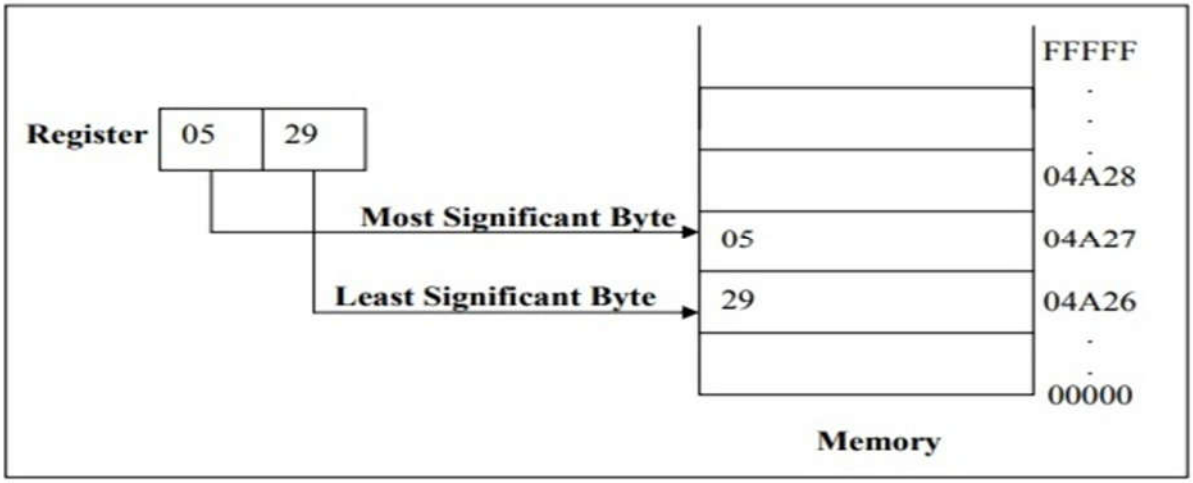
# Chapter Two Addressing Data Memory

## Addressing Data Memory

Depending on the model, the processor can access one or more bytes of memory at a time. Consider the Hexa value **(0529H)** which requires two bytes or one word of memory. It consist of high order **(most significant)** byte 05 and a low order **(least significant)** byte 29.

The processor store the data in memory in reverse byte sequence i.e. the low order byte in the low memory address and the high order byte in the high memory address. For example, the processor transfer the value **0529H** from a register into memory address **04A26 H** and **04A27H** like this:




The processor expects numeric data in memory to be in reverse byte sequence and processes the data accordingly, again reverses the bytes, restoring them to correctly in the register as hexa **0529H**.
When programming in assembly language, you have to distinguish between the address of a memory location and its contents. In the above example the content of address **04A26H** is 29, and the content of address **04A27H** is 05.

**There are two types of addressing schemes:**

1. **An Absolute Address**, such as **04A26H**, is a 20 bit value that directly references a specific location.

2. **A Segment Offset Address**, combines the starting address of a segment with an offset value.

## Segments and Addressing

Segments are special area defined in a program for containing the code, the data, and the stack. **Segment Offset** within **a program**, all memory locations within a segment are relative to the segment starting address. The distance in bytes from the segment address to another location within the segment is expressed as an **offset** (or displacement).

To reference any memory location in a segment, the processor combine the segment address in a segment register with the offset value of that location, that is, its distance in byte from the start of the segment.

## Specifying addresses

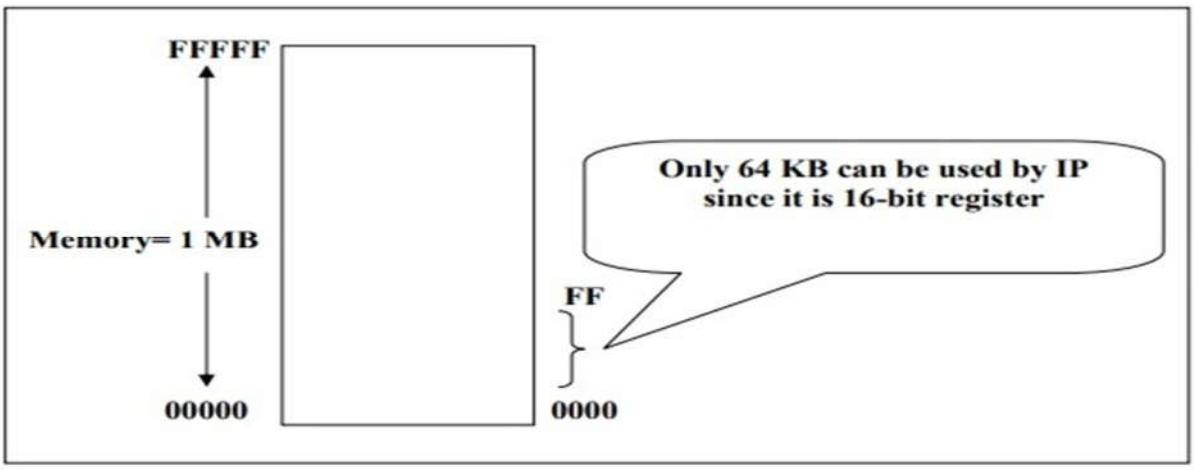
To represent a segment address and its relative offset we use the notation:

**Segment: offset**

Thus **020A:1BCD** denotes offset **1BCDH** from segment **020AH**. The actual address it refers to is obtained in the following way:

- Add zero to the right hand side of the segment address.
- Add to this the offset.

Hence the actual address referred to by 020A:1BCD is 03C6D.




Address Bus in the **8086** is 20 bits wide (20 lines) i.e. the processor can access memory of size 220 or 1048576 bytes **(1MB).**

Instruction Pointer = 16 bit register which means the processor can only address 0 – 216 (65535) bytes of memory. But we need to write instructions in any of the 1MB of memory. This can be solved by using memory segmentation., where each segment register is 16-bit (this 16-bit is the high 16-bit of Address Bus (A4- A19)) i.e. each of the segment registers represent the actual address after shifting the address 4-bit to get 20 bits.

## Registers

Registers are 8, 16, or 32-bit high speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

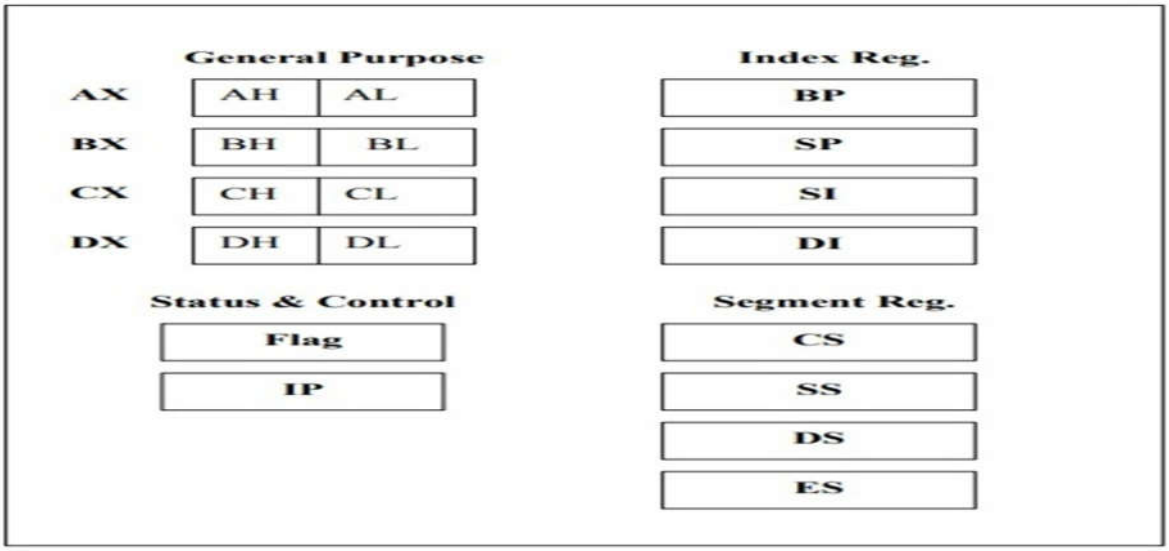



**Figure 7: Intel 16-bit registers**

The CPU has an internal data bus that is generally twice as wide as its external data bus.

**Data Registers**: The general purpose registers, are used for arithmetic and data movement. Each register can be addressed as either 16-bit or 8 bit value. Example, **AX** register is a 16-bit register, its upper 8-bit is called **AH**, and its lower 8-bit is called AL. Bit 0 in AL corresponds to bit 0 in **AX** and bit 0 in AH corresponds to bit 8 in AX. See Figure 8.

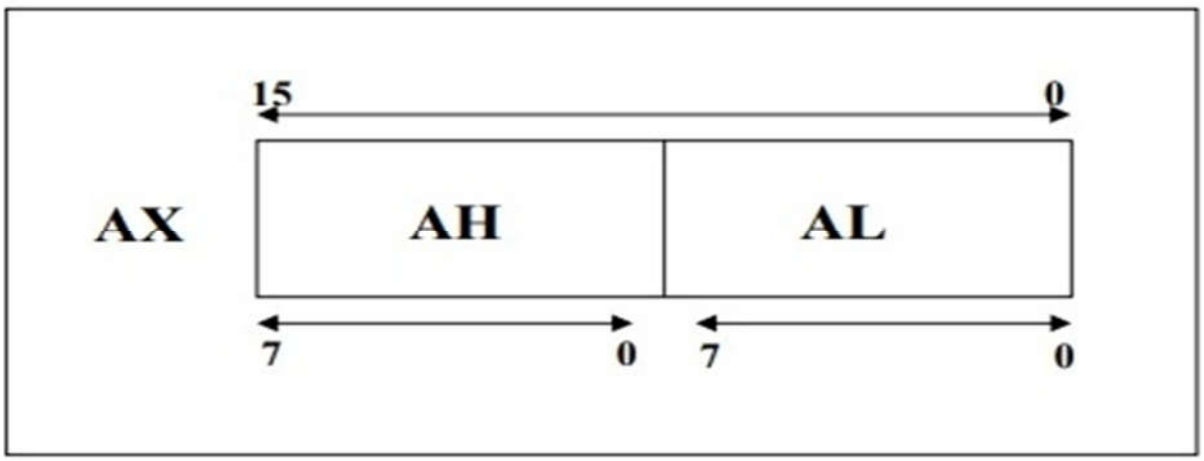



**Figure 8: AX register**

Instructions can address either 16-bit data register as AX, BX, CX, and DX or 8-bit register as AL, AH, BL, BH, CL, CH, Dl, and DH. If we move 126FH to AX then AL would immediately 6FH and AH = 12H.

## General Purpose Register

Each general purpose register has special attributes:

- **AX (Accumulator):** AX is the accumulator register because it is favored by the CPU for arithmetic operations. Other operations are also slightly more efficient when performed using **AX**.
- **BX (Base):** the BX register can hold the address of a procedure or variable. Three other registers with this ability are **SI**, **DI** and **BP**. The **BX** register can also perform arithmetic and data movement.
- **CX (Counter):** the CX register acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement **CX**
- **DX (Data):** the DX register has a special role in multiply and divide operation. When multiplying for example **DX** hold the high 16 bit of the product.

## Segment Registers

**Segment Registers:** the CPU contain four segment registers, used as base location for program instruction, and for the stack.

- **CS (Code Segment):** The code segment register holds the base location of all executable instructions (code) in a program.

- **DS (Data Segment):** the data segment register is the default base location for variables. The CPU calculates their location using the segment value in DS.
- **SS (Stack Segment):** the stack segment register contain the base location of the stack.
- **ES (Extra Segment):** The extra segment register is an additional base location for memory variables.

## Index Registers

**Index registers** contain the offset of data and instructions. The term offset refers to the distance of a variable, label, or instruction from its base segment. The index registers are:

- ✓ **BP (Base Pointer):** the **BP** register contain an assumed offset from the stack segment register, as does the stack pointer. The base pointer register is often used by a subroutine to locate variables that were passed on the stack by a calling program.
- ✓ **SP (Stack Pointer):** the stack pointer register contain the offset of the top of the stack. The stack pointer and the stack segment register combine to form the complete address of the top of the stack.
- ✓ **SI (Source Index):** This register takes its name from the string movement instruction, in which the source string is pointed to by the source index register.
- ✓ **DI (Destination Index):** the **DI** register acts as the destination for string movement instruction