Chapter 04

# Addressing modes

## Table of content

## Introduction

- The different ways that a processor can access data are referred to as its **addressing mode —** how the CPU can access operands (data)
- In assembly language statements the addressing mode is indicated in the instruction
- The number of addressing modes is determined when the microprocessor is designed and cannot be changed
- For example, the ability to access successive memory locations with a sequence of instructions such as

```
MOV AX,[SI]
    ▪
    ▪
    ▪
MOV AX,[SI+1]
    ▪
    ▪
MOV AX,[SI+2]
```

  was provided specifically to assist in the setting up and manipulation of arrays. Accessing memory by using SI to provide an index in this way is one of seven addressing modes available from 8086

- Efficient software development for the MP requires a complete familiarity with the addressing modes employed by each instruction
- The study of these addressing mode in relation to the problem of implementing the machine code generation phase of a compiler for a high-level language such as PASCAL is presented.
- The 80x86 provides a total of seven distinct addressing modes

- **Register**
- **Immediate**
- **Direct**
- **Register indirect**
- Based relative
- Indexed relative
- Based indexed relative

- The 8086 has eight general purpose 16-bit registers, eight general purpose 8-bit registers, four segment registers and the IP register (which the programmer cannot access directly)

- Register addressing mode

  - Both destination and source operands reside in registers
  - Memory is not accessed
  - It is relatively fast — instructions which affect registers only (for instance, MOV CX,DX) execute faster than instructions involving memory access as no time is spent fetching operands
  - Consequently, a good Pascal compiler will attempt to ensure that data items required in repeatedly executed sections of a program are held in registers whenever possible

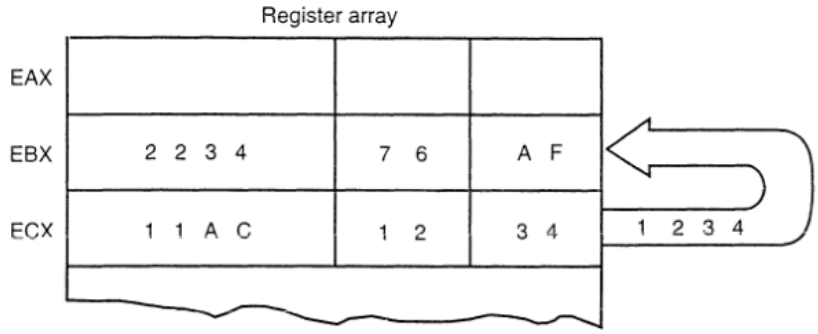
    Thus, if **a** and **b** are identifiers corresponding to integer variables, by the time it comes to executing:

```
a:=4; b:=3
for i:=1 to 10000 do
     begin
     write(i,a,b,a*b)
     a:= a+2;
     b:= b+1
```

```
end
```

a Pascal compiler would have already assigned memory locations to **i**, **a**, and **b** but would try to arrange that the values of **i**, **a**, and **b** are moved into registers for the duration of the FOR-loop execution

- **Example**




The effect of executing the MOV BX,CX

| | | |
|---|---|---|
| MOV AL,BL | 8-bits | Copies BL into AL |
| MOV CH,CL | 8-bits | Copies CL into CH |
| MOV AX,CX | 16-bits | Copies CX into AX |
| MOV SP,BP | 16-bits | Copies BP into SP |
| MOV DS,AX | 16-bits | Copies AX into DS |
| MOV SI,DI | 16-bits | Copies DI into SI |
| MOV BX,ES | 16-bits | Copies ES into BX |

- Source and destination must match in size

  MOV CL, AX ➔ will give an error
- CS is not allowed to be destination
- Segment-to-segment is not allowed

- Immediate addressing mode

  - In the immediate addressing mode, the **source operand is a constant**. In immediate addressing mode, as the name implies, when the instruction is assembled, **the operand comes immediately after the opcode**. For this reason, this addressing mode executes quickly. However, in programming it has limited use.

    **`MOV CX, 2346H`** ➔ copy into CX the 16-bit data 2346H

    **`SUB AL,24H`** ➔ AL = AL-24H

    ```
    MOV AX,2550H      ;move 2550H into AX
    MOV CX,625        ;load the decimal value 625 into CX
    MOV BL, 40H       ;load 40H into BL
    ```

  - Immediate addressing mode can be used to load information into any of the registers except the segment registers and flag registers.
  - For example, **MOV CL,61H** – the immediate operand 61H is stored as part of the instruction in memory. When that instruction comes to be executed, the immediate operand is thus fetched from memory at the same time as the instruction, once again reducing execution time — this is similar to the way the **port address was put in memory immediately after the code** for the input instruction in the three-instruction program.
  - Not to store information directly into segment register and flag register

    To move information to the segment registers, the data must first be moved to a general-purpose register and then to the segment register. Examples

**MOV AX,2550H**

**MOV DS,AX**

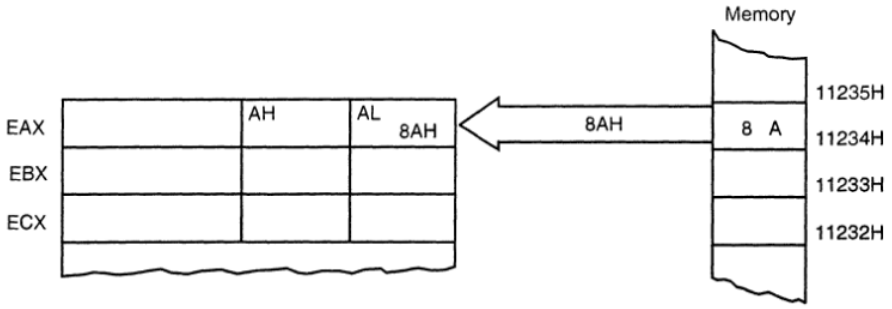
In other words, the following would produce an error

**MOV DS,0123H ; illegal**

- Direct addressing mode

  - In the direct addressing mode, the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction.
  - Note that in immediate addressing, the operand itself is provided with the instruction, whereas in direct addressing mode, the address of the operand is provided with the instruction. The address is the offset address and one can calculate the physical address by shifting left the DS register and adding it to the offset.

**MOV DL, [2400] ; move contents of DS:2400H into DL**




The operation of the MOV AL,[1234H] instruction when DS = 1000H

- **Example.**

```
OR AL,[3030H]
```

ORing the contents of AL register with the contents of memory location calculated from DS register and offset 3030H

AL ← (DS × 10 + 3030H) OR AL

If AL contains 70H before instruction execution and DS contains 3745H, then the memory location of operand

$= DS \times 10H + 3030H$

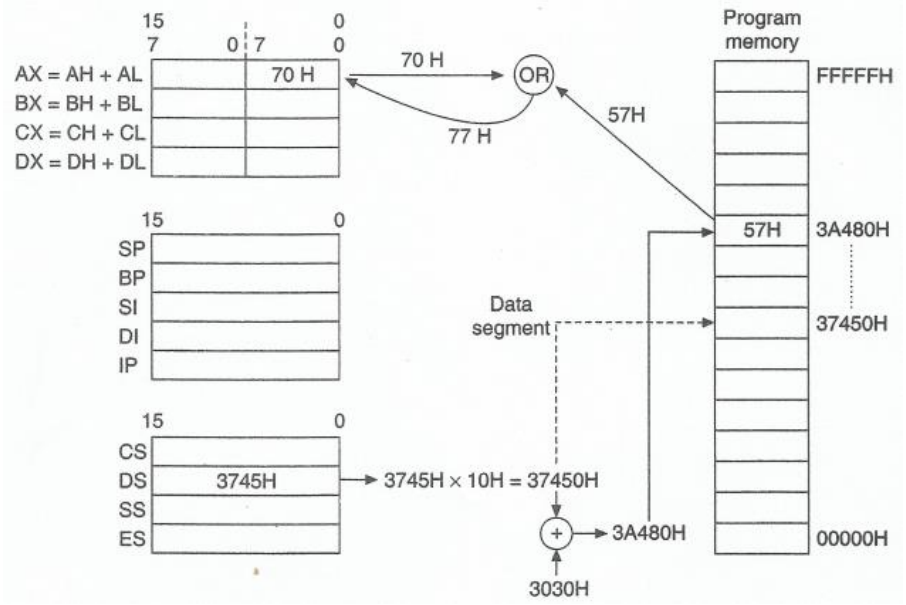
$= 37450H + 3030H$

$= 3A480H$

If the memory location 3A480H has 57H as contents, then

AL ← 57H OR 70H

AL← 77H




- **Example.**

  Find the physical address of the memory location and its contents after the execution of the following, assuming that DS = 1512H

**MOV AL,99H**

**MOV [3518], AL**

**Solution**

First AL is initialized to 99H, then in line two, the contents of AL are moved to logical address DS:3518 which is 1512:3518. Shifting DS left and adding it to the offset gives the physical address of 18638H (15120H + 3518H = 18638H). That means after the execution of the second instruction, the memory location with address 18638H will contain the value 99H

- **Example.**

  MOV AX, TOTAL

  Where for example, TOTAL is an assembly language variable corresponding to location 210H (relative to DS), the machine code version of the instruction is **`A1 1002`**

  so, that TOTAL is addressed directly by the instruction since 1002 is the memory representation of 0210H

  only one operand of an instruction may be directly addressed, so that if **`total`** and **`item_sum`** are identifiers of two Pascal integer variables which we assume are represented in signed 16-bit form then, for optimal execution speed not involving arithmetic in registers, the Pascal

  ```
  total := total + item_sum
  ```

  could be coded by a compiler using two directly addressed instructions:

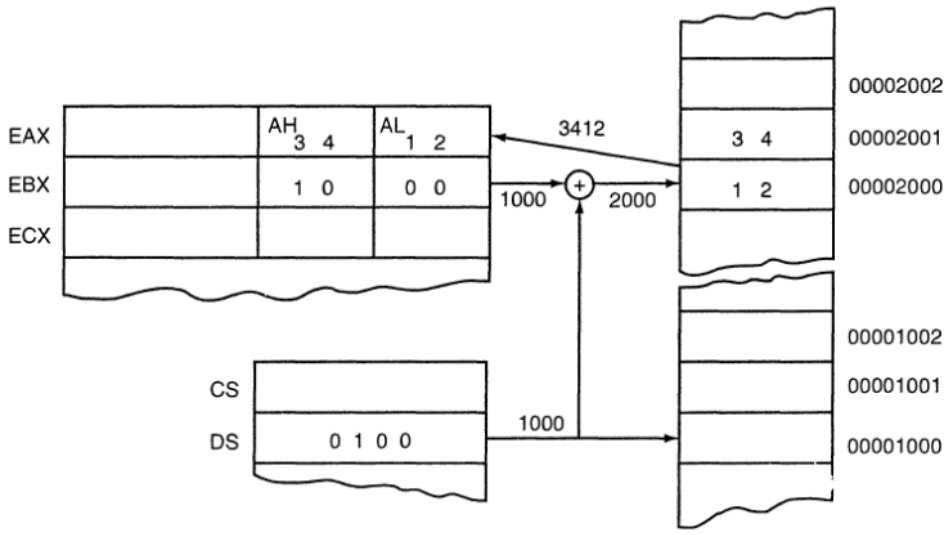
  ```
  MOV AX, ITEM_SUM
  ```

ADD TOTAL, AX

## ■ Indirect addressing mode

- In the register-indirect addressing mode, the address of the memory location where the operand resides is held by a register.
- The registers used for this purpose are **SI, DI, and BX**. If these three registers are used as pointers, that is, if they hold the offset of the memory location, they must be combined with DS in order to generate the 20-bit physical address.
- For example:

  **MOV AL, [BX]** **;** moves into AL the contents of the memory location pointed to by DS:BX




- **Problem**

  Assume that **DS = 1120**, **SI = 2498**, and **AX = 17FE**. Show the contents of memory locations after the execution of

  **MOV [SI], AX**

- In a Pascal compiler, indirect addressing can be useful for matching formal parameters and actual parameters. For example, given declarations

```
proc example (var f:integer);
begin
:
:
end;
var a:integer
```

the assembly language or machine code equivalent of **example** produced by a compiler cannot use any specific address corresponding to the variable **f** since this will not be known until a call such as **example(a)** occurs and the actual parameter (**a**) corresponding to the formal parameter (**f**) is known

if the address of the actual parameter is put into BX and indirect addressing is used to refer to **f** in the compiler's implementation of the body of the example, the problem is solved

- Stack

  - The stack is a section of read/write memory (RAM) used by the CPU to store information temporarily. The CPU needs this storage area since there are only a limited number of registers. There must be some place for the CPU to store information safely and temporarily. The main disadvantage of the stack is its access time. The registers are inside the CPU and RAM is outside.
  - If the stack is a section of RAM, there must be registers inside the CPU to point to it. The two main registers used to access the stack are the **SS** (stack segment) register and the **SP** (stack pointer). These registers must be loaded before any instructions accessing the stack are used.
  - Every register inside the 80x86 (except segment registers and SP) can be stored in the stack and brought back into the CPU from the stack memory.
  - The storing of a CPU register in the stack is called a **push**, and loading the contents of the stack into the CPU register is called a **POP**.
  - Typically, the programmer's stack would start at a location near the end of available memory and would grow backwards in memory as demand warranted

    PUSH 16-bit register name ➔ **`PUSH AX`**

    POP 16-bit register name ➔ **`POP DX`**

    PUSH flags ➔ **`PUSHF`**

    Retrieve flags ➔ **`POPF`**
  - In order that PUSH and POP can operate successfully it is necessary only to specify how big a memory stack is required

and to set the 16-bit SP (stack pointer) register to the address of the top of the stack (which is where items are added and removed — compare with the dinner plate dispenser)

- SS, and SP are initialized automatically by TASM as

  ```
  .stack 64
  ```

  Which instruct the linker to set SS to a value compatible with the memory and to initialize SP to the top of the stack
- Whenever a word of data is pushed onto the stack the high-order 8-bits are placed in the location addressed by SP-1. The low-order 8-bits are placed in the location addressed by SP-2. The SP is then decremented by 2 so the next word of data is stored in the next available stack memory location.
- **Pushing onto the stack**

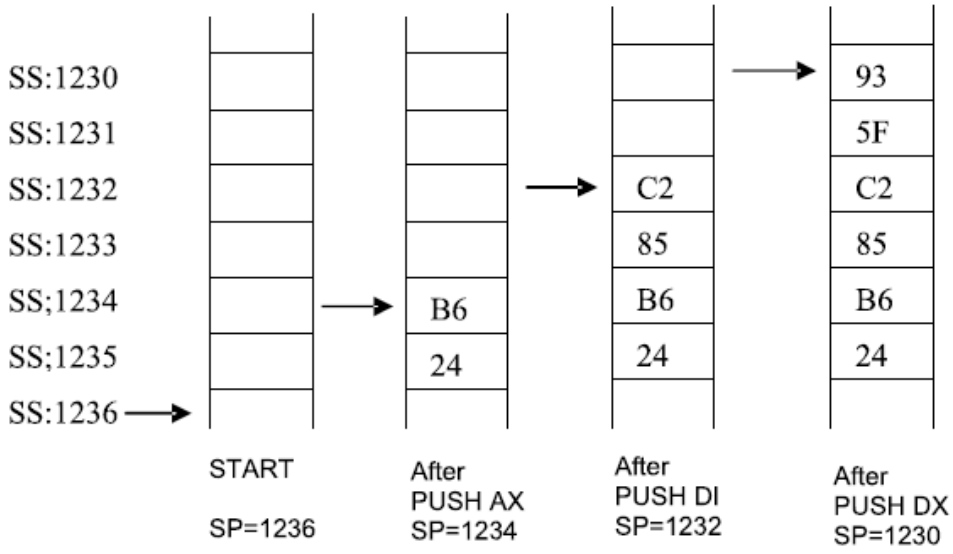
  Storing the CPU register in the stack is called a push.

  **Ex:**

  SP=1236, AX=24B6, DI=85C2, and DX=5F93, show the contents of the stack as each instruction is executed:

  ```
  PUSH AX
  PUSH DI
  PUSH DX
  ```

  Note that in 80x86 the lower byte of the register is stored to the lower address




- **Popping the stack**

  Loading the contents of the stack into the CPU register is called a pop.

  **Example:**

  Assume that the stack is shown below, and SP=18FA, show the contents of the stack and registers as each of the following instructions is executed. **`POP CX POP DX POP BX`**