# *The 80x86 family instruction set*

1. **Data transfer** : `MOV, XCHG, LEA, ...`
2. **Arithmetic** : `ADD, DEC, NEG, ...`
3. **Logic** : `AND, NOT, OR, XOR`
4. **Shift and rotate** : `RCL, RCR, ROL, ROR,...`
5. **Bit test and bit scan** : `BT, BTC, ...`
6. **Flag operations** : `CLC, CLD, CLI, STI, ...`
7. **Compare and set** : `CMP`
8. **Jump** : `JMP, JA, JAE, JB, JNZ,...`
9. **Stack** : `PUSH, POP, PUSHA,POPA,...`
10. **Subroutines** : `CALL, RET`
11. **Loop** : `LOOP, LOOPE, LOOPZ, ...`
12. **String** : `MOVSB, REP, CMPSP, ...`
13. **Interrupt** : INT 21h, ...

15. **I/O** : IN, OUT
16. **Miscellaneous** : NOP, ...

## *Example 3.1: **INC, CMP, JNE** instructions*

```
                DOSSEG
                .MODEL SMALL
                .CODE
                mov   ah,2          ; Load ah with 2
                mov   dl,20h        ; load dl with 20h
ASCIIloop:      int   21h           ; print character function
                inc   dl            ; increment dl
                cmp   dl,7fh        ; compare dl with 7fh
                jne   ASCIIloop     ; if not equal jump to
ASCIIloop

                mov   ah,4ch
                int   21h           ; Terminate program function
                END
```

***Example:***

```
; This program converts lowercase characters to
; uppercase before printing them.

                DOSSEG
                .MODEL SMALL
                .CODE
start:          mov  ah,8
                int  21h
                cmp al,'a'
                jl donothing
                cmp al,'z'
                jg donothing
                sub al,20h
donothing:      mov dl,al
                mov ah,2
                int 21h
                jmp  start
                END
```

| ASCII | CHAR |
|---|---|
| . | |
| . | |
| . | |
| 30 | 0 |
| 31 | 1 |
| 32 | 2 |
| 33 | 3 |
| 34 | 4 |
| 35 | 5 |
| 36 | 6 |
| 37 | 7 |
| 38 | 8 |
| 39 | 9 |
| . | |
| . | |
| . | |
| 41 | A |
| 42 | B |
| 43 | C |
| 44 | D |
| 45 | E |
| 46 | F |
| . | |
| . | |
| 61 | a |
| 62 | b |
| 63 | c |
| 64 | d |
| 65 | e |
| 66 | f |
| . | |
| . | |

## ***Comments about ASCII code***

To convert from upper case to lower case add 20h

To convert from lower case to upper case subtract 20h

To convert a hexadecimal digit (x) to its ASCII code (y):

$$y = \begin{cases} x+30h & \text{if } 0<x<9 \\ x+37h & \text{if } A<x<F \\ \text{ERROR} & \text{else} \end{cases}$$

To recover a hexadecimal digit (x) from its ASCII code (y):

$$x = \begin{cases} y-30h & \text{if } '0'<y<'9' \\ y-37h & \text{if } 'A'<y<'F' \\ y-57h & \text{if } 'a'<y<'f' \\ \text{ERROR} & \text{else} \end{cases}$$

# *Registers of the 8086 µp*

| | | | |
|---|---|---|---|
| Flag reg. | F | 16-bit registers | |
| | AX | ah \| al | Accumulator |
| | BX | | Pointer (DS) |
| General | CX | | Counter |
| purpose | DX | | I/O |
| registers | SI | | Source Index Pointer(DS) |
| | DI | | Dest. Index Pointer (DS,ES) |
| | BP | | Pointer (SS) |
| | SP | | Stack Pointer |
| Inst. poin. | IP | | Instruction pointer |
| | CS | | Code Segment |
| Segment | DS | | Data Segment |
| registers | ES | | Extra Segment |
| | SS | | Stack Segment |

**Flags Register:** is never directly modified or read.

| | | | | O | D | I | T | S | Z | | A | | P | | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

C : Carry flag
P : Parity flag (Parity of lower 8 bits of last result generated)
A : Auxiliary carry flag ( BCD operations)
Z : Zero flag
S : Sign flag
T : Trap flag ( debugging software)
I : Interrupt flag
D : Direction flag (direction of string instructions)
O : Overflow flag

## AX Register: (Accumulator)

⊗ Most efficient register used in arithmetic, logic, and data movement operations
⊗ Always involved in multiplication and division
⊗ Lower and upper parts can be accessed as Al and AH, respectively

```
mov  ah,0        ; load ah with 0
mov  al,ah       ; copy ah to al
inc  al          ; increment al   (al ← 1)
```

## BX Register: Used in referencing memory locations

```
mov al,[bx]      ;  means load  al  with the content of  the memory location
                 ;  whose effective address is in bx
```

Can also be treated as bl, and bh

## CX Register: Used in counting

```
      mov cx,10
start:  .
        .
        .
      sub  cx,1   ; subtract 1 from cx
      jnz  start  ; if result is not 0, jump to start
```

```
      mov  cx,10
start:  .
        .
        .
      loop start
```

Again, lower and upper parts are cl and ch respectively.

## DX Register: I/O addressing.

```
mov  al,62
mov  dx,1000
out  dx,al       ;  output the content of al to the output port whose
                 ;  address is in dx
```

**SI, DI, and BP Registers:** Memory pointers ( to be explained later).

**SP Registers:** Stack pointer (to be explained later).

**IP Register:** Instruction pointer or program counter. It contains the address of the next instruction to be executed.

**CS, DS, ES, and SS Registers:** Segment registers

**Problem: The 8086 is capable of addressing 1 MB of memory. Therefore, 20-bit addresses are required, however, the 8086 only uses 16-bit pointers. How, then, does the 8086 reconcile 16-bit pointers with 20-bit address space?**

**Solution**: By using ***Memory Segmentation***

**Physical Address = segment address*16 + offset**

**PA = SA*16 + EA**

⊗ **The offset is also called *Effective Address* (The name seams to be confusing !)**

⊗ **The Segment Beginning (SB) = segment address*16**

⊗ **Multiplying by 16 means shifting to left 4-bits.**

⊗ **Physical addresses also may be denoted as SA:[EA]**

***Ex:*** **Calculate the physical address if segment address=3AFF and offset=0002**

```
SA   3AFF        SB    3AFF0
EA   0002        EA  + 0002
                 PA    3AFF2
```

***Ex:*** **Calculate the physical address denoted by 5AFB:[0EC7] " SA:[EA]"**

```
SA   5AFB        SB    5AFB0
EA   0EC7        EA  + 0EC7
                 PA    5BE77
```

***Ex:*** **Calculate an offset and segment address if physical address is 6DE2A.**

```
SA        6DE2
EA        000A
```

***I.e*** **6DE2A may be denoted by 6DE2:[000A]**

# Pointers of the 8086

BX
SI,
DI
BP

SP
IP

## Valid Instructions

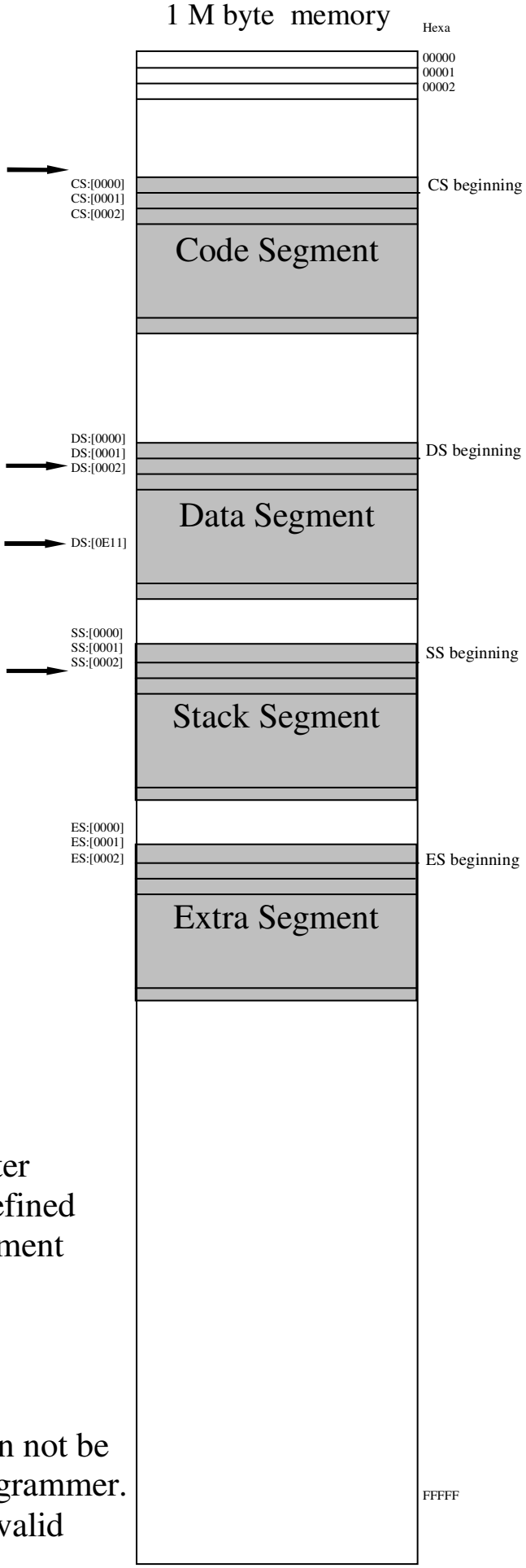
```
mov  AL, DS:[2]
mov  AH, ES:[BX ]
mov  CL, CS:[SI]
mov  Bl, DS:[BP]
mov  Bl, SS:[BP ]

mov  Bl, [ BP]   ; SS is default
mov  Dl, [BX]    ; DS is default
mov  Cl, [SI]    ; DS is default
mov  Al, [DI]    ; DS is default
```

***With string instructions, DI points to ES.* (To be explained later)**

## Invalid Instructions

```
mov AL, DS:[CX]     ;  cx is not a pointer
mov BL, [00F2h]     ;  segment is undefined
mov AH, DX:[BX]     ;  DX is not a segment
mov  Bl, [ SP]
mov  Bl, DS:[SP]
mov  AL,[ IP]
```

Although SP and IP are pointers they can not be used for memory referencing by the programmer. This is why the last 3 instructions are invalid



# *Writing your 1st 80x86 Assembly language program*

step 1:Get into your text editor and type in the following lines that make up the source code of program HELLO.ASM:

```
        DOSSEG
        .MODEL SMALL
        .DATA
msg DB   'Hello, EE411',13,10,'$'
        .CODE
        mov ax,@DATA
        mov ds,ax
        mov ah,9
        mov dx, offset msg          ;lea  dx , msg
        int    21h
        mov ah,4Ch
        int    21h
        END
```

step 2:Assemble your source code program to generate the object module. HELLO.OBJ:

```
TASM  hello.asm
```

step 3:Link the object file to generate the executable program HELLO.EXE:

```
TLINK   hello.obj
```

step 4:Execute the program by writing hello at the DOS prompt. You will get the message:

```
Hello, EE411
```

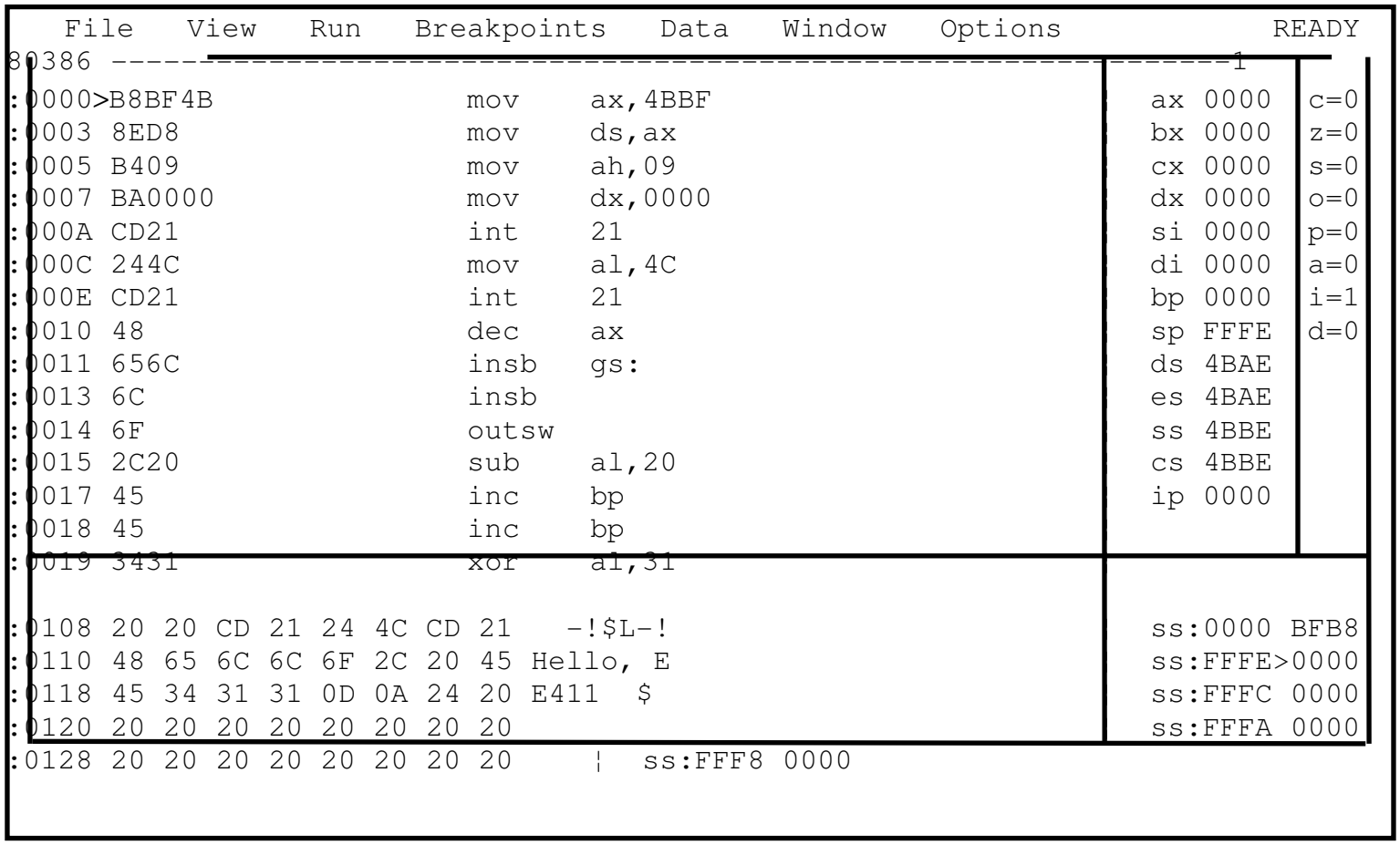
***Notes:***

*When you assembled HELLO.ASM, Turbo Assembler turned the text instructions in HELLO.ASM into their binary equivalents in the object file HELLO.OBJ which is an intermediate file partway between source code and an executable file.*

*When you linked HELLO.OBJ, TLINK converted it into the executable file HELLO.EXE and also generated a redundant file called HELLO.MAP which contains information about the memory usage.*

# *Debugging your programs using TD.EXE*

```
        DOSSEG
        .MODEL SMALL
        .DATA
    msg DB  'Hello, EE411',13,10,'$'
        .CODE
        mov     ax,@DATA
        mov ds,ax
        mov ah,9
        mov dx,offset msg
        int 21h
        mov ah,4Ch
        int 21h
        END
```



# *Machine language and Assembly language:*

Remember that:

**Computer:** is a machine that can solve problems by carrying instructions given to it.

**Program:** is a sequence of instructions describing how to perform a certain task.

**Machine language:** is the computer primitive instructions consisting of 0's and 1's.

This program prints ASCII characters 32-127 onto screen !

```
CS:0000    B4
CS:0001    02
CS:0002    B2
CS:0003    20
CS:0004    CD
CS:0005    21
CS:0006    FE
CS:0007    C2
CS:0008    80
CS:0009    FA
CS:000A    7F
CS:000B    75
CS:000C    F7
```

It is almost impossible to write programs directly in machine directly in machine language. However, they may be written in a human readable form of the machine language which is called the ***Assembly language***

```
CS:0000    B4      MOV AH,2
CS:0001    02
CS:0002    B2      MOV DL,20H
CS:0003    20
CS:0004    CD      INT 21H
CS:0005    21
CS:0006    FE      INC DL
CS:0007    C2
CS:0008    80      CMP DL,7FH
CS:0009    FA
CS:000A    7F
CS:000B    75      JNE 4
CS:000C    F7
```

The 80x86 µp family has about 150 machine language instructions each one is referred by 2-5 letters mnemonics in the assembly language. The mnemonics may be followed by one or more operands.

```
B055        MOV AL,55H
FA          CLI
AC          LODSB
74XX        JZ  XX
```

## Each instruction is divided into fields:

⊗ *Operation code (op code):* indicating what the processor is to do.
⊗ *Operands:* indicating the information needed by the instruction (datum or its location)

The way in which an operand is specified is called its addressing mode. There are 3 basic addressing modes:

## *Addressing modes (AMs):*

**1. Immediate operand AM** `mov ax,5`

**2. Register Operand AM** `mov ax,bx`

**3. Memory operand AMs**

**Direct AM** `mov ax,BETA`
**Register indirect AM** `mov ax,DS:[BX]`
**Based AM**
**Indexed AM**
**Based Indexed AM**

*less frequently used (to be explained later)*

# *DOS interrupt 21h:* ***(services for basic I/O programming)***

**Service 01h:** DOS get character function

```
mov ah,01h          ; returns ASCII code of character to AL
int 21h             ; and echo it to the monitor
```

**Service 02h:** DOS print character function

```
mov ah,02h
mov dl,ASCII#       ; ASCII code of character for print in DL
int 21h
```

**Service 08h:** Get character without echo

```
mov ah,08h          ; returns ASCII code of character to AL
int 21h             ; but don't echo it to the monitor
```

**Service 09h:** DOS print string function

```
mov ah,09h
mov dx,offset       ; the effective address of the massage is in
DX
int 21h             ; the string should terminate with a $ sign.
```

**Service 4Ch:** DOS terminate program function.

```
mov ah,4Ch          ; leave the control to DOS
int 21h
```

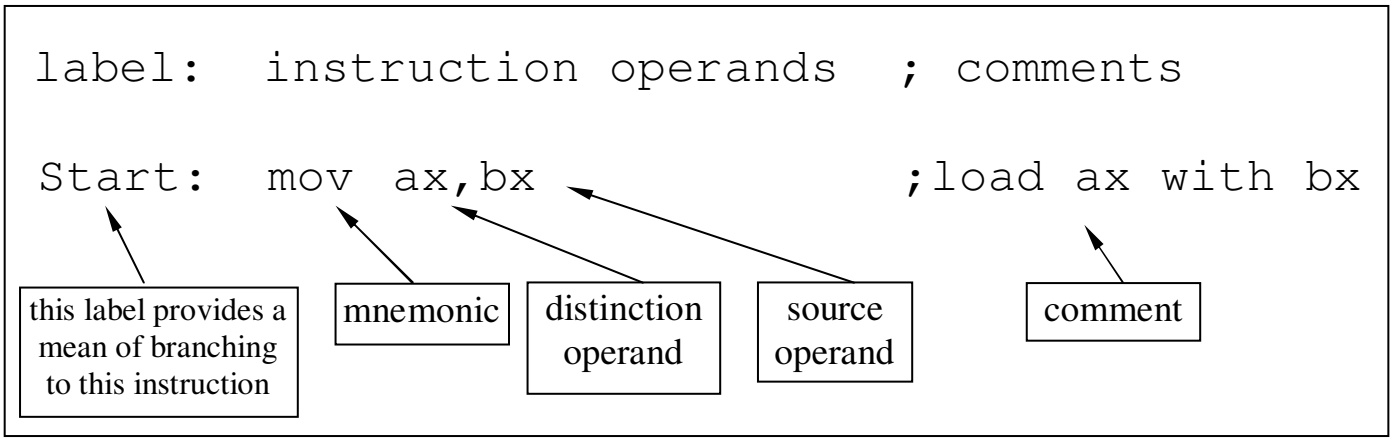
*(Note that the service number is always specified in ah)*

# *Notes about TASM*

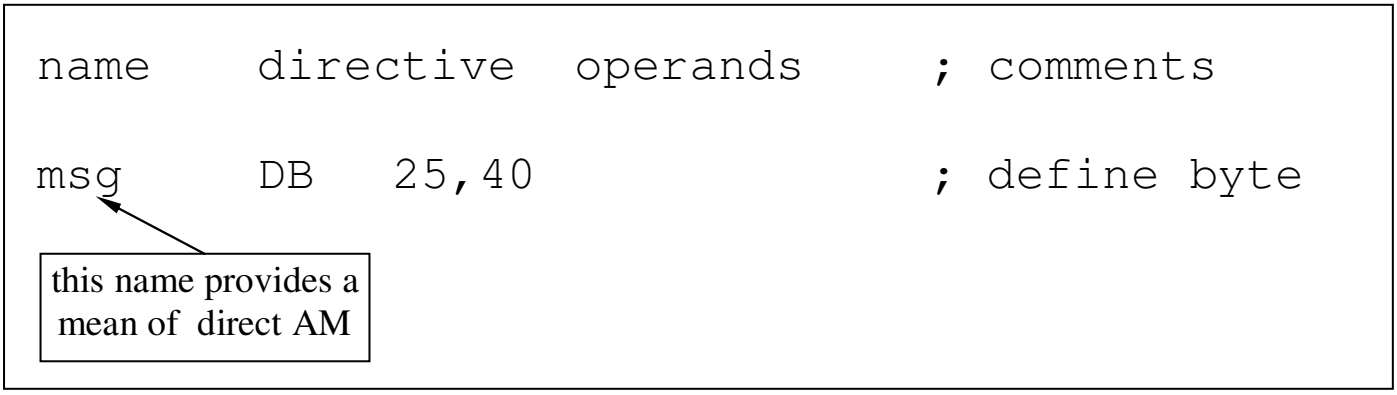
- There are 2 types of statements in an assembly language:

  1. **Instructions**, which are translated into machine instructions by the assembler.
  2. **Directives**, which give directions to the assembler during the assembly process

- **The general format of an assembler instruction**




- **The general format of an assembler directive**



- The fields in the assembler statements are separated by 1 or more spaces.
- Avoid using reserved words (AX, INT,...) for identifiers (labels & names).
  - TASM is insensitive to letter case (mov = MOV)
  - Hexadecimal numbers are denoted by the suffix h
- Binary numbers are denoted by the suffix b
- Decimal numbers are denoted by the suffix d (default is decimal)
- String constants are enclosed in single quotes (‘)
- The first digit in a hexa number must be 0 through 9.

Ex: Discus the validity of the following instructions

```
mov ax,12A3h      ; OK

mov ax,1237d      ; OK

mov ax,1237       ; equivalent to above instruction

mov ax,10110b     ; OK

mov ax,‘$’        ; moves the ASCII code of $ to ax

mov ax,24h        ; equivalent to above instruction

mov ax,10110      ; OK
```

```
    mov ax,E2A8h  ; wrong

    mov ax,0E2A8h ; OK

    mov ax,12030b ; wrong " binary 0,1"

    mov ax,12030h ; wrong, the number is too
large 5 digit "max 4 digit";
    int 21            ; common mistake!, should be
21h
                      ; if int 15h is meant then OK!
```

## *Frequently used TASM directives:*

- **END** informs the assembler that the source code is finished and he has to stop assembling.

- **.CODE** indicates the beginning of the code segment

- **.DATA** indicates the beginning of the data segment

  ⊗ DS has to be explicitly loaded

```
mov AX,@Data
mov DS,AX
```

  ⊗ CS, SS, are set by DOS

- **DOSSEG** tells the assembler to adopt the DOS segment sequence ; code, data, and stack.

- **.MODEL** defines a memory model:

  **tiny**, Both code and data are in the same 64kB segment (used for .com programs)

  **small**, All data in 1 segment and all code in 1 segment

- **.STACK** defines the size of the stack

- **DB**, **DW**, **DD**, and **DUP** are used to allocate memory locations

  **DB** (Define Byte) : Each operand occupies 1 byte.

  **DW** (Define Word) : Each operand occupies 2 bytes (a word). The low significant byte is followed by the msb

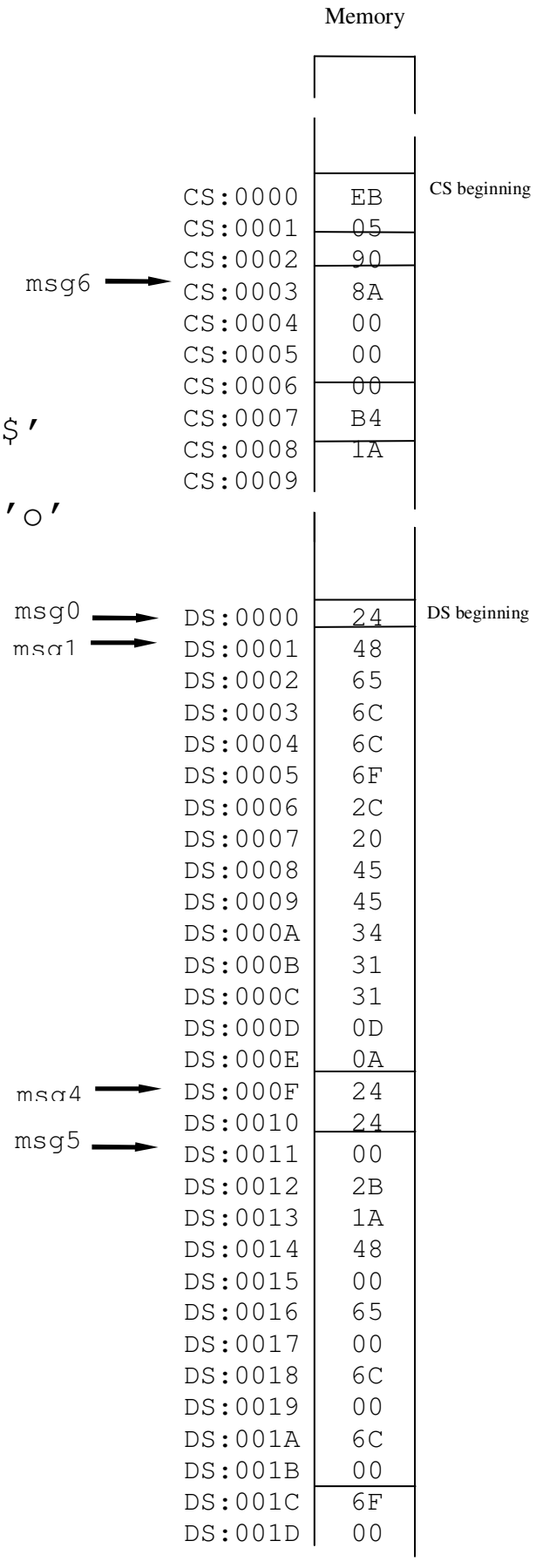
  **DD** (Define Double Word) : Each operand occupies 4 bytes (double word). The low significant word is followed by the msw

Ex: Bellow are some examples of memory alocation directives

```
msg0   DB 24h  ; inetialize a memory location with the data 24h
               ; the PA of that location is denoted by msg0

msg1   DB 'Hello, EE411',13,10, '$'
               ; reserve 15 consecutive memory locations and
               ; Inetialize it with the data 48, 65, 6C, 6C, 6F, 2C,
20
               ; 45, 45, 34, 31, 31, 0D, 0A, 24 and assign the PA of
               ; The first location to msg1.

msg2   DW 'Hello'
               ; wrong, only DB can be used for strings of any
length.

msg3   DB  ?   ; reserve an uninetialized memory location and
               ; assign its PA to msg3.

msg4   DW 24h  ; inetialize 2 memory locations with data 24h (lsb) and
               ; 00(msb).The PA of the first byte is assigned to msg4.

msg5   DW 1A2Bh, 'H','e','l','l','o'
               ; inetialize 12 memory locations with the hexa data:
               ; 2B,1A,48,00,65,00,6C,00,6C,00,6F,00. The
               ; PA of the first byte is assigned to msg5.

msg6   DD 8Ah  ; inetialize 4 memory locations with the hexa data
               ; 8A 00 00. The PA of the first byte is msg6.

array DB 100 DUP(?)
               ; (duplicated) reserve 100 uninetialized bytes.
               ; The PA of the first byte is array
```

# *Modelling the memory after loading a program*

```
        DOSSEG
        .MODEL small
        .DATA
msg0    DB 24h
msg1    DB 'Hello, EE411',13,10,'$'
msg4    DW 24h
msg5    DW 1A2Bh,'H','e','l','l','o'
        .CODE
        jmp   start
msg6    DD 8Ah
start:  mov  ah,1Ah
          .
          .
          .
        mov ah,4ch
        int 21h
        END
```



# *Revision of Addressing modes*

```
        DOSSEG
        .MODEL SMALL
        .DATA
n1      dw    5555h
n2      db    88h
n3      dd 11h
        .CODE
        mov ax,@data
        mov ds,ax
        jmp start
n4      dw 7777h
start:  mov al,77h                  ;  immediate operand AM
        mov di,offset start         ;  immediate operand AM
        mov si,offset n4            ;  immediate operand AM
        mov cx,offset n1            ;  immediate operand AM
        mov ds,@data                ;  wrong(1)
        mov ax,@data                ;  immediate operand AM

        mov ds,ax                   ;  register operand AM
        mov bx,cx                   ;  register operand AM

        mov ax,n1                   ;  memory operand AM, direct
        mov n4,bx                   ;  memory operand AM, direct
        mov ax,n2                   ;  wrong(2)
        mov al,n2                   ;  memory operand AM, direct
        mov ax,start                ;  wrong(2)
        mov ax,word ptr start       ;  memory operand AM, direct(3)
        mov al,byte ptr start       ;  memory operand AM, direct
        mov al,n1                   ;  wrong(2)
        mov al,byte ptr n1          ;  memory operand AM, direct

        mov ax,DS:[offset n1]       ;  memory operand AM, direct
        mov CS:[offset n4],bx       ;  memory operand AM, direct
        mov al,DS:[offset n2]       ;  memory operand AM, direct
        mov ax,cs:[offset start]    ;  memory operand AM, direct
        mov al,cs:[offset start]    ;  memory operand AM, direct
        mov al,DS:[offset n1]       ;  memory operand AM, direct
        mov al,DS:[byte ptr n1]     ;  memory operand AM, direct
        mov al,DS:[word ptr n1]     ;  wrong(2)

        mov ax,[bx]                 ;  memory operand AM, indirect
        mov ax,[si]                 ;  memory operand AM, indirect
```

```
        mov ax,[di]                 ;  memory operand AM, indirect
        mov ah,4ch
        int 21h
        END
```

(1) Segment registers can not be immediately loaded.
(2) Operand types do not match.
(3) The `ptr` operator specifies the length of a quantity in ambiguous situations.

# *A program example*

```
; This program illustrates the use of mov and xchg
; instructions.

        DOSSEG
        .MODEL SMALL
        .DATA
n1      dw   5555h
n2      dw   7777h
        .CODE
        mov  ax,@data
        mov  ds,ax

        mov  ax,n1          ; This block exchanges the
        mov  bx,n2          ; contents of n1, and n2
        mov  n2,ax
        mov  n1,bx

        mov  ax,n1          ; xchg n1,n2 is illegal
        xchg ax,n2
        mov  n1,ax

        mov  ah,4ch
        int  21h
```

Notes

- Store a sample assembly program on disk, and for each new program that you want to create, COPY the sample program into a file with its correct name, and use your editor to complete the additional instructions.
- AS you will repeat the steps (`TASM fn.asm, TLINK fn.obj, fn`) frequently, it is recommended to prepare a patch file in your disk, call it `asm.bat`, in which you write

```
TASM  %1
TLINK %1
del *.map
del *.obj
%1
```

- A header is to be written at the beginning of your programs, which contains name, surname, ID no, and the name (ore description) of the program.
- Programs must be well committed. It is not necessary to write a comment for each line but any block having a particular function must be thoroughly explained.
    - Programs must be indented as much as the longest identifier.