**Table of Content:**

**Objective:**

**Introduction:**

 **Methodology**

* Description of the Vigenere Cipher algorithm
* Steps involved in encryption and decryption
* Tools and technologies used

 **Implementation**

* Detailed description of your implementation process
* Code snippets (if applicable)
* Screenshots or outputs of the implementation

**Conclusion:**

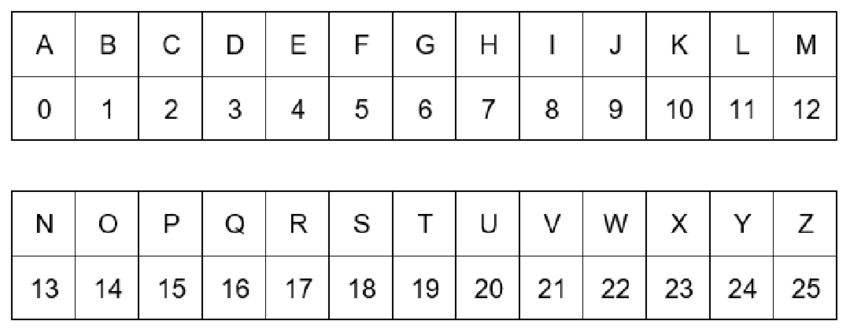
**References:**

this is my explanation of encoding just it in report

**Encoding:**

**Methodology:**We use the (p + k) mod 26 formula to encode character by character. Here p is the character form the plain text and k is character form the key. We also use key in repeated mode, which means that when we reach the end of the key and plain text is still remaining to encode, we go to the start key and repeat the same process until we all the plaintext is encoded.

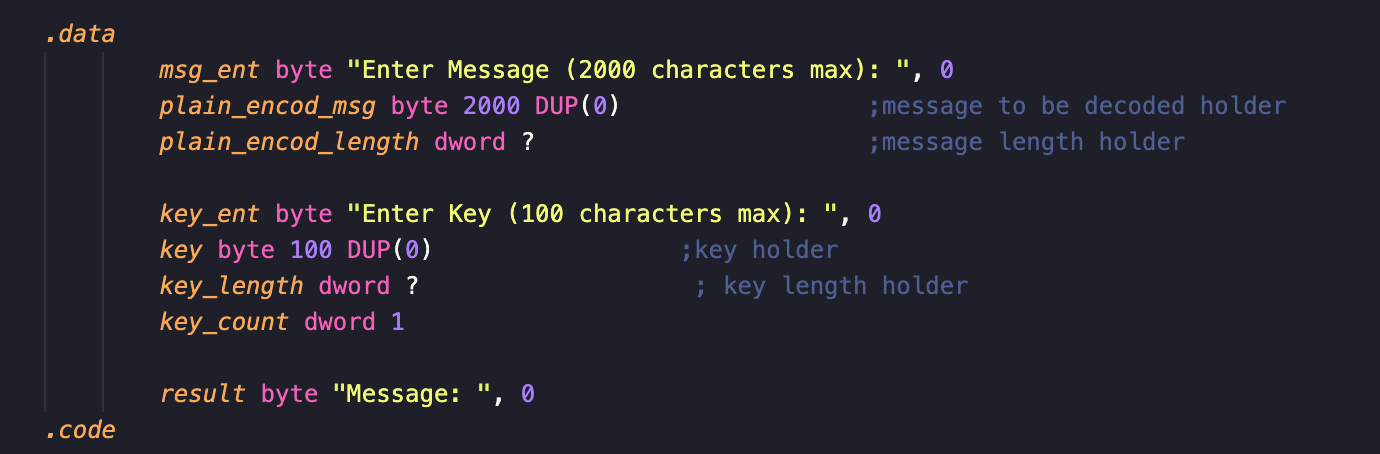
Before encoding we convert both key and plaintext to upper case, it makes applying this formula much more easily. During encoding, after moving a character form key and plaintext into the register we subtract 65 form both register (capital alphabets in ASCII code starts form 65) because this formula is applied on alphabet with zero index. This help us to each this;



Then we apply the formula (p + k) mod 26 and after applying the formula we again add 65 to the resultant value to print that character on console.

**Code Snippets:**

**Variable and Array Declaration:**

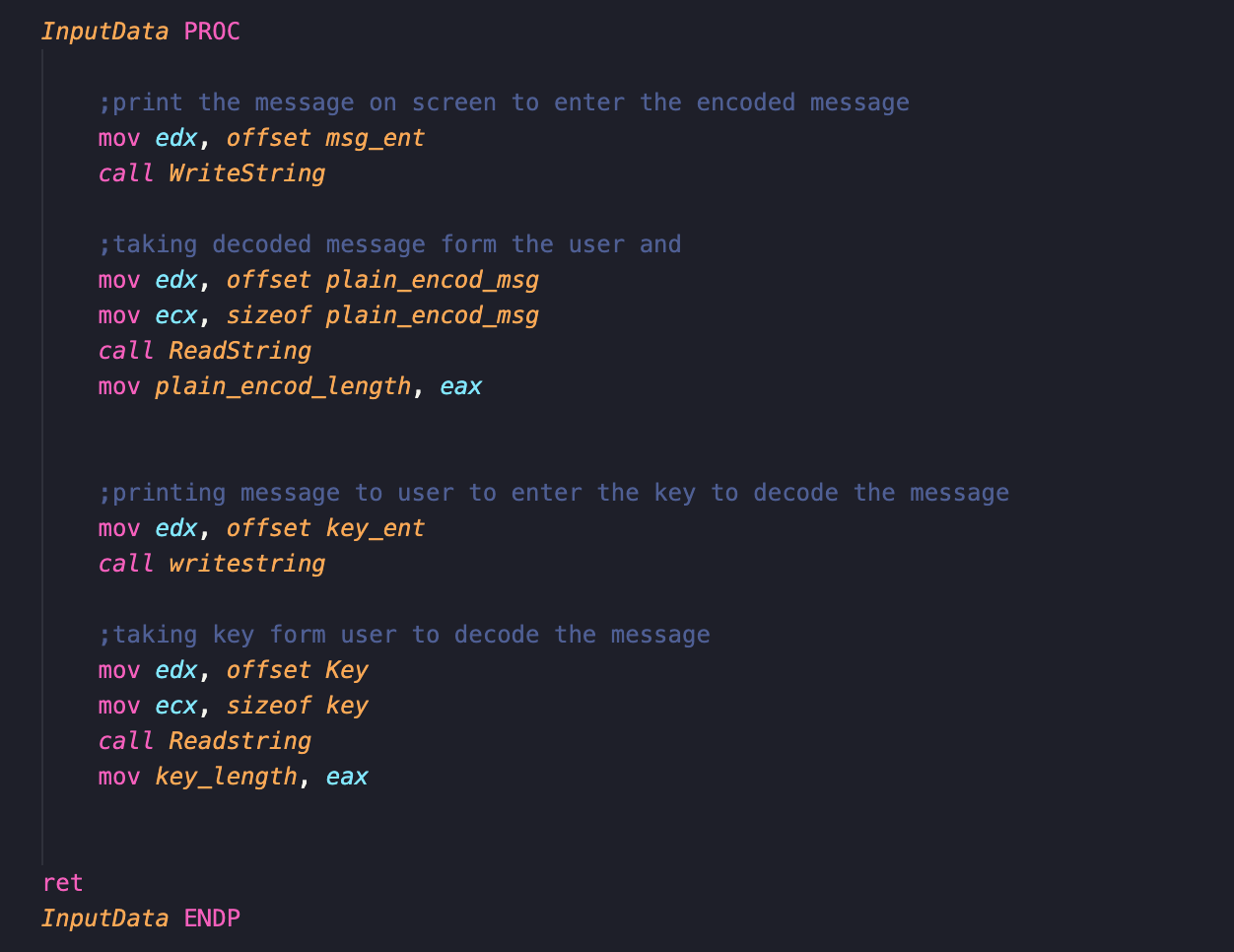


In this code snippet, I declared the array for plaintext which can hold up to 2000 characters it can be changed later.

In **msg\_ent,** I’m printing a message user to enter the plaintext and tell him the length of the characters that he can enter, **plain\_encod\_length** stores the number of characters that user entered.

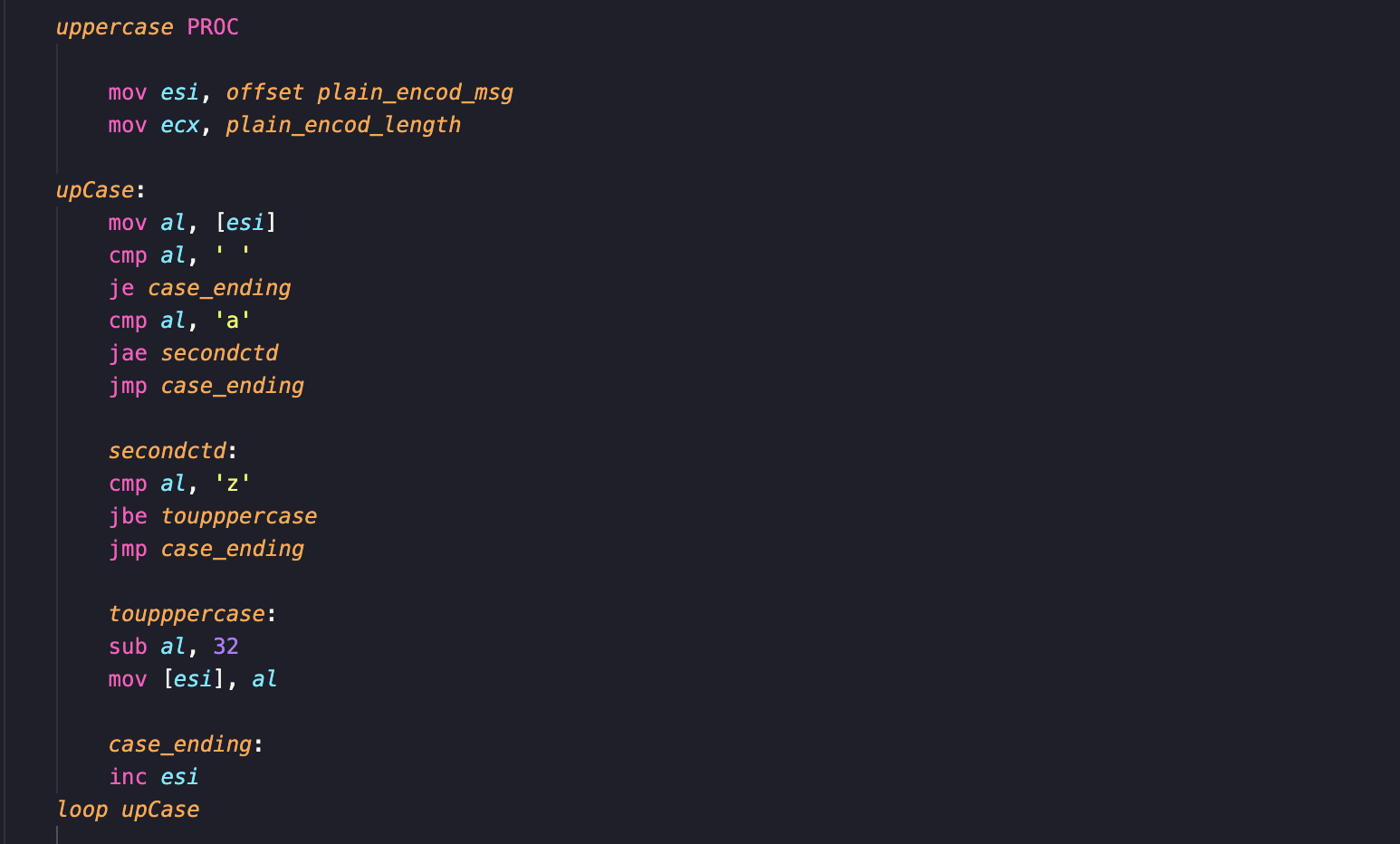
For key, I’ve a message in **key\_ent** that enter key and telling him the maximum length of key which is 100 characters and it can also be change late on. The **key\_length** stores the length of characters that user entered and **key\_count** is the counter which help us to achieve the key repeated mode as you will see in later code snippets. The **result** message will used when printing encoded text to user.

**Taking Input form User**



This **InputData** is the procedure that will take the take the input form the user (plaintext and key). Frist we are printing the message to user that we stored in above code snippets (“Enter Message (2000 characters max)”) then calling Readstring to get string form user, before calling ReadString we need to move the address of the array where value will be store to edx and maximum number of characters that user can enter in ecx. After calling ReadString, we move the value in eax (number of characters that user entered) in variable **plain\_encod\_length.** We follow the same procedure while taking taking the key as input form user.

**Converting Small characters in Input to Captial:**



This the first part of the **uppercase** procedure, in which we will convert all the small characters of plaintext into the capital alphabets. In this code snippet, we first move the address of array where plaintext is to esi and number of characters in array to ecx which will act as loop counter for upCase loop.

Frist we move the value character into al, first checking if it is space then skip this and increment, otherwise check if value in al is greater than equal to ’a’ and value in al is less than equal to ‘z’ (al>=’a’ && al<=’z’) and I’m using short circuit evaluation for this, if one condition is false it will jump out of the condition.

If the condition is true, it will convert the small alphabet to capital by subtracting 32 form it because in ASCII code the difference between small and capital character is 32. Suppose ‘A’ is at 65 and ‘a’ is at 97 in ASCII code (97-32 = 65). This will convert any small characters in plaintext into the capital characters.

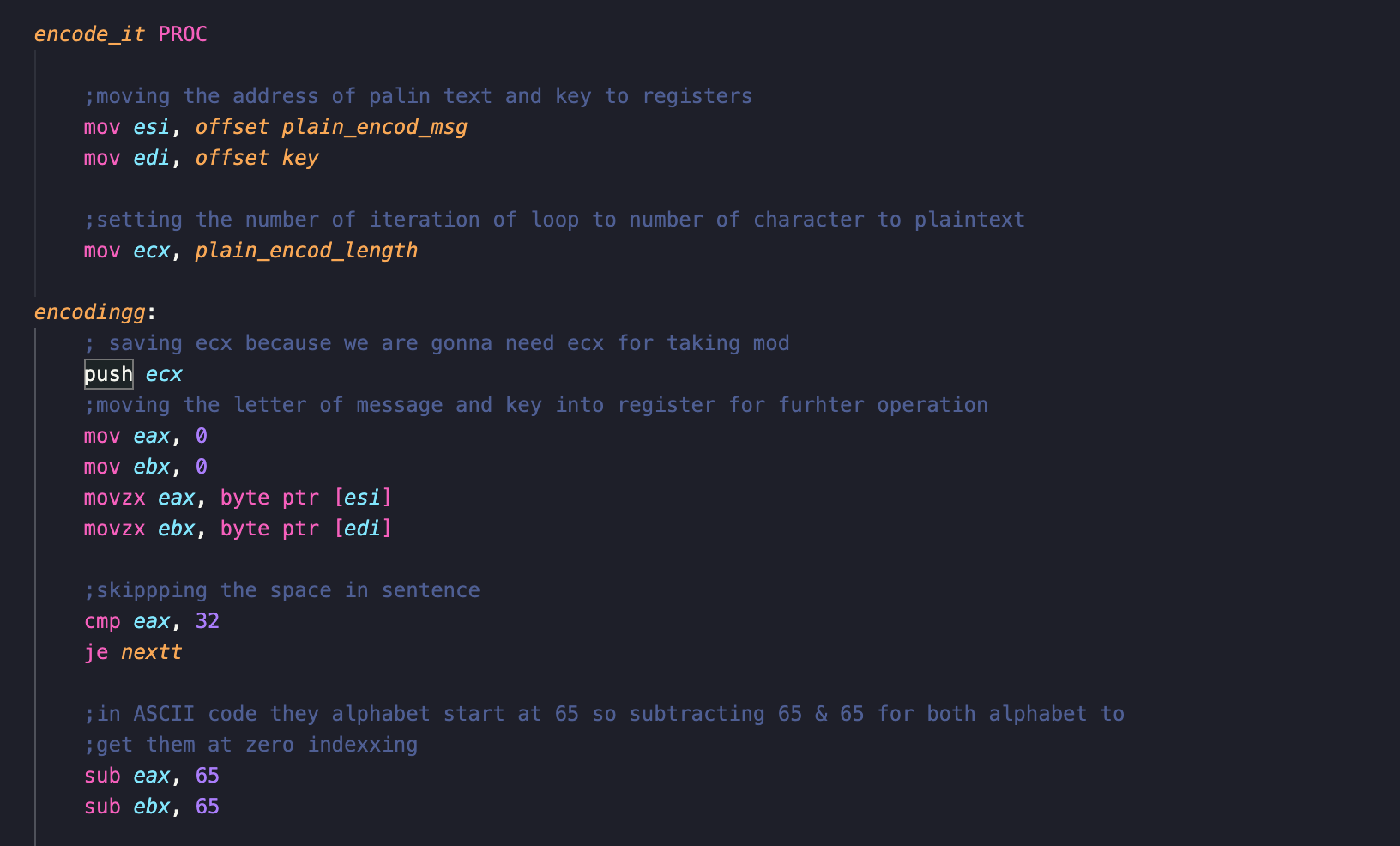
Once the number is converted into capital we are over writing it on original value in the array.



In this part of **uppercase** procedure, I’m converting any small characters in key to upper case.

Firstly, I’m moving address of key array to esi and key length to loop counter ecx. Then I’m moving the character of key to al and using the same condition we used when converting the plaintext to capital (mentioned above) if the character is between a-z then subtract 32 form it and it will convert it into the capital character. I’m also over writing the original value in array.

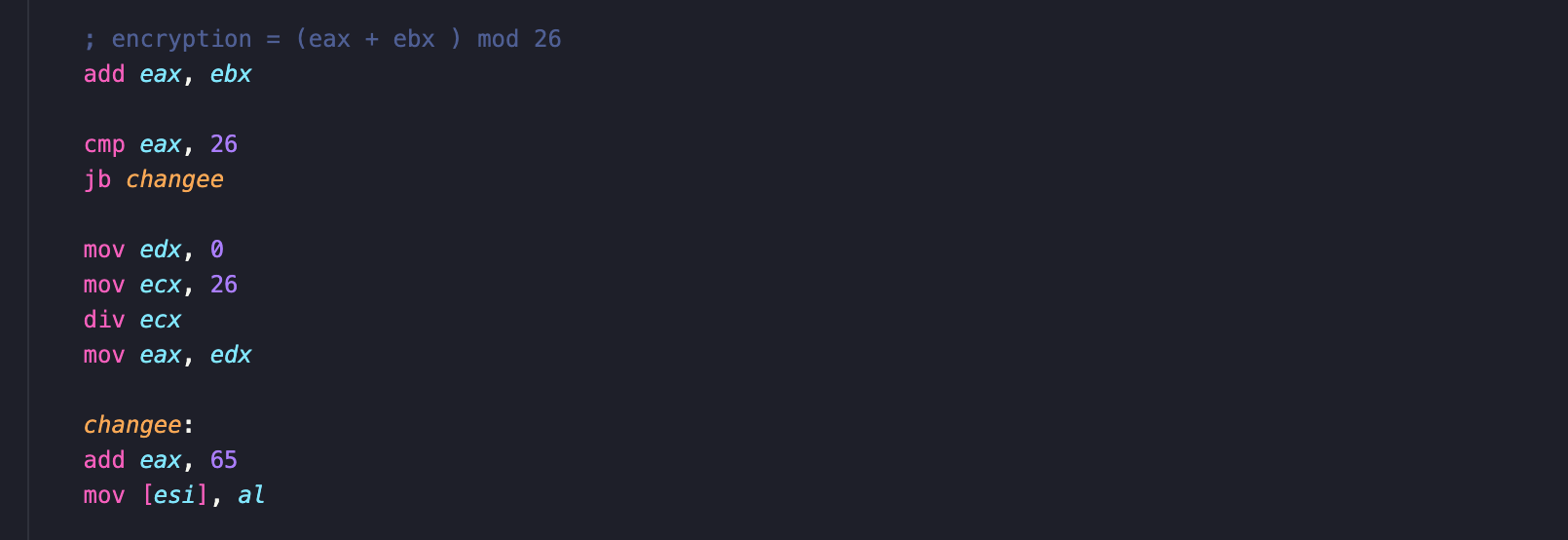
**Encoding Function:**



The **encode\_it** is the function where all encoding happens. Firstly, we move the address of plaintext to esi and key’s address to edi, also move the length of plaintext to loop counter ecx. Because we only want to execute loop as long as plaintext.

The **encodingg** is the loop where operation happens. Frist we push the ecx into stack because we will need a register when performing calculation so we will be reusing the ecx but we don’t want to lose the loop counter. Then clearing the eax and ebx and then moving the plaintext character into eax and key character to ebx. I used movzx (extended extra bits as zero) and byte ptr when moving value form array to register because the character of 8 byte and register is of 32 bits.

Then comparing then it will 32 which is space, if it equal to space then jump to next character otherwise subtract 65 form both eax and ebx, which will move the characters to according zero index as I mentioned above. It will allow us to apply the formula (p + k) mod 26.

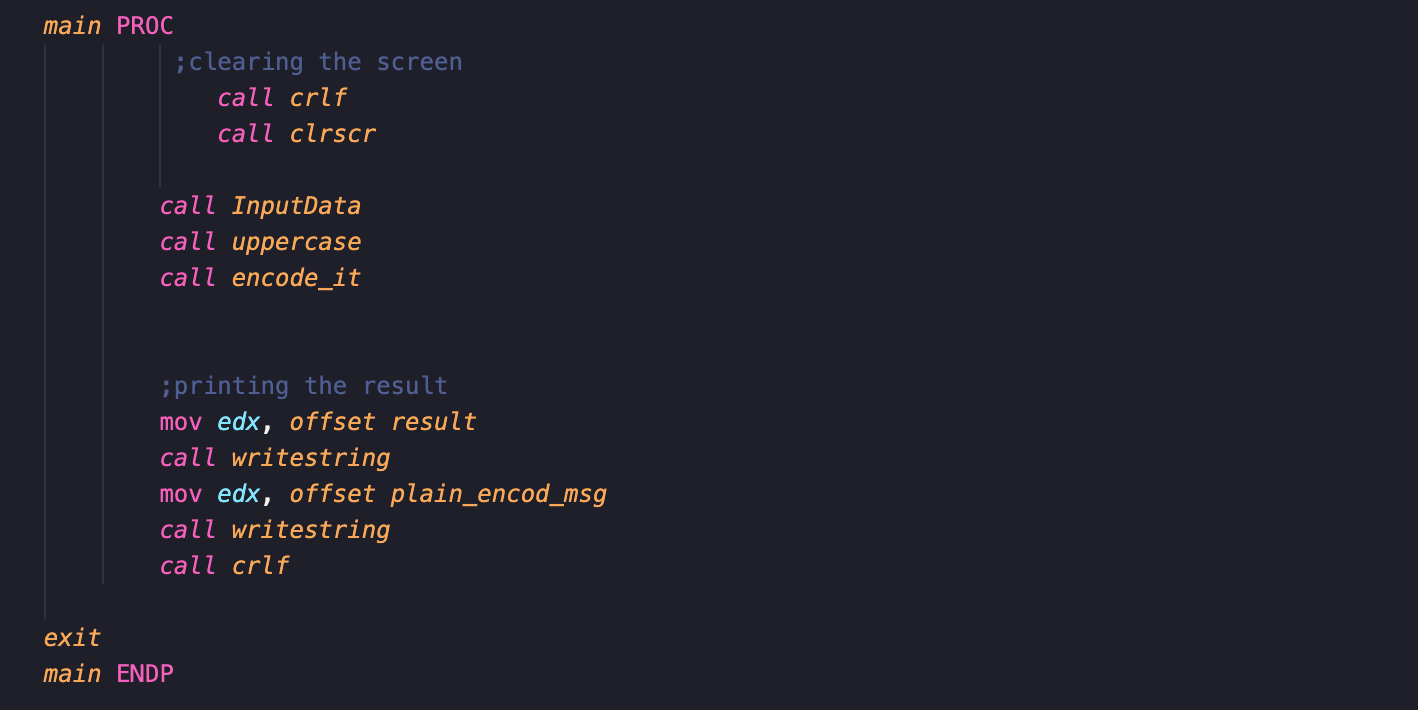


After subtracting 65 form both register then I add them, If the resultant value is greater than 26, then clearing the edx before division because that’s where remainder will be stored, moving the divider (26) to ecx that’s why I pushed value of ecx in stack earlier, eax has the value. Div will divide the eax by ecx and store the remainder in edx. That we will move in eax.

If the value is less than 26 then don’t need to perform any division. Then add 65 to eax to get the character we want in ASCII code and over writing it on original value in array.

A screenshot of a computer code

Description automatically generated

Frist incrementing the key pointer to next character also incrementing the **key\_count** variable. Then moving the **key\_count** value to eax and **key\_length** to ebx, then comparing them because if key counter is greater than key length then it means we have passed the end of key and reset the value, point the edi to beginning to key and move 1 to **key\_count.** If key counter is less than key length then jump to **ending\_encod.** Then we increment the esi (plaintext pointer) and pop the value of ecx and loop countineue until ecx is zero which means we have reached the end of plaintext.

This the **main** procedure where I will call the functions, first I’m clearing the screen then calling the **InputData** procedure to get the input form the user and then using **uppercase** procedure to convert the input data into capital letters. Then calling **encode\_it** function to encode the plaintext that user entered. Once that is done, I’m printing the “Message: “ which is store in result variable to console and in-front of printing the encoded message that is stored in **plain\_encode\_msg** using call writestring.