Chapter 03

Assembly language

Table of content

Introduction

- The process of developing assembly language programs is a path that runs from what we call **source code** that you can read, to something called **machine code** that the CPU can execute. In the middle is a resting point called **object code** that we'll take up a little later.
- The process of creating true machine code programs is one of translation. You must start with something that you and the rest of us can read and understand, and then somehow convert that to something the CPU can understand and execute.
- While the CPU can work only in binary, it can do so at very high speeds. However, it is quite tedious and slow for humans to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called **machine language** and in the early days of the computer, programmers actually coded programs in machine language. Eventually, **assembly languages** were developed, which provided **mnemonics** for the machine code instructions, plus other features that made programming faster and less prone to error.
- The term **mnemonics** is frequently used in computer science and engineering literature to refer to **codes and abbreviations** that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an assembler.
- **Assembly language** is referred to as **low-level language** because it deals directly with the internal structure of the CPU.
- **To program in assembly language**, the programmer must know the number of registers and their size, as well as other details of the CPU

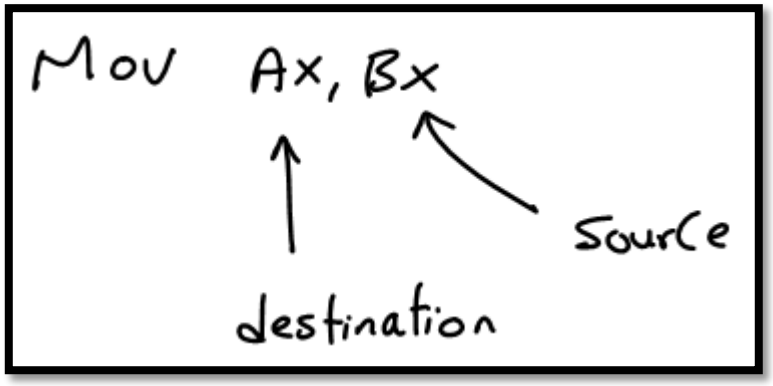
- Today, one can use many different programming languages, such as Pascal, BASIC, C, and numerous others. These languages are called high-level languages because the programmer does not have to be concerned with the internal details of the CPU. Whereas an **assembler** is used to translate an assembly language program into machine code, high-level languages are translated into machine code by a program called a **compiler**.
- There are numerous assemblers available for translating 80x86 assembly language programs into machine code. One of the most commonly used assemblers, **TASM** by **Borland**. But here in this chapter we are focused on **DEBUG**.
- Each of the CPU's many machine instructions has a corresponding mnemonic in assembly language. As the word suggests, these mnemonics began as devices to help programmers **remember** a particular binary machine instruction.
    - For example, the **mnemonic** for binary machine instruction **9CH**, which **pushes the flags register onto the stack**, is PUSHF—which is a country mile **easier to remember** than 9CH.
- When you write your source code file in assembly language, you will arrange series of mnemonics, typically one mnemonic per line in the source code text file. A portion of a source code file might look like this:

```
MOV  AH,12H        ; 12H is Motor Information Service
MOV  AL,03H        ; 03H is Return Current Speed function
XOR  BH,BH         ; Zero BH for safety's sake
INT  71H           ; Call Motor Services Interrupt
```

- Here, the words **MOV**, **XOR**, and **INT** are the mnemonics. The numbers and other items to the immediate right of each mnemonic are that mnemonic's **operands**. There are various kinds of operands for various machine instructions, and some instructions (such as PUSHF mentioned previously) have no operands at all.

■ MOV

- We introduce assembly language programming with two widely used instructions: the **MOV** and **ADD** instruction.



- Transfer (make a copy) the word contents of the source register BX into the destination register
- The source never changes, but the destination almost always changes
- It is important for instructions to use registers that are the same size.
    - Never mix an 8-bit register with a 16-bit register, an 8-bit register with a 32-bit register, or a 16-bit register with 32-bit register, because this is not allowed by the microprocessor and results in an error when assembled.
    - This is even true when a **MOV AX, AL** or a **MOV EAX, AL** instruction may seem to make sense. Of course, the **MOV**

AX, AL or **MOV EAX, AL** instructions are not allowed, because these registers are of different sizes.

- Note that a few instructions, such as **SHL DX, CL**, are exceptions to this rule, as indicated in later chapters.

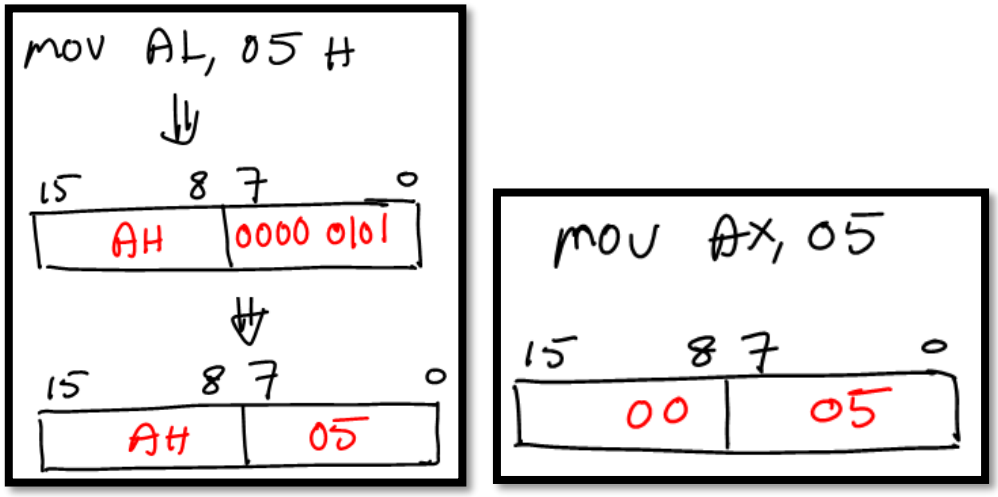
- It is also important to note that none of the MOVS instructions affect the flag bits.
- A segment-to-segment register MOV instruction is virtually the only type of register MOV instruction not allowed

MOV DS,2345H	Illegal instruction
MOV AX,2345H MOV DS,AX	Do the same instruction, but is it is acceptable

- Also, note that the code segment register may not be changed by a MOV instruction, because the address of the next instruction is found in both IP/EIP and CS. If only CS were changed, the address of the next instruction would be unpredictable. Therefore, changing the CS register with a MOV instruction is not allowed.

MOV CL,55H	move 55H into register CL
MOV DL,CL	copy the contents of CL into DL (now DL=CL=55H)
MOV AH,DL	copy the contents of DL into AH (now AH=DL=55H)
MOV AL,AH	copy the contents of AH into AL (now AL=AH=55H)
MOV BH,CL	copy the contents of CL into BH (now BH=CL=55H)
MOV CH,BH	copy the contents of BH into CH (now CH=BH=55H)
MOV AL,**0**AH	Always start the hexadecimal number with a number AL = **0AH** (**this is a number**, and is not the most significant byte of AX)
MOV CH,**0**BH	_____

- **Example**



- Data can be moved among all the registers except the **flag register**. The exception of the flag register means that there is no such instruction as MOV FR, AX.
  - Loading the flag register is done through other means.

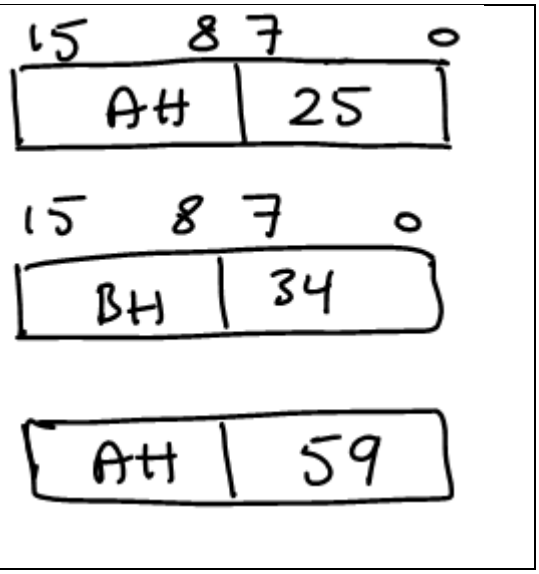
MOV CX,58FCH	Move 58FCH into AX (legal)
MOV DX,6678H	Move 6678H into DX (legal)
MOV SI,924BH	legal
MOV BP,2459H	legal
MOV DS,2341H	illegal
MOV CX,8876H	legal
MOV CS,3F47H	illegal
MOV BH,99H	legal

ADD

- The ADD instruction has the following format:

```
ADD destination, source
```

  The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination.
- To add two numbers such as 25H and 34H, each can be moved to a register and then added together:

MOV AL,25H
MOV BL,34H
ADD AL,BL
	

✍ **Note:** **Immediate** addressing o MOV AL, 25H **Register** addressing o ADD AL, BL	

- Executing the program above results in AL=59H (25H +34H = 59H) and BL = 34H. Notice that the contents of BL do not change. The program above can be written in many ways, depending on the registers used. Another way might be:

```
MOV DH, 25H     ;move 25 into DH
MOV CL, 34H     ;move 34 into CL
ADD DH, CL      ;DH=DH+CL
```

- The results above results in DH = 59H and CL = 34H. There are always many ways to write the same program.
  - One question that might come to mind after looking at the program above is whether it is necessary to move both data items into registers before adding them together.
  - The answer is no, it is not necessary. Look at the following variation of the same program

```
MOV DH, 25H     ;move 25 into DH
ADD DH, 34H     ;DH=DH+34H
```

  - In the case above, while one register contained one value, the second value followed the instruction as an operand. This is

called an immediate operand. The examples shown so far for the ADD and MOV instructions show that the source operand can be either a register or immediate data. In the examples above, the destination operand has always been a register.

- The largest number that an 8-bit register can hold is FFH. To use numbers larger than FFH (255 decimal), 16-bit registers such as AX, BX, CX, or DX must be used. For example, such as 34EH and 6A5H, the following program can be used:

```
MOV AX,34EH      ;move 34EH into AX
MOV DX,6A5H      ;move 6A5H into DX
ADD DX,AX        ;add AX to DX➔   DX = DX +AX
```

- Running the program above gives DX = 9F3H (34E+6A5 = 9F3) and AX = 34E. Again, any 16-bit non-segment registers could be used to perform the action above:

```
MOV CX,34EH      ;move 34EH into AX
ADD CX,6A5H      ; CX = CX +6A5H
```

- The general-purpose registers are typically used in arithmetic operations. Register AX is sometimes called the accumulator.

■ LOOP

Example

Assume that a program is being written to add 5 bytes of data, such as 25H, 12H, 15H, 1FH, and 2BH, where each byte represents a person's daily overtime pay

❶ one way to add this is,

```
MOV AL,00H      ;initialize AL
ADD AL,25H      ;add 25H to AL
ADD AL,12H      ;add 12H to AL
ADD AL,15H      ;add 15H to AL
ADD AL,1FH      ;add 1FH to AL
ADD AL,2BH      ;add 2BH to AL
```

- In the program above, the data and code are mixed together in the instructions. The problem with writing the program this way is that if the data changes, the code must be searched for every place the data in included and the data retyped.
- For this reason, the idea arose to set aside an area of memory strictly for data. In 80x86 microprocessors, the area of memory set aside for data is called the **data segment**. Just as the code segment is associated with CS and IP, the data segment uses register **DS** and an offset value

❷ **another way**

Assume that the offset for the data segment begins at 200H.

DS:0200 = 25H

DS:0201 = 12H

DS:0202 = 15H

DS:0203 = 1FH

DS:0204 = 2BH

And the program can be rewritten as follows

```
MOV AL,0        ; clear AL
ADD AL, [0200]  ;add the contents of DS:200 to AL
ADD AL, [0201]  ;add the contents of DS:201 to AL
ADD AL, [0202]  ;add the contents of DS:202 to AL
ADD AL, [0203]  ;add the contents of DS:203 to AL
ADD AL, [0204]  ;add the contents of DS:204 to AL
```

- Notice that the **offset address is enclosed in brackets**. The brackets indicate that the operand represents the address of the

data and not the data itself. If the brackets were not included, as in "**ADD AL,200**", the CPU would attempt to add 200 to the contents of AL.

✍ **NOTE:**

DEBUG assumes that all the numbers are in hex (no **H** suffix is required), whereas **MASM** assumes that they are in decimal and the **H** must be included

- This program will run with any set of data. Changing the data has no effect on the code. Although this program is an improvement over the preceding one, it can be improved even further.
  - If the data had to be stored at a different offset address, say 450H, the program would have to be rewritten. One way to solve this problem would be to use a **register** to hold the **offset** address.
  - The 8088/86 allows only the use of registers BX, SI, and DI as offset registers for the data segment. The term pointer is often used for a register holding an offset address. Below we use BX as a pointer

❸ another way: use a register to hold the offset address

```
MOV AL,0          ;initialize AL
MOV BX,0200H      ;BX points to the offset address of first byte
ADD AL,[BX]       ;add the first byte to AL
INC BX            ; increment BX to point to the next byte
ADD AL,[BX]       ;add the next byte to AL
INC BX            ; increment the pointer
ADD AL,[BX]       ;add the next byte to AL
INC BX            ; increment the pointer
ADD AL,[BX]       ;add the next byte to AL
INC BX            ; increment the pointer
ADD AL,[BX]       ;add the last byte to AL
```

- The INC instruction adds 1 to (increments) its operand. INC BX achieves the same results as ADD BX,1. For the program above if the offset address where data is located is changed, only one instruction will need to be modified and the rest of the program will be unaffected.

✍ **NOTE:**

Examining the program above shows that there is a pattern of two instructions being repeated. This leads to the idea of using a **LOOP** to repeat certain instructions. Implementing a loop requires familiarity with the **flag register**.

❹ LOOP

- The term loop refers to a set of instructions that is repeated a number of times
- One of the most widely used applications of the flag register is the use of the zero flag to implement program loops.
- For example, to add 5 bytes of data, a counter can be used to keep track of how many times the loop needs to be repeated.
- Each time the addition is performed the counter is decremented and the zero flag is checked. When the counter becomes zero, the zero flag is set (ZF = 1) and the loop is stopped
- The following shows the implementation of the looping concept in the program, which adds 5 bytes of data. Register CX is used to hold the counter and BX is the offset pointer (SI or DI could have been used instead).

```
        MOV CX,05
        MOV BX,0200H
        MOV AL,00
ADD_LP: ADD AL,[BX]
        INC BX
        DEC CX
        JNZ ADD_LP
```

■ DEBUG Session

- DEBUG is called an assembly level debugger because it displays only assembly mnemonics and M/C instruction
- Even if you use it to debug a compiled C++ program, you will not see the program's source code, instead you will see a disassembly of the program's machine instructions
    - Example: **`debug sample.exe`**

      To trace or execute a M/C language program

DEBUG's Bag of Tricks

It's well worth taking a page or so simply to describe what sorts of things **DEBUG** can do before actually showing you how they're done. It's actually quite a list:

- Display the contents of memory (D), change memory (E) and lets you view registers (R) and variables
- Step through a program one line at a time (tracing)
- Fill a region of memory with a single value. If you have an area of memory that you want blanked out, DEBUG will allow you to fill that area of memory with any character or binary value.
- Search memory for sequences of binary values.
- **Assemble** new machine instructions into memory. DEBUG contains a simple assembler that does much of what NASM can do-one machine instruction at a time. If you want to replace a machine instruction somewhere within your program, you can type **MOV AX, BX** rather than have to look up and type the raw binary machine code values 8BH 0C3H — **a** : assemble
- Unassemble binary machine instructions into their mnemonics and operands. The flip side of the last feature is also possible:

DEBUG can take the two hexadecimal values 8BH and 0C3H and tell you that they represent the assembly language mnemonic **MOV AX, BX**.

Default values

when debug is first loaded, the following defaults are in effect

- all segment registers are set to the bottom of free memory
- IP is set to 0100H
- DEBUG reserves 256 bytes of stack space at the end of the current segment
- All of the available memory is allocated (reserved)
- BX:CX is set to the length of the current program
- The flags are set to the following values

NV	no overflow	OV
UP	direction flag up	DN
EI	interrupts enabled	DI
PL	sign flag = positive	NG
NZ	Zero flag = not zero	ZR
NA	No auxiliary	AC
NC	No carry	CY
PO	Parity ODD	PE

Command parameters

- **Hyphen (-):** command prompt
- Not case sensitive
- A command may be followed by one or more parameters
- A comma or space may be used to separate any two parameters

Address

- F000:100 — segment, offset
- DS:200 — segment register, offset
- 0AF5 — offset

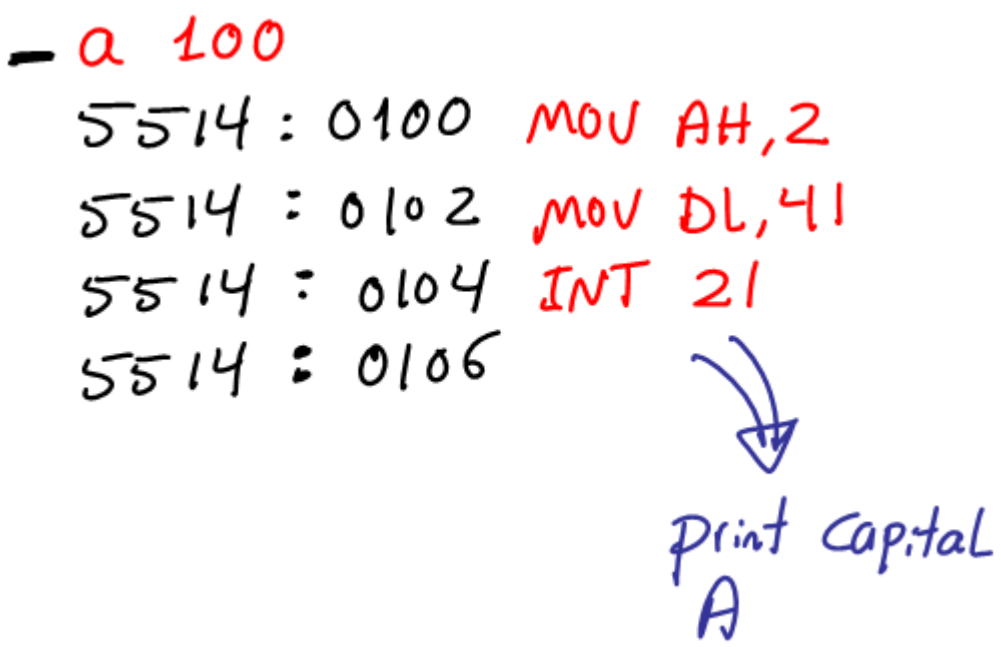
A (Assemble)

- Assemble a program into machine language
- **`A 100`**

  Assemble at CS:100H
- **`A`**

  Assemble from current location
- **`A DS:2000`**

  Assemble at DS:2000H

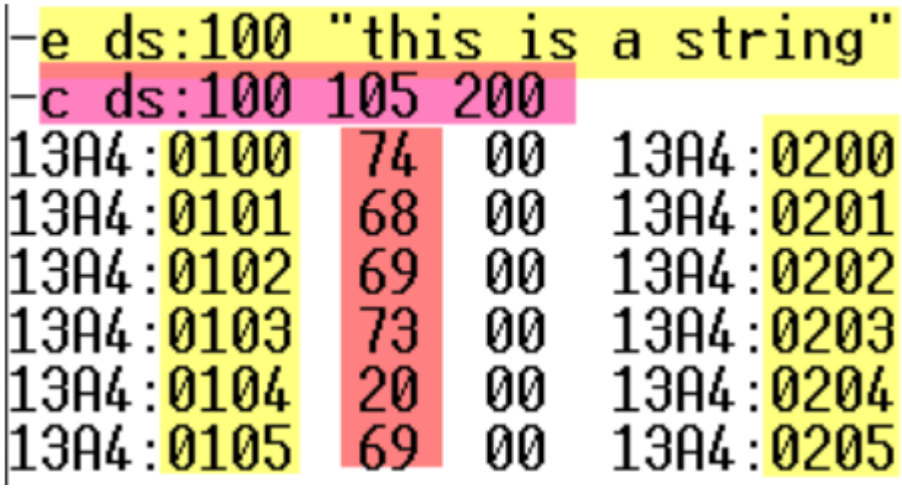


C (Compare)

- Compares bytes between a specified range with the same number of bytes at a target address
- C range address

C 100 105 200

- The bytes between DS:0100 and DS:0105 are compared to the bytes at DS:0200



D (Dump)

- Display memory on the screen as single bytes in both hexadecimal and ASCII
  - **D**

    Dump 128 bytes from the last referenced location

```
-d
13A4:0100  74 68 69 73 20 69 73 20-61 20 73 74 72 69 6E 67   this is a string
13A4:0110  00 00 00 00 00 00 00 00-00 00 00 00 34 00 93 13   ............4...
13A4:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13A4:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13A4:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13A4:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13A4:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
13A4:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

  - **D 100**

    The default segment is DS

  - **D F000:0**

    Segment – offset

  - **D ES:100**

    Segment register – offset

  - **D DS:0100 010F**

```
-d ds:0100 010f
13A4:0100  74 68 69 73 20 69 73 20-61 20 73 74 72 69 6E 67   this is a string
```

  - **D 150 15A**

Dump DS:0150 through 015A

- **D 915:0**

  Dump 128 bytes at offset zero from segment 0915H

- **D 0 200**

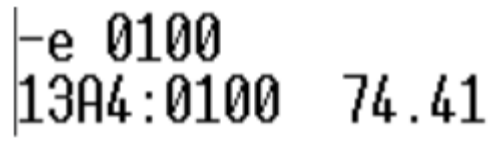
  Dump offsets 0-200 from DS

- **D 100 L 20**

  Dump 20H bytes starting at offset 100H from DS

E (Enter)

- Places individual bytes in memory
- Default: data segment register DS




```
-e 0100
13A4:0100  74.41
-d 0100
13A4:0100  41 68 69 73 20 69 73 20-61 20 73 74 72 69 6E 67   Ahis is a string
13A4:0110  00 00 00 00 00 00 00 00-00 00 00 00 34 00 93 13   ............4...
13A4:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

G (GO)

- Execute the program in memory
- **G**

  Execute from the current location to the end of the program

- **G 50**

  Execute from the current location and stop before the instruction at offset CS:50

```
-a 0100
13A4:0100 mov al,40
13A4:0102 add al,50
13A4:0104
-g 0104
```

```
AX=0090  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=13A4  ES=13A4  SS=13A4  CS=13A4  IP=0104   OV UP EI NG NZ NA PE NC
13A4:0104 4D            DEC     BP
```

L (Load)

- Load a file into memory at a given address
- To read a file, you must first initialize its name with **N** (name) command
- **L**

  Load named file into memory at CS:0100
- **L DS:0200**

  Load named file into memory at CS:0200

N (Name)

- Initialize a filename in memory before using the load or write commands

  N D:printme.exe

  L

```
-n d:printme.exe
-l
-u 0000
1417:0000 B81914        MOV     AX,1419
1417:0003 8ED8          MOV     DS,AX
1417:0005 B300          MOV     BL,00
1417:0007 B500          MOV     CH,00
1417:0009 B10A          MOV     CL,0A
1417:000B 8AD3          MOV     DL,BL
1417:000D 80C230        ADD     DL,30
1417:0010 B402          MOV     AH,02
1417:0012 CD21          INT     21
1417:0014 FEC3          INC     BL
1417:0016 E2F3          LOOP    000B
1417:0018 B44C          MOV     AH,4C
1417:001A CD21          INT     21
1417:001C 0000          ADD     [BX+SI],AL
1417:001E 0000          ADD     [BX+SI],AL
```

```
-g
0123456789
Program terminated normally
```

I (Input)

- Input a byte from a specified I/O port and displays the value in hexadecimal
- **`I port`**

  Port number between 0 and FFFF

P (Proceed)

- Executes one or more instructions or subroutines
- **`P = 200`**

  Executes a single instruction at CS:0200
- **`P = 150 6`**

  Executes 6 instructions starting at CS:0150
- **`P 5`**

  Executes the next 5 instruction

```
-a 0100
13A4:0100 mov cx,5
13A4:0103 mov ax,0
13A4:0106 add ax,cx
13A4:0108 loop 106
13A4:010A

-p

AX=0000  BX=0000  CX=0005  DX=0000  SP=FFEE
DS=13A4  ES=13A4  SS=13A4  CS=13A4  IP=0103
13A4:0103 B80000        MOV     AX,0000
```

```
-p

AX=0000  BX=0000  CX=0005  DX=0000  SP=FFEE
DS=13A4  ES=13A4  SS=13A4  CS=13A4  IP=0106
13A4:0106 01C8          ADD     AX,CX
-p

AX=0005  BX=0000  CX=0005  DX=0000  SP=FFEE
DS=13A4  ES=13A4  SS=13A4  CS=13A4  IP=0108
13A4:0108 E2FC          LOOP    0106
```

Q (Quit)

- Quit DEBUG

R (Register)

- Display the contents of one register – change – display registers – display flag – next instruction to be executed
- **R**

  Display the contents of all registers
- **R IP**

  Display the contents of IP and prompt for a new value
- **R CX**

  Display the contents of CX and prompt for a new value
- **R f**

  Display all flags and prompt for a new flag value

```
-r f
NV UP EI PL NZ NA PO NC  -ng
-r f
NV UP EI NG NZ NA PO NC  -pe
-rf
NV UP EI NG NZ NA PE NC  -zr
-rf
NV UP EI NG ZR NA PE NC  -pl
```

```
-r f
NV UP EI PL ZR NA PE NC  -ov
-r f
OV UP EI PL ZR NA PE NC  -cy
-r f
OV UP EI PL ZR NA PE CY  -ac
-r f
OV UP EI PL ZR AC PE CY  -
```

T (Trace)

- Execute one or more instructions starting at the current CS:IP location or at optional address
- **`T`**

  Trace the next instruction
- **`T 5`**

  Trace the next 5 instructions
- **`T = 105 10`**

  Trace 16 instructions starting at CS:105
- This command traces individual loop iterations, so you may want to use it to debug statements within a loop
- The **T** command traces into procedure calls, whereas the **P** (proceed) command executes a called procedure in its entirety

U (Unassemble)

- The U command translates memory into assembly language mnemonic — this is called disassembling memory
- **`U`**

  Disassemble the next 32 byte
- **`U 0`**

  Disassemble the next 32 byte at CS:0

- **`U 100 108`**

  Disassemble the bytes from CS:100 to CS:108

W (write)

- Writes a block of memory to a file.
- To write to a file, its name must first be initialized with the N command

  `N example.com`

  `R BX 0 R CX 20`

  `W`

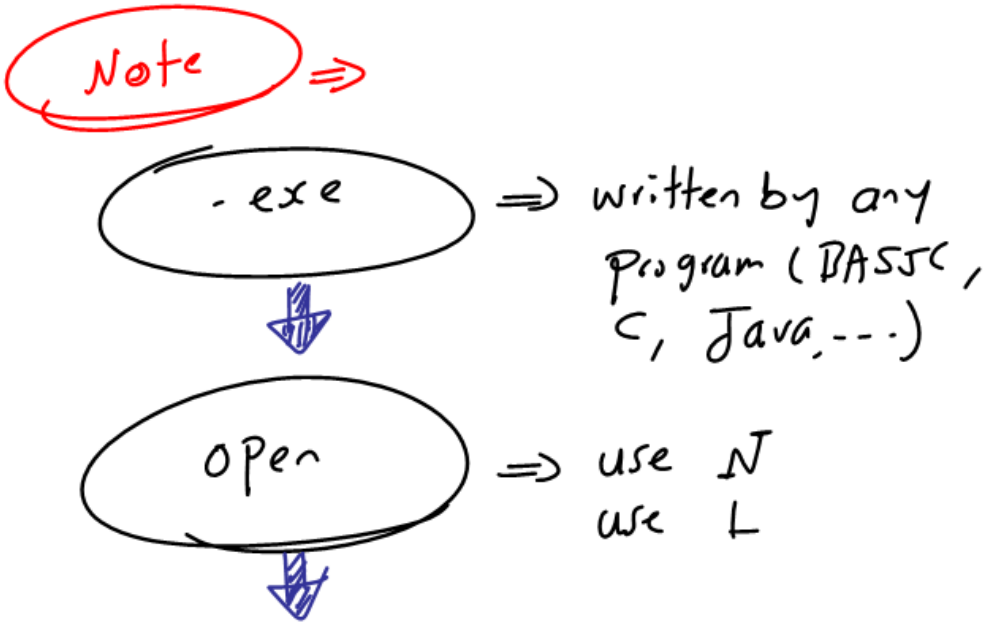
  Write 20H bytes to the file starting at CS:100

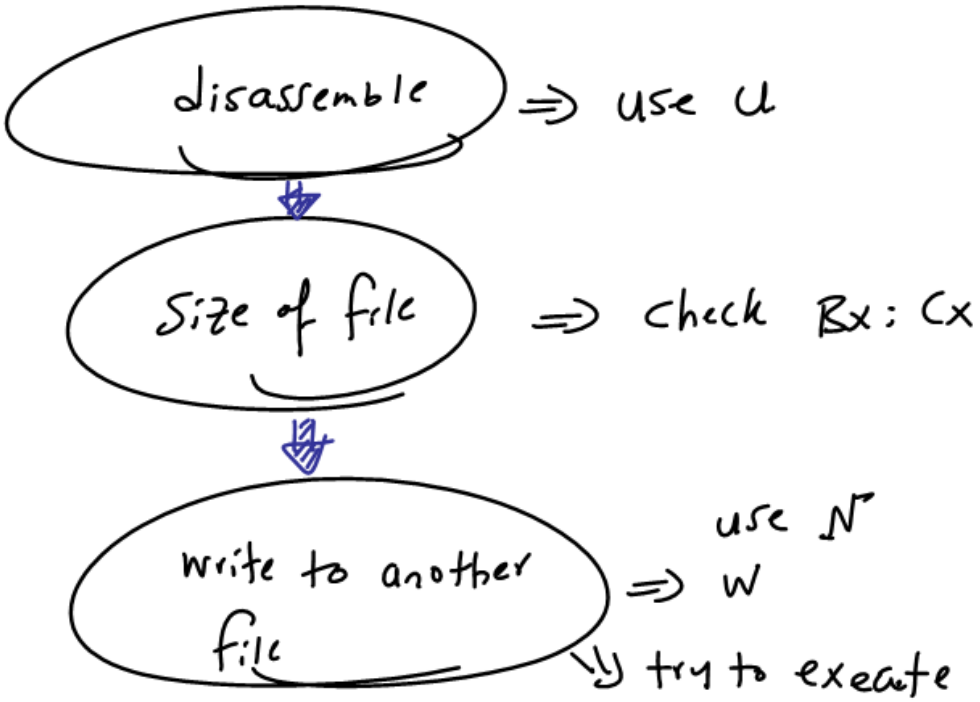
  `W 0`

  Writing from location CS:0 to the file

  `W DS:0200`

  Write named file from location DS:0200






```
-n d:printme.exe
-l
-u
1417:0000 B81914          MOV     AX,1419
1417:0003 8ED8            MOV     DS,AX
1417:0005 B300            MOV     BL,00
1417:0007 B500            MOV     CH,00
1417:0009 B10A            MOV     CL,0A
1417:000B 8AD3            MOV     DL,BL
1417:000D 80C230          ADD     DL,30
1417:0010 B402            MOV     AH,02
1417:0012 CD21            INT     21
1417:0014 FEC3            INC     BL
1417:0016 E2F3            LOOP    000B
1417:0018 B44C            MOV     AH,4C
1417:001A CD21            INT     21
1417:001C CC              INT     3
1417:001D 0000            ADD     [BX+SI],AL
1417:001F 0000            ADD     [BX+SI],AL
```

```
-r cx
CX 001C
:0021
-w 0000
EXE and HEX files cannot be written
-n d:test.com
-w 0000
Writing 00021 bytes
-q

D:\>test
0123456789
```

✍ **Note:**

- **DEBUG doesn't know what kind of file is**. The file could as well be a **program file** as a **text file**; DEBUG makes no assumptions based on the file's contents or its file extension.
- **DEBUG examines memory at the current address and displays it as though it were a machine instruction**. If memory contains data instead of machine instructions, the disassembly line should be ignored, even though it will always be there.
- *This is once again an example of the problems you can have in assembly language if you don't know exactly what you're doing*. ***Code and data look the same in memory***.
- They are **only different in how you interpret them**.
- **Example**: the hex value **53H** is the letter **S**; in an executable program file, **53H** would be the machine instruction **PUSH BX**.

- When DEBUG loads a file from disk, it places the number of bytes in the file in the **CX** register. **CX** is a general-purpose register, but

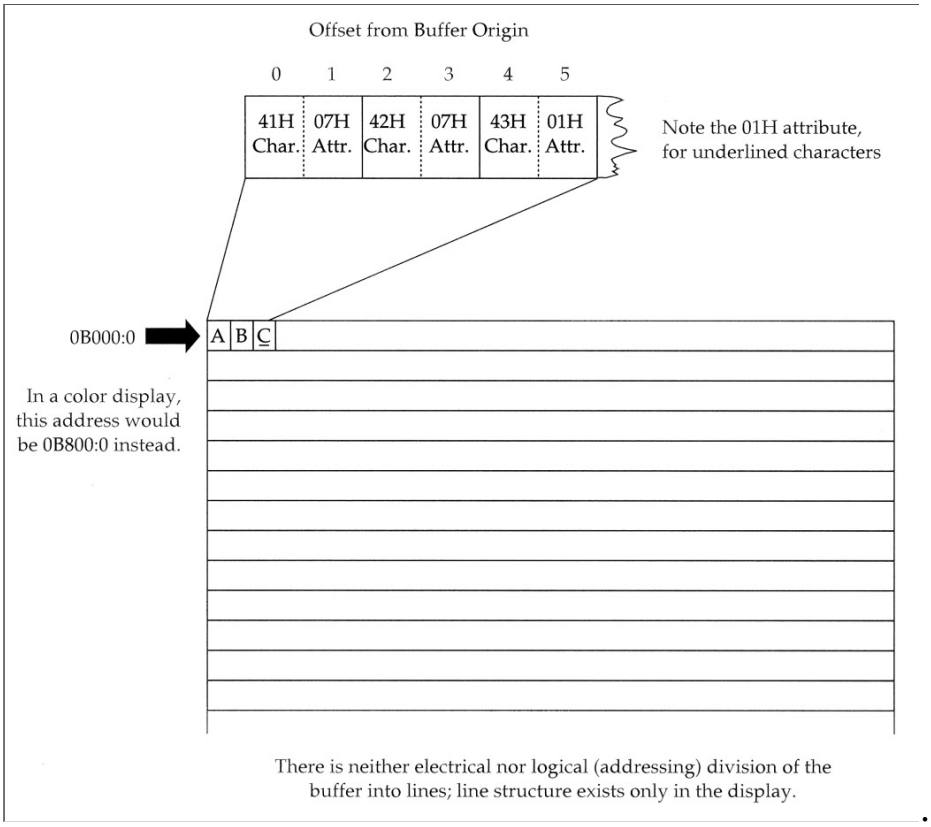
it is often used to contain such count values and is therefore sometimes called the count register.

Inspecting the Video Refresh Buffer with DEBUG

- One easy thing to do is look at the **PC's video display adapter's** text screen video refresh buffer. A video refresh buffer is a region of memory with a difference: *Any characters written to buffer memory are instantly displayed on the computer's screen*. This is accomplished electrically through special use of the information that comes out of the memory data pins. Precisely how it is done is outside the scope of this book.
    - **For now**, simply understand that writing a character to your text mode display screen (which is not the Windows GUI screen!) can be done *by writing the ASCII code for that character into the correct address in the video refresh buffer portion of memory.*
- The text mode display buffer is the screen that appears when you're running DOS or else working in a DOS window (or "DOS box") from within MS Windows. It consists not of icons or graphical images or figures but simple textual characters, arranged in a matrix typically 25 high and 80 wide.
- As with any memory location anywhere within the PC, the video refresh buffer has a segment address. What that segment address is depends on the kind of display installed in the PC. There are two separate possibilities, and which is present is easy enough to determine: If your PC has a **color screen**, the segment address of the video refresh buffer is **0B800H**. If you

have a **monochrome screen** (a situation now becoming vanishingly rare), the segment address is **0B000H** instead.

- It takes 2 bytes in the buffer to display a character. The first of the two (that is, first in memory) is the ASCII code of the character itself. For example, an **A** would require the ASCII code **41H**; a **B** would require the ASCII code **42H**, and so on. The second of the two bytes is the **character's attribute**. Think of it this way: In the display of a character on the screen, *the ASCII code says what and the attribute says how*. The attribute dictates the color of a character and its background cell on a color screen. On a monochrome screen, the attribute specifies whether a character is underlined or displayed in reverse video.



- The very first character/attribute pair in the video refresh buffer corresponds to the character you see in the upper-left-

hand corner of the text screen. The next character/attribute pair in the buffer is the character on the second position on the top line of the screen, and so on

- In the Figure above, the three letters "ABC" are displayed in the upper-left corner of the screen. Notice that the "C" is underlined. The screen shown in the Figure a monochrome screen. The video refresh buffer therefore begins at **0B000:0**. The byte located at address **0B000:0** is ASCII code 41H, corresponding to the letter "A." The byte at address 0B00:0001 is the corresponding attribute value of 07H. The 07H value as an attribute dictates normal text in both color and monochrome displays, in which normal means white characters on a black background.
- The byte at 0B000:0005 is also an attribute byte, but its value is 01H. On a monochrome screen, 01H makes the corresponding character underlined. On a color display, 01H makes the character blue on a black background.
- There is nothing about the video refresh buffer to divide it into the lines you see on the display. The first 160 characters (80 ASCII codes plus their 80 attribute bytes) are shown as the first line, and the subsequent 160 characters are shown on the next line down the screen.
- You might rightfully ask what ASCII code is in the video refresh buffer for locations on the screen that show no character at all. The answer, of course, is that there is a character there in every empty space: the space character, whose ASCII code is 20H.

You can inspect the memory within the video refresh buffer directly, through DEBUG. Take the following steps:

- ✓ Clear the screen by entering **CLS** at the **DOS** prompt and pressing Enter.
- ✓ Invoke **DEBUG**.
- ✓ Decide where your video refresh buffer is located, and enter the proper segment address into the **ES** register through the **R** command. Remember: *Color screens use the 0B800H segment address, while monochrome screens use the 0B000H segment* address. Note from the following session dump that **0B800H** must be entered into DEBUG as "**B800**," without the leading zero. NASM (your assembler) must have that leading zero, and DEBUG cannot have it. Sadly, no one ever said that all parts of this business had to make perfect sense.
- ✓ Dump the first 128 bytes of the video refresh buffer by entering **D ES:0** and pressing Enter.
- ✓ Dump the next 128 bytes of the video refresh buffer simply by entering the **D** command by itself a second time.

What you'll see should look a lot like the following session dump:

```
C:\ASM>debug
- r es
ES 1980
:b800
- d es:0
B800:0000 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0010 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0020 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0030 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0040 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0050 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0060 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0070 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
- d
B800:0080 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:0090 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:00A0 43 07 3A 07 5C 07 41 07-53 07 4D 07 3E 07 64 07 C.:.\.A.S.M.>.d.
B800:00B0 65 07 62 07 75 07 67 07-20 07 20 07 20 07 20 07 e.b.u.g. . . . .
B800:00C0 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:00D0 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:00E0 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
B800:00F0 20 07 20 07 20 07 20 07-20 07 20 07 20 07 20 07  . . . . . .
```

The first 80 character/attribute pairs are the same: **20H/07H**, which display as plain, ordinary blank space. When you execute the CLS command on most machines, the screen is cleared, and the DOS prompt reappears on the second line from the top of the screen, not the top line. The top line is typically left blank, as is the case here.

You'll see in the second block of 128 dumped bytes the *DOS prompt and the invocation of DEBUG in lowercase*. Keep in mind when reading DEBUG hex dumps that any character not readily displayed as one of the standard ASCII letters, numbers, or punctuation marks is represented as a period character. This is why the 07H attribute character is shown on the right portion of DEBUG's display as a period character, since the ASCII code 07H has no displayable equivalent.

You can keep dumping further into the video refresh buffer by pressing **DEBUG's D** command repeatedly.

Reading the Basic Input / Output System Revision Date

Another interesting item that's easy to locate in your PC is the revision date in the **ROM BIOS**. ROM (read-only memory) chips are special memory chips that retain their contents when power to the PC is

turned off. *The BIOS (Basic Input / Output System) is a collection of assembly language routines that perform basic services for the PC: disk handling, video handling, printer handling, and so forth*. The BIOS is kept in ROM at the very top of the PC's megabyte of address space.

The BIOS contains a date, indicating when it was declared finished by its authors. This date is always at the same address and can be easily displayed using **DEBUG's D** command. The address of the date is **0FFFF:0005**. The DEBUG session is shown in the following listing. Note again that the hex number 0FFFFH must be entered without its leading zero:

```
- d ffff:0005
FFFF:0000                30 34 2F-33 30 2F 39 37 00 FC B8    04/30/97...
FFFF:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
FFFF:0080 00 00 00 00 00                                   .....
```

✍ Note:

- **Microsoft codeview** is an advanced option for DEBUG
- **Borland Turbo Debugger**

■ INTERRUPT

- Most microprocessors allow normal program execution to be interrupted by some external signal or by a special instruction in the program
- When a microprocessor is interrupted, it stops executing its current program and calls a procedure which "services" the interrupt
- At the end of the interrupt service procedure, execution is usually returned to the interrupted program
- An 8086 interrupt can come from any one of three sources
    - One source is from an external signal applied to the **nonmaskable interrupt** (**NMI**) input pin, or the interrupt (**INTR**) input pin

      An interrupt caused by a signal applied to one of these inputs is referred to as **hardware interrupt**
    - A second source of an interrupt is execution of the **interrupt instruction INT**. This is referred to as a **software interrupt**
    - The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction.
        - An example of this is the divide by zero interrupt. Program execution will automatically be interrupted if you attempt to divide an operand by zero
        - Conditional interrupts such as this are also referred to as software interrupts
- At the end of each instruction cycle the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions

(1) It decrements the stack pointer by two and pushes the flag register on the stack

(2) It disables the INTR interrupt input by clearing the interrupt flag in the flag register

(3) It resets the trap flag in the flag register

(4) It decrements the stack pointer by two and pushes the current code segment register contents on the stack

(5) It decrements the stack pointer again by two and pushes the current instruction pointer contents on the stack

(6) It does an indirect far jump to the start of the procedure you wrote to respond to the interrupt

To summarize these steps, the 8086 pushes the flag register on the stack, disables the single step and the INTR input, and does essentially an indirect far call to the interrupt service procedure

MAINLINE PROGRAM

⇩

⇩

⇩

PUSH FLAGS

CLEAR IF

CLEAR TF

PUSH CS

PUSH IP

FETCH ISR ADDRESS

INTERRUPT SERVICE PROCEDURE

PUSH REGISTERS

⇩

⇩

POP REGISTERS

IRET

POP IP

POP CS

POP FLAGS

⇩

⇩

⇩

✍ **Note:**

- An **IRET** instruction at the end of the interrupt service procedure returns execution to the main program

✍ **Note:**

- When the 8086 does a far call to a procedure, it puts a new value in the code segment register and a new value in the instruction pointer
- For an indirect call the 8086 gets the new values for CS and IP from your memory addresses
- Likewise, when it responds to an interrupt the 8086 goes to memory locations to get the CS and IP values for the start of the interrupt service procedure

- ✓ In the 8086 Microprocessor, the **first 1 Kbyte** of memory from 00000H to 003FFH is set aside as a **table** for storing the **starting addresses of interrupt service procedures**
- ✓ Since **4 bytes** are required to store the **CS and IP** values for each interrupt service procedures, the table can hold the starting addresses for up to **256 interrupt procedures**
- ✓ The **starting address** of an interrupt service procedure stored in this table is often called the **interrupt vector or the interrupt pointer**, and the table itself is then referred to as the **interrupt vector table** or the **interrupt pointer table**

The 8086 instruction which allows this to happen is the INT instruction (INT is short for INTerrupt). Altogether 256 different sorts of interruption are allowed, so it is necessary to specify which one is required.

The interrupt instruction which allows operations such as reading a character from the keyboard, printing a character on the display screen and handling files on disk is number **21H**

Thus, executing the instruction INT 21H gives you access to a whole number of input and output functions. Specifying which function, you want involves putting a code number in register AH

Function number	Description	What it does
1	Keyboard input	Wait until a character is typed at the keyboard and then put the ASCII code for that character in register AL. whatever is typed is also printed on the display screen
2	Output on the display screen	Print on the display screen the character whose ASCII code is contained in register DL

----	------	------
5	Print character	The character in DL is sent to the printer
7	Keyboard input/ no display	Wait for an input character from the keyboard, and when it arrives places it in AL. the character is not automatically printed on the display screen – CTRL-BRK cannot be used to return to PC-DOS
8	Keyboard input/ no display	As in function 7 except that normal CTRL-BRK service is provided
9	Display string	Print a whole series of characters stored in memory starting with the one in address given in DX (relative to DS). Stop when a memory location containing the ASCII code for a $ sign is encountered. Don't print the $ sign
A	Keyboard string	A character string is read into store beginning at the address given in **DS: DX**. The first byte typed by the user specifies the length of the string. The actual characters of the string are now read in and stored in the third and following bytes of the designated group of locations until either the ENTER key is pressed or one less than the stated number of locations has been filled If the ENTER key is pressed, then the second byte of the store area is set to the number of characters read (excluding CR). The last byte of the string is set to CR. Extra characters are ignored and the bell is rung if an overflow condition is about to occur

Example — print Char A

```
MOV AH,02H
MOV DL,41H
INT 21H
```

Example — Subroutine to print Char

```
PRINTCHAR:    MOV AH,02H
              INT 21H
```

- **Printing new lines**

When using a typewriter, starting a new line involves two operations: returning the carriage which holds the paper to the beginning of the line (carriage return) and moving the paper up one line (line feed).

The display of text on a computer's screen has been designed to emulate this, so taking a new line involves 'printing' the ASCII codes for the line feed and carriage return.

Since we already have a subroutine printing a single character on the screen, we can use that to create a PRINT_NEWLINE subroutine

Example — Subroutine to print newline

```
PRINT_NEWLINECHAR:    MOV DL,0AH; a carriage return
                      CALL PRINTCHAR
                      MOV DL,0DH; a line feed
                      CALL PRINTCHAR
                      RET
```

✍ **Note:**

- As with a typewriter, the operations of carriage return and line feed can be done in either order, but there may be problems with some makes of printer if the carriage return is not done first