mouLAB 1: Learning how to use DOSBox and Debugx and design an Assembly Language Program

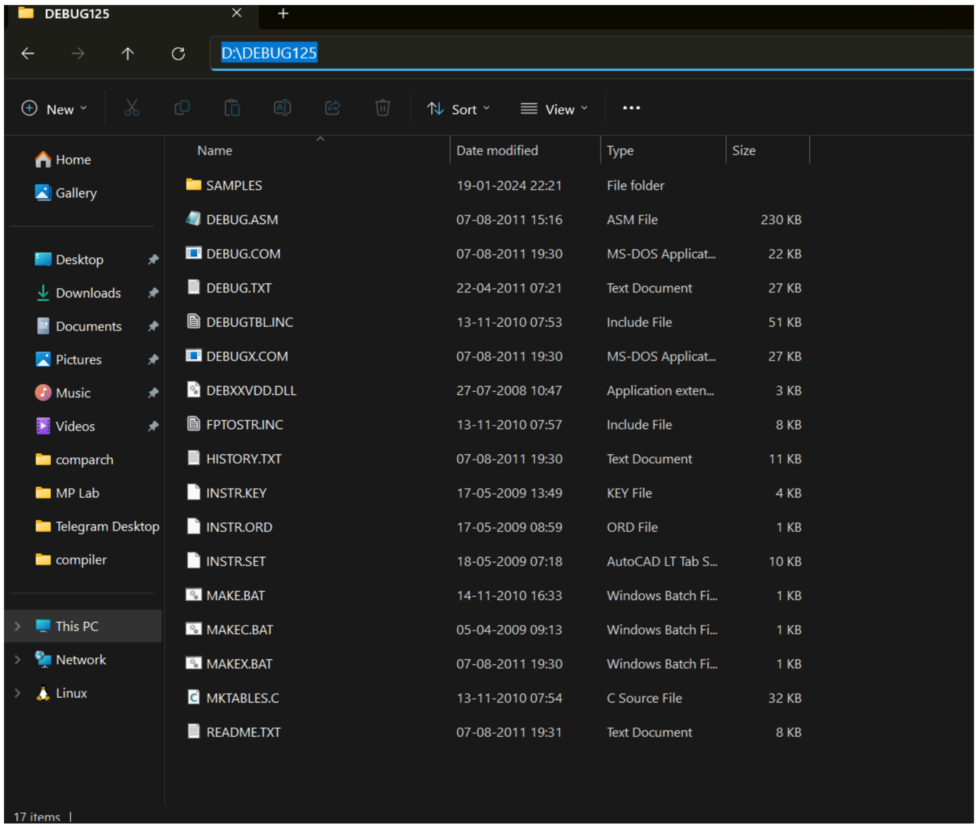
I. Introduction to Debugx and How to use it?

DEBUG, supplied by MS-DOS, is a program that traces the 8086 instructions. Using DEBUG, you can easily enter an 8086 machine code program into memory and save it as an executable MS-DOS file (in .COM/.EXE format). DEBUG can also be used to test and debug 8086 and 8088 programs. The features include examining and changing memory and register contents including the CPU register. The programs may also be single-stepped or run at full speed to a breakpoint.

You will be using DEBUGX which is a program similar to DEBUG but offers full support for 32-bit instructions and addressing. DEBUGX includes the 80x86 instructions through the Pentium instructions.

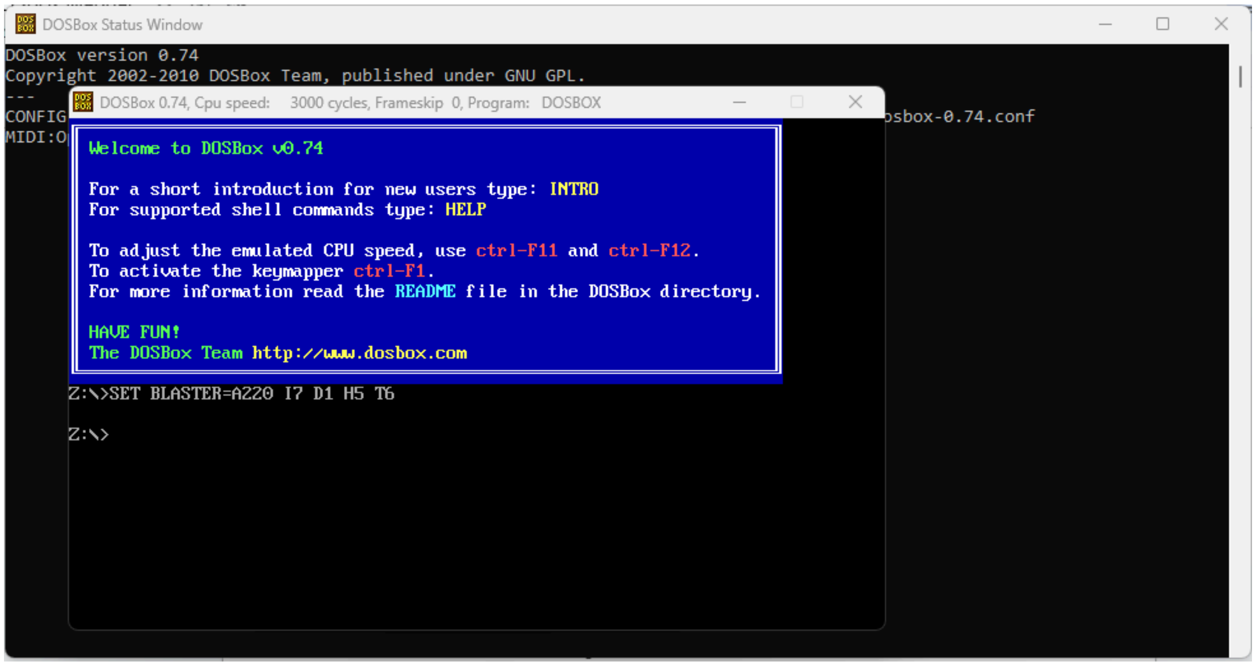
Path of Debug Folder

Find out the path of debug folder: **Note that it is stored in C:\ in CC Lab Machines**



Name	Date modified	Type	Size
SAMPLES	19-01-2024 22:21	File folder	
DEBUG.ASM	07-08-2011 15:16	ASM File	230 KB
DEBUG.COM	07-08-2011 19:30	MS-DOS Applicat...	22 KB
DEBUG.TXT	22-04-2011 07:21	Text Document	27 KB
DEBUGTBL.INC	13-11-2010 07:53	Include File	51 KB
DEBUGX.COM	07-08-2011 19:30	MS-DOS Applicat...	27 KB
DEBXXVDD.DLL	27-07-2008 10:47	Application exten...	3 KB
FPTOSTR.INC	13-11-2010 07:57	Include File	8 KB
HISTORY.TXT	07-08-2011 19:30	Text Document	11 KB
INSTR.KEY	17-05-2009 13:49	KEY File	4 KB
INSTR.ORD	17-05-2009 08:59	ORD File	1 KB
INSTR.SET	18-05-2009 07:18	AutoCAD LT Tab S...	10 KB
MAKE.BAT	14-11-2010 16:33	Windows Batch Fi...	1 KB
MAKEC.BAT	05-04-2009 09:13	Windows Batch Fi...	1 KB
MAKEX.BAT	07-08-2011 19:30	Windows Batch Fi...	1 KB
MKTABLES.C	13-11-2010 07:54	C Source File	32 KB
README.TXT	07-08-2011 19:31	Text Document	8 KB

Open DOSBOX


DOSBox Status Window
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
DOSBox version 0.74
Copyright 2002-2010 DOSBox Team, published under GNU GPL.
---
CONFIG                                                        osbox-0.74.conf
MIDI:O
```

```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>
```

Full Screen Mode

Press alt-enter. Alternatively: Edit the configuration file of DOSBox and change the option fullscreen=false to fullscreen=true. If fullscreen looks wrong in your opinion: Play with the options: fullresolution, output and aspect in the configuration file of DOSBox. To get back from fullscreen mode: Press alt-enter again.

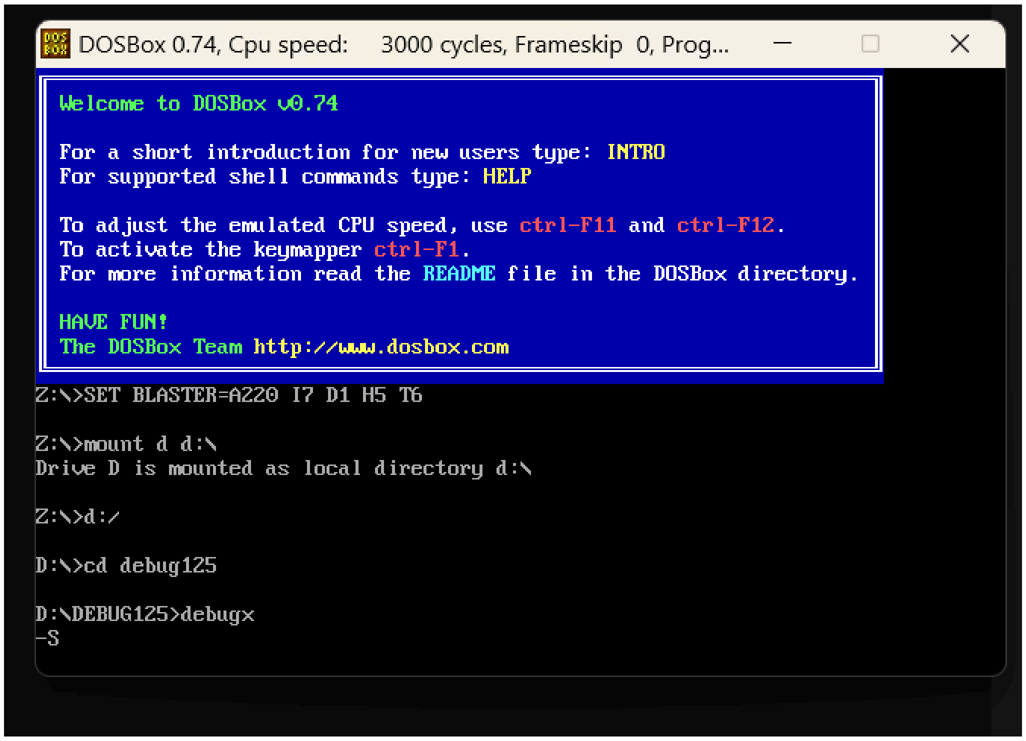
Mounting

In Dosbox, at the beginning you've got a Z:\> instead of a C:\> at the prompt. You have to make your directories available as drives in DOSBox by using the "mount" command

Windows . in Windows "mount C D:\" will give you a C drive in DOSBox which points to your Windows D:\ directory.

Linux "mount c /home/username" will give you a C drive in DOSBox which points to /home/username in Linux.Now with this you can get to any directory you want

To change to the drive mounted like above, type "C:". If everything went fine, DOSBox will display the prompt "C:\>"
cd into the debug folder and type ***debugx*** to get started Debug prompt **"-"** should appear.



DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Prog...

```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount d d:\
Drive D is mounted as local directory d:\

Z:\>d:/

D:\>cd debug125

D:\DEBUG125>debugx
-S
```

II. Getting started with debug commands

Type in “?” for a listing of the DEBUG commands you can enter.

NOTE: debugx does not distinguish between lowercase and uppercase letters.
It assumes that all numbers are in Hexadecimal.
Segment and offset are specified as segment: offset.
Spaces in commands are used only to separate parameters

Commands common to DEBUG & DEBUGX

A - Assembles symbolic instructions into machine code

D - Display the contents of an area of memory in hex format

E - Enter data into memory beginning at the specific location

G - Run the executable program in the memory (Go)

P - Proceed, Execute a set of related instructions

Q - Quit the debug session

R - Display the contents of one or more registers in hex format

T - Trace the execution of one instruction

U - Unassemble machine code into symbolic code

? - Debug Help

R, the Register command: examining and altering the content of registers

In DEBUGX, the register command “R” allows you to examine and/or alter the contents of the internal CPU registers.

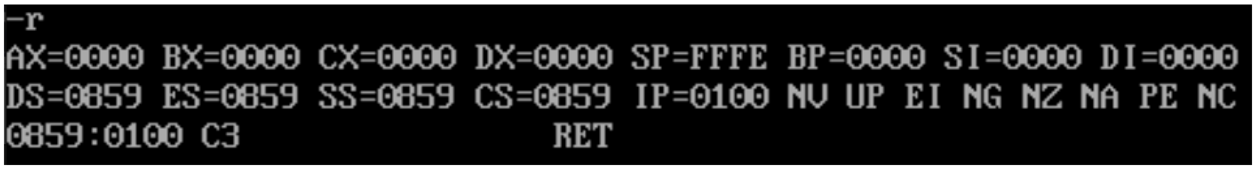
R <register name>

The “R” command will display the contents of all registers unless the optional <register name> field is entered. In which case, only the content of the selected register will be displayed. The R command also displays the instruction pointed to by the IP. The “R” command without the register name field, responds with three lines of information. The first two lines show you the programmer’s model of the 80x86 microprocessor. (The first line displays the contents of the general-purpose, pointer, and index registers. The second line displays the current values of the segment registers, the instruction pointer, and the flag register bits. The third line shows the location, machine code, and Assembly code of the next instruction to be executed.)

If the optional <register name> field is specified in the “R” command, DEBUGX will display the contents of the selected register and give you an opportunity to change its value. Just type a <return> if no change is needed, e.g.

```
-r ECX

ECX 00000000

: FFFF

-r ECX

ECX 0000FFFF

:
```

Note that DEBUGX pads input numbers on the left with zeros if fewer than four digits are typed in for 16-bit register mode and if fewer than eight digits in 32-bit addressing mode



```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0100 NV UP EI NG NZ NA PE NC
0859:0100 C3            RET
```

From the above screenshot, you can find out the details of DS, ES, SS, CS, and IP

Symbol/Register	Meaning
AX	Accumulator register
BX	Base register
CX	Count register
DX	Data register
SI	Source index register
DI	Destination index register
SP	Stack pointer register
BP	Base pointer register
CS	Code segment register
DS	Data segment register
SS	Stack segment register
ES	Extra segment register
F	Flag register

Flag	Meaning	Set	Reset
OF	Overflow	OV	NV
DF	Direction	DN	UP
IF	Interrupt	EI	DI
SF	Sign	NG	PL
ZF	Zero	ZR	NZ
AF	Auxiliary Carry	AC	NA
PF	Parity	PE	PO
CF	Carry	CY	NC

Flag Registers

```
This is an 8-bit register that holds 1-bit of data for each
of the eight "Flags" which are to be interpreted as follows:

Textbook abbrev. for Flag Name =>   of df if sf zf af pf cf
If the FLAGS were all SET (1),      -- -- -- -- -- -- -- --
they would look like this...   =>   OV DN EI NG ZR AC PE CY
If the FLAGS were all CLEARed (0),
they would look like this...   =>   NV UP DI PL NZ NA PO NC


          FLAGS           SET (a 1-bit)  CLEARed (a 0-bit)
 ---------------          --------------- -------------------
       Overflow   of  =   OV (OVerflow)  NV [No oVerflow]

      Direction   df  =   DN (decrement) UP (increment)

      Interrupt   if  =   EI  (enabled)  DI (disabled)

           Sign   sf  =   NG (negative)  PL (positive)

           Zero   zf  =   ZR   [zero]    NZ  [ Not zero]

Auxiliary Carry   af  =   AC             NA  [ No AC ]

         Parity   pf  =   PE  (even)     PO  (odd)

          Carry   cf  =   CY  [Carry]    NC  [ No Carry]


==============
The individual abbreviations appear to have these meanings:

OV = OVerflow, NV = No oVerflow.    DN = DowN,  UP (up).
EI = Enable Interupt, DI = Disable Interupt.

NG = NeGative, PL = PLus; a strange mixing of terms due to the
     fact that 'Odd Parity' is represented by PO (so it can't
     be as POsitive here), but they still could have used 'MI'
     and for the word, MInus; instead of 'NeGative'.

ZR = ZeRo,  NZ = Not Zero.
AC = Auxiliary Carry, NA = Not Auxiliary carry.
PE = Parity Even, PO = Parity Odd.  CY = CarrY, NC = No Carry.
```

A, the Assemble command

The assemble command is used to enter Assembly language instructions into memory.

Syntax: A <starting address>

The starting address may be given as an offset number, in which case it is assumed to be an offset into the code segment. Otherwise, CS can also be specified explicitly. (eg. “A 100” and “A CS:100” will achieve the same result). When this command is entered, DEBUG/DEBUGX will begin prompting you to enter Assembly language instructions. After an instruction is typed in and followed by <return>, DEBUG/DEBUGX will prompt for the next instruction. This process is repeated until you type a <return> at the address prompt, at which time DEBUG/DEBUGX will return you to the debug command prompt.

Use the “A” command to enter the following instructions starting from offset 0100H.

EXAMPLE :

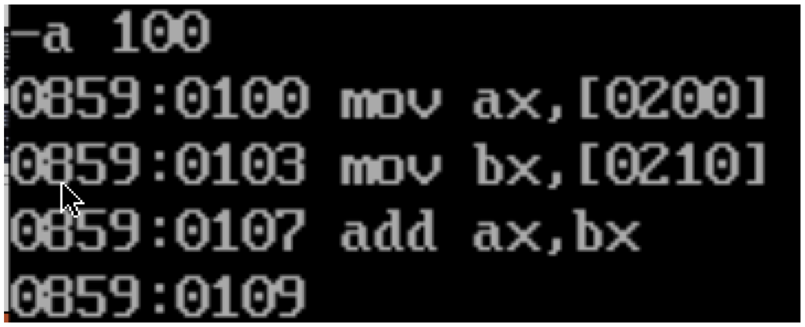
A. **Example of 16 bit format**

```
-a 100
xxxx:0100 mov ax,1
xxxx:0103 mov bx,2
xxxx:0106 mov cx,3
xxxx:0109 add ax, cx
xxxx:010B add ax, bx
xxxx:010D jmp 100
xxxx:010F <enter>
```

B. **Example of 32 bit format**

```
-a 100
xxxx:0100 mov eax,1
xxxx:0106 mov ebx,2
xxxx:010C mov ecx,3
xxxx:0112 add eax, ecx
xxxx:0115 add eax, ebx
xxxx:0118 jmp 100
xxxx:011A <enter>
```

Where xxxx specifies the base value of instruction address in the code segment. Please note that the second instruction starts at xxxx:0106 in 32 bit format and xxxx:0103 in 16-bit format. This implies that the first instruction is six bytes long in 32-bit format and three byte long in 16-bit format.
Similarly note the size of all instructions.
When DEBUGX is first invoked, what are the values in the general-purpose registers?
What is the reason for setting [IP] = 0100 ?

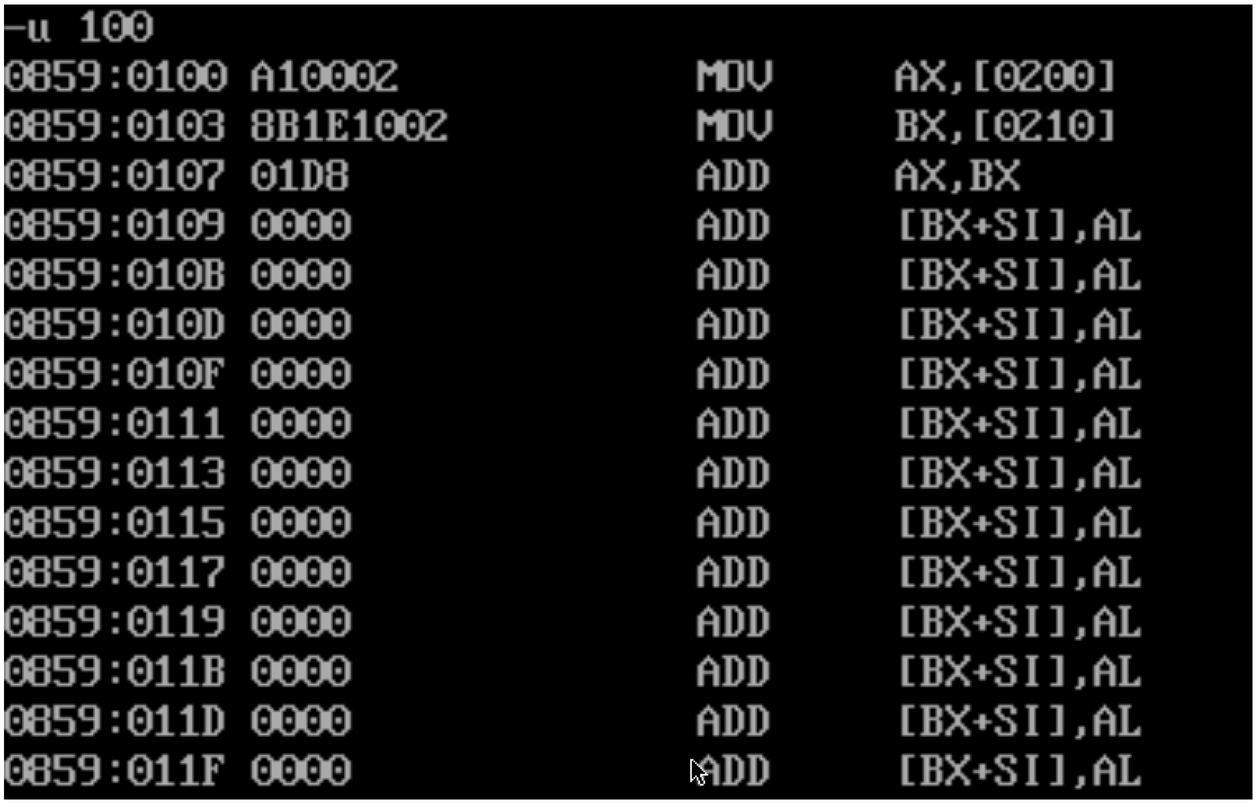


```
-a 100
0859:0100 mov ax,[0200]
0859:0103 mov bx,[0210]
0859:0107 add ax,bx
0859:0109
```

U, the Unassemble command: looking at machine code

Syntax: U <starting address>

The unassemble command displays the machine code in memory along with their equivalent Assembly language instructions.



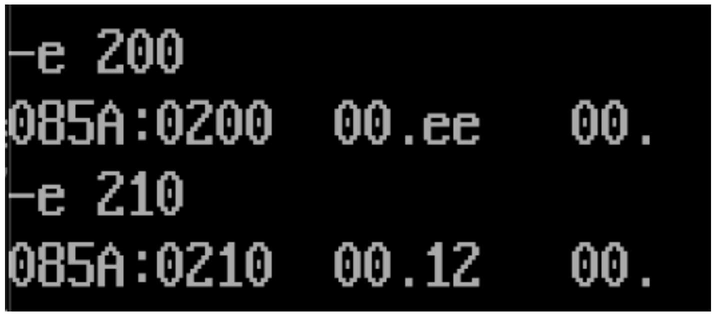
```
-u 100
0859:0100 A10002                  MOV     AX,[0200]
0859:0103 8B1E1002                MOV     BX,[0210]
0859:0107 01D8                    ADD     AX,BX
0859:0109 0000                    ADD     [BX+SI],AL
0859:010B 0000                    ADD     [BX+SI],AL
0859:010D 0000                    ADD     [BX+SI],AL
0859:010F 0000                    ADD     [BX+SI],AL
0859:0111 0000                    ADD     [BX+SI],AL
0859:0113 0000                    ADD     [BX+SI],AL
0859:0115 0000                    ADD     [BX+SI],AL
0859:0117 0000                    ADD     [BX+SI],AL
0859:0119 0000                    ADD     [BX+SI],AL
0859:011B 0000                    ADD     [BX+SI],AL
0859:011D 0000                    ADD     [BX+SI],AL
0859:011F 0000                    ADD     [BX+SI],AL
```

E, the Enter Command

Syntax: E address

You can modify the content of any memory location in the data segment using the “E” command

Example:
e 200
Write the value you want to write at memory location 200



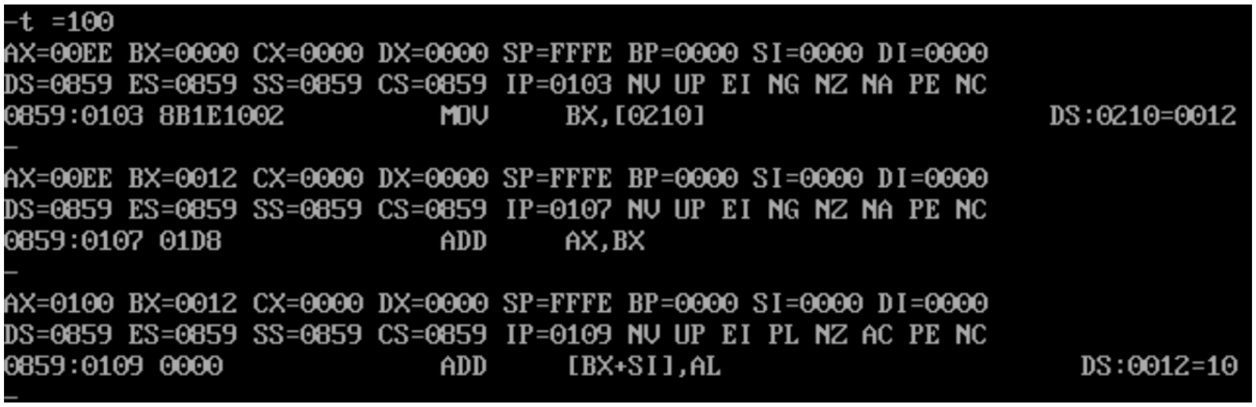
```
-e 200
085A:0200  00.ee   00.
-e 210
085A:0210  00.12   00.
```

T, the Trace command: single-step execution

The trace command allows you to trace through the execution of your programs one or more instructions at a time to verify the effect of the programs on registers and/or data.

Syntax : T =<starting address>
T =100
If you do not specify the starting address then you have to set the IP using the R command before using the T command

How to do this?
Using r command to set IP to 100 and then execute T 5 Using r command to set IP to 100 and then execute T
Trace through the above sequence of instructions that you have entered and examine the registers and Flag registers



```
-t =100
AX=00EE BX=0000 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0103 NV UP EI NG NZ NA PE NC
0859:0103 8B1E1002          MOV     BX,[0210]                  DS:0210=0012
-
AX=00EE BX=0012 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0107 NV UP EI NG NZ NA PE NC
0859:0107 01D8              ADD     AX,BX
-
AX=0100 BX=0012 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0109 NV UP EI PL NZ AC PE NC
0859:0109 0000              ADD     [BX+SI],AL                   DS:0012=10
-
```

G, the Go command

The go command instructs DEBUG to execute the instructions found between the starting and stop addresses.

Syntax: G < = starting address > < stop address >

Execute the following command and explain the result (this is with respect to code you have written in 32-bit format earlier)
G = 100 118

Caution: the command G= 100 119 will cause the system to hang. Explain why?
Often, it is convenient to pause the program after executing a few instructions, thus effectively creating a breakpoint in the program. For example, the command “g 10c” tells the DEBUG to start executing the next instruction(s) and pause at offset 010CH.
The main difference between the GO and TRACE commands is that the GO command lists the register values after the execution of the last instruction while the TRACE command does so after each instruction execution
Try executing the sequence of instructions you entered using several options of the “G” command. Remember to reload 0100 into IP every time you use G without the starting address.

III. Some assembly instructions you will need

MOV instruction

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The source and destination in an instruction cannot both be memory locations. For the MOV instruction, the src and dest addresses must both be of type byte or both of type word.
Usage MOV dest, src
dest - Reg/Mem
src - Reg/Mem/Imm

EXAMPLES
- **MOV AX, BX** — copy the value in BX into AX
- **MOV byte ptr [var], 5** — store the value 5 into the byte at location var

ADD instruction
It carries out the addition of source and destination, storing the result in the destination.

USAGE: ADD dest, src
dest-mem/reg
src -mem/reg/imm

Note: The only types of addition not allowed are memory to memory and segment register.

EXAMPLES:
- **ADD AL,BL;** AL=AL+BL

- **ADD CL,44h;** CL=CL+44hr

- **ADD [BX],AL;** AL adds byte contents of the data segment memory location addressed by BX with the sum stored in the same memory location

- **ADD BX,[SI+2];** The word contents of the data segment memory location addressed by SI+2 add to BX with the sum stored in BX

- **ADD BX,TEMP[DI];** The word contents of the data segment memory location addressed by TEMP+DI add to BX with the sum stored in BX

- **ADD BYTEPTR[DI],3;** A 3 adds to the byte contents of the data segment memory location addressed by DI with the sum stored in the same location

CMP instruction

DESCRIPTION: Subtracts source from destination and updates the fl not save result. Flags can subsequently be checked for condition

USAGE: CMP dest, src
dest-mem/reg/immed
src-mem/reg/immed

EXAMPLES:

- **CMP CL,BL** CL-BL and flags updated

- **CMP AX,2000H** AX-2000H and flags updated

- **CMP [DI],CH** CH is subtracted from the byte contents of the data segment addressed by DI and flags updated.

- **CMP AH,[1234H]** The byte contents of data memory location 1234h is subtracted from AH and flags updated.

JNZ/JNE instruction - Jump if not equal/ Jump if not zero

DESCRIPTION: JNZ : Jumps to the destination label mentioned in the instruction if the ZF is 0, else no action is taken.

Syntax: JNZ dest, JNE dest
[dest: address in the range of -128 bytes to +127 bytes from the address of instruction after JNZ/JNE]
Flags: the instruction has no effect on any flags.

EXAMPLE:
JNZ LABEL1; jumps to the address specified by LABEL1 if ZF=0

RET instruction - RET (Return) instruction is used to return control from a subroutine (function) to the calling program or routine.

Syntax: RET

RET instruction performs the following actions:

Stack Pop: The RET instruction pops data from the top of the stack. In the x86 architecture, the stack is used to store return addresses, local variables, and other data during subroutine calls.

Jump: It transfers control to the address popped from the stack. This address is usually the return address saved on the stack during a previous CALL instruction.

INC Instruction - is used to increment (add 1 to) the value of the specified operand. This instruction is often used for incrementing the value of a register or a memory location.

Syntax: INC destination

Example:

INC SI; Increment the value in register SI by 1

IV. Trying out Assembly Language Programs in DebugX

Example 1: Write an ALP to exchange a byte of data between two locations

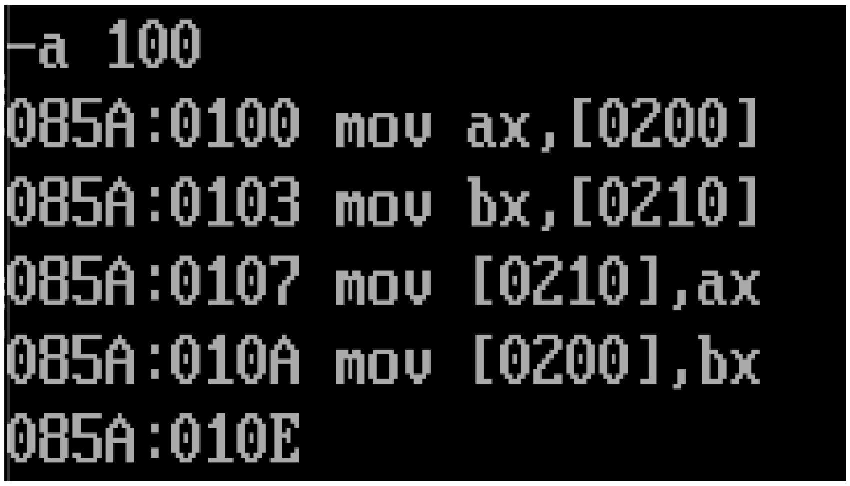
Solution:
Enter the following instruction in the code segment using -a.

mov ax, [0200] // content stored in the memory location 0200 will be moved/copied to AX register

mov bx, [0210] // content stored in the memory location 0210 will be moved/copied to BX register

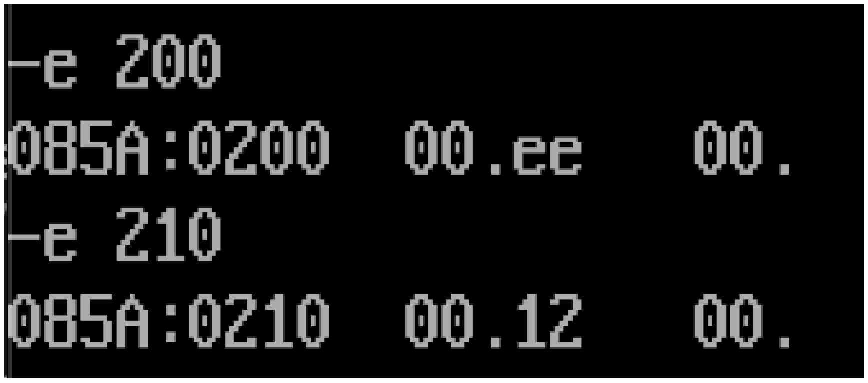
mov [0200], bx // content stored in BX will be copied to memory location 0200

mov [0210], ax // content stored in AX register will be copied to the memory location 0210



```
-a 100
085A:0100 mov ax,[0200]
085A:0103 mov bx,[0210]
085A:0107 mov [0210],ax
085A:010A mov [0200],bx
085A:010E
```

Use the e command to enter the data into the memory location 0200 & 0210



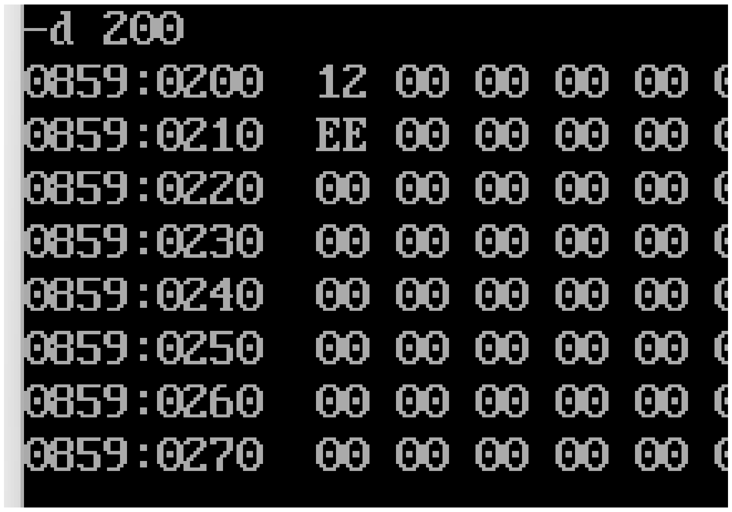
```
-e 200
085A:0200  00.ee   00.
-e 210
085A:0210  00.12   00.
```

ee is the value entered at memory location 0200
12 is the value entered at memory location 0210

Seeing the output before execution:

- Use the d command to view the data stored in memory
- location starting from 0200 till the memory location 0210
- Use the t command to run each line of machine code and see step-by-step execution

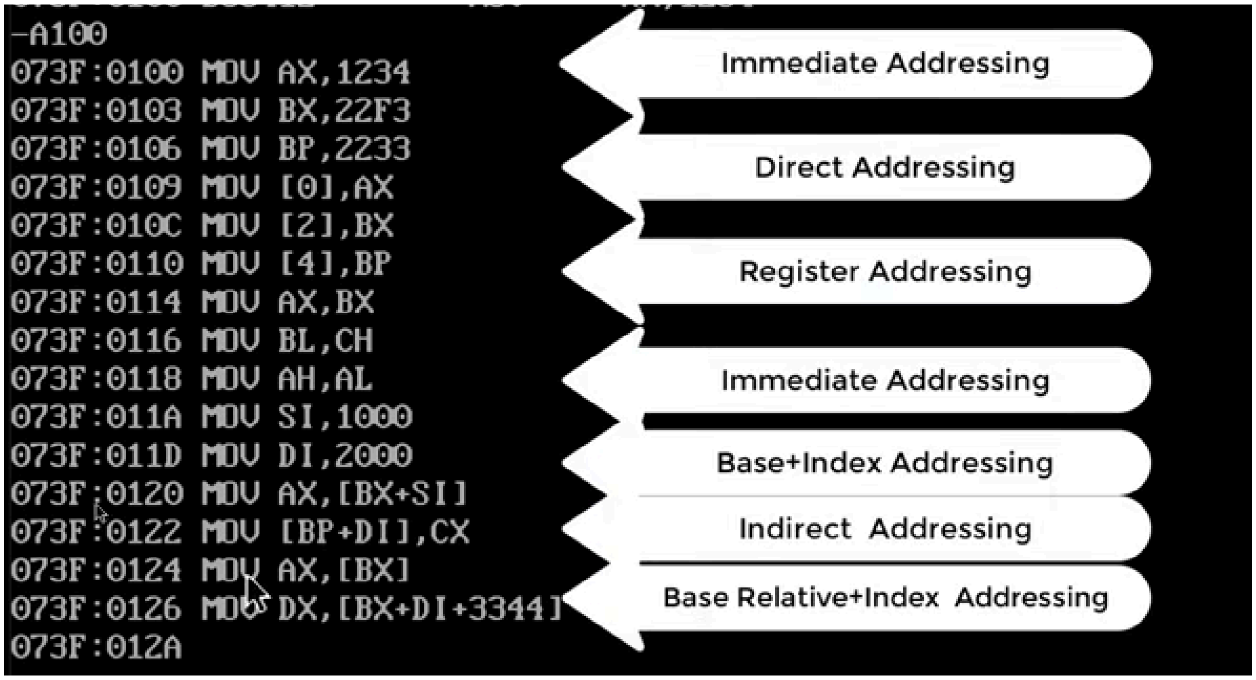
Output:
After executing the four instruction



```
-d 200
0859:0200  12 00 00 00 00
0859:0210  EE 00 00 00 00
0859:0220  00 00 00 00 00
0859:0230  00 00 00 00 00
0859:0240  00 00 00 00 00
0859:0250  00 00 00 00 00
0859:0260  00 00 00 00 00
0859:0270  00 00 00 00 00
```

Example 2: Using different addressing modes

a) Enter the following code instructions which represent different addressing modes



```
-A100
073F:0100 MOV AX,1234
073F:0103 MOV BX,22F3
073F:0106 MOV BP,2233
073F:0109 MOV [0],AX
073F:010C MOV [2],BX
073F:0110 MOV [4],BP
073F:0114 MOV AX,BX
073F:0116 MOV BL,CH
073F:0118 MOV AH,AL
073F:011A MOV SI,1000
073F:011D MOV DI,2000
073F:0120 MOV AX,[BX+SI]
073F:0122 MOV [BP+DI],CX
073F:0124 MOV AX,[BX]
073F:0126 MOV DX,[BX+DI+3344]
073F:012A
```

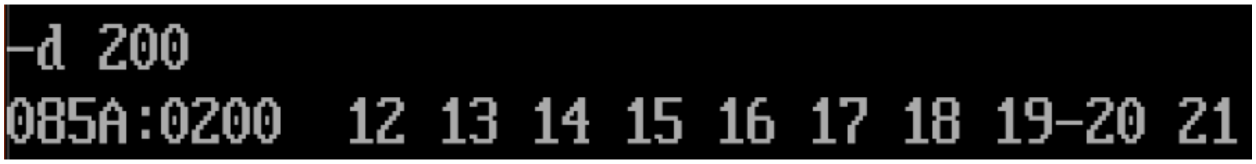
b) Type U to observe how these instructions unassemble

c) Type T to trace each instruction

d) Keep checking the change in IP register using the R command.

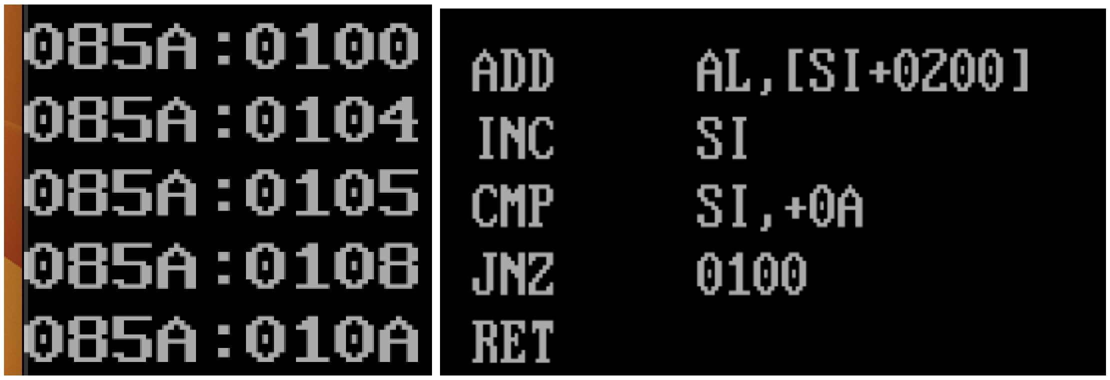
e) Check the change in data memory using D <address> which will show the content of memory that is updated.

Example 3: Write an ALP to calculate the sum of 10 array numbers 12,13,14,15,16,17,18,19,20,21 present in the memory location data segment with an offset of 0200.



```
-d 200
085A:0200  12 13 14 15 16 17 18 19-20 21
```

Solution:



```
085A:0100   ADD     AL,[SI+0200]
085A:0104   INC     SI
085A:0105   CMP     SI,+0A
085A:0108   JNZ     0100
085A:010A   RET
```

Note and follow-up question -> we are adding numbers to al thus what if al overflows that it becomes greater than ff say for example array numbers are 12 and ff then the sum will overflow how can we handle this in our code think about this!

Also, learn more variations of jump instructions.