INTEL 8085 Implementation of Byte-Wise Data Encryption using Successive RSA Algorithm coupled with Random Initial Primes

Ashish Menon
Indian Institute of Technology Guwahati
ashish18@iitg.ac.in

June 15, 2020

Abstract

This paper, discusses the implementation of RSA (Rivest–Shamir–Adleman) algorithm of encrypting and decrypting data. This is an assymetric algorithmic approach, which implies that two different keys, one public and another private have been used to accomplish the purpose. This paper also extends the application of RSA algorithm by choosing random prime numbers for every byte of transferred data. The programs involved are entirely based on the microprocessor INTEL 8085.

Contents

1	Introduction	2
2	RSA Algorithm	2
2.1	Creating public and private key	2
2.2	Encrypting the message	2
2.3	Decrypting the message	2
2.4	Working example	2
3	Implementation with 8085	3
3.1	Sender's program	3
3.2	Receiver's program	8
4	Utilizing random primes for encrypting long data byte-wise	12
4.1	Choosing random primes	13
4.2	8085 program for generating random numbers between 0-12 (inclusive)	14
4.3	8085 program for choosing random primes	18
5	Choose value of public and private key exponent for random primes	19
6	Final implementation of encryption for transferring 1kB of data	21
7	Conclusion	30
	References	30

1 Introduction

RSA (Rivest–Shamir–Adleman) is a public key encryption technique and is considered one of the most secure encryption techniques used by modern computers. This is an asymmetric algorithm, which means that two separate keys are involved. One of them is called the public key and is visible to everyone. The other is a private key and should not be shared with anyone other than the receiver. The algorithm is centered around the fact that finding the factors of a sufficiently large composite number is impossible in practice when the factors are prime numbers. The algorithm of this technique is discussed in the following section. Later, we deal with implementation of choosing random set of prime numbers for encrypting every byte of data by the Linear Congruential Generator (LCG) algorithm. To modularize the work done in this paper, we will take a bottom-up approach. First, we will deal with implementing a classic RSA algorithm to encrypt (at the sender's end) one byte of data and decrypt it (at the receiver's end) upon successful receipt. After this is done, we will discuss how the random choice of a set of initial prime numbers (which is the trademark of the RSA algorithm) is implemented.

2 RSA Algorithm

The classic RSA algorithm involves three steps: Creating public and private keys, encrypting message at sender's end and decrypting message at receiver's end. Each of these is discussed below:

2.1 Creating public and private key

- Step 1. Choose two large random prime numbers p and q .
- Step 2. Compute the product of the two primes $n = p \cdot q \rightarrow$ (eq. 2.01).
- Step 3. Compute totient $\phi(n) = (p - 1)(q - 1) \rightarrow$ (eq. 2.02).
- Step 4. Select an integer e such that $1 \leq e \leq \phi(n)$ and $\gcd(e, \phi(n)) = 1$ where $\gcd(\cdot)$ refers to the greatest common divisor.
- Step 5. Compute d , such that the following congruence relation is satisfied: $de \equiv 1 \pmod{\phi(n)} \rightarrow$ (eq. 2.03).

The public key is the set $P = \{e, n\}$ and the private key is the set $P' = \{d, p, q\}$. The strength of the encryption increases exponentially with the increasing number of key bits. Typically 1024 or 2048 bits long key size are used.

2.2 Encrypting the message

Usually while implementation, the RSA must be combined with some sort of padding scheme so that no message results in insecure cipher texts. In this project however, we will not discuss this bit. It is assumed that all the input information that is to be sent to the receiver is already processed by the agreed-upon reversible protocol. Say the sender wants to send a message m to the receiver. Due to padding, $m \leq n$. First the receiver should share his/her public key $P = \{e, n\}$ with the sender. The sender will then compute the corresponding cipher text by:

$$c = m^e \pmod{n} \rightarrow \text{(eq.2.04)}$$

Then the sender will send the cipher text c to the receiver.

2.3 Decrypting the message

To decrypt the cipher text, the receiver has to make use of his/her private key as follows:

$$m = c^d \pmod{pq} \rightarrow \text{(eq.2.05)}$$

Ans thus the original message m is recovered back from the cipher text.

2.4 Working example

In this paper, we will see the encryption and decryption pertaining to one byte of data. Let us choose the prime numbers as $p = 67$ and $q = 83$. Then from eq. 2.01 and 2.02 we have, $n = 5561$ and $\phi(n) = 5412$. We choose $e = 17$ because $\gcd(17, 5412) = 1$. From eq. 2.03, we set $d = 9869$.

Say the sender wants to transfer the number $m = 103$ to the receiver. From eq. 2.04, we have $c = 103^{17} \pmod{5561} = 1413$. Similarly, in the receiver's end, from eq. 2.05 we have, $m = 1413^{9869} \pmod{5561} = 103$. We will implement this transmission using 8085 in the following sections.

Note that the values of the key components are unique to this simulation. But, the message m is not. $m = 103$ was just a random choice that was made. We want to build a program which can transfer any data that is one byte long. So, in eq. 2.04,

$\max(m^e) = 256^{17}$ which would be 17 bytes long. Also $\max(c) = 5560$ In eq. 2.05, $\max(c^d) = 5560^{9869}$. Taking the logarithm of this value to the base of 256 we get 15347.367, which means that the maximum value is atleast 15348 bytes long.

3 Implementation with 8085

Now, in practice the prime numbers that are used in the RSA algorithm are 1024 or 2048 bits (128 or 256 bytes) long. But for simplicity, we've constrained ourselves to 1 byte long primes ($p = 67$ and $q = 83$). In this section, the working example discussed in section 2.4 is programmed. Two separate programs have to be written each for the sender's and receiver's end. In writing each of these programs, much energy is spent in calculating the exponents and modulo given in eq. 2.04 and 2.05, primarily because of the large number of bytes. The algorithms for the same and the program codes are discussed in the following sections.

3.1 Sender's program

As discussed in section 2.4, in the sender's end a maximum of 17 bytes would be involved in the calculations of eq. 2.04. We require atleast 17 bytes of data to store m^e . We will utilize the addresses FFEF to FFFF to store the bytes of the same in decreasing order of significance i.e. MSB (most significant byte) will be stored in address FFEF and LSB (least significant byte) will be store in address FFFF. The calculation of m^e is done on the following understanding:

$$\begin{aligned}
 m^e &= m.m^{e-1} \\
 &= \underbrace{m^{e-1} + m^{e-1} + \dots + m^{e-1}}_{m \text{ times}} \\
 &= \underbrace{\left(\underbrace{m^{e-2} + \dots + m^{e-2}}_{m \text{ times}} \right) + \dots + \left(\underbrace{m^{e-2} + \dots + m^{e-2}}_{m \text{ times}} \right)}_{m \text{ times}} \\
 &\vdots \\
 &= \underbrace{\left(\dots \left(\underbrace{m + m + \dots + m}_{m \text{ times}} \right) + \left(\underbrace{m + m + \dots + m}_{m \text{ times}} \right) + \dots + \left(\underbrace{m + m + \dots + m}_{m \text{ times}} \right) \right) + \dots}_{e-1 \text{ times}}
 \end{aligned}$$

The flowcharts for calculating m^e and modulo n of the same are given below.

Figure 1: Flowchart for calculation of m^e

Figure 2: Flowchart for calculation of $m^e \bmod n$

	LABEL	MNEMONICS	HEX CODES	COMMENTS
C000		MVI A, 67(H)	3E,67	
C002		STA C407(H)	32,07,C4	Store m = 103 = 0x67
C005		MVI A, 11(H)	3E,11	
C007		STA C408(H)	32,08,C4	Store e = 17 = 0x11
C00A		MVI A, 11(H)	3E,11	
C00C		STA C409(H)	32,09,C4	Store max byte size = 17 = 0x11
C00F		MVI A, 15(H)	3E,15	n = 5561 = 0x15B9
C011		STA C40A(H)	32,0A,C4	Store higher order bit of n = 0x15
C014		MVI A, B9(H)	3E,B9	
C016		STA C40B(H)	32,0B,C4	Store lower order bit of n = 0xB9
C019		LDA C408(H)	3A,08,C4	
C01C		MOV E, A	5F	Move e to reg. E
C01D		LXI H, FFEF(H)	21,EF,FF	Initialize reg. pair HL with address of MSB
C020	LOOP1	MVI A, 00(H)	3E,00	
C022		MOV M,A	77	Initialize data at memory location to 0x00
C023		DCR E	1D	
C024		JZ LOOP2	CA,2B,C0	If all 17 address locations are initialized to 0x00, go to LOOP2
C027		INX H	23	
C028		JMP LOOP1	C3,20,C0	If all 17 address location are not initialized point memory next address byte & go to LOOP1
C02B	LOOP2	STC	37	
C02C		CMC	3F	Reset Carry flag
C02D		LDA C408(H)	3A,08,C4	
C030		MOV E,A	5F	Load e to reg. E
C031		LDA C409(H)	3A,09,C4	
C034		MOV B,A	47	Store max byte size in reg. B
C035	REPEAT1	LXI H FFDE(H)	21,DE,FF	H-L points to content at address FFDE
C038		LXI D FFEF(H)	11,EF,FF	D-E points to MSB which will converge to c
C03B	REPEAT	LDAX D	1A	
C03C		MOV M,A	77	Copy content of DE reg, pair to memory
C03D		DCR B	05	Check if all 17 bytes are copied
C03E		JZ DONE	CA,46,C0	If yes, go to DONE
C041		INX H	23	Else , increment contents of HL reg. pair
C042		INX D	13	And increment contents of DE reg. pair
C043		JMP REPEAT	C3,3B,C0	And then go to REPEAT
C046	DONE	LDA C407(H)	3A,07,C4	
C049		MOV C,A	4F	Store m in reg. C
C04A	LOOP4	LDA C409(H)	3A,09,C4	
C04D		MOV B,A	47	Store max byte size in reg. B
C04E		LXI H FFFF(H)	21,FF,FF	Memory points to actual LSB
C051		LXI D FFEE(H)	11,EE,FF	DE reg. pair points to copy LSB
C054		STC	37	
C055		CMC	3F	Reset carry flag
C056	LOOP5	LDAX D	1A	
C057		ADC M	8E	
C058		MOV M,A	77	Add copy LSB to actual LSB
C059		DCR B	05	Indicates that current byte is updated
C05A		JZ NEXT	CA,62,C0	If all bytes are updated go to NEXT
C05D		DCX H	2B	Else , point memory to next LSB
C05E		DCX D	1B	Point DE reg. pair to next LSB of copy
C05F		JMP LOOP5	C3,56,C0	And go to LOOP5
C062	NEXT	DCR C	0D	Indicates that current iteration is done
C063		JZ LOOP3	CA,69,C0	If all m iterations are done, go to LOOP3
C066		JMP LOOP4	C3,4A,C0	If not , go to LOOP4
C069	LOOP3	DCR E	1D	Indicates that current iteration is done
C06A		JZ LOOP8	CA,70,C0	If all e iterations are completed go to LOOP8

	LABEL	MNEMONICS	HEX CODES	COMMENTS
C06D		JMP REPEAT1	C3,35,C0	Else , go to REPEAT1
C070	LOOP8	STC	37	
C071		CMC	3F	Reset Carry flag
C072		LDA C409 (H)	3A,09,C4	
C075		MOV B,A	47	Load max byte no. to reg. B
C076		LDA C40A (H)	3A,0A,C4	
C079		MOV D,A	57	Load higher order bit of n to reg. D
C07A		LDA C40B (H)	3A,0B,C4	
C07D		MOV E,A	5F	Load lower order bit of n to reg. E
C07E		LXI H,FFFF (H)	21,FF,FF	Initialize reg. pair HL with address of LSB
C081		MOV A,M	7E	
C082		SUB E	93	Subtract lower order bit of n from LSB
C083		MOV M,A	77	Update value of LSB
C084		DCR B	05	Indicates that LSB is updated
C085		DCX H	2B	Point memory to next LSB
C086		MOV A,M	7E	
C087		SBB D	9A	Subtract higher order bit of n from second LSB with borrow
C088		MOV M,A	77	Update value of second LSB
C089		DCR B	05	Indicates that second LSB is updated
C08A	LOOP6	DCX H	2B	Point memory to next LSB
C08B		MOV A,M	7E	
C08C		SBI 00(H)	DF,00	Subtract 0x00 from current byte with borrow
C08E		MOV M,A	77	Update value of current byte
C08F		DCR B	05	Indicates that current byte has been updated
C090		JZ LOOP7	CA,96,C0	If all bytes of current iteration are updated go to LOOP7
C093		JMP LOOP6	C3,8A,C0	If all bytes of current iteration are not updated , go to LOOP6
C096	LOOP7	LDA C409 (H)	3A,09,C4	
C099		SUI 02(H)	D6,02	Subtract 0x02 from max bit size
C09B		MOV B,A	47	Store max bit size – 2 in reg. B
C09C		LXI H FFEF (H)	21,EF,FF	Load memory with address of MSB
C09F	LOOP9	MOV A,M	7E	
C100		CPI 00(H)	FE,00	Compare current byte with 0x00
C102		JNZ LOOP8	C2,70,C0	If current byte is not 0x00 , go to LOOP8
C105		DCR B	05	Indicates that current byte has been found to be 0x00
C106		JZ LOOP10	CA,0D,C1	If first 15 bytes have been compared go to LOOP10
C109		INX H	23	If first 15 bytes have not yet been compared point memory to next MSB
C10A		JMP LOOP9	C3,9F,C0	
C10D	LOOP10	INX H	23	
C10E		MOV A,D	7A	
C10F		CMP M	BE	Compare higher order bits of n with data at address FFFE
C110		JZ LOOP11	CA,19,C1	If they are equal , go to LOOP 11
C113		JC LOOP8	DA,70,C0	If they are not equal and address at FFFE is greater than higher order bits of n, go to LOOP8
C116		JMP LOOP12	C3,2C,C1	Else if they are equal and address at FFFE is less than higher order bits of n, terminate the program.
C119	LOOP11	INX H	23	
C11A		MOV A,E	7B	
C11B		CMP M	BE	Compare lower order bits of n with data at address FFFF

	LABEL	MNEMONICS	HEX CODES	COMMENTS
C11C		JC LOOP8	DA,70,C0	If address at FFFF is less than the lower order bits of n, go to LOOP8
C11F		JNZ LOOP12	C2,2C,C1	Else if they are not equal, terminate the program
C122		MVI A,00(H)	3E,00	Else if they are equal, proceed
C124		STA FFFE (H)	32,FE,FF	Set value at address FFFE = 0x00
C127		MVI A,00(H)	3E,00	
C129		STA FFFF (H)	32,FF,FF	Set value at address FFFF = 0x00
C12C	LOOP12	HLT	76	

Note that since the $n = 5561$, maximum value of c from eq. 2.04 is 5560 which is a 2 byte data. Hence, after running the code above, the MSB and LSB of the cipher text will be stored in addresses FFEF and FFFF respectively. Now, these bytes will be transferred from the sender to the receiver. In this paper, this transmission is not discussed. However, we will implement the program to decrypt the cipher test at the receiver's end in the section 3.2.

3.2 Receiver's program

The algorithm used to decrypt the cipher text is exactly the same as that used to encrypt the message, the only difference being the constants over which the algorithm is performed. By comparing eq. 2.04 and 2.05, it becomes evident that substituting c in place of m and d in place of e , the code in section 3.1 can be easily modified to suit the receiver. In addition to this, it is also necessary to explicitly calculate $n = p.q$. This is because n is not provided in the private key P' . As section in section 2.4, atleast 15348 = 0x3BF4 bytes would be required for the arithmetic involved in eq. 2.05. We use the addresses C40C - C40B to store the byte values. We use also assume that the 2 bytes of the cipher text $c = 1413 = 0x0585$ (from section 2.4) that was sent from the sender is stored in the microprocessor of the receiver in addresses C404 (MSB) and C405 (LSB). The program is implemented below.

	LABEL	MNEMONICS	HEX CODES	COMMENTS
C000		MVI A,05(H)	3E,05	
C002		STA C404(H)	32,04,C4	Store higher order bits of $c = 0x05$
C005		MVI A,85(H)	3E,85	
C007		STA C405(H)	32,05,C4	Store lower order bits of $c = 0x85$
C00A		MVI A,26(H)	3E,26	
C00C		STA C406(H)	32,06,C4	Store higher order bits of $d = 0x26$
C00F		MVI A,8D(H)	3E,8D	
C011		STA C407(H)	32,07,C4	Store lower order bits of $d = 0x8D$
C014		MVI A,3B(H)	3E,3B	
C016		STA C408(H)	32,08,C4	Store higher order bits of max. byte size required = 0x3B
C019		MVI A,F4(H)	3E,F4	
C01B		STA C409(H)	32,09,C4	Store lower order bits of max. byte size required = 0xF4
C01E		MVI A,43(H)	3E,43	
C020		STA C40A(H)	32,0A,C4	Store $p = 67 = 0x43$
C023		MVI A,53(H)	3E,53	
C025		STA C40B(H)	32,0B,C4	Store $q = 83 = 0x53$
C028		LDA C40A(H)	3A,0A,C4	
C02B		MOV E,A	5F	Store p in reg. E
C02C		MVI D,00(H)	3E,00	Initialize reg. D = 0x00
C02E		LXI H,0000(H)	21,00,00	Set contents of HL reg. pair to 0x0000
C031		LDA C40B(H)	3A,0B,C4	
C034		MOV B,A	47	Load q to reg. B
C035	LOOP12	DAD D	19	Add DE reg. pair to HL reg. pair
C036		DCR B	05	
C037		JNZ LOOP12	D2,35,C0	If LOOP12 has not been executed q times repeat LOOP12
C03A		MOV A,H	7C	
C03B		STA C402(H)	32,02,C4	Store higher order bits of n in address C402
C03E		MOV A,L	7D	

	LABEL	MNEMONICS	HEX CODES	COMMENTS
C03F		STA C403(H)	32,03,C4	Store lower order bits of n in address C403
C042		LDA C408(H)	3A,08,C4	
C045		MOV D,A	57	Load higher order bits of no of max. bytes required in reg. D
C046		LDA C409(H)	3A,09,C4	
C049		MOV E,A	5F	Load lower order bits of no of max. bytes required in reg. E
C04A		LXI H C40C(H)	21,0C,C4	Load memory with address of MSB
C04D	LOOP1	MVI A,00(H)	3E,00	
C04F		MOV M,A	77	Initialize current byte to 0x00
C050		DCX D	1B	Decrement count of no. of byte
C051		MOV D,A	57	
C052		ORI 00(H)	F6,00	Check if MSB of count = 0x00
C054		JZ LOOP13	CA,5B,C0	If yes, go to LOOP13
C057		INX H	23	Load memory with address of next MSB
C058		JMP LOOP1	C3,4D,C0	And then go to LOOP1
C05B	LOOP13	MOV E,A	5F	
C05C		ORI 00(H)	F6,00	Check if LSB of count = 0x00
C05E		JZ LOOP2	CA,71,C0	If yes, go to LOOP2
C061		INX H	23	Load memory with address of next MSB
C062		JMP LOOP1	C3,4D,C0	And then go to LOOP1
C065		LDA C406(H)	3A,06,C4	
C068		STA C400(H)	32,00,C4	Store higher byte of d in address C400
C06B		LDA C407(H)	3A,07,C4	
C06E		STA C401(H)	32,01,C4	Store lower byte of d in address C401
C071	LOOP2	STC	37	
C072		CMC	3F	Reset Carry Flag
C073		LDA C408(H)	3A,08,C4	
C076		MOV B,A	47	Store higher byte of max byte size in reg. B
C077		LDA C409(H)	3A,09,C4	
C07A		MOV C,A	4F	Store lower byte of max byte size in reg. C
C07B	REPEAT1	LXI H 940C(H)	21,0C,94	Point memory to MSB of copy
C07E		LXI D C40C(H)	21,0C,C4	Point DE reg. pair to MSB of actual data
C081	REPEAT	LDAX D	1A	
C082		MOV M,A	77	Copy current byte
C083		DCX B	0B	
C084		MOV A,B	78	
C085		ORA C	B1	Are all bytes copied?
C086		JZ DONE	CA,8E,C0	If yes, go to DONE
C089		INX H	23	Else, point memory to next MSB of copy
C08A		INX D	13	And point DE reg. pair to next actual MSB
C08B		JMP REPEAT	C3,81,C0	And then, go to REPEAT
C08E	DONE	LDA C404(H)	3A,04,C4	
C091		STA C402(H)	32,02,C4	Store higher byte of c in address C402
C094		LDA C405(H)	3A,05,C4	
C097		STA C403(H)	32,03,C4	Store lower byte of c in address C403
C09A	LOOP4	LDA C408(H)	3A,08,C4	
C09D		MOV B,A	47	Store higher byte of max byte size in reg. B
C09E		LDA C409(H)	3A,09,C4	
C101		MOV C,A	4F	Store lower byte of max byte size in reg. C
C102		LXI H FFFF(H)	21,FF,FF	Point memory to LSB of actual data
C105		LXI D BFFF(H)	11,FF,BF	Point DE reg. pair to LSB of copy
C108		STC	37	
C109		CMC	3F	Reset carry flag
C10A	LOOP5	LDAX D	1A	
C10B		ADC M	8E	
C10C		MOV M,A	77	Add current copy byte to current actual byte with carry

	LABEL	MNEMONICS	HEX CODE	COMMENTS
C10D		DCX B	0B	
C10E		MOV A,B	78	
C10F		ORA C	B1	Are all bytes added?
C110		JZ NEXT	CA,18,C1	If yes, go to NEXT
C113		DCX H	2B	Else , point memory to next actual LSB
C114		DCX B	1B	And point DE reg pair to next copy LSB
C115		JMP LOOP5	C3,0A,C1	And then, go to LOOP5
C118	NEXT	LDA C402(H)	3A,02,C4	
C11B		MOV D,A	57	
C11C		LDA C403(H)	3A,03,C4	
C11F		MOV E,A	5F	D-E = count of c
C120		DCX D	1B	Decrement D-E reg. pair
C121		MOV A,D	7A	
C122		ORA E	B3	Is D-E = 0x0000?
C123		JZ LOOP3	CA,13,C1	If yes, go to LOOP3
C126		MOV A,D	7A	Else , count of c = new D-E
C127		STA C402(H)	32,02,C4	
C12A		MOV ,E	7B	
C12B		STA C403(H)	32,03,C4	
C12E		JMP LOOP4	CA,9A,C0	And then go to LOOP4
C131	LOOP3	LDA C400(H)	3A,00,C4	
C134		MOV D,A	57	
C135		LDA C401(H)	3A,01,C4	
C138		MOV E,A	5F	D-E = count of d
C139		DCX D	1B	Decrement count of d
C13A		MOV A,D	7A	
C13B		ORA E	B3	Is D-E = 0x0000?
C13C		JZ LOOP8	CA,4A,C1	If yes, go to LOOP8
C13F		MOV D,A	57	Else , update value of d – count
C140		STA C400(H)	32,00,C4	
C143		MOV E,A	5F	
C144		STA C401(H)	32,01,C4	
C147		JMP REPEAT1	C3,7B,C0	And then, go to REPEAT1
C14A	LOOP8	STC	37	
C14B		CMC	3F	Reset Carry flag
C14C		LDA C408(H)	3A,08,C4	
C14F		MOV B,A	47	Load MSB of max bit size in reg. B
C150		LDA C409(H)	3A,09,C4	
C153		MOV C,A	4F	Load LSB of max bit size in reg. C
C154		LDA C402(H)	3A,02,C4	
C157		MOV D,A	57	Load MSB of n in reg. D
C158		LDA C403(H)	3A,03,C4	
C15B		MOV E,A	5F	Load LSB of n in reg. E
C15C		LXI H FFFF(H)	21,FF,FF	Load address of LSB to memory
C15F		MOV A,M	7E	Load A with data that memory points to
C160		SUB E	93	Subtract LSB of n from this data
C161		MOV M,A	77	Update value of LSB
C162		DCX B	0B	Indicates that LSB has been updated
C163		DCX H	2B	Point memory to next LSB
C164		MOV A,M	7E	Load A with data that memory points to
C165		SBB D	9A	Subtract MSB of n from this data with borrow
C166		MOV M,A	77	Update value of second LSB
C167		DCX B	0B	Indicates that second LSB has been updated
C168	LOOP6	DCX H	2B	Point memory to next LSB
C169		MOV A,M	7E	
C16A		SBI 00(H)	DF,00	Subtract 0x00 from current byte with borrow

	LABEL	MNEMONICS	HEX CODE	COMMENTS
C16C		MOV M,A	77	
C16D		DCX B	0B	Indicates that current byte is updated
C16E		MOV A,B	78	
C16F		ORI 00(H)	F6,00	Check if MSB of count is 0x00
C171		JZ LOOP15	CA,77,C1	If yes, go to LOOP15
C174		JMP LOOP6	C3,68,C1	If not , go to LOOP6
C177	LOOP15	MOV A,C	79	
C178		ORI 00(H)	F6,00	Check if LSB of count is 0x00
C17A		JZ LOOP7	CA,80,C1	If yes, go to LOOP15
C17D		JMP LOOP6	C3,68,C1	If not , go to LOOP6
C180	LOOP7	LDA C408(H)	3A,08,C4	
C183		MOV B,A	47	Load MSB of max bit size to reg. B
C184		LDA C409(H)	3A,09,C4	
C187		MOV C,A	4F	Load LSB of max bit size to reg. C
C188		MOV A,C	79	
C189		SUI 02(H)	D6,02	Subtract 0x02 from LSB of max bit size
C18B		MOV C,A	4F	Update value of this LSB
C18C		MOV A,B	78	
C18D		SBI 00(H)	DF,00	Subtract 0x00 from MSB of max bit size with borrow
C18F		MOV B,A	47	Update value of this MSB
C190		LXI H C40C(H)	21,0C,C4	Load address with MSB of data bytes
C193	LOOP9	MOV A,M	7E	
C194		CPI 00(H)	FE,00	Compare current value of byte with 0x00
C196		JNZ LOOP8	C2,4A,C1	If not true, go to LOOP8
C199		DCX B	0B	Indicates that current byte is compared
C19A		MOV A,B	78	
C19B		ORI 00(H)	F6,00	Check if MSB of count is 0x00
C19D		JZ LOOP16	CA,04,C2	If yes, go to LOOP16
C200		INX H	23	Else if not , point memory to next MSB
C201		JMP LOOP9	C3,93,C1	And go to LOOP9
C204	LOOP16	MOV A,C	79	
C205		ORI 00(H)	F6,00	Check if LSB of count is 0x00
C207		JZ LOOP10	CA,0E,C2	If yes, go to LOOP10
C20A		INX H	23	Else if no, point memory to next MSB
C20B		JMP LOOP9	C3,93,C1	And go to LOOP9
C20E	LOOP10	INX H	23	
C20F		MOV A,D	7A	
C210		CMP M	BE	Compare value at second LSB with that of MSB of n
C211		JZ LOOP11	CA,1A,C2	If equal, go to LOOP11
C214		JC LOOP8	DA,4A,C1	Else if greater, go to LOOP8
C217		JMP LOOP12	C3,2D,C2	Else if lesser, terminate the program
C21A	LOOP11	INX H	23	
C21B		MOV A,E	7B	
C21C		CMP M	BE	Compare value LSB of n with that at address FFFF
C21D		JC LOOP8	DA,4A,C1	If lesser, go to LOOP8
C220		JNZ LOOP12	C2,2D,C2	Else If not equal, terminate the program
C223		MVI A,00(H)	3E,00	Else if equal, execute lines 157–161
C225		STA FFFE(H)	32,FE,FF	Store 0x00 at address FFFE
C228		MVI 00(H)	3E,00	
C22A		STA FFFF(H)	32,FF,FF	Store 0x00 at address FFFF
C22D	LOOP12	HLT	76	

After running the above code, the required message data *m* is stored at address locations FFFE (MSB) and FFFF (LSB).

4 Utilizing random primes for encrypting long data byte-wise

Till now, we considered the transmission of only one byte of data from the sender to the receiver. In practice however, this will not be the case. Usually multiple data bytes are send in one transmission. While using a single encryption key for all the bytes also works pretty well if the key size is quite big, the strength of encryption increases by leaps and bounds if random set of prime numbers p and q are chosen for every byte of data. For this, we'll have to generate random primes for encrypting each byte of data. In this section we will take following approach:

1. Store a set of known prime numbers in a certain array of memory locations.
2. Randomly choose two prime numbers to set as p and q respectively.
3. Choose a suitable value of e which is co-prime which $\phi(n) = (p-1)(q-1)$
4. Choose a suitable value for d from eq. 2.03.

In the previous example, we used $e = 17$. Consequently, we observed that at least 17 bytes were required to do the calculations in eq. 2.04 in the sender's microprocessor. Also p and q were chosen as 67 and 83 respectively. This meant that the cipher text would be atleast 15348 bytes long, depending on the message m transmitted. We wish to utilize the codes given in section 3.1 and 3.2 in cycles for encrypting each byte of message data m . But to do so, we should ensure that the number of bytes required for new values of p, q (which are randomly chosen), d and e (which are chosen on the basis of p and q) do no require more bytes for calculations than the ones that are already implemented. So, we impose that p and q should only be primes starting from 17 to 67. Also we impose that $e = 17$ The reasons for these are as follows:

1. The lower bound 17 is chosen because it is mandatory that n be greater than the message byte. Since the maximum decimal value of the message byte is 256, $\min(p) \min(q) > 256$. It turns out that the least value of a prime integer h such that $h^2 > 256$ is $h = 17$ and hence the value of lower bound is chosen to be 17.
2. In the example dealt with previously c was maximum 2 bytes long, this is because $n = 5561 = 0x15B9$ is 2 bytes long (see eq. 2.04). There we had chosen $p = 67$ and $q = 83$. So, for any prime numbers p and q such that $\max(p) = \max(q) = 67$ would imply that $\max(pq) < 5561$ and hence would still be at most 2 bytes long.
3. In the previous example e was chosen to be as 17. For the next example (which includes utilizing random prime numbers), we keep this value unchanged. The only restriction on the value of e is it should be co-prime with $\phi(n) = (p-1)(q-1)$. For $e = 17$, if this were not so, it would imply that atleast one of $(p-1)$ or $(q-1)$ is of the form $17l$ where $l \in \mathbb{Z}^+$ (because the only factor of 17 other than 1 is 17). This implies that either p or q (or both) is of the form $17l + 1$. Fortunately, primes of such form does not exists between our interval choice of 17 - 67. Hence, the choice $e = 17$ for any set $\{p, q\}$.
4. In the previous example d was chosen to be 9869. The equation solved for the same is eq. 2.03. It turns out that a general form of the solution for this equation is $x\phi(n) + y$ such that $x \in \mathbb{Z}^+ \cup \{0\}$ and $y \in \{1, 2, \dots, \phi(n) - 1\}$. For the next example (including random primes) we'll choose $x = 0$. This implies that $1 \leq d \leq \phi(n) - 1$. In the previous example $\phi(n)$ was equal to 5412 which is a 2 byte number. So, using this choice, d would also be of atmost 2 bytes (just the way it was in previous example: $d = 9869 = 0x268D$).
5. In the previous example, we saw in section 2.4, that $\max(c) = n - 1 = 5560$. For our next example, it will be $\max(n) - 1 = \max(p) \max(q) - 1 = 67 \times 67 - 1 = 4488$. Also $\max(d) = \max(\phi(n) - 1) = (\max(p) - 1)(\max(q) - 1) - 1 = (67 - 1)(67 - 1) - 1 = 4355$. This implies $\max(c^d) = \max(c)^{\max(d)} = 4488^{4355}$, which is less that the value 5560^{9869} obtained in the previous example. Hence, considering that the cipher text generated will be atleast 15348 bytes long is a safe assumption. Therefore, we can make use of the code written in section 3.2 directly in a cyclic manner to decrypt the long message data which includes multiple number of bytes.

4.1 Choosing random primes

As described above, the purpose of this section is to not generate random primes, but randomly choose a certain prime from a list of prime numbers. There are 13 primes numbers between 17 and 67 (both inclusive). So, we will use the following approach.

Figure 3: Algorithm for choosing random prime numbers

In the above algorithm, step two indicates that a random number has to be generated between 0 and 12. We will achieve this using the Linear Congruential Generator (LCG) of random numbers. LCG is an algorithm with generates a sequence of pseudo random numbers with following recurrence relation:

$$X_{n+1} = (AX_n + C) \bmod M \longrightarrow (\text{eq.4.01})$$

where X is the sequence of pseudo random numbers, A is called the multiplier, C is called the increment and M is called the modulus.

When the values of A , C and M are sufficiently large, this method will generate a good uniform distribution of pseudo random numbers in the range $\{0, 1, \dots, M - 1\}$. To restrict the numbers to the set $\{0, 1, \dots, 12\}$, we will take the remainder of the numbers thus generated upon division by the number 13, i.e.

$$I_n = X_n \bmod 13 \longrightarrow (\text{eq.4.02})$$

where I_n is the random number which would serve as the random index for our array of primes. Some choices of A , B and M is commonly used by various compilers for generating random numbers. For this project, we will use the choice made by Numerical Recipes i.e. $M = 2^{32} = 0x100000000$, $A = 1664525 = 0x19660D$ (three bytes) and $C = 1013904223 = 0x3C6EF35F$ (4 bytes). The flowchart for generating I_n is given below:

Figure 4: Flowchart for generating random numbers between 0-12 (inclusive)

In this example, X_n will always be a 4 byte data owing to the fact that the choice of $M = 2^{32} = 0x100000000$. Also A is chosen to be $1664525 = 0x19660D$ i.e. a 3 byte data. Therefore $A.X_n$ is atmost of 7 bytes. Also C is chosen to be $1013904223 = 0x3C6EF35F$ i.e. a 8 byte data. Hence, $A.X_n + C$ is atmost of 8 bytes. This is why 8 address locations have been allotted for storing the initial value of the seed (C3F4 - C3FB). It does not mean that the seed is of 8 bytes, this has been introduced just for ease of arithmetic. Initially the first four bytes will always be set to 0x00. The modulo with M of $A.X_n + C$ does not need to be explicitly computed, because the choice of M suggests that the result will be the last four bytes of the result $A.X_n + C$. The final result I_n is stored in address C3F3 (as shown in the third last step of the above flowchart). Thereafter, new value of seed X_{n+1} that is obtained is substituted in place of the initial seed X_n . For find the next seed, whole flowchart from step 4 to the end needs to be executed again.

4.2 8085 program for generating random numbers between 0-12 (inclusive)

Note that in the subsequent codes, we will not display the address of the program counter or the hex codes, because they are just sub-codes which will all be used together when we implement the final program in section 6. There are the necessary data will be given. Using the flowchart given in Figure 4, the program for generating random integers between 0 and 12 (both inclusive) is given below:

	LABEL	MNEMONICS	COMMENTS
1	STARTRAND	MVI A,19(H)	
2		STA C400(H)	Store MSB of A
3		MVI A,66(H)	
4		STA C401(H)	Store second byte of A
5		MVI A,0D(H)	
6		STA C402(H)	Store LSB of A
7		MVI A,3C(H)	
8		STA C3FC(H)	Store MSB of C
9		MVI A,6E(H)	
10		STA C3FD(H)	Store second MSB of C
11		MVI A,F3(H)	
12		STA C3FE(H)	Store second LSB of C
13		MVI A,5F(H)	
14		STA C3FF(H)	Store LSB of C
15		MVI C,08(H)	
16		LXI H C3F4(H)	
17	SET1	MVI A,00(H)	
18		MOV M,A	
19		DCR C	
20		JZ SET2	
21		INX H	
22		JMP SET1	Set initial seed = 0x00000000
23	SET2	LDA C3F8(H)	
24		STA C403(H)	Copy 5th byte of seed
25		LDA C3F9(H)	
26		STA C404(H)	Copy 6th byte of seed
27		LDA C3FA(H)	
28		STA C405(H)	Copy 7th byte of seed
29		LDA C3FB(H)	
30		STA C406(H)	Copy 8th byte (LSB) of seed
31	ANOTHER	LDA C400(H)	
32		MOV C,A	Store MSB of A in reg. C
33		LDA C401(H)	
34		MOV D,A	Store second byte of A in reg. D
35		LDA C402(H)	
36		MOV E,A	Store LSB of A in reg. E
37	LOOP14	LXI H C3FB(H)	Point memory to LSB of seed
38		LDA C406(H)	
39		ADD M	Add LSB of initial seed to current memory
40		MOV M,A	Update value of current byte
41		DCX H	Point memory to second LSB of seed
42		LDA C405(H)	
43		ADC M	Add second LSB of initial seed to memory with carry
44		MOV M,A	Update value of current byte
45		DCX H	Point memory to third LSB of seed
46		LDA C404(H)	
47		ADC M	Add third LSB of initial seed to memory with carry
48		MOV M,A	Update value of current byte
49		DCX H	Point memory to fourth LSB of seed
50		LDA C403(H)	
51		ADC M	Add fourth LSB of initial seed to memory with carry
52		MOV M,A	Update value of current byte
53		DCX H	Point memory to next LSB of seed
54		MVI B,04(H)	Initialize reg. B to 04(H)

	LABEL	MNEMONICS	COMMENTS
55	LOOP26	MOV A,M	
56		ACI 00(H)	Add 0x00 to current seed byte with carry
57		MOV M,A	Update value of current byte
58		DCR B	Indicates that current byte has been updated
59		JZ LOOP25	If all bytes of seed are updated, go to LOOP25
60		DCX H	Else , point memory to next LSB of seed
61		JMP LOOP26	And then go to LOOP26
62	LOOP25	MOV A,E	
63		SUI 01(H)	
64		MOV E,A	Subtract 0x01 from reg. E
65		MOV A,D	
66		SBI 00(H)	
67		MOV D,A	Subtract 0x00 from reg. D with borrow
68		MOV A,C	
69		SBI 00(H)	
70		MOV C,A	Subtract 0x00 from reg. C with borrow
71		MOV A,C	
72		ORA D	Is reg. C = reg. D = 0x00?
73		JZ LOOP13	If yes, go to LOOP13
74		JMP LOOP14	If not , go to LOOP14
75	LOOP13	MOV A,E	
76		CPI 00(H)	Is reg.E = 0x00?
77		JZ LOOP15	If yes, go to LOOP15
78		JMP LOOP14	If not , go to LOOP14
79	LOOP15	LXI H, C3FB(H)	Point memory to LSB of seed
80		LDA C3FF(H)	
81		ADD M	
82		MOV M,A	Add LSB of C to LSB of seed
83		DCX H	Point memory to second LSB of seed
84		LDA C3FE(H)	
85		ADC M	
86		MOV M,A	Add second LSB of C to second LSB of seed with carry
87		DCX H	Point memory to third LSB of seed
88		LDA C3FD	
89		ADC M	
90		MOV M,A	Add second LSB of C to second LSB of seed with carry
91		DCX H	Point memory to third LSB of seed
92		LDA C3FC	
93		ADC M	
94		MOV M,A	Add second LSB of C to third LSB of seed with carry
95		DCX H	Point memory to third fourth of seed
96		MVI B, 04(H)	Set reg. B = 0x04
97	LOOP17	ACI 00(H)	
98		MOV M,A	Add 0x00 to current byte of seed
99		DCR B	Indicates that current seed byte is updated
100		JZ LOOP16	If all seed bytes are updated go to LOOP16
101		DCX H	If not , point memory to next LSB of seed
102		JMP LOOP17	And then go to LOOP17
103	LOOP16	LDA C3F8(H)	
104		STA C403(H)	Store MSB of seed in address C403
105		LDA C3F9(H)	
106		STA C404(H)	Store second MSB of seed in address C404
107		LDA C3FA(H)	
108		STA C405(H)	Store second LSB of seed in address C405
109		LDA C3FB(H)	

	LABEL	MNEMONICS	COMMENTS
110		STA C406(H)	Store LSB of seed in address C406
111		MVI B, 0D(H)	Store 13 = 0x0D in reg. B
112	LOOP21	LXI H C3FB(H)	Point memory to LSB of seed
113		MOV A,M	
114		SUB B	
115		MOV M,A	Subtract reg. B from LSB of seed
116		DCX H	Go to next LSB of seed
117		MVI D,03(H)	Store 0x03 in reg. D
118	LOOP19	MOV A,M	
119		SBI 00(H)	
120		MOV M,A	Subtract 0x00 from current byte of seed with borrow
121		DCR D	Indicates that current seed byte is updated
122		JZ LOOP18	If all seed bytes are updated go to LOOP18
123		DCX H	If not , point memory to next LSB of seed byte
124		JMP LOOP19	And then go to LOOP19
125	LOOP18	LXI H, C3FB(H)	Point memory to MSB of 4- byte seed
126		MOV A,M	Load memory contents in accumulator
127		INX H	Point memory to second MSB of seed
128		ORA M	Are both MSB and second MSB of seed = 0x00?
129		JZ LOOP20	If yes, go to LOOP20
130		JMP LOOP21	If not , go to LOOP21
131	LOOP20	INX H	Point memory to second LSB of seed
132		MOV A,M	
133		CPI 00(H)	Compare second LSB of seed byte with 0x00
134		JZ LOOP22	If they are equal, go to LOOP22
135		JMP LOOP21	Else , go to LOOP21
136	LOOP22	INX H	Point memory of LSB of 4- byte seed
137		MOV A,M	Load this byte to accumulator
138		CMP B	Compare this byte with contents of reg. B
139		JZ LOOP23	If they are equal go to LOOP23
140		JC LOOP24	If byte is less than 0x0D, go to LOOP24
141		JMP LOOP21	Else , go to LOOP21
142	LOOP23	MVI A,00(H)	
143		MOV M,A	Set LSB of seed to 0x00
144	LOOP24	LDA C3FB(H)	
145		STA C3F3(H)	Store mod 13 random no. in address C3F3
146		MVI A,00(H)	
147		STA C3F4(H)	Set byte 1 of next initial seed to 0x00
148		MVI A,00(H)	
149		STA C3F5(H)	Set byte 2 of next initial seed to 0x00
150		MVI A,00(H)	
151		STA C3F6(H)	Set byte 3 of next initial seed to 0x00
152		MVI A,00(H)	
153		STA C3F7(H)	Set byte 4 of next initial seed to 0x00
154		LDA C403(H)	
155		STA C3F8(H)	Set byte 5 of next initial seed to MSB of the 4- byte new seed generated
156		LDA C404(H)	
157		STA C3F9(H)	Set byte 6 of next initial seed to second MSB of the 4- byte new seed generated
158		LDA C405(H)	
159		STA C3FA(H)	Set byte 7 of next initial seed to second LSB of the 4- byte new seed generated
160		LDA C406(H)	
161		STA C3FB(H)	Set byte 8 (LSB) of next initial seed to LSB of the 4- byte new seed generated
162		RET	

One run of the code above will create a random number between 0 and 12 (inclusive) in the address location C3F3. To generate another random number run the code from label ANOTHER to RET again.

4.3 8085 program for choosing random primes

As mentioned in the beginning of section 4, the approach we will take for choosing random primes in this paper is by first storing an array of prime numbers in some memory locations. Then generate a random number which will serve as the index in the list of prime numbers. The prime number whose index matches with the the random number generated (between 0-12) will be chosen for encryption. A pair of primes will be selected for every byte of data by utilizing the program given in section 4.2.

	LABEL	MNEMONICS	COMMENTS
1	CHOOSE	MVI A, 11 (H)	
2		STA FFD1 (H)	Store 17 = 0x11 in address FFD1
3		MVI A, 13 (H)	
4		STA FFD2 (H)	Store 19 = 0x13 in address FFD2
5		MVI A, 17 (H)	
6		STA FFD3 (H)	Store 23 = 0x17 in address FFD3
7		MVI A, 1D (H)	
8		STA FFD4 (H)	Store 29 = 0x1D in address FFD4
9		MVI A, 1F (H)	
10		STA FFD5 (H)	Store 31 = 0x1F in address FFD5
11		MVI A, 25 (H)	
12		STA FFD6 (H)	Store 37 = 0x25 in address FFD6
13		MVI A, 29 (H)	
14		STA FFD7 (H)	Store 41 = 0x29 in address FFD7
15		MVI A, 2B (H)	
16		STA FFD8 (H)	Store 43 = 0x2B in address FFD8
17		MVI A, 2F (H)	
18		STA FFD9 (H)	Store 47 = 0x2F in address FFD9
19		MVI A, 35 (H)	
20		STA FFDA (H)	Store 53 = 0x35 in address FFDA
21		MVI A, 3B (H)	
22		STA FFDB (H)	Store 59 = 0x3B in address FFDB
23		MVI A, 3D (H)	
24		STA FFDC (H)	Store 61 = 0x3D in address FFDC
25		MVI A, 43 (H)	
26		STA FFDD (H)	Store 67 = 0x43 in address FFDD
27		CALL STARTRAND	CALL STARTRAND function given in section 4.2
28		LDA C3F3 (H)	Load the random no. generated
29		ADI 01 (H)	Add 0x01 to this random no.
30		MOV C, A	And store it in reg. C
31		LXI H FFD0 (H)	Load address preceding that of first prime in HL reg. pair
32	NEXTPRIME	INX H	Go to next prime
33		DCR C	Have we reached the required index?
34		JNZ NEXTPRIME	If not , go to NEXTPRIME
35		MOV A, M	Else load current prime no. to accumulator
36		STA C40C (H)	And store it in C40C (this is p)
37		CALL ANOTHER	Go to ANOTHER label in 4.2 code
38		LDA C3F3 (H)	Do the same steps as done for p for q
39		ADI 01 (H)	
40		MOV C, A	
41		LXI H FFD0 (H)	
42		INX H	
43		DCR C	
44		JNZ NEXTPRIME	
45		MOV A, M	
46		STA C40D (H)	Store q in address C40D
47		RET	

5 Choose value of public and private key exponent for random primes

We have discussed how to set random values for p and q in section 4. Now, we'll have to set suitable values of d (private key exponent) and e (public key exponent). As mentioned in the beginning of section 4, we will impose $e = 17 = 0x11$ and d is some integer such that $1 \leq d \leq \phi(n) - 1$. The flowchart for deciding d is given below:

Figure 5: Flowchart for deciding private key exponent

Note that in this program, e is of 1 byte and d is of 2 bytes. Hence, the product $d.e$ is atmost 3 bytes. This is why the register triplet C-D-E is used for all the arithmetic calculations. The 8085 program of the above flowchart for $p = 23 = 0x17$ and $q = 47 = 0x2F$ is implemented below:

	LABEL	MNEMONICS	COMMENTS
1		MVI A,17(H)	
2		STA C40C(H)	Store p = 23 = 0x17 in address C40C
3		MVI A,2F(H)	
4		STA C40D(H)	Store q = 47 = 0x2F in address C40D
5		LDA C40C(H)	
6		SUI 01(H)	
7		MOV C,A	Store p-1 in reg. C
8		MVI B,00(H)	Store 0x00 in reg. B
9		LDA C40D(H)	
10		SUI 01(H)	
11		MOV E,A	Store q-1 in reg. E
12		MVI D,00(H)	Store 0x00 in reg. D
13		LXI H 0000(H)	H-L = 0x0000
14	LOOP28	DAD B	H-L = H-L + B-C
15		DCX D	Indicates that current iteration is done
16		MOV D,A	
17		ORA E	Is D-E = 0x0000?
18		JZ LOOP27	If yes, go to LOOP27
19		JMP LOOP28	Else, go to LOOP28
20	LOOP27	MOV A,H	
21		STA C40E(H)	Store MSB of totient in address C40E
22		MOV A,L	
23		STA C40F(H)	Store LSB of totient in address C40F
24		MVI A, 11(H)	
25		STA C408(H)	Store e in address C408
26		MVI A,00(H)	
27		STA C410(H)	Store MSB of d-count
28		MVI A,01(H)	
29		STA C411(H)	Store LSB of d-count
30	LOOP32	STC	
31		CMC	Reset Carry flag
32		MVI C,00(H)	
33		MVI D,00(H)	
34		MVI E,00(H)	C-D-E = 0x000000 (product)
35		LDA C408(H)	
36		MOV B,A	Store e in reg. B
37		LDA C410(H)	
38		STA C412(H)	Store copy of MSB of d-count in C412
39		LDA C411(H)	
40		STA C413(H)	Store copy of LSB of d-count in C413
41	LOOP30	MOV A,E	
42		ADD B	
43		MOV E,A	Add e to LSB of product
44		MOV A,D	
45		ACI 00(H)	
46		MOV D,A	Add 0x00 to second byte of product with carry
47		MOV A,C	
48		ACI 00(H)	
49		MOV C,A	Add 0x00 to MSB of product with carry
50		LDA C412(H)	
51		MOV H,A	Store MSB of d-count copy in H
52		LDA C413(H)	
53		MOV L,A	Store LSB of d-count copy in L
54		DCX H	Decrement d-count copy
55		MOV A,H	
56		ORA L	Is copy of d-count = 0x0000?
57		JZ LOOP29	If yes, go to LOOP29

	LABEL	MNEMONICS	COMMENTS
58		MOV A,H	Else , load MSB of decremented d-count copy to A
59		STA C412(H)	Store back in C412(H)
60		MOV A,L	Load LSB of decremented d-count copy to A
61		STA C413(H)	Store back in C413(H)
62		JMP LOOP30	And then , go to LOOP30
63	LOOP29	STC	
64		CMC	Reset Carry flag
65		LDA C40F(H)	
66		MOV B,A	Load LSB of totient to reg. B
67		MOV A,E	
68		SUB B	Subtract LSB of totient from LSB of product
69		MOV E,A	Update value of LSB of product
70		LDA C40E(H)	
71		MOV B,A	Load totient MSB to second product byte
72		MOV A,D	
73		SBB B	Subtract totient MSB from second product byte
74		MOV D,A	Update value of second product byte
75		MOV A,C	
76		SBI 00(H)	Subtract 0x00 from MSB of product with borrow
77		MOV C,A	Update value of MSB of product
78		MOV A,C	
79		CPI 00(H)	Is MSB of product 0x00?
80		JNZ LOOP29	If not , go to LOOP29
81		LDA C40E(H)	Load MSB of totient to accumulator
82		CMP D	Compare it with second product byte
83		JC LOOP29	If less , go to LOOP29
84		LDA C40F(H)	Else , load LSB of totient to accumulator
85		CMP E	Compare it with LSB of product
86		JC LOOP29	If less , go to LOOP29
87		JZ LOOP30	Else if , equal go to LOOP30
88		MOV A,E	Else if not equal, do lines 88–90
89		CPI 01(H)	Compare LSB of product with 0x01
90		JZ LOOP31	If equal , go to LOOP31
91	LOOP30	LDA C411(H)	Else if not equal, do lines 91–97
92		ADI 01(H)	Add 0x01 to LSB of d-count
93		STA C411(H)	Update LSB of d-count
94		LDA C410(H)	
95		ACI 00(H)	Ad 0x00 to MSB of d-count with carry
96		STA C410(H)	Update MSB of d-count
97		JMP LOOP32	And then , go to LOOP32
98	LOOP31	LDA C410(H)	
99		STA C406(H)	Store MSB of d-count in address C406
100		LDA C411(H)	
101		STA C407(H)	Store LSB of d-count in address C407
102		RET	

6 Final implementation of encryption for transferring 1kB of data

We now have all the sub-codes that are necessary to implement the transfer of multiple bytes of data with random pair of primes used for the encryption of every byte of data. In this section, we'll program the sender's multiprocessor to transfer a data of size 1kB to the receiver. To achieve this, codes in section 3.1, 4.2, and 4.3 needs to be applied in a cyclic form for each byte of data. For this all the 1000 message bytes needs to be stored as an array. We'll use memory locations FBE8 - FFCF for the same. Even the 1000 cipher texts (each of two bytes) that will be generated needs to be stored in a ciphertext array. We will utilize addresses F000 - F7CF for the same such that F000 and F001 contain the bytes of cipher text pertaining to first message byte, F002-F003 contains the second cipher text and so on. The flowchart for this implementation is given below:

Figure 6: Final flowchart for transferring 1kB data

Parts of the codes in 3.1, 4.2 and 4.3 needs to be applied in a specific pattern cyclically. The following program is the final implementation that needs to be done on the microprocessor in the receiver's end for encrypting 1kB of data byte-wise. Before execution of the program, it is assumed that the concerned 1000 bytes of data are stored sequentially in address locations FBE8-FFC4

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D000	CHOOSE	MVI A,11(H)	3E,11	
D002		STA FFD1(H)	32,D1,FF	Store 17 = 0x11 in address FFD1
D005		MVI A,13(H)	3E,13	
D007		STA FFD2(H)	32,D2,FF	Store 19 = 0x13 in address FFD2
D00A		MVI A,17(H)	3E,17	
D00C		STA FFD3(H)	32,D3,FF	Store 23 = 0x17 in address FFD3
D00F		MVI A,1D(H)	3E,1D	
D011		STA FFD4(H)	32,D4,FF	Store 29 = 0x1D in address FFD4
D014		MVI A,1F(H)	3E,1F	
D016		STA FFD5(H)	32,D5,FF	Store 31 = 0x1F in address FFD5
D019		MVI A,25(H)	3E,25	
D01B		STA FFD6(H)	32,D6,FF	Store 37 = 0x25 in address FFD6
D01E		MVI A,29(H)	3E,29	
D020		STA FFD7(H)	32,D7,FF	Store 41 = 0x29 in address FFD7
D023		MVI A,2B(H)	3E,2B	
D025		STA FFD8(H)	32,D8,FF	Store 43 = 0x2B in address FFD8
D028		MVI A,2F(H)	3E,2F	
D02A		STA FFD9(H)	32,D9,FF	Store 47 = 0x2F in address FFD9
D02D		MVI A,35(H)	3E,35	
D02F		STA FFDA(H)	32,DA,FF	Store 53 = 0x35 in address FFDA
D032		MVI A,3B(H)	3E,3B	
D034		STA FFDB(H)	32,DB,FF	Store 59 = 0x3B in address FFDB
D037		MVI A,3D(H)	3E,3D	
D039		STA FFDC(H)	32,DC,FF	Store 61 = 0x3D in address FFDC
D03C		MVI A,43(H)	3E,43	
D03E		STA FFDD(H)	32,DD,FF	Store 67 = 0x43 in address FFDD
D041		CALL STARTRAND	CD,62,D2	Call STARTRAND function
D044		MVI A,FB(H)	3E,FB	
D046		STA C414(H)	32,14,C4	Store MSB of 1st message byte address in C414
D049		MVI A,E8(H)	3E,E8	
D04B		STA C415(H)	32,15,C4	Store LSB of 1st message byte address in C415
D04E		MVI A,F0(H)	3E,F0	
		STA C416(H)		Store MSB of pointer to ciphertext array in C416
D050			32,16,C4	
D053		MVI A,00(H)	3E,00	
D055		STA C417(H)	32,17,C4	Store LSB of pointer to ciphertext array in C417
D058	START	LDA C414(H)	3A,14,C4	
D05B		MOV D,A	57	
D05C		LDA C415(H)	3A,15,C4	
D05F		MOV E,A	5F	D-E = first message byte
D060		LDAX D	1A	
D061		STA C407(H)	32,07,C4	Store current message byte in address C407
D064		MVI A,11(H)	3E,11	
D066		STA C408(H)	32,08,C4	Store e in address C408
D069		MVI A,11(H)	3E,11	
D06B		STA C409(H)	32,09,C4	Store max. byte size in address C409
D06E		CALL ANOTHER	CD,0D,D3	Call function ANOTHER
D071		LDA C3F3(H)	3A,F3,C3	Load the random no. generated
D074		ADI 01(H)	C6,01	Add 0x01 to this random no.
D076		MOV C,A	4F	And store it in reg. C
D077		LXI H FFD0(H)	21,D0,FF	Load address preceding that of first prime in HL reg. pair
D07A	NEXTPRIME	INX H	23	Go to next prime
D07B		DCR C	0D	Have we reached the required index?
D07C		JNZ NEXTPRIME	C2,7B,D0	If not, go to NEXTPRIME
D07F		MOV A,M	7E	Else load current prime no. to accumulator
D080		STA C40C(H)	32,0C,C4	And store it in C40C (this is p)

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D083		CALL ANOTHER	CD,0D,D3	Go to embedded function ANOTHER
D086		LDA C3F3(H)	3A,F3,C3	Do the same steps as done for p for q
D089		ADI 01(H)	C6,01	
D08B		MOV C,A	4F	
D08C		LXI H FFD0(H)	21,D0,FF	
D08F		INX H	23	
D090		DCR C	0D	
D091		JNZ NEXTPRIME	C2,7B,D0	
D094		MOV A,M	7E	
D095		STA C40D(H)	32,0D,C4	Store q in address C40D
D098		LDA C40C(H)	3A,0C,C4	
D09B		MOV E,A	5F	Store p in reg. E
D09C		MVI D,00(H)	16,00	Initialize reg. D = 0x00
D09E		LXI H,0000(H)	21,00,00	Set contents of HL reg. pair to 0x0000
D101		LDA C40D(H)	3A,0D,C4	
D104		MOV B,A	47	Load q to reg. B
D105	NXT	DAD D	19	Add DE reg. pair to HL reg. pair
D106		DCR B	05	
D107		JNZ NXT	C2,05,D1	If NXT has not been executed q times repeat NXT
D10A		MOV A,H	7C	
D10B		STA C40A(H)	32,0A,C4	Store higher order byte of n in address C40A
D10E		MOV A,L	7D	
D10F		STA C40B(H)	32,0B,C4	Store lower order byte of n in address C40B
D112		LDA C408(H)	3A,08,C4	
D115		MOV E,A	5F	Move e to reg. E
D116		LXI H FFEF(H)	21,EF,FF	Initialize reg. pair HL with address of MSB
D119	LOOP1	MVI A,00(H)	3E,00	
D11B		MOV M,A	77	Initialize data at memory location to 0x00
D11C		DCR E	1D	
D11D		JZ LOOP2	CA,24,D1	If all 17 address locations are initialized to 0x00, go to LOOP2
D120		INX H	23	
D121		JMP LOOP1	C3,19,D1	If all 17 address location are not initialized point memory next address byte & go to LOOP1
D124	LOOP2	STC	37	
D125		CMC	3F	Reset Carry flag
D126		LDA C408(H)	3A,08,C4	
D129		MOV E,A	5F	Load e to reg. E
D12A		LDA C409(H)	3A,09,C4	
D12D		MOV B,A	47	Store max byte size in reg. B
D12E	REPEAT1	LXI H FFDE(H)	21,DE,FF	H-L points to content at address FFDE
D131		LXI D FFEF(H)	11,EF,FF	D-E points to MSB which will converge to c
D134	REPEAT	LDAX D	1A	
D135		MOV M,A	77	Copy content of DE reg, pair to memory
D136		DCR B	05	Check if all 17 bytes are copied
D137		JZ DONE	CA,3F,D1	If yes, go to DONE
D13A		INX H	23	Else, increment contents of HL reg. pair
D13B		INX D	13	And increment contents of DE reg. pair
D13C		JMP REPEAT	C3,34,D1	And then go to REPEAT
D13F	DONE	LDA C407(H)	3A,07,C4	
D142		MOV C,A	4F	Store m in reg. C
D143	LOOP4	LDA C409(H)	3A,09,C4	
D146		MOV B,A	47	Store max byte size in reg. B
D147		LXI H FFFF(H)	21,FF,FF	Memory points to actual LSB
D14A		LXI D FFEE(H)	11,EE,FF	DE reg. pair points to copy LSB
D14D		STC	37	
D14E		CMC	3F	Reset carry flag

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D14F	LOOP5	LDAX D	1A	
D150		ADC M	8E	
D151		MOV M,A	77	Add copy LSB to actual LSB
D152		DCR B	05	Indicates that current byte is updated
D153	NEXT	JZ NEXT	CA,5B,D1	If all bytes are updated go to NEXT
D156		DCX H	2B	Else , point memory to next LSB
D157		DCX D	1B	Point DE reg. pair to next LSB of copy
D158		JMP LOOP5	C3,4F,D1	And go to LOOP5
D15B	NEXT	DCR C	0D	Indicates that current iteration is done
D15C		JZ LOOP3	CA,62,D1	If all m iterations are done, go to LOOP3
D15F		JMP LOOP4	C3,43,D1	If not, go to LOOP4
D162	LOOP3	DCR E	1D	Indicates that current iteration is done
D163		JZ LOOP8	CA,69,D1	If all e iterations are completed go to LOOP8
D166	LOOP8	JMP REPEAT1	C3,2E,D1	Else , go to REPEAT1
D169		STC	37	
D16A		CMC	3F	Reset Carry flag
D16B		LDA C409 (H)	3A,09,C4	
D16E		MOV B,A	47	Load max byte no. to reg. B
D16F		LDA C40A (H)	3A,0A,C4	
D172		MOV D,A	57	Load higher order bit of n to reg. D
D173		LDA C40B (H)	3A,0B,C4	
D176		MOV E,A	5F	Load lower order bit of n to reg. E
D177		LXI H,FFFF (H)	21,FF,FF	Initialize reg. pair HL with address of LSB
D17A		MOV A,M	7E	
D17B		SUB E	93	Subtract lower order bit of n from LSB
D17C		MOV M,A	77	Update value of LSB
D17D		DCR B	05	Indicates that LSB is updated
D17E		DCX H	2B	Point memory to next LSB
D17F		MOV A,M	7E	
D180		SBB D	9A	Subtract higher order bit of n from second LSB with borrow
D181		MOV M,A	77	Update value of second LSB
D182		DCR B	05	Indicates that second LSB is updated
D183		DCX H	2B	Point memory to next LSB
D184	LOOP6	MOV A,M	7E	
D185		SBI 00(H)	DF,00	Subtract 0x00 from current byte with borrow
D187		MOV M,A	77	Update value of current byte
D188		DCR B	05	Indicates that current byte has been updated
D189		JZ LOOP7	CA,8F,D1	If all bytes of current iteration are updated go to LOOP7
D18C		JMP LOOP6	C3,83,D1	If all bytes of current iteration are not updated, go to LOOP6
D18F	LOOP7	LDA C409 (H)	3A,09,C4	
D192		SUI 02(H)	D6,02	Subtract 0x02 from max bit size
D194		MOV B,A	47	Store max bit size – 2 in reg. B
D195	LOOP9	LXI H FFEF (H)	21,EF,FF	Load memory with address of MSB
D198		MOV A,M	7E	
D199		CPI 00(H)	FE,00	Compare current byte with 0x00
D19B		JNZ LOOP8	C2,69,D1	If current byte is not 0x00, go to LOOP8
D19E		DCR B	05	Indicates that current byte has been found to be 0x00
D19F		JZ LOOP10	CA,05,D2	If first 15 bytes have been compared go to LOOP10
D202		INX H	23	If first 15 bytes have not yet been compared point memory to next MSB
D203	LOOP10	JMP LOOP9	C3,98,D1	
D205		INX H	23	
D206		MOV A,D	7A	

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D207		CMP M	BE	Compare higher order bits of n with data at address FFFE
D208		JZ LOOP11	CA,11,D2	If they are equal, go to LOOP 11
D20B		JC LOOP8	DA,69,D1	If they are not equal and address at FFFE is greater than higher order bits of n, go to LOOP8
D20E		JMP LOOP12	C3,24,D2	Else if they are equal and address at FFFE is less than higher byte of n, go to LOOP12
D211	LOOP11	INX H	23	
D212		MOV A,E	7B	
D213		CMP M	BE	Compare lower order bits of n with data at address FFFF
D214		JC LOOP8	DA,69,D1	If address at FFFF is less than the lower order bits of n, go to LOOP8
D217		JNZ LOOP12	C2,24,D2	Else if they are not equal, go to LOOP12
D21A		MVI A,00(H)	3E,00	Else if they are equal, proceed
D21C		STA FFFE(H)	32,FE,FF	Set value at address FFFE = 0x00
D21F		MVI A,00(H)	3E,00	
D221		STA FFFF(H)	32,FF,FF	Set value at address FFFF = 0x00
D224	LOOP12	LDA C416(H)	3A,16,C4	Load MSB of ciphertext array pointer address to reg. D
D227		MOV D,A	57	
D228		LDA C417(H)	3A,17,C4	Load LSB of ciphertext array pointer address to reg. E
D22B		MOV E,A	5F	Load MSB of ciphertext generated to A
D22C		LDA FFFE(H)	21,FE,FF	And store in ciphertext array pointer
D22F		STAX D	12	Increment pointer to point to next address
D230		INX D	13	Load LSB of ciphertext generated to A
D231		LDA FFFF(H)	21,FFFF	And store in ciphertext array pointer
D234		STAX D	12	Increment pointer to point to next address
D235		INX D	13	
D236		MOV A,D	7A	Store MSB of incremented pointer back to C416
D237		STA C416(H)	32,16,C4	
D23A		MOV A,E	7B	Store LSB of incremented pointer back to C417
D23B		STA C417(H)	32,17,C4	
D23E		LDA C414(H)	3A,14,C4	
D241		MOV D,A	57	
D242		LDA C415(H)	3A,15,C4	D-E = pointer of message byte array
D245		MOV E,A	5F	Current byte has been encrypted
D246		INX D	13	
D247		MOV A,D	7A	Check if MSB of incremented pointer = FF
D248		CPI FF(H)	FE,FF	If yes, go to CHECK2
D24A		JZ CHECK2	CA,58,D2	Else , proceed
D24D	AGAIN	MOV A,D	7A	Store MSB of incremented pointer back to C414
D24E		STA C414(H)	32,14,C4	
D251		MOV A,E	7B	Store LSB of incremented pointer back to C415
D252		STA C415(H)	32,15,C4	And then, go to START
D255		JMP START	C3,58,D0	
D258	CHECK2	MOV A,E	78	Check if LSB of incremented pointer = D0
D259		CPI D0(H)	FE,D0	If yes, terminate the program
D25B		JZ STOP	CA,61,D2	Else , go to AGAIN
D25E		JMP AGAIN	C3,4D,D2	
D261	STOP	HLT	76	

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D262	STARTRAND	MVI A,19(H)	3E,19	
D264		STA C400(H)	32,00,C4	Store MSB of A
D267		MVI A,66(H)	3E,66	
D269		STA C401(H)	32,01,C4	Store second byte of A
D26C		MVI A,0D(H)	3E,0D	
D26E		STA C402(H)	32,02,C4	Store LSB of A
D271		MVI A,3C(H)	3E,3C	
D273		STA C3FC(H)	32,FC,C3	Store MSB of C
D276		MVI A,6E(H)	3E,6E	
D278		STA C3FD(H)	32,FD,C3	Store second MSB of C
D27B		MVI A,F3(H)	3E,F3	
D27D		STA C3FE(H)	32,FE,C3	Store second LSB of C
D280		MVI A,5F(H)	3E,5F	
D282		STA C3FF(H)	32,FF,C3	Store LSB of C
D285		MVI C,08(H)	0E,08	
D287		LXI H C3F4(H)	21,F4,C3	
D28A	SET1	MVI A,00(H)	3E,00	
D28C		MOV M,A	77	
D28D		DCR C	0D	
D28E		JZ SET2	CA,95,D2	
D291		INX H	23	
D292		JMP SET1	C3,8A,D2	Set initial seed = 0x00000000
D295		LDA C3F8(H)	3A,F8,C3	
D298		STA C403(H)	32,03,C4	Copy 5th byte of seed
D29B		LDA C3F9(H)	3A,F9,C3	
D29E		STA C404(H)	32,04,C4	Copy 6th byte of seed
D301		LDA C3FA(H)	3A,FA,C3	
D304		STA C405(H)	32,05,C4	Copy 7th byte of seed
D307		LDA C3FB(H)	3A,FB,C3	
D30A		STA C406(H)	32,06,C4	Copy 8th byte (LSB) of seed
D30D		LDA C400(H)	3A,00,C4	
D310		MOV C,A	4F	Store MSB of A in reg. C
D311		LDA C401(H)	3A,01,C4	
D314		MOV D,A	57	Store second byte of A in reg. D
D315		LDA C402(H)	3A,02,C4	
D318		MOV E,A	5F	Store LSB of A in reg. E
D319	LOOP14	LXI H C3FB(H)	21,FB,C3	Point memory to LSB of seed
D31C		LDA C406(H)	3A,06,C4	
D31F		ADD M	86	Add LSB of initial seed to current memory
D320		MOV M,A	77	Update value of current byte
D321		DCX H	2B	Point memory to second LSB of seed
D322		LDA C405(H)	3A,05,C4	
D325		ADC M	8E	Add second LSB of initial seed to memory with carry
D326		MOV M,A	77	Update value of current byte
D327		DCX H	2B	Point memory to third LSB of seed
D328		LDA C404(H)	3A,04,C4	
D32B		ADC M	8E	Add third LSB of initial seed to memory with carry
D32C		MOV M,A	77	Update value of current byte
D32D		DCX H	2B	Point memory to fourth LSB of seed
D32E		LDA C403(H)	3A,03,C4	
D331		ADC M	8E	Add fourth LSB of initial seed to memory with carry
D332		MOV M,A	77	Update value of current byte
D333		DCX H	2B	Point memory to next LSB of seed
D334		MVI B,04(H)	06,04	Initialize reg. B to 04(H)

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D336	LOOP26	MOV A,M	7E	
D337		ACI 00(H)	CF,00	Add 0x00 to current seed byte with carry
D339		MOV M,A	77	Update value of current byte
D33A		DCR B	05	Indicates that current byte has been updated
D33B		JZ LOOP25	CA,42,D3	If all bytes of seed are updated, go to LOOP25
D33E	LOOP25	DCX H	2B	Else , point memory to next LSB of seed
D33F		JMP LOOP26	C3,36,D3	And then go to LOOP26
D342		MOV A,E	7B	
D343		SUI 01(H)	D6,01	
D345		MOV E,A	5F	Subtract 0x01 from reg. E
D346	LOOP13	MOV A,D	7A	
D347		SBI 00(H)	DF,00	
D349		MOV D,A	57	Subtract 0x00 from reg. D with borrow
D34A		MOV A,C	79	
D34B		SBI 00(H)	DF,00	
D34D	LOOP15	MOV C,A	4F	Subtract 0x00 from reg. C with borrow
D34E		MOV A,C	79	
D34F		ORA D	B2	Is reg. C = reg. D = 0x00?
D350		JZ LOOP13	CA,56,D3	If yes, go to LOOP13
D353		JMP LOOP14	C3,19,D3	If not , go to LOOP14
D356	LOOP13	MOV A,E	7B	
D357		CPI 00(H)	FE,00	Is reg.E = 0x00?
D359		JZ LOOP15	CA,5F,D3	If yes, go to LOOP15
D35C		JMP LOOP14	C3,19,D3	If not , go to LOOP14
D35F		LXI H, C3FB(H)	21,FB,C3	Point memory to LSB of seed
D362	LOOP15	LDA C3FF(H)	3A,FF,C3	
D365		ADD M	86	
D366		MOV M,A	77	Add LSB of C to LSB of seed
D367		DCX H	2B	Point memory to second LSB of seed
D368		LDA C3FE(H)	3A,FE,C3	
D36B	LOOP16	ADC M	8E	
D36C		MOV M,A	77	Add second LSB of C to second LSB of seed with carry
D36D		DCX H	2B	Point memory to third LSB of seed
D36E		LDA C3FD	3A,FD,C3	
D371		ADC M	8E	
D372	LOOP17	MOV M,A	77	Add second LSB of C to second LSB of seed with carry
D373		DCX H	0B	Point memory to third LSB of seed
D374		LDA C3FC	3A,FC,C3	
D377		ADC M	8E	
D378		MOV M,A	77	Add second LSB of C to third LSB of seed with carry
D379	LOOP17	DCX H	0B	Point memory to third fourth of seed
D37A		MVI B, 04(H)	06,04	Set reg. B = 0x04
D37C		ACI 00(H)	CF,00	
D37E		MOV M,A	77	Add 0x00 to current byte of seed
D37F		DCR B	05	Indicates that current seed byte is updated
D380	LOOP16	JZ LOOP16	CA,87,D3	If all seed bytes are updated go to LOOP16
D383		DCX H	2B	If not , point memory to next LSB of seed
D384		JMP LOOP17	C3,7C,D3	And then go to LOOP17
D387		LDA C3F8(H)	3A,F8,C3	
D38A		STA C403(H)	32,03,C4	Store MSB of seed in address C403
D38D	LOOP16	LDA C3F9(H)	3A,F9,C3	
D390		STA C404(H)	32,04,C4	Store second MSB of seed in address C404
D393		LDA C3FA(H)	3A,FA,C3	
D396		STA C405(H)	32,05,C4	Store second LSB of seed in address C405
D399		LDA C3FB(H)	3A,FB,C3	

	LABEL	MNEMONICS	HEX CODE	COMMENTS
D39C		STA C406(H)	32,06,C4	Store LSB of seed in address C406
D39F		MVI B,0D(H)	06,0D	Store 13 = 0x0D in reg. B
D401	LOOP21	LXI H C3FB(H)	21,FB,C3	Point memory to LSB of seed
D404		MOV A,M	7E	
D405		SUB B	90	
D406		MOV M,A	77	Subtract reg. B from LSB of seed
D407		DCX H	2B	Go to next LSB of seed
D408		MVI D,03(H)	16,03	Store 0x03 in reg. D
D40A	LOOP19	MOV A,M	7E	
D40B		SBI 00(H)	DF,00	
D40D		MOV M,A	77	Subtract 0x00 from current byte of seed with borrow
D40E		DCR D	15	Indicates that current seed byte is updated
D40F		JZ LOOP18	CA,16,D4	If all seed bytes are updated go to LOOP18
D412		DCX H	2B	If not , point memory to next LSB of seed byte
D413		JMP LOOP19	C3,0A,D4	And then go to LOOP19
D416	LOOP18	LXI H, C3FB(H)	21,FB,C3	Point memory to MSB of 4- byte seed
D419		MOV A,M	7E	Load memory contents in accumulator
D41A		INX H	23	Point memory to second MSB of seed
D41B		ORA M	B6	Are both MSB and second MSB of seed = 0x00?
D41C		JZ LOOP20	CA,22,D4	If yes, go to LOOP20
D41F		JMP LOOP21	C3,01,D4	If not , go to LOOP21
D422	LOOP20	INX H	23	Point memory to second LSB of seed
D423		MOV A,M	7E	
D424		CPI 00(H)	FE,00	Compare second LSB of seed byte with 0x00
D426		JZ LOOP22	CA,2C,D4	If they are equal, go to LOOP22
D429		JMP LOOP21	C3,01,D4	Else , go to LOOP21
D42C	LOOP22	INX H	23	Point memory of LSB of 4- byte seed
D42D		MOV A,M	7E	Load this byte to accumulator
D42E		CMP B	B8	Compare this byte with contents of reg. B
D42F		JZ LOOP23	CA,38,D4	If they are equal go to LOOP23
D432		JC LOOP24	DA,3B,D4	If byte is less than 0x0D, go to LOOP24
D435		JMP LOOP21	C3,01,D4	Else , go to LOOP21
D438	LOOP23	MVI A,00(H)	3E,00	
D43A		MOV M,A	77	Set LSB of seed to 0x00
D43B	LOOP24	LDA C3FB(H)	3A,FB,C3	
D43E		STA C3F3(H)	32,F3,C3	Store mod 13 random no. in address C3F3
D441		MVI A,00(H)	3E,00	
D443		STA C3F4(H)	32,F4,C3	Set byte 1 of next initial seed to 0x00
D446		MVI A,00(H)	3E,00	
D448		STA C3F5(H)	32,F5,C3	Set byte 2 of next initial seed to 0x00
D44B		MVI A,00(H)	3E,00	
D44D		STA C3F6(H)	32,F6,C3	Set byte 3 of next initial seed to 0x00
D450		MVI A,00(H)	3E,00	
D452		STA C3F7(H)	32,F7,C3	Set byte 4 of next initial seed to 0x00
D455		LDA C403(H)	3A,03,C4	
D458		STA C3F8(H)	32,F8,C3	Set byte 5 of next initial seed to MSB of the 4- byte new seed generated
D45B		LDA C404(H)	3A,04,C4	
D45E		STA C3F9(H)	32,F9,C3	Set byte 6 of next initial seed to second MSB of the 4- byte new seed generated
D461				
D464		LDA C405(H)	3A,05,C4	
D467		STA C3FA(H)	32,FA,C3	Set byte 7 of next initial seed to second LSB of the 4- byte new seed generated
D46A				
		LDA C406(H)	3A,06,C4	
D46D		STA C3FB(H)	32,FB,C3	Set byte 8 (LSB) of next initial seed to LSB of the 4- byte new seed generated
		RET	C9	

7 Conclusion

Hence, as discussed in section 6, 1kB of data have been encrypted byte-wise with random pair of prime numbers being used as initial conditions for the same. After running this program, the message bytes which are stored in addresses FBE8 - FFCF are encrypted and the respective ciphertext bytes are stored in address locations F000 - F7CF. F000 and F001 contain the MSB and LSB of the first ciphertext respectively, F002 and F003 contain those of the second ciphertext and so on. In a similar fashion, the decryption of the received bytes can be performed cyclically at the receiver's end utilizing codes in section 3.2 and 5. Now, the private key will contain all the pairs of prime numbers p and q used for encryption and also the respective private key exponent (d) values i.e.

$P' = \{p_1, p_2, \dots, q_1, q_2, \dots, d_1, d_2, \dots\}$ where the pair $\{p_i, q_i\}$ is the pair of primes used to encrypt i^{th} byte of message data and d_i is the corresponding private key exponent. Similarly, the public key will contain the common public key exponent and the products of every set of prime numbers used in encryption i.e $P = \{e, n_1, n_2, \dots\}$ where $n_i = p_i q_i$.

References

- [1] Evgeny Milanov. *The RSA Algorithm*. (2009) Available: https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf
- [2] Cesar Alison Monteiro Paixão. *An efficient variant of the RSA cryptosystem*. (2003) (preprint). Available: <https://eprint.iacr.org/2003/159.pdf>
- [3] Seema Verma, Deepak Garg. *An Improved RSA Variant*. International Journal of Advancements in Technology. ISSN 0976-4860
- [4] Mihir Bellare, Shafi Goldwassery and Daniele Micciancio. "Pseudo-Random" Number Generation within Cryptographic Algorithms: the DSS Case. Appears in Advances in Cryptology - Crypto 97 Proceedings, Lecture Notes in Computer Science Vol. 1294, B. Kaliski ed., Springer-Verlag, 1997.