Dublin City University

School of Electronic Engineering

Multi-Precision Arithmetic on a DSP

Dara Murtagh B. Eng.

A Dissertation submitted for the degree of Master of Engineering.

Supervisors : Dr. Sean Marlow
Dr. Michael Scott

September 1991

Dedication

This thesis is dedicated to my parents.

Acknowledgements

I would like to thank my supervisors, Dr. Sean Marlow and Dr. Michael Scott, for their continual encouragement and advice. My colleagues, Niall Byrne, Paddy Gibbs, Brendan McKittrick, and many other friends in Engineering provided help and support at various stages of the project. In addition I appreciate the support given by members of the administrative, technical and academic staff along the way. Thanks again to my parents.

Declaration

I hereby declare that this dissertation is entirely of my own work and has not been submitted as an exercise to any other university.

Dara Murtagh

ABSTRACT

The aim of this project has been to develop the assembly language functions needed to allow easy implementation in real-time of a secure speech channel. The theory of security systems is introduced and developed. Encryption algorithms are described. A library of multi-precision arithmetic routines has been written for use on the TMS320C25 digital signal processor. These routines are compatible with code produced by the TMS320C25 C Compiler. Multi-precision arithmetic is used in public key encryption which requires large number arithmetic for security and which also has real-time operation requirements. An overview of DSP use in this kind of application is given, the design, implementation and test of these routines is described and some application examples and timings are shown.

CONTENTS

1. INTRODUCTION

1.1 Introduction to Security Systems

The recent communications revolution has resulted in the proliferation of technological data transfer systems. There has been a large increase in the availibility of various communication devices - television, radio, cellular telephones and computer data links - with a corresponding decrease in the cost to the user. Previously information transfer using letters and telegrams had more clearly defined levels of security. If the letters were personally delivered or the telegraph operators were trustworthy only a physical interception of the letter or a wire-tap could compromise the security of the channel. The workings of modern communications systems are generally invisible to the user, so the security offered is also not readily apparent.

The growth in the communication of information raises questions such as when does a system require security and, following on from that, assuming that it does need to be made secure, just how much security is required? If information is worth transmitting, it is because there is a value associated with it. If access to this information by an unauthorised party can cause any loss in value, then making the system more secure will have cost benefits. While both the communication system and the security enhancement method used may be highly technical, these kinds of systems are often best understood by comparing them with ordinary mail.

If the information is very valuable, it is possible that the usual delivery system is not adequate and a courier may have to be used at a much higher cost. Or it may be that a courier needs to be used just once to allow the transmission of some secret information which can be used in the future to make secure the transfer of information over an insecure channel i.e. the usual mailing system. Questions also arise concerning the possibility of not only passive eavesdropping, but active interference with the information being sent, by an unauthorised user of the system, so that the authentication of the origins and of the complete correctness of the message is also an issue. These topics must be considered in general conceptual terms before relating them to a specific application. The study of the theory of mathematical systems for solving these security problems is known as cryptography.

Ordinary paper mail is an example of a possible application for cryptographic security methods. Other applications include electronic mail, automated teller machines, computer password systems, military friend-or-foe identification systems, nuclear test-ban treaty monitoring and voice communication. The electronic method of information transfer is closely analagous to the paper mailing method and thus uses all the standard encryption schemes which are outlined later. Automated Teller Machines (ATMs) and computer password systems have similar security problems : it can be dangerous to store Personal Identification Numbers (PINs) or passwords in a direct form because access to a table of these values would seriously compromise the security of the system. The method used to overcome this threat is to store only the result of an encryption of the PINs or passwords, which is either impossible or infeasible to decrypt, so that access to the table is not useful to an intruder in the system. Military identification friend-or-foe systems rely on a friendly aircraft being able to encrypt correctly a message sent by the challenging aircraft. The message is never repeated so the recording of previous challenges does not allow the system to be compromised. Public-key cryptography with its digital signature capabilities facilitates the transmission of seismic observatory data for monitoring nuclear test bans. The method used allows the host nation to decrypt the transmitted message to ensure that the appropriate data and nothing else is being transmitted but the message cannot be altered without detection by the monitoring country. Voice communication systems have seen fewer applications of cryptographic security methods but because of their widespread use it is logical that speech systems require at least as much security enhancement as text-based communication systems [1].

1.2 Project Motivation and Aims

In spite of the proliferation of text-based communication and its improved quality both in terms of speed and ease of use, people in general still prefer to use direct verbal communication methods. Speech can be used to communicate a lot of information quickly and it does not require training for use. It is not surprising therefore that the telephone is a more favoured communication medium than the letter or electronic mail. Speech raises technical problems in its generation, storage, recognition and transmission but these problems are in the domain of scientists and engineers, while the ordinary user wants and expects high quality easy-to-use voice systems [2].

Low transmission bit rates are required for the Mobile SATellite communication (MSAT) applications which need speech coding in the range of 4 kbits/sec according to Jayant [3]. INternational MARitime SATellite (INMARSAT) applications require transmission rates as low as 6.4 kbits/sec for speech communication which includes error protection coding. Another use of low bit rates is in speech storage : if speech data can be reduced to 4 kbits/sec it will be possible to store an hour of speech on a single 16 MegaByte memory chip [3]. Present digital speech technology allows "network quality" transmission - subjectively rated as high-quality or near-transparent coding - at transmission rates of 16 kbits/sec or more. "Communications quality" which allows natural telephone communication with easily detectable speech degradation is achievable at rates around 8 kbits/sec. Below this transmission rate the speech becomes "synthetic quality", still with high levels of intelligibility but with inadequate naturalness and speaker recognizability. It is at this lower transmission rate and level of speech quality that encryption-based secure voice systems are availible [2]. This kind of poor transmission quality is not generally acceptable and it is found in working systems only where security is at a very high premium, particularly in military applications. In this kind of environment the lack of clarity in transmitted message does not matter as much because users are usually trained individuals working with a limited message vocabulary.

The development of secure speech systems at "communications quality" transmission rates, approximately 8 kbits/sec, will make such systems as cellular telephone networks secure. At present accidental "crossed-lines" or pick-up of a transmitted signal can compromise the integrity of a link. Once the aim of practical encryption techniques at these medium range transmission rates is realised, any reduction in bit rate due to developments in speech coding technology will facilitate enhanced or at least equal levels of security. Also speech quality at about 8 kbits/sec is likely to improve to "communications quality", so secure communication can become achievable without unacceptable signal degradation [3].

At present there are two general classes of secure speech systems in use. One relies on scrambling the signal while the other is encryption-based. An example of a scrambling device is the DVS200 produced by Marconi which allows scrambling at rates up to 4.8 kbits/sec [4]. The use of scramblers to make transmitted speech more secure encounters severe limitations due to the nature of speech. Scrambling tries to separate a signal into small segments and to juggle these segments around before transmission, in a manner that will make the transmitted message unintelligible to an eavesdropper. The segments are then reassembled at the receiver. The scrambling is done in the analog domain and may involve time elements or frequency elements or both. Even in its most complicated form scrambling is limited both by the inherent redundancy of speech and by the requirements of a speech channel. The segments cannot be made arbitrarily small because this would result in a lack of intelligibility, caused by the smoothing sections between segments at the receiver being too significant compared to the actual signal. There is a small number of possible frequencies that can be scrambled within a speech bandwidth. Also a large proportion of speech is silent and one piece of silence will fit in a possible descrambled solution just as well as any other piece of silence. Finally the high level of redundancy inherent in speech means that the message can be made out if only a small number of segments have been correctly realigned. Scrambling is like a jig-saw puzzle with a limited number of pieces and observable continuity within the underlying picture, while a small proportion of correctly aligned pieces will let the solver in on the bigger picture. A computer can be used relatively easily for this puzzle solution, and the breaking of the security of a scrambling system in this jig-saw way is not even dependent on any lack of security in the key generation and distribution method [2].

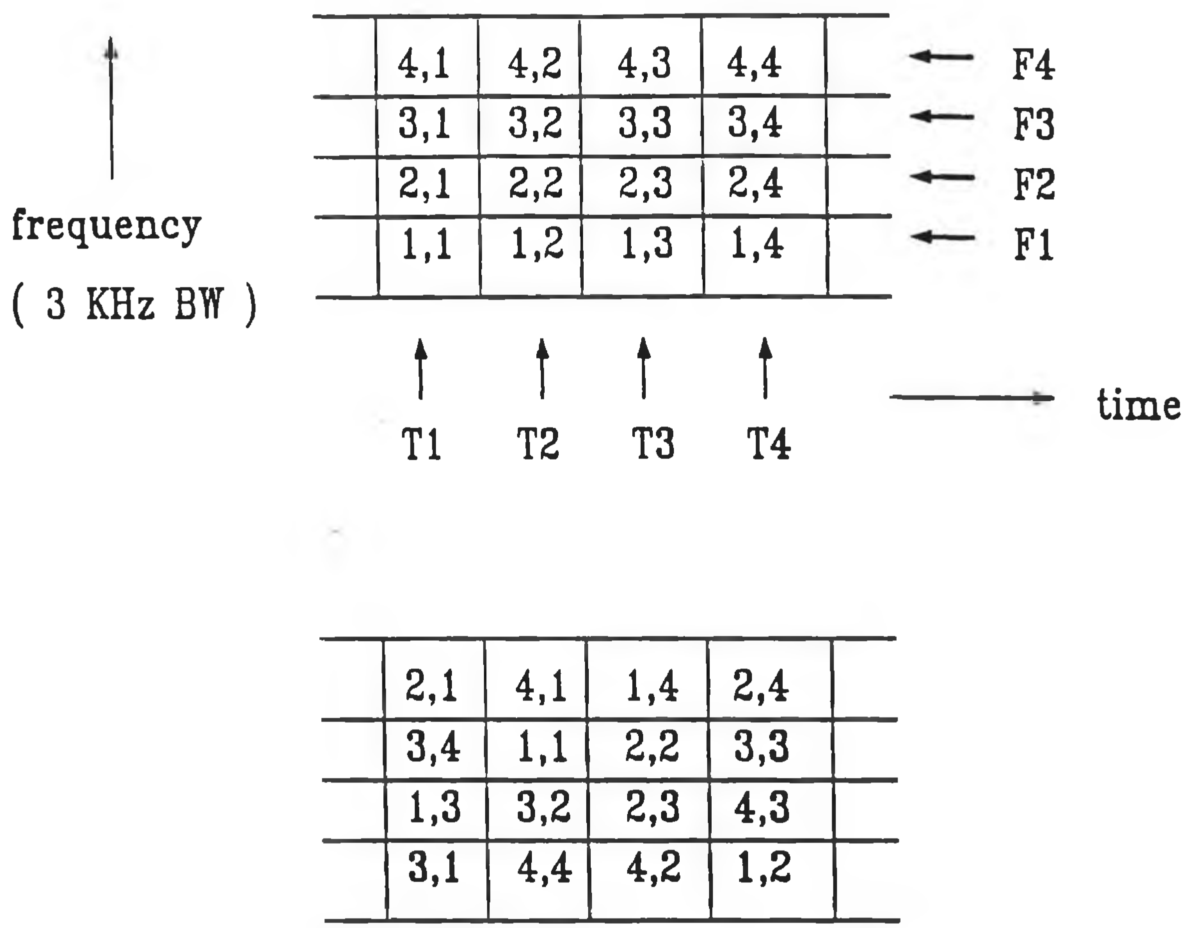


Fig. 1.1: *Speech scrambling : Each segment is modulated to a different frequency band and is moved up or delayed by a discrete multiple of a timing period* [2].

Encryption methods are not effected by the problems which limit the security of scrambling but they are effected by other problems. Encryption is carried out in the digital domain, operating on a speech signal either in block or bit-stream format. Scrambling offers limited security because it is unable to remove the underlying structure of a speech signal. Because encryption takes in and sends out data in a digital format, the underlying form is more easy to disguise. The outputted data should appear completely random and unintelligible, with no detectable structure, to the eavesdropper [2]. The drawbacks of encryption methods are due to the difficulty in carrying out this procedure at high enough bit rates. The problem of key distribution is an important issue in scrambling and is even more significant in encryption as encryption offers higher levels of security. A secure method of key distribution is crucial to the maintenance of the security offered by the general workings of an encryption algorithm. While conventional secret-key cryptosystems offer good security, the key exchange problem is such that for an encryption scheme to work in a large computer or telephone network, it is likely that either a public-key system or a hybrid system, involving public key initialisation of a private key for a secret-key system, must be used. The security implications of using a secret-key cryptosystem are such that Diffie in his paper "The First Ten Years of Public-Key Cryptography" [1] can cite two known cases in which key information was sold by workers in sensitive American installations to the Russians. If a hybrid system or a public-key system had been used the keys would have been for short term transmission periods and less information would have been jeopardised.

Over the last few years several technological developments have come together to make secure encryption-based speech systems possible at high enough bit rates. The whole area of encryption, particularly public key encryption, has seen innovations which have moved number theory algorithms from being theoretical to commercial. From being a challenging hypothesis in 1975, public key encryption has progressed to working systems. There are still limits : "The fastest RSA implementations run at only a few thousand bits per second, while the fastest DES implementations run at many million" [1] (RSA is the Rivest, Shamir and Adleman algorithm which is the leading public-key system, while DES is the Data Encryption Standard which is the industry standard secret key-system). It is true though, in spite of these limitations, that public key encryption has moved from being a new and fertile area for research to an area which has seen many commercial applications.

In parallel with theoretical advancements, there has been an evolution of microprocessors which now have the capability to carry out numerically intensive number theory operations at high speed. A digital signal processor (DSP) is a chip on which the type of operation that occurs frequently in digital signal processing applications, such as correlations and fast fourier transforms, is implemented in a single instruction. Operations which are complex and slow on a general purpose microprocessor are made simple and fast on a digital signal processor by being implemented in hardware. In particular, digital signal processors have a hardware multiply, generally with parallel accumulate, data move and register manipulation operations.

All public key encryption techniques rely on observations from number theory and very large numbers must be used in these number theory operations for genuine security. In order to deal with numbers which are larger than the single-precision word length imposed by a microprocessor the numbers must be represented in a multi-word or multi-precision format. Hardware devices which support multi-precision arithmetic are not yet availible so there is a need for the software development of multi-precision arithmetic routines.

The Multi-Precision Arithmetic C Library (MIRACL) developed by Dr. Michael Scott [5] is a library of nearly 100 routines written mostly in C which covers all aspects of multi-precision arithmetic. The package also allows the use of large rational numbers without rounding. The multi-precision integer routines in the library are based on Knuth's classic algorithms for multi-precision arithmetic which are presented in Volume 2 of his work "The Art of Computer Programming" [6]. In MIRACL these routines are optimised for speed and efficiency in C code but with the time-critical numerically intensive sections written in assembly language for a wide range of machines. This library provides the basis for the development of cryptographic applications and includes two public-key cryptography systems : the Rivest Shamir and Adleman [7], and the Blum and Goldwasser systems [13]. Several routines are also provided for factoring large numbers which is necessary both in key generation for public-key encryption and in attempts at cryptanalysis. The encryption routines provided in the library can be used for secure data transmission.

The same classical algorithms by Knuth, upon which the integer routines in MIRACL are based, have been used as the basis for the multi-precision arithmetic routines written for the TMS320C25 digital signal processor. The aim of this project is to provide similar encryption facilities to those in MIRACL which are for data transmission, at a speed which is appropriate for speech. The DSP routines are compatible with cross-compiled C code and the aim has been to make them as compatible as possible with the MIRACL package. The encryption applications developed have been written in C code and the Texas Instruments C Compiler links in the DSP routines for the time-critical numerically intensive operations. The benefits of high level coding in C are apparent as applications are easy to write while the low level DSP routines ensure a fast implementation for real-time applications. The multi-precision arithmetic DSP routines have been written to be general purpose so, while the primary objective has been their use in the encryption schemes outlined in this thesis, they are flexible enough to be used in other cryptographic or number theoretical applications.

1.3 Thesis Overview

Section 1.1 is concerned with giving a general overview of security systems including examples and some ideas about when security is necessary, without getting too specific. Following on from these ideas it is possible to outline the motivation and aims for this project. The importance of speech as a communication system is developed in Section 1.2. The state of the art in speech transmission is outlined and then a description is given of speech scrambling methods which shows the need for the more secure encryption-based approach. Next a brief introduction is given which describes the reasons why speech encryption has become possible, including hardware developments and theory improvements over the last sixteen years. The Multi-Precision Arithmetic C Library is described. This leads to the aim of the project which can be summed up as providing some of the functionality of the MIRACL package for the purposes of easy development of public-key encryption algorithms for real-time applications.

Cryptography is the area of study that deals with the solution of the security problems which have been outlined in the introduction. It is necessary to look at this theory before attempting to produce any solution to these kinds of problem, particularly when, as in this project, it is hoped to develop general-purpose tools for differing applications. Chapter 2 provides an introduction to cryptography which includes definitions and explanations. The leading secret-key system, the Data Encryption Standard, and its limitations are discussed. Public-key systems are introduced and examples are given. Issues such as the authentication problem and the evaluation of encryption schemes are discussed. Finally another algorithm is introduced : the Blum, Blum and Shub pseudo-random number generator, which has potential in public-key speech applications.

The development of multi-precision arithmetic functions in TMS320C25 assembler has formed the bulk of the work in this project. These functions are described in Chapter 4. Along with a prototype and a brief outline of the use and peculiarities (if any) of each function, a low level description, a general explanation, a test procedure with results and a sub-section on timing is given for each function. Before these results are given, the general design and test philosophies applied in library development are outlined in Chapter 3. The usual ideas about modularity and extensive documentation for high-level programming are both more difficult and more important to apply in assembler development. An introduction is also given to digital signal processors with multi-precision arithmetic applications in mind. The number representation method employed and function usage are explained in this chapter. A brief description is given of the tools used in the development of the library.

The next stage in the project was to look at using the library functions which had been developed for some specific applications. These include the general arithmetic operation of exponentiation, which is regularly used in public-key sytems, the Rivest, Shamir and Adleman algorithm [7] and the Blum, Blum and Shub generator [8]. Code is given for these applications, which uses the DSP functions, in Chapter 5, and timings are outlined including comparisons with some results available in research literature. This naturally leads to Section 6, the Conclusion. Here the benefits and limitations of the project are discussed and recommendations are given for improved methods of tackling the problem of speech security.

The appendices include the assembly language source code for all the library functions and an example of the division algorithm which was used in the early testing of that function. The code developed for a demonstration of a Rivest Shamir Adleman algorithm file encryption is given. This example illustrates an application which makes use of the assembly language library.

2. INTRODUCTION TO CRYPTOGRAPHY

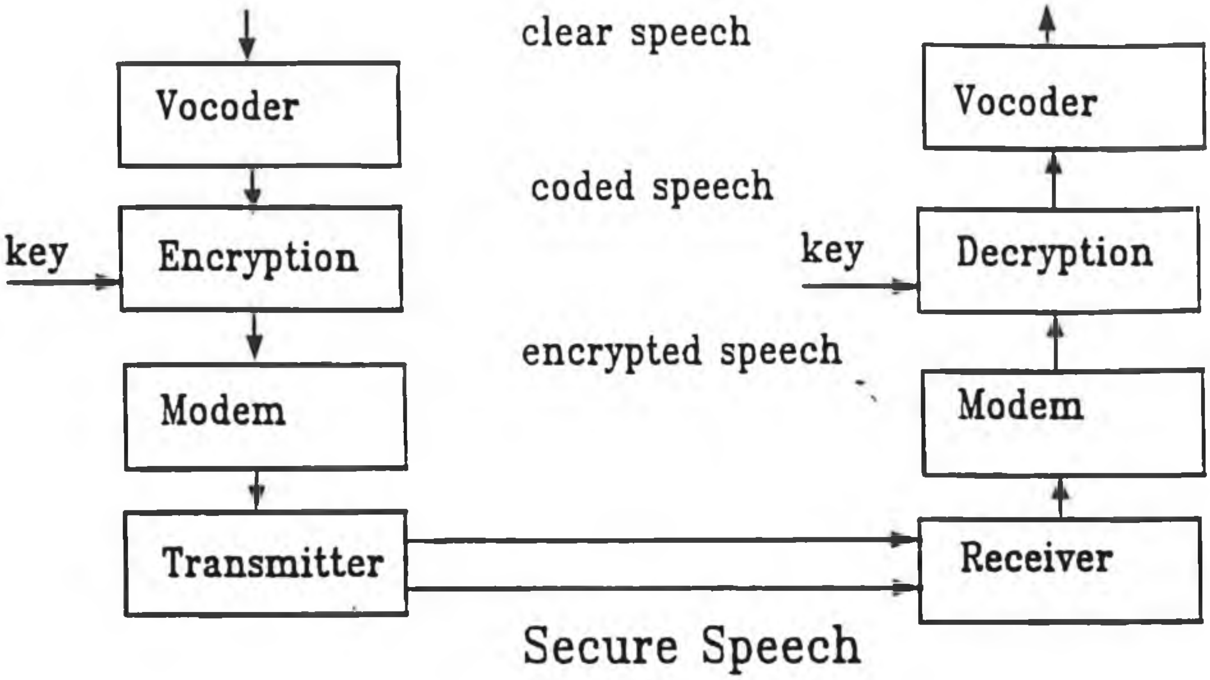


Fig. 2.1 *Block diagram of secure speech communication system* [2]

2.1 Definitions and Explanations of Terms

Cryptography is the study of mathematical systems for solving the security problems of privacy and authentication. The **sender enciphers** the **message** or **plaintext** into **ciphertext** which should appear random and meaningless to anyone without full knowledge of the system. **Decryption** is the inverse operation to encryption and is carried out by the **receiver** who converts the ciphertext back to plaintext form. The difference between coding and encryption is that if the set of rules or **algorithm** used is public knowledge no further information is required in a coding scheme to decode a message. Knowledge of the key is required as well as knowledge of the algorithm in an encryption scheme in order to be able to decrypt a message. The algorithm is in fact assumed to be public knowledge in encryption schemes, with security dependent on the key and the ability of the algorithm to magnify apparent randomness. In conventional **secret-key** cryptosystems the additional information to allow both encryption and decryption is called the **key**. Therefore the key must be kept secret and must be transmitted over a **private channel**. The encryption scheme is used to enhance the security of the **public channel** over which the message is transmitted. These terms are use-dependent : a telephone or letter may be considered to be private channels for most people's needs but public for sensitive or classified information transfer. In this case a courier is often used to transmit the key providing security at a cost.

Most encryption algorithms are not absolutely secure. The major exception is the **one-time pad** which is a special type of **mono-alphabetic cipher**. A mono-alphabetic cipher is a system in which each letter of the alphabet of the message space is mapped directly to a letter in the cipher space. The one-time pad requires different offsets for each letter in the message so that the key is as long as the message. While the encrypted message is provably secure, it remains necessary to securely transmit the key.

Practical encryption requires if not unconditional security, **computational security**. This means that the message should not be determinable at any less expense in computational power and time than the value of the message [9]. Therefore the evaluation of a system should take into account the value of the message and likely theoretical or technical developments which could reduce the **cryptanalyst**'s or encryption system breaker's costs.

2.2 The Data Encryption Standard [10 (pp.503-510),11]

Because there are so many possible secret-key encryption schemes and because of the sensitivity of the security area, the National Bureau of Standards in the United States decided on one particular algorithm as the standard for the transfer of non-classified information. This standard is only binding on government agencies who must come up with a valid reason if they do not use it, but because of the large market these agencies constitute and the general acceptance of the algorithm for non-governmental business applications, the Data Encryption Standard (DES) has become the de facto secret-key encryption industry standard [10 (pp.503-510),11].

The DES algorithm is a product cipher which works by performing a series of relatively simple permutations and substitutions on the message block based on a secret key. The strength of the algorithm is due to the non-linear increase in complexity which these steps produce. Shannon describes the use of permutations and substitutions in an encryption algorithm as resulting in diffusion and confusion of the message respectively [30 (p.92)].

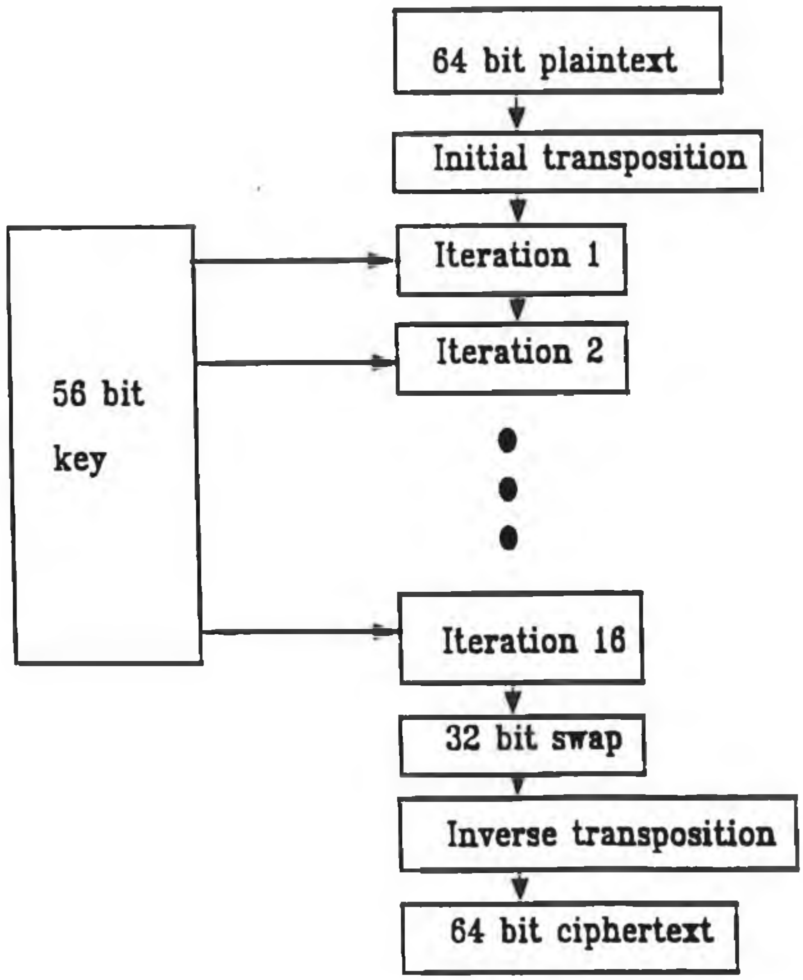


Fig. 2.2 *Block diagram of the Data Encryption Standard Algorithm* [10 (pp.503-510)].

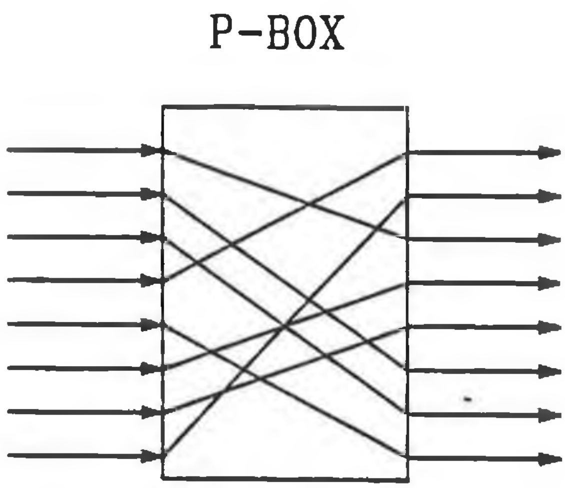


Fig. 2.3 *Permutation Box*

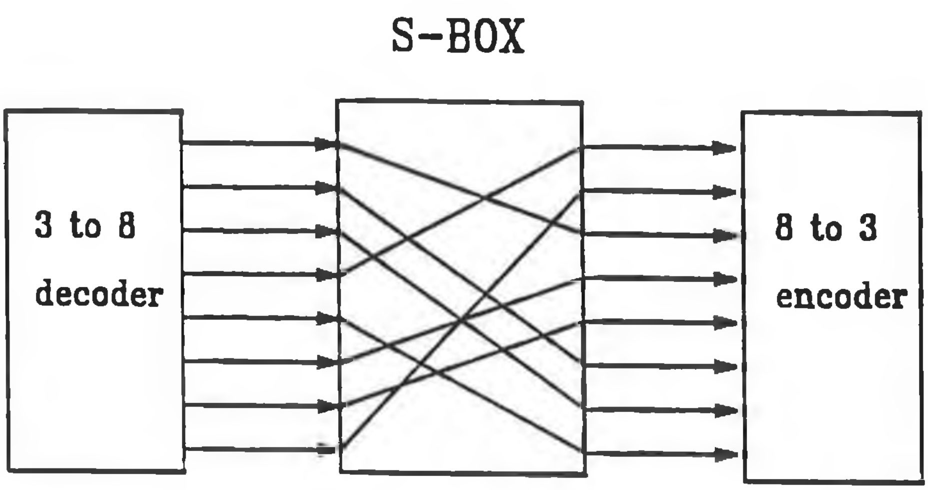


Fig. 2.4 *Substitution Box*

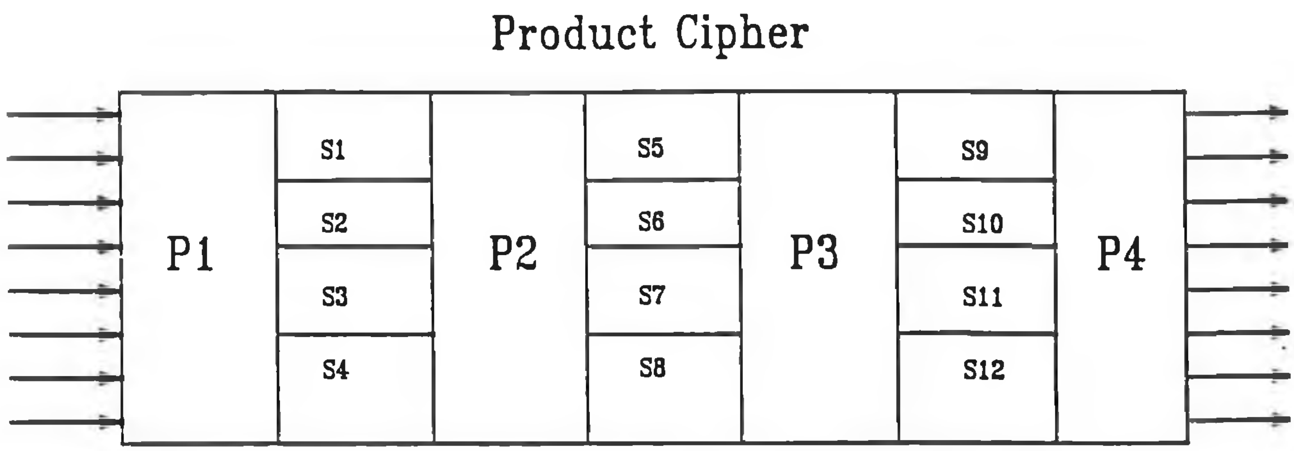


Fig. 2.5 *Product Cipher*

There has been controversy over the small size of key chosen for the DES algorithm. A 56 bit key size is used, following recommendations from the National Security Agency (N.S.A.), even though I.B.M., who designed the algorithm, originally wanted a 128 bit key. There have been suggestions that the N.S.A. wanted a key size which would allow them to cryptanalyse ciphertexts by exhaustive key search using their massive computational resources while keeping the key long enough to prevent others doing the same. This allegation should be taken into account when evaluating which encryption scheme should be used in any application. The cost of cryptanalysing a ciphertext should be greater than the value of the message. If the key is changed regularly, perhaps using a public system to set up the secret key, then the DES can still be an appropriate algorithm to use in some applications.

2.3 Public Key Systems

The point which causes the most difficulty in the working of a conventional cryptosystem is that the key must be kept secret and thus transmitted over a secure channel. This gives rise to a key management scheme with an exponential overhead for secure operation : to increase a network from size N to N+1 requires the generation of N new keys.

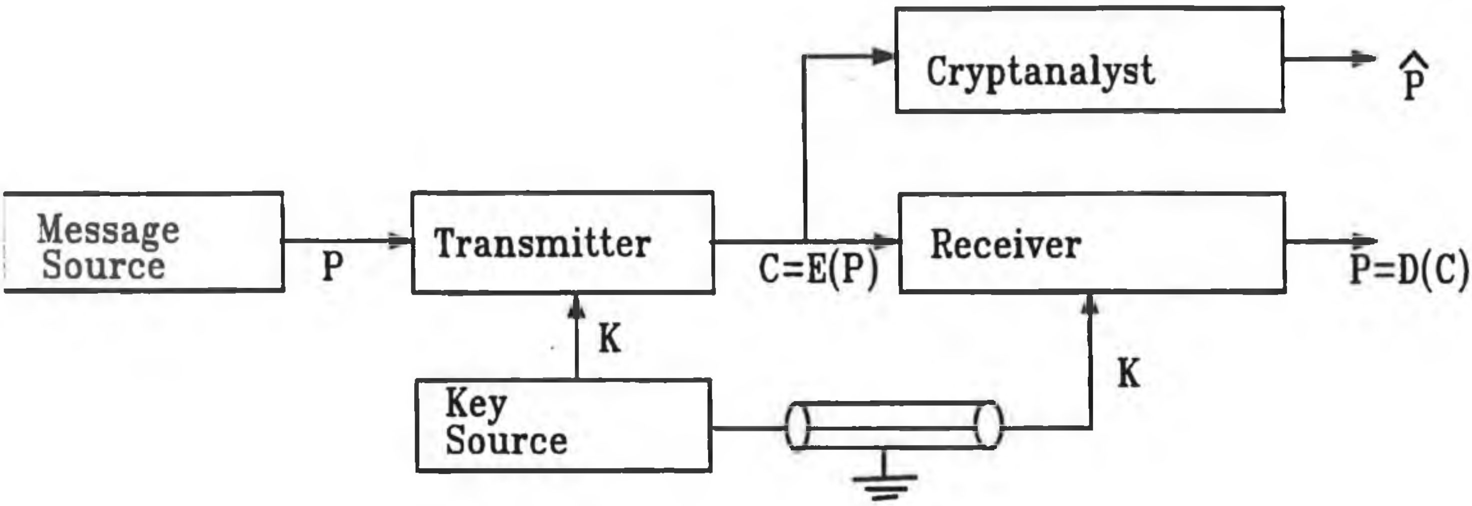


Fig. 2.6 ***Conventional Cryptosytem [6]***

$\hat{P}$: *Estimated plaintext*	P : *Plaintext*	C : *Ciphertext*
D : *Decryption function*	E : *Encryption function*	K : *Key*

The most notable feature of conventional secret-key cryptosystems (Fig. 2.6) is that the same key is used at the transmitter for encryption and at the receiver for decryption and as a result the key must be transmitted over a secure channel. In describing and designing cryptosystems, the cryptanalyst is assumed to have full access to any insecure channels and also to have knowledge of the encryption scheme being used so that the security of the system is purely based on the security provide by the key. The cryptanalyst tries to guess the message by exploiting any weakness in the encryption algorithm used or by using exhaustive key search. An analysis of the ciphertext is usually considered to be successful when a message which makes sense has been derived from the ciphertext.

Public key encryption is a new approach to the key distribution problem first expounded by Diffie and Hellman in their landmark paper : New Directions in Cryptography [12]. The idea is partly developed from the concept of **one-way functions**. One-way functions are easy to compute but difficult to invert. This characteristic has been used to protect computer password tables. Instead of storing passwords, the result of password mappings by a one-way function are stored. Thus unauthorized access to the table does not compromise security and the table can still establish the validity of a password [1].

If the one-way function has a "trap-door", some secret information which makes inversion computationally feasible, we have a public key cryptosystem. The forward mapping is encryption, while use of the trapdoor to invert this mapping is decryption.

These properties were outlined by Diffie and Hellman more formally, describing the requirements on both the encryption and decryption functions :

$$E_{K2} : \{ P \} \to \{ C \}$$

$$D_{K1} : \{ C \} \to \{ P \}$$

with constraints :

i) for every K in { K }, E_{K2} is the inverse of D_{K1} .

ii) for every K in { K } and P in { P }, the algorithms E_{K2} and D_{K1} are easy to compute.

iii) for almost every K in { K }, each easily computed algorithm equivalent to D_{K1} is computationally infeasible to derive from E_{K2} .

iv) for every K in { K }, it is feasible to compute inverse pairs E_{K2} and D_{K1} from K.

{ P } => plaintext space : set of all possible plaintext values.
{ K } => key space : set of all possible key values.
{ C } => cipthertext space : set of all possible ciphertext values.

This scheme is outlined in Diffie and Hellman's New Directions in Cryptography [12].

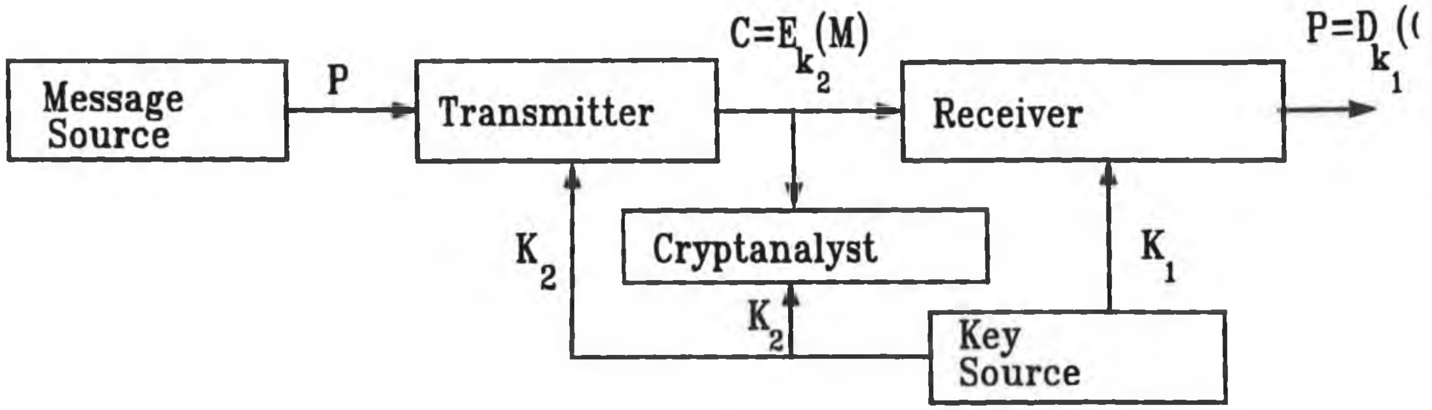


Fig. 2.7 : ***Public-Key Cryptosystem [6]***

P : Plaintext   *C : Ciphertext*   *K : Key*

E_{K2} *: Encryption function* D_{K1} *: Decryption function*

In a public-key cryptosystem the cryptanalyst has the same key information as the transmitter and access to the same ciphertext that is received by the receiver. The security of the system relies on the infeasibility of computing K2 from knowledge of K1.

2.4 Examples

It is illustrative to look at examples of practical public key encryption schemes. The Rivest Shamir Adleman cryptosystem was among the first produced. It is also among the most resilient to cryptanalysis and there has been no refutation of the postulation that breaking the system is at least equivalent to the problem of factoring. Exponential key exchange relies on the comparative difficulty in taking logarithms compared to raising a number to a power.

2.4.1 R.S.A. [1,7]

1) N is the product of 2 primes P and Q.

2) $\Phi(x)$ Euler totient function

   => number of numbers less than x relatively prime to x.

3) Euler's theorem $x^{\Phi(N)} \bmod N = 1 \Rightarrow x^{k\Phi(N)} \bmod N = 1$

4) Observe $\Phi(N) = (P-1)(Q-1)$

5) Pick e.

6) Calculate d such that $(e * d) \bmod \Phi(N) = 1$

$$\Rightarrow (e * d) = k\,\Phi(N) + 1$$

7) Publish e, N.

8) $C = M^e \bmod N$

9) $M = C^d \bmod N$

$$= M^{ed} \bmod N$$

$$= M^{k\Phi(N)+1} \bmod N$$

$$= M \bmod N$$

eg $P = 17$ $Q = 31$

$N = P * Q = 527$ $\Phi(N) = (P-1)(Q-1) = 480$

Choose $e = 7$ Calculate $d = 343$

If $M = 2$, $C = M^e \bmod N = 2^7 \bmod 527 = 128$

Decryption :

$$M = C^d \bmod N = 128^{343} \bmod 527$$

$$= 128^{256} * 128^{64} * 128^{16} * 128^{4} * 128^{2} * 128^{1} \bmod 527$$

$$= 2$$

because $128 \bmod 527 = 128$,

$$128^2 \bmod 527 = 16384 \bmod 527 = 47,$$
$$128^4 \bmod 527 = 47^2 \bmod 527 = 2209 \bmod 527 = 101$$
$$128^8 \bmod 527 = 101^2 \bmod 527 = 10201 \bmod 527 = 188$$
$$128^{16} \bmod 527 = 188^2 \bmod 527 = 35344 \bmod 527 = 35$$
$$128^{32} \bmod 527 = 35^2 \bmod 527 = 1225 \bmod 527 = 171$$
$$128^{64} \bmod 527 = 171^2 \bmod 527 = 29241 \bmod 527 = 256$$
$$128^{128} \bmod 527 = 256^2 \bmod 527 = 65536 \bmod 527 = 188$$
$$128^{256} \bmod 527 = 188^2 \bmod 527 = 35344 \bmod 527 = 35$$

and

$$35 * 256 * 35 * 101 * 47 * 128 \bmod 527 = 2$$

The primes chosen for use in an RSA application should be chosen according to the desired encryption block size with full regarded for the latest factoring developments. Depending on the application the choice of primes may or may not be done in real time. Public-key encryption is the main proposed application of this project but the aim of the project is to produce a library of multi-precision arithmetic routines on a digital signal processor which are compatible with cross-compiled C code. The discussion of various encryption algorithms is therefore mainly for background theory and it has not been attempted to be exhaustive in dealing with these algorithms. For this reason such issues as the choice of suitable RSA primes are left to be dealt with when a specific application has been chosen by any user of the library developed in this project.

2.4.1 Exponential Key Exchange [1]

1) X_A randomly chosen over 1,2,...,q-1

$$Y_A = \alpha^{X_A} \mod q$$

α is a fixed primitive element of GF(q) i.e known to all.

Alice publishes Y_A.

2) Bob publishes $Y_B = \alpha^{X_B} \mod q$

3) Alice calculates $K_{AB} = Y_B^{X_A} \mod q$

$$= \alpha^{X_A X_B} \mod q$$

4) Bob calculates $K_{AB} = Y_A^{X_B} \mod q$

5) If Y_A, Y_B only known, not X_A, X_B, must perform discrete logarithm on Y_A or Y_B.

2.5 Digital Signatures

If a public key cryptosystem has the commutative property :

$$E_{K2}\ (\ D_{K1}(M)\ )\ =\ M \qquad \text{and} \qquad D_{K1}\ (\ E_{K2}(M)\ )\ =\ M$$

it can be used to solve the problem of authentication i.e. to provide a digital signature. The ciphertext space and plaintext space must also be equal. Authenication is an important application which can solve disputes over message validity and origin. It is best illustrated by a diagram :

$$M \longrightarrow \underset{\text{A's secret key}}{D_{K1}(M)} \longrightarrow \underset{\text{B's public key}}{E_{K4}(D_{K1}(M))} \xRightarrow[\text{B's secret key}]{\text{Channel}} D_{K3}(E_{K4}(D_{K1}(M))) \xrightarrow[\text{A's public key}]{} E_{K2}(D_{K1}(M))$$

Fig. 2.8 *Using public-key encryption to produce a digital signature* [9]

Note: (K1, K2) and (K3, K4) are Alice's and Bob's (private key, public key) pairs respectively.

When M is calculated by Bob using Alice's public key it is clear that Alice and only Alice could have produced the cryptogram, using her secret key, so in effect the whole message is signed by Alice [9].

2.6 Evaluation of Encryption Schemes

Shannon outlined five criteria for the evaluation of cryptosystems in his "Communication Theory of Secrecy Systems" [9] :

(1) the amount of secrecy offered.

(2) the size of the key.

(3) the simplicity of the encryption and decryption operations.

(4) the propagation of errors.

(5) the extension of the message.

The level of security offered in public key schemes is always an important part of the number theory that is developed about each algorithm, even though it is possible to deal only in terms of computational security rather than absolute security with practical public-key encryption schemes. It is conjectured that the difficulty in cryptanalysing the RSA algorithm is equivalent to the factorisation of the product of the two primes which is used. This is a conjecture and so has not been proved. However a large amount of work investigating other methods of cryptanalysing the RSA has been done, so it is a reliable conjecture and it is reasonable to assume that it is true. For secure speech communication the simplicity of the enciphering and deciphering operations should allow operation at about 8kbits/s for natural telephone communication. This is subject to the complexity of the algorithm, block size used and method of implementation chosen. For any given algorithm with adjustable number theory problem size (like the factorisation problem) there are trade-offs between speed, security and cost. As successful block encryption schemes produce a ciphertext block which appears totally random, and a message which has just one single bit different to another message should produce a completely different ciphertext for reasons of security, it is important that the ciphertext received is not corrupted by the channel. This can be ensured by using error checking coding methods and re-transmitting incorrectly received data. Of course it may be acceptable that an occasional block or an occasional bit in a bit-stream encryption scheme is wrong, provided the final decrypted result is fully intelligible. No error can be allowed if a feedback back mode is used in encryption because the apparent randomness amplification which normally enhances security would amplify the error and cause the message after it to be incorrectly decrypted. RSA does not extend the message. Some other schemes do. Message extension is obviously a drawback but it usually has security enhancement properties so again it should be evaluated as being a possibly worthwhile trade-off.

2.7 The Blum, Blum and Shub Generator [8]

A **pseudo-random sequence** is a sequence which appears random but is in fact produced from a random starting point, termed the **seed**, by a deterministic process. A pseudo-random generator is **cryptographically strong** if the sequence it produces from a short seed is essentially as good as a random sequence for use as a one-time pad. This means that it should not be computationally feasible to derive any probabalistic information about the plaintext from the ciphertext which is the result of an eXclusive-ORing of the plaintext and the pseudo-random bit stream.

The Blum, Blum and Shub generator is defined recursively :

$$x_{i+1} = x_i^2 \bmod N$$

with N a Blum integer, i.e. the product of two primes P and Q both congruent to 3 mod 4, and x_0 a random quadratic residue. The sequence of least significant bits of x_i is cryptographically secure and recent research suggests that up to $\log_2$ of the number of bits in N bits of each x_i may be secure [13]. The BBS generator can be used as a one-time pad with N public domain and exponential key exchange of x_0.

Another feature noted in "A Simple Unpredictable Pseudo-Random Number Generator" [8] allows a complete public key cryptosystem :

Knowledge of N leaves the sequence unpredictable to the left but knowledge of N's factorization allows polynomial time calculation of the previous x_i's. Therefore N provides the encryption key, P and Q, the secret decryption key. In this scheme Alice publishes N, keeping P and Q secret. Bob picks x_0 at random and eXclusive-ORs the message with the resultant random bit stream, transmitting this cryptogram and the next x_i in the sequence. Only Alice using her knowledge of P and Q can determine any term to the left of x_i thus decrypting the cyphertext.

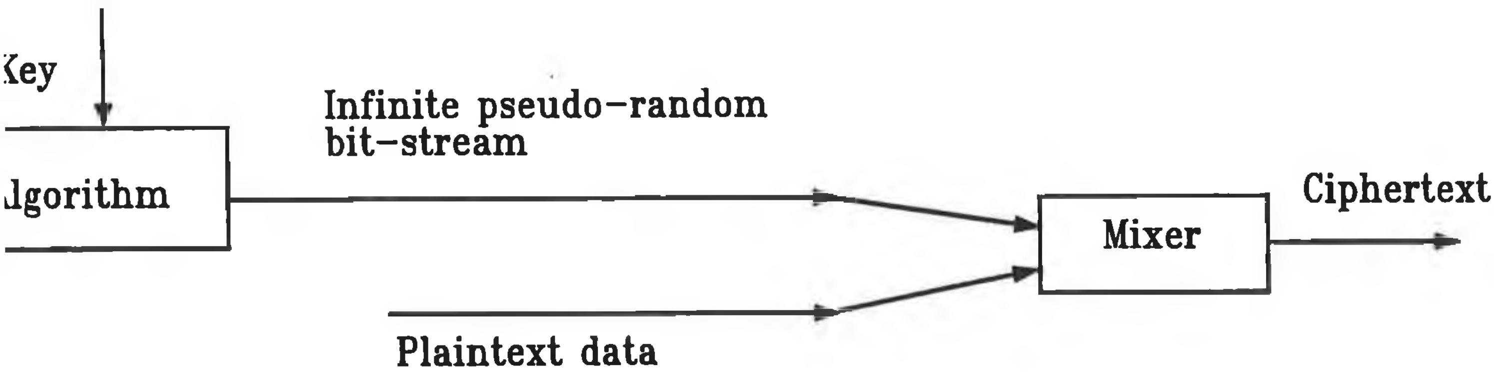


Fig. 2.9 *A bit-stream encryption system*

3 INTRODUCTION TO FUNCTION DEVELOPMENT

3.1 Introduction to Library Development

Public key encryption algorithms involve large number arithmetic operations which are not yet supported in hardware in a general purpose form. One hundred digit numbers can now be factored and the RSA encryption algorithm must use between one hundred and fifty and two hundred digit numbers to stay ahead of the latest advances in factoring algorithms and machine technology. Most of the scientific and applied mathematical applications which require microprocessor-based systems are considerably more dependent on such features as input-output capability and on-chip memory size than on numerical processing power.

The main area of processor use which does require high speed number-crunching is digital signal processing. In such applications as digital filtering, correlations and FFTs, there is a very large proportion of processor time spent performing the fundamental arithmetic operations, particularly multiply and accumulates. This market demand has resulted in the development of digital signal processor chips (DSPs). These devices differ from general purpose microprocessors in their use of a Harvard architecture which allows a higher through-put as instructions are fetched and executed in parallel. A Harvard architecture requires that program and data memory reside in separate address spaces. This is different to the Von Neumann architecture used in most general purpose microprocessors in which the only distinction between a program instruction and a data word is context. DSPs also provide a hardware multiply, and this operation in particular can be executed in parallel with such operations as accumulate and data move which is particularly useful in the implementation of digital filters.

Digital signal processors therefore offer some hardware support for applications like encryption which require fast multiplies. It is necessary in developing encryption functions to evaluate the available DSPs with particular emphasis on their instruction cycle time and how much the parallel capabilities of the chips can be used to improve system throughput. Once a particular digital signal processor has been chosen, the next step is to develop the actual routines. General software design principles of modularity and simplicity of coding should be adhered to, except where there is a significant timing improvement to be gained by, for example, writing in-line code or making more use of the parallel capabilities of the DSP.

The final evaluation of multi-precision arithmetic routines for use in encryption applications must look not only at the speed of operation within real time constraints but also at the ease of use of these routines. The plan in developing these routines has been to try to allow a programmer with minimal understanding of the hardware and the language of the digital signal processor used, to develop applications in C which can call the routines. With this aim in mind, an effort has been made to make these routines compatible with the MIRACL Package [5] with the exception of the input and output functions which naturally must be handled differently on a digital signal processor.

3.2 Digital Signal Processing Chips

The most important development in electronics during the 1970s was the single chip microprocessor. It lead to the development of the personal computer and the widespread possibility of intelligent control. With applications as diverse as washing machine programming and nuclear weapon systems control, the problem for engineers became one of discovering new uses for this new found processing power. Following on from this area of microelectronics development has been the advent of a new type of device designed to allow the more versatile digital algorithms to take over from analog signal processing. The first general-purpose, single-chip, 16-bit digital signal processor was introduced by NEC in 1980 [14]. Since then every major chip manufacturer has produced a digital signal processing chip.

A digital signal processor is a form of microprocessor with three main features [15]:

1. Its architecture is optimised to process sampled data at a high rate.

2. In particular, it achieves this high rate of through-put by having fast multiply and accumulates.

3. It exploits the repetitive nature of signal processing by pipelining data flow for extra speed. This pipelining feature is usually achieved by having a Harvard architecture in which the separateness of program and data memory allow the next instruction to be fetched while the present instruction is executing.

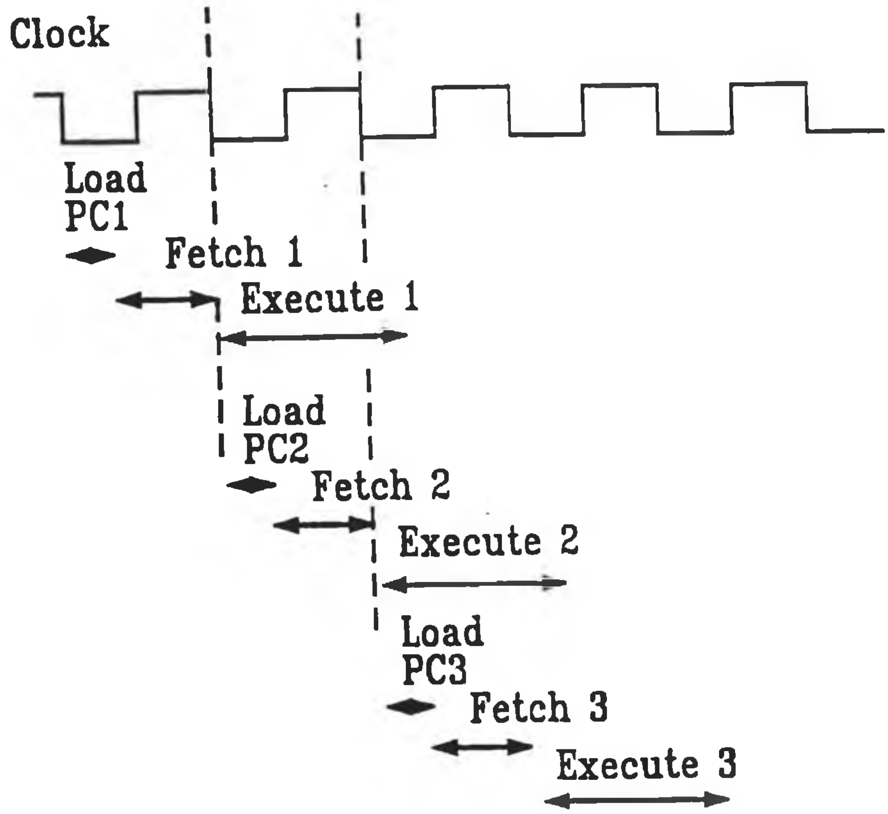


Fig. 3.1 *Instruction pipelining involves instruction 'execute's in parallel with instruction 'load's and 'fetch'es* [15]

While all digital signal processors have these features in common to some extent, each manufacturer emphasises particular features in their own device. NEC state that their basic philosophy in the design of the 7720 family of DSP's has been "to integrate as many resources (memories, periherals) as possible into a DSP to obtain a device that is both powerful and compact. This is due to the fact that as input/output operations increase, the performance of a DSP decreases sharply" [14]. This design principle is one that is noted by all DSP manufacturers. One area of design where a different approach is taken by different manufacturers is the degree of parallel capability offered by their device. NEC uses a highly parallel "horizontal" instruction set. This offers timing benefits for suitable applications in which register manipulation and arithmetic operations can occur simultaneously. Code taking one instruction cycle on the NEC77230 may resemble a page of code comprising of tens of operations, each requiring an instruction cycle, on another signal processor, as many operations are carried out in parallel. Along with the timing benefits these parallel operations produce, the NEC device is difficult to code and its lack of user-friendliness can be a drawback. It is also notable that the parallel programming capability of the NEC device is achieved with a considerably longer instruction cycle than that of the TMS320C25 : 250 nanoseconds as against 100 nanoseconds, so the NEC device is less suitable for applications which do not make full use of its parallel capabilities. Most multi-precision arithmetic operations make use of the parallel capabilities of a DSP primarily when using the basic single-precision add, subtract and multiply instructions, so the Texas Instruments processor provides the parallelism sought after, with a shorter instruction cycle than the NEC DSP.

The TMS320C25 produced by Texas Instruments can do a 16-by-16 bit multiplication in one instruction cycle of 100 nanoseconds. The Motorola MC68020 general purpose microprocessor, unlike digital signal processors, does not have a hardware multiply and as a result requires 25 instruction cycles or 1500 nanoseconds at 60 nanoseconds per instruction cycle to carry out this operation [16]. Texas Instruments have a "vertical" instruction set which is considerably easier to use than the NEC instruction set. T.I. reckon that the throughput of a DSP can be significantly improved by making only a few operations parallel [15].

In summary : the TMS320C25 is easier to program, has a shorter instruction cycle and more on-chip memory than the NEC equivalent. Its main advantage over the DSP16 produced by AT&T Bell Laboratories is its larger on-chip memory size which can ease bottle-necks in input and output [16]. Certain features of the TMS320C25 facilitate the implementation of multi-precision arithmetic. The device has an unsigned multiply instruction (MPYU) which produces the 32-bit product of two 16-bit unsigned numbers in one machine cycle. Extension beyond single precision is made easier by the absence of a sign bit as the 32-bit product value does not need to be split into 15-bit parts but instead can use its full range for a product result.

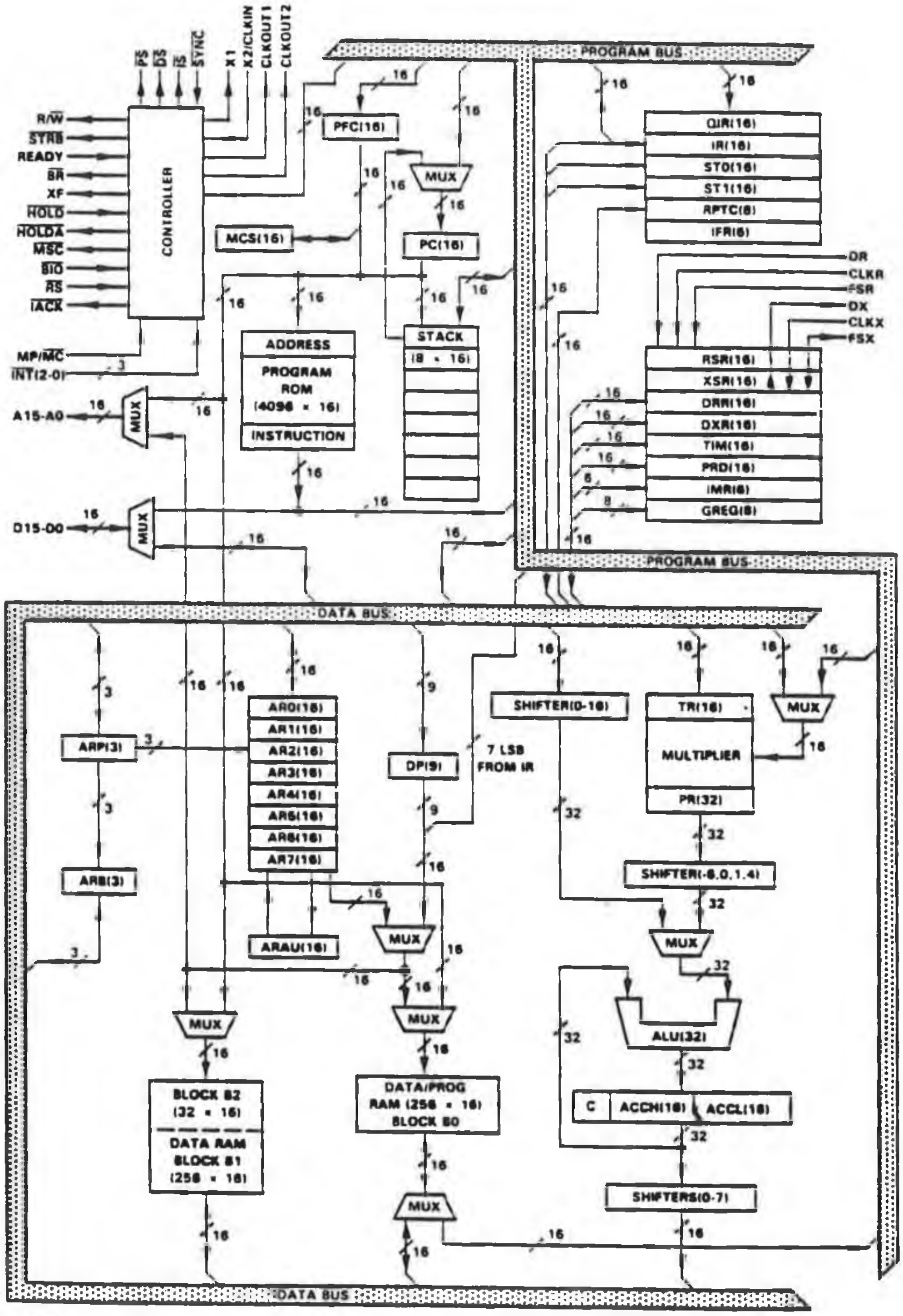
The carry status bit is a hardware flag which is affected by all the arithmetic operations of the accumulator and by the rotate and shift accumulator instructions. The carry bit is set when an addition overflows the 32-bit accumulator and it is reset when a subtraction results in a borrow into the most significant bit of the accumulator. It may also be explicitly set or reset. Multi-precision addition and subtraction can use the carry status bit in a software implementation similar to a hardware configuration involving half and full adders and subtractors which access the carry bit to determine if a previous operation resulted in a carry or a borrow.

While single-precision multiplication is implemented in hardware on the TMS320C25, single-precision division can be coded simply although with a higher timing overhead. The DSP has a repeat instruction RPTK, which, used in conjunction with the special condition subtraction instruction SUBC, can divide a 16-bit dividend by a 16-bit divisor placing the quotient in the 16 low-order bits of the accumulator and the remainder in the 16 high-order bits, in 17 machine cycles. This entire operation is coded in two lines once the accumulator has been loaded with the dividend:

```
RPTK    15
SUBC    DIVISOR
```

RPTK N results in the instrution which follows it being executed N+1 times [17].

Digital signal processors have features which make the implementation of multi-precision arithmetic routines possible in faster time than on general purpose microprocessors. The TMS320C25 has comparatively large on-chip memory including 4K words of data ROM and a short instruction cycle time. In addition its software development tools including the Software Development System (SWDS) and the TMS320C25 C Compiler make it a suitable device for the development of these routines.



LEGEND:

ACCH = Accumulator high
ACCL = Accumulator low
ALU = Arithmetic logic unit
ARAU = Auxiliary register arithmetic unit
ARB = Auxiliary register pointer buffer
ARP = Auxiliary register pointer
DP = Data memory page pointer
DRR = Serial port data receive register
DXR = Serial port data transmit register
IFR = Interrupt flag register
IMR = Interrupt mask register
IR = Instruction register
MCS = Microcall stack
QIR = Queue instruction register
PR = Product register
PRD = Period register for timer
TIM = Timer
TR = Temporary register
PC = Program counter
PFC = Prefetch counter
RPTC = Repeat instruction counter
GREG = Global memory allocation register
RSR = Serial port receive shift register
XSR = Serial port transmit shift register
AR0-AR7 = Auxiliary registers
ST0,ST1 = Status registers

Fig. 3.2 *TMS320C25 Hardware block diagram* [17]

3.3 Design Philosophy

Before looking at the specifics of the problems involved in the development of a real time assembly language library it is a good idea to stand back from the problem and to take a general overview of algorithms and software design which can be applied to the task at hand. According to Knuth [18] an algorithm is distinguished by having five important features :

1. Finiteness. An algorithm must always terminate after a finite number of steps.

2. Definiteness. Each step of an algorithm must be precisely defined.

3. Input. An algorithm has zero or more inputs.

4. Output. An algorithm has at least one output and possibly more.

5. Effectiveness. It should be possible to carry out the steps of an algorithm using pencil and paper in a finite length of time.

If a multi-precision arithmetic algorithm is properly defined its inputs and outputs should be clear and the steps involved should be easily determined to result in the termination of the algorithm in a finite amount of time. These then are three important points which should be examined in the early stages of looking at the algorithm. The routines developed have been based on Knuth's classical algorithms.

Full input/output specification not only shows very early in the design stage exactly what an algorithm will be able to do, but also, often more importantly, it shows what an algorithm will not be able to do. An example is Knuth's "division of nonnegative integers" which requires a multi-precision divisor (i.e. at least double precision). This is not a mistake : the algorithm does precisely what it sets out that it will do in the input/output specification. However to make the algorithm work for all cases, it is necessary to explicitly cater for the single precision divisor case. This fact is readily apparent only because of the use of an input/output specification.

Definiteness will be imposed on the problem by the act of coding but it is important that the algorithmic description must be made up of precisely defined and understood steps so that no incorrect operations result from a misinterpretation or an inadequete specification. The final feature, effectiveness, is more of an empirical concept. It is a requirement that the algorithms used should be effective so that they can operate in real time.

A good way to approach the problem of algorithm effectiveness is to attempt to look at it in both the small and large scale. The small scale should ensure that as efficient use as possible is made of the processor's hardware and instruction set, particularly in high iteration loops. The large scale should involve trying to minimise the use of high iteration sections as much as is possible [19]. Looking at the problem in this way means that on the small scale good assembly coding is important while on the large scale it is important that an efficient algorithm is chosen.

It is difficult to adhere to some software design principles when programming in assembly language. High level software projects usually allow comprehensive testing of the algorithms used when the code has been written. This is because the code can be made modular and clearly defined. Often complications and errors can be identified in the algorithm as well as in the code because the code is relatively easy to understand. In comparison assembly language, even when written as clearly as possible, is more difficult to debug. The algorithm being implemented should be tested comprehensively first, preferably both by implementation in high level language and a complete walk-through examination.

There appears to be very little literature availible on design principles for low-level language applications. The reason for this is that in general the same good programming practices should be adhered to as for high level language applications, and even though these practices may be harder to apply in assembly language development, they are more necessary the lower the level of the language used. Some idea of the large increase in complexity due to implementation in assembly language is given by David Wong, the Director of Signal Processing at Digital Sound, when he states that a rough estimate at the size of a program written in DSP assembly can be made by multiplying the number of lines in a high-level simulation by ten [20]. It is no wonder therefore that simplification ideas which may seem superfluous in a small high-level application are vital in almost all low-level applications. One recommendation is that clear use should be made of symbols to represent "magic numbers". "Magic numbers" are constants and address values which make the program easier to read and to change if assigned a symbol at the start but which can appear confusing if they are written within the program as their numerical value. Comments are also more necessary the lower the language level used. High level languages should comment themselves as much as possible but assembly language cannot do this to the same extent. If there is too much commenting of code, it can be easily ignored but under-commenting makes code very difficult to decipher [15].

Flexibility in the way a function can be called is a desirable characteristic to include in the design. If the general operation of the add function is to add x to y giving result z after a call of the form : add(x,y,z), it would be an asset to allow the programmer to call the function with forms : add(x,y,x) or add(x,y,y), thus facilitating update of a variable in one step. However this flexibility often requires an overhead both in memory within the function, as a temporary variable will have to be stored, and, more importantly, in the number of instructions used to copy to and from this temporary variable. It has therefore been decided that since the aim of the library is to allow fast multi-precision calculations, this type of additional functionality will only be provided where it fits in with the algorithm without a significant overhead.

The restrictions on the calling function are listed in each function description so it will be clear where :

```
add(x,y,z);
copy(z,y);
```

will have to be used instead of :

```
add(x,y,y);
```

3.4 Test Philosophy

The starting point for these routines has been Knuth's description of the classical algorithms for multi-precision arithmetic. In Seminumerical Algorithms [6] the algorithms are given in a form which is purposely not machine dependent. Their implementation is then worked through on a theoretical processor. In order to code these routines on the TMS320C25 a representation closer to the workings of that device is necessary. The design of this function therefore started with the initial algorithm in a step form with little attention payed to the hardware and the code that would be used. Next, an outline similar to the low level algorithmic description which is given for each function in the next chapter, was produced and the code was written according to this description.

The process of producing the final versions of the algorithm steps and then the code was not entirely sequential as would be implied in a completely top-down design procedure. An effort was made to produce optimal coding of some of the more crucial low level steps (i.e. those in the loops with the highest number of iterations) and then to incorporate these as efficiently as possible into the algorithm. The low level algorithmic descriptions should be close enough to the more general descriptions in Knuth's work to be verifiable while they are also easy to relate to the actual code. Thus they facilitate full test of those features of the algorithm which must be clearly defined, while remaining close enough to the code to allow it to be tested in blocks corresponding to each step. This is about as close to modularity in design as is feasible in real time assembly language library development.

Once the algorithm has been verified in this manner and the code has been written, it is necessary to test the code. Errors are easy to make in assembly language programming. Mistakes can be made in the coding procedure by selecting the wrong auxiliary register or failing to note which hardware flag gets set by particular instructions. The code has to be as modular as possible and attention should be payed to having correct and failsafe terminating conditions on all loops.

The procedure chosen for testing the routines has been to examine every case of each algorithm, as much as it is possible to do so. This means that each algorithm step, particularly if its execution is optional, should be tested at least once. The test cases can be derived from the low level algorithmic description. When this method results in more than three or four test cases being necessary, the testing procedure is made clearer by illustrating exactly what steps each case tests, on a test case grid. This method should result in a degree of confidence in each finished routine. A less exhaustive but more realistic test of the routines can then be carried out. This involves checking the routines against each other with problems which are of a precision size that is likely to used in the encryption applications. For example division, multiplication and addition can be cross checked by comparing the results of multiplying the quotient result of a division by the divisor and adding the division remainder.

The use of test case grids to test software applications in which exhaustive test is not feasible was encountered by the student during work with the Quality Assurance Group of ROLM Systems. This group had responsibility for the development of test plans to be used in the testing of new releases of the PhoneMail voice messaging system.

3.5 Number Representation

The method used to represent numbers in the assembly language library is based on the MIRACL package. Unlike MIRACL, functions are only provided to operate on positive integers which is all that is necessary for all the well known public-key encryption algorithms. No use is made of a sign bit as it is not needed. Brent in his Fortran Multiple-Precision Package [21] uses the first word in an array to give the sign of the number, the second word lists the exponent and the rest of the array contains the number. Buell and Ward in their Multiprecision Integer Arithmetic Package [22] use the same system as MIRACL : the first array place contains the number of multiprecision digits in the number and the sign of the first array place is the sign of the number. Zero is represented by a zero in the first array place which represents size and also a zero in the second place.

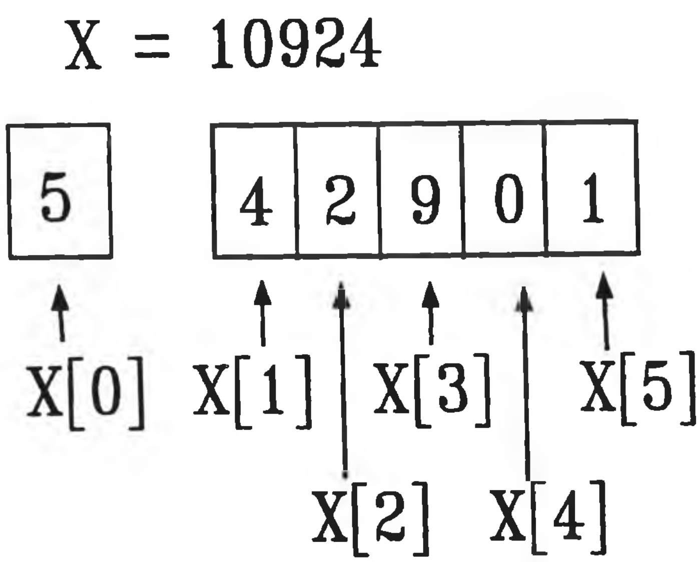


Fig. 3.3 *An example of number representation method used. Base 65536, not decimal, is used in the library.*

The above number, 10924, would be represented in this thesis and in programs which use the assembly language functions as : X[0]=5, X[1]=4, X[2]=2, X[3]=9, X[4]=0, X[5]=1 or X={ 5,4,2,9,0,1 }, if a decimal base was used. This would be equivalent to saying : "There are 5 significant digits in the number in question. The number is $4*10^0 + 2*10^1 + 9*10^2 + 0*10^3 + 1*10^4$." In the assembly language package base 10000_{16} (65536_{10}) is used so the first digit has the same meaning and following digits are weighted by $(10000_{16})^0$, $(10000_{16})^1$, $(10000_{16})^2$ and so on. For example the number { 2,0,1 } is $0*(10000_{16})^0 + 1*(10000_{16})^1$ or 65536 in decimal.

3.6 Development Environment

The assembly language functions which form the multi-precision arithmetic library were developed using the TMS320C25 Software Development System (SWDS) [23]. The SWDS is a software development tool that provides both hardware and software support for application development on the TMS30C25. The system's hardware consists of a board which plugs into the expansion bus of a personal computer and which contains a TMS320C25 digital signal processor and 24K words of program and data memory. Additional hardware facilities are provided for input/output from a target system. The software allows TI-tagged object format files to be loaded and run. Debug facilities include single-step and breakpoint operation and the values of the registers and hardware flags of the DSP can be observed. The SWDS used was release version 1.0.

The TMS320C25 C Compiler [24] and the TMS320C1x/TMS320C2x Assembly Language Tools [25] were also used in library development. The compiler converts C code into assembly language code. The assembly tools assemble, link and perform object format conversions on assembly files. An archiver is also availible for the production of function libraries. Therefore there are facilities to compile a C program which calls the low-level multi-precision arithmetic DSP functions and the linker will ensure that the program which is loaded onto the TMS320C25 system contains instructions derived from both the easy-to-write high level calling program and the efficient low-level routines.

3.7 Function Useage

The multi-precision arithmetic functions are compatible with cross-compiled C code. The aim in developing these functions has been to allow calls from C code using the conventions of the MIRACL Package. This means that the operands should be declared as unsigned integer arrays, with the first array position containing the number of array places used to represent the multi-precision number. This is equivalent to the precision of the number while the maximum number representable in each array place plus one is the base. For example, the following C program could be used :

```
main()
{
unsigned int x[8]; y[8]; z[8];

x[0]=7; x[1]=6; x[2]=5; x[3]=4;
x[4]=3; x[5]=2; x[6]=1; x[7]=9;
y[0]=2; y[1]=8; y[2]=5;

add(x,y,z);
}
```

The number represented by the Z array would be fully defined as would X and Y after the call to add( ). The following command line instructions are used to produce code which can be run on the Texas Instruments Software Development System (SWDS) :

```
DSPA add.asm
DSPC call
DSPLNK -C call.obj add.obj -O call.out
DSPROM -T call.out
```

These instructions result in the following actions being taken by the assembler, compiler, linker and object file format converter : the addition function assembly file is assembled and the C file is cross-compiled, producing two object files. These object files are then linked together to produce an output file. The call object file must be specified to the linker first because the linker produces an undefined reference to the add( ) function which must be resolved by the add object file. The -C option specifies a ROM autoinitialisation model. Finally an object format conversion is done, producing code which can be loaded by the SWDS.

In order to facilitate easy linking of programs which use several multi-precision functions, all of the assembly language object files have been archived, producing a library file call MP.LIB. The above linking command can be replaced by :

```
DSPLNK -C call.obj -L  mp.lib -O call.out
```

This command successfully resolves undefined references to any number of the multi-precision arithmetic functions.

After the SWDS is invoked, the call.tag file is loaded and a debug session is started. As described above, no entry point has been specified so the program starts from a default position of 1000 Hex in program memory and the program counter should be set to this value. If the -E *global symbol* option is used when invoking the linker, the *global symbol* will be the primary entry point for the output module so the program counter will be set to this value. Also before running, the Auxiliary Register Pointer must be set to '1' and both AR0 and AR1 must be loaded to point to some appropriately chosen value in data memory. AR0 is the Frame Pointer which points to the beginning of the current frame. A new frame is created for each function at the top of the stack and both local and temporary variables are stored there. AR1 is the Stack Pointer which points to the current top of the stack or the word following the current top of stack [24]. This value will be the start of stored variables used by the program and it should be high enough to avoid globally defined variables being written over. Functions in the library select data page number six and thus store local temporary variables at 300 hex plus their EQU directive defined value. Therefore a value for the start of stacks must be chosen which is higher than the highest EQU defined local variable address plus 300 hex.

An example of register settings and memory assignments follows :

PC :	1000	AR0 :	31B
ARP :	1	AR1 :	31B

(set after loading program into the Software Development System, SWDS)

Data Memory : Local/Temporary Variables => 300h → 31Ah
(assigned in assembly language functions)
: Function Variables => 31Bh + ...
(assigned by compiler).

It is a convention with the TMS320C25 compiler to return return values in the accumulator. The compare function adheres to this convention allowing easy control of program flow based on comparing multi-precision number values. An example of the use of a return variable is shown in the function description for the compare function.

3.8 Dynamic Memory Allocation

Memory allocation functions such as malloc(), calloc() and free() are supposed to be fully supported by the TMS320C25 C Compiler. These are useful functions for the type of programming application in which large amounts of memory space are required for different variables for some of the program, as they facilitate run-time initialisation of memory and allow memory to be freed when it is no longer required. These functions have been observed to cross-compile successfully but the monitoring of a program containing a calloc() showed that this function returned a pointer to availible memory starting at address zero even though memory from this value is not availible for program use as it is required by the monitor. This problem was not resolved by attempting to specify a valid start address for free memory by specifying a .bss section in a program link command file. Therefore the cross compiler does not appear to support the memory allocation functions and they have not been used with the assembly language functions from the library.

4 ASSEMBLY FUNCTIONS

4.1.1 ADDITION FUNCTION

Function:	void add(x,y,z) unsigned int x[SIZE],y[SIZE1], z[SIZE+1];
Files:	add.asm, add.obj
Description:	Adds two multi-precision numbers.
Parameters:	Three unsigned integer arrays in big format. On exit z = x + y.
Return Value:	None
Restrictions:	x >= y i.e. SIZE >= SIZE1. z must be distinct from y : add(x,y,x) allowed but add(x,y,y) not allowed. z must be one precision bigger than x to allow carry.

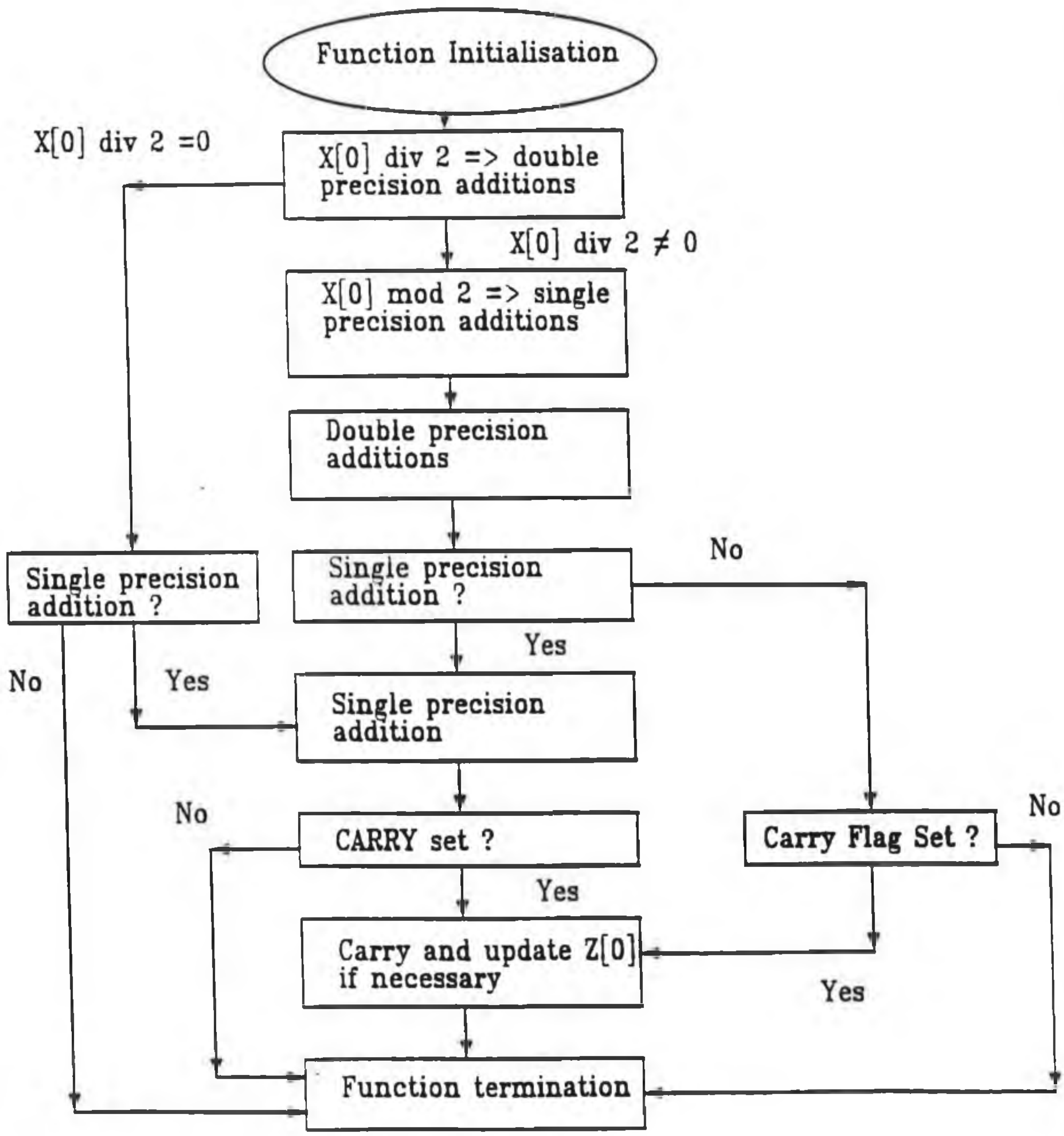


Fig. 4.1 *Block diagram showing the main cases in the addition algorithm*

4.1.2 Low Level Algorithm Description : ADD

1. Function initialisation.

2. Register and precision count iteration set up.
   Register store. Copy : Z := X. Z[N+1] = 0 (to facilitate carry propagation).
   Double precision count = Y[0] div 2. Single precision count = Y[0] mod 2.
   If no double precision additions, go to 4.

3. Double precision add loop.
   If single precision add left, go to 5.
   Else go to 7.

4. Set up for no double precision add.
   If no single precision addition left, go to 9.
   Zero carry.

5. Single precision add.
   If carry, go to 6.
   Else go to 9.

6. Single precision carry.
   Propagate carry forward and go to 8.

7. Double precision carry.
   Propagate carry forward.

8. Z[0] adjust.
   Check if Z[0] should be incremented and increment if necessary.

9. Function termination.

4.1.3 Explanation of Z[0] Adjust

Z[0] needs to be incremented after the single precision and double precision additions have been carried out only if the last addition ended in a carry. This occurs when the last addition operation results in a carry and there are the same number of places in X and Y or when a propagated carry results in the last Z term equalling '1'. This last case is when the carry propagation generates a new Z term because the previous X term was equal to the base minus one and propagated a previous borrow forward.

4.1.4 Implementation Notes

There are two points which should be noted resulting from the implementation method chosen. Firstly it is taken as convention that when add(x,y,z) operates on the two numbers X and Y, that X is greater than or equal to Y. This is necessary for correct carry propagation and Z[0] increment. Secondly if there are up to N places defined in X, there should be N+1 places in Z so that a carry at the end of X can occur without error or overwriting of other data.

4.1.5 Test Case Grid: ADD

In the following test case grid the major program/algorithm steps are listed as column labels. Any particular test case will result in some steps being used, represented by a "one" being entered into the appropriate column, and other steps not being used, represented by a "zero" in those columns. A case number is allotted to each case so that when a test case is being produced it is possible to look at the grid and to say, for example : " Test case number 3 needs a single precision add with carry. This should result in a sum which has more digits than either of the summed numbers (i.e. a Z[0] adjust). { 1,FF88 } + { 1,78 } will involve all of these steps, so it is an an appropriate test example for this case number". The idea of the test case grid is to facilitate and to document the exhaustive test of all steps, taking into account the fact that if two steps have no interaction in the way that they have been implemented, they can be tested in parallel. This means that the testing of five major steps does not require thirty two test cases and it also allows the lack of interaction between some of the steps to be spotted more easily.

Double prec. add	Single prec. add	Single prec. carry	Double prec. carry	Z[0] inc.	Case
0	0	0	0	0	1
0	1	0	0	0	2
0	1	1	0	1	3
1	0	0	0	0	4
1	0	0	1	1	5
1	1	0	0	0	6
1	1	0	1	0	7
1	1	1	0	0	8
1	1	1	1	0	9

Fig. 4.2 *Addition test case grid.*

4.1.6 Test Cases

1. { 0,0 } + { 0,0 } = { 0,0 }

2. { 1,FF07 } + { 1,19 } = { 1,FF20 }

3. { 1,FF88 } + { 1,78 } = { 2,0,1 }

4. { 3,9,0,FF00 } + { 2,3,FF00 } = { 3,C,FF00,FF00 }

5. { 4,4,3,2,1 } + { 4,4,3,FFFF,FFFE } = { 5,8,6,1,0,1 }

6. { 6,6,5,4,3,2,1 } + { 5,5,4,3,2,1 } = { 6,B,9,7,5,3,1 }

7. { 5,5,4,3,2,1 } + { 5,1,FFFF,2,FFFF,1 } = { 5,6,3,6,1,3 }

8. { 6,6,5,4,3,2,1 } + { 5,5,4,3,2,FFFF } = { 6,B,9,7,5,1,2 }

9. { 4,1,2,3,4 } + { 3,8,FFFF,FFFC } = { 4,9,1,0,5 }

These cases were tested during a SWDS debug session for add(x,y,z) and add(x,y,x). The above results were obtained as expected.

4.1.7 Timing

Knuth's implementation of addition in his MIX language requires 10N + 6 cycles where N is the precision of the numbers being added. In MIX the size of a word is not specified exactly : it is only stated that each word can represent at least 64 distinct values and not more than 100 distinct values. The maximum base for multi-precision operations is therefore 100. The word length on the TMS320C25 is 16 bits, giving a base of 65536. The timing advantage due to the larger word size alone of the routine implemented on the DSP is a factor of over 655.

The implementation of the addition algorithm requires : 7M + 3N + 47 instruction cycles where N is the precision of the larger number, X, and M is the precision of the smaller number, Y. The MIX machine has an instruction cycle of at best 1 microsecond based on what Knuth estimated to be the cycle time of a relatively high-priced machine. The TMS320C25 has an instruction time of 100 nanoseconds. The timing value given above for the implementation of this function does not include a fixed overhead for function initialisation and termination. The above value is for comparison with Knuth's MIX implementation which would also have additional instructions to make it a function which could be called. The overhead amounts to 12 more instructions so calculations of timings involving the addition routine should use the value : 7M + 3N + 59.

Knuth's MIX implementation :	10N + 6
TMS320C25 code :	7M + 3N + 47
TMS320C25 independent function :	7M + 3N + 59

4.2.1 SUBTRACTION FUNCTION

Function:	void sub(x,y,z) unsigned int x[SIZE],y[SIZE1],z[SIZE];
Files:	sub.asm, sub.obj
Description:	Subtracts two multi-precision numbers.
Parameters:	Three unsigned integer arrays in big format. On exit z = x - y.
Return Value:	None
Restrictions:	x >= y z must be distinct from y : sub(x,y,x) allowed but sub(x,y,y) not allowed.

4.2.2 Low Level Algorithm Description : SUB

1. Function initialisation.

2. Register and precision count iteration set up.
   Register store. Copy : Z := X. Double precision count = Y[0] div 2.
   Single precision count = Y[0] mod 2.
   If no double precision subtractions, go to 5.

3. Double precision subtraction with carry generation.
   If more double precision subtractions left, go to 4.
   If single precision subtraction left, go to 6.
   If borrow to be taken care of, go to 7. Else go to 8.

4. Double precision subtraction loop.
   If single precision subtraction left, go to 6.
   If borrow to be taken care of, go to 7. Else go to 8.

5. Set up for no double precision subtraction.
   If no single precision subtraction left, go to 8.
   Else register set up for single precision subtraction.

6. Single precision subtraction.

7. If borrow, propagate borrow forward.

8. Function termination.

4.2.3 Explanation:

The central operation used in the implementation of this function is a double precision subtraction with hardware generated borrow. The carry bit is set to '1' if the result of an addition generates a carry, or reset to '0' if the result of a subtraction generates a borrow. Otherwise, it is reset after an addition or set after a subtraction, except if the instruction is an ADDH or a SUBH. ADDH can only set and SUBH only reset the carry bit but do not affect it otherwise [17]. The carry bit can therefore be '0' before an initial double precision add or subtract operation only if a multi-precision subtraction function ensures that this is not interpreted as a borrow. This can be done by having an ADDS as the first operation of the first double precision subtraction, as this instruction does not check for the carry bit being set, but sets or resets the carry bit depending on whether a carry is generated. This means the initial '0' carry bit value has no undesired effects. Also the first subtraction operation within this first double precision subtraction uses the SUBS instruction which similarly does not check the carry bit for a borrow but does set or reset the carry bit appropriately. The next double precision subtraction sum(s) must interprete the carry bit as a possible borrow, so these should be implemented differently with a subtraction operation first which does check for a borrow (SUBB). The easiest way of describing the multi-precision implementation is that the first double precision subtraction is like that carried out by a half subtractor with no borrow-in, while succeeding operations are like full subtractors.

4.2.4 Test Case Grid: SUB

Double precision with carry generation	Double precision loop	Single precision	Borrow	Case number
0	0	0	0	1
0	0	1	0	2
0	0	1	1	3
1	0	0	0	4
1	0	0	1	5
1	0	1	0	6
1	1	0	0	7
1	1	0	1	8
1	1	1	1	9

Fig. 4.3 *Subtraction test case grid*

4.2.5 Test Cases

1. { 0,0 } - { 0,0 } = { 0,0 }

2. { 2,7,6 } - { 1,5 } = { 2,2,6 }

3. { 2,7,6 } - { 1,9 } = { 2,FFFE,5 }

4. { 3,7,6,5 } - { 2,6,5 } = { 3,1,1,5 }

5. { 5,7,6,5,4,3 } - { 2,8,6 } = { 5,FFFF,FFFF,4,4,3 }

6. { 4,6,3,7,2 } - { 3,5,4,3 } = { 4,1,FFFF,3,2 }

7. { 5,7,6,5,4,2 } - { 4,1,7,5,1 } = { 5,6,FFFF,FFFF,2,2 }

8. { 7,7,6,5,4,0,0,2 } - { 4,8,6,5,4 } = { 7,FFFF,FFFF,FFFF,FFFF,FFFF,FFFF,1}

9. { 7,7,6,5,4,3,2,1 } - { 5,1,2,3,4,5 } = { 7,6,4,2,0,FFFE,1,1 }

These cases were tested during a SWDS debug session and the expected results were obtained.

4.2.6 Timing

Knuth's MIX implementation :	$12N + 3$
TMS320C25 code :	$7 \lfloor M/2 \rfloor + 3N + 36.75$
TMS320C25 independent function :	$7 \lfloor M/2 \rfloor + 3N + 48.75$

Note : $\lfloor M/2 \rfloor$ means the integer part of M/2 i.e. the division result is truncated. This notation is used throughout the report.

Knuth's N is the precision of the numbers being subtracted. The TMS320C25 N is the precision of the number from which an M precision number is being subtracted.

4.3.1 MULTIPLICATION FUNCTION

Function:

```
void mult(x,y,z)
unsigned int x[SIZE1],y[SIZE2],
z[SIZE1 + SIZE2  - 1];
```

Files: mult.asm, mult.obj

Description: Multiplies two multi-precision numbers.

Parameters: Three unsigned integer arrays in big format. On exit z = x * y.

Return Value: None

Restrictions: z must be distinct from both x and y

Example: `mult(x,x,z); /* This squares x */`

4.3.2 Low Level Algorithm Description : MULT

1. Function initialisation.

2. Register set-up and partial product initialisation.
   W[0] ← n + m. W[1]..W[n] ← 0. Store &U[n] and &V[m].
   Zero carry. V[j] counter set to point to V[1].
   W[i+j] counter set to point to W[n+1].

3. Initialise i.
   U[i] counter set to point to U[1].
   W[i+j] counter ← W[i+j] counter - n.
   Comparison register (AR0) ← &U[n].

4. Multiply and Add.
   t ← U[i] * V[j] + carry.
   W[i+j] ← t mod BASE. (BASE = 10000 (hex) = 65536).
   Carry ← $\lfloor$ t/BASE $\rfloor$. i ++, i+j ++.

5. Loop on i.
   If i ≤ n, go to 4, else W[i+j] ← carry.
   Zero carry.

6. Loop on j.
   j++. If j ≤ m, go to 3.

7. W[0] adjust.
   If W[n+m] = 0 decrement W[0] by one.

8. Function termination.

4.3.3 Explanation

This algorithm is similar to the conventional pencil-and-paper method in which the partial products U[1]...U[n] * V[j], $1 \leqslant j \leqslant m$ are calculated. Unlike the pencil-and-paper method in which all the partial products are summed at the end, it is more convenient for the processor to add each partial product within each V[j] ($1 \leqslant j \leqslant m$) multiplication loop.

The n least significant places of the product result, W[1]...W[n], must be initialised to zero at the start. The required result of the multiplication is :

W[1]...W[n+m] ← (U[1]...U[n]) * (V[1]...V[m])

Without the zeroing step the calculation would result in :

W[1]...W[n+m] ← (U[1]...U[n]) * (V[1]...V[m]) + W[1]...W[n] [6]

4.3.4 Test Cases

1. { 3,13,1,21 } * { 2,F015,F243 } = { 5,D187,EB1F,E50A,3AC2,1F }
   (141,733,986,323 * 4,064,538,645 = 576,083,264,719,734,952,335)

2. { 0,0 } * { 2,F015,F243 } = { 0,0 }

3. { 1,1 } * { 2,F015,F243 } = { 2,F015,F243 }

4. { 3,13,1,21 } * { 0,0 } = { 0,0 }

5 { 3,13,1,21 } * { 1,1 } = { 3,13,1,21 }

Multiply has no optional steps so one general test case and some additional test cases involving zero and unity multipliers are judged to sufficiently examine its general operation. A complete test of the function is best done using encryption sized examples. This is documented both for multiply and division in the add back test section of the division function.

4.3.5 Algorithm Steps in Example

Multiplication of { 3,13,1,21 } by { 2,F015,F243 }

Step	i	j	U[i]	V[j]	t	W[1]	W[2]	W[3]	W[4]	W[[5]
5	1	1	13	F015	11D18F	D18F	0	0	X	X
5	2	1	1	F015	F026	D18F	F026	0	X	X
5	3	1	21	F015	1EF2B5	D18F	F026	F2B5	X	X
6	3	1	21	F015	1EF2B5	D18F	F026	F2B5	1E	X
5	1	2	13	F243	12EB1F	D18F	EB1F	F2B5	1E	X
5	2	2	1	F243	1E50A	D18F	EB1F	E50A	1E	X
5	3	2	21	F243	1F3AC2	D18F	EB1F	E50A	3AC2	X
6	3	2	21	F243	1F3AC2	D18F	EB1F	E50A	3AC2	1F

Fig. 4.4 *Multiplication Example*

4.3.6 Timing

Knuth's MIX implementation :	28NM + 7M + 4N + 3
TMS320C25 code :	9NM + 2M + 17N + 38
TMS320C25 independent function :	9NM + 2M + 17N + 49

Knuth's implementation has M places in the multiplier and N places in the multiplicand. The TMS320C25 version has M places in the multiplier which is the first argument in the function call, and N places in the multiplicand. As would be expected, this is the first routine to show a significant improvement over its MIX implementation in terms of number of instructions to be executed, in addition to the observed improvement due to word length and instruction cycle time.

4.4.1 DIVISION FUNCTION

Function:	void div(u,v,q,r) unsigned int u[SIZE1],v[SIZE2], q[SIZE1], r[2*SIZE1+1]
Files:	div.asm, div.obj
Description:	Divides one multi-precision number by another.
Parameters:	Three unsigned integer arrays in big format. On exit $q = \lfloor u/v \rfloor$ and $r = u \bmod v$.
Return Value:	None
Restrictions:	u must be non-zero. q must be distinct from both u and v, r must be distinct from v i.e. div(u,v,q,u) allowed but div(u,v,u,r) not allowed.

4.4.2 Low Level Algorithm Description : DIV

1. Function initialisation.

2. Special cases, register set-up, store and norm determination.
   Copy : R := U.
   If U[0] < V[0] go to 14 => zero Q and finished.
   If V[0] = 1 go to 16 : single precision division case.
   Calculate loop counter for entire function : main loop must execute m - n times.
   Calculate norm : NORM = $\lfloor$ BASE / (V[n] + 1) $\rfloor$

3. Normalize V : V[1]..V[n] = NORM.(V[1]..V[n])
   The multiplication will not introduce an extra term in V because of the method used to calculate it.

4. Normalize R : R[1]..R[m](R[m+1] ?) = NORM.(R[1]..R[m])
   There may be an extra term introduced in the normalised R.
   R[0] is incremented anyway so that $\lfloor$ u/v $\rfloor$ < BASE even though this sometimes produces a first quotient digit equal to zero.
   Division : U div V = Q. U mod V = R. $\Rightarrow$ R is what is left of U at the end of the division after U has been successively reduced by the product of V and quotient terms. The method employed uses R set equal to U at the start and R, not U, changes during the function.

5. Register initialisation for loop.
   U[j] $\Rightarrow$ j = m.

6. Calculate quotient term, Qhat.
   If U[j] = V[n], Qhat $\leftarrow$ BASE - 1 (= FFFF hex),
   else Qhat $\leftarrow$ $\lfloor$ (U[j].BASE + U[j-1]) / V[n] $\rfloor$

7. Check if Qhat too big : Test if
   V[n-1] * Qhat >
   (U[j].BASE + U[j-1] - Qhat.V[n]).BASE + U[j-2]
   If it is, decrease Qhat by one and repeat the test.

8. Multiply and subtract.
   R[j-n]..R[j] = R[j-n]..R[j] - Qhat.V[1]..Qhat.V[n]
   Calculate Qhat.V[1]..Qhat.V[n]. Carry out entire subtraction function.
   If the subtraction ends in a borrow must add back V[1]..V[n] and increment Qhat. Else go to 10.

9. Add back.
   Note that if the subtraction ends in a borrow the last borrow is not propagated, so here when a carry occurs at the end of the sum, it must also be ignored to cancel with that borrow.

10. Test loop condition and loop if not finished.

11. Store Q correctly with calculated Q[0].
    Q is calculated during the division by working out Qhat terms from the most significant down. The first Qhat is written to the location where the most significant term of the largest possible Q for the division would be. Thus the completed quotient will always run to the end of the memory space allotted to the quotient but it may not start at the start of allotted quotient space, so in addition to calculating the number of terms in Q and storing this at Q[0], the computed quotient is copied to start from the start of its allotted space in memory.

12. Strip leading zeros from R.

13. Unnormalisation.
    The computed quotient is correct but the computed remainder must be unnormalised : R := R div NORM.
    V must also be unnormalise : V := V div NORM.
    Go to 15.

14. Zero Q.

15. Function termination.

16. Single precision division case.
    Go to 15.

4.4.3 Explanation

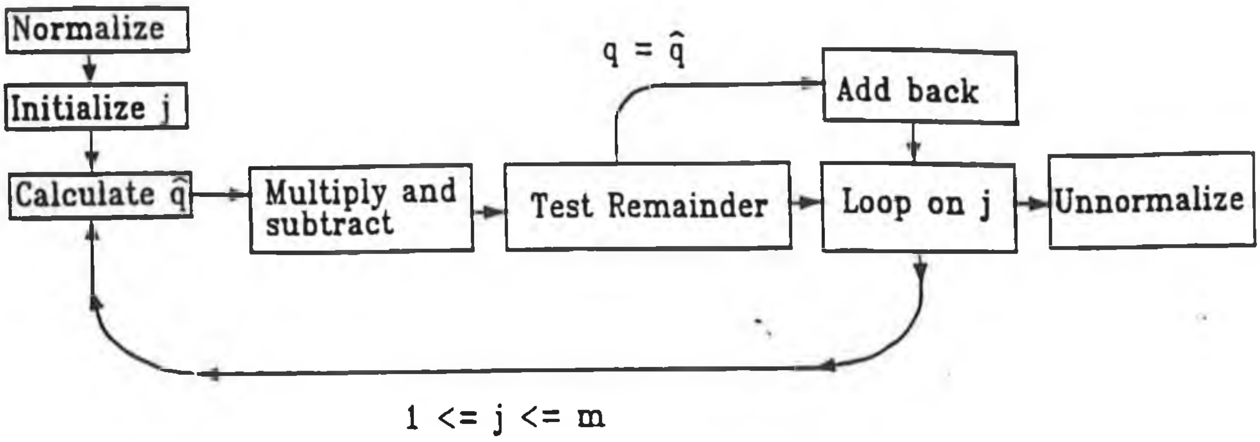


Fig. 4.5 *Block diagram showing the main cases of the division algorithm [4]*

The division algorithm used is based on the standard pencil-and-paper method in much the same way as the multiplication algorithm. The significant difference between division and multiplication is that the multiplication process already involves a clearly laid-out procedure based on knowledge of the single-precision multiplication tables, while division involves a degree of guesswork to produce a trial partial quotient which can then be checked for correctness by doing a multiplication at each step. The procedure that is followed once the trial quotient has been determined is well defined and easy to specify in algorithm steps. The method used to determine the trial quotient must be resolved into an algorithmic description. Since this value may not necessarily be the correct result, it is necessary to investigate any theory which describes test conditions for its correctness, which can show how many adjustments to the first trial quotient produced may be needed and also which outlines whether this first test value could be either too big or too small.

The general problem is one of finding the result of dividing an m-place nonnegative integer u = u[0]...u[m] by an n-place nonnegative integer v = v[0]...v[n]. The remainder can be easily calculated from u and the division result and is in fact produced as a by-product of the quotient calculation in Knuth's algorithm. The normal pencil-and-paper approach to this problem would involve writing it in the following manner :

```
                              q[j]...q[1]
                         ____________________________
v[n]...v[1]          )      u[m]......u[n]...u[1]
                            q[j]  *  ( v[n]...v[1] )
                         ____________________
                            ( Remainder 1 ) u[n+1]
                                  .          .
                                  .          .
                                  .          .
```

Fig. 4.6 *Pencil-and-paper division method*

This method can be broken down into steps in which v[1]...v[n] is divided into u[i]...u[i+n+1], starting with i+n+1 = m, with condition : u/v < BASE to ensure a single precision result. The same operation can be tried first on u[i]...u[i+n] (i+n initially = m) to ensure that the condition is true. This step will then involve the division of an n+1 precision number by an n precision number giving a single precision result. The method used in the pencil-and-paper method is to guess the trial quotient based on the most significant digits in the division, noting that the result cannot exceed the maximum single precision result value representable in the base being used, namely the value of the base minus one.

This suggests the following formula for Qhat :

$$\text{Qhat} = \min\left(\lfloor (u[j].b + u[j-1]) / v[n] \rfloor,\ \text{BASE} - 1 \right)$$

Using this formula for Qhat, Knuth shows that it is a good approximation for the true quotient value by proving:

1. $\text{Qhat} \geqslant \text{Q}$
2. If $v[1] \geqslant \lfloor \text{BASE}/2 \rfloor$, then $(\text{Qhat} - 2) \leqslant \text{Q} \leqslant \text{Qhat}$

The condition necessary for the second theorem to hold is a normalization requirement which can easily be fulfilled by multiplying both dividend and divisor by an appropriate scaling factor : $\lfloor \text{BASE} / (v[n]+1) \rfloor$. Once this requirement has been met, the method used to determine the trial quotient can be ensured to give the true quotient or the true quotient plus one or the true quotient plus two.

A test can be carried out on the trial quotient which eliminates all cases of it being two too large and most cases of it being one too large. This involves determining whether

v[n-1] * Qhat >
    (u[j] * BASE + u[j-1] - Qhat * v[n]) * BASE + u[j-2]

and if it is, decreasing Qhat by one and repeating this test. The theory is therefore availible to ensure that steps D6 and D7 in the algorithm description result in the true quotient value in most cases, while all other cases can be handled by the using the add back step D9.

4.4.4 Test Cases

There are a large number of permutations and combinations of sections of the division function which can come into play in any given division problem. These include :

1. U[0] < V[0]. This is the case $q = \lfloor u/v \rfloor = 0$ and $r = u \bmod v = u$.

2. V[0] = 1. This is the single precision division case which is handled completely separately to general multi-precision division. This is because the general multi-precision approach makes use of v[n] and v[n-1] and v[n-1] does not exist in the single precision case.

3. Qhat may be too big after the initial Qhat calculation.

4. Qhat may still be too big after it has been decreased by one, giving a borrow after the multiply and subtract section, and thus requiring use of the add back section.

5. Along with the different possible division cases, the optional parts of both the subtraction and addition sections will be accessed in much the same way as they are accessed in the subtraction and addition functions from which they are taken.

In addition to a relatively simple small division example for general test, it has been decided to test the cases where U[0] < V[0] and also where V is single precision. An initially too large Qhat is quite common and will be tested by any large division operation. The add back case occurs once in approximately every 32768 quotient calculations (in the order of $\lfloor 2/BASE \rfloor = \lfloor 2/65536 \rfloor$ times [6]). Knuth suggests that test data should therefore be contrived for testing this section. It was decided that it would be better to make use of the breakpoint facilities offered by the Software Development System to detect where the add back section is entered and then check the result against a value calculated using the MIRACL system [5]. This use of a real case would ensure that test data which requires add back would also be valid data that was generated by the multiply and subtract section. This test method for add back can be assumed to provide an adequate test for the last three of the above mentioned possibilities.

1. *Small general case division example.*

u = { 5,D190,EB1F,3AC2,1F }
v = { 3,13,1,21 }
q = ⌊ u/v ⌋ = { 2,F015,F243 }
r = u mod v = { 1,1 }

2. *V single precision test case.*

u = { 5,D190,EB1F,3AC2,1F }
v = { 1,2 }
q = ⌊ u/v ⌋ = { 5,E8C8,758F,7285,9D61,F }
r = u mod v = { 0,0 }

3. *U[0] < V[0] test case.*

u = { 2,D190,EB1F }
v = { 3,13,1,21 }
q = ⌊ u/v ⌋ = { 0,0 }
r = u mod v = { 2,D190,EB1F }

4. *Add back / large division general test case.*

Enciph.c is a program which is part of the MIRACL Package. The program enciphers a file using the Blum, Blum and Shub algorithm. It takes in a nine digit number from the user and uses this as a seed value for a pseudo-random number generator to produce a pseudo-random initialisation. The enciphering is then done by exclusive-oring the result of the :

$$x_{i+1} = x_i^2 \mod ke$$

operation with the message, character by character. X is initally the pseudo-random value while ke is a set value chosen to fulfil the requirements of the BBS algorithm and which also serves to keep the calculation within a given precision length. The program uses the MIRACL function mad( ) thus : mad(x,x,x,ke,y,x), with inputs x and ke which gives results : y = ⌊ x.x / ke ⌋ and x = x.x mod ke. This program was run and the initial x value and the value of ke produced by the C program after a seed value equal to 123456789 had been specified, were used in a program to test the division function. This test program contained the x and ke initialisations followed by the loop :

```
for (;;)
{
mult(x,x,y);
div(y,ke,q,x);
}
```

The test program was cross-compiled and a breakpoint was inserted in the resulting assembly language program within the division function at the start of the add back section. As expected it took a large number of iterations before this breakpoint was reached : a counter variable indicated that the add back section was only encountered for the first time after 7444 iterations of this loop using the data generated by enciph.c. Enciph.c was then run with x values printed out after 7444 mad(x,x,x,ke,y,x) operations. It was found that the resultant x corresponded with the value produced by the test program :

After 7444 iterations :
x[0] = 20, x[1] = 7772, x[2] = C6EF, ..., x[20] = 2159.

This method was used for two more similar tests to show that the add back section works in more than just one test case. It was found that add back was required after 7888 iterations and then again after 18683 iterations of the mult(x,x,y) and div(y,ke,q,x) functions using the same initial data. Again the results corresponded with the values produced by the MIRACL enciph.c program :

After 7888 iterations :
x[0] = 20, x[1] = 95C4, x[2] = A5DA, ..., x[20] = 1E73.

After 18683 iterations :
x[0] = 20, x[1] = 7508, x[2] = 7EBF, ..., x[20] = A4.

4.4.5 Timing

The division function is the largest and the most complicated of the multi-precision functions which have been developed. This can be observed in the timing calculation in which several approximations and assumptions must be made. A timing of individual steps is shown below, with notes of assumptions made.

Function initialisation :	5
Register set-up :	10
Register store :	5
Copy U to R :	3 + 3 * (M + 1)
Check if U[0] < V[0] and if V[0] = 1 :	7
Calculate loop counter :	3
Norm calculation :	27
Scale V :	4 + 7N
Scale U :	11 + 7M
Register initialisation for loop :	6

The following loop is executed M times :

{

Calculate Qhat (i) :	25
Check if Qhat too big (ii) :	22 + .5 * (6 + 22) + 3
Multiply and subtract (iii) :	16 + 7N + 41 + 7 $\lfloor M/2 \rfloor$
Add back (iv) :	Negligible
End sub :	6

}

Q adjust :	12 + 3N
Strip leading zeros form Q (v) :	9 + M
Strip leading zeros form R (v) :	9 + N
Unnormalize R :	16.5 + 23M
Unnormalize V :	14 + 23N
Zero Q (vi) :	Negligible
Function Termination :	7
Single precision division (vi) :	Negligible

i. R[j] is assumed not to be equal to V[n] in working out this timing. This assumption should be correct 65535 times out of 65536, on average.

ii. It is assumed here that the trial quotient, Qhat, must be reduced by one due to the results of the test half the time. This has been observed in practice to be more often than this possibility occurs.

iii. Since the aim in producing this timing is mainly to look at the large multi-precision operations that the function will mainly be used for, it is assumed here that there is more than one double precision subtraction.

iv. Add back occurs of the order of once in every 32768 cases so it can be neglected as having little effect on the timing.

v. Q and R are assumed to have the most significant half of their precisions equal to zero.

vi. Timings can be worked out for the cases of $U[0] < V[0]$ resulting in $Q = 0$, and V single precision, but it is reasonable to assume that these types of operations are not likely to occur often enough to impact on timing.

The total time summed from the time for all the sections is :

$7NM + 7M\lfloor M/2 \rfloor + 151M + 34N + 151.5$ instruction cycles.

where M is the precision of U and N is the precision of V. In these calculations Q is assumed to be the same precision as U, and R the same precision as V. This assumption may result in the calculated number of instructions being bigger than the number that will be executed in an average case, but it is appropriate to err on the side of calculating too large an estimate of the number of instruction cycles rather than too small.

Knuth's MIX implementation takes $30NM + 97N + 326M + 115$ instruction cycles where there are M+1 words in the quotient and N words in the divisor. If this is expressed in the same form as the TMS320C25 implementation i.e. with M words in U and N words in the divisor, V, the result is the following timings :

Knuth's MIX implementation :	$30NM + 67N + 326M - 211$
TMS320C25 code :	$7NM + 7M\lfloor M/2 \rfloor + 34N + 151M + 139.5$
TMS320C25 independent function :	$7NM + 7M\lfloor M/2 \rfloor + 34N + 151M + 151.5$

4.5.1 SQUARING FUNCTION

Function:	void square(x,y) unsigned int x[SIZE], y[2*SIZE - 1];
Files:	square.asm, square.obj
Description:	Squares a multi-precision number.
Parameters:	One unsigned integer array in big format. On exit y = x * x.
Return Value:	None
Restrictions:	y must be distinct from x.

4.5.2 Low Level Algorithm Description : SQUARE

1. Function initialisation.

2. Register and counter set-up.
   Initialise $i \leftarrow 1$, $j \leftarrow i + 1$.
   $Z[0] \leftarrow 2N$.

3. Start of i loop.
   Zero carry.

4. Start of j loop.
   If $j > N$ branch to 6, end j loop.
   Acc $\leftarrow x[i] * x[j]$ + carry.
   Carry $\leftarrow$ AccH.
   $Z[i+j-1] \leftarrow Z[i+j-1]$ + AccL
   If last calculation resulted in a carry above one precision, carry ++.

5. Branch to start j loop.

6. End j loop.
   $Z[n+i] \leftarrow$ carry. If $i < n$, go to 3.

7. Double Z.
   $Z[2N] \leftarrow$ carry. If carry = 0, Z[0]--.

8. Square loop.
   i counter = 1. Zero carry.
   Acc $\leftarrow x[i] * x[i]$ + carry.
   Carry $\leftarrow$ AccH. $Z[2i-1] \leftarrow Z[2i-1]$ + AccL
   If this sum causes a single precision carry, carry++.
   $Z[2i] \leftarrow$ ($Z[2i]$ + carry) % BASE.
   If $Z[2i]$ + carry produces a single precision carry, carry = 1, else carry = 0.

9. End square loop.
   If $i <= n$, go to 8.

10. Function termination.

4.5.3 Explanation

Multiplication involves computing all the cross products of the multiprecision digits of the numbers being multiplied. When a number is being squared there is a considerable amount of repetition of cross product calculation. On a general purpose microprocessor which has a high overhead for each multiplication operation, a large time saving can be obtained by ensuring that those cross product results that are used more than once, are not calculated more than once. It is worth considering the benefit of implementing an assembly language squaring routine on a digital signal processor because public key encryption involves exponentiation which can be broken down into a number of squaring, multiplication and division operations. It is not immediately clear that there will be a timing benefit from a separate implementation to the standard multiplication function for squaring on a digital signal processor. Unlike the case with a general purpose microprocessor, the multiplication operation is comparatively cheap on the DSP, so the overhead involved in determining which cross product results can be used twice may be more than the saving from doing these multiplication operations only once.

The following examples illustrate where the savings can be obtained in a squaring operation:

$$
\begin{array}{llll}
 & a & b & \\
 & a & b & \\
\cline{2-3}
 & & & * \\
 & \underline{a.b} & b.b & \\
a.a & \underline{a.b} & & \\
\cline{1-3}
 & & & +
\end{array}
$$

$$
\begin{array}{llllll}
 & & a & b & c & \\
 & & a & b & c & \\
\cline{3-5}
 & & & & & * \\
 & & \underline{a.c} & \underline{b.c} & c.c & \\
 & \underline{a.b} & b.b & \underline{b.c} & & \\
a.a & \underline{a.b} & \underline{a.c} & & & \\
\cline{1-5}
 & & & & & +
\end{array}
$$

$$
\begin{array}{ccccccc}
 & & & a & b & c & d \\
 & & & a & b & c & d \\
\cline{4-7}
 & & & & & & * \\
 & & & \underline{a.d} & \underline{b.d} & \underline{c.d} & d.d \\
 & & \underline{a.c} & \underline{b.c} & c.c & \underline{c.d} & \\
 & \underline{a.b} & b.b & \underline{b.c} & \underline{b.d} & & \\
a.a & \underline{a.b} & \underline{a.c} & \underline{a.d} & & & \\
\hline
 & & & & & & +
\end{array}
$$

Fig. 4.7 *Illustration of repeated multiplications in squaring*

Each letter represents a single precision of the number, so the product of two letters represents a double precision number. The formula for the square of a multiprecision number with n places is :

$$(X[n])^2 = 2 * \sum_{i=1}^{i=n} \sum_{j=i+1}^{j=n} (X[i] * X[j]) + \sum_{i=1}^{i=n} (X[i])^2$$

The second term involving single precision square terms requires a loop which executes n times. The easiest way to implement the first term is to calculate and sum all the cross product terms and then multiply the result by 2. It would be nice to be able to efficiently calculate the first term completely in just one loop but, because a cross product term multiplied by 2 can give a triple precision result, this result cannot be simply expressed as just a product and a carry, and it is therefore easier to break the calculation into two loops.

An example of how the main loop for this term might look if implemented in high level is shown below :

```
for ( i=1; i<n; i++)
     {
     carry=0;
     for  ( j=i+1; j<=n; j++)
          {
          product=x[i]*x[j]+carry;
          carry=product/BASE;       /* implicit div operation */
          z[i+j-1] = ( z[i+j-1] + product%BASE )%BASE ;
          if ( z[i+j-1]< (product%BASE) ) carry++;
          }
     z[n+i]=carry;
     }
```

The important feature of this piece of code to be noted in the specification stage of the design of an assembly language routine for squaring is the loop structure. The instructions in the inner loop are carried out N(N-1)/2 times. The highest order term in the doubling and in the calculation of the second term is N so the dominant term in the timing of the squaring of a large multiprecision number is the N*N term from the above loop. This means that the efficient implementation of the above loop is critical for the function to run quickly. Specifically it means that because the squaring operation using the general multiplication function takes : 9N*N + 19N + 35 instruction cycles, the loop must be implemented in less than 18 instructions to produce a timing benefit, because the N*N term is scaled by a factor of 1/2.

4.5.4 Test Cases

It was decided that the squaring function could be effectively tested by looking at the zero case, the unity case, two single precision cases, a small multi-precision example and finally a large application type example. The following test cases worked as expected :

1. x = { 0,0 }, y = { 0,0 }.

2. x = { 1,1 }, y = { 1,1 }.

3. x = { 1,3 }, y = { 1,9 }.

4. x = { 1,FFFF }, y = { 2,1,FFFE }.

5. x = { 3,FFFF,FFFF,FFFF }, y = { 6,1,0,0,FFFE,FFFF,FFFF }.

   This test case was easy to check theoretically using the formula :

   $(BASE^3 - 1) * (BASE^3 - 1) = BASE^6 - 2 * BASE^3 + 1$

6. The large application example was the same as the one used to test the add back section in the division function. Mult(x,x,y) was replaced by square(x,y) and the result after 7444 iterations of the loop:

```
for(;;)
{
    square(x,y);
    div(y,ke,q,x);
}
```

   corresponded with the result produced by the enciph.c program from the MIRACL package.

4.5.5 Timing

Number of instructions executed once : 61.5

Zeroing Z : 2N

Instructions in i loop only : 11 => 11 N

Instuctions in j loop (and i loop) : 16.5 => 16.5 * N(N-1)/2

Doubling Z : 5N

Square loop : 20N

Total : 8.25 N*N + 29.75 N + 61.5 instruction cycles.

The equivalent multiplication function takes : 9 N*N + 19 N + 49 cycles. Solving for N, it can be shown that the squaring function gives a timing benefit over multiplication for operands with 16 places of precision or more.

4.6 House-keeping Functions

The basic multi-precision arithmetic routines are addition, subtraction, multiplication and division. Squaring is a special case of multiplication which can show timing benefits over the general case multiplication function when implemented separately. These routines provide the tools necessary to allow the implementation of any multi-precision arithmetic application. They are also the primary number-crunching functions in any such application and, as a result, the fast operation of any application which uses them depends almost exclusively on their efficient implementation.

In addition to having routines which make the application program run fast, it is desirable to ensure that programs which use the multi-precision arithmetic assembly library are as easy to write as possible. The compatibility of the assembly functions with cross-compiled C code means that most of the coding facilities which a programmer can require are provided by the C programming language. However there are functions which allow simple manipulation and inspection of multi-precision numbers and which are practically essential for the house-keeping type operations which are likely to arise in a multi-precision application.

These house-keeping functions are the zero, lzero, copy and compare functions. Zero sets a multi-precision number to zero. Lzero strips leading zeros from a multi-precision number. If a previous operation has resulted in a multi-precision number having one or more of its most significant digits equal to zero, lzero will ensure that the digit counter in the first array position is decremented until the most significant digit is no longer zero. Copy copies one multi-precision number to another. Compare compares two multi-precision numbers and returns a different value depending on whether the first number is less than, equal to or greater than the second number. Other useful functions could be written either in assembler or C to facilitate other simple operations, but it is felt that this small set of functions provides enough facilities for most multi-precision applications.

4.6.1 ZERO FUNCTION

Function:	void zero(x) unsigned int x[SIZE];
Files:	zero.asm, zero.obj
Description:	Sets a multi-precision number to zero.
Parameters:	One unsigned integer array in big format. On exit x[0]=0, x[1]=0.
Return Value:	None
Restrictions:	None
Example:	/* Before : x[0]=2; x[1]=3; x[2]=8; */ zero(x); /* After : x[0]=0; x[1]=0; */

4.6.2 LZERO FUNCTION

Function: void lzero(x)
unsigned int x[SIZE];

Files: lzero.asm, lzero.obj

Description: Strips the leading zeroes from a multi-precision number. If a big format number contains a most significant digit or digits which is/are equal to zero, the first digit is reduced until the most significant digit in the big format number is non-zero.

Parameters: One unsigned integer array in big format. On exit the most significant digit of the big number is non-zero.

Return Value: None

Restrictions: None

Example:

```
/* Before : x[0]=3; x[1]=5; */
/* x[2]=x[3]=0;             */

lzero(x);

/* After  : x[0]=1; x[1]=5; */
```

4.6.3 COMPARE FUNCTION

Function: int compare(x,y)
unsigned int x[SIZE1],y[SIZE2];

Files: compare.asm, compare.obj

Description: Compares two multi-precision numbers.

Parameters: Two unsigned integer arrays in big format.

Return Value: Returns 2 if x > y, returns 1 if x < y and returns 0 if x = y. The value is returned in the accumulator. This is the convention for the TMS320C25 compiler so control of program flow statements can be used as in the example below.

Restrictions: None

Example:

```
main()
{
int i,x[3],y[2],z[4];
x[0]=2; x[1]=8; x[2]=3;
y[0]=1; y[1]=9;

i = compare(x,y); /* i gets set = 2 */

if ( i==0 ) zero(z);
else if ( i==1 ) add(x,y,z);
else sub(x,y,z);
}
```

4.6.4 COPY FUNCTION

Function: void copy(x,y)
unsigned int x[SIZE], y[SIZE];

Files: copy.asm, copy.obj

Description: Copies a multi-precision number to another.

Parameters: Two unsigned integer arrays in big format.
On exit Y = X.

Return Value: None

Restrictions: None

Example:

```
/* Before : x[0]=2; x[1]=3; x[2]=8; */
/*          y[0]=1; y[1]=9;          */

copy(x,y);

/* After  : x[0]=2; x[1]=3; x[2]=8; */
/*          y[0]=2; y[1]=3; y[2]=8; */
```

5 APPLICATIONS

5.1 Exponentiation

Exponentiation is the central operation in most public-key encryption schemes. The RSA algorithm uses the formula :

$$C = M^K \mod N$$

for both the encryption and decryption operations. The message is raised to the power of the public-key K, modulus N, the product of two suitable primes, to produce the ciphertext. At the receiving end the message is calculated to be equal to the ciphertext raised to the power of a secret key also modulus N. The operands are different at the transmitter and receiver but the operation is the same. The most obvious method to compute an exponent is to simply multiply repeatedly. This however is not necessarily the cheapest method in terms of the number of instructions.

Looking at a simple example, X^{32} , the repeated multiplication method would require 31 multiplies. Only 5 would be required if the result was calculated by taking repeated squares :

$$X^2 , X^4 , X^8 , X^{16} , X^{32}.$$

This example is a special case in that it involves raising the number to a power of 2 but the important point to note is that it illustrates that there may be a way to calculate an exponentiation which requires significantly fewer multiplication operations than might be thought if multiplication by the original X value was the only method availible.

Two exponentiation methods which can exhibit such savings as shown above are the binary methods, either operating on the exponent from left to right or from right to left. These methods both involve looking at the exponent in binary format and deciding whether to square a partial result or multiply it by X, depending on whether the bit under examination in the exponent is a zero or a one. While the left-to-right method is fully practical, it is generally more convenient to look at a number from right to left because, in an implementation which makes use of the hardware, the number can be shifted right or divided by two until it equals zero and the parity bit can be observed in a hardware register.

Knuth outlines an algorithm for the right-to-left binary method for exponentiation i.e. calculate X^n , where n is a positive number [6]:

1.	**Initialize :**	$N \leftarrow n$, $Y \leftarrow 1$, $Z \leftarrow x$.
2.	**Halve N :**	Shift N right: $N \leftarrow \lfloor N/2 \rfloor$ Check value of shifted bit. If it is 0, go to 5.
3.	**Multiply Y by Z :**	$Y \leftarrow Z * Y$
4.	**Check : N = 0 ? :**	If $N = 0$, algorithm finished. Answer is Y.
5.	**Square Z :**	$Z \leftarrow Z * Z$. Go to 2.

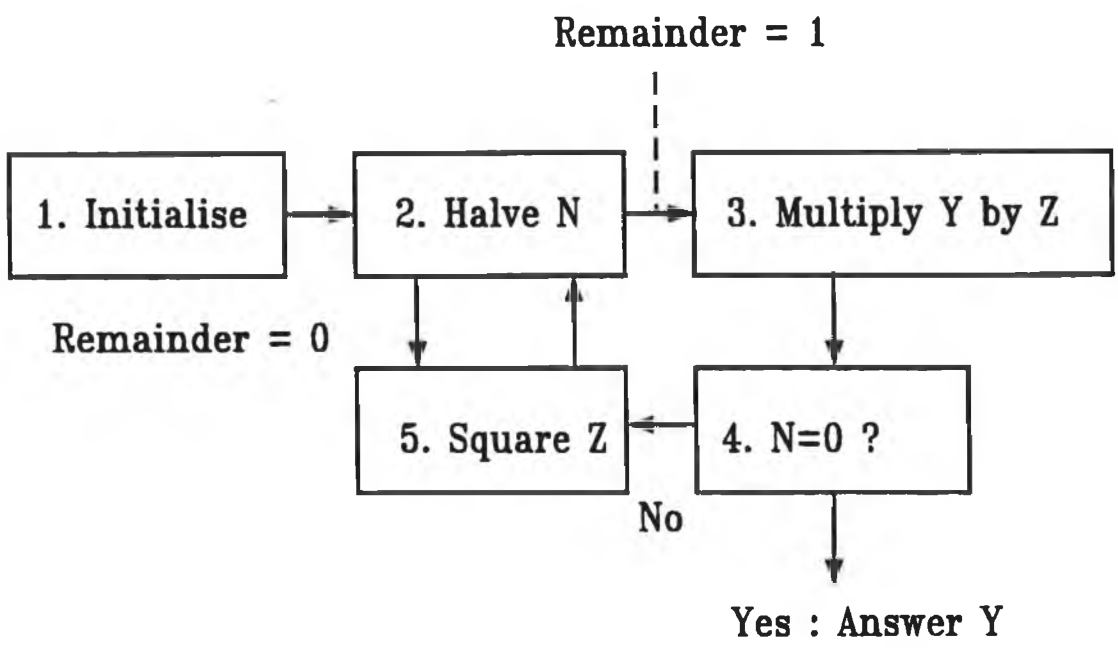


Fig. 5.1 *The left-to-right scan exponentiation method for the calculation of X^n* [6].

An example showing values after steps 1 and 4 in the calculation of X^{25} follows :

Step	**N**	**Y**	**Z**
1	25	1	X
4	12	X	X
4	6	X	X^2
4	3	X	X^4
4	1	X^9	X^8
4	0	X^{25}	X^{16}

Fig. 5.2 *Exponentiation calculation example.*

The result is the Y value when N has been reduced to zero. It can be seen from the example that the number of multiplications of Z times Y is given by the number of ones in the exponent, v(n), while the number of squarings is determined by the length of the binary representation of the exponent : $\lfloor \lg_2 n \rfloor$.

Therefore the total time taken for an exponentiation is : $v(n) + \lfloor \lg_2 n \rfloor$.

A reasonable approximation is that half of the bits in n are equal to one and the other half are equal to zero giving an average of $3/2 * \lfloor \lg_2 n \rfloor$ multiplications. The maximum number of multiplies in an exponentiation for a given length exponent occurs when the binary representation of the exponent is all ones. This will result in $2 * \lfloor \lg_2 n \rfloor$ multiplications. If the exponentiation is being carried out on a number which has 16 places of precision or more, then use of the squaring function offers timing benefits over the multiplication function and the timing for the exponentiation operation will be based on both the number of instructions executed in the multiplication function and the number executed in the squaring function. There is also an overhead due to the use of the copy function.

An example of a high level code implementation of the above algorithm using the functions from the DSP assembly language multi-precision arithmetic library follows :

(The output function is not part of the library and would have to be developed for specific applications.)

```
expo(z,n,y)
unsigned int z[], y[]; int n;
{
int parity;
unsigned int temp[SIZE]
for (;;)
      {
      parity = n%2;
      n /= 2;
      if ( parity )
            {
            mult(y,z,temp);
            copy(temp,y);
            if ( n == 0 )
                  {
                  output("answer = %u", y);
                  break;
                  }
            }
      square(z,temp);
      copy(temp,z);
      }
}
```

If the overhead due to the statements which control program flow is neglected, as an approximation, then the timing for this function can be calculated using the following formula :

$\lfloor lg_2\ n \rfloor$ * (squaring time + copy time) +
    $v(n)$ * (multiplication time + copy time)

multiplication time : 9NM + 2M + 17N + 49
squaring time : 8.25N*N + 29.75 N + 61.5
copy time : 3M + 17
$v(n) \approx 0.5 * \lfloor lg_2\ n \rfloor$.

It is not possible to multiply out and to sum this timing for a general case because the sizes of both z and y increase after each iteration, so a timing would have to be worked out for each individual application. However in almost all applications exponentiation is done modulus a given number which keeps the calculation results smaller than that particular value. If the operation is done "mod value" the multi-precision number value is passed to the expo(z,n,y) function in a similar manner to z and y, and the copy functions is replaced by a division operation :
copy(x,y) → div(x,value,temp,y), copy(x,z) → div(x,value,temp,z).

This would yield a timing sum of :

$\lfloor lg_2\ n \rfloor$ * (squaring time + division time) +
    $v(n)$ * (multiplication time + division time)

multiplication time : 9NM + 2M + 17N + 49,   N = M.
squaring time : 8.25N*N + 29.75N + 61.5,
division time : 7NM + 7M$\lfloor M/2 \rfloor$ + 34N + 151M + 151.5,   M = 2 * N

N ≈ Y[0] ≡ Z[0]. 2 * Z[0] ≡ X[0].
$\lfloor lg_2\ n \rfloor \approx$ 16N
$v(n) \approx 0.5 * \lfloor lg_2\ n \rfloor \approx$ 8N
16N * (9N*N + 19N + 49 + 14N*N + 34N + 302N + 151.5) +
 8N * (8.25N*N + 29.75N + 61.5 + 14N*N + 34N + 302N + 151.5) =
882 N*N*N + 8606 N*N + 4912 N instruction cycles.

There is a cubic term in N in the timing result because of the second order term in N in both the multiplication and division functions, and the exponentiation involves a loop containing these functions which iterates of the order of N times. This means that exponentiation is a very expensive operation in timing terms.

5.2 The Rivest, Shamir and Adleman Algorithm [7]

The RSA algorithm requires one exponentiation modulus the product of two primes for encryption and a similar operation for decryption at either end of a transmission channel. The C code described in the section on exponentiation and the resulting timing therefore show both the implementation and the timing of this public-key algorithm. The timing is done in the exponentiation section without working out complete results because these are dependent on the size of the numbers used in each application. It is useful to determine a rough value for the time taken by this algorithm, based on approximate sizes for the numbers used. In practical applications, these numbers are chosen as small as possible in order to enhance speed but they must be large enough to provide whatever is considered to be adequate security.

The following results are taken from the output of a program written to calculate the time taken by R.S.A.. The program uses functions from the MIRACL package to calculate the number of instructions in each routine that is called by the RSA algorithm. The instructions which control the flow of the encryption algorithm including the loop conditions, are neglected in this calculation because their number has a very minor impact on the result in comparison to that of the numerical functions.

For calculation purposes and to allow for slightly worse than the average case, some of the formulae constants are rounded up in the expressions for the number of instructions in the arithmetic functions :

multiplication time : $9NM + 2M + 17N + 49$
squaring time : $8.25N*N + 29.75N + 62$
division time : $7NM + 7M\lfloor M/2 \rfloor + 34N + 151M + 152$

Number of digits in multiplication, squaring and division denominator : 33
There are twice the number of digits in the division numerator.

Number of instructions in multiplication : 10477
Number of instructions in squaring : 10029
Number of instructions in division : 41732

Number of instructions in multiplication and division : 52209

Number of instructions in squaring and division : 51761

Total number of instructions in encryption : 984355

Encryption is based on an exponent equal to $2^{16} + 1$ (65537).

Total number of instructions in decryption : 41112984

Decryption is based on a 33 word exponent.

Therefore a block of 33 words or 528 bits would require 0.0984 seconds to encrypt and 4.111 seconds to decrypt, based on the 100 nanosecond instruction cycle of the TMS320C25. This is an encryption rate of 5.36 kbits/second with decryption running at 128 bits/second. The same method as that used in calculating these timing estimates was used for some other block lengths :

Modulus Length (words)	(digits)	Enc. time (ms)	Dec. time (ms)
10	49	15	180
20	97	41	1,060
30	147	84	3,170
32	155	94	3,780
33	159	99	4,120
35	169	109	4,840
37	179	121	5,650
40	193	139	7,020
50	241	207	13,150
60	289	290	22,080

Fig. 5.3 *RSA block encryption time*

Modulus Length (words)	(digits)	Enc. rate (kbits/second)	Dec. rate (bits/seconds)
10	49	11.24	891
20	97	7.64	302
30	147	5.76	151
40	193	4.62	91
50	241	3.85	60
60	289	3.3	43

Fig. 5.4 *RSA byte encryption time*

A block length of 160 digits (33 words) could be considered to be a minimum requirement for security [19]. The timing results outlined above indicate that full public key speech encryption is not feasible at this block length. However the timings are within a suitable range for key exchange as would be required in a hybrid public-key/secret-key system. Therefore the use of the multiprecision arithmetic assembly language functions to implement the R.S.A. algorithm in its standard format has produced a viable program for real time data communication but the more stringent specifications of speech communication have not been met.

There is an alteration which can be made to the decryption section of the R.S.A. algorithm to produce better timing values. Time savings result from pre-computing auxiliary numbers which make the exponents used in the general decryption operation smaller [19,26].

In this method the following numbers are computed prior to the decryption calculation:

$$A_p = q^{p-1} \bmod n,\ A_q = n + 1 - A_p,\ D_p = d \bmod (p-1),\ D_q = d \bmod (q-1)$$

Decryption then involves computing :

$$M = (\ A_p * (\ (C \bmod p)^{D_p} \bmod p) + A_q * (\ (C \bmod q)^{D_q} \bmod q)\ ) \bmod n$$

The most time-intensive parts of the above formula are the exponentiations to the power of D_p and D_q. The two primes, p and q, which, when multiplied together, produce n, are approximately half the length of n. D_p and D_q are limited to being smaller than p-1 and q-1 respectively. This means that instead of performing one exponentiation to the power of a number that is approximately the same length as n, two exponentiations are performed to the power of numbers that are approximately half the length of n. Due to the non-linear increase in the time needed to perform an exponentiation to the power of a larger number, there are substantial timing benefits to be obtained from using this method. According to the availible literature [19,26], the resultant timing shows a reduction of approximately 70 percent. The decryption time for a 33 word block size could therefore be reduced from 4.111 seconds to 1.233 seconds. This is an increase in decryption rate to about 384 bits/second which is a significant improvement but is still not adequete for real-time speech communication.

5.3 The Blum, Blum and Shub Algorithm [8]

The BBS algorithm is a pseudo-random number generator with public-key encryption applications. It is based on the iterative sequence :

$$x_{i+1} = x_i^2 \mod ke$$

This can be implemented using the following loop :

```
for(;;)
{
square(x,y);
div(y,ke,q,x);
output(x);
}
```

The output function must be written for each separate application. If a block size of 10 words (160 bits) is used each iteration of the above loop takes approximately 7497 instructions or 0.7497 milliseconds. If the parity bit only is considered to be cryptographically secure, this would give a pseudo-random bit stream producing 1.33 kbits/second. However, as recent number theory suggests that up to $\log_2 l$ bits produced by each iteration (where l is the number of bits in the modulus number n) may be considered to be cryptographically secure [8], this generator can be used to produce up to 9.34 kbits/second which is suitable for eXclusive-ORing with the message stream. This algorithm can therefore be successfully implemented for a realtime speech application using the multi-precision arithmetic assembly language routines from the library.

Modulus Length (words)	(digits)	$\lfloor \text{Log } l \rfloor$	Cryptographically secure bits (kbits/second)
10	49	7	9.34
20	97	8	3.63
30	147	8	1.83
40	193	9	1.47
50	241	9	0.82

Fig. 5.5 *BBS pseudo-random number generator bit rates.*

6 CONCLUSION

In this project the motivation for the use of encryption schemes to enhance security has been discussed and several applications have been outlined. An assembly language multi-precision arithmetic library has been developed for use in encryption applications on the TMS320C25 digital signal processor. The main arithmetic functions have been written in assembly language with significant timing improvements over the classical algorithms by Knuth upon which they are based. Additionally several other useful functions have been coded to facilitate the writing of high-level code for multi-precision applications.

As outlined in the applications section of this report, the library allows easy high-level coding of encryption algorithms and examples have been given. The timing values calculated for these applications show that the library can be used successfully in real-time situations. Due to the high bit-rate required for speech channels it is necessary to use a hybrid encryption scheme involving public-key initialisation of a secret key for a fast secret-key system if use of the Rivest, Shamir and Adleman algorithm is required. The Blum, Blum and Shub pseudo-random number generator has been implemented for real-time speech rates, assuming a number theory fraction of cryptographically secure bits [8], as outlined in Section 5.3.

A complete evaluation of the library could therefore be summarised in the following two points :

1. A reasonably user friendly package has been developed which allows encryption applications to be written in a few lines of C code, resulting in an efficient assembly language coding of the application in question.

2. Real-time constraints for such applications as speech encryption are difficult to meet and the library allows these constraints to be fulfilled only if a suitable algorithm is chosen.

There are three basic approaches which can be taken to tackle the problem of real-time speech encryption more successfully. The first method is to implement the chosen algorithm in hardware or at least to maximise the hardware support for the application. Hardware implementations are, in general, likely to be more expensive, faster and less flexible. Higher security levels can be achieved as a direct result of greater speed and for a specific commercial application flexibility is not the major concern. The trade-off is therefore the old one of cost against security.

An example of a hardware solution to the encryption problem is Rivest's "A Description of a Single-Chip Implementation of the RSA Cipher" [27]. He claims that his device can perform the encrypt/decrypt operation at rates in excess of 1200 bits/second with a maximum length (512-bit) key. This is close to speech rates but not quite good enough for a speech encryption system comprised solely of the RSA algorithm. Another hardware implementation of the RSA algorithm has been proposed, using systolic arrays [28]. The timing claim for this approach is that a solution involving hardware complexity O(n) gives speed proportional to $1/n^2$.

Another way to tackle the encryption problem is to use a software-based approach which sacrifices at least one of the desirable characteristics of the assembly language multi-precision library : size of implementation, simplicity of use and flexibility. Many software approaches to efficient implementation of the RSA algorithm try to minimise the time for arithmetic computations by using look-up tables. This method is used in "Algorithms for Software Implementations of RSA" [29] in which modular reduction and modular multiplication algorithms are described. Both of these algorithms use look-up tables and it is claimed that the largest operation in the reduction is a 'long' subtraction and in the multiplication it is a 'long' addition. The timings for both these functions contain a constant multiplied by an n^2 term, so the benefits of this implementation method are dependent on keeping the constant as small as possible when coding.

Comba in his "Exponentiation Cryptosystems on the IBM PC" [19] relies mainly on the idea of reducing the number of instructions by re-writing loops as in-line code. His unraveled code for multiplication, for example, takes two thirds of the time that the normally looped code takes. Due to the absence of a hardware multiply on his processor, most of his timings fall short of the DSP assembly implementations : Comba's multiplication of two 30 word numbers takes 4.52 milliseconds as against 0.872 milliseconds for the DSP function. His modular exponentiation method again involves pre-computation of a look-up table. As a result his encryption time for a 33 word block-size is 35 milliseconds which is better than the DSP library result of 98.4 milliseconds. Comba's multiplication procedure alone consists of about 12K bytes of code.

The final suggested method of improvement for the performance of algorithms developed using the DSP library, is to try to improve on certain aspects of the library functions keeping encryption in mind, while still maintaining the size, user-friendliness and flexibility of the library. The normalisation procedure used in the division function involves multiplying both the numerator and the divisor by the same constant. The overhead for this procedure can be eliminated if the divisor is chosen as an appropriate value to fit in, without normalisation, with the workings of the division algorithm. However this idea will only save 43 + 7N + 7M instructions using the notation from the division timing section. An effort has been made throughout to optimise in particular the high iteration loops which result in the largest timing expense so it is unlikely that significant improvements over the timings produced can be achieved without taking a radically different approach such as the two already mentioned. The library would also be improved by adding additional functions, including modular multiplication, modular squaring and functions which facilitate the initialisation of public-key encryption schemes, and also by changing the way that the functions have been implemented so that they have fewer calling restrictions.

BIBLIOGRAPHY

Cited References

1. The First Ten Years of Public-Key Cryptography.
W. Diffie, Proceedings of the IEE, Vol. 76, No.5, May 1988.

2. Military Secure Speech Using HF.
P. Hellen, Electronics and Power, March 1985.

3. High-Quality Coding of Telephone Speech and Wideband Audio.
N. S. Jayant, Jan. 1990 IEEE Communications Magazine.
Proceedings of the IEE, Vol. 76, No.5, May 1988.

4. DVS200 Speech Encryption Processor.
Marconi Electronic Devices, February 1990.

5. Multi-precision Integer and Rational Arithmetic C Library Users Manual.
Dr. Michael Scott, Shamus Software Ltd., Dublin, Ireland, October 1, 1990.

6. The Art of Computer Programming, Vol. 2 : Seminumerical Algorithms.
Donald E. Knuth.

7. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.
Rivest, Shamir and Adleman, Communications of the ACM, January 1983, Volume 26, No. 1.

8. A Simple Unpredictable Pseudo-Random Number Generator
L. Blum, M. Blum and M. Shub, SIAM J. Comput., Vol. 15. No.2, May 1986.

9. Cipher Systems.
H. Beker & F. Piper, van Nostrand, London, 1982.

10. Computer Networks.
Andrew Tanenbaum, Englewood Cliffs, N.J. : Prentice-Hall, 1981.

11. Security for Computer Networks.
D. Davies & W. Price. John Wiley & Sons 1984.

12. New Directions in Cryptography.
W. Diffie & M. Hellman, IEEE Transactions on Information Theory, Sep. 1976.

13. Modern Cryptology : A Tutorial.
Gilles Brassard, Lecture Notes in Computer Science.

14. Teamed for Success : NEC Digital Signal Processors and Speech Processing Devices.

15. Signal Processing Chips.
D. Quarmby, Prentice-Hall, Englewood Cliffs, N.J., 1984

16. The Versatility of Digital Signal Processing Chips.
Aliphas and Feldman, IEEE Spectrum, June 1987.

17. TMS320C25 User's Guide.
Digital Signal Processor Products. Texas Instruments.

18. The Art of Computer Programming, Vol. 1 : Fundamental Algorithms.
Donald E. Knuth.

19. Exponentiation Cryptosystems on the IBM PC.
P. G. Comba, IBM Systems Journal, Vol. 29, No. 4, 1990.

20. Wave of Advances Carry DSPs to New Horizons.
Steven Martin, Computer Design, September 15 1987.

21. A Fortran Multiple-Precision Arithmetic Package.
Richard Brent, ACM Transactions on Mathematical Software, Volume 4, March 1978.

22. A Multiprecise Integer Arithmetic Package.
Duncan Buell & Robert Ward, The Journal of Supercomputing 3, 1989.

23. TMS320C25 Software Development System User's Guide.
Digital Signal Processor Products, Texas Instruments (Preliminary), 1987.

24. TMS320C25 C Compiler Reference Guide.
Digital Signal Processor Products. Texas Instruments. 1988

25. TMS320C1x/TMS320C2x Assembly Language Tools User's Guide.
Digital Signal Processor Products, Texas Instruments, 1987.

26. A Review of the DES Algorithm and the RSA Algorithm for Applications in Commercial Cryptography.
D. Treacy, T. Coffey, C. J. Burkley, National Institute for Higher Education Limerick.
J. Owen-Jones, Digital International Clonmel.

27. A Description of a Single-Chip Implementation of the RSA Cipher.
Ronald Rivest, Lambda, Fourth Quarter, 1980.

28. A Method for Public-Key Encryption using Systolic Arrays.
Findlacy & Johnson, Hatfield Polytechnic.

29. Algorithms for Software Implementation of RSA.
A. Selby & C. Mitchell, IEE Proceedings, Volume 136, May 1989.

30. Secure Speech Communications.
H. Beker and F. Piper, Academics Press, 1985.

Additional Reading

Encryption

Lucifer, A Cryptographic Algorithm.
Arthur Sorkin.
Cryptologia, Volume 8, Number 1, January 1984.
- Predecessor of DES. Design principles and properties. FORTRAN implementation.

Design and Deployment of an Integrated Data Ciphering Unit inside a Low Bit Rate Voice Transcoder for Secure Voice Communications over Telephone Networks.
Sundareshan & Ramaswamy
- Data ciphering unit inside low bit rate voice transcoder presented. Accuracy of ADPCM algorithm and impact of errors due to decoding analysed. Cost analysis.

A New Speech Signal Scrambling Method for Secure Communications : Theory, Implementation and Security Evaluation.
Del Re, Fontacci & Maffucci.
IEEE Journal on Selected Areas in Communications, May 1989.
- 2 D signal scrambling method using DSP techniques that eliminate need for frame synchronisation without impairing security. Hardware : use of digital FIR filters. Methods for determining key-space and key.

Encryption.
Mike Sanders.
Practical Electronics, July 1989.
- Introduction to encryption.

A Polynomial-Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem.
Adi Shamir.
IEEE Transactions on Information Theory, Volume IT-30,
Number 5, September 1984.
- Cryptanalysis theory.

New Trapdoor-Knapsack Public-Key Cryptosystem.
Goodman & McAuley.
IEE Proceedings, Volume 132, Part E, Number 6, November 1985.
- Improvement on Merkle-Hellman Scheme with timing benefit over RSA.

Secret-Key Hardware Public-Key Cryptosystem.
S. Kak.
IEE Proceedings, Volume 133, Part E, Number 2, March 1986.

Using the Data Encryption Standard.
Brian McArdle.

A Modification of the RSA Public-Key Encryption Procedure.
H. Williams.
IEEE Transactions on Information Theory, Volume IT-26, Number 6, November 1980.
- Modification of RSA to produce scheme in which breaking of encryption scheme is provable equivalent to factoring.

Factoring Large Integers on Small Computers.
Michael Scott.
School of Computer Applications Working Paper : CA-0189, DCU.
- Modern integer factorisation techniques reviewed and implemented in C++ and stanard portable C. Useable on PCs.

Advertisements and Devices

DPX : A Compact Duplex Speech Scrambler for Use on the Public Telephone Network.
- A brief avertising description of the device.

Telecom NEC Radiocom, Datacom, Telephone-Switch.
- Device Descriptions.

Wave of Advances Carry DSPs to New Horizons.
Steven Martin.
Computer Design, September 15, 1987.
- Descriptions and comparisons of wide range of DSPs.

Speech Coding

Speech Coding Technology for ATM Networks.
Kitawaki, Wagabuchi, Taka & Takahashi.
IEEE Communications Magazine, January 1990.
- Asynchronous Transfer Mode technology expected to enable Broadband Integrated Digital Networks to handle integrated traffic ranging from narrowband voice and data sevices to broadband video services, including High Definition TV. Describes a type of speech coding for ATM networks which improves quality. Applications also discussed.

Linear Predictive Coding of Speech : Review and Current Directions.
Manfred Schroeder.
IEEE Communications Magazine, Volume 23, Number 8, August 1985.
- Overview/review of whole LPC area.

Predictive Coding of Speech at Low Bit Rates.
Bishnu Atal.
IEEE Transactions on Communications, April 1982.
- Lower than 10 kbits/second predictive coding relying on speech redundancy and minimising perceptual distortion.

Binary Code Excited Linear Prediction (BCELP) : New Approach to CELP Coding of Speech without Codebooks.
R. Salami.
Electronics Letters, Volume 25, Number 6, 16 March 1989.
- New analysis-by-synthesis speech coding approach presented : good quality speech coding in the vicinity of 4.8 kbits/second.

Classified Vector Excitation Speech Coding.
S. Marlow & B. Buggy.
- Drawbacks of CELP include large amount of computation, sharp impulse in coded residual waveform. Uses selective modelling of the LPC residual to attempt to overcome both these drawbacks by using separate codebooks for separate excitation types.

Number Theory

RSA and Rabin Functions : Certain Parts are as Hard as the Whole.
Alexi, Chor, Goldreich & Schnorr.
SIAM Journal of Computing, April 1988.
- Proves equivalence of factoring problem to guessing the parity of the least significant bit of plaintext from the ciphertext.

A Key Distribution System Equivalent to Factoring.
Kevin McCurley.
Journal of Cryptology 1988.
- Describes 2 public key systems incorporating factoring and discrete logartihm problems.

Efficient, Perfect Random Number Generator.
Micali & Schnorr.
- Parallelisation of pseudo-random number generators to produce bits at any speed if sufficiently parallel.

Appendix I

Addition Source Code

The following program is the source code for the addition function.
It is included as an illustration of the programing method.

```
* 22/1/91
            .def            _add
*
*  Knuth's add algorithm implemented on the TMS320C25.
*  Compatible with cross-compiled C code from the TMS320C25 C Compiler.
*  Adds two multi-precision numbers x and y giving result z.
*  Note x >= y.
*
ONEPRC      EQU         0               ; Will contain 1 if single precision add
*                                       ; left after all 2 precision adds, else 0
TWOPRC      EQU         1               ; Calculated number of 32-bit digits in
*                                       ; smaller number to be added
AR0STR      EQU         2               ; AR0 stored here
AR1STR      EQU         3               ; AR1 stored here
AR3STR      EQU         4               ; AR3 stored here
AR4STR      EQU         5               ; AR4 stored here
AR5STR      EQU         6               ; AR5 stored here
TEMP        EQU         7
**************************************************************************
****
* FUNCTION INITIALISATION
*
_add:
            POPD        *+              ; Pop return address
            SAR         AR0,*+          ; Push on system stack
            SAR         AR1,*           ; Save old FP
            LARK        AR0,3       ;
            LAR         AR0,*0+,AR1     ; FP = old SP, SP += SIZE
*
**************************************************************************
****
* REGISTER AND PRECISION COUNT ITERATION SET UP
*
            LDPK        6               ; Data page 6 : data memory 300h to 37Fh
            SBRK        6
            LAR         AR3,*-          ; Load      AR3 with &X[0]
            LAR         AR4,*-          ; Load      AR4 with &Y[0]
            LAR         AR5,*,AR2       ; Load      AR5 with &Z[0]
            SAR         AR0,AR0STR      ; Store AR0
            SAR         AR1,AR1STR      ; Store AR1
            SAR         AR3,AR3STR      ; Store AR3
            SAR         AR4,AR4STR      ; Store AR4
            SAR         AR5,AR5STR      ; Store AR5
********
```

```
            LARP    AR3             ; Select &X[] pointer
            LAR     AR6,*,AR3       ; Counter for copy operation : AR6 <- X[0]
COPY        LAC     *+,AR5          ;
            SACL    *+,AR6          ; Z[N] = X[N], N++
            BANZ    COPY,*-,AR3     ;
********
            LARP    AR5             ;
            ZAC                     ;
            SACL    *,AR3           ; Zero Z[N+1], select X[] pointer
            SACL    ONEPRC          ; Zero single precision flag
            LAR     AR3,AR3STR      ; Restore AR3,AR5 = X,Z pointers
            LAR     AR5,AR5STR      ;
            MAR     *+,AR4          ; Point to X[1] initially
            LAC     *+,AR5          ; Acc <- no. of digits in Y,
*                                   ; AR4 points to Y[1]
            MAR     *+,AR3          ; AR5 points to Z[1]
            LARK    AR0,3           ; Double precision addition loop
*                                   ; offset = 3
            RSXM                    ; => C <- l.s.b. , m.s.b. <- 0
            SFR                     ; Double precision count = Y[0] div 2
*                                   ; Single precision count = Y[0] mod 2
            BZ      NODBLP          ; ACC = 0 => no double precision add
            SACL    TWOPRC          ;
            BNC     NOSNGL          ; C = 0 => no single precision add
            LACK    1               ; Set single precision flag
            SACL    ONEPRC          ;
NOSNGL      ADDK    0               ; Ensure C = 0
*
******************************************************************************
* DOUBLE PRECISION ADD LOOP
*
            MAR     *+,AR1          ; AR3 points to X[2]
            LAR     AR1,TWOPRC      ; Double precision add iteration counter
            MAR     *-,AR3          ; Loop executes <AR1> + 1 times
LOOP1       ZALH    *-              ;   ACC = X[N+1] 00,         AR3 <- N
            ADDC    *0+,AR4         ;       = X[N+1] X[N]+C,     AR3 <- N+3
            ADDS    *+              ;       = X[N+1] X[N] + 00 Y[N]+C
            ADDH    *+,AR5          ;      = X[N+1] X[N] + Y[N+1] Y[N]+C
            SACL    *+              ;   =>  = Z[N+1] Z[N]
            SACH    *+,AR1          ;
            BANZ    LOOP1,*-,AR3    ;   N <- N - 1
*
            MAR     *-              ; AR3 <- N+2
            LAC     ONEPRC          ; ONEPRC = 1 if single precision add left
            BZ      CFLGCK          ; Case: single precision: N, carry: ?
            BNZ     SPADD           ; Single precision add
*
******************************************************************************
* SET UP FOR NO DOUBLE PRECISION ADD
```

```
*
NODBLP    BNC       FINISH          ; No single precision add needed
          ADDK      0               ; Zero carry before single precision add
*
****************************************************************************
* SINGLE PRECISION ADD              ; Finished double precision adds =>
*                                   ; here do single precision add, then
*                                   ; carry at end (Y/N)?
SPADD     LARP      AR3             ; Case: single precision: Y, carry: ?
          ZALS      *,AR4           ; ACC = X[N]
          ADDC      *,AR5           ; ACC = X[N] + Y[N] + C : Add
*                                   ; with carry
          SACL      TEMP            ;
          ANDK      1000h,4         ; Checking for single precision carry
          BNZ       SPADDC          ; Case: single precision: Y, carry: Y
          LAC       TEMP            ; Case: single precision: Y, carry: N
          SACL      *               ; Store result of single precision add
          B         FINISH          ; Finish
*
****************************************************************************
* SINGLE PRECISION CARRY
*
SPADDC    LAC       TEMP            ; Case: single precision: Y, carry: Y
          SACL      *+              ; Z[N]
          LAC       *               ;
          ADDK      1               ; Aim : Z[N+1] <- Z[N+1] + 1
*                                   ; but must allow for possible carry
*                                   ; generated by this sum
          SACL      TEMP            ;
          ANDK      1000h,4         ;
          BNZ       SPADDC          ; Carry generated so propagate forward
          LAC       TEMP            ; No additional carry so just Z[N+1]++
          SACL      *               ;
          B         Z0CKIN          ; Have done carry, must check if Z[0]
*                                   ; increment necessary
*
****************************************************************************
* DOUBLE PRECISION CARRY
*
CFLGCK  BNC         FINISH          ; Case: single precision: N, carry flag: N
          LARP      AR5             ; Case: single precision: N, carry flag: Y
          LAC       *               ;
          ADDK      1               ; Aim : Z[N+1] <- Z[N+1] + 1
*                                   ; but must allow for possible carry
*                                   ; generated by this sum
          B         FRWARD          ;
DPCPRP    LAC       TEMP            ; Double precision carry propagation
          SACL      *+              ;
          LAC       *               ;
```

```
        ADDK    1               ;
FRWARD  SACL    TEMP            ;
        ANDK    1000h,4         ;
        BNZ     DPCPRP          ; Carry generated so propagate forward
        LAC     TEMP            ; No additional carry so just Z[N+1]++
        SACL    *               ; Have done carry, must check if Z[0]
*                               ; increment necessary
*
*******************************************************************************
* Z[0] : CHECK & INCREMENT IF NECESSARY
*
Z0CKIN  LAR     AR3,AR3STR      ; The function will only reach this point
        LAR     AR4,AR4STR      ; if the additions ended with a carry.
        LAR     AR6,AR5STR      ; If X[0] = Y[0] ,
*                               ; Z[0] ++ will be necessary.
        LARP    AR3
        LAC     *,AR4           ; Acc <- X[0]
        SUB     *,AR5           ; Acc <- Acc - Y[0]
        BZ      Z0INC           ; Acc = 0 if X[0] - Y[0] = 0
        LAC     *               ; Acc <- Z[n].
*                               ; If Z[n]=1, carry has resulted
        SUBK    1               ; in additional place being generated
        BNZ   FINISH            ; so Z[0]++ necessary.
Z0INC   LARP    AR6
        LAC     *               ; Acc <- Z[0]          |
        ADDK    1               ; Acc <- Acc + 1       -   => Z[0] ++
        SACL    *               ; Z[0] <- Acc          |
*
*******************************************************************************
* FUNCTION TERMINATION
*
FINISH  LAR     AR0,AR0STR      ; Restore AR0
        LAR     AR1,AR1STR      ; Restore AR1
        LARP    AR1             ;
        ADRK    4               ; Deallocate frame
        LAR     AR0,*-          ; Restore FP
        PSHD    *               ; Put return address on internal stack
        RET                     ; Return to caller
        .end
```

Appendix II

Subtraction Source Code

```
* 15/1/91
              .def          _sub
*
*  Knuth's subtract algorithm implemented on the TMS320C25.
*  Subtracts two multi-precision numbers x,y giving result z.
*
*
ONEPRC        EQU           0                  ; Will contain 1 if single precision subtract
*                                              ; left after all double precision subtractions
*                                              ; else 0
TWOPRC        EQU           1                  ; Calculated number of 32-bit digits in
*                                              ; subtraction sum
AR0STR        EQU           2                  ;
AR1STR        EQU           3                  ;
AR3STR        EQU           4                  ;
AR5STR        EQU           5                  ;
*****************************************************************************
* FUNCTION INITIALISATION
*
_sub:
              POPD          *+                 ; Pop return address
              SAR           AR0,*+             ; Push on system stack
              SAR           AR1,*              ; Save old FP
              LARK          AR0,3              ;
              LAR           AR0,*0+,AR1        ; FP = old SP, SP += SIZE
*
*****************************************************************************
* REGISTER AND PRECISION COUNT ITERATION SET UP
*
              LDPK          6                  ; Data memory 300h to 37Fh
              SBRK          6
              LAR           AR3,*-             ; Load AR3 with &X[0]
              LAR           AR4,*-             ; Load AR4 with &Y[0]
              LAR           AR5,*,AR2          ; Load AR5 with &Z[0]
              SAR           AR0,AR0STR         ; Store AR0
              SAR           AR1,AR1STR         ; Store AR1
              SAR           AR3,AR3STR         ; Store AR3
              SAR           AR5,AR5STR         ; Store AR5
              ZAC                              ; Zero single precision flag
              SACL          ONEPRC             ;
********
              LARP          AR3                ; Select &X[] pointer
              LAR           AR6,*,AR3          ; Counter for copy operation: AR6 <- X[0]
COPY          LAC           *+,AR5             ;
              SACL          *+,AR6             ; Z[N] = X[N],      N++
              BANZ          COPY,*-,AR3        ;
********
              LAR           AR3,AR3STR         ; Restore AR3,AR5 = X,Z pointers
              LAR           AR5,AR5STR         ;
```

```
        LARP    AR4             ; Select Y[] pointer
        LAC     *+,AR5          ; ACC <- Y[0], AR4 -> &Y[1]
        MAR     *+              ; AR5 -> &Z[1]
        RSXM                    ; so that SFR produces logical right shift
        SFR                     ; ACC <- Y[0] div 2, C <- Y[0] mod 2
        BZ      NODBLE          ;
        SACL    TWOPRC          ; Number of double precision subtractions
        BNC     NOSNGL          ;
        LACK    1               ;
        SACL    ONEPRC          ; Set single precision subtraction
*
*************************************************************************
* DOUBLE PRECISION SUBTRACTION WITH CARRY GENERATION
*
NOSNGL  LARK    AR0,2           ; Offset used for auxiliary reg. increment
*                               ; between double precision subtracts
        LAR     AR6,TWOPRC      ; Double precision counter
        LARP    AR3             ; Select X pointer
        MAR     *0+             ; AR3 -> &X[2]
        ZALH    *-              ; ACC = X[2] 00
        ADDS    *0+,AR4         ;     = X[2] X[1]
        SUBS    *+              ;     = X[2] X[1] -   00 Y[1]
        SUBH    *+,AR5          ;     = X[2] X[1] - Y[2] Y[1]
        SACL    *+              ; =>  = Z[2] Z[1]
        SACH    *+,AR6          ;
        MAR     *-              ;
        BANZ    TWDBLE          ; AR6 is counter for number of double
*                               ; precision subtractions
        LAC     ONEPRC          ; Case: One double precision
*                               ; subtraction only
        BGZ     SNGLPR          ; Now: Single precision subtraction left?
        BNC     BORROW          ;       Finished except for borrow
        BC      FINISH
*
*************************************************************************
* DOUBLE PRECISION SUBTRACTION LOOP
* This section is only entered if the sum involves at least two
* double precision subtractions
*
TWDBLE  LARP    AR3
DBLEPR  ZALS    *+,AR4          ; ACC = 00 X[N]
        SUBB    *+,AR3          ;     = 00 X[N] - 00 Y[N] - C
        ADDH    *+,AR4          ;     = X[N+1] X[N] - 00 Y[N] - C
        SUBH    *+,AR5          ;     = X[N+1] X[N] - Y[N+1] Y[N] - C
        SACL    *+              ; =>  = Z[N+1] Z[N]
        SACH    *+,AR6
        BANZ    DBLEPR,*-,AR3   ; AR6 is counter for number of double
*                               ; precision subtractions
        LAC     ONEPRC          ; Case: More than one double precision
```

```
*                                       ;       subtraction
          BGZ       SNGLPR              ; Now: Single precision subtraction left?
          BNC       BORROW              ;       Finished except for borrow
          BC        FINISH
*
*****************************************************************************
* SET UP FOR NO DOUBLE PRECISION CARRY
*
NODBLE    BNC       FINISH
          LARP      AR3                 ; Carry = 1 here
          MAR       *+                  ; Point to &X[1]
*
*****************************************************************************
* SINGLE PRECISION SUBTRACTION
*
SNGLPR    LARP      AR3
          LAC       *,AR4
          SUBB  *,AR5                   ; X[N] - Y[N] = Z[N]
          SACL  *+
          BC        FINISH
BORROW    LARP      AR5
          LAC       *
          SUBK  1                       ; Propagate borrow forward
          SACL      *+
          BNC       BORROW
*****************************************************************************
* FUNCTION TERMINATION
*
FINISH    LAR       AR1,AR1STR          ; Restore AR1
          LARP      AR1
          LAR       AR0,AR0STR          ; Restore AR0
          ADRK      4                   ; Deallocate frame
          LAR       AR0,*-              ; Restore FP
          PSHD      *                   ; Put return address on internal stack
          RET                           ; Return to caller
          .end
```

Appendix III

Multiplication Source Code

```
* 11/6/'91
        .def    _mult
* Note : error encountered in last version if U = 0 : catered for here.
* multiprecision multiplication : algorithm Knuth V2 p253
* ( U[1]..U[n] ) * ( V[1]..V[m] ) -> W[1]..W[n+m]
* This algorithm and the MIRACL differ from Knuth's version
* in that here '1' indicates the least significant digit,
* n, m, n + m the most significant digits.
*
* (i)           W[1]..W[n] <- 0
*               j <- 1
*
* (ii)  => product = 0 if multiplier = 0 so can skip iii,iv
*       (ii) has been left out.
*
* (iii) i <- 1, i+j <- 1, k <- 0
*
* (iv)  t <- U[ i ] * V[ j ] + W[ i+j ] + k
*               W[ i+j ] <- t mod b     => acc low
*               k  <- |_ t/b _|         => acc high
*
* (v)   i++   if i <= n go to (iv) else W[ j ] <- k
*
* (vi)  j++   if j <= m go to (iii) else finish
*
* Indexing notes :
*                       AR3 => Loaded with      : &U[0] which contains n
*                                        i      : index of multiplier  U
*                       AR4 => Loaded with      : &V[0] which contains m
*                                        j      : index of multiplicand V
*                       AR5 => Loaded with      : &W[0] which will contain n + m
*                                  i+j          : index of product W
*                       AR0 => Index used to find end of U,V
*
CARRY       EQU     0                   ;
AR0STR      EQU     1                   ;
AR3STR      EQU     2                   ; &U[0] stored here
AR4STR      EQU     3                   ; &V[0] stored here
AR5STR      EQU     4                   ; &W[0] stored here
TEMP0       EQU     5                   ;
U_N         EQU     6                   ; &U[n] stored here
V_M         EQU     7                   ; &V[m] stored here
*****************************************************************************
* FUNCTION INITIALISATION
*
_mult:
            POPD    *+                  ; Pop return address
            SAR     AR0,*+              ; Push on system stack
            SAR     AR1,*               ; Save old FP
```

```
            LARK        AR0,3              ;
            LAR         AR0,*0+,AR1        ; FP = old SP, SP += SIZE
*
*******************************************************************************
*(i)
*
            LDPK        6                  ; Data memory 300h to 37Fh
            SBRK        6
            LAR         AR3,*-             ; AR3 -> &U[0]
            LAR         AR4,*-             ; AR4 -> &V[0]
            LAR         AR5,*,AR3          ; AR5 -> &W[0]
            SAR         AR0,AR0STR         ; Store AR0
            ZAC                            ;
            SACL        CARRY              ; Zero carry
******** AIM : W[0] <- n + m, W[1]..W[n] <- 0
            SAR         AR3,AR3STR         ; Store &U[0]
            SAR         AR4,AR4STR         ; Store &V[0]
            SAR         AR5,AR5STR         ; Store &W[0]
            LAC         *,AR4              ; ACC <- n
            ADD         *,AR5              ; ACC <- n + m
            SACL        *+,AR3             ; W[0] <- n + m, AR5 -> &W[1]
            LAC         *,AR3              ; ACC <- n
            BZ          UZERO              ; If n = 0, zero W and finish
            LAR         AR0,*,AR0          ; AR0 <- n
            MAR         *-,AR5             ; AR0 <- n - 1 : Loop iteration counter
            ZAC                            ;
ZROWN       SACL        *+,AR0             ;
            BANZ        ZROWN,AR5          ; W[1]..W[n] <- 0
******** AIM : Store &U[n] and &V[m]
*             : AR5 -> W[n+1] so AR5 can be decremented by a constant offset
*             : inside the larger loop.
            LARP        AR3
            LAC         AR3STR             ; ACC <- &U[0]
            ADD         *,AR4              ; ACC <- &U[n]
            SACL        U_N                ;
            LAC         AR4STR             ; ACC <- &V[0]
            ADD         *+,AR3             ; ACC <- &V[m]          AR4 -> V[1]
            SACL        V_M
            LAC         AR5STR             ; ACC <- &W[0]
            ADD         *,AR5              ; ACC <- &W[0] + n
            SACL        TEMP0
            LAR         AR5,TEMP0          ; AR5 -> W[n]
            MAR         *+,AR3             ; AR5 -> W[n+1]
********                                   ; Step (i) over
*(iii)
LOOP2       LAR         AR3,AR3STR         ; AR3 -> U[0]
            SAR         AR5,TEMP0          ;
            LAC         TEMP0              ;
            LARP        AR3                ;
```

```
        SUB     *               ; ACC <- AR5 - n
        SACL    TEMP0           ;
        LAR     AR5,TEMP0       ; AR5 <- AR5 - n
        MAR     *+,AR3          ; AR3 -> &U[1]
        LAR     AR0,U_N         ; Set AR0 for comparison with i counter
********                        ; Step (iii) over
*(iv)
*                               ; Note index values initially '1'
LOOP1   LT      *+,AR4          ; U[i]              , i++
        MPYU    *,AR5           ;     + V[j]
        PAC                     ;
        ADDS    *               ;           + W[i+j]
        ADDS    CARRY           ;                   + k
*                               ; k => ACC high, W[i+j] => ACC low
        SACL    *+,AR3          ; At W[i+j], i+j ++
        SACH    CARRY           ;
*
*               ; Step (iv) over
********
*(v)
        CMPR    2               ; Check if AR3 > AR0
        BBZ     LOOP1,*         ; If i <= n go to (iv)
*
        LARP    AR5             ; i = n so W[j] <- k
        SACH    *+,AR4          ; Carry goes into next product place
*                               ; i+j ++
        LAR     AR0,V_M         ; Set AR0 for comparison with j counter
        ZAC                     ;
        SACL    CARRY           ; Zero carry
*
********                        ; Step (v) over
*(vi)
        MAR     *+,AR4          ; j++
        CMPR    2               ; Check if AR4 > AR0
        BBZ     LOOP2,AR0       ; If j <= m go to (iii)
********        ; Step (vi) over
* If W[n+m] = 0 decrement W[0] by one.
        LARP    AR5             ;
        MAR     *-              ; Point to last filled place
WZRODC  LAC     *-,AR5
        CMPR    0               ; Finish if AR5 points to W[0]
        BBNZ    FINISH,*,AR0    ;
        BNZ     FINISH          ; If W[n+m] <> 0 then finished
        LAR     AR0,AR5STR      ;
        LAC     *               ; Acc <- W[0]
        SUBK    1               ; Acc <- W[0] - 1
        SACL    *,AR5           ; W[0] <- W[0] - 1
        B       WZRODC
*
```

```
*******************************************************************************
* FUNCTION TERMINATION
*
UZERO     ZAC
          LARP    AR5
          SACL    *-              ; Z[1] = 0
          SACL    *               ; Z[0] = 0
FINISH    LAR     AR0,AR0STR      ; Restore AR0
          LARP    AR1             ;
          ADRK    4               ; Deallocate frame
          LAR     AR0,*-          ; Restore FP
          PSHD    *               ; Put return address on internal stack
          RET                     ; Return to caller
          .end
```

Appendix IV

Division Source Code

```
* 13/6/91 : Modifying LOOK AFTER ALL CARRIES section
* 2/8/91  : Making changes to \div\div2.asm version in LOOK AFTER
*                   ALL CARRIES section
                .def            _div
* Dara Murtagh
* Multi Precision Division Algorithm based on Knuth V.2 p.257
* div(u,v,q,r) : q = u div v, r = u mod v
* Notes :
* 1.            In Knuth's algorithm, there are initially n places in V and n+m
*               in U. Here there are n in V and p in U, and m refers to the m
*               in Knuth's version.
* 2.            U and V are both scaled by a calculated NORM. V will not gain
*               more places from this scaling but U may. It is therefore necessary
*               that U is declared in the C calling function as having one more
*               place than is used in that function.
* 3.            Knuth's algorithm is for multi-precision division only which implies
*               V cannot be just one precision. This extra case is checked and catered
*               for here to make the routine a general division algorithm.
*
*       The following bugs from the last version of the div function have been
*               rectified in this file:
*               1.              V is now unnormalised at the end.
*               2.              Q may have leading zeroes at the end: these are removed.
*               3.              Even if norm scaling of U doesn't produce an extra place, a
*                               zero extra place is still noted in U[0] for calculation
*                               purposes. ( i.e. U[0] incremented.)
*               4.              Now check if U < V => can bypass div operation.
*
AR0STR          EQU             0
AR1STR          EQU             1
AR3STR          EQU             3
AR4STR          EQU             4
AR5STR          EQU             5
AR6STR          EQU             6
AR7STR          EQU             7
TEMP            EQU             8
ONEPRC          EQU             9               ; Will contain 1 if single precision subtract
*                                                       ; left after all double precision subtractions
*                                                       ; else 0
TWOPRC          EQU             10                      ; Calculated number of 32-bit digits in
*                                                       ; subtraction sum
NORM            EQU             12
CARRY           EQU             13
QHAT            EQU             14
TEMPLO          EQU             15
TEMPHI          EQU             16
VNQHLO          EQU             17
VNQHHI          EQU             18
TEMP1           EQU             19
```

```
TEMP2       EQU        20
RJCMPR      EQU        21
AR3T        EQU        22
AR4T        EQU        23
AR6T        EQU        24
QHATV       EQU        200h
*
*****************************************************************************
****
*               FUNCTION INITIALISATION
*
_div:
            POPD        *+                  ; Pop return address
            SAR         AR0,*+              ; Push on system stack
            SAR         AR1,*               ; Save old FP
            LARK        AR0,4               ;
            LAR         AR0,*0+,AR1         ; FP = old SP, SP += SIZE
*
*****************************************************************************
*               REGISTER SET-UP,STORE AND NORM DETERMINATION
*
* Information on variable passing procedure : Call(u,v,q,r) stores u,v,q,r.
* The base addresses of these variables are stored on the user stack and
* accessed/recovered using AR1. After the recovery the following register
* variable groupings are valid : u (AR5),v (AR3), q (AR4) and r (AR6).
* (i)
********** REGISTER SET-UP
            CNFD                            ; Configure optional area as data
            LDPK        6                   ; Data memory 300h to 3FFh
            SAR         AR1,AR1STR          ; Store AR1 before it is modified in
recovery
*                                           ; of pointers
            SBRK        7
            LAR         AR5,*-              ; AR5 -> &U[0]
            LAR         AR3,*-              ; AR3 -> &V[0]
            LAR         AR4,*-              ; AR4 -> &Q[0]
            LAR         AR6,*,AR5           ; AR6 -> &R[0]
            RSXM                            ; Surpress sign extension
            SPM         0                   ; Zero P reg shift
********** REGISTER STORE
            SAR         AR0,AR0STR          ; Store registers AR0,3,4,5,6
            SAR         AR3,AR3STR          ; AR1 done above.
            SAR         AR4,AR4STR          ; AR0,1 point to initial values on entering
            SAR         AR5,AR5STR          ; function. AR3,4,5,6 point to
V[0],Q[0],U[0]
            SAR         AR6,AR6STR          ; and R[0] respectively.
********** COPY U TO R
            LAR         AR7,*,AR5           ; AR7 <- p
COPYUR      LAC         *+,AR6              ; ACC <- U[k]
```

```
            SACL        *+,AR7
            BANZ   COPYUR,*-,AR5
            LAR         AR5,AR5STR          ; Restore AR5,AR6
            LAR         AR6,AR6STR
********** Check if U[0] < V[0] and if V[0] = 1
            LAC         *,AR3               ; Acc <- U[0]
            LAR         AR3,AR3STR          ; &V[0]
            SUB         *,AR3               ; Acc <- U[0] - V[0]
            BLZ         QZERO               ; If U[0] < V[0] can immediately
*                                           ; say Q= 0
            LAC         *,AR3               ; Check and branch on V[0] = 1, to one
            XORK        1                   ; precision division
            BZ          VZRONE              ;
********** Calculate the loop counter for the entire function
            LAC         *,AR3               ; Loop should execute m+1 times =>
*                                           ; p-n+1 times
*                                           ; ACC <- n
            ADD         AR6STR              ; ACC <- &R[n]
            SACL        RJCMPR              ; R[j] will be compared with this value
*                                           ; at end
*                                           ; of each iteration to find end of main loop
********** NORM CALCULATION
            LAC         *,AR0               ; Acc <- n
            ADD         AR3STR              ; Acc <- &V[0] + n
            SACL        TEMP
            LAR         AR0,TEMP    ; AR0 -> V[n]
            LAC         *,AR3               ; Acc <- V[n]
            ADDK        1                   ; Acc <- V[n] + 1
            SACL        TEMP
            LACK        65535               ; Acc <- FFFFh = BASE - 1
            ADDK        1                   ; Acc <- 1 0000 h = BASE
            RPTK        15                  ; AccH <- Remainder  AccL <- Quotient
            SUBC        TEMP                ; NORM = |_ BASE / ( V[n]+1 ) _|
            SACL        NORM        ; Norm determined by here.
*
*****************************************************************************
*       SCALE V
            ZAC
            SACL        CARRY               ; Carry initialised to zero
            MAR         *+                  ; AR3 -> V[1], lsb of V
            LT          NORM
SCALEV      MPYU        *
            PAC
            ADDS        CARRY
            SACL        *+                  ; V[1]V[2]...V[N] <-
*                                           ;    (V[1]V[2]...V[N]).norm
            SACH        CARRY
            CMPR        2                   ; AR0 -> V[n]
            BBZ         SCALEV              ; branch back if end of V not
```

```
*                                         ; reached: AR3<>AR0
*                                         ; V scaled by norm here
*                                         ; Don't store final carry because norm
*                                         ; calculated in such a way that final
*                                         ; carry will equal zero.
**************************************************************************
*       SCALE U => SCALE R
            LARP    AR5
            LAC     *,AR6                 ; Acc <- p
            ADD     AR6STR                ; Acc <- &R[0] + p
            SACL    TEMP
            LAR     AR0,TEMP              ; AR0 -> R[p]
            MAR     *+                    ; AR6 -> R[1]
SCALEU      MPYU    *                     ; T already contains NORM
            PAC
            ADDS    CARRY
            SACL    *+                    ; R[1]R[2]...R[p] <- (R[1]R[2]...R[p]).norm
            SACH    CARRY
            CMPR    2
            BBZ     SCALEU
            SACH    *,AR1                 ; Store final carry
            LAR     AR1,AR6STR            ; Increment R[0] even if final carry was
*                                         ; zero
            LAC     *                     ; so that Qhat
            ADDK    1                     ; = |_( R[j].BASE + R[j-1] ) / V[n]_|
            SACL    *,AR3                 ; < BASE even if only because R[j] = 0
*                                         ; R scaled by NORM here
*                                         ; End of D1
**************************************************************************
*(ii)   REGISTER INITIALISATION FOR LOOP
RGINIT      MAR     *-,AR4                ; AR3 -> V[n], AR6 -> R[p]
            LAC     AR4STR                ; ACC <- &Q[0]
            LAR     AR4,AR5STR            ;
            ADD     *,AR6                 ; ACC <- &Q[0] + p
            SACL    TEMP                  ;
            LAR     AR4,TEMP              ; AR4 -> Q[p] i.e. point to msdigit of Q
*
**************************************************************************
*(iii)  CALCULATE QHAT
*                                         ; AR3 -> V[n]
*                                         ; AR4 -> Q[j], j initially p
*                                         ; AR6 -> R[j], ARP -> AR6
CALCQH  LAC         *,AR3                 ; compare R[j] with V[n]
            XOR     *,AR6                 ; ACC <- 0 only if R[j] = V[n]
            BZ      RJEQVN                ; if equal Qhat <- FFFFh ( BASE-1 )
            ZALH    *-                    ; else Qhat <- |_ (R[j].BASE
            ADDS    *+,AR3                ;                        +R[j-1])
            RPTK    15                    ;
            SUBC    *,AR3                 ;                           / V[n] _|
```

```
            B           RJNEVN
RJEQVN      LACK        65535
*
RJNEVN      LARP        AR4
            SACL        *               ; Qhat calc.ed here
********** NOW CHECK IF QHAT TOO BIG
BACK1   LARP            AR4
            LT          *,AR3           ; Qhat
            MPYU        *-,AR6          ; V[n]
            PAC
            SACL        VNQHLO
            SACH        VNQHHI          ; V[n] * Qhat
            ZALH        *-              ; R[j].BASE +
            ADDS        *-              ;           R[j-1]
            SUBS        VNQHLO          ;                   - V[n] * Qhat
            SUBH        VNQHHI
********
            SACL        TEMP
            ANDK        8000h,1         ; If R[j].BASE + R[j-1] -
*                                       ; V[n] * Qhat > FFFFh
*                                       ; multiplying it by 10000h will produce
*                                       ; a 3 digit number ( base 65536)
*                                       ; which is always > V[n-1] * Qhat so
*                                       ; branch
            BNZ         NTGRTR
            ZALH        TEMP            ; ( R[j].BASE + R[j-1] - V[n] * Qhat)
*                                                                    * BASE
            ADDS        *,AR3           ;                            + R[j-2]
            SACL        TEMPLO
            SACH        TEMPHI          ; Note that T already contains Qhat
            MPYU        *,AR4           ; V[n-1]
            PAC                         ; Acc <- V[n-1] * Qhat
            SUBS        TEMPLO
            SUBH        TEMPHI
            BNC         NTGRTR          ; Branch if V[n-1].Qhat not >
*                                       ; (U[j].BASE + U[j-1] - QhatV[n]).
*                                       ; BASE + U[j-2]
            LAC         *               ; else Qhat-- and repeat test
            SUBK        1
            SACL        *,AR3           ; Qhat
            MAR         *+,AR6          ; V[j] => j must be incremented by 1
            ADRK        2               ; R[j] => j must be incremented by 2
            B           BACK1
NTGRTR      LARP        AR6             ; AR6 points to R[j],R[j-1],R[j-2]
*                                       ; next iteration, will have to point
*                                       ; to R[j-1],R[j-2],R[j-3] so j -> j-2 -> j-1
            MAR         *+,AR3          ; R[j] => j must be incremented by 1
            MAR         *+,AR7          ; V[j] => j must be incremented by 1
*
```

```
********************************************************************************
*(iv)    MULTIPLY AND SUBTRACT
*
         ZAC
         SACL      CARRY         ; clear carry
         LALK      QHATV
         SACL      TEMP1
         LAR       AR5,TEMP1     ; AR5 points to Qhat.V[1]
         LAR       AR7,AR3STR
         MAR       *+,AR1        ; AR7 -> V[1]
         LAC       AR3STR        ; ACC <- &V[0]
         LAR       AR1,AR3STR    ; AR1 -> V[0]
         ADD       *,AR7         ; ACC <- &V[0] + n
         SACL      TEMP
         LAR       AR0,TEMP      ; AR0 points to end of V i.e. V[n]
QHATVN   MPYU      *+,AR5        ; Qhat * V[i], Note T already contains
*                                ; Qhat
         PAC
         ADDS      CARRY
         SACL      *+,AR7
         SACH      CARRY
         CMPR      2
         BBZ       QHATVN
         LARP      AR5           ; Select Qhat.V[] pointer
         SACH      *+            ; Qhat.(V[1]....V[N]) calculated
*                                ; AR5 -> Qhat.V[N+2]. Qhat.V[1]..V[n]
*                                ; can have n or n+1 terms depending on
*                                ; if the last carry is zero : must adjust
*                                ; Qhat.V[] pointer back to point at the
*                                ; N+1th term so that its value
*                                ; can be used to calculate the two precision
*                                ; subtraction iteration counter and the one
*                                ; precision flag.
         BNZ       AR5OK,AR5
         MAR       *-
*
* Now do R[j-n]..R[j] = R[j-n]..R[j] - Qhat.V[1]..V[n]
* The REGISTER AND PRECISION COUNT ITERATION SET UP section in the sub
* function will be done here and  the rest of the sub function is included
* without alteration except right at the end in the LOOK AFTER ALL CARRIES
* section.
* This is because Qhat.V[1..n] may be greater than R[j-n..j] which is not
* normally allowed for in the sub() function.
* Call to sub : sub(x,y,x) allowed so have integrated sub() function without
* Z[n] = X[n] copy. Want to have registers AR6, AR5 and AR7 contain &X[0],
* &Y[1] and &X[1] respectively => R[j-n-1], Qhat.V[1] and R[j-n].
*
AR5OK    ZAC
         SACL      ONEPRC        ; Zero single precision flag
```

```
        SAR     AR6,TEMP2       ; AR6 -> R[j-1]
        LAC     TEMP2           ; ACC <- &R[j-1]
        LAR     AR6,AR3STR      ; AR6 <- &V[0] , V[0] = n
        LARP    AR6
        SUB     *,AR7           ; ACC <- &R[j-1] - n
        SACL    TEMP
        LAR     AR6,TEMP        ; AR6 -> R[j-n-1]
        LAR     AR7,TEMP
        MAR     *+,AR4          ; AR7 -> R[j-n]
        SAR     AR7,AR7STR      ; AR7STR <- R[j-n]

        SAR     AR5,TEMP        ; &QHAT.V[n+1] or &QHAT.V[n+2]
*                               ; depending on
*                               ; above multiplication result.
        LAR     AR5,TEMP1       ; AR5 -> Qhat.V[1]
*                               ; There are n or n+1 digits (base b) in the
*                               ; result Qhat.V[1]..Qhat.V[n] but must have
*                               ; pointer pointing to one past the end to
*                               ; get correct subtraction of pointers result.
        LAC     TEMP
        SUB     TEMP1           ; ACC <- n or n+1
        SFR
        BZ      NODBLE
        SACL    TWOPRC
        BNC     NOSNGL
        LACK    1
        SACL    ONEPRC          ; Appropriate one and two precision flags
        ZAC                     ; should be calculated by here.
NOSNGL  SAR     AR6,AR6T        ; AR6T <- &R[j-n-1]
********************************************************************************
*           DOUBLE PRECISION SUBTRACTION WITH CARRY GENERATION
*
        LARK    AR0,2           ; Offset used for auxiliary reg. increment
*                               ; between double precision subtracts
        LAR     AR1,TWOPRC      ; Double precision counter
        LARP    AR6             ; Select X pointer
        MAR     *0+             ; AR6 -> &X[2]
        ZALH    *-              ; ACC = X[2] 00
        ADDS    *0+,AR5         ;     = X[2] X[1]
        SUBS    *+              ;     = X[2] X[1] -   00 Y[1]
        SUBH    *+,AR7          ;     = X[2] X[1] - Y[2] Y[1]
        SACL    *+              ; =>  = Z[2] Z[1]
        SACH    *+,AR1          ;
        MAR     *-              ;
        BANZ    TWDBLE          ; AR1 is counter for number of double
*                               ; precision subtractions
        LAC     ONEPRC          ; Case: One double precision subtraction
*                               ; only
        BGZ     SNGLPR          ; Now: Single precision subtraction left ?
```

```
            BNC         BORROW          ;     Finished except for borrow
            BC          ENDSUB
*
******************************************************************************
*              DOUBLE PRECISION SUBTRACTION LOOP
* This section is only entered if the sum involves at least two
* double precision subtractions
*
TWDBLE      LARP        AR6
DBLEPR      ZALS        *+,AR5          ; ACC = 00 X[N]
            SUBB        *+,AR6          ;     = 00 X[N] - 00 Y[N] - C
            ADDH        *+,AR5          ;     = X[N+1] X[N] - 00 Y[N] - C
            SUBH        *+,AR7          ;     = X[N+1] X[N] - Y[N+1] Y[N] - C
            SACL        *+              ; =>  = Z[N+1] Z[N]
            SACH        *+,AR1
            BANZ        DBLEPR,*-,AR6   ; AR1 is counter for number of double
*                                       ; precision subtractions
            LAC         ONEPRC          ; Case: More than one double precision
*                                       ;        subtraction
            BGZ         SNGLPR          ; Now: Single precision subtraction left ?
            BNC         BORROW          ;     Finished except for borrow
            BC          ENDSUB
*
******************************************************************************
*              SET UP FOR NO DOUBLE PRECISION CARRY
*
NODBLE      BNC         ENDSUB
            LARP        AR6             ; Carry = 1 here
            MAR         *+              ; Point to &X[1]
*
******************************************************************************
*              SINGLE PRECISION SUBTRACTION
*
SNGLPR      LARP        AR6
            LAC         *,AR5
            SUBB        *,AR7           ; X[N] - Y[N] = Z[N]
            SACL        *+
            BC          ENDSUB
******************************************************************************
*              LOOK AFTER ALL CARRIES
*
* The subtraction is R[j-n]..R[j] <- R[j-n]..R[j] - Qhat.(V[1]..V[n]).
* There are n+1 places in the part of R under consideration.
* There are n or n+1 places in Qhat.(V[1]..V[n]). This determines the
* number of single and double precision subtractions. The last subtraction
* must be one of the following forms :
* {..        (R[j-1]   ) }  R[j]        or    {(R[j-1]    )  R[j]        }
* {..        (Qhat.V[n]) }          -         {(Qhat.V[n] )  Qhat.V[n+1] } -
*            --------------------             -------------------------
```

```
*
* If a borrow has occured in the second case, ADDBCK should be branched to
* immediately (i). If a borrow occurs in the first case, it should be
* propagated one place forward and if another borrow occurs then ADDBCK
* should be branched to (ii).
*
BORROW    LARP    AR0            ;
          LAR     AR0,AR3STR     ; AR0 -> V[0]
          LAC     AR7STR         ; ACC <- &R[j-n]
          ADD     *,AR7          ; ACC <- &R[j]
          SACL    TEMP           ; AR7 -> R[j] or R[j+1] depending on
*                                ; number of places in Qhat.V[1..n]
          LAR     AR0,TEMP       ; AR0 -> R[j]
          CMPR    2              ; Check if AR7 > AR0
          BBNZ    ADDBCK         ; (i) Branch if trying to propagate carry
*                                ; at end of Z
          LAC     *              ; R[j] <- R[j] - 1
          SUBK    1              ; Propagate borrow forward
          SACL    *+
          BNC     ADDBCK         ; (ii)
          B       ENDSUB
*
*******************************************************************************
*(vi)          ADD BACK
*
* R[j-n]..R[j] <- R[j-n]..R[j] + V[1]..V[n]
*      ^              ^            ^
*      |              |            |
*   n+1 places    n+1 places   n places
*
* Qhat.(V[1]..V[n]) > U => don't continue to propagate borrow forward,
* do add back (ignoring last carry)
* Have included add function with alterations to REGISTER AND PRECISION COUNT
* SET UP  and both SINGLE and DOUBLE PRECISION CARRY sections.
* Final addition whether single or double precision looks like the following :
*             { ..          ( R[j-1] ) }    R[j]
*             { ..          ( V[n]   ) }
*      ---------------------------- +
*
* There will always be a carry out of R[j] which muct be ignored to cancel with
* ignored borrow. Sum always ends with a carry. Sum ends either double
* precision add, carry or double precision add, single precision add, carry.
* Therefore single precision add always followed by carry.
*
ADDBCK    LARP    AR4            ; Q <- Q - 1
          LAC     *
          SUBK    1
          SACL    *,AR6
          LARK    AR0,3          ; Offset = 3
```

```
          LAR     AR6,AR6T        ; AR6 -> R[j-n-1]
          SAR     AR3,AR3T        ; AR3 -> V[n]
          SAR     AR4,AR4T        ; AR4 -> Q[j-1]
********** REGISTER AND PRECISION COUNT ITERATION SET UP
          MAR     *+,AR4          ; AR6 -> R[n-j-1], should -> R[n-j] here
*                                 ; therefore AR6++  => Z[1]
          LAR     AR4,AR3STR      ; AR4 -> V[0]
          MAR     *+,AR3          ; AR4 -> V[1] => Y[1]
          ZAC
          SACL    ONEPRC          ; Zero single precision flag
          LAR     AR3,AR3STR
          LAC     *,AR3           ; ACC <- n
          LAR     AR3,AR6T        ; AR3 -> R[j-n-1]
          MAR     *+              ; AR3 -> R[j-n] => X[1]
*                                 ; RSXM done already => C <- l.s.b.
*                                 ; m.s.b. <- 0
          SFR                     ; Double precision count = X[0] div 2
*                                 ; Single precision count = X[0] mod 2
          BZ      NODBLP          ; ACC = 0 => no double precision add
          SACL    TWOPRC          ;
          BNC     NOSNL           ; C = 0 => no single precision add
          LACK    1               ; Set single precision flag
          SACL    ONEPRC          ;
NOSNL     ADDK    0               ; Ensure C = 0
*
******************************************************************************
*              DOUBLE PRECISION ADD LOOP
*
          MAR     *+,AR1          ; AR3 points to X[2]
          LAR     AR1,TWOPRC      ; Double precision add iteration counter
          MAR     *-,AR3          ; Loop executes <AR1> + 1 times
DPADD     ZALH    *-              ; ACC = X[N+1] 00, AR3<- N
          ADDC    *0+,AR4         ;     = X[N+1] X[N] + C, AR3 <- N+3
          ADDS    *+              ;     = X[N+1] X[N] + 00 Y[N] + C
          ADDH    *+,AR6          ;    = X[N+1] X[N] + Y[N+1] Y[N] + C
          SACL    *+              ; =>  = Z[N+1] Z[N]
          SACH    *+,AR1          ;
          BANZ    DPADD,*-,AR3    ; N <- N + 1
*
          MAR     *-,AR6          ; AR3 <- N+2
          LAC     ONEPRC          ; ONEPRC = 1 if single precision add left
          BZ      CFLGCK          ; Case: single precision: N, carry: ?
          BNZ     SPADD           ; Single precision add
*
******************************************************************************
*              SET UP FOR NO DOUBLE PRECISION ADD
*
NODBLP    BNC     ENDADD          ; No single precision add needed
          ADDK    0               ; Zero carry before single precision add
```

```
*
******************************************************************************
*               SINGLE PRECISION ADD
*                                              ; Finished double precision adds =>
*                                              ; here do
*                                              ; single precision add,then carry at end
SPADD       LARP        AR3                    ; Case: single precision: Y, carry: Y
            ZALS        *,AR4                  ; ACC = X[N]
            ADDC        *,AR6                  ; ACC = X[N] + Y[N] + C : Add with
*                                              ; carry
            SACL        *+                     ; Z[N]
            LAC         *                      ; Z[N+1] = R[j]
            ADDK        1                      ; Aim : Z[N+1] <- Z[N+1] + 1
            SACL        *                      ; Do carry
            B           ENDADD                 ; Ignore further carry, finished
*
******************************************************************************
*               DOUBLE PRECISION CARRY
*
CFLGCK      LAC         *                      ; Case: single precision: N, carry flag: Y
            ADDK        1                      ; Z[N+1] <- Z[N+1] + 1
            SACL        *                      ;
******************************************************************************
ENDADD      LAR         AR3,AR3T               ; Restore AR3 ( = V[n] )
            LAR         AR4,AR4T               ; Restore AR4 ( = Q[j] )
******************************************************************************
ENDSUB      LARP        AR4
            MAR         *-,AR6                 ; AR4 -> Q[j-1]
            LAR         AR6,TEMP2              ; R[j]
            LAR         AR0,RJCMPR             ; R[j]=R[p-n] => finished, else branch back
            CMPR        00
            BBZ         CALCQH
**********
* AR4 should point to correct Q[0], and Q should start at the originally
* alloted location
            LARP        AR1
            LAC         AR4STR                 ; ACC <- &Q[0]
            LAR         AR1,AR6STR             ;
            ADD         *,AR5                  ; ACC <- &Q[0] + p
            SAR         AR4,TEMP               ; AR4 -> Q[0]
            SUB         TEMP                   ; &Q[p] - &Q[0]
            LAR         AR5,AR4STR             ; AR5 -> Q[0]
            SACL        *                      ; Store number of places in Q at Q[0]
            LAR         AR1,*,AR5              ; AR1 <- number of places in Q
            MAR         *+,AR1                 ; AR5 -> new Q[1]
            MAR         *-,AR4                 ; AR1 used as counter for copy,
*                                              ; one too big initially
            MAR         *+                     ; AR4 -> old Q[1]
STOREQ      LAC         *+,AR5                 ; ACC <- Q[n]
```

```
            SACL        *+,AR1
            BANZ        STOREQ,*-,AR4
*****************************************************************************
*               UNNORMALIZE
*
* First strip leading zeros from Q
*           AR1 will contain i
*           AR2 will point to Q[i]
            LAR         AR4,AR4STR          ; ARP -> AR4
            LAC         AR4STR              ; ACC <- &Q[0]
            ADD         *                   ; ACC <- &Q[p]
            LAR         AR1,*,AR2           ; AR1 <- p
            SACL        TEMP                ; TEMP <- &Q[p]
            LAR         AR2,TEMP            ; AR2 -> Q[i], i=p
QLZRO       LAC         *-,AR1              ; ACC <- Q[i], i--
            BZ          QLZRO,*-,AR2        ; i--, Select Q[i] pointer and loop if Q[i]=0
            LARP        AR1
            MAR         *+,AR4              ; Compensate for decrementing n once too
*                                           ; often
            SAR         AR1,*,AR6           ; Store adjusted Q[0]
*                                           ; Leading zeros stripped from Q
*****************************************************************************
*
* Strip leading zeros from R
*           AR1 will contain i
*           AR2 will point to R[i]
            LAR         AR6,AR6STR          ; ARP -> AR6
            LAC         AR6STR              ; ACC <- &R[0]
            ADD         *                   ; ACC <- &R[p]
            LAR         AR1,*,AR2           ; AR1 <- p
            SACL        TEMP                ; TEMP <- &R[p]
            LAR         AR2,TEMP            ; AR2 -> R[i], i=p
RLZRO       LAC         *-,AR1              ; ACC <- R[i], i--
            BZ          RLZRO,*-,AR2        ; i--, Select R[i] pointer and loop if R[i]=0
            LARP        AR1
            MAR         *+,AR6              ; Compensate for decrementing n once too
*                                           ; often
            SAR         AR1,*,AR0           ; Store adjusted R[0]
*                                           ; Leading zeros stripped from R
**********
* Divide R by norm to get unnormalized R
*                                           ; AR5 -> R[0]
            LAR         AR0,AR6STR          ; AR0 -> R[0]
            LAC         AR6STR              ; ACC <- &R[0]
            ADD         *,AR6               ; ACC <- &R[p]
            SACL        TEMP
            LAR         AR6,TEMP            ; AR6 -> R[p]
            CMPR        0                   ; AR6 = AR0 only if R=0 => finished
            BBNZ        NODECR,*,AR0
```

```
        LARP    AR6
        LAC     *               ; ACC <- R[i], i=p
RDVD    RPTK    15
        SUBC    NORM            ; AccH = Remainder, AccL = Quotient
        SACL    *-              ; R[i] <- |_ R[i] / norm _|, i--
        SACH    TEMP
        ZALH    TEMP            ; AccH = Remainder, AccL = 0
        ADD     *               ; AccH = Remainder, AccL = R[i-1]
        CMPR    0               ; Loop unless i = 0
        BBZ     RDVD            ; R[p] may be zero after unnormalization
*                               ; in which case R[0] is one too big.
        LAC     AR6STR          ; ACC <- &R[0]
        ADD     *               ; ACC <- &R[p] as AR6 now -> R[0]
        SACL    TEMP
        LAR     AR6,TEMP        ; AR6 -> R[p]
        LAC     *,AR0           ; ACC <- R[p]
        BNZ     NODECR          ; Branch if no need to decrement R[0]
        LAC     *
        SUBK    1               ; R[0]--
        SACL    *,AR0
**********
* Divide V by norm to get unnormalized V
*
NODECR  LAR     AR0,AR3STR      ; AR0 -> V[0]
        LAC     AR3STR          ; ACC <- &V[0]
        ADD     *,AR3           ; ACC <- &V[p]
        SACL    TEMP
        LAR     AR3,TEMP        ; AR3 -> V[p]
        LAC     *               ; ACC <- V[i], i=p
VDVD    RPTK    15
        SUBC    NORM            ; AccH = Remainder, AccL = Quotient
        SACL    *-              ; V[i] <- |_ V[i] / norm _|, i--
        SACH    TEMP
        ZALH    TEMP            ; AccH = Remainder, AccL = 0
        ADD     *               ; AccH = Remainder, AccL = V[i-1]
        CMPR    0               ; Loop unless i = 0
        BBZ     VDVD            ; V[p] may be zero after unnormalization
*                               ; in which case V[0] is one too big.
        LAC     AR3STR          ; ACC <- &V[0]
        ADD     *               ; ACC <- &V[p] as AR3 now -> V[0]
        SACL    TEMP
        LAR     AR3,TEMP        ; AR3 -> V[p]
        LAC     *,AR0
        BNZ     FINISH          ; Branch if no need to decrement V[0]
        LAC     *
        SUBK    1               ; V[0]--
        SACL    *,AR1
        B       FINISH
*********
```

```
* If U[0] < V[0], Q = 0.
QZERO       LAR         AR4,AR4STR
            LARP        AR4
            ZAC
            SACL        *+              ; Q[0] <- 0
            SACL        *               ; Q[1] <- 0
******************************************************************************
*               FUNCTION TERMINATION
*       RESTORE REGISTERS
FINISH      LAR         AR0,AR0STR      ; Restore AR0
            LAR         AR1,AR1STR      ; Restore AR1
            LARP        AR1
            SBRK        5               ; Deallocate frame
            LAR         AR0,*-          ; Restore FP
            PSHD        *               ; Put return address on internal stack
            RET                         ; Return to caller
*
******************************************************************************
*           SINGLE PRECISION DIVISION CASE
*
* Register initialisation
*
VZRONE  LARP            AR4
            LAC         AR4STR          ; ACC <- &Q[0]
            LAR         AR4,AR5STR      ;
            ADD         *,AR0           ; ACC <- &Q[0] + p
            SACL        TEMP            ;
            LAR         AR4,TEMP        ; AR4 -> Q[p] i.e. point to msdigit of Q
            LAR         AR0,AR6STR      ; AR0 -> R[0]
            LAC         AR6STR          ; ACC <- &R[0]
            ADD         *,AR3           ; ACC <- &R[p]
            SACL        TEMP
            LAR         AR6,TEMP        ; AR6 -> R[p]
*                                       ; Do not need to check here
*                                       ; for AR6 = AR0 which
*                                       ; would imply R = 0 as this case is
*                                       ; checked at start
            LAR         AR3,AR3STR      ; AR3 -> V[0]
            MAR         *+,AR3          ; AR3 -> V[1]
            LAC         *,AR6           ; ACC <- V[1]
            SACL        TEMP1           ; TEMP1 <- V[1]
            LAC         *-,AR4          ; ACC <- R[i], i = p
*********
* Division loop
*
UDVD        RPTK        15
            SUBC        TEMP1           ; AccH = Remainder, AccL = Quotient
            SACL        *-,AR6          ; AR4 -> Q[p] initially
            SACH        TEMP
```

```
            ZALH    TEMP            ; AccH = Remainder, AccL = 0
            ADD     *-              ; AccH = Remainder, AccL = R[ i-1 ]
            CMPR    1
            BBZ     UDVD,*,AR4
*********
            LARP    AR6
            ADRK    2
            SACH    *               ; R[1] = Remainder
            LAC     *-
            BZ      ZERO
            LACK    1               ; R[0] = 1 as U mod V != 0
            SACL    *,AR5
            B       CNTUE
ZERO        ZAC                     ; R[0] = 0 as U mod V = 0
            SACL    *,AR5
*********
* Strip leading zeroes from Q
*
CNTUE       LAR     AR5,AR5STR      ; AR5 <- &U[0]
            LAC   *,AR4
            SACL    *,AR4           ; Q[0] <- p
            LAR     AR5,AR4STR      ; ARP -> AR5
            LAC     AR4STR          ; ACC <- &Q[0]
            ADD     *               ; ACC <- &Q[p]
            LAR     AR1,*,AR2       ; AR1 <- p
            SACL    TEMP            ; TEMP <- &Q[p]
            LAR     AR2,TEMP        ; AR2 -> Q[i], i = p
QLZRO1      LAC     *-,AR1          ; ACC <- Q[i], i--
            BZ      QLZRO1,*-,AR2   ; i--, Select Q[i] pointer and loop if Q[i]=0
            LARP    AR1
            MAR     *+,AR4          ; Compensate for decrementing n once
*                                   ; to often
            SAR     AR1,*,AR1       ; Store adjusted Q[0]
*                                   ; Leading zeros stripped from Q
            B       FINISH
            .end
```

Appendix V

Square Source Code

```
* 14/8/91       SQR.ASM
* Dara  Murtagh
* Modified sqr1.asm. Error in i loop :     Last carry overwriting Z term.
*                                          Fixed by incrementing AR5 in i loop.
                .def          _square
* C call : square(x,z)
* AR3 is loaded with &X[0].
* AR5 is loaded with &Z[0]
*
CARRY           EQU           0
TEMP            EQU           1
TEMP1           EQU           2
AR3STR          EQU           3
AR5STR          EQU           4
********
_square:
                POPD          *+                 ; Pop return address
                SAR           AR0,*+             ; Push on system stack
                SAR           AR1,*              ; Save old FP
                LARK          AR0,1              ;
                LAR           AR0,*0+,AR1        ; FP = old SP, SP += SIZE
********
*                                                ; Register usage
*                                                ; AR0 <- &X[n]
*                                                ; AR2 <- n - 1 , i loop counter
*                                                ; AR3 <- &X[i]
*                                                ; AR4 <- &X[j]
*                                                ; AR5 <- &Z[i+j-1]
                RSXM                             ; Disable sign extension
                LDPK          6                  ; Using data memory 300h to 3FFh
                SBRK          5                  ; Pop passed variable addresses off runtime
*                                                ; stack.
                LAR           AR5,*+,AR1         ; AR5 <- &Z[0]
                SAR           AR5,AR5STR         ; Store &Z[0]
                LAR           AR3,*,AR3          ; AR3 <- &X[0]
********                      if ( x[0]==0 ) { z[1]=z[0]=0; return; }
                LAC           *                  ; Acc <- X[0]
                BNZ           XNTZRO             ; If X[0] != 0 do squaring
                LARP          AR5
                ZAC
                SACL          *+                 ; Z[0]=0
                SACL          *                  ; Z[1]=0
                B             FINISH
XNTZRO          SAR           AR3,AR3STR         ; Store &X[0]
                LAR           AR2,*,AR3          ; AR2 <- n
                LAC           *,1,AR5
                SACL          *,AR5              ; Z[0] = 2 * n
********                      for ( i=1; i<=z[0]; i++ ) z[i]=0;
                LAR           AR6,*,AR6          ; AR6 <- Z[0]
```

```
            MAR     *-,AR5          ; Loop executes <AR6> + 1 times
            MAR     *+              ; AR5 -> Z[1]
            ZAC
ZEROZ       SACL    *+,AR6
            BANZ    ZEROZ,*-,AR5
********
            LAR     AR5,AR5STR      ; AR5 -> Z[1]
            MAR     *+,AR2
            MAR     *-,AR3          ; AR2 <- n-1
            SAR     AR3,TEMP
            LAC     TEMP
            ADD     *,AR3
            SACL    TEMP
            LAR     AR0,TEMP        ; AR0 <- &X[n]
            MAR     *+,AR3          ; AR3 -> X[1]
            SAR     AR5,TEMP
*
*********************** for (i=1; i<=n; i++)
*
ILOOP       LT      *+,AR5          ; T <- X[i]
            MAR     *+,AR4          ; Z[(i++)+j] so that Z[n+i] correctly
*                                   ; accessed
*                                   ; after j loop.
            SAR     AR3,TEMP1
            LAR     AR4,TEMP1       ; AR4 -> X[i+1]
********            carry = 0
            ZAC
            SACL    CARRY           ; Carry <- 0
*
******************  for (j=i+1; j<=n; j++)
*
JLOOP       CMPR    2               ; Branch out of loop if &X[j] > &X[n]
            BBNZ    ENDJLP,*,AR5    ; => if AR4 > AR0
            LAR     AR5,TEMP
            MAR     *+,AR4          ; Z[i+j-1] => i+j-1 ++ each time j++
            SAR     AR5,TEMP        ; Temp <- Z[i + (++j) - 1]
********            product = X[i] * X[j] + carry
            MPYU    *+,AR5          ; X[i] * X[j]
            PAC
            ADD     CARRY           ;             + Carry
********            carry = product div base
            SACH    CARRY
********            Z[i+j-1] += product % base
            ANDK    65535           ; Acc <- product % base
            ADD     *,AR5           ; Acc <- Acc + Z[i+j-1]
            SACL    *+,AR4          ; Z[i+j-1] <- Acc
********            if ( Z[i+j-1] < ( product % base ) ) carry ++
* i.e. if sum of two sixteen bit numbers has produced a seventeen bit result
            ANDK    2,15            ; only need to check single precision
```

```
*                                           ; carry bit
             BZ          JLP
             LAC         CARRY
             ADDK        1
             SACL        CARRY
JLP          B           JLOOP,*,AR4
*
*****************        Z[n+i] = carry
*
ENDJLP       LAC         CARRY
             SACL        *,AR2
             BANZ        ILOOP,*-,AR3
*
*********************** carry = 0
*
             ZAC
             SACL        CARRY
********                 for (i=1; i<= 2*n-1; i++)
             LAR         AR5,AR5STR        ; AR5 -> Z[0] = 2*n
             LARP        AR5
             LAR         AR2,*+,AR2        ; AR2 <- 2*n, AR5 -> Z[1]
             SBRK        2                 ; AR2 <- (2*n) - 2
             LARP        AR5
********                 temp = z[i] * 2 + carry
DBLZ         LAC         *,1               ; Acc <- Z[i]*2
             ADD         CARRY
********                 Z[i] = temp % base
             SACL        *+,AR2
********                 carry = temp div base
             SACH        CARRY
             BANZ        DBLZ,*-,AR5
             LAC         CARRY
********                 Z[2*n] = carry
             SACL        *,AR3
********                 carry = 0
             ZAC
             SACL        CARRY
********                 for (i=1; i<=n; i++)
             LAR         AR3,AR3STR        ; AR3 -> X[0]
             LAR         AR2,*,AR2         ; AR2 <- n
             MAR         *-,AR5            ; AR2 <- n - 1
             LAR         AR5,AR5STR        ; AR5 -> Z[0]
             MAR         *+,AR3            ; AR5 -> Z[1]
             MAR         *+                ; AR3 -> X[1]
*
********                 temp = x[i] * x[i] + carry
*
ISQRLP       LT          *
             MPYU        *+,AR5
```

```
            PAC
            ADD         CARRY
********                carry = temp div base
            SACH        CARRY
            ANDK        65535
            ADD         *
********                Z[2*i-1] += temp % base
            SACL        *+
********                if ( Z[2*i - 1] < (temp%base) ) carry++
            ANDK        2,15
            BZ     CNTU
            LAC         CARRY
            ADDK        1
            SACL        CARRY
********                Z[2*i] = ( Z[2*i] + carry ) % base
CNTU        LAC         *                   ; Acc <- Z[2*i]
            ADD         CARRY
            SACL        *+,AR2
********                if ( Z[2*i] < carry ) carry = 1
********                else carry = 0
            ANDK        2,15
            BZ          CRYZR
            LACK        1
            SACL        CARRY               ; Carry = 1
            B           LOOP
CRYZR       ZAC
            SACL        CARRY               ; Carry = 0
LOOP        BANZ   ISQRLP,*-,AR3
*
********
*
*                      if ( z[z[0]]==0 ) z[0]--
            LAR         AR5,AR5STR          ; AR5 -> Z[0]
            LARP        AR5
            LAC         AR5STR
            ADD         *,AR2
            SACL        TEMP
            LAR         AR2,TEMP            ; AR2 -> Z[Z[0]]
            LAC         *,AR5
            BNZ         FINISH
            LAC         *
            SUBK        1
            SACL        *,AR1
FINISH      LARP        AR1
            ADRK        2                   ; Deallocate frame
            LAR         AR0,*-              ; Restore FP
            PSHD        *                   ; Put return address on stack
            RET                             ; Return to caller
            .end
```

Appendix VI

Comparison Source Code

```
* 22/4/91
* Dara  Murtagh
            .def        _compare
* C call : compare(x,y)
* AR5 is loaded with &X[0].
* AR6 is loaded with &Y[0]
* On return the accumulator is loaded with 2 if x>y, 1 if x<y and 0 if x=y.
*
AR0STR      EQU         0
********
_compare:
            POPD        *+              ; Pop return address
            SAR         AR0,*+          ; Push on system stack
            SAR         AR1,*           ; Save old FP
            LARK        AR0,1           ;
            LAR         AR0,*0+,AR1     ; FP = old SP, SP += SIZE
********
            LDPK        6
            SAR         AR0,AR0STR
            SBRK        5
            LAR         AR4,*+,AR1      ; AR4 <- &X[0]
            LAR         AR5,*,AR5       ; AR5 <- &Y[0]
            LAC         *,AR4           ; Acc <- X[0]
            SUB         *,AR4           ; Acc <- Y[0]
            BGZ         XGRTER
            BLZ         YGRTER
            LAR         AR0,*,AR4
            MAR         *0+,AR5         ; AR4 -> Y[n]
            MAR         *0+,AR5         ; AR4 -> X[n]
LOOP        LAC         *-,AR4          ; Acc <- X[n]
            SUB         *-,AR0          ; Acc <- X[n] - Y[n]
            BGZ         XGRTER
            BLZ         YGRTER
            BANZ        LOOP,*-,AR5
            ZAC
            B           FINISH
XGRTER      LACK        2
            B           FINISH
YGRTER      LACK        1
********
FINISH      LAR         AR0,AR0STR
            LARP        AR1
            ADRK        2               ; Deallocate frame
            LAR         AR0,*-          ; Restore FP
            PSHD        *               ; Put return address on stack
            RET                         ; Return to caller
            .end
```

Appendix VII

Copy Source Code

```
* 6/5/91
* Dara  Murtagh
            .def        _copy
* C call : copy(x,y)
* AR5 is loaded with &X[0].
* AR6 is loaded with &Y[0]
* y := x
*
********
_copy:
            POPD        *+                  ; Pop return address
            SAR         AR0,*+              ; Push on system stack
            SAR         AR1,*               ; Save old FP
            LARK        AR0,1               ;
            LAR         AR0,*0+,AR1         ; FP = old SP, SP += SIZE
********
            SBRK        5
            LAR         AR5,*+,AR1          ; AR5 <- &Y[0]
            LAR         AR4,*,AR4           ; AR4 <- &X[0]
            LAR         AR6,*,AR4           ; AR6 <- n
            LAC         *+,AR5              ; Acc <- X[0]
            SACL        *+,AR6              ; Y[0] <- X[0]
            BZ          ARPSET,*,AR6        ; If X[0]=0, X[1]:=0 and finish
            MAR         *-,AR4              ; Else n-- and continue
            B           COPY
ARPSET      LARP        AR4
COPY        LAC         *+,AR5              ; Acc <- X[n]
            SACL        *+,AR6              ; Y[n] <- X[n]
            BANZ        COPY,*-,AR4
********
FINISH      LARP        AR1
            ADRK        2                   ; Deallocate frame
            LAR         AR0,*-              ; Restore FP
            PSHD        *                   ; Put return address on stack
            RET                             ; Return to caller
            .end
```

Appendix VIII

Leading Zeroes Source Code

```
* 9/4/91
* Dara  Murtagh
            .def        _lzero
* AR5 is loaded with &X[0]. AR7 is used to contain n.
* AR4 is used to point to X[n].
* If X = 0, X[0] is set equal to 0.
*
AR5STR      EQU         0
TEMP        EQU         1
AR1STR      EQU         2
********
_lzero:
            POPD        *+              ; Pop return address
            SAR         AR0,*+          ; Push on system stack
            SAR         AR1,*           ; Save old FP
            LARK        AR0,1           ;
            LAR         AR0,*0+,AR1     ; FP = old SP, SP += SIZE
********
            LDPK        6               ; Data memory 300h to 37Fh
            SAR         AR1,AR1STR
            SBRK        4
            LAR         AR5,*,AR5       ; AR5 <- &X[0]
            SAR         AR5,AR5STR      ; Store &X[0]
            LAC         AR5STR          ; ACC <- &X[0]
            ADD         *               ; ACC <- &X[n]
            LAR         AR7,*,AR4       ; AR7 <- n
            SACL        TEMP            ; TEMP <- &X[n]
            LAR         AR4,TEMP        ; AR4 -> X[j], j=n
LOOP        LAC         *-,AR7          ; ACC <- X[j], j--
            BZ          LOOP,*-,AR4     ; n--, Select X[j] pointer and loop if X[j]=0
            LARP        AR7
            MAR         *+,AR5          ; Compensate for decrementing n once too
*                                       ; often
            SAR         AR7,*,AR1       ; Store X[0]
********
            LAR         AR1,AR1STR
            SBRK        2               ; Deallocate frame
            LAR         AR0,*-          ; Restore FP
            PSHD        *               ; Put return address on stack
            RET                         ; Return to caller
            .end
```

Appendix IX

Zero Source Code

```
* 28/4/91
* Dara  Murtagh
            .def            _zero
* C call : zero(x)
* AR5 is loaded with &X[0].
* X[0] is set equal to 0, X[1] set equal to 0.
*
********
_zero:
            POPD        *+              ; Pop return address
            SAR         AR0,*+          ; Push on system stack
            SAR         AR1,*           ; Save old FP
            LARK        AR0,1           ;
            LAR         AR0,*0+,AR1     ; FP = old SP, SP += SIZE
********
            SBRK        4
            LAR         AR5,*,AR5       ; AR5 <- &X[0]
            ZAC
            SACL        *+,AR5          ; X[0] <- 0
            SACL        *,AR1           ; X[1] <- 0
********
            ADRK        2               ; Deallocate frame
            LAR         AR0,*-          ; Restore FP
            PSHD        *               ; Put return address on stack
            RET                         ; Return to caller
            .end
```

Keys and Formulae used

ke =
2D5D076C3C27AFA45A0682F04F72EF8539C9A15BA46A4FD1CF175E39E1A8C599714
41C6D6D60EB1CB545F2646733E5D6781B18DED86AC7315030741A134BB4099
kd =
1E3E04F2D2C51FC2E6AF01F58A4C9FAE26866B926D9C35368A0F9426967083BB7CB
D029954ABD656E63CD6AF1035692D042D7C470387A08C9F5B562259ACCE2C3

e = m^3 mod ke
m = e^kd mod ke

Encrypt.bat Batch File

```
echo off
echo The file which is to be encrypted gets converted to a standard
echo hex input format for the TMS320C25.
echo The SWDS is then invoked and the encryption program is run.
echo The encrypted result is the file a:three.dat
echo on
hex
swds
type a:three.dat|more
```

Appendix X

Demonstration of the Multi-Precision Arithmetic Assembly Library using an RSA File Encryption Example on the SWDS

The aim of this demonstration example is to show how the digital signal processor assembly routines can be used in a file RSA encryption application. The Software Development System can be operated in a mode that allows IN and OUT instructions to read from and write to disk files respectively. Any legal MS/DOS filename can be associated with any of the 16 input and 16 output ports and these files will be read from and written to by the IN and OUT instructions provided that data logging has been enabled. The logging mode data files must be in the form of a list of 16 bit hexadecimal ASCII numbers :

```
.
.
.
0002
0004
0008
000C
.
.
eof
```

For the purposes of demonstration the example has been limited to encrypt up to 32 characters. The steps of the example are as follows :

1. The message text is written to a file.
2. This file is converted to 16 bit hexadecimal ASCII number format by the HEX program. The output file containing data ready for encryption is a:\test.dat.
3. The SWDS system is invoked. The following data logging files are assigned :

```
     Input  Port 1 : a:\d1.out          ( ke )
     Input  Port 2 : a:\test.dat        ( m  )
     Output Port 3 : a:\three.dat       ( e  )
```

Data logging mode is enabled. A debug session is enabled. The encryption program en.tag is loaded. The Program Counter is set to 1000, auxiliary registers 0 and 1 to 31B and auxiliary register pointer to 1. A GO to 1033 completes all the input. EXECUTE to BREAKPOINT 1075 results in the encryption being

carried out. A GO to 1078 completes the output.
The SWDS session is terminated by a QUIT.
4. The encrypted file containing : $e = m^3 \bmod ke$ is a:\three.dat. This is typed to the screen.
The above steps are carried out by the encrypt.bat batch file, although the data logging, program running and SWDS termination steps in Step 3 must be carried out manually using the keyboard.

Decryption is carried out as follows :
1. The SWDS system is invoked. The following data logging files are assigned :

Input	Port 0	: a:\e1.out	(kd)
Input	Port 1	: a:\d1.out	(ke)
Input	Port 2	: a:\three.dat	(e)
Output	Port 4	: a:\out.dat	(m)

Data logging mode is enabled. A debug session is enabled. The encryption program en.tag is loaded. The Program Counter is set to 1000, auxiliary registers 0 and 1 to 31B and auxiliary register pointer to 1. A GO to 103A completes all the input. EXECUTE to BREAKPOINT 107C results in the encryption being carried out. A GO to 107F completes the output.
The SWDS session is terminated by a QUIT.
2. The decrypted file containing : $m = e^{kd} \bmod ke$ is a:\out.dat. This is converted to character format by the ASCII program before being typed to the screen.
The above steps are carried out by the decrypt.bat batch file, although the data logging, program running and SWDS termination steps in Step 3 must be carried out manually using the keyboard.

Hex.c File Format Conversion Program

```
/*      Dara Murtagh 2/1/1992
        Demo program
        Input text from one file,
        output hex formatted values to another.           */

#include <stdio.h>

void strip(name)
char name[];
{ /* strip off filename extension */
    int i;
    for (i=0;name[i]!='\0';i++)
    {
        if (name[i]!='.') continue;
        name[i]='\0';
        break;
    }
}

main()
{
    FILE *ifile;
    FILE *ofile;
    char ifname[13];
    char ofname[13]="a:test.dat";
    char ch[34];
    int count,i;

    printf("File to be enciphered = ");
    gets(ifname);
    ifile=fopen(ifname,"r");
    printf("\nConverting file from ascii format to hex format for input to\n");
    printf("the digital signal processing system encryption algorithm.\n");
    ofile=fopen(ofname,"wb");       /* NOT standard C! */
    count=0;
    do
        {
        ch[count]=fgetc(ifile);
        count++;
        }
    while ( (ch[count]!=EOF) && (count<=32) );
    fprintf(ofile,"%04X\r\n",count+1);
    for (i=0; i<=count; i++)
        fprintf(ofile,"%04X\r\n",ch[i]);
    fclose(ofile);
    fclose(ifile);
}
```

RSAEN.C : Encryption Program which uses the Assembly Language Library

```
/* Encryption: e = (m expo 3) mod ke */

main()
{
unsigned int ke[34],e[34],m[44],two[2],three[2];
unsigned int zero[2],parity[2],temp[67],temp1[67];

rd1(ke);
rd2(m);

three[0]=1;    three[1]=3;
two[0]=1;      two[1]=2;
zero[0]=0;     zero[1]=0;
e[0]=1;             e[1]=1;

for (;;)
    {
        div(three,two,temp,parity);
        copy(temp,three);
        if (parity[1]==1)
        {
            mult(e,m,temp);
            div(temp,ke,temp1,e);
            if (compare(three,zero)==0)
            {
                out3(e);
                asm("  IDLE");
                break;
            }
        }
        mult(m,m,temp);
        div(temp,ke,temp1,m);
    }
}
```

Decryption.bat Batch File

```
echo off
echo The SWDS is invoked and the decryption program is run,
echo taking as input the encypted file a:three.dat.
echo The decrypted result is the file a:out.dat.
echo This is then converted back to standard ascii in the file a:out.res.
echo on
swds
ascii
type a:three.dat|more
```

RSADE.C : Decryption Program which uses the Assembly Language Library

```
/* Decryption : d = (e expo kd) mod ke */

main()
{
unsigned int ke[34],kd[34],e[67],d[67],two[2],three[2];
unsigned int zero[2],parity[2],temp[67],temp1[67];

rd1(ke);
rd2(e);
rd0(kd);

three[0]=1;    three[1]=3;
two[0]=1;      two[1]=2;
zero[0]=0;     zero[1]=0;
d[0]=1;             d[1]=1;

for (;;)
    {
        div(kd,two,temp,parity);
        copy(temp,kd);
        if (parity[1]==1)
        {
                mult(d,e,temp);
                div(temp,ke,temp1,d);
                if (compare(kd,zero)==0)
                {
                        out4(d);
                        asm("  IDLE");
                }
        }
        mult(e,e,temp);
        div(temp,ke,temp1,e);
    }
}
```

Ascii.c File Format Conversion Program

```
/*      Dara Murtagh 2/1/1992
        Demo program
        Input hex formatted values from file.
        Output text to another file,

 */

#include <stdio.h>

void strip(name)
char name[];
{ /* strip off filename extension */
    int i;
    for (i=0;name[i]!='\0';i++)
    {
        if (name[i]!='.') continue;
        name[i]='\0';
        break;
    }
}

main()
{
    FILE *ifile;
    FILE *ofile;
    char /*ifname[13],*/ofname[13];
    long ch;
    int count;
    char ifname[13]="a:out.dat";
    strcpy(ofname,ifname);
    strip(ofname);
    strcat(ofname,".res");
    ifile=fopen(ifname,"r");
    printf("Converting file from hex to character format\n");
    ofile=fopen(ofname,"wb");        /* NOT standard C! */
    fscanf(ifile,"%04X",&ch);
    count=(int)ch;
    do
        {
        count--;
        fscanf(ifile,"%04X",&ch);
        fprintf(ofile,"%c",ch);
        }
    while ( (ch!=65535)&&(count>0) );
    fclose(ofile);
    fclose(ifile);
}
```