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 805 ............. 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 one 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 \( n = p \cdot q \rightarrow \text{(eq. 2.01)} \).
• Step 3. Compute totient \( \phi(n) = (p - 1)(q - 1) \rightarrow \text{(eq. 2.02)} \).
• 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: \( de \equiv 1 \pmod{\phi(n)} \rightarrow \text{(eq. 2.03)} \).

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:

\[
    c = m^e \pmod{n} \quad \text{(eq.2.04)}
\]

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:

\[
    m = c^d \pmod{pq} \quad \text{(eq.2.05)}
\]

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 at least 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 at least 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 stored in address FFFF. The calculation of \(m^e\) is done on the following understanding:

\[
m^e = m \cdot m^{e-1} = m^{e-1} + m^{e-1} + \cdots + m^{e-1} \\
= \underbrace{(m^{e-2} + \cdots + m^{e-2}) + \cdots + (m^{e-2} + \cdots + m^{e-2})}_{m \text{ times}} + \cdots \\
= \underbrace{(m + m + \cdots + m) + (m + m + \cdots + m) + \cdots + (m + m + \cdots + m)}_{e-1 \text{ times}} + \cdots \\
\]

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

Initialize contents of address locations FFEF - FFFF to 00

Set E = e

Copy values of FFEF - FFFF in FFDE - FFEE

Set C = m

Set B = Max. no. of bytes required

H-L = FFFF and D-E = FFEE

Reset carry flag

Load contents pointed by D-E to accumulator

Add contents of memory to accumulator with carry

Set value of content at address H-L equal to accumulator

Decrement B

Is Zero flag set?

Decrement C

Is Zero flag set?

Decrement E

Is Zero flag set?

Stop

H-L = H-L - 1 and D-E = D-E - 1

Figure 1: Flowchart for calculation of $m^e$
Figure 2: Flowchart for calculation of $m^e \mod n$
| LABEL | MNEMONICS | HEX CODES | COMMENTS |
|-------|-----------|-----------|----------|
| C00   | MVI A, 67(H) | 3E,67 | Store m = 103 = 0x67 |
| C002  | STA C407(H) | 32,07,C4 | |
| C005  | MVI A, 11(H) | 3E,11 | Store e = 17 = 0x11 |
| C007  | STA C408(H) | 32,08,C4 | |
| 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 = 0x0B |
| C019  | LDA C408(H) | 3A,08,C4 | |
| C01C  | MOV E, A | 5F | Move e to reg. E |
| C01D  | LXH H, FFEF(H) | 21,EF,FF | Initialize reg. pair HL with address of MSB |
| C020  | LOOP1 | | |
| 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 | If all 17 address location are not initialized point memory next address byte & go to LOOP1 |
| C028  | JMP LOOP1 | C3,20,C0 | Reset Carry flag |
| C02B  | LOOP2 | | |
| C02C  | STC | 37 | |
| C02D  | LDA C408(H) | 3A,08,C4 | |
| C030  | MOV E, A | 5F | |
| C031  | LDA C409(H) | 3A,09,C4 | |
| C034  | MOV B, A | 47 | |
| C035  | REPEAT1 | | |
| C037  | LXH 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 | | |
| 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 | An then go to REPEAT |
| C046  | DONE | | |
| C047  | LDA C407(H) | 3A,07,C4 | Store m in reg. C |
| C04A  | LOOP4 | | |
| 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 | Reset carry flag |
| C055  | CMC | 3F | |
| C056  | LOOP5 | | |
| C057  | LDA D | 1A | |
| C058  | MOV M, A | 77 | |
| C059  | DCR B | 05 | Add copy LSB to actual LSB |
| C05A  | JZ NEXT | CA,62,C0 | Indicates that current byte is updated |
| C05D  | DCX H | 2B | If all bytes are updated go to NEXT |
| C05E  | DCX D | 1B | Else, point memory to next LSB |
| C05F  | JMP LOOP5 | C3,56,C0 | Point DE reg. pair to next LSB of copy |
| C062  | NEXT | | |
| C063  | DCX C | 0D | Indicates that current iteration is done |
| C066  | JMP LOOP4 | C3,4A,C0 | If all m iterations are done, go to LOOP3 |
| C069  | LOOP3 | | |
| C06A  | JZ LOOP8 | CA,70,C0 | If not, go to LOOP4 |
| C06E  | DCR E | 1D | Indicates that current iteration is done |
| C06F  | JZ LOOP8 | CA,70,C0 | If all e iterations are completed go to LOOP8 |
| LABEL | MNEMONICS | HEX CODES | COMMENTS |
|-------|-----------|-----------|----------|
| C06D  | JMP       | C3,35,C0  | Else, go to REPEAT1 |
| LOOP8 | REPEAT1   |           |          |
| C070  | STC       | 37        | Reset Carry flag |
| C071  | CMC       | 3F        |          |
| C072  | LDA       | 3A,09,C4  | Load max byte no. to reg. B |
| C075  | MOV       | 47        |          |
| C076  | LDA       | 3A,0A,C4  | Load higher order bit of n to reg. D |
| C079  | MOV       | 57        |          |
| C07A  | LDA       | 3A,0B,C4  | Load lower order bit of n to reg. E |
| C07D  | MOV       | 5F        | Initialize reg. pair HL with address of LSB |
| C07E  | LXI       | 21,FF,FF  |          |
| C081  | MOV       | 7E        |          |
|       | A,M       |           | Subtract lower order bit of n from LSB |
| C082  | SUB       | 93        |          |
| C083  | MOV       | 77        | Update value of LSB |
|       | M,A       |           | Indicates that LSB is updated |
| C084  | DCR       | 05        |          |
| C085  | DCX       | 2B        |          |
| C086  | MOV       | 7E        | Point memory to next LSB |
|       | A,M       |           | Subtract higher order bit of n from second LSB with borrow |
| C087  | SUBB      | 9A        |          |
|       | D         |           | Update value of second LSB |
| C088  | MOV       | 77        |          |
| C089  | DCR       | 05        | Indicates that second LSB is updated |
| C090  | DCX       | 2B        |          |
|       | H         |           | Point memory to next LSB |
| C08A  | MOV       | 7E        | Subtract 0x00 from current byte with borrow |
| LOOP6 | A,M       |           | Update value of current byte |
| C08B  | MOV       | 7E        | Indicates that current byte has been updated |
|       | A,M       |           | If all bytes of current iteration are updated go to LOOP7 |
| C08C  | SBI       | DF,00     |           |
|       | 0(H)      |           | Subtract 0x00 from current byte with borrow |
| C08E  | MOV       | 77        |          |
|       | M,A       |           | Update value of current byte |
| C08F  | DCR       | 05        | Indicates that current byte has been updated |
| C090  | JZ        | CA,96,C0  | If all bytes of current iteration are not updated, go to LOOP6 |
|       | LOOP7     | C3,8A,C0  |           |
| C096  | LDA       | 3A,09,C4  | Subtract 0x02 from max bit size |
| LOOP7 | C409(H)   |           | Store max bit size − 2 in reg. B |
| C099  | SUI       | D6,02     | Load memory with address of MSB |
| C09B  | MOV       | 47        | Compare current byte with 0x00 |
|       | B,A       |           | If current byte is not 0x00, go to LOOP8 |
| C09C  | LXI       | 21,EF,FF  | Indicates that current byte has been found to be 0x00 |
| LOOP9 | HFF(H)    |           | If first 15 bytes have been compared go to LOOP10 |
| C09F  | MOV       | 7E        | If first 15 bytes have not yet been compared point memory to next MSB |
| C100  | CPI       | FE,00     | Compare higher order bits of n with data at address FFFE |
| C102  | JNZ       | C2,70,C0  | If they are equal, go to LOOP 11 |
|       | LOOP8     |           | If they are not equal and address at FFFE is greater than higher order bits of n, go to LOOP8 |
| C105  | DCR       | 05        | Else if they are equal and address at FFFE is less than higher order bits of n, terminate the program. |
| C106  | JZ        | CA,0D,C1  | Compare lower order bits of n with data at address FFFF |
| C109  | INX       | 23        |          |
|       | H         |           |          |
| C10A  | JMP       | C3,9F,C0  |          |
| LOOP9 | LOOP10    |           |          |
| C10D  | INX       | 23        |          |
| LOOP10| H         |           |          |
| C10E  | MOV       | 7A        |          |
| C10F  | CMP       | BE        |          |
|       | B         |           |          |
| C110  | JMP       | C3,2C,C1  |          |
| LOOP12| LOOP11    |           |          |
| C111  | INX       | 23        |          |
|       | H         |           |          |
| C11A  | MOV       | 7B        |          |
| C11B  | CMP       | BE        |          |
### 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, at least 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    | Store higher order bits of \( c = 0x05 \) |
| C002  | STA C404(H) | 32,04,C4 | Store lower order bits of \( c = 0x85 \) |
| C005  | MVI A, 85(H) | 3E,85    | Store higher order bits of \( d = 0x26 \) |
| C007  | STA C405(H) | 32,06,C4 | Store lower order bits of \( d = 0x8D \) |
| C00C  | MVI A, 26(H) | 3E,26    | Store higher order bits of max. byte size required = 0x3B |
| C00F  | STA C406(H) | 32,07,C4 | |
| C011  | MVI A, 8D(H) | 3E,8D    | |
| C014  | STA C407(H) | 32,08,C4 | |
| C016  | MVI A, 3B(H) | 3E,3B    | |
| C019  | MVI A, F4(H) | 3E,F4    | Store lower order bits of max. byte size required = 0xF4 |
| C01B  | STA C409(H) | 32,09,C4 | |
| C01E  | MVI A, 43(H) | 3E,43    | Store \( p = 67 = 0x43 \) |
| C020  | STA C40A(H) | 32,0A,C4 | Store \( q = 83 = 0x53 \) |
| C023  | MVI A, 53(H) | 3E,53    | |
| C025  | STA C40B(H) | 32,0B,C4 | |
| 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     | 19       | Add DE reg. pair to HL reg. pair |
| C036  | DCR B      | 05       | If LOOP12 has not been executed \( q \) times repeat LOOP12 |
| C037  | JNZ LOOP12 | D2,35,C0 | |
| C03A  | MOV A,H    | 7C       | Store higher order bits of \( n \) in address C402 |
| C03B  | STA C402(H) | 32,02,C4 | |
| C03E  | MOV A,L    | 7D       | |
C03F  STA  C403 (H)  32,03,C4  Store lower order bits of n in address C403
C042  LDA  C408 (H)  3A,08,C4  Load higher order bits of n of max. bytes required in reg. D
C045  MOV  D,A  57  Load lower order bits of n of max. bytes required in reg. E
C046  LDA  C409 (H)  3A,09,C4  Load memory with address of MSB
C049  MOV  E,A  5F  Initialize current byte to 0x00
C04A  LXI  H  C40C (H)  21,0C,C4  Decrement count of no. of byte
C04D  LOOP1  MVI A,00 (H)  3E,00  Check if MSB of count = 0x00
C04F  MOV  M,A  77  If yes, go to LOOP13
C050  DCX  D  1B  Load memory with address of next MSB
C051  MOV  D,A  57  And then go to LOOP1
C052  ORI  00 (H)  F6,00  Check if LSB of count = 0x00
C054  JZ  LOOP13  CA,5B,C0  If yes, go to LOOP2
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  Store higher byte of d in address C400
C05C  ORI  00 (H)  F6,00  Store lower byte of d in address C401
C05E  JZ  LOOP2  CA,71,C0  Reset Carry Flag
C061  INX  H  23  Store higher byte of max byte size in reg. B
C062  JMP  LOOP1  C3,4D,C0  Store lower byte of max byte size in reg. C
C065  LDA  C406 (H)  3A,06,C4  Point memory to MSB of copy
C068  STA  C400 (H)  32,00,C4  Point DE reg. pair to MSB of actual data
C06B  LDA  C407 (H)  3A,07,C4  Copy current byte
C06E  STA  C401 (H)  32,01,C4  Are all bytes copied?
C071  LOOP2  STC  37  If yes, go to DONE
C072  CMC  3F  Else, point memory to next MSB of copy
C073  LDA  C408 (H)  3A,08,C4  And point DE reg. pair to next actual MSB
C076  MOV  B,A  47  And then, go to REPEAT
C077  LDA  C409 (H)  3A,09,C4  Store higher byte of c in address C402
C07A  MOV  C,A  4F  Store lower byte of c in address C403
C07B  REPEAT1  LXI  H  940C (H)  21,0C,94  Store higher byte of max byte size in reg. B
C07E  LXI  D  C40C (H)  21,0C,C4  Store lower byte of max byte size in reg. C
C081  REPEAT  LDAX  D  1A  Point memory to LSB of actual data
C082  MOV  M,A  77  Point DE reg. pair to LSB of copy
C083  DCX  B  0B  Reset carry flag
C084  MOV  A,B  78  Add current copy byte to current actual byte
C085  ORA  C  B1  with carry
C086  JZ  DONE  CA,8E,C0
C089  INX  H  23
C08A  INX  D  13
C08B  JMP  REPEAT  C3,81,C0
C08E  DONE  LDA  C404 (H)  3A,04,C4
C091  STA  C402 (H)  32,02,C4
C094  LDA  C405 (H)  3A,05,C4
C097  STA  C403 (H)  32,03,C4
C09A  LOOP4  LDA  C408 (H)  3A,08,C4
C09D  MOV  B,A  47
C09E  LDA  C409 (H)  3A,09,C4
C101  MOV  C,A  4F
C102  LXI  H  FFFF (H)  21,FF,FF
C105  LXI  D  BFFF (H)  11,FF,BF
C108  STC  37
C109  CMC  3F
C10A  LOOP5  LDAX  D  1A
C10B  ADC  M  8E
C10C  MOV  M,A  77
| LABEL | MNEMONICS | HEX CODE | COMMENTS |
|-------|-----------|----------|----------|
| C10D  | DCX B     | 0B       | Are all bytes added? |
| C10E  | MOV A, B  | 78       | If yes, go to NEXT |
| C10F  | ORA C     | B1       | Else, point memory to next actual LSB |
| C110  | JZ NEXT   | CA, 18, C1 | And point DE reg pair to next copy LSB |
| C113  | DCX H     | 2B       | And then, go to LOOP5 |
| C114  | DCX B     | 1B       | |
| C115  | JMP LOOP5 | C3, 0A, C1 | D–E = count of c |
| C118  | LDA C402(H) | 3A, 02, C4 | Decrement D–E reg. pair |
| C11B  | MOV D, A  | 57       | |
| C11C  | LDA C403(H) | 3A, 03, C4 | |
| C11F  | MOV E, A  | 5F       | Is D–E = 0x0000? |
| C120  | DCX D     | 1B       | If yes, go to LOOP3 |
| C121  | MOV A, D  | 7A       | Else, count of c = new D–E |
| C122  | ORA E     | B3       | And then go to LOOP4 |
| C123  | JZ LOOP3  | CA, 13, C1 | |
| C126  | MOV A, D  | 7A       | |
| C127  | STA C402(H) | 32, 02, C4 | D–E = count of d |
| C12A  | MOV E, A  | 5F       | Decrement count of d |
| C12B  | STA C403(H) | 32, 03, C4 | |
| C12E  | JMP LOOP4 | CA, 9A, C0 | Is D–E = 0x0000? |
| C131  | LDA C400(H) | 3A, 00, C4 | If yes, go to LOOP8 |
| C134  | MOV D, A  | 57       | Else, update value of d – count |
| C135  | LDA C401(H) | 3A, 01, C4 | And then, go to REPEAT1 |
| C138  | MOV E, A  | 5F       | Reset Carry flag |
| C139  | DCX D     | 1B       | Load MSB of max bit size in reg. B |
| C13A  | MOV A, D  | 7A       | Load LSB of max bit size in reg. C |
| C13B  | ORA E     | B3       | Load MSB of n in reg. D |
| C13C  | JZ LOOP8  | CA, 4A, C1 | Load LSB of n in reg. E |
| C13F  | MOV D, A  | 57       | Load address of LSB to memory |
| C140  | STA C400(H) | 32, 00, C4 | Load A with data that memory points to |
| C143  | MOV E, A  | 5F       | Subtract LSB of n from this data |
| C144  | STA C401(H) | 32, 01, C4 | Update value of LSB |
| C147  | JMP REPEAT1 | C3, 7B, C0 | Indicates that LSB has been updated |
| C14A  | LOOP3     | 37       | Point memory to next LSB |
| C14B  | STC       | 3F       | Load A with data that memory points to |
| C14C  | LDA C408(H) | 3A, 08, C4 | Subtract MSB of n from this data with borrow |
| C14F  | MOV B, A  | 47       | Update value of second LSB |
| C150  | LDA C409(H) | 3A, 09, C4 | Indicates that second LSB has been updated |
| C153  | MOV C, A  | 4F       | Point memory to next LSB |
| C154  | LDA C402(H) | 3A, 02, C4 | Subtract MSB of n from this data with borrow |
| C157  | MOV D, A  | 57       | Update value of second LSB |
| C158  | LDA C403(H) | 3A, 03, C4 | Indicates that second LSB has been updated |
| C15B  | MOV E, A  | 5F       | Point memory to next LSB |
| C15C  | LXI H FFFF(H) | 21, FF, FF | Subtract 0x00 from current byte with borrow |
| C15F  | MOV A, M  | 7E       | |
| C160  | SUB E     | 93       | |
| C161  | MOV M, A  | 77       | |
| C162  | DCX B     | 0B       | |
| C163  | DCX H     | 2B       | |
| C164  | MOV A, M  | 7E       | |
| C165  | SBB D     | 9A       | |
| C166  | MOV M, A  | 77       | |
| C167  | DCX B     | 0B       | |
| C168  | DCX H     | 2B       | |
| C169  | MOV A, M  | 7E       | |
| C16A  | SBI 00(H) | DF, 00  | |
| LABEL | MNEMONICS | HEX CODE | COMMENTS |
|-------|-----------|----------|----------|
| C16C  | MOV M,A   | 77       | Indicates that current byte is updated |
| C16D  | DCX B     | 0B       |          |
| 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 | Load LSB of max bit size to reg. C |
| C187  | MOV C,A   | 4F       |          |
| C188  | MOV A,B   | 78       |          |
| 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       | Compare current value of byte with 0x00 |
| C194  | CPI 00(H) | FE,00    | If not true, go to LOOP8 |
| C196  | JNZ LOOP8 | C2,4A,C1 | Indicates that current byte is compared |
| C199  | DCX B     | 0B       |          |
| 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 And go to LOOP9 |
| C201  | JMP LOOP9 | C3,93,C1 |          |
| 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 And go to LOOP9 |
| C20B  | JMP LOOP9 | C3,93,C1 |          |
| 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 \( c \) 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 at least 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 \( y \in \{1, 2, \cdots \phi(n) - 1\} \). 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\text{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 prime numbers between 17 and 67 (both inclusive). So, we will use the following approach.

![Flowchart](image)

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:

\[ X_{n+1} = (AX_n + C) \mod M \quad \text{eq.4.01} \]

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 \( \{0, 1, \cdots, M - 1\} \). To restrict the numbers to the set \( \{0, 1, \cdots, 12\} \), we will take the remainder of the numbers thus generated upon division by the number 13, i.e.

\[ I_n = X_n \mod 13 \quad \text{eq.4.02} \]

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:
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 |
|--------|-----------|----------|
| STARTRAND | MVI A, 19(H) | Store MSB of A |
|          | STA C400(H) |          |
|          | MVI A, 66(H) | Store second byte of A |
|          | STA C401(H) |          |
|          | MVI A, 0D(H) | Store LSB of A |
|          | STA C402(H) |          |
|          | MVI A, 3C(H) | Store MSB of C |
|          | STA C3FC(H) |          |
|          | MVI A, 6E(H) | Store second MSB of C |
|          | STA C3FD(H) |          |
|          | MVI A, F3(H) | Store second LSB of C |
|          | STA C3FE(H) |          |
|          | MVI A, 5F(H) | Store LSB of C |
| SET1    | MVI A, 00(H) |          |
| MOV M, A | DCR C |          |
| JZ SET2 | INX H |          |
| JMP SET1 | LDA C3F8(H) | Set initial seed = 0x00000000 |
| SET2    | STA C403(H) | Copy 5th byte of seed |
|          | LDA C3F9(H) |          |
|          | STA C404(H) | Copy 6th byte of seed |
|          | LDA C3FA(H) |          |
|          | STA C405(H) | Copy 7th byte of seed |
|          | LDA C3FB(H) |          |
|          | STA C406(H) | Copy 8th byte (LSB) of seed |
| ANOTHER | LDA C400(H) | Store MSB of A in reg. C |
| MOV C, A | DCR C |          |
|          | LDA C401(H) | Store second byte of A in reg. D |
| MOV D, A | LDA C402(H) |          |
| MOV E, A | LDA C403(H) | Store LSB of A in reg. E |
| LOOP14  | LDA C406(H) | Point memory to LSB of seed |
| ADD M   | LDA C404(H) | Add LSB of initial seed to current memory |
| MOV M, A | LDA C405(H) | Update value of current byte |
| DCX H   | LDA C406(H) | Point memory to second LSB of seed |
| ADD M   | LDA C404(H) | Add second LSB of initial seed to memory with carry |
| MOV M, A | LDA C405(H) | Update value of current byte |
| DCX H   | LDA C406(H) | Point memory to third LSB of seed |
| ADD M   | LDA C404(H) | Add third LSB of initial seed to memory with carry |
| MOV M, A | LDA C405(H) | Update value of current byte |
| DCX H   | LDA C406(H) | Point memory to fourth LSB of seed |
| ADD M   | LDA C404(H) | Add fourth LSB of initial seed to memory with carry |
| MOV M, A | LDA C405(H) | Update value of current byte |
| DCX H   | MVI B, 04(H) | Point memory to next LSB of seed |
|        |           | Initialize reg. B to 04(H) |
| LABEL | MNEMONICS | COMMENTS |
|-------|-----------|----------|
| 55    | MOV A,M  | Add 0x00 to current seed byte with carry |
| 56    | ACI 00(H) | Update value of current byte |
| 57    | MOV M,A  | Indicates that current byte has been updated |
| 58    | DCR B    | If all bytes of seed are updated, go to LOOP25 |
| 59    | JZ LOOP25| Else, point memory to next LSB of seed |
| 60    | DCX H    | And then go to LOOP26 |
| 61    | JMP LOOP26| |
| 62    | LOOP25   | |
| 63    | MOV A,E  | Subtract 0x01 from reg. E |
| 64    | SUI 01(H)| Subtract 0x00 from reg. D with borrow |
| 65    | MOV E,A  | Subtract 0x00 from reg. C with borrow |
| 66    | MOV A,D  | Is reg. C = reg. D = 0x00? |
| 67    | SBI 00(H)| If yes, go to LOOP13 |
| 68    | MOV A,C  | If not, go to LOOP14 |
| 69    | SBI 00(H)| |
| 70    | MOV C,A  | Point memory to LSB of seed |
| 71    | MOV A,C  | Add LSB of C to LSB of seed |
| 72    | ORA D    | Point memory to second LSB of seed |
| 73    | JZ LOOP13| Add second LSB of C to second LSB of seed with carry |
| 74    | JMP LOOP14| Point memory to third LSB of seed |
| 75    | LOOP13   | Add second LSB of C to second LSB of seed with carry |
| 76    | MOV A,E  | Point memory to third LSB of seed |
| 77    | CPI 00(H)| Add second LSB of C to third LSB of seed with carry |
| 78    | JZ LOOP15| Set reg. B = 0x04 |
| 79    | LOOP15   | Add 0x00 to current byte of seed |
| 80    | LXI H, C3FB(H) | Indicates that current seed byte is updated |
| 81    | LDA C3FF(H) | If all seed bytes are updated go to LOOP16 |
| 82    | ADD      | If not, point memory to next LSB of seed |
| 83    | MOV M,A  | And then go to LOOP17 |
| 84    | DCX H    | Store MSB of seed in address C403 |
| 85    | LDA C3FE(H) | Store second MSB of seed in address C404 |
| 86    | ADC M    | Store second LSB of seed in address C405 |
| 87    | MOV M,A  | |
| 88    | DCX H    | |
| 89    | LDA C3FD | |
| 90    | ADC M    | |
| 91    | MOV M,A  | |
| 92    | DCX H    | |
| 93    | LDA C3FC | |
| 94    | ADC M    | |
| 95    | MOV M,A  | |
| 96    | DCX H    | |
| 97    | LOOP17   | |
| 98    | MOV M,A  | |
| 99    | ACI 00(H)| |
| 100   | DCR B    | |
| 101   | JZ LOOP16| |
| 102   | DCX H    | |
| 103   | JMP LOOP17| |
| 104   | LOOP16   | |
| 105   | LDA C3F8(H) | |
| 106   | STA C403(H) | |
| 107   | LDA C3F9(H) | |
| 108   | STA C404(H) | |
| 109   | LDA C3FA(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 | Point memory to LSB of seed |
| 113   | LXI H C3FB(H) | |
| 114   | MOV A,M | |
| 115   | SUB B | Subtract reg. B from LSB of seed |
| 116   | MOV M,A | Go to next LSB of seed |
| 117   | DCX H | |
| 118   | LOOP19 | Store 0x03 in reg. D |
| 119   | MVI D,03(H) | |
| 120   | MOV A,M | Subtract 0x00 from current byte of seed with borrow |
| 121   | DCX H | Indicates that current seed byte is updated |
| 122   | JZ LOOP18 | If all seed bytes are updated go to LOOP18 |
| 123   | JMP LOOP19 | If not, point memory to next LSB of seed byte |
| 124   | LOOP18 | And then go to LOOP19 |
| 125   | 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 | Point memory to second LSB of seed |
| 132   | INX H | |
| 133   | MOV A,M | |
| 134   | CPI 00(H) | Compare second LSB of seed byte with 0x00 |
| 135   | JZ LOOP22 | If they are equal, go to LOOP22 |
| 136   | JMP LOOP21 | Else, go to LOOP21 |
| 137   | LOOP22 | Point memory of LSB of 4-byte seed |
| 138   | INX H | Load this byte to accumulator |
| 139   | MOV A,M | Compare this byte with contents of reg. B |
| 140   | CMP B | |
| 141   | JZ LOOP23 | If they are equal go to LOOP23 |
| 142   | JMP LOOP21 | If byte is less than 0x0D, go to LOOP24 |
| 143   | LOOP23 | Else, go to LOOP21 |
| 144   | MVI A,00(H) | |
| 145   | MOV M,A | Set LSB of seed to 0x00 |
| 146   | LDA C3FB(H) | Store mod 13 random no. in address C3F3 |
| 147   | STA C3F3(H) | Set byte 1 of next initial seed to 0x00 |
| 148   | MVI A,00(H) | Set byte 2 of next initial seed to 0x00 |
| 149   | STA C3F4(H) | Set byte 3 of next initial seed to 0x00 |
| 150   | MVI A,00(H) | Set byte 4 of next initial seed to 0x00 |
| 151   | STA C3F5(H) | Set byte 5 of next initial seed to MSB of the 4-byte new seed generated |
| 152   | MVI A,00(H) | Set byte 6 of next initial seed to second MSB of the 4-byte new seed generated |
| 153   | STA C3F6(H) | Set byte 7 of next initial seed to second LSB of the 4-byte new seed generated |
| 154   | MVI A,00(H) | Set byte 8 (LSB) of next initial seed to LSB of the 4-byte new seed generated |
| 155   | LDA C403(H) | |
| 156   | STA C3F7(H) | |
| 157   | LDA C404(H) | |
| 158   | STA C3FA(H) | |
| 159   | LDA C405(H) | Set byte 7 of next initial seed to second MSB of the 4-byte new seed generated |
| 160   | STA C3FB(H) | Set byte 8 (LSB) of next initial seed to LSB of the 4-byte new seed generated |
| 161   | 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                |
|--------|----------------------------|-------------------------|
| CHOOSE | MVI A, 11(H)                | Store 17 = 0x11 in address FFD1 |
|        | STA FFD1(H)                |                         |
| 2      | MVI A, 13(H)               | Store 19 = 0x13 in address FFD2 |
| 3      | STA FFD2(H)                |                         |
| 4      | MVI A, 17(H)               | Store 23 = 0x17 in address FFD3 |
| 5      | STA FFD3(H)                |                         |
| 6      | MVI A, 1D(H)               | Store 29 = 0x1D in address FFD4 |
| 7      | STA FFD4(H)                |                         |
| 8      | MVI A, 1F (H)              | Store 31 = 0x1F in address FFD5 |
| 9      | STA FFD5(H)                |                         |
| 10     | MVI A, 25(H)               | Store 37 = 0x25 in address FFD6 |
| 11     | STA FFD6(H)                |                         |
| 12     | MVI A, 29(H)               | Store 41 = 0x29 in address FFD7 |
| 13     | STA FFD7(H)                |                         |
| 14     | MVI A, 2B(H)               | Store 43 = 0x2B in address FFD8 |
| 15     | STA FFD8(H)                |                         |
| 16     | MVI A, 2F (H)              | Store 47 = 0x2F in address FFD9 |
| 17     | STA FFD9(H)                |                         |
| 18     | MVI A, 35(H)               | Store 53 = 0x35 in address FFDA |
| 19     | STA FFDA(H)                |                         |
| 20     | MVI A, 3B(H)               | Store 59 = 0x3B in address FFDB |
| 21     | STA FFDB(H)                |                         |
| 22     | MVI A, 3D(H)               | Store 61 = 0x3D in address FFDC |
| 23     | STA FFDC(H)                |                         |
| 24     | MVI A, 43(H)               | Store 67 = 0x43 in address FFDD |
| 25     | STA FFD(H)                 | CALL STARTRAND function given in section 4.2 |
| 26     | CALL STARTRAND             |                         |
| 27     | LDA C3F3(H)                | Load the random no. generated |
| 28     | ADI 01(H)                  | Add 0x01 to this random no. |
| 29     | MOV C,A                    | And store it in reg. C |
| 30     | LXI H FFD0(H)              | Load address preceding that of first prime |
| 31     | JNZ NEXTPRIME              | in HL reg. pair |
| 32     | INX H                      | Have we reached the required index? |
| 33     | DCR C                      | If not, go to NEXTPRIME |
| 34     | MOV A,M                    | Else load current prime no. to accumulator |
| 35     | STA C40C(H)                | And store it in C40C (this is p) |
| 36     | CALL ANOTHER               | Go to ANOTHER label in 4.2 code |
| 37     | LDA C3F3(H)                | Do the same steps as done for p for q |
| 38     | ADI 01(H)                  |                         |
| 39     | MOV C,A                    |                         |
| 40     | LXI H FFD0(H)              |                         |
| 41     | INX H                      |                         |
| 42     | DCR C                      |                         |
| 43     | JNZ NEXTPRIME              |                         |
| 44     | MOV A,M                    |                         |
| 45     | STA C40D(H)                | Store q in address C40D |
| 46     | RET                         |                         |
| 47     |                             |                         |
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:

![Flowchart for deciding private key exponent](image)

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:
MVI A, 17(H)

STA C40C(H)

MVI A, 2F(H)

STA C40D(H)

LDA C40C(H)

SUI 01(H)

MOV C, A

MVI B, 00(H)

LDA C40D(H)

SUI 01(H)

MOV E, A

MVI D, 00(H)

LXI H 0000(H)

MOV D, A

ORA E

JZ LOOP27

JMP LOOP28

MOV A, H

STA C40E(H)

MOV A, L

STA C40F(H)

MVI A, 11(H)

STA C408(H)

MVI A, 00(H)

STA C410(H)

MVI A, 01(H)

STA C411(H)

LOOP28

DAD B

DCX D

MOV D, A

ORA E

JZ LOOP27

JMP LOOP28

MOV A, H

STA C40E(H)

MOV A, L

STA C40F(H)

MVI A, 11(H)

STA C408(H)

MVI A, 00(H)

STA C410(H)

MVI A, 01(H)

STA C411(H)

LOOP27

MVI C, 00(H)

MVI D, 00(H)

MVI E, 00(H)

LDA C408(H)

MOV B, A

LDA C410(H)

STA C412(H)

LDA C411(H)

STA C413(H)

LOOP32

STC

CMC

MVI C, 00(H)

MVI D, 00(H)

MVI E, 00(H)

LDA C408(H)

MOV B, A

LDA C410(H)

STA C412(H)

LDA C411(H)

STA C413(H)

LOOP30

MOV A, E

ADD B

MOV E, A

MOV A, D

ACI 00(H)

MOV D, A

MOV A, C

ACI 00(H)

MOV C, A

LDA C412(H)

MOV H, A

LDA C413(H)

MOV L, A

DCX H

MOV A, H

ORA L

JZ LOOP29

Store p = 23 = 0x17 in address C40C

Store q = 47 = 0x2F in address C40D

Store p−1 in reg. C

Store 0x00 in reg. B

Store q−1 in reg. E

Store 0x00 in reg. D

H−L = 0x0000

H−L = H−L + B−C

Indicates that current iteration is done

Is D−E = 0x0000?

If yes, go to LOOP27

Else, go to LOOP28

Store MSB of totient in address C40E

Store LSB of totient in address C40F

Store e in address C408

Store MSB of d−count

Store LSB of d−count

Reset Carry flag

C−D−E = 0x000000 (product)

Store e in reg. B

Store copy of MSB of d−count in C412

Store copy of LSB of d−count in C413

Add e to LSB of product

Add 0x00 to second byte of product with carry

Add 0x00 to MSB of product with carry

Store MSB of d−count copy in H

Store LSB of d−count copy in L

Decrement d−count copy

Is copy of d−count = 0x0000?

If yes, go to LOOP29
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:
Start

Store all 1000 bytes in addresses FBE8 - FFCF

Choose random primes $p$ and $q$ for first data byte

Choose $e = 17 = 0x11$ and set appropriate value of $d$

Find cipher text for current data byte

Store bytes in next two locations of ciphertext array

Are all data bytes encrypted?

yes

Stop

no

Go to next data byte

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
Store 17 = 0x11 in address FFD1
Store 19 = 0x13 in address FFD2
Store 23 = 0x17 in address FFD3
Store 29 = 0x1D in address FFD4
Store 31 = 0x1F in address FFD5
Store 37 = 0x25 in address FFD6
Store 41 = 0x29 in address FFD7
Store 43 = 0x2B in address FFD8
Store 47 = 0x2F in address FFD9
Store 53 = 0x35 in address FFDA
Store 59 = 0x3B in address FFDB
Store 61 = 0x3D in address FFDC
Store 67 = 0x43 in address FFDD
Store MSB of 1st message byte address in C414
Store LSB of 1st message byte address in C415
Store MSB of pointer to ciphertext array in C416
Store LSB of pointer to ciphertext array in C417
D–E = first message byte
Store current message byte in address C407
Store e in address C408
Store max. byte size in address C409
Call function ANOTHER
Load the random no. generated
Add 0x01 to this random no.
And store it in reg. C
Load address preceding that of first prime in HL reg. pair
Go to next prime
Have we reached the required index?
If not, go to NEXTPRIME
Else load current prime no. to accumulator
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
D09D | 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
D10D | 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 | 
D116 | LXI H FFEF(H) | 21,EF,FF | 
D119 | LOOP1 | MVI A,00(H) | 3E,00 | Initialize reg. pair HL with address of MSB
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 locations are not initialized point memory next address byte & go to LOOP1
D124 | LOOP2 | STC | 37 | Reset Carry flag
D125 | CMC | 3F | 
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 | 
D12E | REPEAT1 | LXI H FFDE(H) | 21,DE,FF | Store max byte size in reg. B
D131 | LXI D FFFE(H) | 11,EF,FF | H-L points to content at address FFDE
D134 | REPEAT | LDAX D | 1A | D-E points to MSB which will converge to c
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 | An 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 | Reset carry flag
D14E | CMC | 3F |
LDAX D
ADC M
MOV M,A
DCR B
JZ NEXT
DCX H
DCX D
JMP LOOP5
DCR C
JZ NEXT
DCX H
DCX D
JMP LOOP3
DCR E
JZ LOOP3
JMP LOOP4
DCR B
JZ LOOP4
LXI H FFFF (H)
MOV A,M
SUB E
MOV A,M
DCR B
JZ LOOP6
LXI H FFFF (H)
MOV A,M
SBB D
MOV A,M
7E
SUB E
MOV M,A
77
DCR B
05
DCX H
2B
MOV D,A
57
LDA C40A (H)
3A,0A,C4
MOV E,A
5F
LXI H, FFFE (H)
21,FF,FF
MOV A,M
7E
SBI 00 (H)
MOV A,M
77
DCR B
05
JZ LOOP6
LDA C40B (H)
3A,0B,C4
MOV E,A
5F
LXI H,FFFE (H)
21,FF,FF
MOV A,M
7E
LDA C409 (H)
3A,09,C4
MOV M,A
77
DCR B
05
DCX H
2B
LDA C409 (H)
3A,09,C4
MOV A,M
7E
SBI 00 (H)
MOV M,A
77
DCR B
05
JZ LOOP6
LDA C40A (H)
3A,0A,C4
MOV B,A
47
SUI 02 (H)
MOV M,A
77
DCR B
05
JZ LOOP6
JMP REPEAT1
C3,2E,D1
STC
C M C
LDA C409 (H)
MOV B,A
LDA C40A (H)
MOV D,A
LDA C40B (H)
MOV E,A
LXI H, FFFF (H)
MOV A,M
SUB E
MOV M,A
SBB D
MOV M,A
DCR B
JZ LOOP6
LDA C409 (H)
SUI 02 (H)
LXI H FFEF (H)
CPI 00 (H)
JNZ LOOP6
DCR B
JZ LOOP6
INX H
23
JMP LOOP9
C3,98,D1
INX H
23
MOV A,D
7A

Add copy LSB to actual LSB
Indicates that current byte is updated
If all bytes are updated go to NEXT
Else, point memory to next LSB
Point DE reg. pair to next LSB of copy
And go to LOOP5
Indicates that current iteration is done
If all m iterations are done, go to LOOP3
If not, go to LOOP4
Indicates that current iteration is done
If all e iterations are completed go to LOOP8
Else, go to REPEAT1
Reset Carry flag
Load max byte no. to reg. B
Load higher order bit of n to reg. D
Load lower order bit of n to reg. E
Initialize reg. pair HL with address of LSB
Subtract lower order bit of n from LSB
Update value of LSB
Indicates that LSB is updated
Point memory to next LSB
Subtract higher order bit of n from second LSB with borrow
Update value of second LSB
Indicates that second LSB is updated
Point memory to next LSB
Subtract 0x00 from current byte with borrow
Update value of current byte
Indicates that current byte has been updated
If all bytes of current iteration are updated go to LOOP7
If all bytes of current iteration are not updated, go to LOOP6
Subtract 0x02 from max bit size
Store max bit size − 2 in reg. B
Load memory with address of MSB
Compare current byte with 0x00
If current byte is not 0x00, go to LOOP8
Indicates that current byte has been found to be 0x00
If first 15 bytes have been compared go to LOOP10
If first 15 bytes have not yet been compared point memory to next MSB
**MNEONICS**

- CMP M
- JZ LOOP11
- JC LOOP8
- JMP LOOP12
- INX H
- MOV A, E
- CMP M
- JNZ LOOP12
- MVI A, 00(H)
- STA FFFE (H)
- MVI A, 00(H)
- STA FFFF (H)
- LDA C416 (H)
- MOV D,A
- LDA C417 (H)
- MOV E, A
- LDA FFFE (H)
- STAX D
- INX D
- LDA FFFF (H)
- STAX D
- INX D
- MOV A, D
- STA C416 (H)
- MOV A, E
- STA C417 (H)
- LDA C414 (H)
- MOV D, A
- LDA C415 (H)
- MOV E, A
- INX D
- MOV A, D
- CPI FF (H)
- JZ CHECK2
- MOV A, D
- STA C414 (H)
- MOV A, E
- STA C415 (H)
- JMP START
- MOV A, E
- CPI D0(H)
- JZ STOP
- JMP AGAIN
- HLT

**HEX CODE**

- BE
- CA, 11, D2
- DA, 69, D1
- C3, 24, D2
- 23
- 3E, 00
- 32, FE, FF
- 3E, 00
- 32, FF, FF
- 3A, 16, C4
- 57
- 3A, 17, C4
- 5F
- 21, FE, FF
- 12
- 32, FE, FF
- 13
- 21, FFFF
- 12
- INX D
- 13
- MOV A, D
- 7A
- 32, 16, C4
- 7B
- 32, 17, C4
- 3A, 14, C4
- 57
- 3A, 15, C4
- 5F
- 13
- 7A
- 32, FF, FF
- 58, D2
- CA, 58, D2
- 58, D2
- 32, 14, C4
- 7B
- 32, 15, C4
- 58, D0
- 3C, 58, D0
- 78
- 8E, D0
- CA, 61, D2
- C3, 4D, D2
- C3, 4D, D2
- C3, 4D, D2
- HLT

**COMMENTS**

- Compare higher order bits of n with data at address FFFE
  - If they are equal, go to LOOP11
  - If they are not equal and address at FFFE is greater than higher order bits of n, go to LOOP8
- Else if they are equal and address at FFFE is less than higher byte of n, go to LOOP12

- Compare lower order bits of n with data at address FFFF
  - If address at FFFF is less than the lower order bits of n, go to LOOP8
  - Else if they are not equal, go to LOOP12
  - Else if they are equal, proceed

- Set value at address FFFE = 0x00
- Set value at address FFFF = 0x00

- Load MSB of ciphertext array pointer address to reg. D
  - Load LSB of ciphertext generated to A
  - And store in ciphertext array pointer
  - Increment pointer to point to next address
  - Load LSB of ciphertext generated to A
  - And store in ciphertext array pointer
  - Increment pointer to point to next address

- Store MSB of incremented pointer back to C416
  - Store LSB of incremented pointer back to C417

- \( D-E = \text{pointer of message byte array} \)
  - Current byte has been encrypted
  - Check if MSB of incremented pointer = FF
  - If yes, go to CHECK2
  - Else, proceed

- Store MSB of incremented pointer back to C414
  - Store LSB of incremented pointer back to C415
  - And then, go to START

- Check if LSB of incremented pointer = D0
  - If yes, terminate the program
  - Else, go to AGAIN
| LABEL     | MNEMONICS                  | HEX CODE   | COMMENTS                                           |
|-----------|----------------------------|------------|----------------------------------------------------|
| D262      | STARTRAND                  |            |                                                   |
| D264      | MVI A, 19(H)               | 3E,19      | Store MSB of A                                     |
| D267      | STA C400(H)                | 32,00,C4   |                                                   |
| D269      | MVI A, 66(H)               | 3E,66      |                                                   |
| D26C      | STA C401(H)                | 32,01,C4   |                                                   |
| D26E      | MVI A, 0D(H)               | 3E,0D      |                                                   |
| D271      | STA C402(H)                | 32,02,C4   |                                                   |
| D273      | MVI A, 3C(H)               | 3E,3C      |                                                   |
| D276      | STA C3FC(H)                | 32,FC,C3   |                                                   |
| D278      | MVI A, 6E(H)               | 3E,6E      |                                                   |
| D27B      | STA C3FD(H)                | 32,FD,C3   |                                                   |
| D27D      | MVI A, F3(H)               | 3E,F3      |                                                   |
| D280      | STA C3FE(H)                | 32,FE,C3   |                                                   |
| D282      | MVI A, 5F(H)               | 3E,5F      |                                                   |
| D285      | STA C3FF(H)                | 32,FF,C3   |                                                   |
| D288      | MVI C, 08(H)               | 0E,08      |                                                   |
| D28A      | LXI H C3F4(H)              | 21,F4,C3   |                                                   |
| 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   | Copy 5th byte of seed                              |
| D298      | STA C403(H)                | 32,03,C4   |                                                   |
| D29B      | LDA C3F9(H)                | 3A,F9,C3   |                                                   |
| D29E      | STA C404(H)                | 32,04,C4   |                                                   |
| D301      | LDA C3FA(H)                | 3A,FA,C3   |                                                   |
| D304      | STA C405(H)                | 32,05,C4   |                                                   |
| 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   | Store MSB of A in reg. C                           |
| D310      | MOV C,A                    | 4F         | Store second byte of A in reg. D                   |
| D311      | LDA C401(H)                | 3A,01,C4   |                                                   |
| D314      | MOV D,A                    | 57         |                                                   |
| D315      | LDA C402(H)                | 3A,02,C4   |                                                   |
| D318      | MOV E,A                    | 5F         |                                                   |
| D319      | LXI H C3FB(H)              | 21,FB,C3   |                                                   |
| 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)                         |

**MVI** - Move Immediate Value

**STA** - Store Accumulator

**LDA** - Load Accumulator

**MOV** - Move

**DCR** - Decrement

**JMP** - Jump

**INX** - Increment X Register

**ADC** - Add with Carry

**ADD** - Add

**LXI** - Load Index Register

**MVI** - Move Immediate Value

**STA** - Store Accumulator

**MOV** - Move

**DCX** - Decrement X Register

**LDA** - Load Accumulator

**MOV** - Move

**ADC** - Add with Carry

**ADD** - Add

**LXI** - Load Index Register

**MVI** - Move Immediate Value

**STA** - Store Accumulator

**MOV** - Move

**DCX** - Decrement X Register

**LDA** - Load Accumulator

**MOV** - Move
| LABEL   | MNEMONICS         | HEX CODE | COMMENTS                                                                 |
|---------|-------------------|----------|--------------------------------------------------------------------------|
| D336    | LOOP26 MOV A,M    | 7E       | Add 0x00 to current seed byte with carry                                 |
| D337    | ACI 00(H)         | CF,00    | Update value of current byte                                            |
| D339    | MOV M,A           | 77       | Indicates that current byte has been updated                             |
| D33A    | DCR B             | 05       | If all bytes of seed are updated, go to LOOP25                          |
| D33B    | JZ LOOP25         | CA,42,D3 | Else, point memory to next LSB of seed                                  |
| D33E    | DCX H             | 2B       | And then go to LOOP26                                                  |
| D33F    | JMP LOOP26        | C3,36,D3 |                                                                           |
| D342    | LOOP25 MOV A,E    | 7B       |                                                                           |
| D343    | SUI 01(H)         | D6,01    |                                                                           |
| D345    | MOV E,A           | 5F       |                                                                           |
| D346    | MOV A,D           | 7A       |                                                                           |
| D347    | SBI 00(H)         | DF,00    |                                                                           |
| D349    | MOV D,A           | 57       |                                                                           |
| D34A    | MOV A,C           | 79       |                                                                           |
| D34B    | SBI 00(H)         | DF,00    |                                                                           |
| D34D    | MOV C,A           | 4F       |                                                                           |
| D34E    | MOV A,E           | 79       |                                                                           |
| D34F    | ORA D             | B2       |                                                                           |
| D350    | JZ LOOP13         | CA,56,D3 |                                                                           |
| D353    | JMP LOOP14        | C3,19,D3 |                                                                           |
| D356    | LOOP13 MOV A,E    | 7B       |                                                                           |
| D357    | CPL 00(H)         | FE,00    |                                                                           |
| D359    | JZ LOOP15         | CA,5F,D3 |                                                                           |
| D35C    | JMP LOOP14        | C3,19,D3 |                                                                           |
| D35F    | LOOP15 LXI H, C3FB(H) | 21,FB,C3 | Point memory to LSB of seed                                             |
| D362    | LDA C3FF(H)       | 3A,FF,C3 |                                                                           |
| D365    | ADD M             | 86       |                                                                           |
| D366    | MOV M,A           | 77       |                                                                           |
| D367    | DCX H             | 2B       |                                                                           |
| D368    | LDA C3FE(H)       | 3A,FE,C3|                                                                           |
| D36B    | ADC M             | 8E       |                                                                           |
| D36C    | MOV M,A           | 77       |                                                                           |
| D36D    | DCX H             | 2B       |                                                                           |
| D36E    | LDA C3FD          | 3A,FD,C3|                                                                           |
| D371    | ADC M             | 8E       |                                                                           |
| D372    | MOV M,A           | 77       |                                                                           |
| D373    | DCX H             | 0B       |                                                                           |
| D374    | LDA C3FC          | 3A,FC,C3|                                                                           |
| D377    | ADC M             | 8E       |                                                                           |
| D378    | MOV M,A           | 77       |                                                                           |
| D379    | DCX H             | 0B       |                                                                           |
| D37A    | MVI B, 04(H)      | 06,04    |                                                                           |
| D37C    | LOOP17 ACI 00(H)  | CF,00    |                                                                           |
| D37E    | MOV M,A           | 77       |                                                                           |
| D37F    | DCR B             | 05       |                                                                           |
| D380    | JZ LOOP16         | CA,87,D3 |                                                                           |
| D383    | DCX H             | 2B       |                                                                           |
| D384    | JMP LOOP17        | C3,7C,D3 |                                                                           |
| D387    | LOOP16 LDA C3F8(H) | 3A,F8,C3 |                                                                           |
| D38A    | STA C403(H)       | 32,03,C4|                                                                           |
| D38D    | LDA C3F9(H)       | 3A,F9,C3|                                                                           |
| D390    | STA C404(H)       | 32,04,C4|                                                                           |
| D393    | LDA C3FA(H)       | 3A,FA,C3|                                                                           |
| D396    | STA C405(H)       | 32,05,C4|                                                                           |
| D399    | LDA C3FB(H)       | 3A,FB,C3|                                                                           |

### Comments
- **Add 0x00 to current seed byte with carry**
- **Update value of current byte**
- **Indicates that current byte has been updated**
- **If all bytes of seed are updated, go to LOOP25**
- **Else, point memory to next LSB of seed**
- **And then go to LOOP26**
- **Subtract 0x01 from reg. E**
- **Subtract 0x00 from reg. D with borrow**
- **Subtract 0x00 from reg. C with borrow**
- **Is reg. C = reg. D = 0x00?**
- **If yes, go to LOOP13**
- **If not, go to LOOP14**
- **Point memory to LSB of seed**
- **Add LSB of C to LSB of seed**
- **Point memory to second LSB of seed**
- **Add second LSB of C to second LSB of seed with carry**
- **Point memory to third LSB of seed**
- **Add second LSB of C to second LSB of seed with carry**
- **Point memory to third LSB of seed**
- **Add second LSB of C to third LSB of seed with carry**
- **Point memory to third fourth of seed**
- **Set reg. B = 0x04**
- **Add 0x00 to current byte of seed**
- **Indicates that current seed byte is updated**
- **If all seed bytes are updated go to LOOP16**
- **If not, point memory to next LSB of seed**
- **And then go to LOOP17**
- **Store MSB of seed in address C403**
- **Store second MSB of seed in address C404**
- **Store second LSB of seed in address C405**
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  57E  Point memory to LSB of seed
D404   MOV A,M    90  Subtract reg. B from LSB of seed
D405   SUB B      7E  Go to next LSB of seed
D406   MOV M,A    77  Store 0x03 in reg. D
D407   DCX H      2B  Subtract 0x00 from current byte of seed with borrow
D408   MVI D,03(H) 16,03  Indicates that current seed byte is updated
D40A   LOOP19  57E  If all seed bytes are updated go to LOOP18
D40B   MOV A,M    7E  If not, point to next LSB of seed byte
D40D   MOV M,A    77  And then go to LOOP19
D40E   DCR D      15  Load memory contents in accumulator
D40F   JZ LOOP18  CA,16,D4  Point memory to MSB of 4-byte seed
D412   DCX H      2B  Point memory to second MSB of seed
D413   JMP LOOP19 C3,0A,D4  Are both MSB and second MSB of seed = 0x00?
D416   LOOP18  57E  If yes, go to LOOP20
D419   MOV A,M    7E  If not, go to LOOP21
D41A   INX H      23  Point memory to second LSB of seed
D41B   ORA M      B6  Load this byte to accumulator
D41C   JZ LOOP20  CA,22,D4  Compare this byte with contents of reg. B
D41F   JMP LOOP21 C3,01,D4  If they are equal, go to LOOP22
D422   INX H      23  Else, go to LOOP21
D423   MOV A,M    7E  Point memory of LSB of 4-byte seed
D424   CPI 00(H)  FE,00  Load this byte to accumulator
D426   JZ LOOP22  CA,2C,D4  Compare second LSB of seed byte with 0x00
D429   JMP LOOP21 C3,01,D4  If they are equal, go to LOOP23
D42D   LOOP22  57E  Else, go to LOOP21
D42E   JMP LOOP21 C3,01,D4  Point memory of LSB of 4-byte seed
D432   JZ LOOP24  CA,38,D4  Load this byte to accumulator
D435   JMP LOOP21 C3,01,D4  If byte is less than 0x0D, go to LOOP24
D438   LOOP23  57E  Else, go to LOOP21
D43A   MOV A,M    7E  Set LSB of seed to 0x00
D43B   LOOP24  57E  Store mod 13 random no. in address C3F3
D43C   STA C3F3 (H) 32,F3,C3  Set byte 1 of next initial seed to 0x00
D441   MVI A,00(H) 3E,00  Set byte 2 of next initial seed to 0x00
D443   STA C3F4(H) 32,F4,C3  Set byte 3 of next initial seed to 0x00
D446   MVI A,00(H) 3E,00  Set byte 4 of next initial seed to 0x00
D448   STA C3F5(H) 32,F5,C3  Set byte 5 of next initial seed to MSB of the 4-byte new seed generated
D44B   MVI A,00(H) 3E,00  Set byte 6 of next initial seed to second MSB of the 4-byte new seed generated
D44D   STA C3F6(H) 32,F6,C3  Set byte 7 of next initial seed to second LSB of the 4-byte new seed generated
D450   MVI A,00(H) 3E,00  Set byte 8 (LSB) of next initial seed to LSB of the 4-byte new seed generated
D452   STA C3F7(H) 32,F7,C3
D455   LDA C403(H) 3A,03,C4
D458   STA C3F8(H) 32,F8,C3
D45B   LDA C404(H) 3A,04,C4
D45E   STA C3F9(H) 32,F9,C3
D461   LDA C405(H) 3A,05,C4
D464   STA C3FA(H) 32,FA,C3
D46A   LDA C406(H) 3A,06,C4
D46D   STA C3FB(H) 32,FB,C3

 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, \ldots, q_1, q_2, \ldots, d_1, d_2, \ldots\} 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 $P = \{e, n_1, n_2, \ldots\}$ 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 Micciancioz. “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.