Government Polytechnic, Pune

(An Autonomous Institute of Govt. of Maharashtra)

Scheme: 180 OB

---

Government Polytechnic, Pune-16

(An Autonomous Institute of Government of Maharashtra)

A

Micro Project Report

On

# "Assembly Language Programming"

SUBMITTED BY:

| | |
|---|---|
| Vedant B. Ahire | 2106002 |
| Shravan G. Deshpande | 2106043 |
| Sujay H. Deshpande | 2106044 |

Under the Guidance of

Mrs. B. R. Amrutkar

GOVERNMENT POLYTECHNIC PUNE

(An Autonomous Institute of Government of Maharashtra)

Department Of Computer Engineering

ACADEMIC YEAR:2022-23

CERTIFICATE

This is to certify that the micro-project work entitled

“NETWORK CARD.” is a project work carried out by

| | |
|---|---|
| Vedant B. Ahire | 2106002 |
| Shravan G. Deshpande | 2106043 |
| Sujay H. Deshpande | 2106044 |

of class Second Year in partial fulfillment of the requirement for the completion of the course- Principles of Digital Technique and Microprocessor Programming (CM3105) of

Diploma in Computer Engineering

from Government Polytechnic, Pune. The report has been approved as it satisfies the academic requirements in respect of the micro-project work prescribed for the course.

……………………

Mrs. B. R. Amrutkar

Micro-Project Guide

………………………

Mrs. M. U. Kokate

Head of the Department

…………………………

Dr. V. S. Bandal

Principal

Government Polytechnic, Pune

## ACKNOWLEDGEMENT

We would like to express our deepest appreciation to all those who provided us with the possibility to complete this report. A special gratitude gives to our computer department Hod Smt. M. U. Kokate whose contribution in stimulating suggestions and encouragement, helped us to coordinate my project especially in preparing this report. Furthermore, we would also like to acknowledge with much appreciation the crucial role of the project mentor Mrs. B.R.Amrutkar who permitted us to use all required equipment and the necessary materials to complete the report on “Assembly Language Programming”. Last but not least, again many thanks go to our project mentor. Mrs. B.R.Amrutkar has invested her full effort in guiding us in achieving the goal. We have to appreciate the guidance from another supervisor and the students, especially in our project presentation, which has improved our presentation skills thanks to their comments and advice.

INDEX

# ASSEMBLY LANGUAGE

## INTRODUCTION-

In computer programming, assembly language (or assembler language),[1] sometimes abbreviated asm, is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions.[2] Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called symbolic machine code.[3][4]

Assembly code is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as assembly, as in assembling the source code. Assembly language usually has one statement per machine instruction (1:1), but comments and statements that are assembler directives,[5]macros,[6][1] and symbolic labels of program and memory locations are often also supported.

The term "assembler" is generally attributed to Wilkes, Wheeler and Gill in their 1951 book The Preparation of Programs for an Electronic Digital Computer,[7] who, however, used the term to mean "a program that assembles another program consisting of several sections into a single program".[8]

Each assembly language is specific to a particular computer architecture and sometimes to an operating system.[9] However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling, a much more complicated task than assembling.

The computational step when an assembler is processing a program is called assembly time.

Assembly language uses a mnemonic to represent each low-level machine instruction or opcode, typically also each architectural register, flag, etc. Many operations require one or more operands in order to form a complete instruction. Most assemblers permit named constants, registers, and labels for program and memory locations, and can calculate expressions for operands. Thus, programmers are freed from tedious repetitive calculations and assembler programs are much more readable than machine code. Depending on the architecture, these elements may also be combined for specific instructions or addressing modes using offsets or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.

## TERMINOLOGY-

- A macro assembler includes a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.
- A cross assembler (see also cross compiler) is an assembler that is run on a computer or operating system (the host system) of a different type from the system on which the resulting code is to run (the target system). Cross-assembling facilitates the development of programs for systems that do not have the resources to support software development, such as an embedded system or a microcontroller. In such a case, the resulting object code must be transferred to the target system, via read-only memory (ROM, EPROM, etc.), a programmer (when the read-only memory is integrated in the device, as in microcontrollers), or a data link using either an exact bit-by-bit copy of the object code or a text-based representation of that code (such as Intel hex or Motorola S-record).

- A high-level assembler is a program that provides language abstractions more often associated with high-level languages, such as advanced control structures (IF/THEN/ELSE, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.
- A microassembler is a program that helps prepare a microprogram, called firmware, to control the low level operation of a computer.
- A meta-assembler is "a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language".[10] "Meta-Symbol" assemblers for the SDS 9 Series and SDS Sigma series of computers are meta-assemblers.[11][nb 1]Sperry Univac also provided a Meta-Assembler for the UNIVAC 1100/2200 series.[12]
- inline assembler (or embedded assembler) is assembler code contained within a high-level language program.[13] This is most often used in systems programs which need direct access to the hardware.

## Assembler

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.[14] The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of called subroutines.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Called jump-sizing,[14] most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.[citation needed]

Assemblers have been available since the 1950s, as the first step above machine language and before high-level programming languages such as Fortran, Algol, COBOL and Lisp. There have also been several classes of translators and semi-automatic code generators with properties similar to both assembly and high-level languages, with Speedcode as perhaps one of the better-known examples.

There may be several assemblers with different syntax for a particular CPU or instruction set architecture. For instance, an instruction to add memory data to a register in a x86-family processor might be add eax,[ebx], in original Intel syntax, whereas this would be written addl (%ebx),%eax in the AT&T syntax used by the GNU Assembler. Despite different appearances, different syntactic forms generally generate the same numeric machine code. A single assembler may also have different modes in order to support variations in syntactic forms as well as their exact semantic interpretations (such as FASM-syntax, TASM-syntax, ideal mode, etc., in the special case of x86 assembly programming).

Number of passes

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.

Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary, pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was memory size and speed of assembly – often a second pass would require storing the symbol table in memory (to handle forward references), rewinding and rereading the program source on tape, or rereading a deck of cards or punched paper tape. Later computers with much larger memories (especially disc storage), had the space to perform all necessary processing without such re-reading. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.[15]

Example: in the following code snippet, a one-pass assembler would be able to determine the address of the backward reference BKWD when assembling statement S2, but would not be able to determine the address of the forward reference FWD when assembling the branch statement S1; indeed, FWD may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known when generating code in pass 2.

## Assembler

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.[14] The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of called subroutines.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Called jump-sizing,[14] most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.[citation needed]

Assemblers have been available since the 1950s, as the first step above machine language and before high-level programming languages such as Fortran, Algol, COBOL and Lisp. There have also been several classes of translators and semi-automatic code generators with properties similar to both assembly and high-level languages, with Speedcode as perhaps one of the better-known examples.

There may be several assemblers with different syntax for a particular CPU or instruction set architecture. For instance, an instruction to add memory data to a register in a x86-family processor might be add eax,[ebx], in original Intel syntax, whereas this would be written addl (%ebx),%eax in the AT&T syntax used by the GNU Assembler. Despite different appearances, different syntactic forms generally generate the same numeric machine code. A single assembler may also have different modes in order to support variations in syntactic forms as well as their exact semantic interpretations (such as FASM-syntax, TASM-syntax, ideal mode, etc., in the special case of x86 assembly programming).

Number of passes

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

One-pass assemblers go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.

Multi-pass assemblers create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary, pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was memory size and speed of assembly – often a second pass would require storing the symbol table in memory (to handle forward references), rewinding and rereading the program source on tape, or rereading a deck of cards or punched paper tape. Later computers with much larger memories (especially disc storage), had the space to perform all necessary processing without such re-reading. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.[15]

Example: in the following code snippet, a one-pass assembler would be able to determine the address of the backward reference BKWD when assembling statement S2, but would not be able to determine the address of the forward reference FWD when assembling the branch statement S1; indeed, FWD may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known when generating code in pass 2.

## Instructions for Data Processing

The following instructions manipulate data. This can be arithmetic operations that perform math functions, comparison operations, or data movement.

Addition (ADD)

Addition (ADD) adds R2 to R1 and puts the result in R0. Addition with Carry (ADC) adds R2 to R1, along with the carry flag. This is used when dealing with numbers larger than a single 32 bit word.

```
ADD R0, R1, R2

ADC R0, R1, R2
```

Subtraction (SUB)

Subtraction (SUB) subtracts R2 from R1 and puts the result in R0. Subtraction with Carry (SBC) subtracts R2 from R1 and, if the carry flag is cleared, subtracts one from the result. This is equivalent to borrowing in arithmetic and ensures that multi-word subtraction works correctly.

```
SUB R0, R1, R2

SBC R0, R1, R2
```

Compare (CMP) and Compare Negative (CMN)

Compare (CMP) and Compare Negative (CMN) compare two operands. CMP subtracts R1 from R0 and CMN adds R2 to R1, and then the status flags are updated according to the result of the addition or subtraction.

```
CMP R0, R1

CMN R1, R2
```

Move (MOV)

The Move (MOV) operation does exactly what it sounds like. It moves data from one place to another. Below, R1 is copied into R0. The second line puts the immediate value 8 into R0.

```
MOV R0, R1

MOV R0, #8
```

Move Negative (MVN)

Move negative (MVN) performs a similar operation, but complements (inverts) the data first. This is useful when performing operations with negative numbers, in particular with two's complement notation. The instruction below puts NOT 8, better known as –9, into R0. Add one to that result and you have performed the two's complement and obtained -8.

```
MVN R0, #8
```

AND performs the bitwise AND of R2 and R1 and puts the result in R0. An immediate value can be used instead of R2.

```
AND R0, R1, R2
```

ORR and EOR

ORR and EOR perform the bitwise OR and XOR, respectively, of R2 and R1.

```
ORR R0, R1, R2

EOR R0, R1, R2
```

Bit Clear (BIC)

Bit Clear (BIC) performs a bitwise AND of R2 and R1, but first complements the bits in R2. This operation is often used with immediate values, as in the second line, where the immediate value, 0xFF, is inverted and subsequently ANDed with R1. ANDing eight zeros with the first byte of R1 will clear those bits, i.e., set them equal to zero, and the result will be put in R0.

```
BIC R0, R1, R2

BIC R0, R1, #0xFF
```

Test Bits (TST) and Test Equivalence (TEQ)

TeST Bits (TST) and Test Equivalence (TEQ) exist to test the bits located in registers. These instructions do not use a destination register, but simply update the status register based on the result. TST essentially performs a bitwise AND of the two operands. By using a mask for operand two, we can test if an individual bit in R0 is set.

In this case, we check bit 3 (bitmask = 1000b = 8) and set the Z flag based on the outcome. TEQ performs a similar function to exclusive or and is great for checking whether two registers are equal. This updates the N and Z flag, therefore it also works on signed numbers; N is set to one if their signs are different.

```
TST R0, #8

TEQ R1, R2
```

Multiplication (MUL)

Multiplication (MUL) multiplies R1 by R2 and puts the result in R0. Multiplication cannot be used with an immediate value.

```
MUL R0, R1, R2
```

Instructions for Shifting and Rotating

Logical Shift Left (LSL)

Logical Shift Left (LSL) shifts the bits in R1 by a shift value. In this case, the immediate value 3, and drops the most significant bits. The last bit that was shifted out is put into the carry flag, and the least significant bits are filled with zeros. Below, R1 gets shifted left by the immediate value 3, or a value between 0 and 31 in R2, and put in R0. One logical left shift multiplies a value by two. This is an inexpensive way to do simple multiplication.

```
LSL R0, R1, #3

LSL R0, R1, R2
```

Logical Shift Right (LSR)

Logical Shift Right (LSR) works in the reverse fashion as LSL and effectively divides a value by two. The most significant bits are filled with zeros, and the last least significant bit is put into the carry flag.

```
LSR R0, R1, #2
```

Arithmetic Shift Right (ASR)

Arithmetic Shift Right (ASR) performs the same work as LSR but is designed for signed numbers. It copies the sign bit back into the last position on the left.

```
ASR R0, R1, #4
```

Rotate Right (ROR)

Rotate Right (ROR) rotates all the bits in a word by some value. Instead of filling the bits on the left with zeros, the bits shifted out are simply put back into the other end.

```
ROR R0, R1, #5
```

Instructions for Branching Operations

One important function of a processor is the ability to choose between two code paths based on a set of inputs. This is exactly what branching operations do. A processor normally executes one instruction after the other by incrementing R15, the program counter (PC), by four bytes (i.e., the length of a single instruction). Branching changes the PC to another location denoted by a label that represents that part of the assembly code.

Branch (B)

Branch (B) moves the PC to an address specified by a label. The label ("loop" in the example below) represents a section of code that you want the processor to execute next. Labels are just text, usually a meaningful word.

```
B    loop
```

Branch Link (BL)

Branch Link (BL) performs a similar operation, but it copies the address of the next instruction into R14, the link register (LR). This works great when performing subroutine/procedure calls, because as soon as the section of code at the label is finished we can use the LR to get back to where we branched. Below, we branch to the label "subroutine" and then use the link register to get back to the next instruction.

```
BL    subroutine
```

```
…

subroutine:

…

MOV    PC, LR
```

We use a MOV instruction to put the link register back into the program counter. This returns the program to the spot right after our subroutine call, here labeled . Notice the use of LR and PC above. ARM assemblers recognize these as R14 and R15, respectively. This provides a convenient reminder to the programmer about the operation being performed.

Instructions for Load and Store

A computer's memory stores data that is needed by the processor. This data is accessed by using an address. By first putting an address into a register, we can then access the data at that address. This is why we use load and store operations.

Load Register (LDR)

Load register (LDR) loads the data located at an address into the destination register. The brackets around R1 signify that the register contains an address. By using the brackets we put the data at that address into R0, instead of the address itself. We can also use this notation to locate data offset from a certain address, as shown on the second line. R0 would contain the data two words away from whatever address R1 contains.

```
LDR R0, [R1]

LDR R0, [R1, #8]
```

We can also use labels to represent an address, and the corresponding data can then be loaded into a register. The first line below loads the address of the label “info” into R0. The value stored at that address is then accessed and put into R1 in the second line.

```
LDR R0, =info

LDR R1, [R0]
```

Store (STR)

Store (STR) performs the complementary operation to load. STR puts the contents of a register into a memory location. The code below stores the data in R1 at the address in R0. Again, the brackets signify that R0 holds an address, and we want to modify the data at that address.

```
STR R1, [R0]
```

Load and Store Types: Byte (B), Halfword (H), Word (Omitted), Signed (SB), Unsigned (B)

Both load and store can be written with a type appended to them. This type signifies whether the instruction will manipulate a byte (B), halfword (H), or word (omitted) and whether the data is signed (SB) or unsigned (B).

One place this may come in handy is for string manipulation, as ASCII characters have a length of one byte. These operations also allow for the use of offsets when loading or storing, as seen in the last line.

```
   LDR R0, =text          @ load a 32 bit address into R0

   STRB R1, [R0]          @ store byte at address in memory

   STRB R1, [R0, + R2]      @ store byte at address + offset R2
```

Instructions for Conditionals

As mentioned earlier, the mnemonics used in an instruction can have optional condition codes appended to them. This allows for conditional execution.

Remember, the flags (as laid out in the previous article) are Z (zero), C (carry), N (negative), and V (overflow).

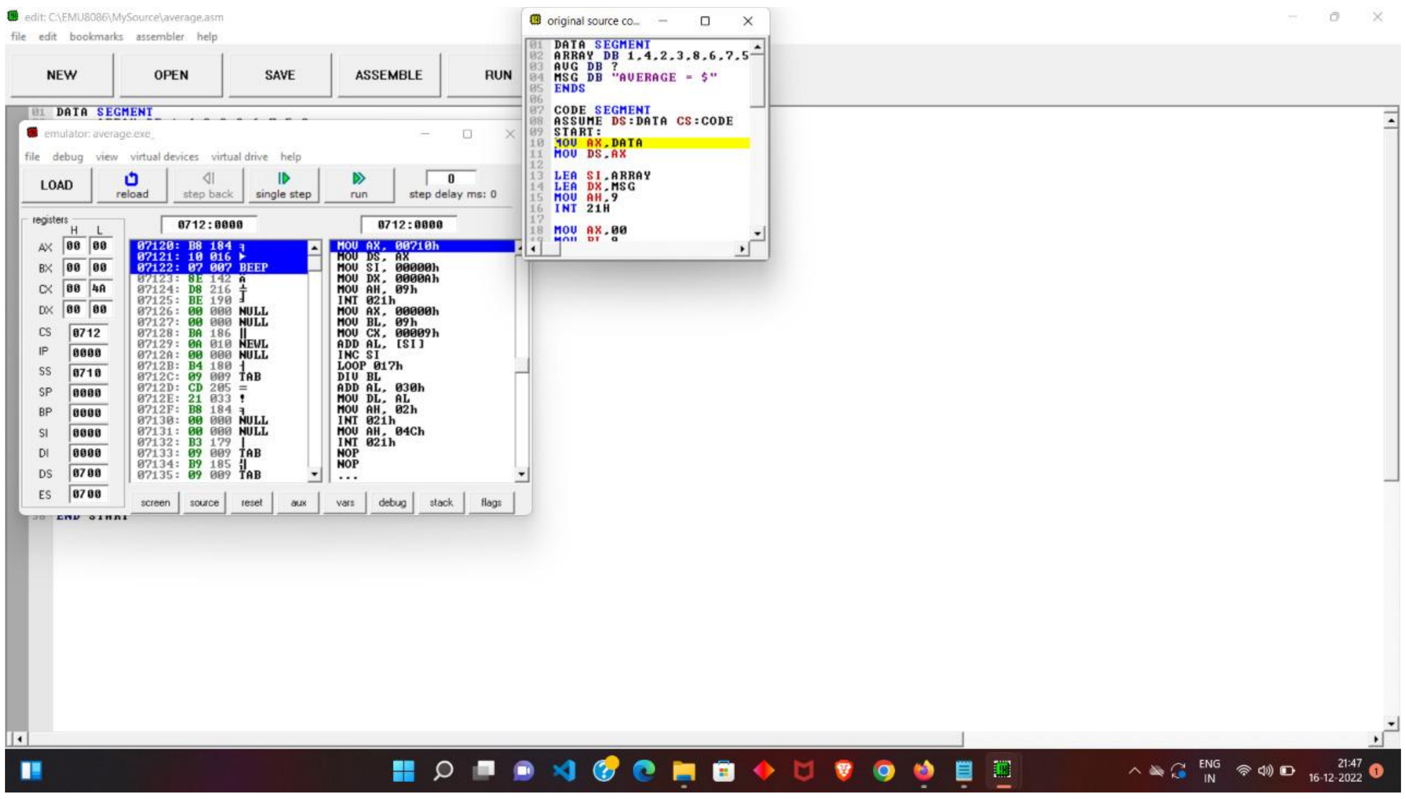
To force instructions to update the status register, an optional S can be appended to most mnemonics mentioned thus far. Once the status register is updated, a number of conditional suffixes, shown below, can be used to control whether the instruction executes. The binary codes for these suffixes correspond to the first four bits of the data-processing instruction shown above .

These suffixes are appended to the mnemonic when writing assembly. The listing below shows a few of the conditional suffixes used with instructions mentioned earlier.

## ASSEMBLY LANGUAGE PROJECT CODE

```
DATA SEGMENT

    ARRAY DB 1,4,2,3,8,6,7,5,9

    AVG DB ?

    MSG DB "AVERAGE = $"

ENDS



CODE SEGMENT

   ASSUME DS:DATA CS:CODE

START:

     MOV AX,DATA

     MOV DS,AX



     LEA SI,ARRAY

     LEA DX,MSG

     MOV AH,9

     INT 21H
     MOV AX,00
     MOV BL,9


     MOV CX,9
     LOOP1:
        ADD AL,ARRAY[SI]
        INC SI
     LOOP LOOP1


     DIV BL


     ADD AL,30H


     MOV DL,AL
     MOV AH,2
     INT 21H


     MOV AH,4CH
     INT 21H
ENDS
END START
```

## OUTPUT

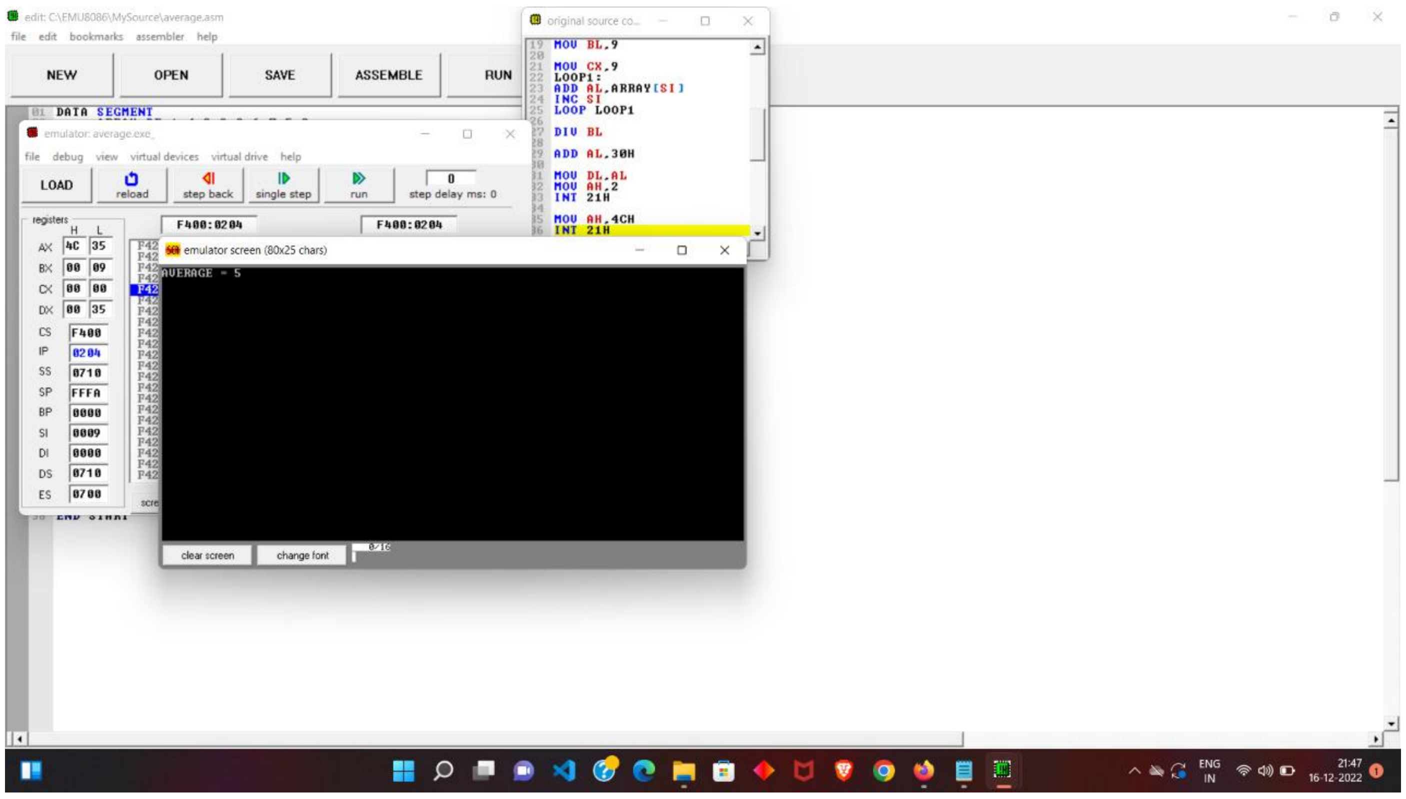

edit: C:\EMU8086\MySource\average.asm

original source code:

```
DATA SEGMENT
ARRAY DB 1,4,2,3,8,6,7,5
AVG DB ?
MSG DB "AVERAGE = $"
ENDS

CODE SEGMENT
ASSUME DS:DATA CS:CODE
START:
MOV AX,DATA
MOV DS,AX

LEA SI,ARRAY
LEA DX,MSG
MOV AH,9
INT 21H

MOV AX,00
```

emulator: average.exe_

```
registers     0712:0000                      0712:0000
   H  L
AX 00 00      07120: B8 184 ╕                MOV AX, 00710h
BX 00 00      07121: 10 016 ►                MOV DS, AX
CX 00 4A      07122: 07 007 BEEP             MOV SI, 00000h
DX 00 00      07123: 8E 142 Ä                MOV DX, 0000Ah
CS 0712       07124: D8 216 ╪                MOV AH, 09h
IP 0000       07125: BE 190 ╛                INT 021h
SS 0710       07126: 00 000 NULL             MOV AX, 00000h
SP 0000       07127: 00 000 NULL             MOV BL, 09h
BP 0000       07128: BA 186 ║                MOV CX, 00009h
SI 0000       07129: 0A 010 NEWL             ADD AL, [SI]
DI 0000       0712A: 00 000 NULL             INC SI
DS 0700       0712B: B4 180 ┤                LOOP 017h
ES 0700       0712C: 09 009 TAB              DIV BL
              0712D: CD 205 =                ADD AL, 030h
              0712E: 21 033 !                MOV DL, AL
              0712F: B8 184 ╕                MOV AH, 02h
              07130: 00 000 NULL             INT 021h
              07131: 00 000 NULL             MOV AH, 04Ch
              07132: B3 179 │                INT 021h
              07133: 09 009 TAB              NOP
              07134: B9 185 ╣                NOP
              07135: 09 009 TAB              ...
```


edit: C:\EMU8086\MySource\average.asm

original source code:

```
MOV BL,9

MOV CX,9
LOOP1:
ADD AL,ARRAY[SI]
INC SI
LOOP LOOP1

DIV BL

ADD AL,30H

MOV DL,AL
MOV AH,2
INT 21H

MOV AH,4CH
INT 21H
```

emulator: average.exe_

```
registers     F400:0204                      F400:0204
   H  L
AX 4C 35
BX 00 09
CX 00 00
DX 00 35
CS F400
IP 0204
SS 0710
SP FFFA
BP 0000
SI 0009
DI 0000
DS 0710
ES 0700
```

emulator screen (80x25 chars)

```
AVERAGE = 5
```

## CONCLUSION-

We have successfully applied the concepts learnt in Assembly language in this Microproject and executed the mentioned code using Emulator 8086.