Course Code : CSE360

Course Title : Computer Architecture

Section : 01

Semester : Spring-2024

Project Report

Submitted to :

Dr. Md. Nawab Yousuf Ali

Professor

Department of Computer Science & Engineering

East West University

Submitted by :

Name	ID	Roll
Tasmia Islam	2021-2-60-062	26
Tasnim Israk Synthia	2021-2-60-097	31
Prinom Mojumder	2021-2-60-098	32
Jubaer Ahmed	2021-2-60-139	39
Bahauddin Ahmed	2020-1-60-271	3

Date of submission : 26th May, 2024

Title : Implement Matrix Multiplication Using 8085 Assembly Language

Objective :

- The main objective of this project is to implement an algorithm that multiplies two matrices and stores the resulting matrix.
- To demonstrate the steps involved in performing arithmetic operations on matrix elements, which are accessed and stored in memory, and managing the indices to ensure the correct elements are accessed and stored during the computation.

Theory :

Assembly Language : Assembly Language is a type of low-level programming language that is created to communicate directly with a computer's hardware.
Every computer has a microprocessor that manages the computer's arithmetic, logical and control activities. A processor only understands machine language instructions, which are strings of 1's and 0's and complex for use in software development. So, assembly language is designed to translate various instructions in symbolic code and to a more understandable form.

Advantages :

1. Execution may be more simple and faster compared to other languages
2. Requires less memory
3. Allows direct control over hardware

Disadvantages :

1. Syntax of Assembly Language is difficult
2. Not portable between machines

8085 Microprocessor : The 8085 microprocessor features an 8-bit architecture with a rich Instruction set for data manipulation and control operations.

Matrix : A matrix is a rectangular array or table containing numbers, symbols, or expressions.

Matrix Multiplication: It is the product of two matrices which produces a single matrix. It is a type of binary operation. To perform matrix multiplication, we should make sure that the number of columns of the 1st matrix is equal to the number of rows of the 2nd matrix. The algorithm is straightforward in high-level languages but needs careful handling of memory and registers in assembly language.

Formula :

Let's say A and B are two matrices, such that,

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ & \ldots\ldots\ldots\ldots & & \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1n} \\ B_{21} & B_{22} & \cdots & B_{2n} \\ & \ldots\ldots\ldots\ldots & & \\ B_{m1} & B_{m2} & \cdots & B_{mn} \end{bmatrix}$$

Then Matrix C = AB is denoted by

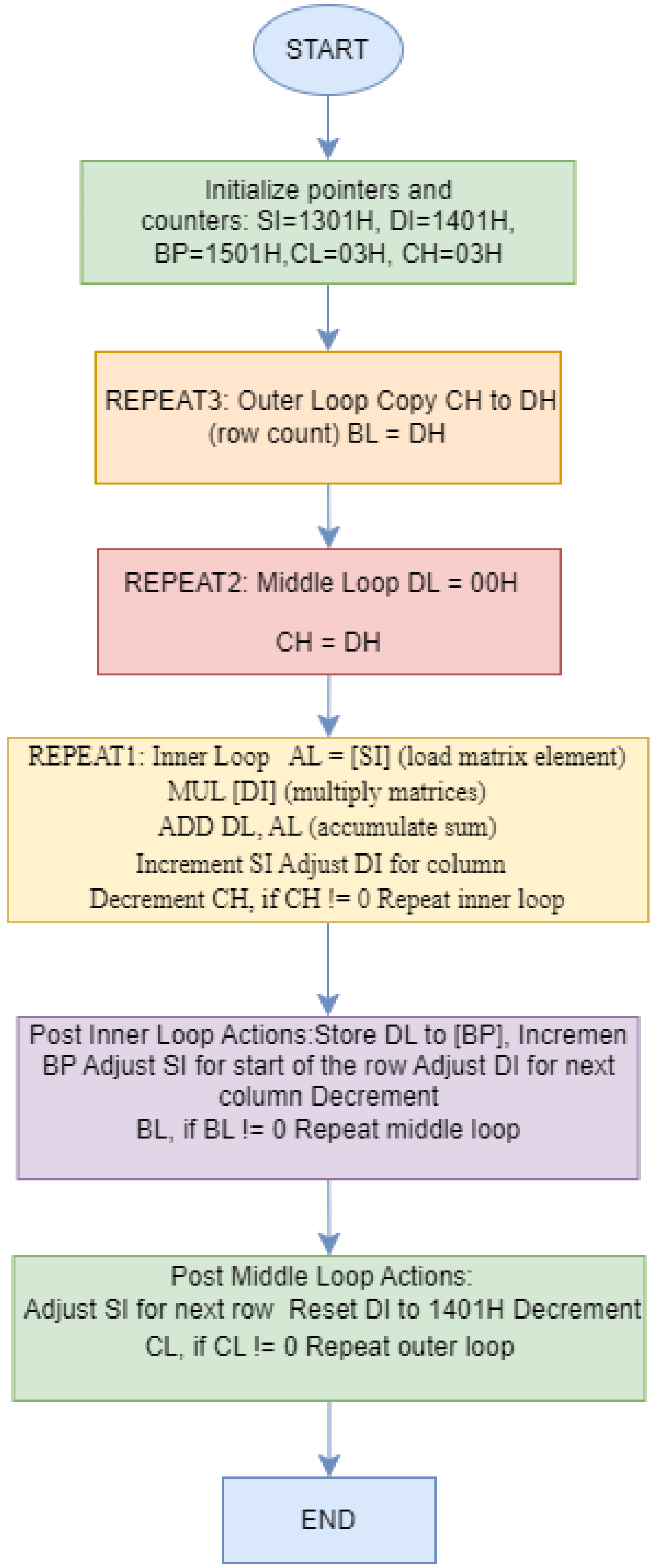
$$C = \begin{bmatrix} C_{11}C_{12}\ldots\ldots.C_{1c} \\ C_{21}C_{22}\ldots\ldots.C_{2c} \\ \ldots\ldots\ldots\ldots\ldots \\ C_{a1}C_{a2}\ldots\ldots.C_{ac} \end{bmatrix}$$

An element in matrix C where C is the multiplication of Matrix A X B.

$$C_{xy} = A_{x1}B_{y1} + \ldots . + A_{xb}B_{by} = \sum_{k=1}^{b} A_{xk}B_{ky}$$

For x = 1...... a and y= 1.......c

Design :


START
Initialize pointers and counters: SI=1301H, DI=1401H, BP=1501H,CL=03H, CH=03H
REPEAT3: Outer Loop Copy CH to DH (row count) BL = DH
REPEAT2: Middle Loop DL = 00H
CH = DH
REPEAT1: Inner Loop AL = [SI] (load matrix element)
MUL [DI] (multiply matrices)
ADD DL, AL (accumulate sum)
Increment SI Adjust DI for column
Decrement CH, if CH != 0 Repeat inner loop
Post Inner Loop Actions:Store DL to [BP], Incremen BP Adjust SI for start of the row Adjust DI for next column Decrement
BL, if BL != 0 Repeat middle loop
Post Middle Loop Actions:
Adjust SI for next row Reset DI to 1401H Decrement
CL, if CL != 0 Repeat outer loop
END

Implementation :

In our project we used the emu8086 simulator software to perform the matrix multiplication. Which is the updated version of the 8085 assembly language.

Source Code :

```
MOV SI,1301H        ; SET SI AS THE POINTER FOR FIRST INPUT MATRIX
MOV DI,1401H        ; SET DI AS THE POINTER FOR SECOND INPUT MATRIX
MOV BP,1501H        ; SET BP AS THE POINTER FOR PRODUCT MATRIX
MOV CL,03H          ; SET CL S COUNT FOR ELEMENTS IN A ROW
MOV CH,03H          ; SET CH S COUNT FOR ELEMENTS IN A COLUMN
MOV DH,CH
REPEAT3:
MOV BL,DH           ; COPY THE COLUMN COUNT IN BL REGISTER
REPEAT2:
MOV DL,00H          ; INTIALIZE SUM AS ZERO
MOV CH,DH           ; GET THE COULUMN COUNT IN DH
REPEAT1:
MOV AL,[SI]         ; GET ONE ELEMENT OF THE ROW IN AL REGISTER
MUL [DI]            ; GET THE PRODUCT OF ROW AND COLUMN ELEMENT IN AL
ADD DL,AL           ; ADD THE PRODUCT TO SUM
INC SI              ; INCREMENT THE FIRST INPUT MATRIX POINTER
ADD DI,03           ; LET DI POINTER TO NEXT ELEMENT OF SAME COLUMN OF 2ND MATRIX
DEC CH              ; DECREMENT OF COLUMN COUNT
JNZ REPEAT1         ; REPEAT MULTIPLICATION AND ADDITION UNTIL DH COUNT IS ZERO
MOV [BP],DL         ; STORE AN ELEMENT OF PRODUCT MATRIX IN MEMORY
INC BP              ; INCREMENT THE PRODUCT MATRIX POINTER
SUB SI,03H          ; MAKE SI TO POINT THE FIRST ELEMENT OF THE ROW
SUB DI,09H
INC DI
DEC BL              ; DECREMENT COLUMN COUNT
JNZ REPEAT2
ADD SI,03H          ; LET SI POINTER FIRST ELEMENT OF SECOND MATRIX
MOV DI,1401H        ; MAKE DI TO POINT TO FIRST ELEMENT OF 2 ND MATRIX
DEC CL              ; DECREMENT OF ROW COUNT
JNZ REPEAT3
HLT                 ; HALT AND EXECUTION
```

Fig-01 : Source code

Summary of the Code :

The code initializes pointers for the two input matrices and the product matrix. It uses three nested loops:

1. The outer loop (REPEAT3) iterates over rows of Matrix A.
2. The middle loop (REPEAT2) iterates over columns of Matrix B.
3. The inner loop (REPEAT1) iterates over elements in the current row of Matrix A and column of Matrix B, calculating the sum of products for each element of the product matrix.
4. The results of the multiplications are accumulated and stored in the product matrix.

Debugging-Test-Run :

1. In the emu8086 compiler, we emulate the code. Then it shows two interfaces : 'original source code' and 'emulator: project main code.bin_'.

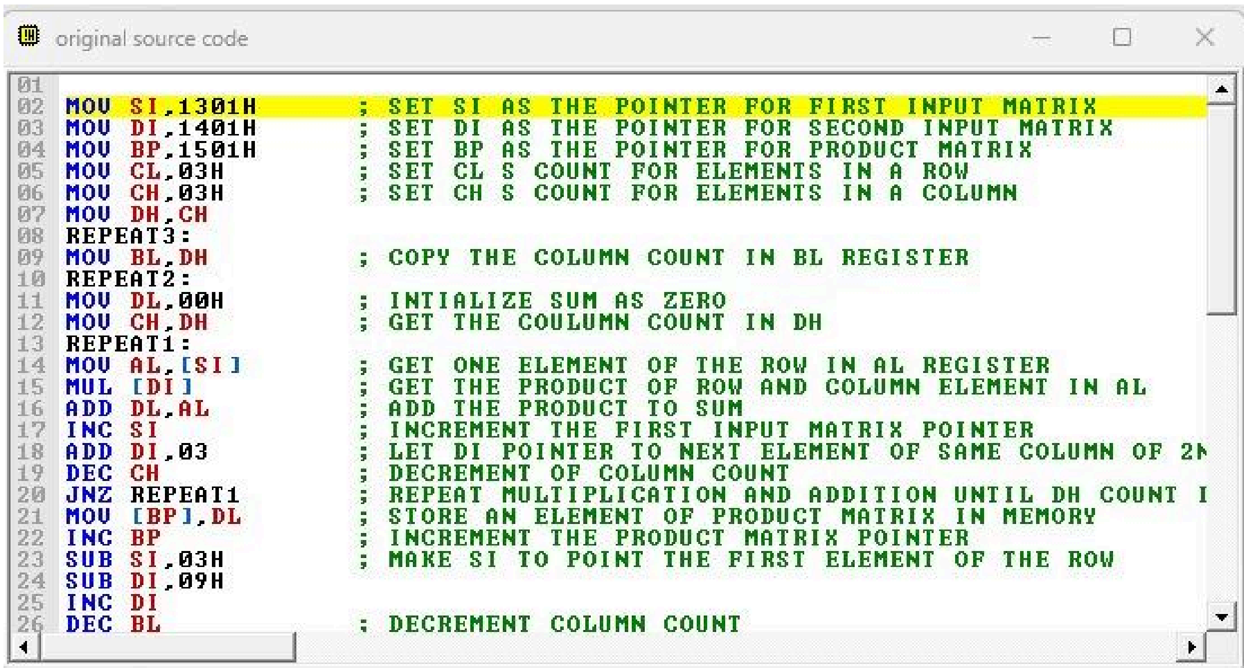


Fig-02 : 1st interface 'original source code'

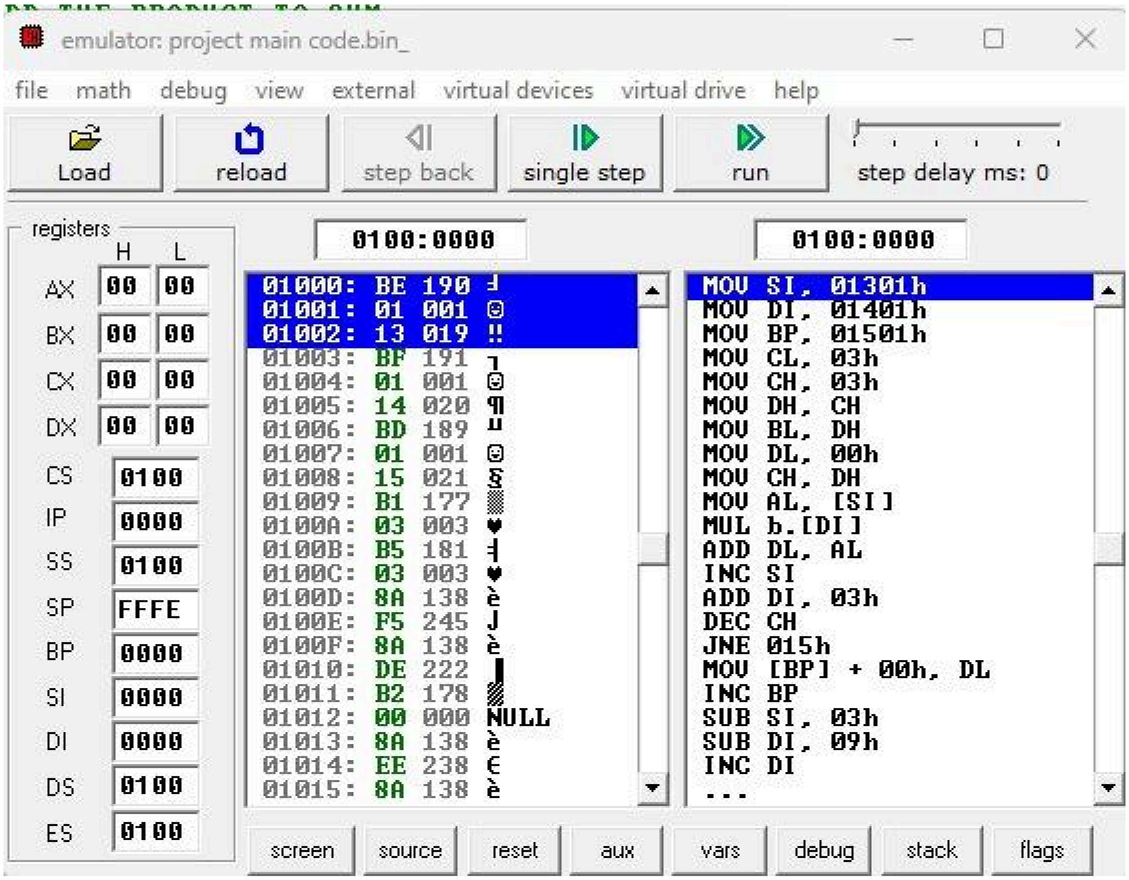
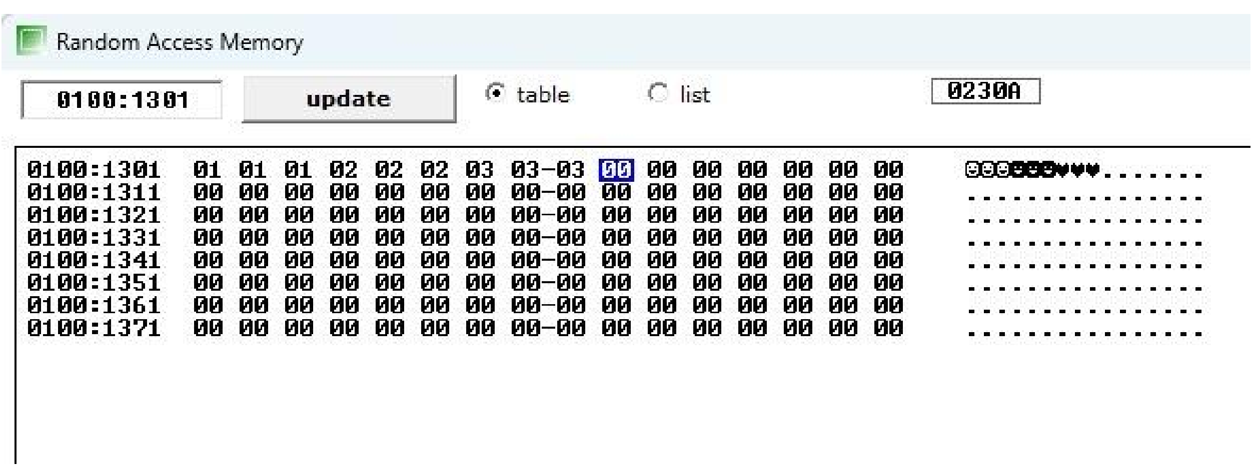


Fig-03 : 2nd interface ‘emulator: project main code.bin_’

2. Then from the ‘emulator: project main code.bin_’ interface we go to ‘view’, which leads to another interface ‘Random Access Memory’.
3. In the ‘Random Access Memory’ interface we update the matrix element in the designated address for the two matrices. Next, we go to the resultant matrix address update and click the ‘run’ button from the ‘emulator: project main code.bin_’.



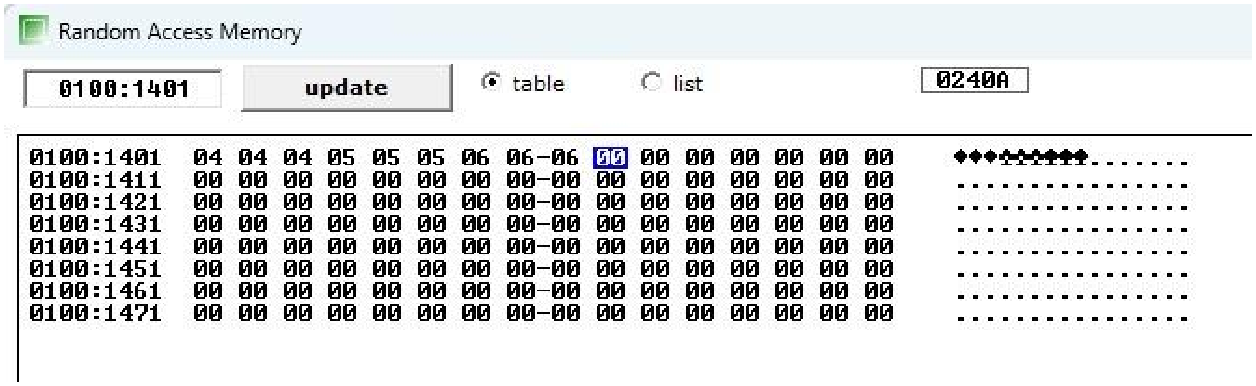


Fig-04 : 'Random Access Memory'-(a)1st matrix address, (b) 2nd matrix address

4. Then, we can see the changes in the 'Random Access Memory' that gives us the resultant matrix.

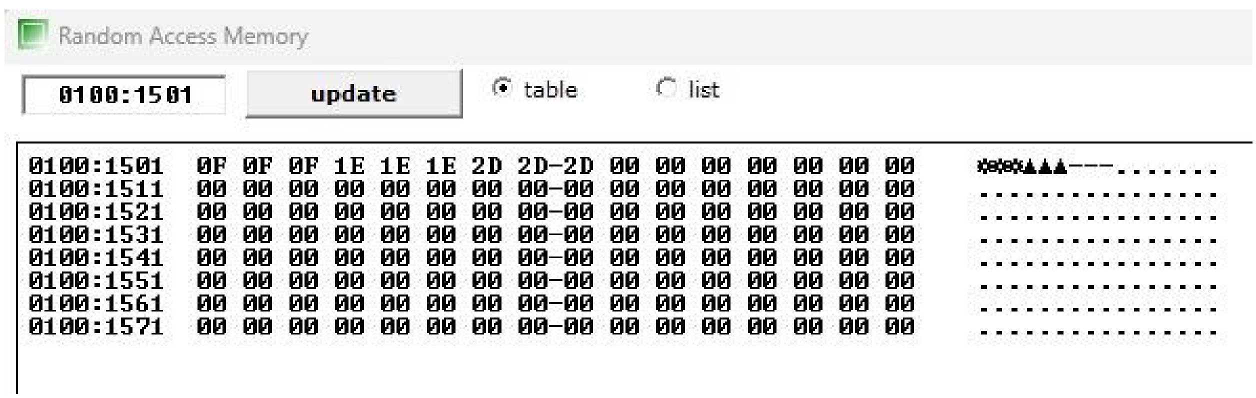


Fig-05 : 'Random Access Memory'-resultant matrix

Result Analysis :

The resultant matrix matched expected values, verifying the accuracy of the implementation. Detailed analysis of the results showed correct computation and memory usage.

Time-complexity : $O(n^2)$

Space : 27 memory addresses. As a matrix contains 3×3, so for 3 matrices it takes 27 addresses.

Discussion :

Challenges :

- ➢ Managing memory and registers efficiently within the constraints of the 8085 architecture.
- ➢ Debugging and running assembly code due to its low-level nature.
- ➢ Ensuring correct data handling and avoiding common issues like overwriting data.

Learnings :

- ➢ Enhanced understanding of microprocessor architecture and low-level programming.
- ➢ Appreciation for high-level programming languages that abstract these complexities, making programming more accessible and less error-prone.

Conclusion :

The assembly language program for multiplying two 3×3 matrices show how basic concepts in low-level programming work, like using registers, accessing memory, and controlling loops. By writing and running this program, we learned how to multiply matrices directly using the hardware. The program passes through each row and column of the input matrices, performs the necessary calculations, and stores the results in the output matrix. This project highlights the need for careful planning and accuracy when programming in assembly language.

Future Improvements :

Though the current program works there are Several ways to make it better :

1. Handle different size of matrices, not only 3×3.
2. Speed can be optimized, to make the program execute faster. Such as, reducing memory accesses and better loop techniques.
3. Manage issues like memory overflow, invalid input sizes, etc.
4. Develop a library of common matrix operations in assembly language, including addition, subtraction, and inversion. This would provide a comprehensive toolkit for handling various matrix-related tasks in assembly.
5. We can create a simple interface for users to input matrix values and see the results, using keyboard input and screen output, like shown in the Fig-06 below.

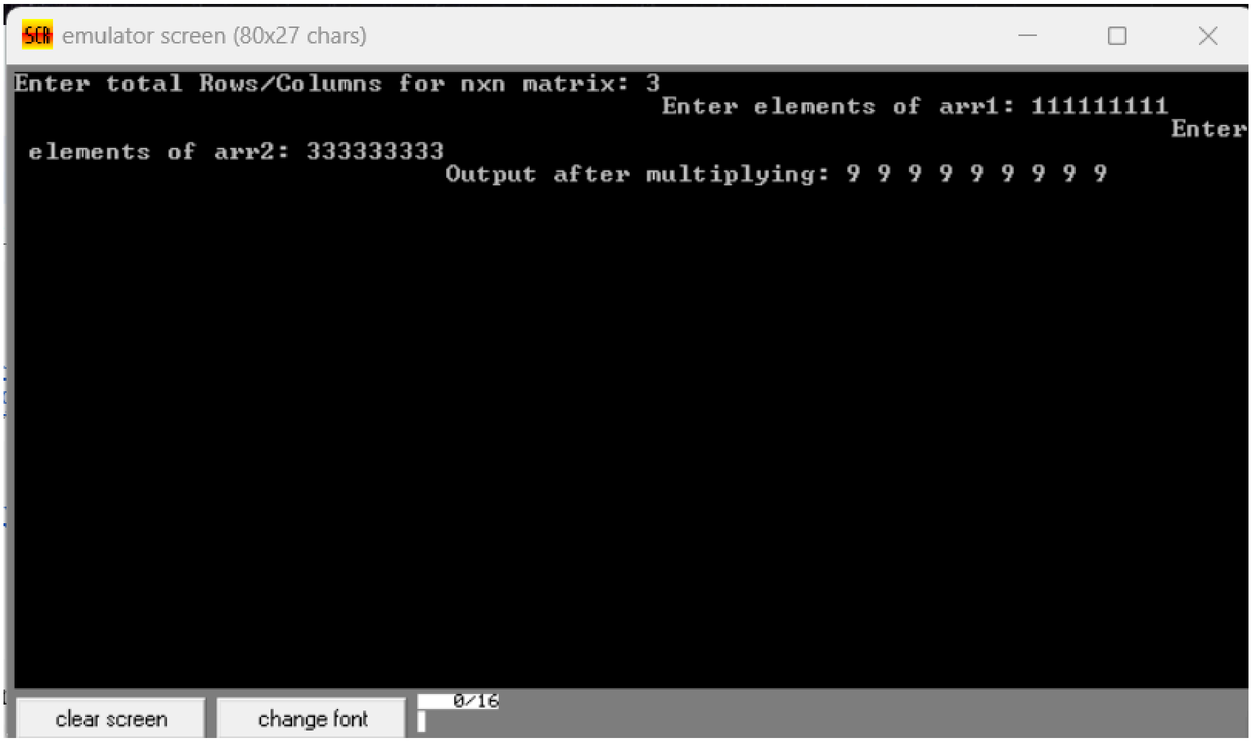


Fig-06 : Interface for users

By working on these future improvements, the project can evolve into a more versatile, efficient, and user-friendly tool, suitable for both educational purposes and practical applications.

Bibliography :

1. https://www.investopedia.com/terms/a/assembly-language.asp#:~:text=An%20assembly%20language%20is%20a%20type%20of%20programming%20language%20that,and%20their%20underlying%20hardware%20platforms.

2. https://www.tutorialspoint.com/assembly_programming/assembly_memory_segments.htm

3. https://byjus.com/maths/matrix-multiplication/