**8086 Memory Addressing Modes**

Table of Contents

[Segmentation Models 2](#_Toc83226898)

[Defining the Data Segment 3](#_Toc83226899)

[Arrays 4](#_Toc83226900)

[Addressing Modes 5](#_Toc83226901)

[Addressing Data 5](#_Toc83226902)

[Addressing Program Codes in Memory 9](#_Toc83226903)

[Addressing Stack in Memory 11](#_Toc83226904)

[Addressing Input and Output Ports 11](#_Toc83226905)

[Implied Addressing 12](#_Toc83226906)

[Memory Banks 13](#_Toc83226907)

## Segmentation Models

There are actually three **models** of segmentation, which we will be looking into in depth later on:

1. **.model small** – This model is the most widely used. It requires that 64KB be allocated for code and another 64KB be allocated for data.
2. **.model medium** – This model requires data to be less than 64KB but allows code to exceed that limit.
3. **.model compact** – This model requires code to be less than 64KB but allows data to exceed that limit.

Notice that the medium and compact models are opposites.

We can apply a model by specifying the model we want to use at the top of our assembly language code.

.MODEL SMALL  
.STACK 100H ; Override default stack segment size (1KB) to 100h (512 bytes)

.DATA

; Variables declared here occupy memory from the data segment.  
  
.CODE

; The code here occupies memory from the code segment.  
MAIN PROC  
 MOV AX, @DATA  
 MOV DS, AX

; Any Push or Pop operations in the code occupy memory in the stack segment.  
MAIN ENDP  
 END MAIN  
RET

ASSEMBLY

## Defining the Data Segment

The DB keyword is used to define a **byte**. The DW keyword is used to define a **word** (2 bytes).

The EQU keyword is used to define a **numeric constant** to a name.

.data  
avariable DB 100 *; Define a data with 1-byte*astring DB "Hello" *; define 5 consecutive bytes with ASCII values*maxint EQU 35535 *; define maxint = 35535*

ASSEMBLY

## Arrays

The following is an **array** declaration:

.DATA  
arr DB 01h, 0Ah, 02h, 0Bh, 03h, 0Ch

ASSEMBLY

In the **data segment** of the memory, six **consecutive bytes** are occupied by the six values in this array.

## Addressing Modes

The different ways in which a microprocessor can **access data** are called **addressing modes**. For Intel 8086, the addressing modes are:

1. Addressing Data
2. Addressing Program Code in Memory
3. Addressing Stack in Memory
4. Addressing I/O
5. Implied Addressing

### Addressing Data

The first category, **Addressing Data**, refers to **user-defined data**. When addressing user defined data, we can take several approaches.

#### Immediate Addressing

In **Immediate Addressing**, the data is immediately given in the instruction itself.

MOV BL, 44

ASSEMBLY

The above code is an example of Immediate Addressing, since we are providing the data itself as one of the arguments of the MOV instruction.

#### Direct Addressing

In **Direct Addressing**, the **address** of the data is provided in the instruction itself.

MOV AX, [1234h]

ASSEMBLY

In the above code, we provided an address (in square brackets) to a memory location. This is actually the **offset number**. The microprocessor will use this along with the **segment number** stored in the **DS register** to calculate the complete memory location. The actual data is stored at that location.

One thing to note is, since the AX register is a 16-bit register with two halves, the AL register and the AH register, **2 consecutive bytes** of memory starting at the provided location will be copied into the register. This means that the data in the 1234h offset will be stored in the AL register and the data in the 1235h offset will be stored in the AH register. If we had used just AL in the instruction, only 1 byte would be copied.

Alternatively, we could do this:

ORG 0100h  
  
.DATA  
DATA DW AB12h  
  
.CODE  
 MAIN PROC  
 MOV BX, DATA  
 MAIN ENDP  
END MAIN  
RET

ASSEMBLY

In the above code, DATA is a **variable** of 16 bits that holds a particular offset value. Later, we copy the value of this variable into the BX register. However, this works a little differently than the previous method for Direct Addressing. BX does not hold data from 2 consecutive memory locations in its two halves. Instead, the BH register holds the **first part** of the **actual offset** value, AB, while the BL register holds the **second part** of the **actual offset** value, 12.

#### Register [Direct] Addressing

In **Register Direct Addressing**, the data from **one register** is copied to **another register**.

MOV AX, BX

ASSEMBLY

The **sizes** of both registers must be the **same**.

#### Register [Indirect] Addressing

In **Register Indirect Addressing**, one register holds the **offset number** for a particular memory location. This offset number is used along with the value of the **DS register** to calculate the complete memory location address.

MOV CX, BX

ASSEMBLY

Similar to Direct Addressing, **2 bytes** worth of data starting at the specified memory location will be copied into the two halves of the CX register.

#### Base-Plus-Index Addressing

In **Base-Plus-Index Addressing**, we use **2 registers** to access specific **elements** within **arrays**.

To identify the actual **array address**, we can use either the **BX** or the **BP** base register. This is just the **first element** of the array. To identify an **index position** within the array, we can use the **DI** or the **SI** index register.

MOV BX, 1234h  
MOV DI, 3  
MOV DX, [BX + DI]

ASSEMBLY

In the above code, we get the data from the third index position (fourth element) of the array and store it in the DX register. Of course, DX is a 16-bit register, so technically the fourth and fifth elements will both be stored.

#### Register Relative Addressing

In **Register Relative Addressing**, we do everything in exactly the same way as we did with **Base-Plus-Index Addressing**. The only difference is we **do not store the index** position in an index register. Instead, we provide it **directly** alongside the base address.

MOV BX, 1234h  
MOV DX, [BX + 3]

ASSEMBLY

Again, two consecutive bytes, the fourth and fifth locations, are being copied to DX.

#### Base-Relative-Plus-Index Addressing

In **Base-Relative-Plus-Index Addressing**, we combine **Base-Plus-Index Addressing** with **Register Relative Addressing**.

MOV BX, 1234h  
MOV DI, 3  
MOV DX, [BX + DI + 2]

ASSEMBLY

### Addressing Program Codes in Memory

**Program code** is stored in the **code segment**. To access this code, we use **jump** and **CALL** instructions.

**Jump** instructions, also called far jump instructions, are used to actually move to a specified position within a program. The position is marked using a **label**.

*; some code*JMP some\_label  
*; this code is skipped over*some\_label: *; code for label processed  
; program resumes here*

ASSEMBLY

Alternatively, we can also specify a **segment number** and an **offset number** to specify a location in the code segment when using the jump instruction, i.e. JMP 1023:0027.

**CALL** instructions, also called far call instructions, are used with **procedures**.

*; some code*CALL some\_procedure  
*; program resumes here once procedure ends*some\_procedure PROC  
 *; some code*END some\_procedure

ASSEMBLY

Alternatively, we can also specify a **segment number** and an **offset number** to specify a location in the code segment when using the CALL instruction, i.e. CALL 1023:0027.

We can use one of three addressing modes:

1. **Direct Addressing** – This is when we use a **segment number** and an **offset number** with the jump and CALL instructions, as shown above. Direct addressing is very troublesome to use.
2. **Indirect Addressing** – This is when we use **labels** or **procedure names** with jump and CALL instructions respectively, as shown above. Indirect addressing is far easier to use.
3. **Relative Addressing** – This is when we use a **register** to store the memory location to which we want to go using a jump or CALL instructions, as in JMP [DI] or CALL [BX]. The register will hold the **offset number** and the **segment number** will be obtained from the **CS register**. We can use general purpose registers (AX, BX, CX, DX), relative registers (BP, BX, DI, SI) or a combination of a relative register and a displacement.

### Addressing Stack in Memory

The **PUSH** and **POP** instructions can be used to move data to and from the **stack segment**.

PUSH AX *; PUSH the data in the AX register to the stack*

PUSH BXPOP AX *; POP data from the stack into the AX register*

POP BX

ASSEMBLY

The above operations cause the contents of AX and BX to be swapped. This is an alternative to the XCHG AX, BX operation. That instruction does the same thing.

The **CALL** instruction also uses the stack to hold the **return address** of the procedure being called.

### Addressing Input and Output Ports

The IN and OUT instructions are used to address **I/O ports**. They can use either direct or indirect addressing.

IN AL, 05h *; direct addressing; 05h is the port number*OUT DL, AL *; indirect addressing; DL contains the port number*

ASSEMBLY

Note that only the **DL** register can be used to hold the address of an I/O port in indirect addressing.

### Implied Addressing

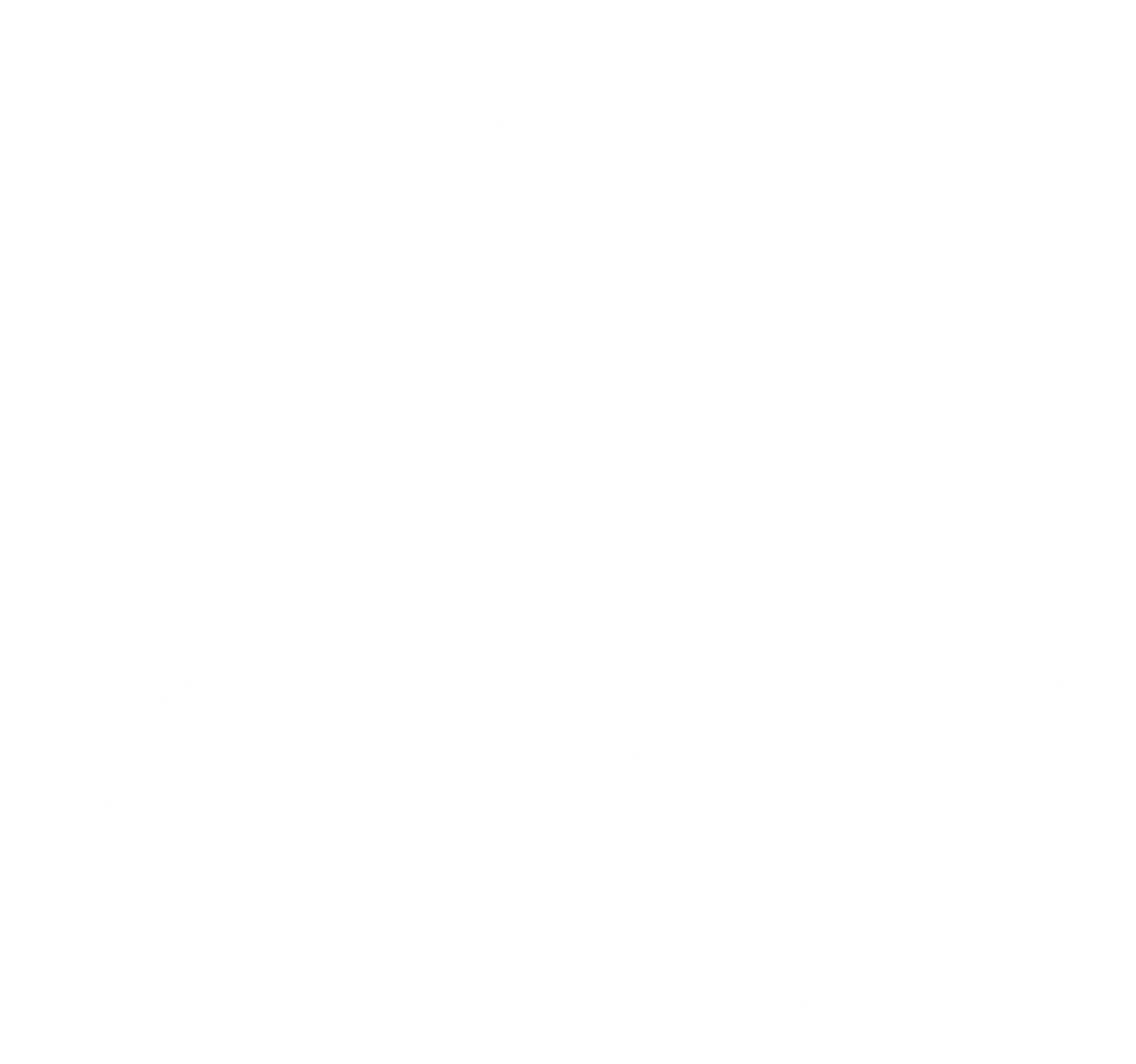
For some instructions, we do not even have to explicitly provide an address. The address required is **implied** within the instruction itself.

CLC *; clear carry flag*HLT *; halts the program*RET *; return control to OS  
; equivalent to*MOV AH, 4Ch *; funciton number 4Ch*INT 21h *; interrupt*

ASSEMBLY

## Memory Banks

The system’s **memory** can be stored in two **memory banks**, an **Odd Bank** and an **Even Bank**.



The content of **one row** can be accessed in **one cycle**. Thus, for example, if we wanted to access 00000h and 00001h, we could do so in one cycle. However, if we wanted to access 00001h and 00002h, we would need two cycles.

Because of this, if we store 16-bits of data in a memory location starting with an **even address**, we can read it in **one cycle**, but if it stored in a memory location starting with an **odd address**, we will need **two cycles**. For 8-bits of data, either case will require just one cycle.