MOV Instruction in 8086 Microprocessor

The **MOV instruction** in the 8086 microprocessor is primarily used for data transfer between registers, between a register and memory, or between memory and a register. It loads data from a source operand into a destination operand, following the syntax: `MOV destination, source`. For example, `MOV AX, BX` moves the contents of BX into AX.

Important Characteristics:

- You can transfer data between registers or between a register and a memory location.
- **Direct memory-to-memory transfer is not allowed.** The MOV instruction cannot move data directly from one memory location to another. This restriction exists because both operands cannot be memory at the same time in a MOV instruction.
- To transfer between two memory locations, you have to use a register as an intermediate:
  1. Read from the source memory location into a register.
  2. Write from that register to the destination memory location.

> Example for memory-to-memory transfer workaround:

```
MOV AX, [SOURCE] ; Load data from SOURCE memory to AX
MOV [DEST], AX   ; Store data from AX register to DEST memory
```

This two-step process is required because MOV does not allow `[mem1], [mem2]` operations.[1]

Role of PUSH and POP Instructions

PUSH and **POP** are stack operations used to store and retrieve data using the stack segment in the 8086 microprocessor.

- **PUSH**: Places data onto the stack. It decrements the SP (stack pointer) and writes the specified register or memory value to the stack.
- **POP**: Retrieves data from the stack. It reads the value from the stack and increments SP.

Use Case & Example:
Suppose you want to temporarily store the contents of AX and BX on the stack and then retrieve them:

```
PUSH AX   ; Store AX on the stack
PUSH BX   ; Store BX on the stack
; ... (other operations)
```

```
POP BX    ; Retrieve BX from the stack
POP AX    ; Retrieve AX from the stack
```

This pattern is useful during procedure calls where you need to save and restore register values. The stack structure ensures last-in, first-out (LIFO) retrieval, with POP restoring values in reverse order of PUSH.[1]

Indexed Addressing Mode in 8086

Indexed Addressing Mode utilizes index registers (SI - Source Index, DI - Destination Index) to calculate the effective memory address for an operation. This mode enables flexible access to elements within arrays or data structures.

- The **effective address** for memory access is calculated as: `BASE + INDEX + OFFSET`
- Common usage: accessing elements of an array, iterating through strings.

Example of Array Access:
Assume an array starts at address DS:1000h, SI points to the index:

```
MOV SI, 0000h       ; Initialize index to first element
MOV CX, LENGTH      ; Set number of elements
MOV AX, DS:[1000h]  ; Load first element of array
; Loop to access the entire array
LOOP_START:
 MOV AX, [1000h+SI] ; Access array element
 ; process AX
 ADD SI, 2          ; Move to next element (assuming 2 bytes per element)
 LOOP LOOP_START    ; Repeat for all elements
```

This addressing mode is especially useful for **string operations** or manipulating arrays, as you can update the index register to move to the next element in each loop iteration.[2] [1]

Summary Table

Instruction	Purpose	Direct Memory-to-Memory Transfer	Example Usage
MOV	Data transfer register↔register, register↔memory	✖	MOV AX, [1234h] / MOV [3456h], AX
PUSH/POP	Store/Retrieve register to/from stack	N/A	PUSH AX / POP AX
Indexed Addr.	Access arrays via index registers (SI or DI)	N/A	MOV AX, [BASE+SI]

⁂

1. https://www.youtube.com/watch?v=LnDMIaTJF6s
2. https://www.youtube.com/watch?v=EfwZ_TgfM_g
3. 1000014791.jpeg

4. 1000014796.jpg
5. https://www.youtube.com/watch?v=EfwZ