T125
8

To the Graduate Council:

I am submitting herewith a thesis written by Robert Ewing Gibson entitled "Implementation of a Public-Key Cryptosystem on the VAX 11/780." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

David W. Straight, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Robert E. Gibson

Date July 28, 1986

IMPLEMENTATION OF A PUBLIC-KEY CRYPTOSYSTEM ON THE VAX 11/780

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Robert Ewing Gibson

August 1986

ACKNOWLEDGEMENTS

The author is indebted to Dr. David Straight, chairman and to Dr. Charles Pfleeger and Dr. Kevin O'Kane, committee members, for their assistance and guidance in the preparation of this thesis.

Appreciation is also expressed to Dr. Samuel Wagstaff of Purdue University for his assistance in verifying probable prime numbers.

ABSTRACT

There is a growing need for secure transmission of messages and data via non-secure communication links. Most conventional cryptographic systems require the delivery, usually by couriers, of keys to sending and receiving locations. Public-key cryptosystems partially eliminate this requirement by allowing the sender's key to be made public.

Two public-key cryptosystems were considered in this thesis and one, the RSA system, was implemented on the VAX 11/780 computer. The various program segments were described in detail. Demonstrations of encoding-decoding operations were made and included as an Appendix. Generation of large prime numbers needed by the RSA system was described. Finally, it was concluded that, with minor improvements, the demonstration program could be upgraded to a commercially useful public-key cryptosystem.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1

2. THE KNAPSACK PROBLEM 3

3. THE RSA SYSTEM 8

4. GENERATING PRIME NUMBERS 12
   Introduction 12
   Random Number Generation 13
   Determination of Primality 15
   Distribution of Primes 17

5. PROGRAM MACROS AND PROCEDURE MODULES 19
   Introduction 19
   Large Number Macros 19
   Procedure Modules 23

6. THE MAIN PROGRAM 32

7. USING THE PROGRAM 36

8. SUMMARY AND GROWTH FACTORS 39

LIST OF REFERENCES 41

APPENDIXES 44
   Appendix A. Large Number Macros 45
   Appendix B. Procedure Modules 57
   Appendix C. Main Program 67
   Appendix D. Sample Program Run 74

VITA 77

CHAPTER 1

INTRODUCTION

The world is experiencing an enormous growth in the area of electronic data storage and transmission. Electronic banking--including automated teller machines--nationwide credit bureaus, medical and criminal data banks, and computer networks such as BITNET, ARPANET and the like are expanding rapidly. Military uses are growing apace. Most large data banks are accessible through telephone lines, and a very large part of modern electronic communication is done via satellites or microwave relays. Accessible data banks can be given some protection against intrusion by the integrity and built-in safeguards of a system. In and by themselves satellites and microwave communications offer very little protection against interception, and even hard-wired ground links can be tapped into. Diplomatic channels as well as military aircraft, ships, and front-line posts in wartime must all rely on insecure communication media. As a consequence, since so much of today's communication is vulnerable to interception and through interception to misuse, there is a greatly increasing need to protect such communication through the use of encryption. This thesis will examine the implementation of one cryptographic system that can enhance the security of both stored and transmitted data.

A cryptographic system involves the use of an algorithm or a cipher which operates on a plaintext message to generate a ciphertext which may then be transmitted to its destination via an insecure channel. At the receiving end, an inverse operation is performed on the ciphertext to recover

the intended plaintext message. Most cryptosystems now in use depend on the secure delivery of some set of parameters known as the key. Since these cryptosystems are useless if the key is revealed, the keys are kept secure by having couriers carry the keys to remote locations. Keys should be changed frequently and many receiving locations may be involved; thus the provision of secure keys can be a major expense item. Moreover, since in most encryption systems an identical key is in use at both the sending and receiving end, a dishonest receiving clerk might generate an outgoing message for his or her own benefit such as "Transfer $100,000 to this account." (1).

In 1976 Diffie and Hellman (2) proposed the idea of a public-key cryptosystem. In such a system separate enciphering and deciphering keys are used. These keys are generated in inverse pairs, but it must be infeasible to compute the deciphering key given the enciphering key or vice-versa. Thus the enciphering key for each user can be made public, perhaps being published in a directory. The deciphering key must be kept secure, but it is required only at the message receiving point. No transmission of keys between each transmit - receive station pair is needed. A dishonest clerk can decode incoming messages, but lacking an enciphering key, cannot send messages out.

Two public-key cryptosystems are described in the following chapters; the remaining portion of the thesis describes the implementation of one of these cryptosystems on the VAX 11/780 computer at The Computer Science Department of The University of Tennessee.

CHAPTER 2

THE KNAPSACK PROBLEM

Merkle and Hellman in 1978 (3) proposed a public-key cryptosystem based on the knapsack or subset sum problem. This system will be described and a small-scale example will be presented.

Suppose one has a cylindrical knapsack of length L and a set of n rods having the same diameter as the knapsack, but of varying lengths $a_1, a_2, \cdots, a_n$. It is required to find if there is a subset of the rods that will exactly fill the knapsack.

One may represent the set of n rods by A, an n-dimensional vector of integers, which is the public encryption key. For an example (4) with $n = 8$ let

$$A = (8022,6463,7587,7986,64,8005,6592,3893).$$

Suppose the ASCII character K is to be encoded. Represent K by its binary code in the form of a vector X:

$$X = (0,0,1,0,1,0,1,1).$$

Obtain the dot product Y:

$$Y = A \bullet X = \sum_{i=1}^{n} a_i * x_i.$$

For this example:

$$Y = (7587 + 64 + 6592 + 3893) = 18136.$$

Integer Y is the encrypted form of the plaintext character K and is transmitted to the intended receiver.

At this point an interceptor might try to obtain X from Y by an exhaustive search. The interceptor does have available A, the public

encryption key. Vector X may assume 256 values so assuming random generation of X one would expect on the average that 128 values of X must be tried before a particular subset of A is discovered that sums to Y. The value $n = 8$ is unrealistically small for a working cryptosystem; if $n = 20$ there are approximately one million possible values for X and for $n = 30$, X may have about one billion values.

The encoding method shown is an example of a one-way function. Given an n-dimensional encoding vector A and an n-bit plaintext message X, the encoded message Y is easily computed; in fact the computation simply involves n or fewer additions. However, if one is given only A and Y, the original message X is very difficult to compute; the complexity is proportional to 2^n. For reasonably large values of n trying to obtain the plaintext message by exhaustive search involves such a vast number of calculations that it is computationally infeasible.

The legitimate receiver of an encoded message should of course be able to recover the plaintext with a reasonable amount of computation. Thus, additional information must be available only to the receiver. Decoding messages and recovering plaintext is computationally infeasible unless the receiver has this additional information, which may be compared to knowledge of a hidden trapdoor. Thus, the complete encoding - decoding scheme may be referred to as a one-way trapdoor system.

The originator of a cryptosystem based on the knapsack problem must initially generate an n-dimensional vector A'. This is the secret decoding vector and it will be given to the legitimate receiver. Vector A' has a special property in that each element is greater than the sum of its preceding

elements. The public encoding vector A is derived from A'. Now if the single number Y is received the original message vector X can be easily calculated. This calculation and the derivation of vector A from A' are shown below.

Continuing the above example (4) let

$$A' = (15,92,108,279,563,1172,2243,4486).$$

The transformation from A' to A requires the proper selection of a modulus and a multiplier, and some modular arithmetic. The modulus, r, must be greater than the sum of all components of A'. The multiplier, t, must be less than r and relatively prime to r. For this example $r = 9291$ and $t = 2393$ are satisfactory choices. Now each element of A' is transformed to an element of A via the relation $a_i = a'_i * t \ (mod\ r)$.

Thus for $i = 1$:

$$a_1 = 15 * 2393 \ (mod\ 9291)$$
$$a_1 = 35895 \ (mod\ 9291)$$
$$a_1 = 8022.$$

The complete transformation yields vector A, the public encryption key previously given.

Another factor, S, the multiplicative inverse of $t \ mod\ r$ must be calculated. (Details of this calculation will be discussed in Chapter 5.) By definition

$$S * t \ (mod\ r) \equiv 1,$$

and for this example $S = 1025$. The triple equality sign ($\equiv$) denotes congruence, or a correspondence of factors relative to some measure or modulus. To be congruent, the difference of two numbers must be evenly divisible by the modulus. For example:

$$14 \equiv 1 (mod\ 13) \text{ and}$$
$$-7 \equiv 15 (mod\ 11).$$

Now the received message $Y = 18136$ must be transformed by the relation:

$$Y' = Y * S \pmod{r}$$
$$Y' = 18136 * 1025 \pmod{9291}$$
$$Y' = 18589400 \pmod{9291}$$
$$Y' = 7400.$$

An algorithm can be stated for recovery of the original message vector X:

```
Y(8) = Y'
For i = 8 down to 2 do begin
    if Y(i) ≥ A'(i) then begin
        X(i) = 1
        Y(i-1) = Y(i) - A'(i) end
    else begin
        X(i) = 0
        Y(i-1) = Y(i) end
end
    if Y(1) = A'(1) then begin
        X(1) = 1
        go to exit end
    if Y(1) = 0 then begin
        X(1) = 0
        go to exit end
    else begin
        go to error stop end.
```

The label exit will be reached on successful completion of the algorithm. If a subset of A' summing to Y' does not exist, label error will be reached.

Restating the decryption vector:

$$A' = (15,92,108,279,563,1172,2243,4486),$$

and applying the algorithm it can be seen that:

$$Y_8 = Y' = 7400$$

$$\begin{array}{llllllll}
X_8 = 1, & \text{since} & Y_8 & (=7400) & > & a_8 & (=4486) \\
X_7 = 1, & \text{since} & Y_7 & (=2914) & > & a_7 & (=2243) \\
X_6 = 0, & \text{since} & Y_6 & (=671) & < & a_6 & (=1172) \\
X_5 = 1, & \text{since} & Y_5 & (=671) & > & a_5 & (=563) \\
X_4 = 0, & \text{since} & Y_4 & (=108) & < & a_4 & (=279) \\
X_3 = 1, & \text{since} & Y_3 & (=108) & = & a_3 & (=108) \\
X_2 = 0, & \text{since} & Y_2 & (=0) & < & a_2 & (=92) \\
X_1 = 0, & \text{since} & Y_1 & (=0) & < & a_1 & (=15)
\end{array}$$

The original vector $X = (0,0,1,0,1,0,1,1)$ has been found and it is the ASCII character K.

To review, a decryption vector A', a multiplier t, and a modulus r are chosen and kept secret. The multiplicative inverse of t *mod* r is S, and this is calculated and kept secret. The encryption vector A is derived from A' and is made public. A binary message vector X is encrypted as $Y = A \bullet X$ and is transmitted as vector Y. At the receiving end Y is decoded using vector A', and the original message is then available.

Algorithms based on super-increasing sequences (sequences in which each element is greater than the sum of its predecessors) such as the trapdoor knapsack algorithm are not secure and have been broken quite easily. If one is given the public encoding vector A it may be possible to generate another decoding vector A'' which has the special property:

$$A''_n > A''_1 + A''_2 + \cdots + A''_{n-1}.$$

Then by performing a transformation on the received message decoding will be possible. There are discussions in the literature on this subject; two appropriate references are Herlestam (5) and Shamir (6).

CHAPTER 3

THE RSA SYSTEM

In 1978 Rivest, Shamir and Adleman designed a public-key cryptosystem (7) based on the difficulty of factoring large composite numbers. This system and its underlying mathematics will be discussed as will the idea of a digital signature.

Calculating the product of a pair of n-digit numbers is of time complexity proportional to n^2. However, if only the product is known, determining the factors is of time complexity proportional to an exponential function of n. Thus product is effectively a one-way function and, given a trap door solution, a workable cryptosystem may be implemented.

The RSA algorithm utilizes the Euler Function of number theory. This function, $\emptyset(m)$ is the number of integers less than m which are relatively prime to m. If m is a prime number, obviously $\emptyset(m) = m-1$.

An extension of the Euler Function (4) when X is relatively prime to r is the congruence

$$X^{\emptyset(r)} \equiv 1 \ (mod\ r).$$

Since both sides of a congruence may be raised to a power or multiplied by a constant, the following may be stated

$$\begin{aligned} X^{m\emptyset(r)} &\equiv 1 \ (mod\ r) \\ X^{m\emptyset(r)+1} &\equiv X \ (mod\ r). \end{aligned}$$

It can be proved (4) that if r is the product of two prime numbers the above congruence holds even if X is not relatively prime to r. Now suppose that X represents a plaintext message to be transmitted and that raising X to the exponent $m\emptyset(r)+1$ represents the total encoding-decoding process. If X does

not exceed $r-1$ there can be no ambiguity, that is, no two messages can encode to the same ciphertext and the plaintext message is recovered. This successive exponentiation is the heart of the RSA system.

To implement such a system one must first select two large prime numbers p and q and form their product r. The factors p and q must remain secret but r, the modulus for exponentiation in both encoding and decoding operations, is made public. The Euler Function, $\emptyset(r)$, is then calculated. It may be shown that $\emptyset(r) = (p-1)(q-1)$ when p and q are prime numbers. This function is the modulus for generating pairs of keys and it must be kept secret.

If the product of a secret key (SK) and a public key (PK) is expressed as

$$SK * PK = m\emptyset(r) + 1$$

it follows by the definition of congruence that an integer m exists such that

$$SK * PK \equiv 1 \ (mod \ \emptyset(r)).$$

Thus SK and PK are multiplicative inverses to the modulus $\emptyset(r)$. If a public key PK is selected, the above congruence is solvable for secret key SK only if $\emptyset(r)$ and PK are relatively prime, that is, their greatest common divisor (GCD) must be 1. For the system implemented in this thesis the public key was chosen to be a prime number larger than the factors p and q. Given a public key its multiplicative inverse can be calculated and the method will be discussed in Chapter 5.

To summarize the RSA system, a public directory may contain r, the product of two large (secret) primes p and q and PK, the public encryption key.

The sender computes:

$$C = X^{PK} (mod\ r)$$

where X (the plaintext message) does not exceed $r-1$ and C is the cipher text to be transmitted. Should X equal or exceed the modulus the encryption is not unique; various plaintexts may produce identical ciphertext. In such a case the plaintext must be broken into shorter segments before encryption. The receiver will possess the secret key, SK, and may compute:

$$X = C^{SK} (mod\ r)$$

which recovers the original message.

Exponentiation operations are commutative and this fact allows the sender of a message via the RSA system to create an electronic signature (7). Such a signature allows a receiver to prove to some third party that a particular sender originated a message and that the message has not been changed.

Denote sender and receiver secret and public keys as SK_S, SK_R, PK_S and PK_R respectively. Now sender can encode some message Y in this manner:

$$B = Y^{SK_S} (mod\ r).$$

Since only the sender can encrypt using the secret key normally used for decryption, message B can be considered the sender's signature. Then a second encryption can be done using the receiver's public key:

$$C = B^{PK_R} (mod\ r)$$

and C is transmitted. The receiver first uses his own secret key to obtain B:

$$B = C^{SK_R} (mod\ r).$$

The receiver must have prior knowledge of the presumed sender's public key in order to obtain the original message:

$$Y = B^{PK_S} \ (mod \ r).$$

Now the receiver has decoded signature B using the sender's public key normally used for encryption. Obviously proof now exists that the sender did in fact transmit message Y and that the receiver did not alter the message for his or her own benefit.

CHAPTER 4

GENERATING PRIME NUMBERS

Introduction

To implement the RSA system three large prime numbers must be selected. The modulus for encoding and decoding operations, r, is the product of primes p and q. Security of the system depends on the time complexity of factoring r and this complexity is an exponential function of the number of digits in r. For a commercially used public key cryptosystem p and q might be 100 digit primes, giving a modulus r of 200 digits. For the system implemented in this thesis it was necessary to choose a smaller maximum number size. Consideration was given to execution time in a multiprogramming environment and to a reasonable output format. A word size of 256 bits or 32 8-bit bytes was selected. This word size approximates 77 decimal digits. The large primes p and q were limited to 31 digits giving a product r of at most 62 digits. Tables of large prime numbers are available (8) and these were extensively used in program development. A few arithmetic operations were implemented in a 512 bit format.

A random number generator was programmed to produce data sets of 31 digit numbers. Only odd numbers were stored in the data sets. Then a probabilistic algorithm was used to select prime numbers from the data sets. Three prime numbers were passed to a final program for use in demonstrating the RSA system.

Random Number Generation

Random numbers have been used in sampling techniques, simulation of physical systems, recreation such as card games and other applications even before the introduction of computers. Early methods of generating random numbers included rolling dice, dealing cards, and selecting digits from census records or telephone books. In 1939 a table of 100,000 random digits was mechanically generated by a machine developed by Kendal and Babington-Smith (9).

With the development of computers, methods were developed for deterministically generating sequences of numbers. Since a deterministic procedure using identical input parameters will always generate an identical number sequence, the procedure is not truly random. Often such sequences are referred to as quasi-random or pseudo-random. The user of these sequences may change the input parameters at will, and the generated numbers have proved satisfactory in many applications.

Knuth (10) discusses some rather complex procedures for number generation which in actual use prove unsatisfactory. The most satisfactory procedure is the linear congruential method developed in 1948 by D.H. Lehmer (11). Four parameters must be chosen:

X_0, the initial value or sum; $X_0 \geq 0$.

a, the multiplier; $a \geq 0$.

c, the increment; $c \geq 0$.

m, the modulus; $m > X_0$, $m > a$, $m > c$.

Each generated number is defined in terms of its predecessor thus:

$$X_{n+1} = (aX_n + c) \ mod \ m.$$

Careful selection of parameters is necessary if satisfactory number sequences are to be generated. Since the size of generated numbers is finite, a repeating cycle will always occur. This is called the period, and it is desirable to have the longest possible period. Obviously the period cannot exceed the modulus. For the number generator implemented the modulus was chosen to be:

$$m = 2^{100} - 1,$$

or in hexadecimal notation:

$$\text{m} = \text{00FFFFFFFFFFFFFFFFFFFFFFFF},$$

or in decimal notation:

$$\text{m} = 1267650500228229401496703205375.$$

Prime factors of the modulus are necessary to select a, the multiplier, and these factors are tabulated for numbers of the form $2^n \pm 1$ in reference (8). Also numbers of this form are easily entered as program constants. Since m contains 31 decimal digits, no generated random number can exceed this length. Knuth (10) states and proves a theorem to guarantee that a linear congruential sequence has a maximum period of length m if:

1) c, the increment, is relatively prime to m

2) $a-1$ is a multiple of every prime which divides m

3) $a-1$ is a multiple of 4 if m is a multiple of 4.

From (8) the prime factorization of m is found to be:

$$3 \cdot 5 \cdot 5 \cdot 5 \cdot 11 \cdot 31 \cdot 41 \cdot 101 \cdot 251 \cdot 601 \cdot 1301 \cdot 4051 \cdot 8121 \cdot 268501$$

Obviously, part 3) of the theorem does not apply, and by eliminating two duplicate factors a is found to be $m/25 + 1$:

$$\text{a} = 507060240091291760598681232 16,$$

or in hexadecimal notation:

$$a = 00A3D70A3D70A3D70A3D70A3D8.$$

The increment or additive constant c can be selected rather arbitrarily so long as it is both less than and relatively prime to m. From a table of thirty digit primes (8) c was chosen to be:

$$c = 123376132205208335762278423601,$$

or in hexadecimal notation:

$$c = 019043EF56CCA6EA8FDA319C31.$$

The seed number or starting value X_0 can also be selected arbitrarily, especially in this case where values for m, a and c guarantee the maximum period length. X_0 was chosen to be 1. The values for m, a, c and X_0 were entered in the program as 32-byte hexadecimal constants.

Determination of Primality

Deciding whether a given number is prime or composite has interested mathematicians for centuries. The most straightforward method is the trial division algorithm where n, the number under test, is divided by each number (or each prime number) up to $\sqrt{n}$. If all divisions produce non-zero remainders the number is prime. If n is composite, the factorization will have been discovered. Unfortunately, the complexity of this algorithm is an exponential function of the number of digits in n and so, for large n the running time is prohibitive. For example if n is a 40 digit prime and division time is one microsecond, the primality test will require about one million years (12).

Prime numbers may be selected by a sieve process attributed to

Eratosthenes. All numbers from 2 to x are entered in a list. The first number is encircled and 2 is in fact the first prime. Now every second number is marked in the list. The first unmarked number is now 3 and this is of course the second prime. Every third number is marked and the process continues until the circled number exceeds $\sqrt{x}$. Then every unmarked number may be circled and the list of primes is complete. This process is efficient, and computing time for a single x is proportional to $\log(\log x)$. However for large numbers the sieve method becomes impractical. List space must be provided for every number less than the one to be tested, and every number of the group must be considered (12).

For numbers large enough to be useful as parameters in a public key cryptosystem, absolute methods of determining primality are computationally infeasible. Algorithms which are probabilistic must be considered. One definition of a probabilistic algorithm (12) is that an infinite running time--stopped only by some pre-set limit--may be involved, in which case the conclusion is true with a certain determinable probability. However, if the algorithm terminates prior to the pre-set limit, the conclusion is absolutely true. For primality testing algorithms, the absolute conclusion is that the number is composite, and the probability conclusion is that the number is prime.

Rivest, Shamir and Adleman in their original paper (7) recommend a probabilistic algorithm due to Solovay and Strassen (13) as do several other writers on public key cryptosystems. Solovay and Strassen's algorithm determines if an odd integer n is prime through the repeated selection of a random number a from the set $S = \{1,2,\ldots,n-1\}$. Two functions of a and n are

calculated; the greatest common divisor gcd(a,n) and the Jacobi function $J(a,n)$. If the gcd is not equal to 1, n is obviously composite. If gcd $= 1$ the following congruence is tested:

$$J(a,n) \equiv a^{(n-1)/2} \ (mod\ n).$$

Failure of the congruence means that n is composite and the algorithm terminates. If the congruence is true n is taken to be prime and the probability of error is $\leq \frac{1}{2}$. Solovay and Strassen show a proof of the error probability in their paper (13).

Now a new value of a is selected from S, and the algorithm repeats. Each successful repetition halves the probability of error in asserting n to be prime. Thus after 10 cycles there is $\frac{1}{1024}$ chance that n is actually composite. The primality of n can be determined to any desired level of confidence but since a probabilistic scheme is in use an absolute determination is not possible.

Distribution of Primes

Generation of large primes by testing each entry in a table of large random numbers is practical only if the ratio of primes to nonprimes is high enough. In Ore (14) the number of primes in each interval of 100 numbers is shown for four starting points:

Starting Point	End of Interval									
	100	200	300	400	500	600	700	800	900	1000
1	25	21	16	16	17	14	16	14	15	14
10^6+1	6	10	8	8	7	7	10	6	5	8
10^7+1	2	6	6	6	5	4	7	10	9	6
$10^{12}+1$	4	6	2	4	2	4	3	5	1	6

It can be seen that the density of primes decreases as number size increases; however there is an infinity of prime numbers. For large x it can be shown (4) that approximately $ln(x)$ random numbers must be tested before a prime is found. For the 31 decimal digit primes desired $ln(10^{31}) \sim 67$ numbers will on the average, produce a prime number. Since only odd random numbers are considered, about every 34 numbers should yield a prime.

CHAPTER 5

PROGRAM MACROS AND PROCEDURE MODULES

Introduction

In implementing a public-key cryptosystem on the VAX 11/780 consideration was given to some high-level language routines for handling large numbers, especially the FORTRAN Multiple Precision Arithmetic package. However it was desired to have a program which could be enlarged to handle 100 to 200 digit numbers in an actual working cryptosystem. VAX-11 MACRO assembly language was chosen because of its flexibility and the relative ease with which large integer procedures can be expanded. Two useful instrutions ADWC (add with carry) and SBWC (subtract with carry) allow arithmetic routines of any desired precision to be implemented. The IRP (indefinite repeat block) macro directive can be used to generate multiple address offsets when programming such routines.

An aribitrary precision desk calculator program called BC is available under the UNIX operating system. BC provides the four arithmetic operations, exponentiation, remainder division and radix conversions. Operand size is limited only by the user's memory allocation. This routine was used extensively during program development to verify accuracy of the various arithmetic modules.

Large Number Macros

Integer routines are provided for extended (32-byte) and double extended (64-byte) operands. All arithmetic operations are carried out in

procedures with stacked operand addresses. Procedure calling macros are used; these permit simple extended instructions to be used in the main program. The extended instructions are very similar to the standard VAX instruction set with the mnemonic "x" indicating a 32-byte operation and "xx" indicating a 64-byte operation. Thus, subx3 is three operand 32-byte subtraction and addxx2 is two operand 64-byte addition. In general the 64-byte instructions are provided only if they are needed within another instruction; for example, extended multiplication requires the use of double extended addition.

Appendix A contains the procedure calling macros and several free-standing macros used in shifting and number conversion. Six macros (addx2, addx3, addxx2, subx2, subx3, and subxx2) call procedures for all needed addition and subtraction operations. It should be noted that the same procedure is called for both two- and three-operand instructions. The number-of-arguments operand on the CALLS instruction is used by the procedure to determine proper placement of the result. Five macros (mulx2, mulxx2, mulx3, divx4 and divxx4) provide procedure calls for multiplication and division operations. The mulxx2 and divxx4 macros are not full 64-byte instructions; however they do allow a double extended products and dividends respectively. The called multiply procedure, extmul, always generates a 64-byte product. If #1 has been pushed on the stack as in mulx2 and mulx3, the high order 32 product bytes are compared to zero and, if unequal, an error message indicates product overflow. If #2 has been pushed on the stack as in mulxx2, a 64-byte product is accepted. In a similar manner the dividend size is indicated to procedure extdiv by pushing #1 or #2 on the stack.

Two mod or remainder division macros (modx2 and modxx2) are simply special applications of divx4 and divxx4 in which the dividend is replaced by the remainder after division by the modulus. Since the movc3 instruction is used, registers r2, r3, r4 and r5 must be saved on the stack and restored. Macros expx3 and invx2 generate calls to procedures which perform exponentiation to a modulus and multiplicative inversion to a modulus, respectively.

Two macros (extclr and dexclr) set to zero 32- or 64-byte storage areas. They use the CLRQ instruction, and an indefinite repeat block (IRP) supplies address offsets. The dexclr macro uses extclr for the low order 32 bytes of its operand. A two's complement inversion macro (negate) is available for either 32- or 64-byte operands. An exclusive-or process is used with an extended addition providing the add-one correction.

There are four shifting macros, each providing a one-bit logical shift left or right for extended or double extended operands. These are extlls, dexlls, extlrs and dexlrs. Left shifts use the quadword instruction ASHQ beginning with the high order quadword. Bit test, branch and increment instructions are used to shift in the high bit from the next lower quadword. Right shifts also use ASHQ beginning with the low order quadword. The insert bit field instruction INSV brings in the low bit directly from the next higher quadword. Both of the double extended shifting macros use their 32-byte counterparts for a portion of their work.

Several macros are needed for converting ASCII numeric strings to 32-byte binary representation and vice-versa. Macros incmnt and double operate on a 32-byte or 64-digit packed decimal operands to increment or

double such numbers in place. They are used only by the macro genpkd which performs a complete binary to packed decimal conversion. Incmnt examines each four bit numeric, right to left. Anytime a numeric not equal to nine is found, it is incremented and the process terminates. If nine occurs it is set to zero and the next higher-order numeric is examined. The double macro uses the translation instruction MOVTC in a table lookup operation. Agrument for the table is a byte containing a four bit numeric and a four bit carry-in indicator which is zero or one. Ouptut from the table is a byte containing a four bit numeric representing the doubled digit and a carry-out indicator. The operation proceeds from right to left until 64 packed decimal digits have been doubled. Numerous bit-clearing, shifting and convert instructions are required to accomplish the operation.

Macro genpkd converts a 32-byte binary number to a 64-digit pseudo packed decimal representation. Unlike a true packed decimal there is no sign indicator in the low order byte; every byte contain two digits. Genpkd begins with a destination string set to all zeros and proceeds to double and increment this string with increment selected only when a one bit occurs in the binary source string. Source bits are inspected from left to right in a looping operation and when all have been tested the packed decimal is complete in the destination string.

Macro xhexo converts a 64-digit packed decimal number to a 64-byte string containing ASCII characters in the range 0...F. Four bit numerics in the range 0...9 can be converted to their ASCII form by adding hex 30. If a numeric happens to be in the range 10...15 further addition of 7 produces an ASCII A...F. One or both of these additions will convert any four bit numeric

to character form. The xhexo macro then prints its output string using the I/O macro printchrs.

Macro iread obtains an ASCII input string from any of four sources; the terminal, or the files data.dat, rand.num, or jaco.num. The respective source arguments for iread are line, rcrd, rand and jaco. On any read operation iread receives the input string length in r0. The ASCII string is converted to a 32-byte binary representation by a conversion, two bit left shift, and adding process proceeding right to left from the high order ASCII digit. Conversion is based on the fact that an ASCII character byte minus decimal 48 equals the corresponding four bit numeric. As lower order digits are converted they may be added to previous digits which have been left-shifted by two bits to develop a complete binary number. The operation is controlled by a SOBGTR loop having input length as a parameter.

The macro ls2copy, used by iread, produces a copy of a 32-byte source which is left-shifted by two bits. Its uses the insert field instruction INSV in an indefinite repeat loop with address offsets as repeat block arguments.

The itime macro produces an output from the system clock converted to a character string. Macro ring sounds a bell when the program is running on a terminal so equipped. They were used during program testing to monitor algorithm performance and to signal checkpoints.

Procedure Modules

In Appendix B are the procedures for all arithmetic operations, procedures for the Jacobi function, and a random number generator. The

procedures are called by the CALLS instruction and, generally, addresses of the large operands are placed on the stack by a calling macro.

PSECT ADDSUB contains four procedures to perform extended and double extended additions and subtractions. Entry points are EXTADD, DEXADD, EXTSUB and DEXSUB. These procedures are very similar; each uses the ADWC (add with carry) or SBWC (subtract with carry) instruction in a repeat loop with address offsets as repeat block arguments. The movc3 instruction is used to move operands so the register save mask includes r2 through r5. EXTADD and EXTSUB may be called by two-operand or three-operand macros. The number of operands is available at the stack pointer and this is used to determine proper placement of the result. DEXADD and DEXSUB are called only by two-operand macros. All the arithmetic operations are performed in 64-byte local storage areas OP1 and OP2.

PSECT MULDIV contains procedures for multiplication, division, exponentiation, and multiplicative inversion. Entry points are EXTMUL, EXTDIV, EXTEXP and EXTINV. Each uses movc3 instructions and r6 so the register save mask includes r2 through r6. Since multiply and divide are not in use at the same time some overlays are used in their operand storage areas.

Procedure EXTMUL operates correctly on positive factors only so, initially, operand signs are tested and factors are complemented if needed. The byte mpysgn is inverted with each complement operation and upon multiply completion the product is complemented if needed. Actual multiplication is simple and is performed by selectively adding the multiplicand to the product controlled by bits of the multiplier. Macro

DEXLLS and the ASHL instruction accomplish shifting of the multiplicand and multiplier. Two nested ACBB loops provide control of the operation over eight multipler longwords of 32 bits each. As in ADDSUB procedures the number at the stack pointer determines if two or three operands are involved and allows proper placement of the result.

Procedure EXTDIV implements a non-restoring division algorithm for 32-byte operands although the dividend may be 64 bytes if called by divxx4. Negative divide factors do not occur in the demonstration program and EXTDIV makes no provision for such factors. The dividend--right adjusted if it is only 32 bytes--is placed in a 64-byte storage area labeled prtrem (partial remainder). The left adjusted divisor is placed in a holding area labeled divsha. In the division process the divisor is added to or subtracted from the partial remainder depending on the current sign of prtrem. If prtrem is positive, subtraction occurs and one is placed in the low-order quotient bit. If prtrem is negative, addition occurs and zero is placed in the quotient. With each step the divisor is shifted right with respect to the partial remiander and the quotient is shifted left as its low-order bits are moved into place. Looping occurs between labels loop and foot controlled by an ACBL instruction until the divisor has been shifted its full 256 bit length. Now the final sign of the partial remainder is checked. If positive, the low-order quotient bit is set to one. If negative, the quotient is correct but the divisor must be added to the partial remainder to produce the final remainder. Quotient and remainder are moved to their designated storage areas and the division is complete.

Procedure EXTEXP, a three-operand routine, raises a given number

to an exponent, modulo a third number. The exponentiation is accomplished by repeated squaring and multiplying using an algorithm outlined in reference (15). The exponent's highest order one bit is first located in a test, left shift and loop operation beginning at label extst. Exlim, limit for an ACBL instruction, is initialized at 255 and decremented with each bit test cycle. A storage area for the result, ans, is initialized to one. Upon leaving the test loop and entering the main ACBL loop at label exlp the intermediate result is squared and reduced by the given modulus. Here 64-byte factors may occur, so the double-extended macro instructions mulxx2 and modxx2 are used. Next, the high-order exponent bit is tested and, if a one is found, the intermediate result is multiplied by the original number and reduced by the modulus. The exponent is shifted left and the loop to label exlp repeats until all exponent bits have been inspected. The operation is now complete and the original number is replaced by the result.

Procedure EXTINV computes the multiplicative inverse of a given number to a modulus. This is perhaps the most complex of the procedure modules and it will be discussed in some detail. Recall from Chapter 3 that some public key PK has been selected and the Euler Function $\emptyset(r)$ has been calculated. The secret key and public key are related by the congruence $SK * PK \equiv 1 \ (mod \ \emptyset(r))$ and PK must be relatively prime to $\emptyset(r)$ if SK is to be calculated. In the demonstration program PK is selected from a data set of prime numbers and is of course relatively prime to $\emptyset(r)$. However, the procedure verifies this fact by calculating the greatest common divisor (GCD) of these two factors.

Euclid's algorithm dating from 300 BC states that in a division

operation with non-zero remainder the GCD of dividend and divisor equals the GCD of divisor and remainder. Thus the GCD calculation is implemented by successive divisions in which the previous divisor becomes the current dividend and the previous remainder becomes the current divisor. For an example (4) suppose that $\emptyset(r) = 2760$ and a secret key $SK = 167$ has been selected. The initial dividend/divisor pair is 2760/167 and these are referred to as r_0 and r_1 respectively. The following remainder equations are calculated:

2760	$=$	$167 \bullet 16 + 88,$	so $r_2 = 88$	and	$(2760,167)$	$=$	$(167,88)$
167	$=$	$88 \bullet 1 + 79,$	so $r_3 = 79$	and	$(167,88)$	$=$	$(88,79)$
88	$=$	$79 \bullet 1 + 9,$	so $r_4 = 9$	and	$(88,79)$	$=$	$(79,9)$
79	$=$	$9 \bullet 8 + 7,$	so $r_5 = 7$	and	$(79,9)$	$=$	$(9,7)$
9	$=$	$7 \bullet 1 + 2,$	so $r_6 = 2$	and	$(9,7)$	$=$	$(7,2)$
7	$=$	$2 \bullet 3 + 1,$	so $r_7 = 1$	and	$(7,2)$	$=$	$(2,1)$
2	$=$	$1 \bullet 2 + 0,$	so $r_8 = 0.$				

The last non-zero remainder, r_7, is 1 and this is GCD(2760,167). This algorithm is fast and requires relatively few steps even for large arguments. Now the multiplicative inverse of 167 and 2764 may be calculated by expressing each remainder as a function of r_0 and r_1:

$$r_2 = r_0 - 16r_1$$
$$r_3 = r_1 - r_2 = r_1 - (r_0 - 16r_1)$$
$$r_3 = -r_0 + 17r_1$$
$$r_4 = r_2 - r_3 = (r_0 - 16r_1) - (-r_0 + 17r_1)$$
$$r_4 = 2r_0 - 33r_1$$
$$r_5 = r_3 - 8r_4 = (-r_0 + 17r_1) - 8(2r_0 - 33r_1)$$
$$r_5 = -17r_0 + 281r_1$$
$$r_6 = r_4 - r_5 = (2r_0 - 33r_1) - (-17r_0 + 281r_1)$$
$$r_6 = 19r_0 - 314r_1$$
$$r_7 = r_5 - 3r_6 = (-17r_0 + 281r_1) - 3(19r_0 - 314r_1)$$
$$r_7 = -74r_0 + 1223r_1.$$

Substituting for r_7, r_0 and r_1 gives the equality:

$$1 = (-74 * 2760) + (1223 * 167).$$

Since the modulus of interest is 2760 a congruence may be stated:

$$1 \equiv 1223 * 167 \ (mod\ 2760)$$

and the multiplicative inverse of 167 modulo 2760 is 1223.

The GCD determination in EXTINV is similar to that illustrated in the above example. After each divide operation the 32-byte quotient is pushed on the user stack and the remainder is compared to one. If equal, the loop beginning at label lp1 is exited, a message indicating GCD=1 is printed and calculation of the multiplicative inverse is begun. Each remainder is also compared to zero. If zero remainder occurs before one remainder the GCD is not 1 and an error condition exists. In this case an error message is printed and a branch is made to the global label errstp on the main program exit.

At the beginning of EXTINV four initial coefficients C0R0=1, C1R0=0, C0R1=0, and C1R1=1 are placed in storage areas. These factors result from the trivial equations:

$$r_0 = 1 * r_0 + 0 * r_1,$$

$$r_1 = 0 * r_0 + 1 * r_1.$$

The procedure then uses the stacked quotients to calculate coefficients of r_0 and r_1 in each remainder equation as shown in the example. The quotients must be popped from the stack in a first-in, first-out manner. Before the first quotient is pushed in the GCD calculation the stack pointer is stored in r6, so this quotient may be recalled by using the autodecrement mode address -(r6). The user stack in the VAX computer builds toward lower memory addresses.

The coefficients are calculated in a loop beginning at label lp2 and controlled by an ACBW instruction whose limit is dvcnt, the number of

divide operations which occurred in the GCD determination. When this loop is exited, coefficient C2R1--if positive--contains the desired inverse. If C2R1 is negative it must be adjusted by adding the modulus. C2R1 then replaces the original number and the process is complete.

PSECT JACOBI contains two procedures used in calculating the Jacobi symbol. Entry points are JACO1 and JACO2. The Jacobi symbol $J(a,b)$ is defined for two integers a and b if b is odd, $a \leq b$, and $GCD(a,b)=1$.

Procedure JACO1 checks arguments a and b to ensure that these three conditions are met. A binary number is even if its low order bit is zero. So the BLBS (branch if low bit set) instruction tests argument b. Next, a is subtracted from b and the sign of their difference is tested. Finally $GCD(a,b)$ is calculated by successive divisions in a looping routine similar to that used in EXTINV. When zero remainder occurs, the previous remainder is compared to one. If equal, then the GCD is 1. Failure of any of these three tests indicates an error condition and the Jacobi symbol cannot be calculated. In this case an appropriate error message is printed, a flag byte jcflg is set to hex FF, and a return is made to the calling program. If the argument tests are successful, their addresses are pushed on the stack along with the address of longword jcans, a storage location for the result.

A three-operand call is then made to JACO2, the procedure which determines the Jacobi symbol. Upon return from JACO2 the flag byte is set to hex 00 and a final return is made to the calling program.

$J(a,b)$ is in the set $\{1,-1\}$ and a simple recursive algorithm may be used (4). If $a=1$ then:

$$J(a,b) = 1.$$

Otherwise, if a is even:

$$J(a,b) = J(a/2,b) * (-1)^{(b^2-1)/8},$$

and if a is odd:

$$J(a,b) = J(b\,(mod\ a),a) * (-1)^{(a-1)(b-1)/4}.$$

Procedure JACO2 is the only recursive routine used and it simply implements the three equations stated above. Argument a is immediately compared to 1 and, if equal, 1 is moved into the result longword and a return is made to JACO1. If a is not 1, an odd-even determination is made and the appropriate exponent of -1 is calculated. Labels aodd and aeven identify the respective program segments. New arguments for a recursive call are also calculated and placed in locations arga and argb. The exponent of -1 must be tested as odd-even and this is done at label exptst. A longword containing +1 or -1 is pushed on the stack at this point; this is the multiplier for the Jacobi symbol upon return from recursive calls. Addresses for arga, argb and jcans are pushed on the stack at label jct2 and the recursive call to JACO2 is made. Upon return, the result, jcans, is multiplied by the stack-top factor which is always +1 or -1. The product is the current Jacobi symbol and a return is made to the calling procedure. Storing data on the stack in a recursive call situation is made possible by the VAX procedure-handling convention. Upon returning to any particular level of recursion the stack pointer is restored to its value when that level was originally entered.

PSECT RANDOM contains a random number generator which is the single procedure NUMGEN. This procedure implements the mixed linear congruential method for number generation which was discussed in Chapter 4.

Five arguments are involved. Four of these pass addresses of 32-byte parameters and one argument contains the list size, or number of random numbers to be generated. Local storage areas for the 32-byte parameters are labeled pmod, a, c and seed. These represent the modulus, multiplier, additive constant and initial number, respectively. The storage area labeled size receives the list size directly from the stack.

After moving the five arguments to local areas, an ACBW loop is entered at label rnd1. Numbers are generated, converted to the pseudo packed decimal format, and written to the file jaco.num. The low-order bit of each number is tested, and if one, the number is odd and is a candidate for prime testing. In this case the number is also written to the file rand.num. The process continues under control of the ACBW instruction until the specified list size is reached. At this point file rand.num contains list-size entries from the set of odd integers and, if the numbers generated are truly random, file jaco.num contains about twice as many entries from the complete set of integers. Work of the procedure is now complete and return is made to the calling program.

CHAPTER 6

THE MAIN PROGRAM

PSECT PUBKEY, reproduced in Appendix C, contains the main program for demonstration of the RSA public-key cryptosystem. The program may be divided into four main tasks: 1) generation of a set of large random numbers, 2) selection of prime numbers from the set, 3) encoding a sample message, and 4) decoding the message to retrieve the plaintext.

The program begins with generation of the files jaco.num and rand.num by a call to Procedure NUMGEN. Selection of parameters for NUMGEN was described in Chapter 4. These are stored as byte strings near the beginning of the PSECT and are labeled rseed, rmpy, rcon and rmod; representing the seed number, multiplier, additive constant and modulus, respectively. A list size of 500 was chosen; thus rand.num contains 500 odd entries and jaco.num contains approximately 1000 entries. Sorting of these files is accomplished by calls to the system sort procedures. Rand.num is sorted in descending order while jaco.num is sorted in ascending order.

The algorithm of Solovay and Strassen (13) is used to select prime numbers from the file rand.num. Evaluating the Jacobi symbol for the prime number candidate is the first step. Recall from Chapter 4 that the prime candidate (read from rand.num) is the second Jacobi argument and a random number less than or equal to the candidate (read from jaco.num) is the first argument. At label loop1 file jaco.num is reset to its first record by a close-open-connect sequence. Arguments are read into storage areas num1 and num2 and the Jacobi symbol determination is initiated by the calling macro

jacobi. The called procedures JACO1 and JACO2 were described in Chapter 5. Should a failure occur in JACO1, a message will be printed by that procedure, and flag byte jcflg will be set to hex FF. Such failures are limited to three cases: 1) second argument not odd, 2) first argument not less than or equal to second argument, and 3) GCD of the two arguments not equal to one. The flag byte is inspected by a BLBS instruction, and if failure is indicated, the program returns to label loop1. If the procedure JACO1 completes without error, recursive procedure JACO2 will compute the Jacobi symbol as +1 or -1. Beginning at label j2 the remainder of the Solovay-Strassen algorithm is implemented. This is simply a test of the congruence:

$$J(a,n) \equiv a^{(n-1)/2} \ (mod\ n).$$

If the test fails, a compound number has been found. A message is printed at label jfail and the program returns to label loop1. If the test succeeds, an ACBB instruction returns control to label loop2. Here another record is read from jaco.num to replace the first Jacobi argument; the second Jacobi argument num2 is still the prime number candidate and the Jacobi symbol determination is begun again. Assuming that the prime candidate is truly prime, the cycle beginning at loop2 will repeat fifteen times under control of the ACBB instruction. Each successful test using the Solovay-Strassen algorithm halves the probability that a compound number will be announced as prime. So with fifteen such tests complete this probability is approximately one in 32,000.

Upon leaving the ACBB loop a branch is made to label j4 where a prime number message is printed. The prime number, now in storage area num2 in binary form, is converted to an ASCII string and written to file

data.dat by the WTRCRD instruction. A branch is made to label loop1 where file jaco.num is re-opened and another prime number candidate is read from file rand.num. Eventually end-of-file is reached in rand.num and a branch to eof2 is made. Assuming that at least three prime numbers were found, the program can proceed to compute secret and public keys for the message encoding step.

Beginning at label eof2, three prime numbers are read from data.dat into storage areas num3, num2 and num1. Prime candidates from rand.num were sorted in descending order; actual primes in num1, num2 and num3 are now in ascending order as desired. These primes are converted to ASCII form and displayed as factors p, q and d. The modulus r and the Euler Function $\emptyset(r)$ are calculated and displayed. These factors are in storage areas num4 and num7. Factor d, largest of the three primes, is copied into num8 and then replaced by factor h, its multiplicative inverse modulo $\emptyset(r)$. Now storage areas num3 and num8 hold factors d and h which are the secret key and public key respectively. The program is ready to encode a plaintext message.

At label more, data entry is requested by a READLINE instruction. Up to eighty characters may be encoded from a single terminal entry. Register r9 indexes input characters in the 80-byte area inline while r10 indexes output cipher strings in the 128-byte area cph1. The encoding operation proceeds twenty characters at a time in a short loop beginning with label more2 and ending with an ACBB instruction. Macro expx3 calls the exponentiation procedure which performs the encoding by raising a message segment to the power h modulo r. Each 20-character ASCII string encodes to

a 32-byte binary number. Another short loop, three lines beginning at label more3, displays the four encoded strings.

The ciphertext strings can now be decoded to retrieve the plaintext message. The procedure is nearly identical to the encoding process. Register r10 indexes input cipher strings in blocks of 32 bytes while r9 indexes output ASCII strings in 20-character groups. A loop beginning at label more4 is controlled by an ACBB instruction. Macro expx3 calls procedure EXTEXP which raises a cipher text number to the power d modulo r. Finally, the 80-character retrieved message is printed and a branch is made to label more where another input string may be entered.

Two error stops in the main program may be reached if inappropriate ends-of-file are encountered. In reading from jaco.num, an end-of-file condition transfers control to label eof3. This is an error situation since the jacobi symbol determination cannot be completed. A message is printed and the program terminates. If insufficient prime numbers have been stored in file data.dat the label eof may be reached. In this situation public and secret keys cannot be generated so the program again terminates.

CHAPTER 7

USING THE PROGRAM

It is entirely feasible to operate the RSA demonstration program exactly as shown in Appendix C and described in Chapter 6. However, to avoid excessive tie-up of video terminals, most program testing was split into a batch phase and an interactive phase. The program was divided at label eof2 where the file of probable prime numbers has just been completed and generation of public and secret keys is about to begin.

The exponentiation procedure EXTEXP is used in encoding, decoding, and prime number selection. This procedure was found to be noticeably slow, requiring about 25 seconds per execution in a single user situation. In testing records from file rand.num assume that a prime number candidate has passed the preliminary tests of procedure JACO1 and the Jacobi symbol (+1 or -1) has been calculated by recursive procedure JACO2. To complete the test of Solovay and Strassen an exponentiation must now be performed. Thus, only about two prime number candidates can be tested per minute. If a candidate is truly prime (at least to the high probability implied by fifteen Solovay-Strassen tests) some seven minutes is required to complete these tests.

It was observed that the great majority of compound numbers were not discovered by JACO1 but by the first trial of the Solovay-Strassen algorithm. In a test run using a data set of fifty-one odd random numbers, one number was announced as a probable prime, 48 numbers were announced as compound on the first trial, and two numbers were announced as

compound after multiple trials of seven and eleven cycles. Theory discussed in Chapter 4 suggests that for 31-digit odd integers about one number in 34 will prove to be prime.

For the final program tests involving 500 odd random numbers a batch-mode run was made on the VAX 11/780 during off-peak hours. Files rand.num and jaco.num were generated and sorted quite rapidly, but checking the 500 numbers for primality required about 4.5 hours CPU time. Wall-clock time was essentially the same. The following fifteen probable primes were discovered and written to file data.dat:

+0800649833179762467645082232209
+0668695629226292243288967667547
+0627555111875248476074334821099
+0595380952378795720582897529113
+0550356404903950844628706632739
+0306488170617335281843707835463
+0281076531963385766925520312783
+0274121252918992709014057419413
+0267696058929796648782680617219
+0256284916310755594236772872521
+0244825584141242085356311846807
+0224399894117137564752655266631
+0165567608809274045764397716339
+0082136466050670171425605849849
+0017402630653985025766305017749

Some verification of the primality testing process was desired. The file data.dat was transmitted via BITNET to Dr. Samuel Wagstaff of the Computer Science Department, Purdue University, Lafayette, Indiana. Dr. Wagstaff checked the fifteen numbers by another probabilistic prime-testing algorithm. In his opinion, the numbers are all truly prime.

Interactive mode testing of the encoding-decoding process was accomplished by starting the program at label eof2. Using three prime

numbers from file data.dat public and secret keys are generated and a continuous loop is entered at label more. Each traversal of this loop requests terminal input of a plain text message, displays the message after encoding, decodes the message and, finally, displays the retried plain text.

Numerous trials were made of the interactive process. Appendix D shows the various message displays for a typical program run. Since timing of the exponential procedure was of concern, a time-mark is printed at each call to EXTEXP.

In the current program the same public and secret keys are used for each successive encoding-decoding process. With a simple program change new keys could generated for each process. Three prime numbers are needed to calculate a pair of keys. Thus the 500-entry file rand.num which yielded fifteen probable primes could support five distinct user pairs.

CHAPTER 8

SUMMARY AND GROWTH FACTORS

The idea of public-key cryptosystems was presented and two such systems--the knapsack system and the RSA system--were discussed in detail. For the knapsack system, a simple example was given. Since large prime numbers are needed for the RSA system, their generation was discussed in Chapter 4. A VAX-11 MACRO assembler language program to implement the RSA system was developed and tested. Chapters 5 and 6 described the program and Chapter 7 discussed program operation. Changes needed to convert the program to a commercially useful encryption system will now be considered.

The basic representation of large integers may be readily changed from 256 bits to 512 bits. The equivalent decimal representation then will change from 77 digits to 153 digits. This revision would mainly involve changing the procedure modules found in Appendix B. For example, in Procedure EXTADD the first operand of each movc3 instruction would become #64, and the repeat block arguments on the irp instruction would be extended to include byte offsets 32 through 63. Similar changes throughout the procedures and the large number macros will allow full 512-bit or 64-byte operations. Parameters for Procedure NUMGEN may be altered so that the generated random numbers will be equivalent to 75 decimal digits. Thus, factor r, the product of primes, will be approximately 150 decimal digits.

An investigation should be made of Procedure EXTEXP, since this procedure was observed to be noticeably slow. An algorithm involving

repeated squaring and multiplying was used here; in all probability faster algorithms exist and could be implemented.

Various conversions between binary and ASCII representations were made in the demonstration program. Files rand.num, jaco.num and data.dat were written as ASCII character strings but converted to binary integers when read by macro iread. Little overhead is involved here, but in a commercial encryption system no display would be necessary and all numerics would be handled as binary integers.

In conclusion, it is believed that with minor improvements the demonstration program presented in this thesis could be upgraded to a commercially useful public-key encryption system based on the RSA algorithm.

LIST OF REFERENCES

LIST OF REFERENCES

1. Hellman, Martin E., "The Mathematics of Public-Key Cryptography," *Scientific American,* Vol. 241, No. 2, August 1979, pp. 146-157.

2. Diffie, W. and Hellman, Martin E., "New Directions in Cryptography," *IEEE Transactions on Information Theory,* Vol. IT-22, No. 6, November 1976, pp. 644-654.

3. Merkle, R.C. and Hellman, Martin E., "Hiding Information and Signatures in Trapdoor Knapsacks," *IEEE Transactions on Information Theory,* Vol. IT-24, No. 5, September 1978, pp. 525-530.

4. Meyer, Carl H. and Matyas, S.M. *Cryptography: A New Dimension in Computer Data Security,* John Wiley & Sons, New York, NY, 1982.

5. Herlestam, T., "Critical Remarks on Some Public-Key Cryptosystems," *BIT,* Vol. 18, 1978, pp. 493-496.

6. Shamir, A., "A Polynomial Time Algorithm for Breaking Merkle-Hellman Cryptosystems," *Applied Mathematics,* The Weizmann Institute, Rehovst, Israel, April 1982.

7. Rivest, R.L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM,* Vol. 21, No. 2, February 1978, pp. 120-126.

8. Brillhart, John, et. al., "Factorizations," *Volume 22, Contemporary Mathematics,* American Mathematical Society, Providence, RI, 1983.

9. Kendall, M.G. and Babington-Smith, B., *Journal of the Royal Statistical Society,* Series B, Vol. 6, No. 1, 1939, pp. 51-61.

10. Knuth, D.E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* Addison Wesley, Reading, MA, 1969.

11. Lehmer, D.H., *Proceedings of the 2nd Symposium on Large-Scale Computing Machinery,* Harvard University Press, Cambridge, MA, 1951, pp. 141-146.

12. Pomerance, Carl, "Recent Developments in Primality Testing," *Mathematical Intelligencer,* Vol. 3, 1980, pp. 97-104.

13. Solovay, R. and Strassen, V., "A Fast Monte-Carlo Test for Primality," *SIAM Journal on Computing,* vol. 6, March 1977, pp. 84-85.

14. Ore, O. *Number Theory and its History,* McGraw-Hill, New York, NY, 1948.

15. Beker, Henry and Piper, Fred, *Cipher Systems,* John Wiley and Sons, New York, NY, 1982.

APPENDIXES

APPENDIX A

LARGE NUMBER MACROS

```
        .macro  addx2   op1,op2                         ;---------

; TWO OPERAND 32-BYTE ADDITION, OP2 RECEIVES RESULT.

        pushal  op1             ; push two operand addresses
        pushal  op2
        calls   #2,extadd       ; call procedure
        .endm   addx2

        .macro  addx3   op1,op2,sum                     ;---------

; THREE OPERAND 32-BYTE ADDITION, OP3 RECEIVES RESULT.

        pushal  sum             ; push three operand addresses
        pushal  op1
        pushal  op2
        calls   #3,extadd       ; call procedure
        .endm   addx3

        .macro  addxx2  op1,op2                         ;---------

; TWO OPERAND 64-BYTE ADDITION, OP2 RECEIVES RESULT.

        pushal  op1             ; push two operand addresses
        pushal  op2
        calls   #2,dexadd       ; call procedure
        .endm   addxx2

        .macro  subx2   op1,op2                         ;---------

; TWO OPERAND 32-BYTE SUBTRACTION, OP2 RECEIVES RESULT.

        pushal  op1             ; push two operand addresses
        pushal  op2
        calls   #2,extsub       ; call procedure
        .endm   subx2

        .macro  subx3   op1,op2,diff                    ;---------

; THREE OPERAND 32-BYTE SUBTRACTION, OP2 RECEIVES RESULT.

        pushal  diff            ; push two operand addresses
        pushal  op1
        pushal  op2
        calls   #3,extsub       ; call procedure
        .endm   subx3
```

```
        .macro  subxx2  op1,op2                         ;---------

; TWO OPERAND 64-BYTE SUBTRACTION, OP2 RECEIVES RESULT.

        pushal  op1             ; push two operand addresses
        pushal  op2
        calls   #2,dexsub       ; call procedure
        .endm   subxx2

        .macro  mulx2   op1,op2                         ;---------

; TWO OPERAND 32-BYTE MULTIPLICATION, OP2 RECEIVES PRODUCT.
; ERROR MESSAGE FROM PROCEDURE IF PRODUCT EXCEEDS 32 BYTES.

        pushal  op1             ; push two operand addresses
        pushal  op2
        pushl   #1              ; indicate 32-byte product
        calls   #3,extmul       ; call procedure
        .endm   mulx2

        .macro  mulxx2  op1,op2                         ;---------

; TWO OPERAND 64-BYTE MULTIPLICATION,
;  OP2 RECEIVES 64-BYTE PRODUCT.

        pushal  op1             ; push two operand addresses
        pushal  op2
        pushl   #2              ; indicate 64-byte product
        calls   #3,extmul       ; call procedure
        .endm   mulxx2

        .macro  mulx3   op1,op2,prod                    ;---------

; THREE OPERAND 32-BYTE MULTIPLICATION, OP3 RECEIVES 32-BYTE
; PRODUCT. ERROR MESSAGE IF PRODUCT EXCEEDS 32 BYTES.

        pushal  prod            ; push three operand addresses
        pushal  op1
        pushal  op2
        pushl   #1              ; indicate 32-byte product
        calls   #4,extmul       ; call procedure
        .endm   mulx3
```

```
        .macro  divx4   divd,divr,quot,rem                      ;---------

; FOUR OPERAND 32-BYTE DIVISION, OP3 RECEIVES QUOTIENT,
; OP4 RECEIVES REMAINDER.

        pushal  divd            ; push four operand addresses
        pushal  divr
        pushal  quot
        pushal  rem
        pushl   #1              ; indicate 32-byte dividend
        calls   #5,extdiv       ; call procedure
        .endm   divx4

        .macro  divxx4  divd,divr,quot,rem                      ;---------

; FOUR OPERAND 64-BYTE DIVISION, DIVIDEND ONLY IS 64 BYTES.
; OP3 RECEIVES QUOTIENT, OP4 RECEIVES REMAINDER.

        pushal  divd            ; push four operand addresses
        pushal  divr
        pushal  quot
        pushal  rem
        pushl   #2              ; indicate 64-byte dividend
        calls   #5,extdiv       ; call procedure
        .endm   divxx4

        .macro  modx2   nmbr,mod                                ;---------

; TWO OPERAND 32-BYTE MOD FUNCTION,
; OP1 RECEIVES OP1 MOD OP2.

        pushr   #^m<r2,r3,r4,r5>  ; stack registers used by movc3
        divx4   nmbr,mod,quot,rem ; perform 32-byte division
        movc3   #32,rem,nmbr      ; replace nmbr by remainder
        popr    #^m<r2,r3,r4,r5>  ; restore registers
        .endm   modx2

        .macro  modxx2  nmbr,mod                                ;---------

; TWO OPERAND 64-BYTE MOD FUNCTION, OP1 ONLY IS 64 BYTES.
; OP1 RECEIVES OP1 MOD OP2.

        pushr   #^m<r2,r3,r4,r5>  ; stack registers used by movc3
        divxx4  nmbr,mod,quot,rem ; perform 64-byte division
        dexclr  nmbr
        movc3   #32,rem,nmbr      ; replace nmbr by remainder
        popr    #^m<r2,r3,r4,r5>  ; restore registers
        .endm   modxx2
```

```
        .macro  expx3   nmbr,exp.mod                    ;---------

; THREE OPERAND MOD EXPONENTIATION,
;  OP1 RECEIVES OP1 ** OP2 MOD OP3.

        pushal  nmbr            ; push three operand addresses
        pushal  exp
        pushal  mod
        calls   #3,extexp       ; call procedure
        .endm   expx3

        .macro  invx2   nmbr,mod                        ;---------

; TWO OPERAND MULTIPLICATIVE INVERSION,
; OP1 RECEIVES OP1 ** (-1) MOD OP2.

        pushal  nmbr            ; push two operand addresses
        pushal  mod
        calls   #2,extinv       ; call procedure
        .endm   invx2

        .macro  extclr  dest                            ;---------

; 32-BYTE CLEAR FUNCTION.

        .irp    offset,  <0,8,16,24>    ; repeat block args
        clrq    dest+'offset            ; use quadword clear
        .endr
        .endm   extclr

        .macro  dexclr  dest                            ;---------

; 64-BYTE CLEAR FUNCTION.

        extclr  dest                    ; perform 32-byte clear
        .irp    offset, <32,40,48,56>   ; repeat block args
        clrq    dest+'offset            ; use quadword clear
        .endr
        .endm   dexclr

        .macro  jacobi  arga,argb                       ;---------

; JACOBI FUNCTION CALL.

        pushal  arga            ; push two operand addresses
        pushal  argb
        calls   #2,jaco1        ; call procedure
        .endm   jacobi
```

```
        .macro  negate  dest,len                                ;---------

; TWO'S COMPLEMENT INVERSION, ARG MAY BE 32 OR 64-BYTE.

        .irp    offset, <0,4,8,12,16,20,24,28>          ; repeat block args
        xorl    #^xffffffff,dest+'offset                ; ex or inversion
        .endr
        .if     idn     len,64                          ; 64-byte arg
        .irp    offset, <32,36,40,44,48,52,56,60>        ; rept block args
        xorl    #^xffffffff,dest+'offset                ; ex or inversion
        .endr
        .endc
        addl2   #1,dest                                 ; incr one's comp
        .irp    offset, <4,8,12,16,20,24,28>             ; rept block args
        adwc    #0,dest+'offset                         ; add any carry
        .endr
        .if     idn     len,64                          ; 64-byte arg
        .irp    offset, <32,36,40,44,48,52,56,60>        ; rept block args
        adwc    #0,dest+'offset                         ; add any carry
        .endr
        .endc
        .endm   negate

        .macro  extlls  dest,?l1,?l2,?l3                        ;---------

; 32-BYTE LOGICAL LEFT SHIFT FUNCTION, ONE-BIT SHIFT.

        ashq    #1,dest'+24,dest'+24    ; use quadword shift
        bitb    #^x80,dest'+23          ; test hi bit, next quadword
        beql    l1                      ; branch if zero
        incb    dest'+24                ; increment if one
l1:     ashq    #1,dest'+16,dest'+16    ; repeat for next quadword
        bitb    #^x80,dest'+15
        beql    l2
        incb    dest'+16
l2:     ashq    #1,dest'+8,dest'+8      ; repeat for next quadword
        bitb    #^x80,dest'+7
        beql    l3
        incb    dest'+8
l3:     ashq    #1,dest,dest            ; shift final quadword
        .endm   extlls

        .macro  dexlls  dest,?l1,?l2,?l3,?l4                    ;---------

; 64-BYTE LOGICAL LEFT SHIFT FUNCTION, ONE-BIT SHIFT.

        ashq    #1,dest'+56,dest'+56    ; use quadword shift
        bitb    #^x80,dest'+55          ; test hi bit, next quadword
```

```
        beql    l1                      ; branch if zero
        incb    dest'+56                ; increment if one
l1:     ashq    #1,dest'+48,dest'+48    ; repeat for next quadword
        bitb    #^x80,dest'+47
        beql    l2
        incb    dest'+48
l2:     ashq    #1,dest'+40,dest'+40    ; repeat for next quadword
        bitb    #^x80,dest'+39
        beql    l3
        incb    dest'+40
l3:     ashq    #1,dest'+32,dest'+32    ; repeat for next quadword
        bitb    #^x80,dest'+31          ; test hi bit, next 32 bytes
        beql    l4                      ; branch if zero
        incb    dest'+32                ; increment if one
l4:     extlls  dest                    ; perform 32-byte shift
        .endm   dexlls

        .macro  extlrs  dest                                    ;---------

; 32-BYTE LOGICAL RIGHT SHIFT FUNCTION, ONE-BIT SHIFT.

        ashq    #-1,dest,dest           ; use quadword shift
        insv    dest'+8,#7,#1,dest'+7   ; insert lo bit from next
                                        ; higher quadword
        ashq    #-1,dest'+8,dest'+8
        insv    dest'+16,#7,#1,dest'+15
        ashq    #-1,dest'+16,dest'+16
        insv    dest'+24,#7,#1,dest'+23
        ashq    #-1,dest'+24,dest'+24
        bicb2   #^x80,dest'+31          ; clear highest bit
        .endm   extlrs

        .macro  dexlrs  dest                                    ;---------

; 64-BYTE LOGICAL RIGHT SHIFT FUNCTION, ONE-BIT SHIFT.

        extlrs  dest                    ; perform 32-byte shift
        insv    dest'+32,#7,#1,dest'+31 ; insert lo bit from next
                                        ; higher quadword
        ashq    #-1,dest'+32,dest'+32   ; use quadword shift
        insv    dest'+40,#7,#1,dest'+39
        ashq    #-1,dest'+40,dest'+40
        insv    dest'+48,#7,#1,dest'+47
        ashq    #-1,dest'+48,dest'+48
        insv    dest'+56,#7,#1,dest'+55
        ashq    #-1,dest'+56,dest'+56
        bicb2   #^x80,dest'+63          ; clear highest bit
        .endm   dexlrs
```

```
        .macro  incmnt  dest,?l1,?l2,?l3,?l4                    ;---------

; 32-BYTE PACKED NUMBER INCREMENT.

; r6 holds a working longword
; r7 holds address of destination
; r8 holds index for acbb instr

        pushr   #^m<r6,r7,r8>           ; save registers used
        movab   dest,r7                 ; get dest address
        movb    #1,r8                   ; set index register
l1:     cvtbl   (r7),r6                 ; get a data longword
        bicl2   #^xfffffff0,r6          ; clear all but low nibble
        cmpl    #9,r6                   ; compare to lo nibble of '9'
        beql    l2                      ; branch if '9' was present
        incl    r6                      ; increment data longword
        insv    r6,#0,#4,(r7)           ; insert four bits at dest
        brw     l4                      ; complete, branch to end
l2:     cvtbl   (r7),r6                 ; get a data longword
        bicl2   #^xffffff0f,r6          ; clear all but second nibble
        ashl    #-4,r6,r6               ; right adjust data in r6
        cmpl    #9,r6                   ; compare to lo nibble of '9'
        beql    l3                      ; branch if '9' was present
        incl    r6                      ; increment data longword
        insv    r6,#4,#4,(r7)           ; insert four bits at dest
        brb     l4                      ; complete, branch to end
l3:     bicb2   #^xf0,(r7)+             ; clear second nibble, incr @
        acbb    #32,#1,r8,l1            ; repeat for 32 bytes
l4:     popr    #^m<r6,r7,r8>           ; restore altered registers
        .endm   incmnt

        .macro  double  dest,?l1                                ;---------

; 32-BYTE PACKED NUMBER DOUBLE.

; r6 holds address of destination
; r7 holds carry in
; r8 holds index for acbb loop
; r9 holds left nibble temp

        pushr   #^m<r2,r3,r4,r5,r6,r7,r8,r9> ; save registers
        clrb    r7                      ; clear carry in register
        movb    #1,r8                   ; set index register
        movab   dest,r6                 ; get dest address
l1:     bicb3   #^xf0,(r6),rnbyte       ; get right data nibble
        addb2   r7,rnbyte               ; combine with carry in
        movtc   #1,rnbyte,#0,table,#1,rnbyte ; translate with table
```

```
        bicb3   #^x0f,rnbyte,r7                 ; carry out to r7
        bicb2   #^xf0,rnbyte                    ; clear carry out
        cvtbl   (r6),r9                         ; get same data byte
        bicl2   #^xfffff0f,r9                   ; clear all but left nibble
        ashl    #-4,r9,r9                       ; right adjust nibble in r9
        cvtlb   r9,lnbyte                       ; move left nibble
        addb2   r7,lnbyte                       ; combine with carry in
        movtc   #1,lnbyte,#0,table,#1,lnbyte    ; translate with table
        bicb3   #^x0f,lnbyte,r7                 ; carry out to r7
        bicb2   #^xf0,lnbyte                    ; clear carry out
        cvtbl   lnbyte,r9                       ; left nibble to r9
        ashl    #4,r9,r9                        ; left adjust nibble in r9
        cvtlb   r9,lnbyte                       ; move left nibble
        xorb3   rnbyte,lnbyte,(r6)+             ; combine left,right nibbles
        acbb    #32,#1,r8,l1                    ; repeat for 32 bytes
        popr    #^m<r2,r3,r4,r5,r6,r7,r8,r9>    ; restore registers
        .endm   double

        .macro  genpkd  srce,dest,?l1,?l2,?l3,?l4               ;---------

; 64-DIGIT PSEUDO PACKED DECIMAL FROM 32-BYTE BINARY SOURCE.

; r8 holds index3 for outer acbb loop
; r9 holds index2 for inner acbb loop
; r10 holds successive longwords from binary source
; r11 holds address of binary source

        pushr   #^m<r8,r9,r10,r11>      ; save registers used
        extclr  dest                    ; clear destination area
        clrb    r8                      ; clear index register
        moval   srce+32,r11             ; get @ one above srce high byte
l1:     movl    -(r11),r10              ; r10 receives four source bytes
        clrb    r9                      ; clear index register
l2:     double  dest                    ; double dest as bcd integers
        bitl    #^x80000000,r10         ; test high bit of r10
        beql    l3                      ; branch if zero bit
        incmnt  dest                    ; increment if one bit
l3:     ashl    #1,r10,r10              ; left shift binary input
        acbb    #31,#1,r9,l2            ; loop to process all 32 bits
        acbb    #7,#1,r8,l1             ; loop to process eight longwords
        popr    #^m<r8,r9,r10,r11>      ; restore altered registers
        .endm   genpkd
```

```
        .macro  xhexo   srcec,dest,prt=#1,?l1,?l2,?l3,?l4       ;---------

; 64-CHARACTER ASCII HEXADECIMAL STRING FROM
; 64-DIGIT PSEUDO PACKED DECIMAL SOURCE.

; r6 holds index for acbb loop
; r7 holds ptr to low digit of source string
; r8 holds ptr to high byte of dest string

        pushr   #^m<r6,r7,r8>   ; save registers used
        movab   dest+64,r8      ; get @ one above dest high byte
        movab   srce,r7         ; get source address
        clrb    r6              ; clear index register
l1:     cvtbl   (r7),wklwd      ; source byte to a longword
        bicl2   #^xfffffff0,wklwd ; mask all but right nibble
        addl2   #^x30,wklwd     ; generate a numeric cx 1...9
        cmpl    wklwd,#^x39     ; check if cx > 9
        bleq    l2              ; branch if numeric cx
        addl2   #7,wklwd        ; generate a hex cx a...f
l2:     cvtlb   wklwd,-(r8)     ; place cx in dest string
        cvtbl   (r7)+,wklwd     ; get the same source byte
        bicl2   #^xfffffff0f,wklwd ; mask all but second nibble
        ashl    #-4,wklwd,wklwd ; rt adjust the nibble
        addl2   #^x30,wklwd     ; generate a numeric cx 1...9
        cmpl    wklwd,#^x39     ; check if cx > 9
        bleq    l3              ; branch if numeric cx
        addl2   #7,wklwd        ; generate a hex cx a...f
l3:     cvtlb   wklwd,-(r8)     ; place cx in dest string
        acbb    #63,#2,r6,l1    ; loop to process 64 bytes
        movb    #^x2b,dest      ; insert plus sign
        cmpb    #0,prt          ; check for no print
        beql    l4              ; bypass print instr
        printchrs dest,#64      ; print result
l4:     popr    #^m<r6,r7,r8>   ; restore altered registers
        .endm   xhexo
```

```
        .macro  iread  srce,dest,?okstr,?next,?rev,?fwd          ;---------

; VARIABLE LENGTH READ FROM ASCII STRING WITH LEADING
; SIGN OR SPACE. GENERATES 32-BYTE BINARY NUMBER.

; r0 returns length of record
; r9 holds @ for inline
; r10 used for ASCII conversion

        pushr   #^m<r2,r3,r4,r5,r9,r10> ; save registers
        read'srce inline            ; read a record
        cmpb    inline,#^x20        ; check for space
        beql    okstr
        cmpb    inline,#^x2b        ; check for +
        beql    okstr
        cmpb    inline,#^x2d        ; check for -
        beql    okstr
        printchrs badstr            ; leading symbol missing
        brw     errstp              ; exit the macro
okstr:  subl3   #1,r0,inlen         ; adjust input length
        clrl    r10                 ; r10 ready for conversion
        extclr  result
        movab   inline,r9           ; get address for inline
        incl    r9                  ; adjust address
next:   extlls  result              ; left shift result area
        ls2copy result,copy         ; twice-shifted result to copy
        addx2   copy,result         ; sum copy and result
        subb3   #48,(r9)+,r10       ; get numeric equiv of ASCII cx
        extclr  num10               ; clear 32 byte area
        movl    r10,num10           ; move in a numeric
        addx2   num10,result        ; combine with result
        sobgtr  inlen,rev           ; repeat for all input bytes
        brb     fwd                 ; bypass reverse label
rev:    brw     next                ; loop for more input
fwd:    movc3   #32,result,dest     ; result to destination
        popr    #^m<r2,r3,r4,r5,r9,r10> ; restore registers
        .endm   iread

        .macro  ls2copy srce dest                               ;---------

; 32-BYTE LEFT-SHIFT & COPY FUNCTION, TWO-BIT SHIFT.

        .irp    offset, <0,4,8,12,16,20,24,28>      ; rept block args
        insv    srce+'offset,#2,#32,dest+'offset    ; use insert instr
        .endr
        .endm   ls2copy
```

```
        .macro    itime                                          ;---------

; REAL TIME DISPLAY FROM SYSTEM CLOCK.

        $GETTIM_S  TIMADR=TMVAL
        $ASCTIM_S  TIMBUF=ATIME,TIMADR=TMVAL
        printchrs  tmstr,#23
        .endm      itime

        .macro    ring                                           ;---------

; RING BELL AT VIDIEO TERMINAL

        movaw     RG_BL,OUTRAB+ RAB$L_RBF
        movw      #1,OUTRAB+ RAB$W_RSZ
        $PUT      RAB=OUTRAB
        .endm     ring
```

APPENDIX B

PROCEDURE MODULES

```
        .PSECT  ADDSUB

op1:    .blkb   64
op2:    .blkb   64

        .ENTRY  EXTADD,^m<r2,r3,r4,r5>                  ;---------

; 32-BYTE ADDITION, TWO OR THREE OPERANDS.

        movc3   #32,@8(ap),op1          ; operands to local areas
        movc3   #32,@4(ap),op2
        addl2   op1,op2                 ; add low order longwords
        .irp    offset,<4,8,12,16,20,24,28>  ; repeat block args
        adwc    op1+'offset,op2+'offset ; use extended addition instr
        .endr
        cmpl    #2,(ap)                 ; check for two operand case
        bneq    triop1
        movc3   #32,op2,@4(ap)          ; two operands, called by addx2
        ret
triop1: movc3   #32,op2,@12(ap)         ; three operands, called by addx3
        ret

        .ENTRY  DEXADD,^m<r2,r3,r4,r5>                  ;---------

; 64-BYTE ADDITION, TWO OPERANDS.

        movc3   #64,@8(ap),op1          ; operands to local areas
        movc3   #64,@4(ap),op2
        addl2   op1,op2                 ; add low order longwords
        .irp    offset,<4,8,12,16,20,24,28,32,36,40,44,48,52,56,60>
        adwc    op1+'offset,op2+'offset ; use extended addition instr
        .endr
        movc3   #64,op2,@4(ap)          ; two operands, called by addxx2
        ret

        .ENTRY  EXTSUB,^m<r2,r3,r4,r5>                  ;---------

; 32-BYTE SUBTRACTION, TWO OR THREE OPERANDS.

        movc3   #32,@8(ap),op1          ; operands to local areas
        movc3   #32,@4(ap),op2
        subl2   op1,op2                 ; subtract low order longwords
        .irp    offset,<4,8,12,16,20,24,28>  ; repeat block args
        sbwc    op1+'offset,op2+'offset ; use extended subtraction instr
        .endr
        cmpl    #2,(ap)                 ; check for two operand case
        bneq    triop2
        movc3   #32,op2,@4(ap)          ; two operands, called by subx2
```

```
        ret
triop2: movc3   #32,op2,@12(ap)          ; three operands, called by subx3
        ret

        .ENTRY  DEXSUB,^m<r2,r3,r4,r5>                    ;---------

; 64-BYTE SUBTRACTION, TWO OPERANDS.

        movc3   #64,@8(ap),op1           ; operands to local areas
        movc3   #64,@4(ap),op2
        subl2   op1,op2                  ; subtract low order longwords
        .irp    offset,<4,8,12,16,20,24,28,32,36,40,44,48,52,56,60>
        sbwc    op1+'offset,op2+'offset  ; use extended subtraction instr
        .endr
        movc3   #64,op2,@4(ap)           ; two operands, called by subxx2
        ret

        .PSECT  MULDIV

muld:   .blkb   32      ; multiplicand
mulr:   .blkb   32      ; multiplier
prod:   .blkb   64      ; product
muldwk: .blkb   64      ; multiplicand work area
        .=muld          ; begin overlay
quot::  .blkb   32      ; quotient
rem::   .blkb   32      ; remainder
prtrem: .blkb   64      ; partial remainder
divsha: .blkb   64      ; end overlay, divisor work area
ndx1:   .byte           ; index bytes
ndx2:   .byte
wklwd:  .long           ; holds four bytes of multiplier
modl:   .blkb   32      ; modulus for exponentiation
expn:   .blkb   32      ; exponent
nmbr:   .blkb   32      ; number for exponentiation
ans:    .blkb   64      ; result of exponentiation
c0r0:   .blkb   32      ; coefficients in remainder eqn's,
c0r1:   .blkb   32      ;         used in procedure extinv
c1r0:   .blkb   32
c1r1:   .blkb   32
c2r0:   .blkb   32
c2r1:   .blkb   32
tmp0:   .blkb   32      ; temporary storage areas
tmp1:   .blkb   32
rem0:   .blkb   32      ; remainder storage areas
rem1:   .blkb   32
ovflo:  .asciz  /product exceeds 32 bytes/
gcd:    .asciz  /gcd = 1 in procedure extinv, inverse exists/
inverr: .asciz  /error in extinv, gcd not one/
```

```
cquo:   .blkb   32      ; current quotient in extinv
dvcnt:  .word           ; divide count in extinv
lndx1:  .word           ; loop index in extnv

        .ENTRY  EXTMUL,^m<r2,r3,r4,r5,r6>                ;---------

; 32-BYTE MULTIPLICATION, TWO OR THREE OPERANDS,
; 32 OR 64 BYTE PRODUCT.

; r6 holds address of multiplier

        movc3   #32,@12(ap),muld        ; operands to local areas
        movc3   #32,@8(ap),mulr
        dexclr  prod                    ; clear product and work areas
        dexclr  muldwk
        clrb    mpysgn                  ; clear sign byte
        bitb    #^x80,muld+31           ; test multiplicand sign
        beql    posmld                  ; branch if positive
        negate  muld,32                 ; complement if negative
        xorb2   #^xff,mpysgn            ; reverse sign byte
posmld: bitb    #^x80,mulr+31           ; test multiplier sign
        beql    posmlr                  ; branch if positive
        negate  mulr,32                 ; complement if negative
        xorb2   #^xff,mpysgn            ; reverse sign byte
posmlr: movc3   #32,muld,muldwk         ; multiplicand to work area
        clrb    ndx1                    ; clear two index bytes
        clrb    ndx2
        moval   mulr,r6                 ; multiplier address to r6
l2:     movl    (r6)+,wklwd             ; four bytes of mulr to wklwd
l1:     blbc    wklwd,noadd             ; branch if low bit was zero
        addxx2  muldwk,prod             ; add if low bit was one
noadd:  dexlls  muldwk                  ; shift multiplicand left
        ashl    #-1,wklwd,wklwd         ; shift multiplier right
        acbb    #31,#1,ndx1,l1          ; loop for 32 multiplier bits
        clrb    ndx1                    ; reset inner loop index
        acbb    #7,#1,ndx2,l2           ; loop for 8 multiplier longwords
        blbc    4(ap),done1             ; branch if 64-byte product
        cmpc3   #32,prod+32,extzro      ; check for product > 32 bytes
        beql    prodok                  ; branch if no product overflow
        printchrs ovflo                 ; print overflow message
prodok: bitb    #^xff,mpysgn            ; test sign byte
        beql    posprd                  ; branch if positive
        brb     negprd                  ; branch if negative
posprd: brw     done1                   ; secondary branch
negprd: negate  prod,64                 ; complement if negative
done1:  cmpl    #3,(ap)                 ; check for two operand case
        bneq    triop3                  ; branch if three operands
```

```
        blbs    4(ap),done2             ; branch if 32-byte product
        movc3   #64,prod,@8(ap)         ; move a 64-byte product
        ret
done2:  movc3   #32,prod,@8(ap)         ; two operands,called by mulx2
        ret
triop3: movc3   #32,prod,@16(ap)        ; three operands, called by mulx3
        ret

        .ENTRY  EXTDIV,^m<r2,r3,r4,r5,r6>               ;---------

; 32-BYTE NON-RESTORING DIVISION,
; FOUR OPERANDS. DIVIDEND MAY BE 64 BYTES.

; r6 holds index for acbl instruction

        extclr  quot                    ; clear quotient area
        blbc    4(ap),dv64              ; branch if 64-byte dividend
        movc3   #32,@20(ap),prtrem      ; move 32-byte dividend
        extclr  prtrem+32               ; divd now in prtrem, rt adj
        brb     dv32                    ; divd is 32-byte
dv64:   movc3   #64,@20(ap),prtrem      ; move 64-byte dividend
dv32:   movc3   #32,@16(ap),divsha+32   ; move 32-byte divisor
        extclr  divsha                  ; divr now in divsha, left adj
        dexlrs  divsha                  ; initial shift of divisor
        subxx2  divsha,prtrem           ; get first partial remainder
        clrl    r6                      ; clear index register
loop:   extlls  quot                    ; left shift quotient
        dexlrs  divsha                  ; right shift divisor
        bitb    #^x80,prtrem+63         ; check sign of partial rem
        beql    pospr1                  ; branch if positive
        brw     negpr1                  ; branch if negative
pospr1: brw     pospr2                  ; secondary branch
negpr1: addxx2  divsha,prtrem           ; prtrem was neg, add the divr
        brw     foot                    ; branch to loop instr
pospr2: incb    quot                    ; since prtrem was +, incr quot
        subxx2  divsha,prtrem           ; prtrem was pos, subtr the divr
foot:   acbl    #254,#1,r6,loop         ; loop for all bits
        extlls  quot                    ; final shift of quotient
        bitb    #^x80,prtrem+63         ; check sign of final rem
        beql    posrm1                  ; branch if positive
        brb     fixrem                  ; branch if negative
posrm1: brw     fixquo                  ; secondary branch
fixrem: addxx2  divsha,prtrem           ; correct neg remainder
        brb     done3                   ; division complete
fixquo: bisb2   #^x01,quot              ; correct quot on pos rem
done3:  movc3   #32,prtrem,@8(ap)       ; remainder moved
        movc3   #32,quot,@12(ap)        ; quotient moved
        ret
```

```
        .ENTRY EXTEXP,^m<r2,r3,r4,r5,r6>                ;---------

; NUMBER EXPONENTIATION TO A MODULUS,
; ALL THREE OPERANDS ARE 32-BYTE.

; r6 holds index for acbl instruction

        itime   61                      ; supply time mark
        movl    #255,exlim              ; set limit for acbl loop
        clrl    r6                      ; clear index register
        movc3   #32,@4(ap),modl         ; operands to local areas
        movc3   #32,@8(ap),expn
        movc3   #32,@12(ap),nmbr
        movc3   #32,extone,ans          ; initialize ans to one
extst:  bitb    #^x80,expn+31           ; test exponent high bit
        beql    btzro                   ; branch if zero
        brw     exlp                    ; branch if one
btzro:  extlls  expn                    ; left shift exponent
        decl    exlim                   ; decrement main loop limit
        brw     extst                   ; loop to check next bit
exlp:   mulxx2  ans,ans                 ; in main loop, square ans
        modxx2  ans,modl                ; obtain ans to given modulus
ex1:    bitb    #^x80,expn+31           ; test hi bit of exponent
        beql    ex2                     ; branch if zero bit
        brb     ex3                     ; branch if one bit
ex2:    brw     b0                      ; secondary branch
ex3:    mulxx2  nmbr,ans                ; ans times given number
        modxx2  ans,modl                ; obtain ans to given modulus
b0:     extlls  expn                    ; left shift exponent one bit
        acbl    exlim,#1,r6,exlp        ; loop for all exponent bits
        movc3   #32,ans,@12(ap)         ; nmbr**exp mod m is answer
        ret

        .ENTRY  EXTINV,^m<r2,r3,r4,r5,r6>               ;---------

; MULTIPLICATIVE INVERSE TO A MODULUS,
; BOTH OPERANDS ARE 32-BYTE.

; r6 holds stack pointer

        clrw    dvcnt                   ; clear divide count
        clrw    lndx1                   ; clear loop index
        movc3   #32,@4(ap),rem0         ; this is the modulus
        movc3   #32,@8(ap),rem1         ; numbr whose inverse is reqd
        movc3   #32,extone,c0r0         ; set up initial coefficients
        movc3   #32,extzro,c1r0
        movc3   #32,extzro,c0r1
        movc3   #32,extone,c1r1
```

```
        movl    sp,r6                           ; store the stack pointer
lp1:    divx4   rem0,rem1,quot,rem              ; modulus / nmbr
        incw    dvcnt                           ; increment divide count
        .irp    offset,<0,4,8,12,16,20,24,28>
        pushl   quot+'offset                    ; push 32-byte quotient
        .endr
        cmpc3   #32,rem,extone                  ; compare remainder to one
        beql    gcd1                            ; branch if rem equals one
        cmpc3   #32,rem,extzro                  ; compare remainder to zero
        beql    err1                            ; branch to error stop
        brb     in1                             ; bypass code
err1:   brw     error                           ; secondary branch
in1:    movc3   #32,rem1,rem0                   ; previous rem moved to rem0
        movc3   #32,rem,rem1                    ; current rem becomes previous rem
        brw     lp1                             ; loop for another division
gcd1:   printchrs gcd
        decw    dvcnt                           ; adjust divide count
lp2:    .irp    offset,<0,4,8,12,16,20,24,28>
        movl    -(r6),cquo+'offset              ; pop current quotient from stack
        .endr
        mulx3   cquo,c1r0,tmp0                  ; compute coefficients of
        mulx3   cquo,c1r1,tmp1                  ;       r0 and r1 in remainder
        subx3   tmp0,c0r0,c2r0                  ;       expressions
        subx3   tmp1,c0r1,c2r1
        movc3   #32,c1r0,c0r0                   ; move coefficients, prepare
        movc3   #32,c2r0,c1r0                   ;       for calculation with
        movc3   #32,c1r1,c0r1                   ;       next quotient
        movc3   #32,c2r1,c1r1
        acbw    dvcnt,#1,lndx1,lp2              ; loop divcnt times
        bitb    #^x80,c2r1+31                   ; test sign of r1 coefficient
        bneq    c2neg                           ; branch if negative
        movc3   #32,c2r1,rem1                   ; move coeff if positive
        brb     c2pos                           ; bypass code
c2neg:  addx3   c2r1,@4(ap),rem1                ; adjust coeff if negative
c2pos:  movc3   #32,rem1,@8(ap)                 ; number replaced by inverse
        ret
error:  printchrs inverr                        ; error message, gcd not one
        brw     errstp                          ; go to main program exit
        ret

        .PSECT  JACOBI

msg1:   .asciz  /b not odd/
msg2:   .asciz  /a not .LE. b/
msg3:   .asciz  /(a,b) .NE. 1/
arga:   .blkb   32
argb:   .blkb   32
tempa:  .blkb   32
```

```
tempb:  .blkb   32
tempc:  .blkb   32
diff:   .blkb   32
jcans:: .long
jcflg:: .byte
temp1:  .blkl   1
temp2:  .blkl   1

        .ENTRY  JACO1,^m<r2,r3,r4,r5>                   ;--------

; JACOBI SYMBOL DETERMINATION, ARGUMENT TESTING

        movc3   #32,@8(ap),tempa        ; move operands to local areas
        movc3   #32,@4(ap),tempb
        blbs    tempb,t1ok              ; check if argb is odd
        printchrs msg1                  ; error msg, argb even
        brw     quit2                   ; branch to error return
t1ok:   subx3   tempa,tempb,diff        ; check if arga < or = argb
        bitb    #^x80,diff+31           ; test sign bit of diff
        beql    t2ok                    ; diff positive, test ok
        printchrs msg2                  ; error msg, arga > argb
        brw     quit2                   ; branch to error return
t2ok:   divx4   tempb,tempa,quot,rem    ; begin gcd test loop
        cmpc3   #32,rem,extzro
        beql    zrorem                  ; branch if rem is zero
        movc3   #32,tempa,tempb         ; divisor becomes dividend
        movc3   #32,rem,tempa           ; remainder becomes divisor
        brw     t2ok                    ; loop for next division
zrorem: cmpc3   #32,tempa,extone        ; last non-zero rem compared to 1
        beql    t3ok                    ; gcd = 1, test ok
        printchrs msg3                  ; error msg, gcd not = 1
        brw     quit2                   ; branch to error return
t3ok:   pushl   8(ap)                   ; push argument addresses
        pushl   4(ap)
        pushal  jcans                   ; push answer address
        calls   #3,jaco2                ; call calculation procedure
        dumplongjcans
        bicb    #^xff,jcflg             ; set flag for normal return
        ret
quit2:  bisb    #^xff,jcflg             ; set flag for error return
        ret

        .ENTRY JACO2 ^m<r2,r3,r4,r5>                    ;---------

; JACOBI SYMBOL DETERMINATION, RECURSIVE CALCULATION

        movc3   #32,@12(ap),arga        ; move operands to local areas
```

```
        movc3   #32,@8(ap),argb
        cmpc3   #32,arga,extone         ; arga compared to one
        bneq    more                    ; branch if not equal
        movl    #1,@4(ap)               ; arg a is one, jacobi is one
        ret                             ; procedure complete, return
more:   blbs    arga,aodd               ; check if arga is odd
        brw     aeven                   ; branch if even
aodd:   subx3   extone,arga,tempa       ; calculate exponent for -1
        subx3   extone,argb,tempb
        mulx3   tempa,tempb,tempc
        extlrs  tempc                   ; division by 4
        extlrs  tempc
        movc3   #32,tempc,tempb         ; (a-1)(b-1)/4, exp for -1, in tempb
        divx4   argb,arga,quot,rem      ; new arga is b(mod a)
        movc3   #32,arga,argb           ; new argb is old arga
        movc3   #32,rem,arga            ; remainder to arga
        brw     exptst                  ; complete for case of arga odd
aeven:  mulx3   argb,argb,tempb         ; calculate exponent for -1
        subx2   extone,tempb
        extlrs  tempb                   ; division by 8
        extlrs  tempb
        extlrs  tempb                   ; (b**2-1)/8, exp for -1, in tempb
        extlrs  arga                    ; new arga is arga/2
exptst: blbc    tempb,expeve            ; odd-even test on exp for -1
        pushl   #-1                     ; exp was odd, push -1 on stack
        brb     jct2                    ; bypass code
expeve: pushl   #1                      ; exp was even, push +1 on stack
jct2:   pushal  arga                    ; push argument addresses
        pushal  argb
        pushal  jcans                   ; push answer address
        calls   #3,jaco2                ; make recursive call
        movl    (sp)+,temp1             ; pop +1 or -1 from stack
        movl    @4(ap),temp2            ; get current jacobi answer
        mull3   temp1,temp2,@4(ap)      ; get new jacobi answer
        ret
```

```
        .PSECT  RANDOM

size:   .word           ; how many numbers needed
rndx:   .word           ; index for acbw loop
pmod:   .blkb   32      ; modulus
a:      .blkb   32      ; multiplier
c:      .blkb   32      ; additive constant
seed:   .blkb   32      ; initial number
enmgen: .asciz  / entered numgen /

        .ENTRY  NUMGEN      ^m<r2,r3,r4,r5>          ;---------

; RANDOM NUMBER GENERATION, MIXED
; LINEAR CONGRUENTIAL METHOD

        clrw    rndx                    ; clear index word
        printchrs enmgen
        movc3   #32,@4(ap),pmod         ; operands to local areas
        movc3   #32,@8(ap),c
        movc3   #32,@12(ap),a
        movc3   #32,@16(ap),seed
        movw    20(ap),size             ; get desired list length
rnd1:   mulx2   a,seed                  ; seed receives product
        addx2   c,seed                  ; seed now = (a*x+c)
        modx2   seed,pmod               ; seed now (a*x+c)mod pmod
        genpkd  seed,pkd                ; convert num to pkd decimal
        xhexo   pkd,hexout,#0           ; pkd decimal to ascii, no print
        wtjaco  hexout,#64              ; write all numbers to jaco.num
        blbs    seed,rnd2               ; check number for odd vs even
        brw     rnd1                    ; loop if even number
rnd2:   wtrand  hexout,#64              ; write odd numbers to rand.num
        acbw    size,#1,rndx,rnd1       ; loop until desired list complete
        ret

        .END
```

APPENDIX C

MAIN PROGRAM

```
        .PSECT  PUBKEY

rnsrt:  .asciz  /rand.num sorted, descending seq/
jnsrt:  .asciz  /jaco.num sorted, ascending seq/
rderjc: .asciz  /error, at eof in jaco.num/
rderrn: .asciz  /error, at eof in data.dat/
badstr: .asciz  /pls precede input str with +, -, or sp/
cpnum:  .asciz  /compound number found/
prnum:  .asciz  /prime number found/
mmsg1:  .asciz  /primes p & q,d read from data.dat:/
mmsg2:  .asciz  /r = p * q displayed from num4:/
mmsg3:  .asciz  /phi(r) displayed from num7:/
mmsg4:  .asciz  /prime d, its inv h displayed:/
mmsg6:  .asciz  /have d,h & r, ready to encode data/
mmsg7:  .asciz  /asci string to encode, 20 cx segment displayed:/
mmsg10: .asciz  /enciphered msg, four 32-byte blocks displayed:/
mmsg11: .asciz  /ready to decipher message:/
mmsg12: .asciz  /the message deciphered:/

jcndx:  .byte
rg_bl:  .byte   7                               ; ring console bell
extzro:: .byte  0[32]                           ; zero, 32-byte format
extone:: .byte  1,0[31]                         ; one, 32-byte format
rseed:  .byte   1,0[31]                         ; args for procedure NUMGEN
rmpy:   .byte   216,163,112,61,10,215,163,112,61,10,215,163,0[20]
rcon:   .byte   49,156,49,218,143,234,166,204,86,239,67,144,1,0[19]
rmod:   .byte   255[12],15,00[19]
wrd1::  .word
long1:  .long
hexout:: .blkb  64                              ; output line, macro XHEXO
cph1:   .blkb   128                             ; area for ciphertext
numndx: .byte
result: .blkb   32
copy:   .blkb   32
nmbrs:  .blkb   320                             ; computation area, 32-byte nums
        .=nmbrs
num1:   .blkb   32
num2:   .blkb   32
num3:   .blkb   32
num4:   .blkb   32
num5:   .blkb   32
num6:   .blkb   32
num7:   .blkb   32
num8:   .blkb   32
num9:   .blkb   32
num10:  .blkb   32
```

```
inline:   .blkb    80                          ; terminal input/output
inlen:    .long                                ; input length, macro IREAD
wklwd:    .long                                ; longword, macro XHEXO
pkd::     .blkb    64                          ; pseudo-packed decimal area

table::   .byte    0,2,4,6,8,16,18,20,22,24,0,0,0,0,0,0,1,3,5,7,9,17,19,-
                   21,23,25,0,0,0,0,0,0        ; table for macro DOUBLE

lnbyte::  .byte                                ; left nibble, macro DOUBLE
rnbyte::  .byte                                ; right nibble, macro DOUBLE

FNAME1: .ASCII  /RAND.NUM/                     ; file names for sort procedure
FNAME2: .ASCII  /JACO.NUM/

RNFIL:    .LONG   8                            ; sort parameters for rand.num
          .ADDRESS FNAME1
KBUF:     .WORD   1
KTYP:     .WORD   1
KORD:     .WORD   1
KPOS:     .WORD   1
KSIZ:     .WORD   64

JNFIL:    .LONG   8                            ; sort parameters for jaco.num
          .ADDRESS FNAME2
LBUF:     .WORD   1
LTYP:     .WORD   1
LORD:     .WORD   0
LPOS:     .WORD   1
LSIZ:     .WORD   64

ATIME::   .LONG   23                           ; storage areas for time display
          .ADDRESS TMSTR
TMVAL::.BLKQ    1
TMSTR::.BLKB    23
EXPMSG:           .ASCIZ   / (time mark in exponential procedure)/

                           ; declare system sort procedures external

          .EXTRN SOR$PASS_FILES,SOR$INIT_SORT,SOR$SORT_MERGE,-
                 SOR$END_SORT
```

```
        BEGIN   PUBKEY

; DEMONSTRATION OF PUBLIC-KEY CRYPTOSYSTEM
; USING THE RSA ALGORITHM

        bicpsw  #^x00f0
        itime
        $CREATE FAB=DSK2FAB             ; create file rand.num
        $CONNECT RAB=DSK2RAB
        $CREATE FAB=DSK3FAB             ; create file jaco.num
        $CONNECT RAB=DSK3RAB
        pushl   #500                    ; push args for number generator
        pushal  rseed
        pushal  rmpy
        pushal  rcon
        pushal  rmod
        calls   #5,numgen
        $CLOSE FAB=DSK2FAB              ; close file rand.num
        $CLOSE FAB=DSK3FAB              ; close file jaco.num
        pushab  rnfil                   ; sort file rand.num
        pushab  rnfil
        calls   #2,G^SOR$PASS_FILES
        pushab  kbuf
        calls   #1,G^SOR$INIT_SORT
        calls   #0,G^SOR$SORT_MERGE
        calls   #0,G^SOR$END_SORT
        printchrs rnsrt
        pushab  jnfil                   ; sort file jaco.num
        pushab  jnfil
        calls   #2,G^SOR$PASS_FILES
        pushab  lbuf
        calls   #1,G^SOR$INIT_SORT
        calls   #0,G^SOR$SORT_MERGE
        calls   #0,G^SOR$END_SORT
        printchrs jnsrt
        $CREATE FAB=DISKFAB             ; create file data.dat
        $CONNECT RAB=DISKRAB
        $OPEN   FAB=DSK2FAB             ; open file rand.num
        $CONNECT RAB=DSK2RAB
loop1:  $CLOSE  FAB=DSK3FAB             ; rewind & open file jaco.num
        $OPEN   FAB=DSK3FAB
        $CONNECT RAB=DSK3RAB
        clrb    jcndx
        iread   rand,num2               ; read from rand.num
```

```
loop2:  iread     jaco,num1                ; read from jaco.num
        jacobi    num1,num2
        blbs      jcflg,j1                 ; branch if failure in JACO1
        brb       j2                       ; bypass code
j1:     brw       loop1                    ; secondary branch
j2:     subx3     extone,num2,num3         ; b-1 is in num3
        extlrs    num3                     ; num3 is exponent (b-1)/2
        movc3     #32,num1,num4            ; num4 is a
        expx3     num4,num3,num2           ; a**(b-1)/2 mod b in num4
        bitl      #^x80000000,jcans        ; inspect sign of jacobi answer
        beql      jcpos
        addx2     extone,num4              ; jacobi was neg
        brw       j3
jcpos:  subx2     extone,num4              ; jacobi was pos
j3:     divx4     num4,num2,quot,rem
        cmpc3     #32,extzro,rem           ; congruence valid if rem = 0
        bneq      jfail                    ; failed test of s&s algorithm
        acbb      #14,#1,jcndx,loop2       ; loop for 15 jacobi tests
        brw       j4                       ; bypass code
jfail:  printchrs cpnum                    ; compound number found
        brw       loop1
j4:     printchrs prnum                    ; prime number found
        ring
        genpkd    num2,pkd                 ; get prime num in packed form
        xhexo     pkd,hexout
        wtrcrd    hexout,#64               ; write prime to data.dat
        brw       loop1
eof2:   $CLOSE    FAB=DISKFAB              ; rewind & open file data.dat
        $OPEN     FAB=DISKFAB
        $CONNECT RAB=DISKRAB
        iread     rcrd,num3                ; read three large prime numbers
        iread     rcrd,num2
        iread     rcrd,num1
        printchrs mmsg1                    ; primes p,q,d read from data.dat
        genpkd    num1,pkd
        xhexo     pkd,hexout               ; display three prime numbers
        genpkd    num2,pkd
        xhexo     pkd,hexout
        genpkd    num3,pkd
        xhexo     pkd,hexout
        mulx3     num1,num2,num4           ; product r = p * q now in num4
        printchrs mmsg2
        genpkd    num4,pkd
        xhexo     pkd,hexout               ; display factor r
        subx3     extone,num1,num5         ; p - 1 now in num5
        subx3     extone,num2,num6         ; q - 1 now in num6
        mulx3     num5,num6,num7           ; phi(r) now in num7
```

```
        printchrs mmsg3
        genpkd    num7,pkd
        xhexo     pkd,hexout              ; phi(r) displayed
        pushr     #^m<r2,r3,r4,r5>
        movc3     #32,num3,num8           ; copy prime d into num8
        popr      #^m<r2,r3,r4,r5>
        invx2     num8,num7               ; h, mult inv of d mod phi(r)
        printchrs mmsg4
        genpkd    num3,pkd
        xhexo     pkd,hexout              ; display prime factor d
        genpkd    num8,pkd
        xhexo     pkd,hexout              ; display h, inverse of d
more:   printchrs mmsg6                   ; ready to encode data
        dexclr    inline                  ; clear an 80-byte area
        clrq      inline+64
        clrq      inline+72
        readline  inline                  ; get message from terminal
        movab     inline,r9               ; message @ to r9
        movab     cph1,r10                ; cipher string @ to r10
        clrb      numndx
more2:  extclr    num10
        movc3     #20,(r9),num10          ; num10 receives 20 cx of msg
        printchrs mmsg7
        xhexo     num10,hexout            ; asci string to encode, 20 cx
        expx3     num10,num8,num4         ; msg**h mod r now in num10
        movc3     #32,num10,(r10)         ; cipher string moved to cph1
        addl2     #20,r9                  ; set r9 for next 20 cx of msg
        addl2     #32,r10                 ; set r10 for next cipher string
        acbb      #3,#1,numndx,more2      ; repeat for total of 4 strings
        clrb      numndx
        movab     cph1,r10                ; cipher string @ to r10
        printchrs mmsg10
more3:  xhexo     (r10),hexout            ; cipher string displayed
        addl2     #32,r10                 ; set r10 for next cipher string
        acbb      #3,#1,numndx,more3      ; repeat for total of 4 strings
        clrb      numndx
        movab     inline,r9               ; message @ to r9
        movab     cph1,r10                ; cipher string @ to r10
        printchrs mmsg11                  ; ready to decipher
more4:  movc3     #32,(r10),num10         ; 32 bytes of cipher to num10
        expx3     num10,num3,num4         ; cipher**d mod r now in num10
        movc3     #20,num10,(r9)          ; message string moved to inline
        addl2     #32,r10                 ; set r10 for next cipher string
        addl2     #20,r9                  ; set r9 for next message string
        acbb      #3,#1,numndx,more4      ; repeat for total of 4 strings
        clrb      numndx
        printchrs mmsg12
        printchrs inline,#80              ; original message displayed
```

```
            ring
            brw         more                ; branch for new message
eof3:       printchrs   rderjc              ; error, ran out of jacobi args
            brw         errstp
eof:        printchrs   rderrn              ; error, ran out of prime nums
errstp:: nop
            exit
            .end pubkey
```

APPENDIX D

SAMPLE PROGRAM RUN

20-JUL-1986 17:01:52.39

primes p, q & d read from data.dat:

+ 0000000000000000000000000000000062755511187524847607433482109 9

+ 0000000000000000000000000000000066869562922629224328896766754 7

+ 0000000000000000000000000000000800649833179762467645082232209

r = p * q displayed from num4:

+ 0004196433604095955032935278247203785310427632032995015531741 53

phi(r) displayed from num7:

+ 0004196433604095955032935278247190822803016616625801382506855 08

gcd = 1 in procedure extinv, inverse exists

prime d, its inv h shown below:

+ 0000000000000000000000000000000800649833179762467645082232209

+ 0001548886011241557094728069100836919283464509725480244989343 73

have d, h & r, ready to encode data

?? The hills are alive with the sound of music, sounds we have heard for a thousand

asci string to encode, 20 cx segment displayed:

+ 0000000000000000000000206576696C6120657261207 36C6C696820656854

20-JUL-1986 17:02:56.04 (time mark in exponential procedure)

asci string to encode, 20 cx segment displayed:

+ 0000000000000000000000756D20666F20646E756F73206568742068746977 7

20-JUL-1986 17:03:54.63 (time mark in exponential procedure)

asci string to encode, 20 cx segment displayed:

+ 00000000000000000000000206576616820657720736 46E756F73202C636973

20-JUL-1986 17:05:01.79 (time mark in exponential procedure)

asci string to encode, 20 cx segment displayed:

+ 00000000000000000000000646E6173756F6874206120726F66206472616568

20-JUL-1986 17:06:07.69 (time mark in exponential procedure)

enciphered msg, four 32-byte blocks displayed:

+ 0000000000000088C3D1355EAB983FF75CD061ABFB560082F4D4ACCF76BE69B

+ 00000000000002DF92B4C99684855BD4676BECC9A89356EF8B3AF31A4A37204

+ 00000000000001B06370D4F9AA42BBAF1B767742A590DF3A86F698F445A9F35

+ 0000000000000308CAE5EB1DC5CE0DD3CE1D992ADD26D2189B5BDF8AAF5AFD3

ready to decipher message:

20-JUL-1986 17:06:51.37 (time mark in exponential procedure)

20-JUL-1986 17:07:19.24 (time mark in exponential procedure)

20-JUL-1986 17:07:36.83 (time mark in exponential procedure)

20-JUL-1986 17:07:59.58 (time mark in exponential procedure)

the message deciphered:

The hills are alive with the sound of music, sounds we have heard for a thousand

VITA

Robert Ewing Gibson was born in Knoxville, Tennessee on December 22, 1935. He attended elementary and secondary schools in that city, graduating from Central High School in June 1953. He entered the University of Tennessee where he received the degrees of Bachelor of Science in June 1957, and Master of Science in June 1960. His major was Electrical Engineering.

Beginning in February 1960 he was employed by Martin-Marietta Corporation, Orlando, Florida, with assignments in various aerospace and defense projects. In September 1966 Mr. Gibson accepted a position as Research Engineer with the Engineering Experiment Station, Georgia Institute of Technology, Atlanta. His duties were in the field of radar measurement techniques. In October 1979, he was employed by Tennessee State Testing and Evaluation Center, Knoxville, as a part-time programmer, and at the same time he entered Graduate School at the University of Tennessee. He received the degree of Master of Science with a major in Computer Science in August 1986.