8086 Addressing Modes

Definition

Addressing mode = manner in which operand is specified in an instruction.

- An instruction consists of: **Opcode + Operand(s)**.
- Operand may reside in:
  - Accumulator (AX/AL)
  - General Purpose Register
  - Memory Location
  - Immediate constant
- Categories of addressing modes in 8086:

(i) Immediate Addressing

(ii) Register Addressing

(iii) Memory Addressing

- Memory Addressing further includes:

(i) Direct

(ii) Register Indirect

(iii) Based

(iv) Indexed

(v) Based-Indexed

(vi) Relative Based-Indexed

Segment Register Selection in 8086

Default Segment Register Rules

Addressing Mode	Offset Register(s)	Default Segment
Immediate	–	Not applicable
Register	–	Not applicable
Direct	Displacement	DS
Register Indirect	BX, SI, DI	DS
Register Indirect	BP	SS
Based	BX + disp	DS
Based	BP + disp	SS
Indexed	SI + disp	DS
Indexed	DI + disp	DS (can override to ES)
Based-Indexed	BX + SI	DS
Based-Indexed	BP + DI	SS
Based-Indexed + Disp	BX+SI+disp	DS
Based-Indexed + Disp	BP+DI+disp	SS
Relative Based-Indexed	Base + Index + disp	DS/SS (depends on base reg)

Segment Override Prefix

Concept

In 8086, the **physical address (PA)** is calculated as:

$$PA = (\text{Segment Register} \times 10H) + \text{Effective Address (EA)}$$

- By default:
  - Most memory references → **DS**
  - If base register = BP → **SS**
  - Instruction fetch → **CS**
  - String destination → **ES:DI**, source → **DS:SI**
- **Override**: Programmer can explicitly force segment usage with prefixes `CS:`, `DS:`, `SS:`, or `ES:`.

Examples:

- `MOV AX, [DI]` $\rightarrow PA = DS \times 10H + DI$
- `MOV AX, ES:[DI]` $\rightarrow PA = ES \times 10H + DI$
- `MOV DX, SS:[BP+4]` $\rightarrow$ access stack parameter at $SS : (BP + 4)$

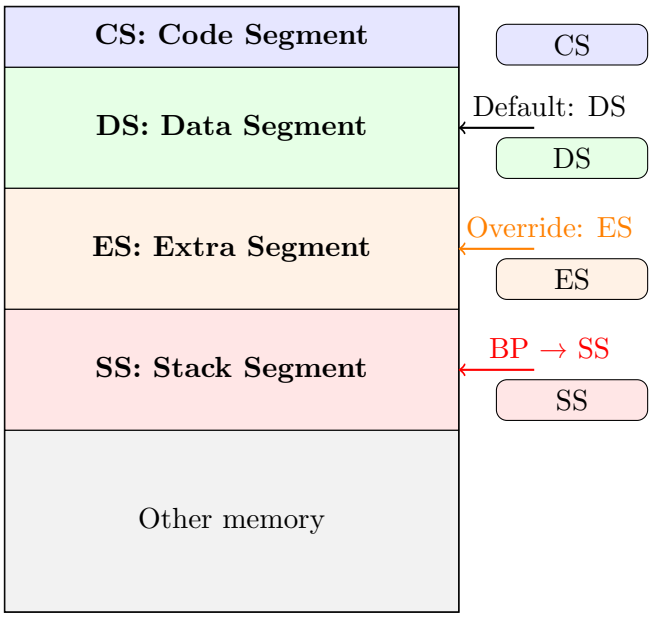

Figure 1: Default vs Segment Override Prefix in 8086

Use Cases:

1. Accessing data stored across multiple segments.
2. String operations (`MOVSB`, `STOSW`) where source = DS:SI and destination = ES:DI.
3. Accessing stack frames and procedure parameters directly.

Detailed Explanation of Addressing Modes

1. Immediate Addressing

Definition

Immediate addressing means the operand (constant value) is *directly embedded inside the instruction.* The processor reads the value from the instruction stream itself (not from memory). No additional memory reference is needed.

General Format:

```
<mnemonic> <destination>, <immediate>
```

Operand Size

- **Immediate byte (imm8):** 8-bit constant following the opcode (for 8-bit registers like `AL`, `BL`, etc.).
- **Immediate word (imm16):** 16-bit constant (2 bytes) following the opcode (for 16-bit registers like `AX`, `BX`, etc.).

Key Properties:

- Operand is part of the *instruction code* (code segment), not stored in data memory.
- Immediates are **read-only constants** ⇒ can appear only as source, not as destination.
- Faster than memory addressing, since no extra memory access is required.
- Increases code size because every immediate value must be stored with the instruction.

Typical Uses

- **Initialization:** setting counters, loop bounds, offsets.
- **Arithmetic/Logic:** adding or masking with fixed constants.
- **Comparisons:** e.g., `CMP reg, imm` for conditional jumps.
- **I/O operations:** when a fixed port address or control constant is required.

Examples (with explanation):

- `MOV AL, 15H` ⇒ Load constant `15H` into `AL`. No flags are affected.
- `MOV AX, 0B14H` ⇒ Load 16-bit constant `0B14H` into `AX`. Common for initializing pointers or counters.
- `ADD AL, 0AH` ⇒ `AL = AL + 0AH`. Updates flags (CF, ZF, SF, OF, PF, AF) according to the result.
- `CMP AX, 0001H` ⇒ Compare `AX` with `0001H` (subtraction internally). Flags updated for conditional branching (`JE`, `JNE`, etc.).

Effect on Flags:

- `MOV reg, imm` → does *not* change flags.
- ALU operations (`ADD, SUB, CMP, AND, OR, XOR`) → *do* update flags.

Advantages & Trade-offs

Advantages:

- Fast (operand is fetched with instruction).
- Simple for constants and initialization.

Trade-offs:

- Increases code size (immediates stored in code).
- Not good for frequently changing values.
- Limited by immediate size (8 or 16 bits).

Example

```
MOV AL, 15H   ; AL ← 15H (imm8), flags unaffected.
ADD AL, 0AH   ; AL ← AL + 0AH (imm8), flags updated.
MOV AX, 0B14H   ; AX ← 0B14H (imm16).
CMP AX, 0001H   ; Compare AX with imm16, flags set.
```

2. Register Addressing

Definition

In register addressing, the operand is stored in a *CPU register* instead of memory. The instruction directly specifies the register containing the operand. Since registers are inside the processor, this is the **fastest addressing mode**.

General Format:

```
<mnemonic> <destination>, <register>
```

Available Registers in 8086

8-bit general purpose: `AL, BL, CL, DL, AH, BH, CH, DH`

16-bit general purpose: `AX, BX, CX, DX, SP, BP, SI, DI`

Key Properties:

- Operand is already in a register — no memory access required.
- Instructions become shorter and faster (fewer clock cycles).
- Registers can serve as both source and destination.
- Operand size depends on register type (8-bit or 16-bit).

Typical Uses

- **Arithmetic and Logic:** e.g., `ADD AL, BL`, `XOR AX, BX`.
- **Data movement:** `MOV AX, BX` (register-to-register copy).
- **Loop counters / index registers:** registers like `CX` in loops.
- **Temporary storage:** holding intermediate values during computation.

Examples (with explanation):

- `ADD AL, BL` ⇒ Adds contents of `BL` to `AL`. Flags updated according to result.
- `MOV AX, BX` ⇒ Copies contents of `BX` into `AX`. No flags affected.
- `XOR CX, CX` ⇒ Clears `CX` (common trick to set register to 0). Flags updated (ZF = 1, others cleared).
- `SUB DX, AX` ⇒ Subtracts `AX` from `DX`, result stored in `DX`. Flags updated (ZF, SF, CF, OF).

Effect on Flags:

- `MOV reg, reg` → no flags affected.
- Arithmetic/Logic ops (`ADD, SUB, XOR, AND, OR`) → update flags normally.

Advantages & Trade-offs

Advantages:

- Fastest mode (all in CPU, no memory cycles).
- Compact instruction encoding.
- Registers reusable for multiple operations.

Trade-offs:

- Limited number of registers available.
- If many variables are needed, data must be spilled to memory.

Example

```
ADD AL, BL   ; AL ← AL + BL, flags updated.
MOV AX, BX   ; AX ← BX, flags unaffected.
XOR CX, CX   ; Clear CX (set to 0), flags updated.
SUB DX, AX   ; DX ← DX - AX, flags updated.
```

3. Direct Addressing

Definition

In **direct addressing**, the instruction contains the *explicit memory displacement (offset)* of the operand. The CPU calculates the physical address using the displacement along with a segment register (by default, DS). This mode is slower than register addressing because it requires a memory access.

Address Calculation Formula:

$$\text{Physical Address (PA)} = (\text{Segment Register} \times 10H) + \text{Displacement}$$

Example with Calculation

`MOV CX, [1234H]`

- Displacement = `1234H`.
- Assume DS = `0200H`.
- Then,

$$\text{PA} = (0200H \times 10H) + 1234H = 03234H$$

- Contents of memory location `03234H` are moved into `CX`.

Key Properties:

- Operand is located in **memory**, not in registers or inside the instruction itself.
- Requires:
  1. A **segment register** (default DS, or SS if BP is used).
  2. An explicit **displacement (offset)** provided in the instruction.
- Supports **segment override prefixes** (e.g., ES:[1234H]).
- Access is **slower** compared to register addressing due to memory fetch.

Typical Uses

- Accessing **global/static variables** stored at fixed addresses.
- Fetching or storing data at known memory locations.
- Useful in low-level code where absolute memory references are required.

Examples (with explanation):

- `MOV AX, [4321H]` ⇒ Copies contents of memory at DS:4321H into AX. Flags unaffected.
- `ADD AL, [2000H]` ⇒ Adds the byte at DS:2000H to AL. Flags updated based on the result.
- `CMP BX, [1000H]` ⇒ Compares BX with contents of DS:1000H (subtraction internally). Flags updated for conditional jumps.

Effect on Flags:

- `MOV` → does not affect flags.
- Arithmetic/logic instructions (`ADD, SUB, CMP, AND, OR`) → update flags normally.

Advantages & Trade-offs

Advantages:

- Simple — operand address is directly specified in instruction.
- Useful for absolute addressing of variables/constants.

Trade-offs:

- Requires memory access → slower than register mode.
- Increases instruction size (needs explicit displacement).
- Less flexible for dynamic data structures (arrays, stacks).

Example

```
MOV CX, [1234H]   ; CX ← [DS:1234H], no flags.
ADD AL, [2000H]   ; AL ← AL + [DS:2000H], flags updated.
CMP BX, [1000H]   ; Compare BX with [DS:1000H], flags updated.
```

4. Register Indirect Addressing

Definition

In **register indirect addressing**, the *effective address (EA)* of the operand is held inside a register. The instruction specifies which register to use, and the CPU accesses the memory location pointed to by that register. This mode is commonly used for working with arrays, pointers, and dynamic data structures.

Address Calculation:

$$\text{EA} = \text{Contents of register}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Registers Allowed

- With **DS (default segment)**: `BX`, `SI`, `DI`.
- With **SS (stack segment)**: `BP`.

Key Properties:

- Operand resides in memory, but the address is taken from a register.
- No explicit displacement needed in instruction (address is indirect).
- Access is slower than register addressing, but more flexible than direct addressing.
- Very useful for **array traversal, pointers, stacks, and parameter passing**.

Typical Uses

- **Arrays:** index register points to elements.
- **Pointers:** register holds a memory address of data.
- **Stack variables:** BP + SS is commonly used for procedure frames.

Examples (with explanation):

- `MOV AX, [SI]` ⇒ Copy word from DS:SI into AX. No flags affected.
- `ADD AL, [BX]` ⇒ Add byte at DS:BX to AL. Flags updated based on result.
- `MOV DX, [BP]` ⇒ Copy word from SS:BP into DX (since BP defaults to SS).
- `CMP AX, [DI]` ⇒ Compare AX with contents at DS:DI. Flags updated.

Effect on Flags:

- `MOV` → does not affect flags.
- Arithmetic/logic instructions (`ADD, SUB, CMP, AND, OR`) → update flags normally.

Advantages & Trade-offs

Advantages:

- More flexible than direct addressing (address stored in register, not fixed).
- Supports dynamic access to memory (arrays, pointers).
- Compact instruction encoding (no displacement required).

Trade-offs:

- Requires memory access (slower than register addressing).
- Still less flexible than indexed + displacement modes.

Example

```
MOV AX, [SI]    ; AX ← [DS:SI], flags unaffected.
ADD AL, [BX]    ; AL ← AL + [DS:BX], flags updated.
MOV DX, [BP]    ; DX ← [SS:BP], flags unaffected.
CMP AX, [DI]    ; Compare AX with [DS:DI], flags updated.
```

5. Based Addressing

Definition

In **based addressing**, the effective address (EA) of the operand is computed as the *sum of a base register and a displacement (offset)*. This mode is useful for accessing structured data, arrays, and stack-based variables with a fixed offset from a base.

Address Calculation:

$$\text{EA} = \text{Base Register (BX or BP)} + \text{Displacement}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Base Registers Allowed

- `BX` → default segment = DS
- `BP` → default segment = SS

Key Properties:

- Combines flexibility of register addressing with ability to access memory locations at fixed offsets.
- Supports access to both global (BX+disp) and stack variables (BP+disp).
- EA can be 8-bit or 16-bit displacement.
- Slower than register addressing but faster than full memory-indirect with multiple calculations.

Typical Uses

- **Structured Data:** Accessing fields within records (e.g., `Employee.salary = [BX+4]`).
- **Stack variables:** Procedure local variables or parameters via BP + offset.
- **Arrays:** When row base is in BX/BP and offset for element is known.

Examples (with explanation):

- `MOV AX, [BX+1000H]` ⇒ Copy word from DS:(BX+1000H) into AX.
- `ADD AL, [BP+04H]` ⇒ Add byte at SS:(BP+04H) to AL. Used to access stack parameters.
- `MOV DX, [BX+20H]` ⇒ Access global array element at DS:(BX+20H).

Effect on Flags:

- `MOV` → no effect.
- Arithmetic/logic instructions → flags updated normally.

Advantages & Trade-offs

Advantages:

- Flexible access to memory via a base + displacement.
- Supports structured and stack-based data efficiently.
- Instruction encoding is compact (displacement + base register).

Trade-offs:

- Requires memory access (slower than pure register addressing).
- Limited by size of displacement (8-bit or 16-bit).

Example

```
MOV AX, [BX+1000H]   ; AX ← [DS:BX+1000H], flags unaffected.
ADD AL, [BP+04H]     ; AL ← AL + [SS:BP+04H], flags updated.
MOV DX, [BX+20H]     ; DX ← [DS:BX+20H], flags unaffected.
```

6. Indexed Addressing

Definition

In **indexed addressing**, the effective address (EA) of the operand is calculated by adding an *index register* (SI or DI) to an optional displacement. This mode is mainly used for accessing elements of arrays or tables where the index points to a specific element.

Address Calculation:

$$\text{EA} = \text{Index Register (SI or DI)} + \text{Displacement (optional)}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Index Registers Allowed

- `SI` → default segment = DS
- `DI` → default segment = DS (can override to ES)

Key Properties:

- Provides flexible access to array or table elements.
- Displacement can be used for accessing specific rows/columns.
- Faster than direct memory addressing if index register already contains the required offset.
- Can combine with segment override prefix to access different segments.

Typical Uses

- Accessing arrays, tables, or buffers.
- Iterating through memory structures using SI or DI as index counters.
- String operations: e.g., source index = SI, destination index = DI.

Examples (with explanation):

- `MOV AL, [SI]` ⇒ Load byte from DS:SI into AL. Useful for array access.
- `ADD AX, [DI+10H]` ⇒ Add word at DS:(DI+10H) to AX. Displacement allows access to array element offset.
- `MOV CX, [SI+200H]` ⇒ Access element at DS:(SI+200H) and move it into CX.

Effect on Flags:

- `MOV` → no effect.
- Arithmetic/logic instructions → flags updated as per operation.

Advantages & Trade-offs

Advantages:

- Efficient access to array and table elements.
- Can combine index and displacement for flexible memory addressing.
- Compact instruction encoding.

Trade-offs:

- Requires memory access (slower than register addressing).
- Limited by index register width (16-bit) and optional displacement size.

Example

```
MOV AL, [SI]    ; AL ← [DS:SI], flags unaffected.
ADD AX, [DI+10H]    ; AX ← AX + [DS:DI+10H], flags updated.
MOV CX, [SI+200H]    ; CX ← [DS:SI+200H], flags unaffected.
```

7. Based-Indexed Addressing

Definition

In **based-indexed addressing**, the effective address (EA) of the operand is calculated by adding a *base register* (BX or BP) and an *index register* (SI or DI). This mode allows flexible access to complex data structures such as multi-dimensional arrays and records.

Address Calculation:

$$\text{EA} = \text{Base Register (BX or BP)} + \text{Index Register (SI or DI)}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Base and Index Registers Allowed

- Base: `BX` (default segment = DS), `BP` (default segment = SS)
- Index: `SI`, `DI` (default segment = DS)

Key Properties:

- Combines two registers for EA calculation → supports structured memory layouts.
- Access is **slower** than register addressing, but provides more flexibility than direct or indexed addressing alone.
- Useful for accessing multi-dimensional arrays (row + column) or records with fields.
- Can combine with displacement or segment override for even greater flexibility.

Typical Uses

- Accessing 2D arrays or matrices (row = base, column = index).
- Accessing records or structures with multiple fields.
- Iterating through tables in memory where base points to start of structure and index points to element offset.

Examples (with explanation):

- `MOV AL, [BX+SI]` ⇒ Load byte from memory at DS:(BX+SI) into AL. Base = start of array, SI = index.
- `ADD AX, [BP+DI]` ⇒ Add word at SS:(BP+DI) to AX. Useful for accessing stack-based array of structures.
- `MOV CX, [BX+DI]` ⇒ Access word at DS:(BX+DI) and move into CX. Combines base and index for structured memory layout.

Effect on Flags:

- `MOV` → no effect on flags.
- Arithmetic/logic instructions → flags updated according to operation.

Advantages & Trade-offs

Advantages:

- Flexible access to complex memory structures.
- Can represent two-level addressing (base + index) efficiently.

Trade-offs:

- Requires memory access → slower than register addressing.
- Slightly larger instruction encoding due to two registers used.

Example

```
MOV AL, [BX+SI]   ; AL ← [DS:BX+SI], flags unaffected.
ADD AX, [BP+DI]   ; AX ← AX + [SS:BP+DI], flags updated.
MOV CX, [BX+DI]   ; CX ← [DS:BX+DI], flags unaffected.
```

8. Based-Indexed with Displacement Addressing

Definition

In **based-indexed with displacement addressing**, the effective address (EA) of the operand is calculated by adding a *base register* (BX or BP), an *index register* (SI or DI), and a *displacement* (offset). This mode is particularly useful for accessing structured data such as arrays of records or multi-dimensional arrays with a starting offset.

Address Calculation:

$$\text{EA} = \text{Base Register (BX/BP)} + \text{Index Register (SI/DI)} + \text{Displacement}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Base, Index, and Displacement Details

- **Base:** BX (DS) or BP (SS)
- **Index:** SI or DI (DS)
- **Displacement:** 8-bit or 16-bit constant value provided in instruction

Key Properties:

- Combines base, index, and displacement → supports advanced memory structures.
- Access is slower than register-only or direct addressing due to memory fetch.
- Highly flexible for accessing arrays of structures with a starting offset.
- Can also work with segment override prefixes for non-default segments.

Typical Uses

- Accessing records or structures within arrays.
- Multi-dimensional array elements with row + column + base offset.
- Stack frame variables with base pointer + index + local offset.

Examples (with explanation):

- `MOV AX, [BP+SI+2000H]` ⇒ Loads word from memory at SS:(BP+SI+2000H) into AX. Base = start of stack frame, SI = index, 2000H = displacement.
- `ADD AL, [BX+DI+10H]` ⇒ Adds byte from DS:(BX+DI+10H) to AL. Useful for arrays of structures with field offsets.
- `MOV CX, [BX+SI+1234H]` ⇒ Access word at DS:(BX+SI+1234H) and move into CX. Combines base, index, and displacement for structured memory layout.

Effect on Flags:

- `MOV` → no flags affected.
- Arithmetic/logic instructions → flags updated according to operation.

Advantages & Trade-offs

Advantages:

- Extremely flexible for complex data structures.
- Efficient for accessing elements in arrays of records.

Trade-offs:

- Requires memory access → slower than register-only addressing.
- Instruction size is larger due to displacement.

Example

```
MOV AX, [BP+SI+2000H]   ; AX ← [SS:BP+SI+2000H], flags unaffected.
ADD AL, [BX+DI+10H]     ; AL ← AL + [DS:BX+DI+10H], flags updated.
MOV CX, [BX+SI+1234H]   ; CX ← [DS:BX+SI+1234H], flags unaffected.
```

9. Relative Based-Indexed Addressing

Definition

In **relative based-indexed addressing**, the effective address (EA) of the operand is computed by adding a *base register* (BX or BP), an *index register* (SI or DI), and a *displacement (relative offset)*. This mode is especially useful for accessing elements in multi-dimensional arrays, tables, or structures with a variable starting point.

Address Calculation:

$$\text{EA} = \text{Base Register (BX/BP)} + \text{Index Register (SI/DI)} + \text{Displacement (rel)}$$

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{EA}$$

Components of EA

- **Base:** BX (default segment DS) or BP (default segment SS)
- **Index:** SI or DI (default segment DS)
- **Relative Displacement:** 8-bit or 16-bit signed offset provided in instruction

Key Properties:

- Combines base, index, and a relative displacement → allows flexible memory addressing.
- Slower than register or direct addressing since memory must be accessed.
- Supports dynamic access to arrays, tables, and stack frames.
- Segment override prefix can modify the default segment if needed.

Typical Uses

- Accessing 2D arrays (Base = row, Index = column, Displacement = starting address).
- Tables with variable offsets.
- Stack frame access with base pointer + index + relative local variable offset.

Examples (with explanation):

- `MOV AH, [BX+SI+1234H]` ⇒ Loads byte from DS:(BX+SI+1234H) into AH. Base = BX (row), SI = column, 1234H = starting offset.
- `ADD AX, [BP+DI+10H]` ⇒ Adds word from SS:(BP+DI+10H) to AX. Useful for accessing stack frame arrays with relative offsets.
- `MOV CX, [BX+DI-8]` ⇒ Accesses word at DS:(BX+DI-8) for table or array indexing with negative displacement.

Effect on Flags:

- `MOV` → no flags affected.
- Arithmetic/logic instructions → flags updated according to operation.

Advantages & Trade-offs

Advantages:

- Extremely flexible for structured and multi-dimensional memory access.
- Can handle dynamic displacement offsets efficiently.

Trade-offs:

- Slower due to multiple memory accesses.
- Instruction size increases with displacement.
- Slightly more complex to calculate effective addresses.

Example

```
MOV AH, [BX+SI+1234H]   ; AH ← [DS:BX+SI+1234H], flags unaffected.
ADD AX, [BP+DI+10H]     ; AX ← AX + [SS:BP+DI+10H], flags updated.
MOV CX, [BX+DI-8]       ; CX ← [DS:BX+DI-8], flags unaffected.
```

Summary of Addressing Modes

Key Takeaways

- **Immediate and Register Addressing:** Fastest modes since operand is inside instruction or CPU register → *no memory access required.*
- **Direct and Register Indirect Addressing:** Access memory via explicit displacement (Direct) or through a pointer in a register (Indirect). Slower than register addressing.
- **Based, Indexed, and Complex Modes (Based-Indexed / With Displacement / Relative Based-Indexed):** Highly flexible for accessing arrays, tables, records, and stack frames. Allows combination of base, index, and displacement.
- **Physical Address Calculation:**

$$\text{PA} = (\text{Segment Register} \times 10H) + \text{Effective Address (EA)}$$

- **Default Segment Registers:**

  - `DS` → Data
  - `SS` → Stack
  - `CS` → Code
  - `ES` → Extra (for strings, etc.)

- **Segment Override Prefix:** Allows accessing memory from a non-default segment, e.g., `MOV AX, ES:[DI]` → fetches operand from ES instead of DS.
- **Usage Tip:** Use simple addressing for speed; complex modes for structured data access.

8086 Addressing Modes Summary

Mode	EA Formula	Default Segment	Registers Used	Typical Use
Immediate	Operand is part of instruction	N/A	N/A	Load constants into registers/memory
Register	Operand is a register	N/A	AX, BX, CX, DX, SI, DI, BP, SP	Fast data operations
Direct	EA = Address field in instruction	DS	N/A	Access variables in data segment
Register Indirect	EA = [Register]	BP → SS Others → DS	BX, BP, SI, DI	Access arrays, stack parameters
Based	EA = Base + Displacement	BP → SS BX → DS	BX, BP	Structures, local variables
Indexed	EA = Index + Displacement	DS	SI, DI	Array element access
Based-Indexed	EA = Base + Index + Displacement	BP → SS Others → DS	BX/BP + SI/DI	Access structured arrays
Based-Indexed with Displacement	EA = Base + Index + Displacement	BP → SS Others → DS	BX/BP + SI/DI	Access complex structures, function parameters