NOVA IMS
Information Management School

CEASER'S CIPHER

ASSEMBLY 8085 PROJECT

January 2024

FRANCISCO MAGALHÃES - 20221883
GONÇALO FARINHA - 20221871
MARTIM PIRES - 20221939
SAZONOV SEMEN - 20221689

PROJECT OBJECTIVES

The project's primary objective is to develop an algorithm that can decrypt messages encrypted with a Caesar cipher without knowing the key beforehand. This involves understanding the cipher's mechanics, where each letter in the message is shifted by a fixed number. The algorithm must efficiently detect the specific word "OTA" in the encrypted message, a known element in all communications of interest. Once "OTA" is identified, the program should deduce the key used for the cipher and then decrypt the entire message using this key.

The decrypted message will replace the original encrypted message, and the key will be stored for future reference. Secondary objectives include ensuring the program is user-friendly, well-documented, and robust enough to handle various message types and lengths. It should also be tested thoroughly to confirm accuracy and efficiency. Finally, the program should be designed with scalability in mind to adapt to potential future needs, such as different encryption schemes or larger messages.

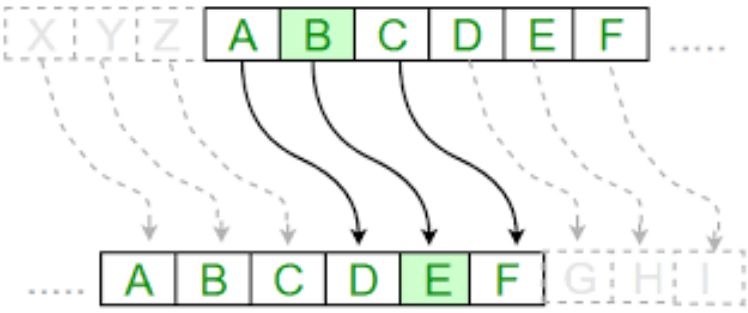

Fig. 1 - Caesar's Cipher visual explanation

APPROACH BREAKDOWN

1. Understanding the Problem :

- Objective: Detect the word "OTA" in a message encoded with a Caesar cipher and decode the entire message.
- Knowns: The word "OTA" will appear in the ciphered text. The Caesar cipher shifts all letters by a certain key (unknown).
- Deliverables: The decoded message and the key used for the cipher.

2. Research and Analysis :

- Caesar Cipher Basics: Understand how the Caesar cipher works—shifting each letter in the plaintext by a fixed number of positions down the alphabet.
- Frequency Analysis: Since the key is unknown, consider frequency analysis or known-plaintext attacks was our main strategy.

3. Planning the Solution :

- Algorithm Design:
  - Detecting "OTA": Iterate through the provided inputs if 3 consecutive characters have the diference of -5 and +19 assuming the input is given in ASCII code, then"OTA" is found.
  - Decoding: Once the correct shift is found (by subtracting the value of the last encoded character by 65, which represents "A" in ASCII). Apply it to the entire message to decode it.
  - Key Storage: Store the key (shift value) that successfully revealed "OTA" in the accumulator.

4. Development :

- As required our program is formated in a main routine that contains:
  - "BLOCKLOOP" to find the word "OTA" and the key of the cipher.
  - "DECODELOOP" to interate through the inputed characters so they can be decoded.
  - "END" just to store the key in the accumulator and halt the program.
- Additionally there are sub-routines that are called in the main routine:
  - "COMPARE" to compare data from the memory.
  - "DECODE" that decodes the character and stores in the respective memory address.

5. Implementation Details :

- To ensure the effectiveness of the program, we had to make sure that the program can function with negative and positive keys, this means that the cipher can move 2 letters infront or behind. Also if the carachets after decoding are outside the range of the ASCII table it is assumed that it loops the table (ex.: key: “3”, “Y”= (9 decoded character “89” -> “92” is outside of the range of 90, 92-90 = 2, the first letter is A = 65, so 65+2 = 67 -> “C”).
- We also made sure the program could be cuted and pasted in any Assembly 8085 simulator and works properly.

6. Testing :

- Test Cases: Create a variety of encoded messages (with known keys) to ensure the program can accurately detect "OTA" and decode messages.
- Edge Cases: Test how the program handles messages where "OTA" occurs multiple times or with different casing.
- Simulator: We used two different simulator softwares, an online format and a github project, “https://www.sim8085.com/” and “https://github.com/8085simulator/8085simulator.github.io” respectivily.

7. Documentation :

- The code is well documented explaining each step and dividing clearly the sub-routines and the main routine.

FLUXOGRAM

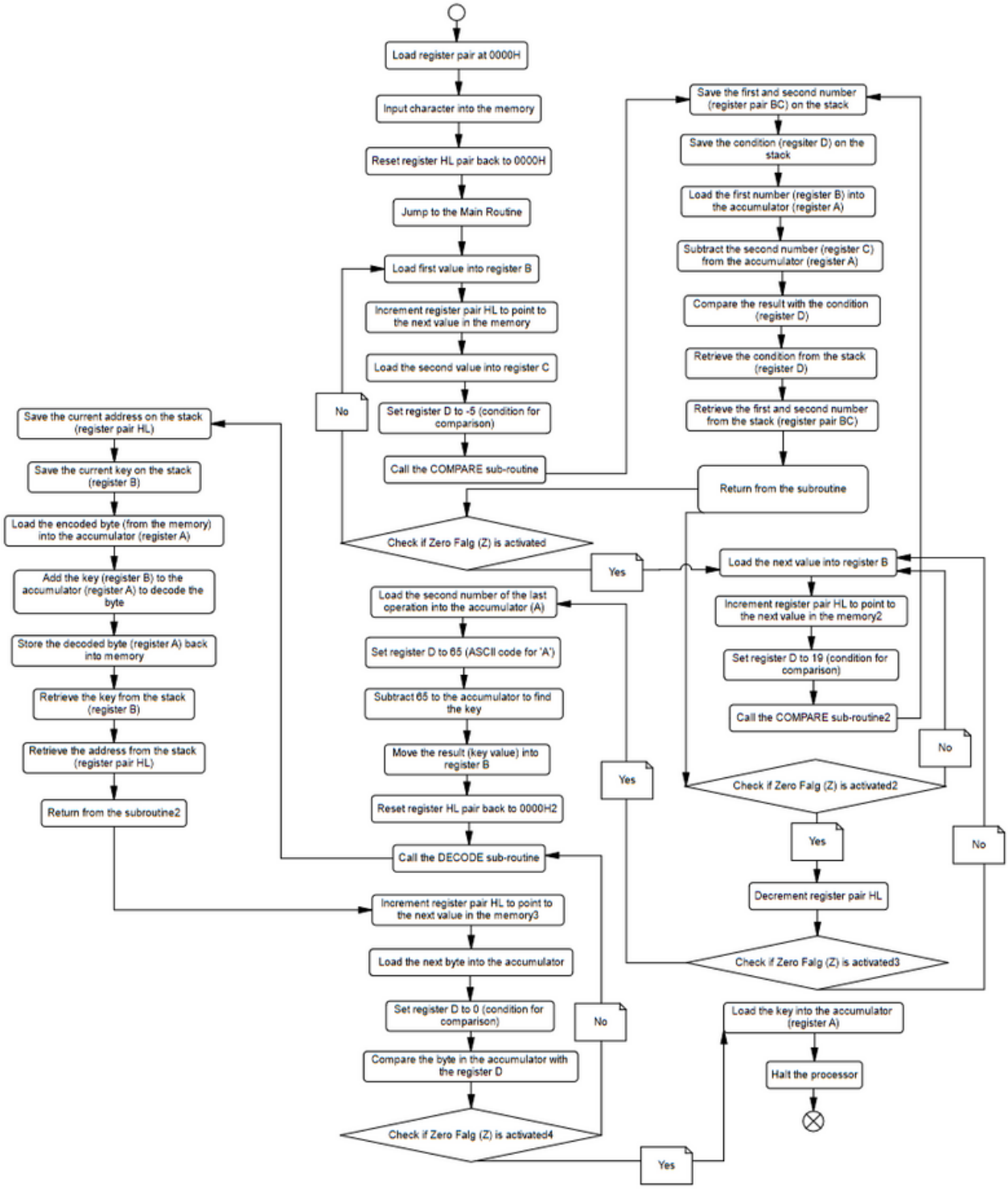

Fig. 2 - Fluxogram of the project

CODE

```
; Initialize HL register pair to address 0000H
LXI H, 0000H

; The following blocks load ASCII values into memory sequentially, representing an encoded message.
; Each 'MVI M,xx' instruction sets the memory location pointed to by HL to a specific ASCII value.
; 'INX H' increments the HL register pair to point to the next memory location.

MVI M,90  ; Load ASCII for 'Z' into memory
INX H     ; Increment HL to point to the next memory location

MVI M,72  ; Load ASCII for 'H'
INX H

MVI M,20  ; Load ASCII for ' ' (space)
INX H

MVI M,82  ; Load ASCII for 'R'
INX H

MVI M,87  ; Load ASCII for 'W'
INX H

MVI M,68  ; Load ASCII for 'D'
INX H

MVI M,20  ; Load ASCII for ' ' (space)
INX H

MVI M,75  ; Load ASCII for 'K'
INX H

MVI M,68  ; Load ASCII for 'D'
INX H
```

CODE

```
MVI M,87  ; Load ASCII for 'W'
INX H

MVI M,72  ; Load ASCII for 'H'
INX H

MVI M,20  ; Load ASCII for ' ' (space)
INX H

MVI M,66  ; Load ASCII for 'B'
INX H

MVI M,72  ; Load ASCII for 'H'
INX H

MVI M,86  ; Load ASCII for 'V'
INX H

MVI M,20  ; Load ASCII for ' ' (space)
INX H

MVI M,82  ; Load ASCII for 'R'
INX H

MVI M,87  ; Load ASCII for 'W'
INX H

MVI M,68  ; Load ASCII for 'D'
INX H

; Reset HL to point to the start of the encoded message
LXI H, 0000H
; Jump to the main routine to start processing
JMP BLOCKLOOP
```

CODE

```
; Sub-routine: DECODE
; Decodes the number and stores it in the respective memory address
; Parameters: H -> Memory address, B -> Key
; Return: None
DECODE: PUSH H ; Save the current address on the stack
PUSH B ; Save the current key on the stack
MOV A, M ; Load the encoded byte into the accumulator
SUB B ; Add the key to the accumulator (decode the byte)
MOV M, A ; Store the decoded byte back into memory
POP B ; Retrieve the key from the stack
POP H ; Retrieve the address from the stack
RET ; Return from the subroutine

; Sub-routine: COMPARE
; Compares two consecutive numbers and checks if the conditions are met
; Parameters: B -> First number, C -> Second number, D -> Condition
; Return: Z flag (Yes if the conditions are met, No otherwise)
COMPARE: PUSH B ; Save the first and the second number on the stack
PUSH D ; Save the condition on the stack
MOV A, B ; Load the first number into the accumulator
SUB C ; Subtract the second number from the accumulator
CMP D ; Compare the result with the condition
POP D ; Retrieve the condition from the stack
POP B ; Retrieve the first and the second number from the stack
RET ; Return from the subroutine
```

CODE

```
; Main Routine
BLOCKLOOP: MOV B, M ; Load the next value into register B
INX H ; Increment HL to point to the next memory location
MOV C, M ; Load the next value into register C
MVI D, -5 ; Set D to -5 (condition for comparison)
CALL COMPARE ; Call the COMPARE subroutine
JNZ BLOCKLOOP ; If the zero flag is not set, jump back to BLOCKLOOP
MOV B, M ; Load the next value into register B
INX H ; Increment HL
MOV C, M ; Load the next value into register C
MVI D, 19 ; Set D to 19 (condition for comparison)
CALL COMPARE ; Call the COMPARE subroutine
DCX H ; Decrement HL
JNZ BLOCKLOOP ; If the zero flag is not set, jump back to BLOCKLOOP

MOV A, C ; Load the second number into the accumulator
MVI D, 65 ; Set D to 65 (ASCII for 'A')
SUB D ; Subtract 65 from the accumulator
MOV B, A ; Move the result to register B

LXI H, 0000H ; Set HL to 0000H

; Loop to decode the inputted phrase
DECODELOOP: CALL DECODE ; Call the DECODE subroutine
INX H ; Increment HL
MOV A, M ; Load the next byte into the accumulator
MVI D, 0 ; Set D to 0 (condition for comparison)
CMP D ; Compare the byte with 0
JZ END ; If the byte is 0, jump to END
JMP DECODELOOP ; Otherwise, jump back to DECODELOOP

END: MOV A, B ; Load the key into the accumulator
HLT ; Halt the processor
```

MEMORY

sim8085 memory addresses view

- Inputed characters:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	5A	48	14	52	57	44	14	4B	44	57	48	14	42	48	56	14
001	52	57	44	00	00	00	00	00	00	00	00	00	00	00	00	00

Fig. 3 - sim8085 memoy adresses view

- Program output (decoded characters):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	5D	4B	17	55	5A	47	17	4E	47	5A	4B	17	45	4B	59	17
001	55	5A	47	00	00	00	00	00	00	00	00	00	00	00	00	00

Fig. 4 - sim8085 memoy addresses view

8085 compiler memory map OPCODE view

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	21	00	00	36	90	23	36	72	23	36	20	23	36	82	23	36
0010	87	23	36	68	23	36	20	23	36	75	23	36	68	23	36	87
0020	23	36	72	23	36	20	23	36	66	23	36	72	23	36	86	23
0030	36	20	23	36	82	23	36	87	23	36	68	23	21	00	00	C3
0040	52	00	E5	C5	7E	80	77	C1	E1	C9	C5	D5	78	91	BA	D1
0050	C1	C9	46	23	4E	16	00	CD	4A	00	C2	52	00	46	23	4E
0060	16	19	CD	4A	00	2B	C2	52	00	79	16	65	92	47	21	00
0070	00	CD	42	00	23	7E	16	00	BA	CA	7F	00	C3	71	00	78
0080	76	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Fig. 5 - 8085 compiler memoy map view

MEMORY MAP

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0000		LXI H,0000	21	3	3	10
	0001			00			
	0002			00			
√	0003		MVI M,90	36	2	3	10
	0004			90			
√	0005		INX H	23	1	1	6
√	0006		MVI M,72	36	2	3	10
	0007			72			
√	0008		INX H	23	1	1	6
√	0009		MVI M,20	36	2	3	10
	000A			20			
√	000B		INX H	23	1	1	6
√	000C		MVI M,82	36	2	3	10
	000D			82			
√	000E		INX H	23	1	1	6
√	000F		MVI M,87	36	2	3	10
	0010			87			
√	0011		INX H	23	1	1	6
√	0012		MVI M,68	36	2	3	10

Fig. 6 - Program memoy map view

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0012		MVI M,68	36	2	3	10
	0013			68			
√	0014		INX H	23	1	1	6
√	0015		MVI M,20	36	2	3	10
	0016			20			
√	0017		INX H	23	1	1	6
√	0018		MVI M,75	36	2	3	10
	0019			75			
√	001A		INX H	23	1	1	6
√	001B		MVI M,68	36	2	3	10
	001C			68			
√	001D		INX H	23	1	1	6
√	001E		MVI M,87	36	2	3	10
	001F			87			
√	0020		INX H	23	1	1	6
√	0021		MVI M,72	36	2	3	10
	0022			72			
√	0023		INX H	23	1	1	6
√	0024		MVI M,20	36	2	3	10
	0025			20			

Fig. 7 - Program memoy map view

MEMORY MAP

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0024		MVI M,20	36	2	3	10
	0025			20			
√	0026		INX H	23	1	1	6
√	0027		MVI M,66	36	2	3	10
	0028			66			
√	0029		INX H	23	1	1	6
√	002A		MVI M,72	36	2	3	10
	002B			72			
√	002C		INX H	23	1	1	6
√	002D		MVI M,86	36	2	3	10
	002E			86			
√	002F		INX H	23	1	1	6
√	0030		MVI M,20	36	2	3	10
	0031			20			
√	0032		INX H	23	1	1	6
√	0033		MVI M,82	36	2	3	10
	0034			82			
√	0035		INX H	23	1	1	6
√	0036		MVI M,87	36	2	3	10

Fig. 8 - Program memoy map view

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0036		MVI M,87	36	2	3	10
	0037			87			
√	0038		INX H	23	1	1	6
√	0039		MVI M,68	36	2	3	10
	003A			68			
√	003B		INX H	23	1	1	6
√	003C		LXI H,0000	21	3	3	10
	003D			00			
	003E			00			
√	003F		JMP BLOCKL...	C3	3	3	10
	0040			52			
	0041			00			
√	0042	DECODE	PUSH H	E5	1	3	12
√	0043		PUSH B	C5	1	3	12
√	0044		MOV A,M	7E	1	2	7
√	0045		ADD B	80	1	1	4
√	0046		MOV M,A	77	1	2	7
√	0047		POP B	C1	1	3	10
√	0048		POP H	E1	1	3	10
√	0049		RET	C9	1	3	10

Fig. 9 - Program memoy map view

MEMORY MAP

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0048		POP H	E1	1	3	10
√	0049		RET	C9	1	3	10
√	004A	COMP...	PUSH B	C5	1	3	12
√	004B		PUSH D	D5	1	3	12
√	004C		MOV A,B	78	1	1	4
√	004D		SUB C	91	1	1	4
√	004E		CMP D	BA	1	1	4
√	004F		POP D	D1	1	3	10
√	0050		POP B	C1	1	3	10
√	0051		RET	C9	1	3	10
√	0052	BLOCK...	MOV B,M	46	1	2	7
√	0053		INX H	23	1	1	6
√	0054		MOV C,M	4E	1	2	7
√	0055		MVI D,-5	16	2	2	7
	0056			00			
√	0057		CALL COMP...	CD	3	5	18
	0058			4A			
	0059			00			
√	005A		JNZ BLOCKL...	C2	3	3	10
	005B			52			

Fig. 10 - Program memoy map view

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	005A		JNZ BLOCKL...	C2	3	3	10
	005B			52			
	005C			00			
√	005D		MOV B,M	46	1	2	7
√	005E		INX H	23	1	1	6
√	005F		MOV C,M	4E	1	2	7
√	0060		MVI D,19	16	2	2	7
	0061			19			
√	0062		CALL COMP...	CD	3	5	18
	0063			4A			
	0064			00			
√	0065		DCX H	2B	1	1	6
√	0066		JNZ BLOCKL...	C2	3	3	10
	0067			52			
	0068			00			
√	0069		MOV A,C	79	1	1	4
√	006A		MVI D,65	16	2	2	7
	006B			65			
√	006C		SUB D	92	1	1	4
√	006D		MOV B,A	47	1	1	4

Fig. 11 - Program memoy map view

MEMORY MAP

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	006C		SUB D	92	1	1	4
√	006D		MOV B,A	47	1	1	4
√	006E		LXI H,0000	21	3	3	10
	006F			00			
	0070			00			
√	0071	DECOD...	CALL DECODE	CD	3	5	18
	0072			42			
	0073			00			
√	0074		INX H	23	1	1	6
√	0075		MOV A,M	7E	1	2	7
√	0076		MVI D,00	16	2	2	7
	0077			00			
√	0078		CMP D	BA	1	1	4
√	0079		JZ END	CA	3	3	10
	007A			7F			
	007B			00			
√	007C		JMP DECOD...	C3	3	3	10
	007D			71			
	007E			00			
√	007F	END	MOV A,B	78	1	1	4

Fig. 12 - Program memoy map view

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
	006F			00			
	0070			00			
√	0071	DECOD...	CALL DECODE	CD	3	5	18
	0072			42			
	0073			00			
√	0074		INX H	23	1	1	6
√	0075		MOV A,M	7E	1	2	7
√	0076		MVI D,00	16	2	2	7
	0077			00			
√	0078		CMP D	BA	1	1	4
√	0079		JZ END	CA	3	3	10
	007A			7F			
	007B			00			
√	007C		JMP DECOD...	C3	3	3	10
	007D			71			
	007E			00			
√	007F	END	MOV A,B	78	1	1	4
√	0080		HLT	76	1	2	5

Fig. 13 - Program memoy map view

CONCLUSION

This project has successfully accomplished the development of a decryption algorithm for the Caesar cipher within the Assembly 8085 environment. Our primary achievement was the capability to decrypt messages, specifically identifying and decoding the word "OTA" without the original encryption key. This was made possible through sophisticated ASCII character handling, where the algorithm adeptly managed character shifts and addressed boundary cases. This ensured precise decryption for both positive and negative key values.
The structure of the program was thoughtfully designed, with a modular framework that included a main routine and several sub-routines. This modular design not only streamlined the programming process but also significantly enhanced the maintainability and scalability of the code, making it easier to debug and update.

A key aspect of our project was rigorous testing, which was performed using various simulator software. This testing phase was critical in verifying the consistency and reliability of the algorithm across different types of encrypted messages. It confirmed the robustness of our solution in a variety of scenarios.
One of the overarching challenges we faced was optimizing the algorithm to function within the resource constraints of Assembly 8085. This involved managing limited memory and processing capabilities, yet our algorithm was developed to operate efficiently within these limitations.

Looking forward, the project lays a solid groundwork for future enhancements. There is potential to expand the algorithm to accommodate other encryption methods, which would widen its scope of application. Optimizing the algorithm for larger datasets is another area for development, enhancing its usefulness for more extensive decryption tasks. Moreover, adapting the algorithm for real-time decryption could open up new practical applications, marking a significant step forward in the field of cryptographic algorithms.

In conclusion, this project stands out for its effective navigation of the intricacies of Assembly 8085 programming and cryptography. It delivers a capable and efficient decryption tool for the Caesar cipher, setting a strong foundation for future advancements in cryptographic algorithm development.

HEX CODE

21 00 00 36 90 23 36 72 23 36 20 23 36 82 23 36 87 23 36 68 23 36 20 23 36 75 23 36 68 23 36 87 23 36 72 23 36 20 23 36 82 23 36 87 23 36 68 23 21 00 00 C3 52 00 E5 C5 7E 80 77 C1 E1 C9 C5 D5 78 91 BA D1 C1 C9 46 23 4E 16 00 CD 4A 00 C2 52 00 46 23 4E 16 19 CD 4A 00 2B C2 52 00 79 16 65 92 47 21 00 00 CD 42 00 23 7E 16 00 BA CA 7F 00 C3 71 00 78 76

Note: The OPCODE in red is only the input text message to test the program, this means that is not essential for the program to work, but merely to demonstrate the procedures.

Memory Adresses

The memory adresses used in the test is between 0000H and 0013H. The program starts at 0000H and the range is set by the number of characters inputed, if the message is 80 characters long it will range between 0000H and 0080H.