DATA TRANSFER IN ASSEMBLY: WHAT'S REALLY GOING ON

At its core, **data transfer** in assembly is about moving values around so the CPU can work with them. The processor itself can only operate on data that's in the right place—usually a register—so most assembly programs spend a lot of time copying values between:

- **Registers** (inside the CPU)
- **Memory** (RAM)
- **The stack** (a special area of memory used for function calls and temporary storage)

Nothing "magical" happens during data transfer. The CPU simply copies bits from one location to another. Understanding *where* data comes from and *where* it goes is the foundation of everything else in assembly.

The Three Basic Data Transfer Instructions

MOV

- Copies data from a source to a destination
- Does **not** modify the original source
- Does **not** perform calculations

Think of MOV as a straight copy-paste operation.

PUSH

- Places a value onto the stack
- Automatically adjusts the stack pointer

POP

- Removes a value from the stack
- Stores it somewhere (usually a register)
- Automatically adjusts the stack pointer back

Together, PUSH and POP are essential for function calls, saving registers, and managing temporary data.

Operand Types: What Instructions Work With

Every assembly instruction operates on **operands.** An operand is simply the thing the instruction uses or modifies.

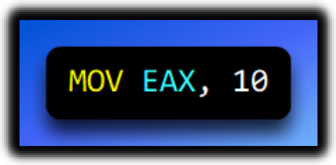
There are **three basic operand types** in x86 assembly:

1. Immediate Operands

Immediate operands are **literal values written directly in the instruction.**

Examples:

- 10
- -255
- 0FFh



```
MOV EAX, 10
```

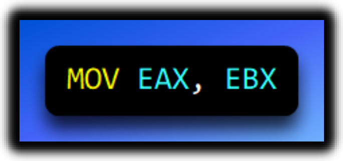
Here, 10 is not stored in memory or a register beforehand—it's embedded directly in the instruction.

Why this matters:
Immediate values are fast and convenient, but they're fixed constants. You can't change them at runtime.

2. Register Operands

Register operands refer to **CPU registers**, such as:

- EAX
- EBX
- ECX
- EDX



```
MOV EAX, EBX
```

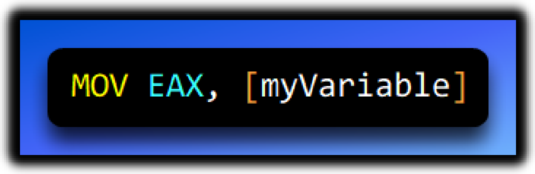
Registers are:

- Extremely fast
- Very limited in number
- Where almost all real computation happens

Key idea:
If the CPU is going to do math or logic, the data usually has to be in registers first.

3. Memory Operands

Memory operands reference **locations in RAM**.



```
MOV EAX, [myVariable]
```

Memory is:

- Much larger than registers
- Slower to access
- Where most program data lives long-term

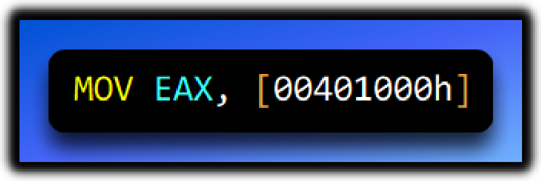
Assembly forces you to be explicit about memory access. You never “accidentally” touch memory—you have to say exactly where.

Addressing Modes: How Memory Is Reached

When an instruction refers to memory, the CPU needs to know **how to find that memory address**. This is where addressing modes come in.

1. Direct Addressing

Direct addressing specifies the memory location explicitly.



```
MOV EAX, [00401000h]
```

What this means:

- myValue represents a fixed memory address
- The CPU goes directly to that address and reads the value

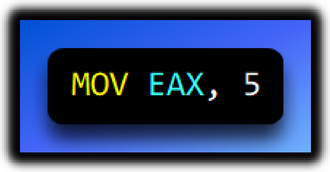
Key characteristics:

- The address is fixed
- Very clear and readable
- Mostly used with labels and global variables

In real programs, direct addressing is common when working with named data defined in the data segment.

2. Immediate Addressing (Not Memory!)

Immediate addressing does **not** access memory at all.



```
MOV EAX, 5
```

What's happening:

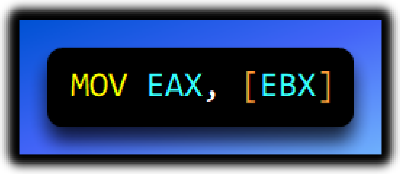
- The value 10 is placed directly into EAX
- No memory lookup occurs

This is included here because beginners often confuse it with memory access — but **it isn't**.

Rule of thumb: No brackets → no memory access

3. Indirect Addressing

Indirect addressing uses a **register that contains a memory address**.



```
MOV EAX, [EBX]
```

Step-by-step:

1. EBX holds a memory address
2. The CPU looks at that address
3. The value stored there is loaded into EAX

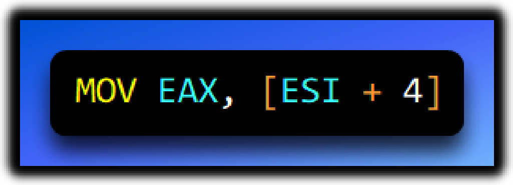
This is **exactly how pointers work** at the assembly level.

Important distinction:

- ebx → the number inside the register
- [ebx] → the data stored at the address contained in EBX

4. Indexed Addressing

Indexed addressing calculates a memory address using a **base register plus an offset**.



```
MOV EAX, [ESI + 4]
```

What's happening:

- Start at the address in EBX
- Move forward by 4 bytes
- Read the value stored there

This addressing mode is commonly used for:

- Arrays
- Structures
- Walking through memory in loops

Mental model:

“Start here, then move forward this many bytes.”

Indexed addressing is the foundation of data structures in assembly.

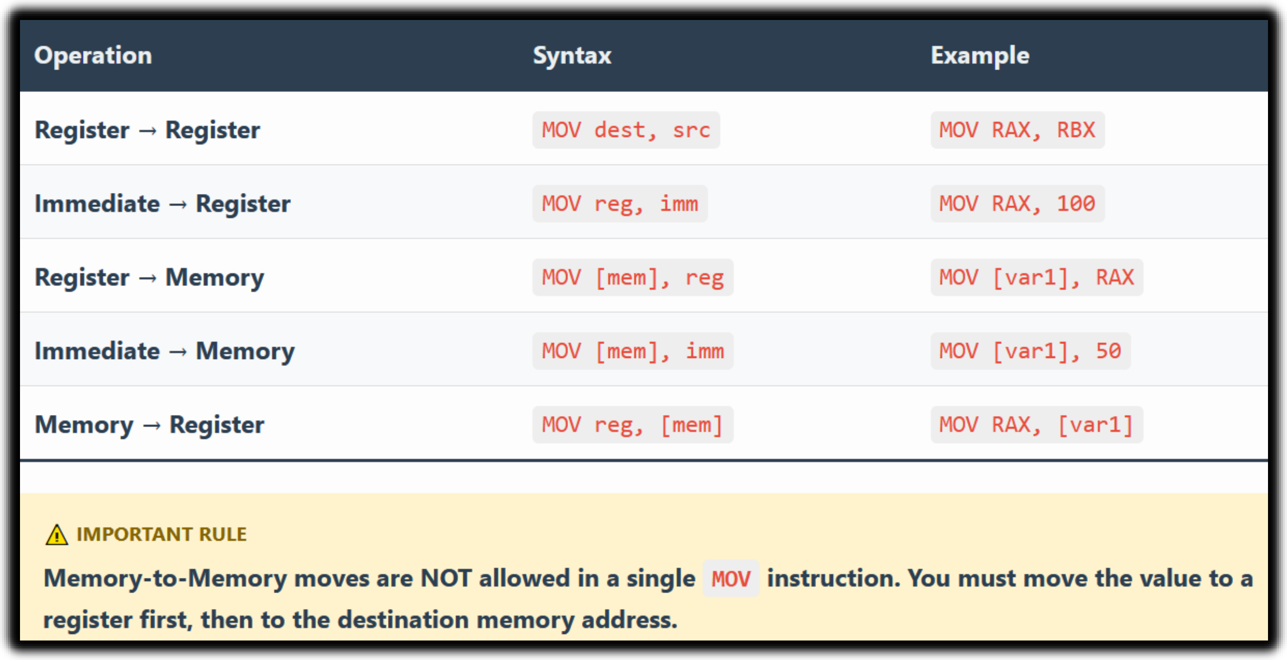
Data Transfer Instructions: MOV, PUSH, and POP

With operands and addressing modes understood, data transfer instructions become much clearer.

MOV — Copy Data

MOV copies data from a source to a destination.

Examples:

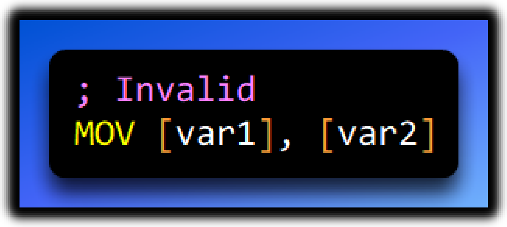


Operation	Syntax	Example
Register → Register	`MOV dest, src`	`MOV RAX, RBX`
Immediate → Register	`MOV reg, imm`	`MOV RAX, 100`
Register → Memory	`MOV [mem], reg`	`MOV [var1], RAX`
Immediate → Memory	`MOV [mem], imm`	`MOV [var1], 50`
Memory → Register	`MOV reg, [mem]`	`MOV RAX, [var1]`

⚠ IMPORTANT RULE

Memory-to-Memory moves are NOT allowed in a single `MOV` instruction. You must move the value to a register first, then to the destination memory address.

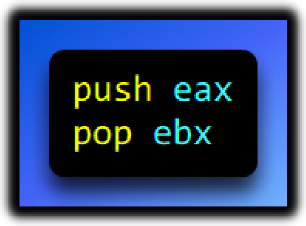
Important rule:
❌ You **cannot** move memory directly to memory and
✅ One operand must be a register.



```
; Invalid
MOV [var1], [var2]
```

PUSH and POP — Stack Transfers

The stack is a special region of memory managed using the stack pointer (ESP).



```
push eax
pop ebx
```

What PUSH does:

1. Decreases ESP
2. Stores the value at the new top of the stack

What POP does:

1. Reads the value at the top of the stack
2. Increases ESP

The stack is heavily used for:

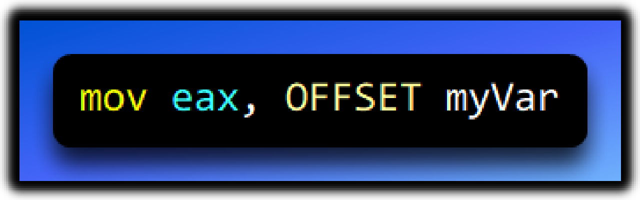
- Function calls
- Passing parameters
- Saving registers

Operators That Help with Memory

Assembly provides operators that help calculate and interpret memory addresses.

I. OFFSET

OFFSET gives the **address** of a variable, not its value.

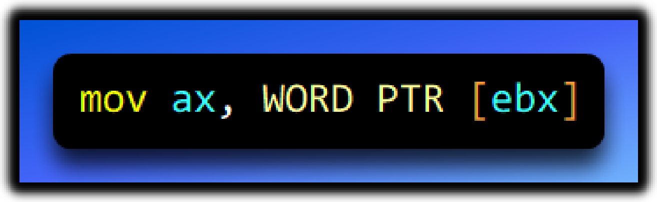


```
mov eax, OFFSET myVar
```

This loads the memory address of myVar into EAX.

II. PTR

PTR tells the assembler **how to treat a memory operand**.



```
mov ax, WORD PTR [ebx]
```

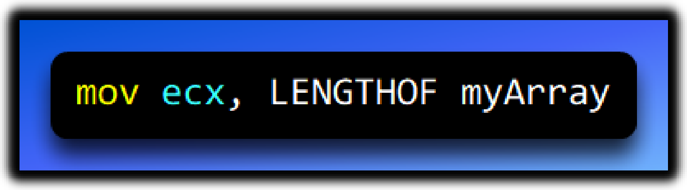
This forces the assembler to treat the memory as a WORD.

This matters because:

- Assemblers do not perform strict type checking
- The CPU needs to know how many bytes to read

III. LENGTHOF

LENGTHOF calculates how many elements are in a data structure.



```
mov ecx, LENGTHOF myArray
```

This is commonly used when writing loops.

Loops and Arithmetic (Preview)

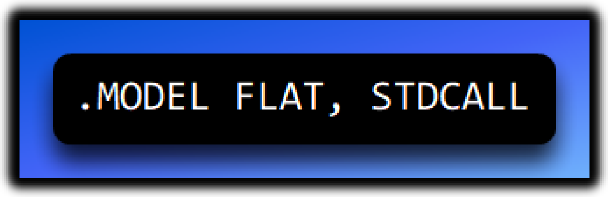
With data transfer understood, you can now:

- Create loops using JMP and LOOP
- Perform arithmetic with ADD, SUB, MUL, and DIV
- Move through arrays and structures using indexed addressing

All of these depend on **correct data movement**.

Flat Memory Model and STDCALL (Windows Context)

When writing 32-bit Windows programs, you'll often see:



```
.MODEL FLAT, STDCALL
```

I. Flat Memory Model

- One continuous 32-bit address space
- No segment juggling
- Memory is treated as a single linear block

This simplifies memory access and matches how modern Windows works.

II. STDCALL Calling Convention

STDCALL defines:

- How parameters are passed (right to left on the stack)
- Who cleans up the stack (the callee)
- How functions interact with the Windows API

This consistency is critical for Windows compatibility.

III. Big Picture Summary

- Data transfer moves values between registers, memory, and the stack
- Operands define *what* data is used
- Addressing modes define *how* memory is reached
- MOV, PUSH, and POP are the core transfer instructions
- OFFSET, PTR, and LENGTHOF help manage memory correctly
- Flat memory and STDCALL define the Windows execution environment

Once this chapter clicks, you're no longer guessing —