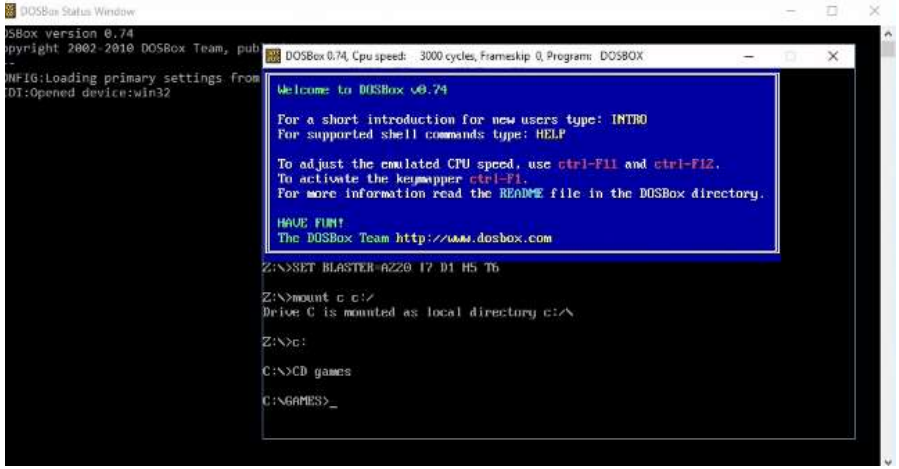
Microprocessing and Interfacing
Lab Session 1
Intro to Debug & DebugX

Anubhav Elhence

What is DOS BOX?

- DOSBox is a free and open-source emulator program that allows you to run MS-DOS applications on modern operating systems, such as Windows, MacOS, and Linux. It creates a virtual environment that simulates the behavior of an IBM PC compatible computer running MS-DOS. This allows you to run old DOS-based software and games that may not be compatible with newer operating systems.

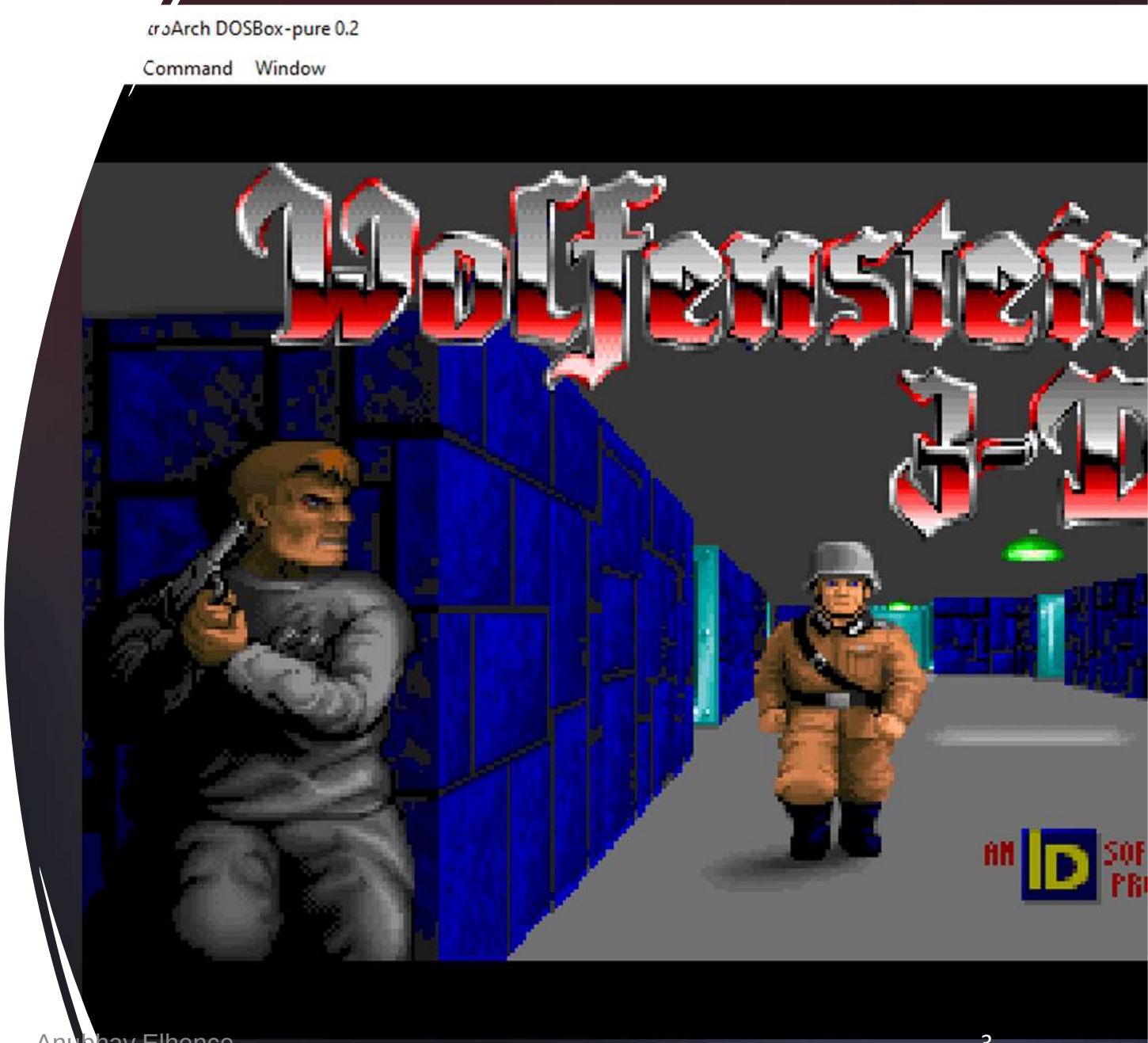
- DOSBox emulates the CPU, memory, and other hardware components of an IBM PC, including the 8086 processor. It also emulates the BIOS and other firmware that is present on a real PC. This allows it to run software that was written for the 8086 processor and the MS-DOS operating system.



DOS-BOX

- The program is configurable, allowing the user to change various settings such as CPU cycles, sound, graphics and even network support. It also supports various types of mount points, which allows you to use files and directories on your host system as if they were on a virtual drive within the DOSBox environment.

- In summary, DOSBox is a powerful emulator that allows you to run old MS-DOS software and games on modern operating systems, by emulating the hardware and firmware of an IBM PC compatible computer, including the 8086 processor and the MS-DOS operating system.



What is MASM?

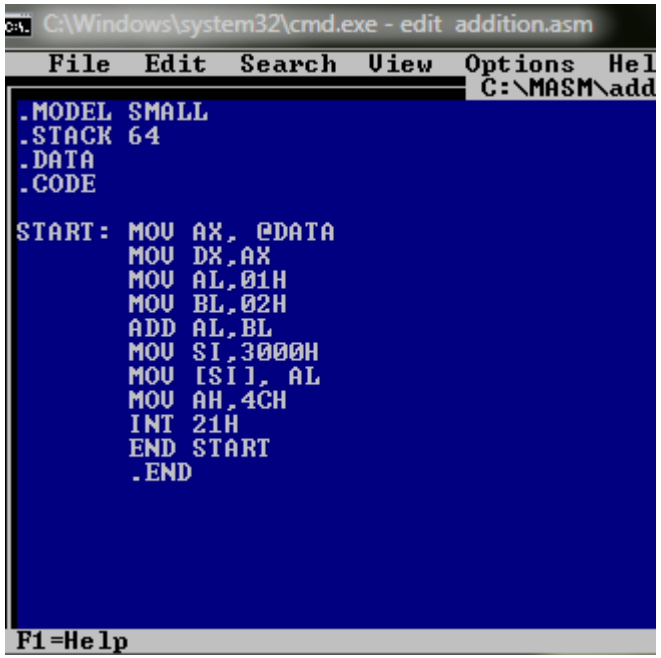
- MASM stands for Microsoft Macro Assembler. It is an x86 assembly language compiler and development environment developed by Microsoft for their operating systems. It was first released in the early 1980s for the 8086 and 8088 processors, and later versions were released for the 286, 386, and higher processors.

- MASM provides a set of macro instructions that can be used to simplify assembly language programming and improve code readability. These macros can be used to define procedures, data structures, and other high-level constructs that are not available in traditional assembly languages.

- MASM also provides a set of powerful debugging and development tools, such as a symbol table, a code disassembler, and a source-level debugger. These tools allow developers to easily identify and fix errors in their assembly language code.

- The most recent version of MASM is the MASM32. It is a free and open-source software and it supports all versions of Windows.

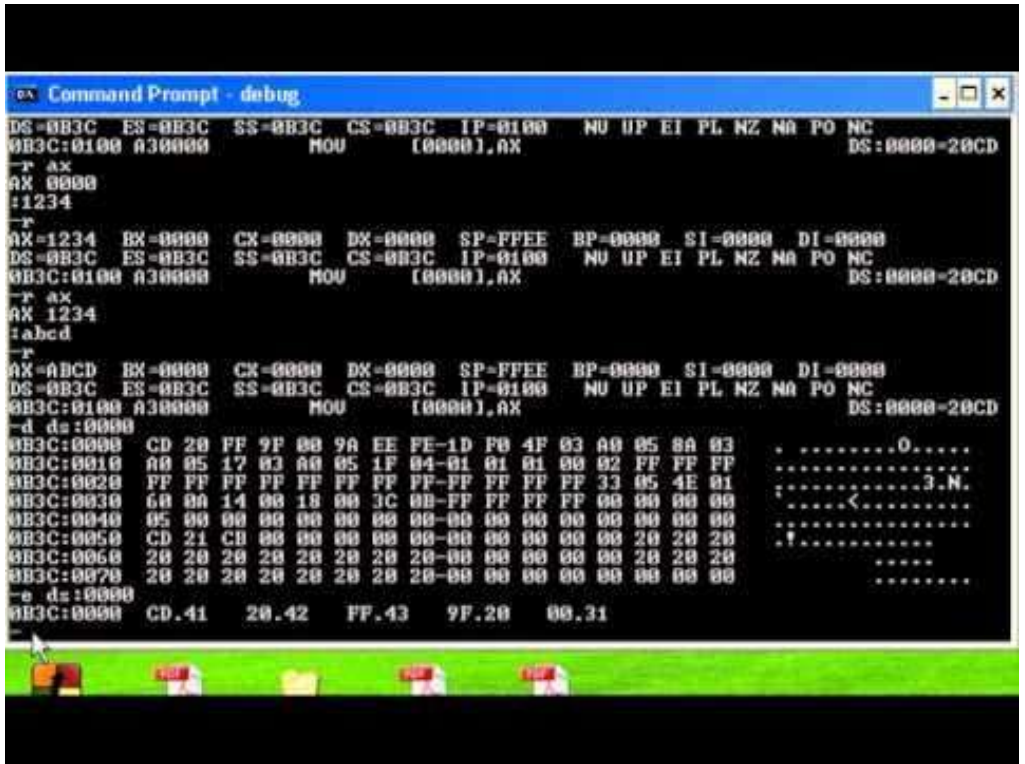
- In summary, MASM is an x86 assembly language compiler and development environment developed by Microsoft for their operating systems, that provides a set of macro instructions and powerful debugging and development tools for assembly language programming.

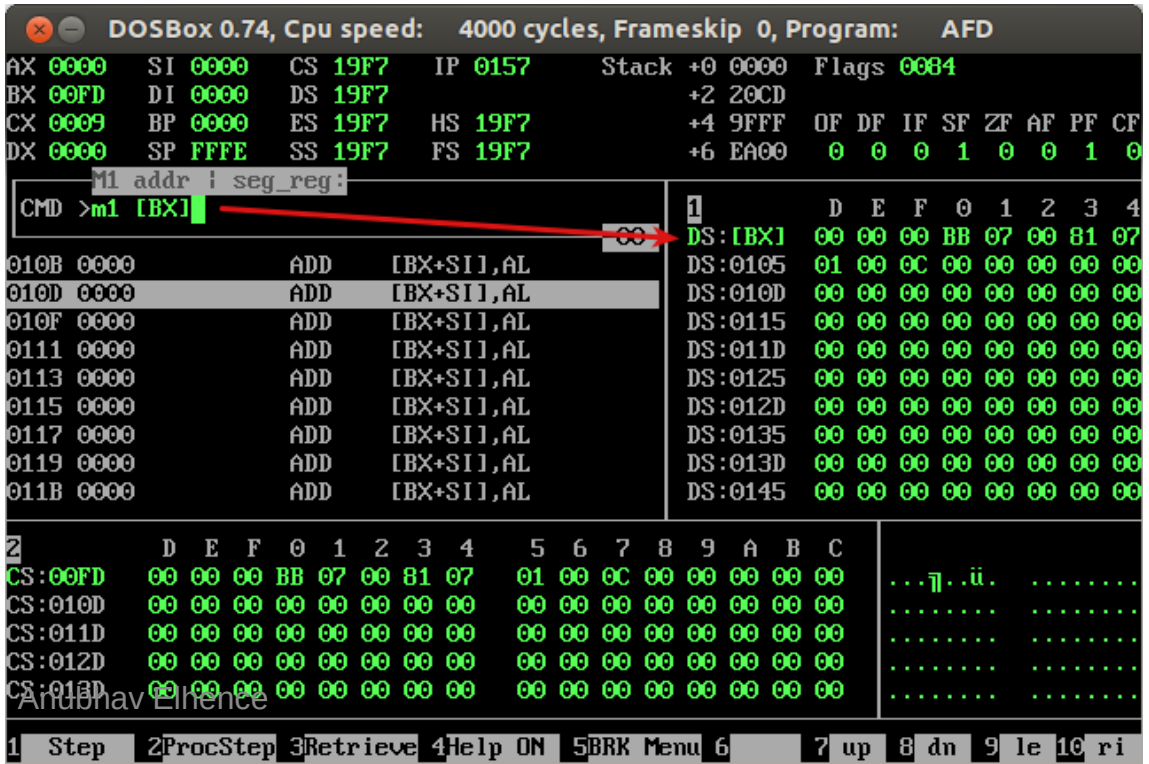


What is DebugX?

- DEBUG, supplied by MS-DOS, is a program that traces the 8086 instructions. Using DEBUG, you can easily enter 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 break point.

- 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.





Installing DosBox and DebugX

DOSBOX

- If you have a sytem with Windows 8 and higher as an operating system - you will not have access to the command line prompt. In this case you can use DOSBOX. Download the software from the link below. The setup file is executable just run it - DosBox gets installed automatically.

- https://bitsiotlab.com/mup-lab-content/
- If you have a Linux system - you can refer to this Wiki on DosBox in Ubuntu. It gives all details on installation and usage of DosBox in Ubuntu
- https://help.ubuntu.com/community/DOSBox

MASM611

- Microsoft Assembler- Download the software from the link. Extract the files into any Folder of your choice. The MASM executable will be in MASM611/BIN

Debug & DebugX

- Debugger - To test output/behavior of Programs. Download the software from the link. Extract the files into MASM611/BIN Folder

Installing DosBox and DebugX

1. On the above page, click on “Link to DosBox” installer, and install DosBox (There should be an icon on the desktop after this).

2. Now, click on the “Link to MASM”, and extract the MASM611 folder.

3. Copy the MASM611 folder directly in the 'D' drive (Remember the drive, this will be important later)

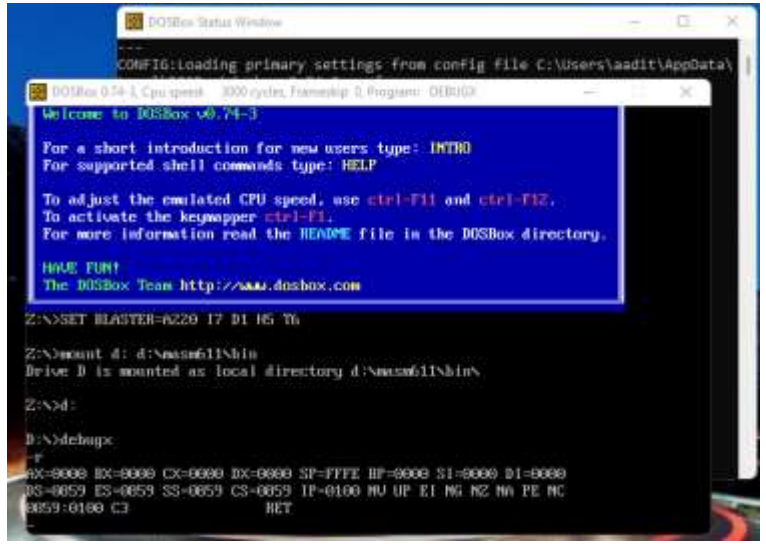
4. Copy the content of "debug125” folder in the MASM611→BIN folder.

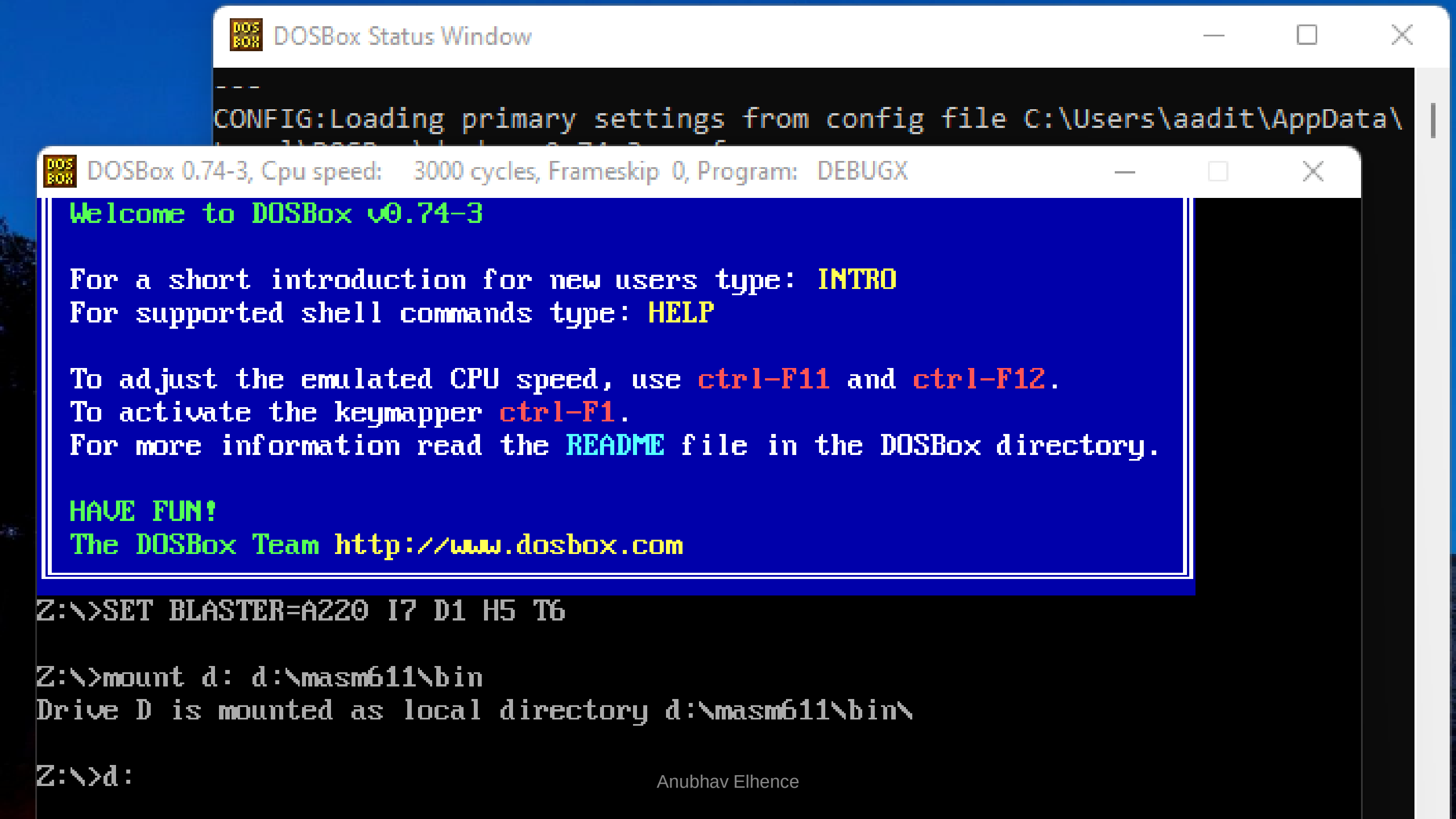
5. You don't need to install anything from/within MASM611 folder (Everything is already pre-compiled in the folder)

How to start DebugX?

- Once you have all the software installed, this is the procedure you'll need to follow to start DebugX. In short, we need to mount the drive where the MASM611 folder is stored (D Drive here). You will have to do this many times, please note it down:-

1. Click on the DosBox icon on your desktop (This opens two windows, you only need to focus on the one with the command prompt).
2. The command prompt will read 'Z:\>', type in "mount d: d:\masm611\bin" (If successful you should get a message 'Drive D is mounted as local directory')
3. Now type "d:" to change the command prompt to 'D:\>'
4. Type "debugx" to start the program (If succsessful, the cursor moving to the next line.)
5. You can return to DosBox, by typing "q" (Quit command)




DOSBox Status Window

CONFIG:Loading primary settings from config file C:\Users\aadit\AppData\
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUGX
Welcome to DOSBox v0.74-3
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:\masm611\bin
Drive D is mounted as local directory d:\masm611\bin\
Z:\>d:
Anubhav Elhence

DebugX Basic Conventions

- DebugX is NOT case sensitive.
- DEBUGX recognizes ONLY Hexadecimal numbers (without a trailing ‘H’ by default, so ‘11’ is interpreted as seventeen, not eleven!!)
- DEBUGX displays a list of the commands and their parameters by entering a "?" at the command prompt.
- Segment and offset are specified as Segment:Offset
- Spaces in commands are only used to indicate separate parameters

List of DebugX commands

Command Syntax	Description
Register	
RX	Activates 32-bit registers (‘386 regs on’)
R	Shows the 16-bit registers (Default) or the 32-bit registers, if rx was used before
R <register>	**View a register** and change its value at the prompt
Execution	
A :<offset>	Assemble- Prompts the code segment to **write instructions** (assembled into machine code)
U <offset>	Unassemble- **Displays the Symbolic code** (instructions) written at the offset (from CS)
T	Trace- Execute commands at CS:IP, i.e. **one at a time** (debugging)
G <address of last instruction>	Go- Executes commands **all at once**, until the address specificed (Change IP Value using R first!)
Data	
D :<offset>	Dump- **View the data** at this address (Little Endian)
E :<offset>	Enter- **Edit data** at this address by changing the value at the prompt
Misc.	
?	**View** all debugx commands
Q	**Quit** debugx

Follow Along exercise:

- Let's first view the contents of the registers

```
D:\>debugx
-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
-
```

- Now let's write a simple program to add two numbers

```
-a
0859:0100 mov ax,01
0859:0103 mov bx,02
0859:0106 add ax,bx
0859:0108 _
```

- Now let's view where the instructions got stored in memory

```
-u 100
0859:0100 B80100          MOV     AX,0001
0859:0103 BB0200          MOV     BX,0002
0859:0106 01D8            ADD     AX,BX
0859:0108 0000            ADD     [BX+SI],AL
0859:010A 0000            ADD     [BX+SI],AL
0859:010C 0000            ADD     [BX+SI],AL
0859:010E 0000            ADD     [BX+SI],AL
```

- Let's execute these commands one by one and keep on looking at the register contents.

- But before executing, Let's see where the IP is pointing to.

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE 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 B80100          MOV     AX,0001
```

- executing MOV AX,0001H

```
-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 B80100            MOV     AX,0001
-
-t
AX=0001 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 BB0200            MOV     BX,0002
```

- Observe now, where the IP is pointing to. Similarly let's do it two more times.

```
-t
AX=0001 BX=0002 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0106 NV UP EI NG NZ NA PE NC
0859:0106 01D8              ADD     AX,BX
-t
AX=0003 BX=0002 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0108 NV UP EI PL NZ NA PE NC
0859:0108 0000              ADD     [BX+SI],AL
```

Another follow along example:

- Now let's try to see the contents of random memory locations

```
-D DS:200
0859:0200  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0240  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0250  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0270  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

- Lol ! Nothing of much value here. Let's try another one.

```
-D DS:100
0859:0100  B8 01 00 BB 02 00 01 D8-00 00 00 00 00 00 00 00 ................
0859:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

- We see something. Does it look familiar. OFCOURSE YESSS !!!

- This is the BYTE equivalent code of our ALP

```
-u DS:100
0859:0100 B80100          MOV     AX,0001
0859:0103 BB0200          MOV     BX,0002
0859:0106 01D8            ADD     AX,BX
0859:0108 0000            ADD     [BX+SI],AL
```

```
-D DS:100
0859:0100  B8 01 00 BB 02 00 01 D8-00 00 00 00 00 00 00 00 ................
0859:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
```

- Isn't this amazing to note that the instructions also lie in the same segment as our data does and can be read/copied/executed/etc. Now let's store 0xDEAD at memory location DS:155h

PAUSE THE VIDEO

- let's store 0xDEAD at memory location DS:155h

```
-a DS:108
0859:0108 MOV BX,DEAD
0859:010B MOV [155],BX
0859:010F _
```

```
-u
0859:0108 BBADDE            MOV     BX,DEAD
0859:010B 891E5501          MOV     [0155],BX
```

```
-r
AX=0003 BX=0002 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0108 NV UP EI PL NZ NA PE NC
0859:0108 BBADDE            MOV     BX,DEAD
```

```
-t
AX=0003 BX=DEAD CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=010B NV UP EI PL NZ NA PE NC
0859:010B 891E5501          MOV     [0155],BX                       DS:0155=0000
```

```
-t
AX=0003 BX=DEAD CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=010F NV UP EI PL NZ NA PE NC
0859:010F 0000              ADD     [BX+SI],AL
```

```
-D DS:100
0859:0100  B8 01 00 BB 02 00 01 D8-BB AD DE 89 1E 55 01 00 .............U..
0859:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0150  00 00 00 00 00 AD DE 00-00 00 00 00 00 00 00 00 ................
0859:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0859:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-
```

- Finally, let's copy the content of memory location DS:0155h to CX register

```
-a
0859:010F MOV CX,[155]
0859:0113
```

```
-r
AX=0003 BX=DEAD CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=010F NV UP EI PL NZ NA PE NC
0859:010F 8B0E5501          MOV     CX,[0155]                    DS:0155=DEAD
-t
AX=0003 BX=DEAD CX=DEAD DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=0859 ES=0859 SS=0859 CS=0859 IP=0113 NV UP EI PL NZ NA PE NC
0859:0113 0000              ADD     [BX+SI],AL                   DS:DEAD=00
-
```

Ten Offences against the MASM Compiler

- If you commit any of these ten offences, no matter how well you have coded in the past, immediately, your program will behave abnormally.... Until you rectify your mistake and say sorry to the compiler.

- 1st offence:
  Thou shalt never attempt to manually change the contents of segments using debugX or fiddle with the data segment where the instructions are written.

- Other offences will be covered later on.

DebugX commands

- A: Assemble

  The Assemble command (A) is used to enter assembly mode. In assembly mode, DEBUG prompts for each assembly language statement and **converts the statement into machine code** that is then **stored in memory**. The optional start address specifies the address at which to assemble the first instruction. The default start address is 100h. A blank line entered at the prompt causes DEBUG to exit assembly mode.

  Syntax: A [address]

- D: Dump

  The Dump command (D), when used without a parameter, causes DEBUG to **display the contents of the 128-byte block of memory** starting at CS:IP if a target program is loaded, or starting at CS:100h if no target program is loaded. The optional range parameter can be used to specify a starting address, or a starting and ending address, or a starting address and a length. Subsequent Dump commands without a parameter cause DEBUG to display the contents of the 128-byte block of memory following the block displayed by the previous Dump command.

  Syntax: D [range]

- R: Register

  The Register command (R), when used without a parameter, causes DEBUG to **display the contents of the target program's CPU registers**. The optional register parameter will cause DEBUG to display the contents of the register and prompt for a new value.

  Syntax: R [register]

  Syntax: R [register] [value]

DebugX commands

- T: Trace
  - The Trace command (T), when used without a parameter, causes DEBUG to **execute the instruction at CS:IP**. Before the instruction is executed, the contents of the register variables are copied to the actual CPU registers. After the instruction has executed, and updated the actual CPU registers (including IP), the contents of the actual CPU registers are copied back to the register variables and the contents of these variables are displayed. The optional address parameter can be used to specify the starting address. The optional count parameter can be used to specify the number of instructions to execute. To differentiate the address parameter from the count parameter, the address parameter must be preceded with an "=". Note that the first byte at the specified address must be the start of a valid instruction.

    Syntax: T [=address] [number]

Tasks to be completed

1. Using three addressing modes (Immediate, Register, Register-Indirect), write instructions to
   - Move the value 1133 into the register AX.
   - Swap the lower and higher bytes in AX and move them into BX (If AX is pqrs, BX should be rspq)
   - Move the value in BX to the memory location at an offset of 20 (from BX)

   - Note down the machine code equivalents of the four MOV statements.
   - (***Hint:*** You need to use the following commands- A to write the instructions, and U to view the machine code and unassembled instructions, T to execute and D to view the memory location)

2. Move the first letter of your name (ASCII Character) to the location DS:0120
   - (Hint: Recall the rules for the Immediate addressing mode)

3. Fill 32 (decimal) bytes of the Extra Segment with ASCII characters for the first two letters of your name. (Like "ABABAB…")
   - (***Hint:*** Use the F (Fill) command to fill a memory region with a byte pattern
   - To fill for example the first 8000h bytes of current data segment with pattern 55:
   - **F** 0 L 8000 55
   - [Syntax: **F** <start-address> L <range> <pattern>])

Thank You