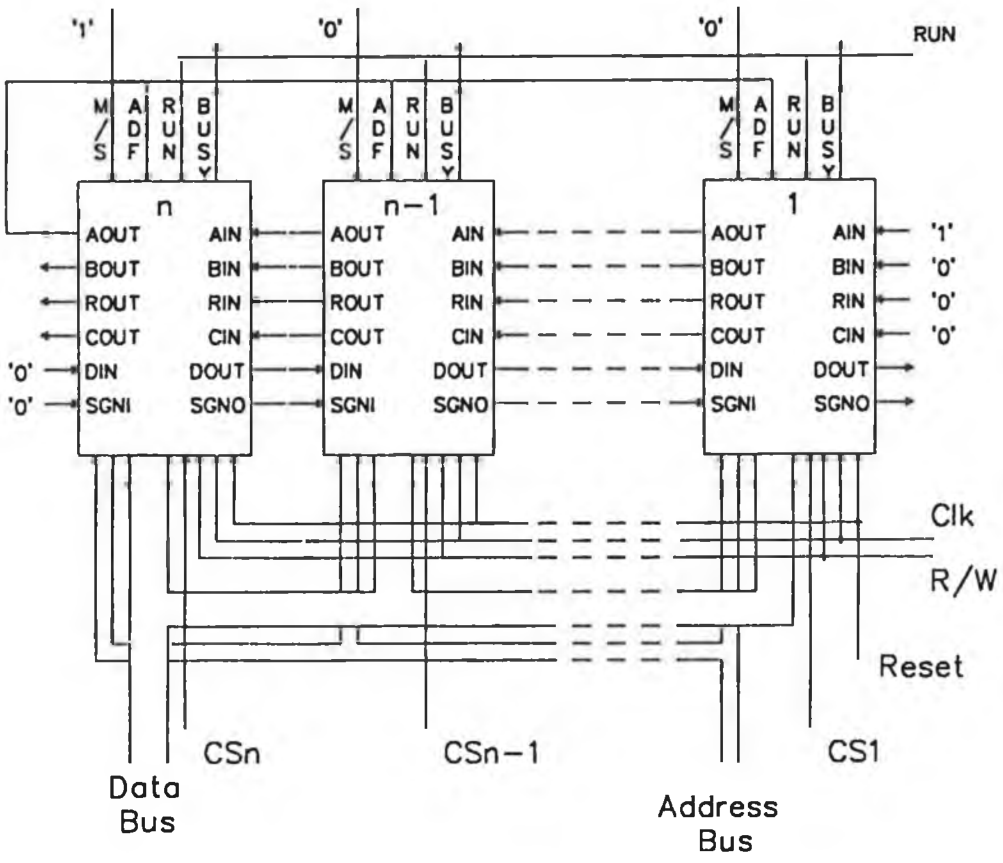
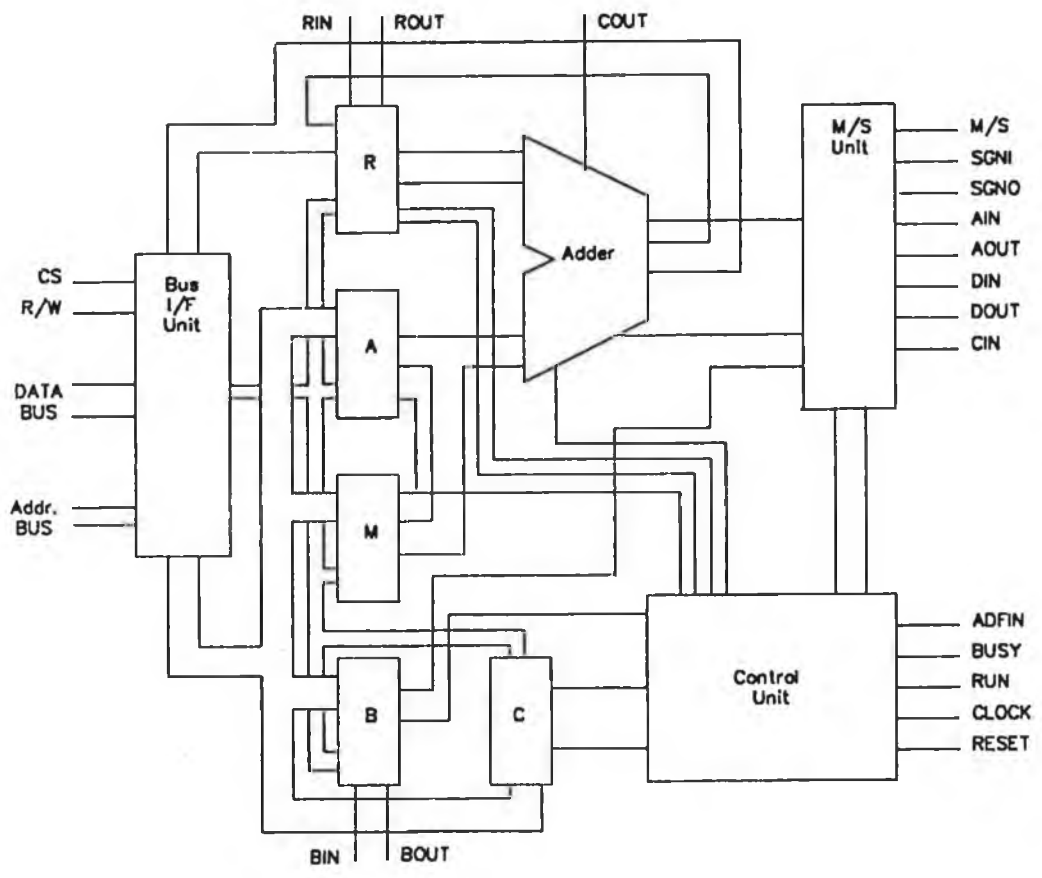


Fig. (6.1) Integrated Circuit Design Process



AOUT	:	Addition complete out
BOUT	:	MSB of control register B
ROUT	:	MSB of result register R
COUT	:	Carry out from adder
DIN	:	MSB of B from Master Device
SGNI	:	Sign input
SGNO	:	Sign output
DOUT	:	MSB of B from Master Device
CIN	:	Carry in from previous stage
RIN	:	MSB of R from previous stage
BIN	:	MSB of B from previous stage
AIN	:	Addition complete from previous stage
R/W	:	Read/write signal from microprocessor

Fig. (6.3) Cascading of devices



Registers:

A:	Operand 1
B:	Operand 2
M:	Modulus
R:	Result
C:	Bit Counter

Fig. (6.4) Modular Multiplication Device

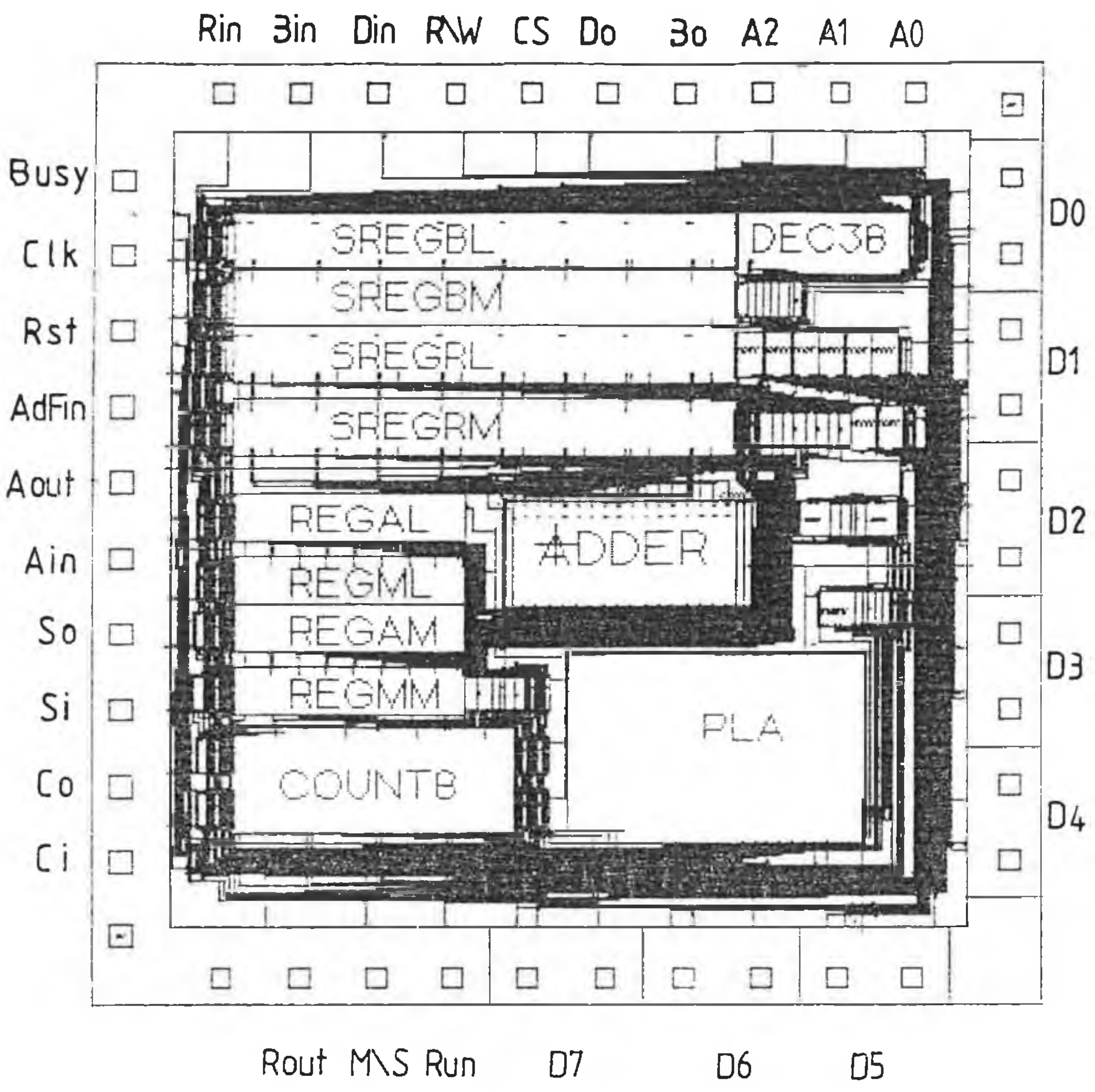
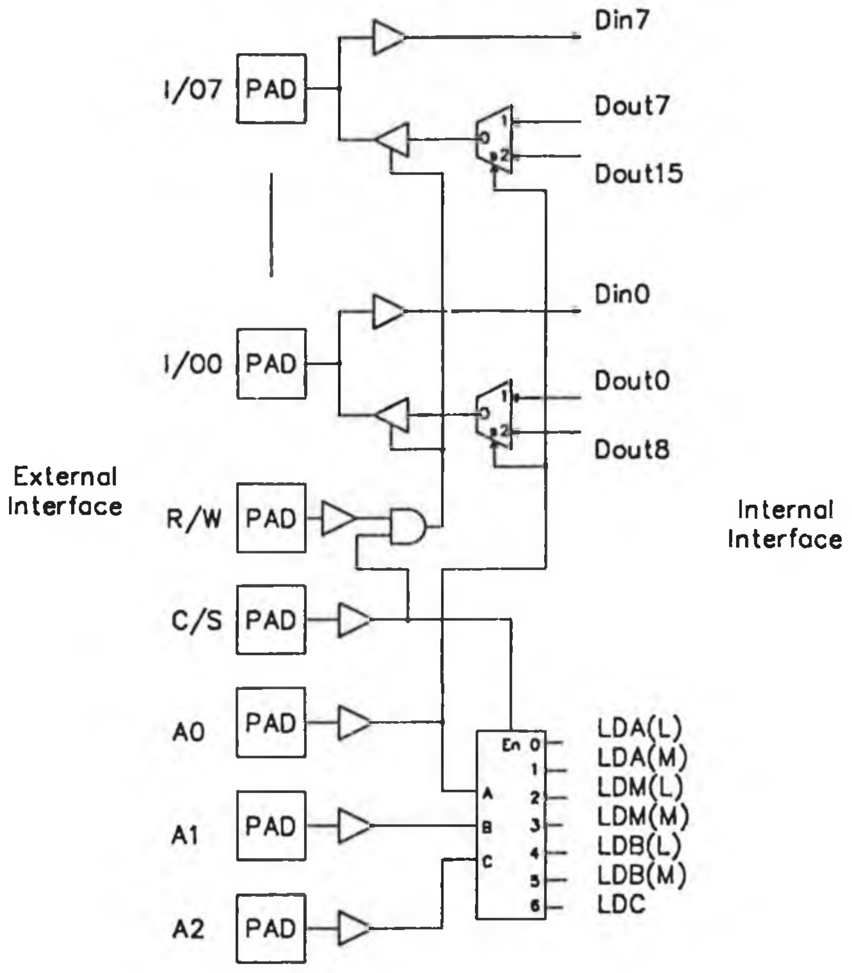


Fig. (6.4a) Chip layout - Floor Plan



A0,A1,A2	:	External address bus
I/O0..I/O7	:	External data bus
R/W	:	Read/Write signal
C/S	:	Chip select
Din0..Din7	:	Internal input data bus
Dout0..15	:	Internal output data bus
LDx(L)	:	Select least significant byte of x
LDx(M)	:	Select most significant byte of x

Fig. (6.6 Interface Unit

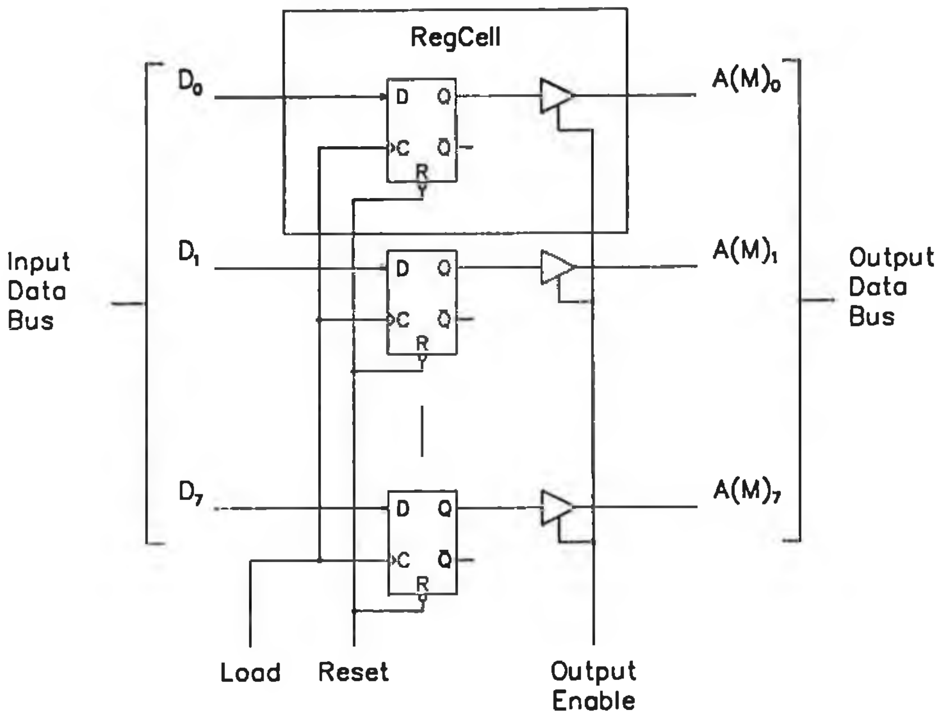


Fig. (6.6a) Logic diagram – Registers A and M

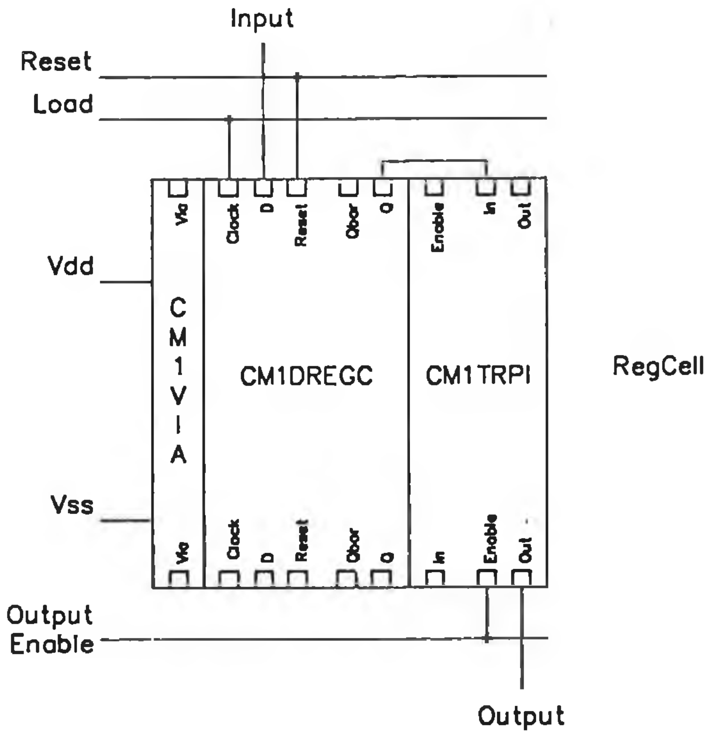


Fig. (6.6b) Layout of simple register cell

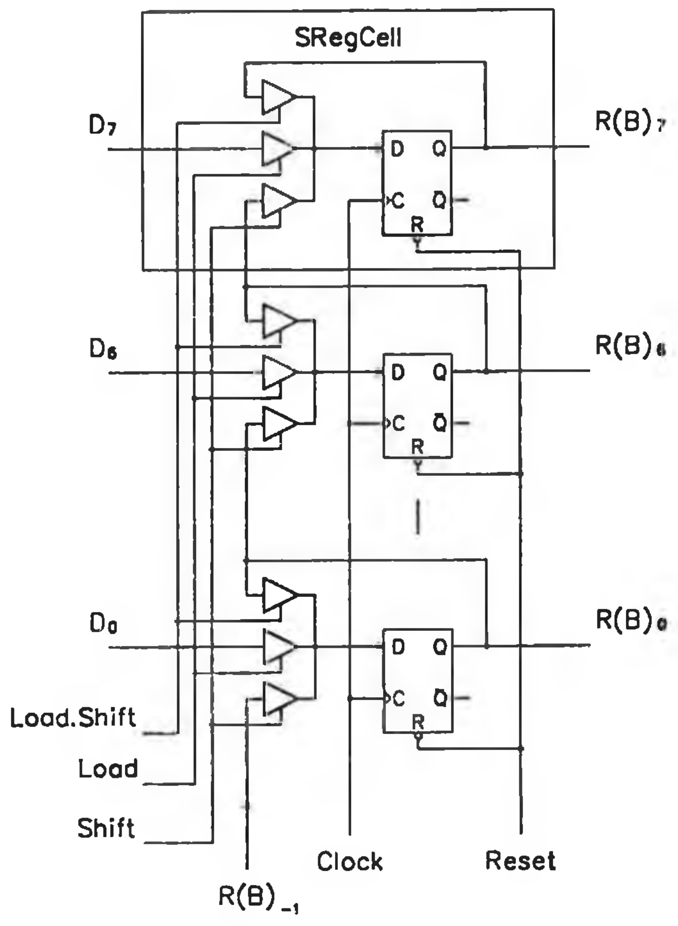


Fig. (6.7a) Logic diagram – Shift Register R,B

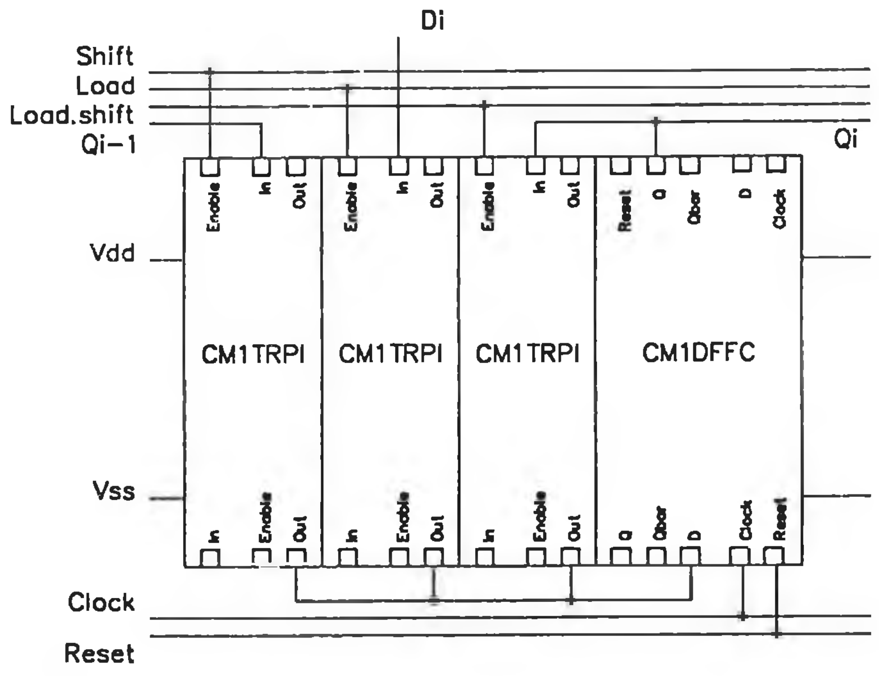


Fig. (6.7b) Shift Register Cell – Layout

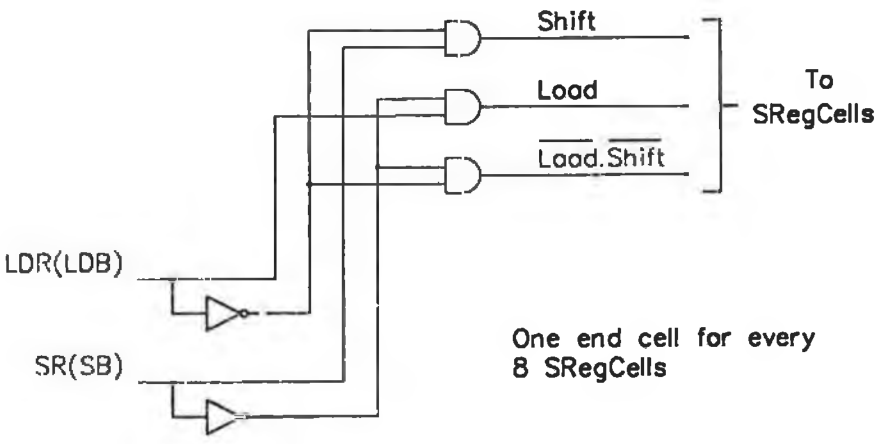

Shift
Load
$\overline{\text{Load}}.\overline{\text{Shift}}$
To SRegCells
LDR(LDB)
SR(SB)
One end cell for every 8 SRegCells

Fig. (6.8a) End cell for shift registers R, B

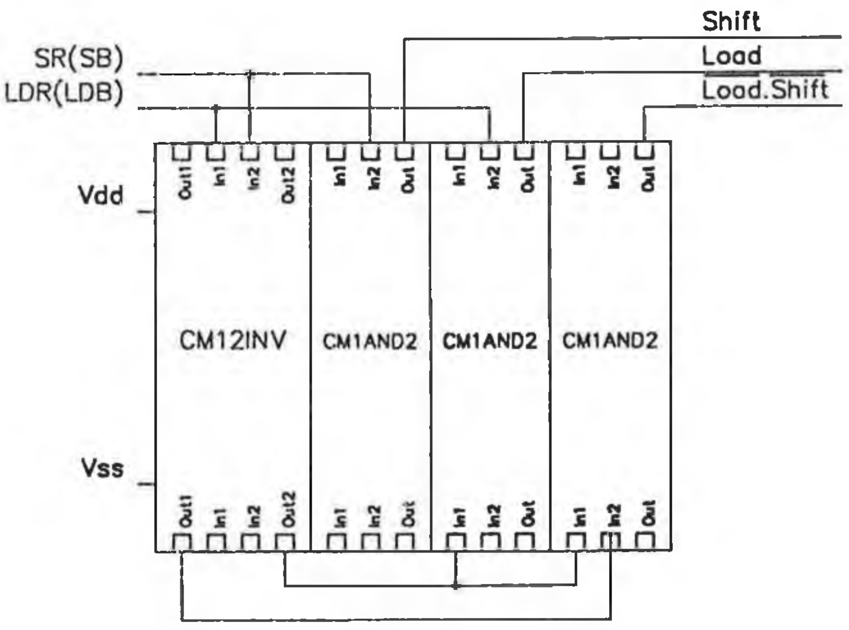


Fig. (6.8b) Layout of end cell

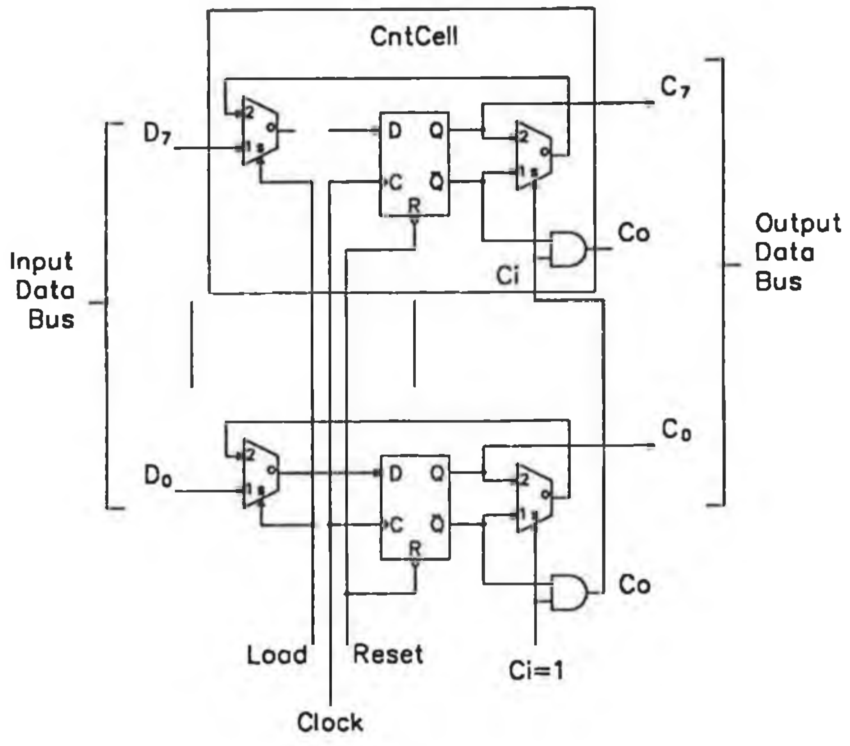


Fig. (6.9a) Logic diagram – Down counter

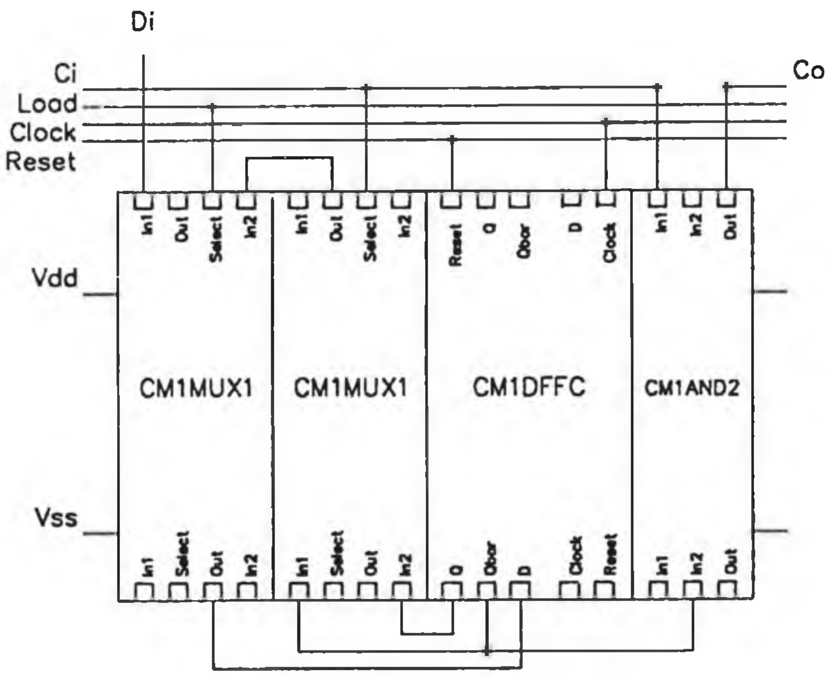
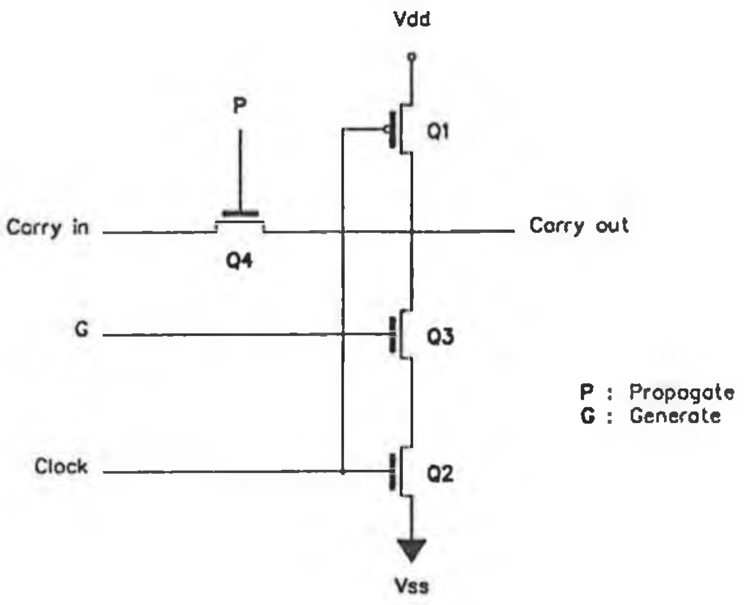
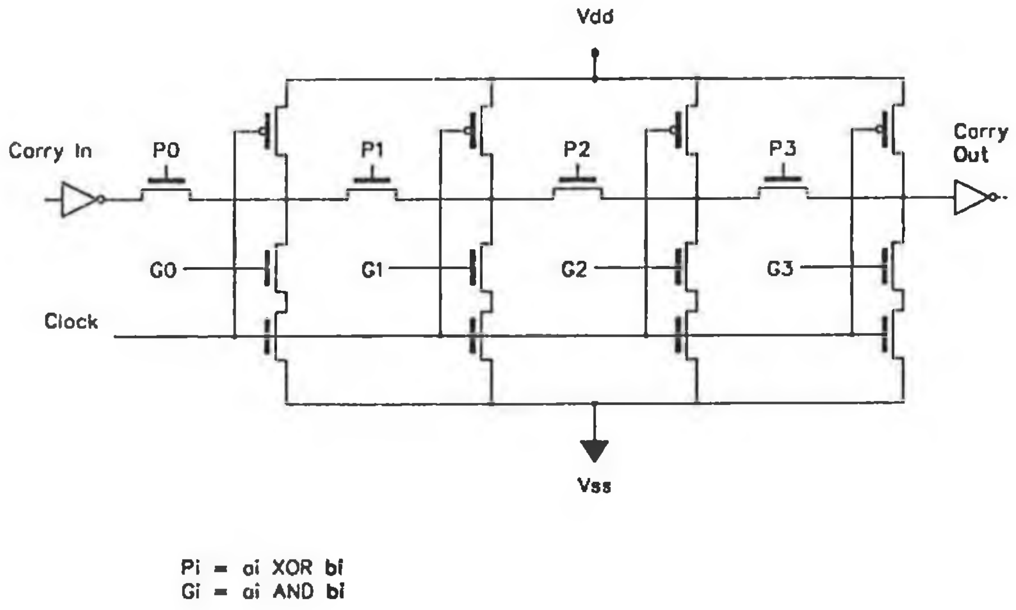


Fig. (6.9b) Layout of down counter cell



Principle of Manchester carry-chain



4 bit carry chain

Fig. (6.10) Manchester carry–chain

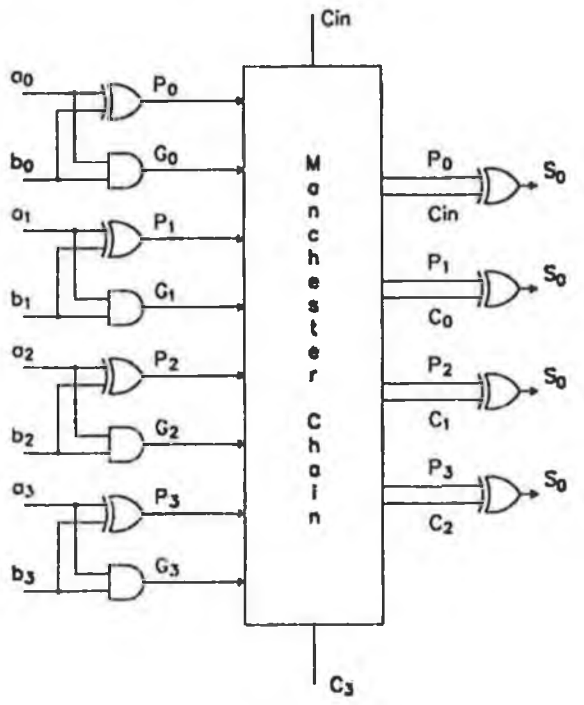
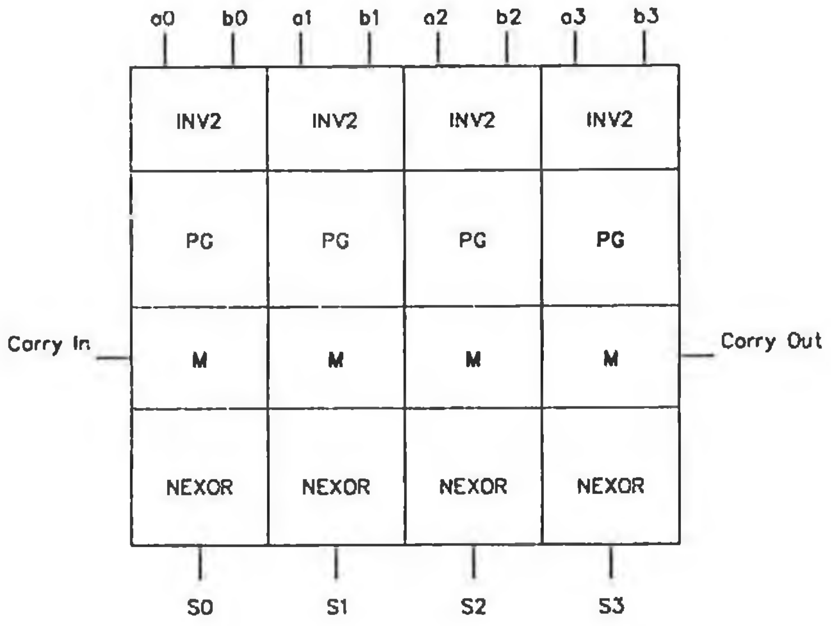
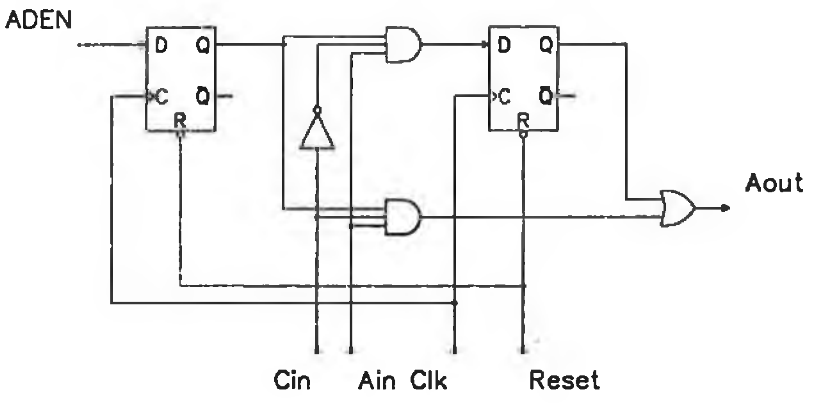


Fig. (6.11a) 4 bit adder – logic



M : Manchester carry cell
PG : Generation of propagate and generate terms

Fig. (6.11b) 4 bit adder – layout



ADEN	:	Adder enable signal from control unit
Ain	:	Addition complete signal from previous stage
Aout	:	Addition complete signal to next stage
Cin	:	Carry in from previous stage

Fig. (6.12a) Addition Complete circuit

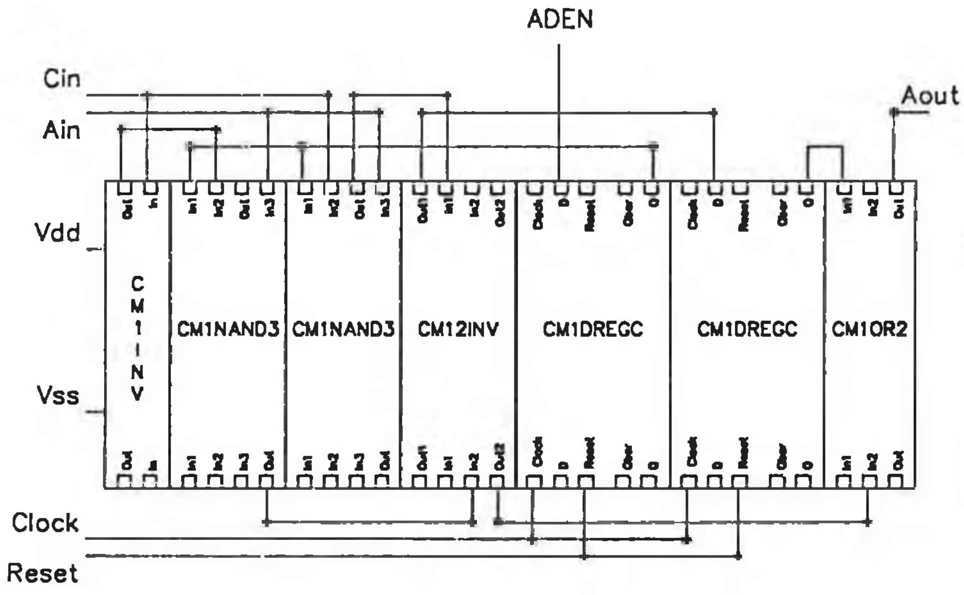


Fig. (6.12b) Addition complete circuit
– Layout

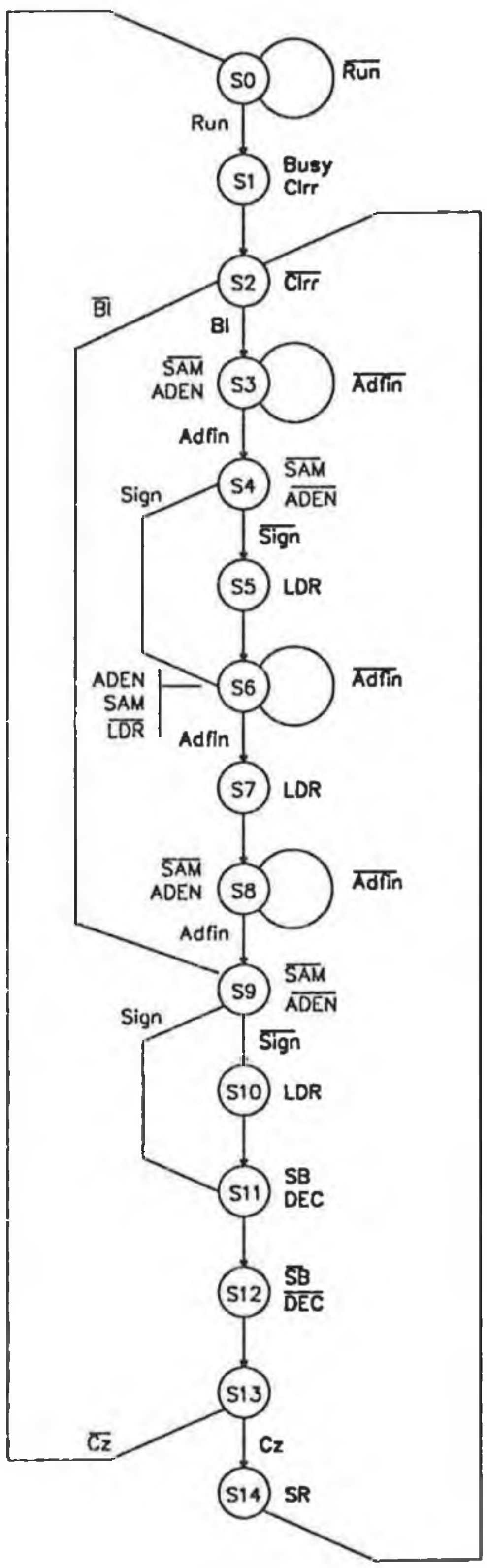


Fig. (6.13) State diagram – Blakelys algorithm

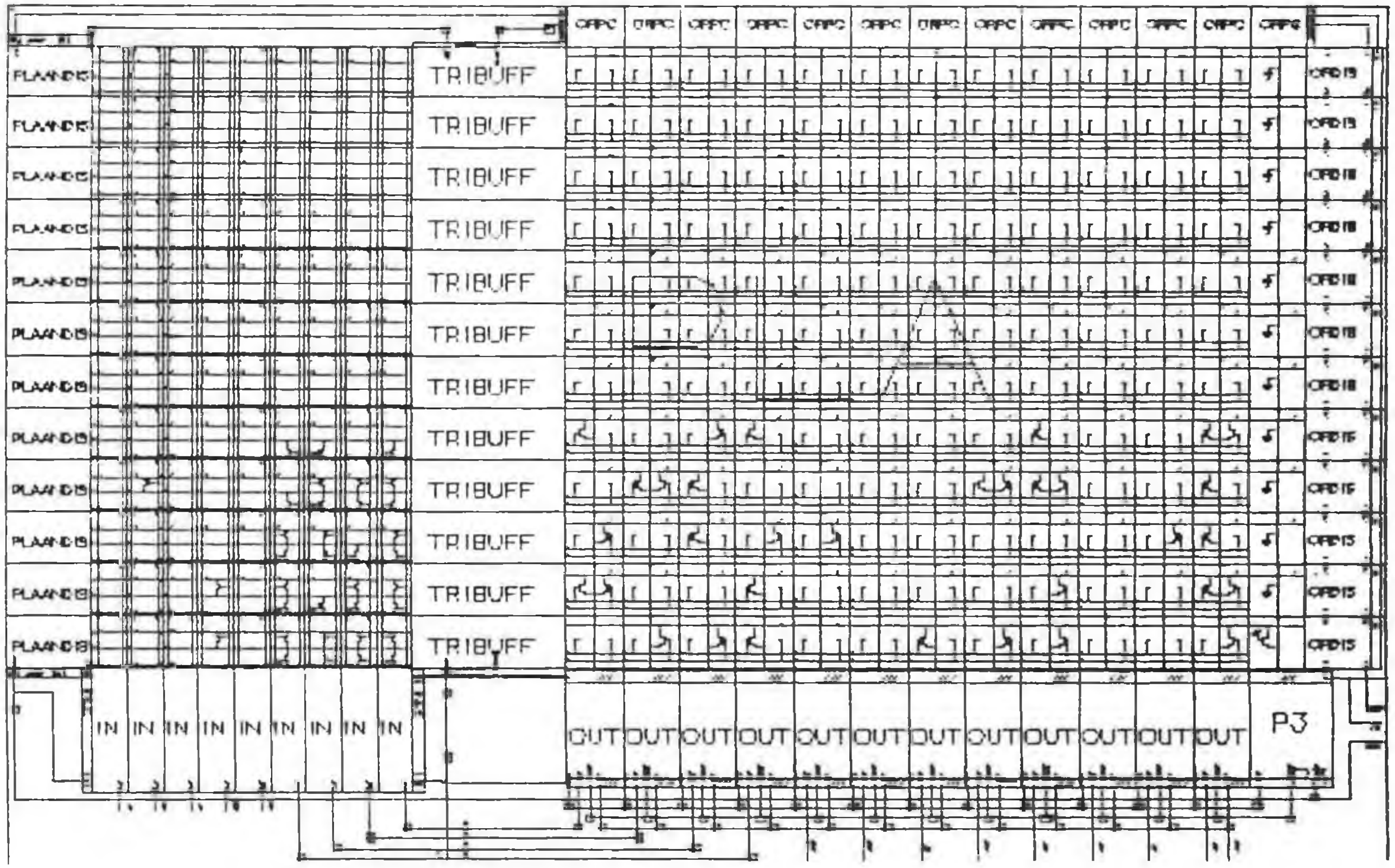


Fig. (6.14) Block diagram of PLA

Key Length (Bytes)	Multiplication Time (uS)	Key Length (Bytes)	Multiplication Time (uS)
2	280	42	26040
4	656	44	28336
6	1128	46	30728
8	1696	48	33216
10	2360	50	35800
12	3120	52	38480
14	3976	54	41256
16	4928	56	44128
18	5976	58	47096
20	7120	60	50160
22	8360	62	53320
24	9696	64	56576
26	11128	66	59928
28	12656	68	63376
30	14280	70	66920
32	16000	72	70560
34	17816	74	74296
36	19728	76	78128
38	21736	78	82056
40	23840	80	86080

Fig. (6.15) Multiplication time using the MMD

Key Length (Bytes)	Encryption Time (Sec)	Key Length (Bytes)	Encryption Time (Sec)
10	1.0	46	50
12	1.5	48	56
14	2.2	50	63
16	3.1	52	70
18	4.2	54	78
20	5.5	56	86
22	7.1	58	95
24	8.9	60	105
26	10	62	115
28	13	64	125
30	15	66	137
32	18	68	149
34	22	70	161
36	25	72	175
38	30	74	189
40	34	76	204
42	39	78	219
44	44	80	236

Fig. (6.16) Encryption time using the MMD

7. Conclusion

An investigation into the implementation of the RSA cipher on a standard 8 bit microprocessor, the Motorola M6809 was carried out. A programme for modular exponentiation, which is the basis of the RSA public key cipher, was written. The algorithm used is known as exponentiation by repeated squaring and multiplying. The run time of this algorithm was calculated and was found to be dependent on the time required for modular multiplication.

Two forms of modular multiplication algorithm were identified. The first calculated the product using normal arithmetic and then calculated the residue. The second form calculated the residue simultaneously with the product.

In the first type of algorithm the formation of the product did not change, however various ways of calculating the residues were implemented with significant differences in performance. The similarity between the formation of residues and division allowed division algorithm, with the emphasis on the remainder and not on the quotient to be used. The methods used for calculating the residues were, division by repeated shift and subtract, division by forming the reciprocal and multiplying and division using Knuths algorithm.

The second type of modular multiplication algorithm, which calculated the residues simultaneously with the product was implemented in assemler language using an algorithm by Blakely. the advantage of this approach is that a double length product is never formed, the largest number that must be dealt with is twice the modulus.

The performance of all of the modular multiplication algorithms, in terms of their run time for different modulus length, was evaluated. The results showed that forming the reciprocal and multiplying is the fastest method followed closely by Knuths algorithm. These are then followed by the shift and subtract algorithm nad Blakelys algorithm, both of which are significantly slower than either of the first two.

The encryption time using these microprocessor programmes was found to be quite significant. Rivest, Shamir and Adleman in their original paper recommended that key lengths of the order of 200 decimal digits (or approximately 80 bytes) should be used if a high level of security and protection against future developments is required. The time taken to encrypt messages of this size using the fastest of the above software algorithms is approximately 40 minutes. This implies a bit rate of 1/4 bits per second. If we reduce the key to 40 bytes (approx. 100 decimal digits) then the encryption time reduces to 322 seconds or a bit rate of one bit per second.

This slow transmission rate may be of some use in a limited number of cases e.g encryption of files prior to long term storage but it is of no use for a general purpose encryption device. Several solutions to the problem were considered. These were :

1. Reduce the key length even further (i.e <40bytes)
2. Find a more efficient algorithm.
3. Implement part of the algorithm in hardware.

The third solutiion, hardware implementation of some of the key elements of the encryption process, appeared to offer the best chance for significantly reducing the encryption time and it was decided to follow this option.

The generation of keys for use in the RSA cipher was described. The security of the RSA cipher was shown to reside in the key itself. Various forms of attacks against the RSA cipher have a higher probability of suceeding if the key is not chosen correctly. A procedure, developed by Knuth, was implemented in FORTRAN on the VAX. The run time of this programme is significant (15 minutes for 100 digit keys) and the limitation this places on frequent changing of the key is particulary severe in a personal computer or microprocessor environment.

An investigation into the various methods of hardware multiplication resulted in the choice of a serial/parallel multiplier. It was thought that this offered the best compromise in terms of speed and space. Blakelys algorithm although the slowest of the software implementations lends itself, with small modifications, to a hardware solution. It was thus decided the hardware should implement Blakelys algorithm on a serial/parallel multiplier.

Three approaches for producing the required hardware solution were considered. These were, a dedicated circuit usin standard SSI and MSI logic, a circuit using bit slice microprocessor techniques and the design of an application specific integrated circuit (ASIC) using gate array or standard cell techniques.

The implementation of the hardware using a standard cell ASIC was chosen. The reason for this choice was the availibility of the necessary facilities at the National Microelectronic Research Centre (NMRC) and the receipt of a grant from the National Board for Science and Technology (NBST now Eolas) to carry out the work.

The design of the integrated circuit was carried out at N.I.H.E Dublin and at the N.M.R.C in Cork. A 16 bit modular multiplier based on Blakelys algorithm and using a serial/parallel multiplier was designed. The multiplier was implemented using CMOS standard cells and the 5 micron process technology at the N.M.R.C. After simulation of the design layout of the device was carried out. Further simulation and verification of the device were necessary before fabrication of the device.

Simulation results from the device show that an order of magnitude increase in encryption time can be expected. Fabrication of the integrated circuit has just been completed and testing to verify these figures has yet to be done.

The increase in speed of the algorithm by an order of magnitude, whic will give a bit rate of 10bps for a 40 byte key, is still not sufficient for a general purpose encryption device. Conventional cipher systems, such as DES, can achieve bit rates in the megabits per second range. Further increases in the speed of public key ciphers will occur with advances in integrated circuit technology and with the discovery of more efficient algorithms. Advances in integrated circuit technology will allow a greater density of devices per chip and thus the integration of more or all of the public key cipher. This would reduce the bottlenecks inherent in transferring information to an from the encryption device. Different algorithms, more suited to hardware implementation, may also increase the overall level of performance.

Until then the best compromise would appear to be a combination of public and private key ciphers. The public key cipher could be used to distribute session keys for use by a conventional cipher, such as DES.

REFERENCES

1. KAHN, D
'The Codebreakers' (Macmillan, 1967)

2. DIFFIE, W. and HELLMAN, M
'New Directions in Cryptography.' (IEEE Trans. Info. Theory IT-22, 6-Nov-1976, pp. 644-654)

3. MERKLE, R. and HELLMAN, M.
'Hiding Information and signatures in trap-door knapsacks' (IEEE TRans. Info. Theory, IT-24, Sept-1978 pp. 525-530

4. RIVEST, R., SHAMIR, A. and ADLEMAN, L.
'A Method of obtaining digital signatures and public-key cryptosystems' (Comm. ACM, 21, 2, Feb-1978, pp. 120-126)

5. DENNING, D. E.
'Cryptography and data security' (Addison-Wesley 1983)

6. NATIONAL BUREAU OF STANDARDS
'Data Encryption Standard' (FIPS Publication 46 Jan-1977)

7. ZIMMERMANN, P.
'A proposed standard format for RSA cryptosystems' (IEEE Computer, Sept-1986, pp 21-34)

8. KNUTH, D. E.
'The Art of Computer Programming Vol. 2 Seminumerical Algorithms (Addison-Wesley, 1981)

9. BLAKELY, G. R.
'A computer algorithm for calculating the product AB modulo M' (IEEE Trans. computers, Vol. C-32, No. 5, May-1983)

10. POHLIG, S. and HELLMAN, M.
'An improve algorithm for computing logarithms over GF(p) and its cryptographic significance' (IEEE Trans. Info. Theory, IT-24, Jan-1978)

11. ADLEMAN, L.
'A Subexponential algorithm for the discrete logarithm problem with applications to cryptography' (Proc. IEEE 20th Symp. on Foundations of Computer Science, Oct.-1979)

12. SCHROEPPEL, R and SHAMIR, A.
'A T*(S**2) = O(2**n) time/space trade-off for certain NP-complete problems' (Proc. 20th IEEE Symp. on Foundations of computer science, Ost-1979)

13. QUISQUATER, J and Couvreur, C.
'Fast Decipherment Algorithm for RSA public-Key cryptosystem' (Electronic Letters 14-Oct-1982 Vol. 18 No. 21.)

14. BLAKELY, B. and BLAKELY, G.
'Security of number theoretic public key cryptosystems against random attack' (Cryptologia 4-Oct.-1978, 1-Jan.-1979, 2-April-1979)

15. NORRIS, M. and SIMMONS, G.
'Preliminary comments on the MIT public key cryptosystem' (Cryptologia 1,4,1977)

16. SOLOVAY, R. and STRASSEN, V.
'A fast Monte-Carlo test for primality' (SIAM J. Computing, 6, 1, March-1977)

17. CRISPIE, F.
'CMOS Cell Library' (NMRC, Sept.-1985)

18. WESTERN DIGITAL
'WD2001/02 Data Encryption Devices'
(Western Digital Corporation, July-1984)

19. SHAMIR, A. and ZIPPEL, R.
'On the security of the Merkle-Hellman cryptographic scheme' (IEEE Trans. Info. Theory, IT-24, 3, May-1980)

20. CURRAN, T. and BRADY, P.
'Research Report (Jan - May 1985)'
(N.I.H.E Dublin, May-1985)

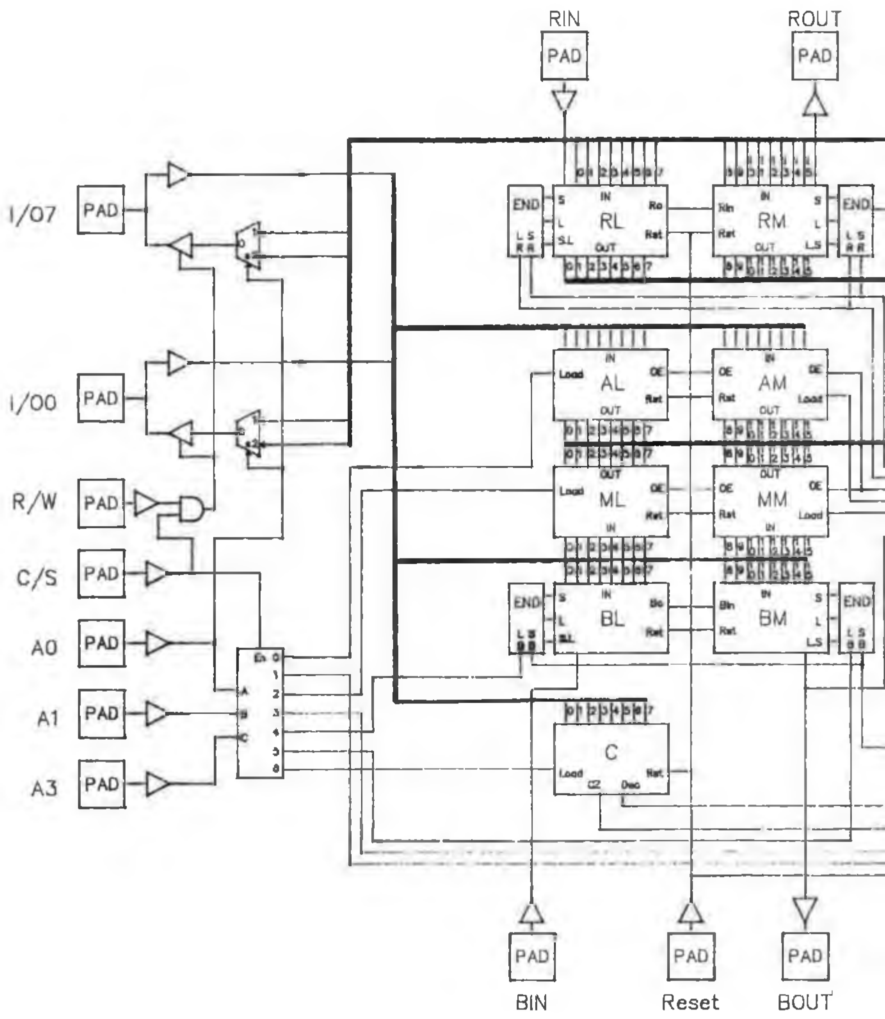
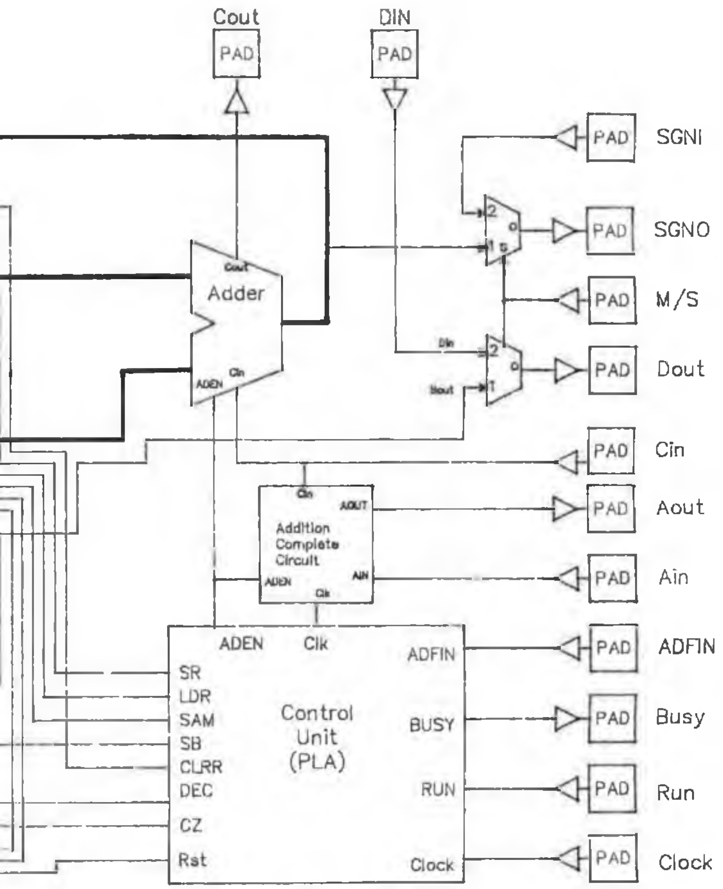


Fig. (6.4b) Circuit Diagram


Cout
PAD
DIN
PAD
SGNI
SGNO
M/S
Dout
Cin
Aout
Ain
ADFIN
Busy
Run
Clock
Adder
Cout
Cin
ADEN
Din
Sout
Addition Complete Circuit
AOUT
AIN
ADEN
Clk
Control Unit (PLA)
ADEN
Clk
SR
LDR
SAM
SB
CLRR
DEC
CZ
Rst
ADFIN
BUSY
RUN
Clock

ADDENDUM

A1. The use of a PLA in the Control Unit.

Two ways of implementing the control unit were considered :

1. Dedicated Design
2. PLA

The dedicated design implemented the control unit using standard logic gates and flip-flops. An investigation was carried out to determine the space required by this approach. The logic elements occupied 1.7mm^2. This did not include routing between the devices. A rule of thumb, used at the N.M.R.C, requires that an equal amount of space be allowed for routing. This resulted in a total space requirement of 3.4mm^2 .

The PLA is designed on a modular basis. This allows for flexibility in the number of inputs, outputs and product terms that can be accomadated by the PLA. The PLA used is a dynamic type and requires a clock and precharging. To ease the construction process the PLA was designed so that cells could be placed together to form the complete unit. Programming of the AND and OR planes is carried out by placing a cell (ANDPROG or ORPROG) at the junction of the AND or OR plane and the product line.

The size of the PLA (based on 9 inputs, 12 outputs and 21 product terms) was calculated to be 2.5mm^2. Routing within the control unit is not required in this implementation due to the regular nature of the PLA.

Choice of method.

The PLA was chosen as it requires less space than the discrete version. Another advantage of the PLA is that the control algorithm can be modified, without affecting the size of the PLA (providing the number of inputs, outputs and product terms remains constant).

A2. Control and Data Path.

The modular multiplication device could be broken into two different areas. The first was called the data path, as it involved the storage and manipulation of the data. The second, called the control path, implemented the control algorithm. The reason for this division was that some of the functions of the multiplication device were dependent on the length of the operands while others were not. The size, or width, of the data path (i.e the number of bits required for the operands) could be changed to suit space restraints whereas the size of the control path was fixed and could not be changed.

A3. Width of Data Path.

One of the requirements of the modular multiplication device was the it should handle as long a key length as possible. The ultimate limit on the key size would be determined by the space available on the silicon die. The elements of the data path were therefore designed on a modular basis, e.g a one bit shift register was designed. When all the elements were designed an estimate of the space required for different key lengths was made. This exercise showed that a key length of 16 bits was the maximum that could be accomadated on the 6.1mm by 6.1mm die.