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.

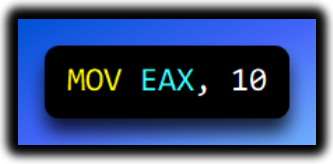
There are **three basic operand types** in x86 assembly:

1. Immediate Operands

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

Examples:

- 10
- -255
- 0FFh



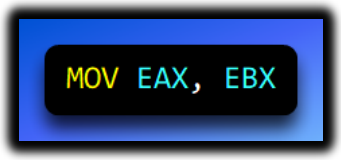
Here, 10 is not stored in memory or a register beforehand—it's embedded directly in the instruction.

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



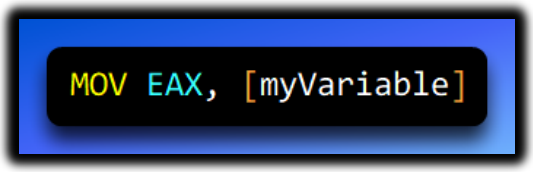
Registers are:

- 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**.



Memory is:

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

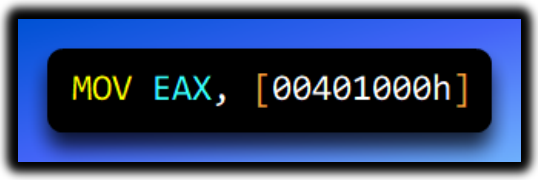
Assembly forces you to be explicit about memory access. You never “accidentally” touch memory—you have to say exactly where.

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.



What this means:

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

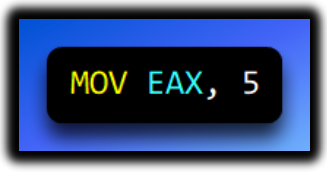
Key characteristics:

- 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.



What's happening:

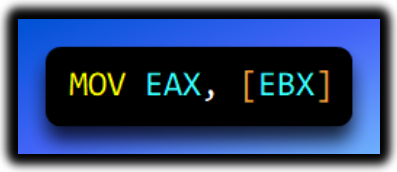
- The value 10 is placed directly into EAX
- No memory lookup occurs

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**.



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

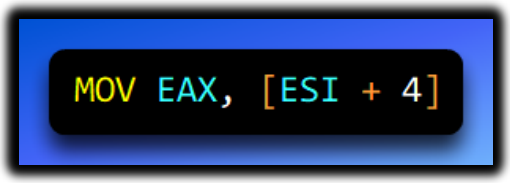
This is **exactly how pointers work** at the assembly level.

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**.



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.

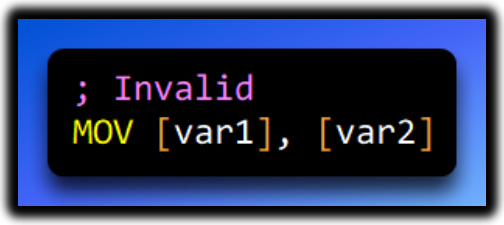
Examples:

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.



PUSH and POP — Stack Transfers

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

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:

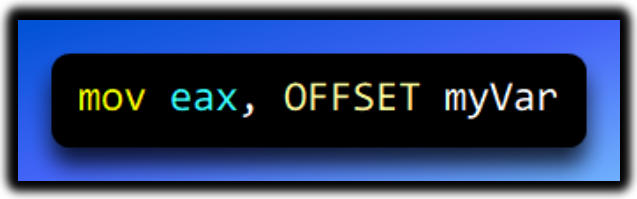
- Function calls
- Passing parameters
- Saving registers

Operators That Help with Memory

Assembly provides operators that help calculate and interpret memory addresses.

I. OFFSET

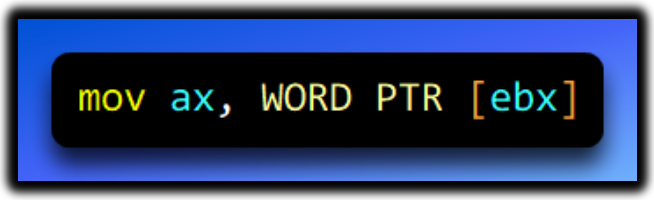
OFFSET gives the **address** of a variable, not its value.



This loads the memory address of myVar into EAX.

II. PTR

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



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 —

DIRECT MEMORY OPERANDS

The real concepts being discussed here are:

1. **Direct memory operands (a form of direct addressing)**
2. **The difference between a value, an address, and the contents at an address**
3. **How notation (var, [var], hex literals) changes meaning**

Direct Memory Operands: Talking to Memory by Name

A **direct memory operand** means:

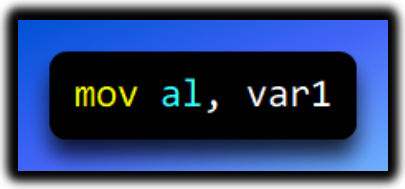
"Access the contents of a specific memory location whose address is known at assembly time."

In plain English:

- The assembler knows *exactly where this variable lives in memory*
- The instruction hardcodes that address into the machine code

I. What "direct" really means here

When you write:

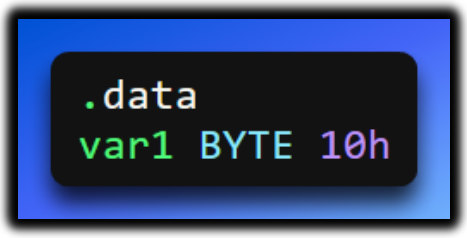


You are saying:

"Go to the memory location associated with var1, read the byte stored there, and copy it into AL."

var1 is **not** the value itself.
var1 is a **label** that represents a **fixed memory address**.

II. Declaring a variable: clearing the confusion



This line means **three separate things**:

1. var1
   → a label (a name for a memory location)
2. BYTE
   → the size of memory being reserved (1 byte)
3. 10h
   → the *initial value* stored in that byte
   → hexadecimal 10h = decimal 16

So what does var1 contain?

- It contains **one byte**
- That byte's value is **16 (decimal)**

What it does *not* mean

- ❌ It does NOT mean "a string"
- ❌ It does NOT mean "hex data is a string"
- ❌ It does NOT mean "10 characters"

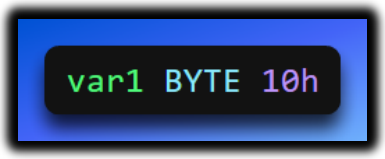
Hexadecimal is just a **number format**, not a data type.

III. Hex ≠ string (important mental reset)

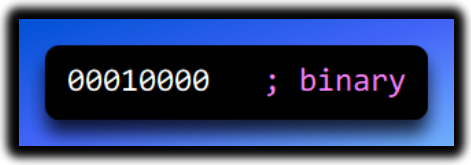
This is where many people get tripped up.

- **Hexadecimal** → a way to *write numbers*
- **String** → a sequence of characters stored as numeric codes (ASCII / Unicode)

Example:



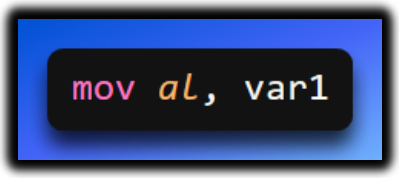
Stores:



That byte does *not* represent text unless *you interpret it as text* — and even then, ASCII 16 is a non-printable control character.

So no, a BYTE holding 10h cannot secretly be a string.

IV. Direct memory operand in action

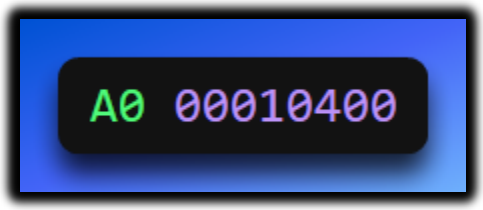


What the CPU actually does:

1. The assembler replaces var1 with its memory address
2. That address is embedded into the instruction
3. At runtime, the CPU:
    - goes to that address
    - reads **1 byte**
    - loads it into AL

V. Why machine code looks like this

We mentioned this:



Breaking it down:

- A0 → opcode (MOV AL, moffs8)
- 00010400 → 32-bit memory address of var1

This is why it's called **direct**:

the address is literally baked into the instruction.

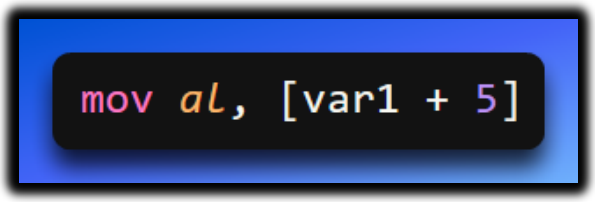
VI. Direct memory operands with other instructions

Anywhere a memory operand is allowed, direct memory operands can be used:

These all mean:

"Use the contents of the memory location named var1."

VII. Direct memory + expressions



This is called a **direct-offset operand**.

Meaning:

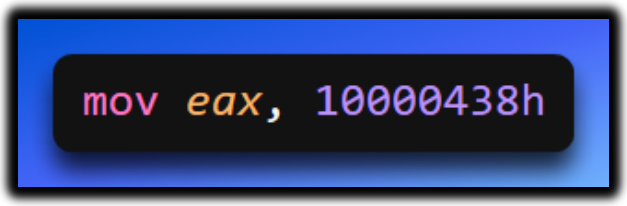
1. Take the address of var1
2. Add 5 bytes
3. Access the memory at that computed address

This is commonly used for:

- arrays
- structure fields
- table lookups

VIII. Why brackets matter (this is HUGE)

No brackets → value or address

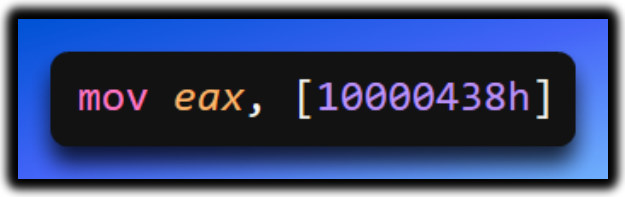


This means:

"Load the *number* 10000438h into EAX."

The CPU does **not** touch memory.

Brackets → dereference (go to memory)



This means:

"Go to memory address 10000438h and load what's stored there."

Golden rule:

Brackets mean "treat this as an address and go there."

This applies whether the thing inside is:

- a label → [var1]
- a register → [ebx]
- a hex literal → [10000438h]

IX. Can a hex number be a value *or* an address?

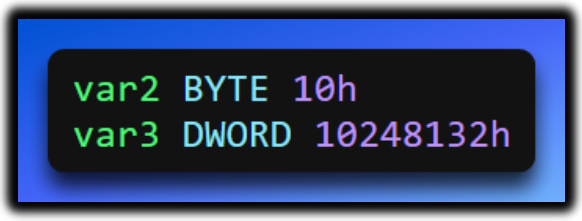
Yes — and this is why context matters.

Hex is just a number.

These are different even though the hex is the same:

Same number.
Completely different meaning.

X. BYTE vs DWORD (size clarity)



- BYTE → 1 byte (0–255)
- DWORD → 4 bytes (32 bits)

The type:

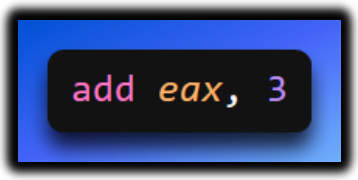
- tells the assembler how much space to reserve
- tells the CPU how many bytes to read/write

It does **not** decide whether something is a value or an address — brackets do.

XI. Opcode vs Operand (clean mental split)

- **Opcode** → what the CPU should do
  (MOV, ADD, SUB)
- **Operands** → what data the operation works on
  (registers, memory, immediates)

Example:



- Opcode → ADD
- Operands → EAX, 3

XII. Big picture takeaway (this is the anchor)

Direct memory operands mean:

- You name a variable
- The assembler knows its exact address
- The instruction directly accesses that memory location

Brackets decide everything:

- No brackets → value
- Brackets → memory dereference

Once this clicks, pointers, arrays, and structures stop feeling mysterious — they're just **address + size + interpretation**.

MOV - THE BACKBONE OF ASSEMBLY

The MOV instruction as a controlled copy operation, governed by operand roles, sizes, and allowed combinations.

The MOV instruction is used to **copy data from one place to another**.

```
MOV destination, source


; You can also write it in lowercase – assembly is not case-sensitive:

mov destination, source
```

What MOV actually does

MOV **copies** data.
It does **not**:

- add
- swap
- compare
- or modify the source

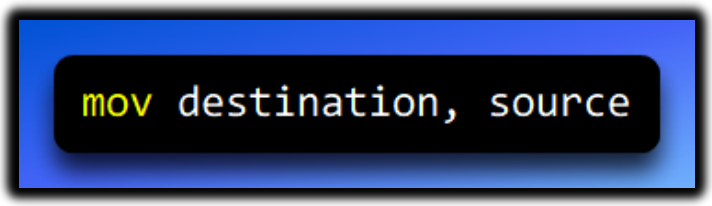
After a MOV:

- the **destination changes**
- the **source stays exactly the same**

Think of it like copying text:

You paste the text somewhere else, but the original is untouched.

Destination vs Source (this matters)



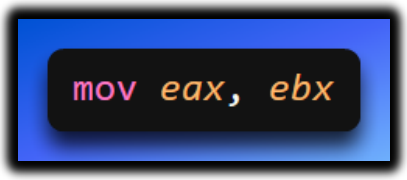
Destination operand

- The place where the data ends up
- This operand is **modified**

Source operand

- The data being copied
- This operand is **not modified**

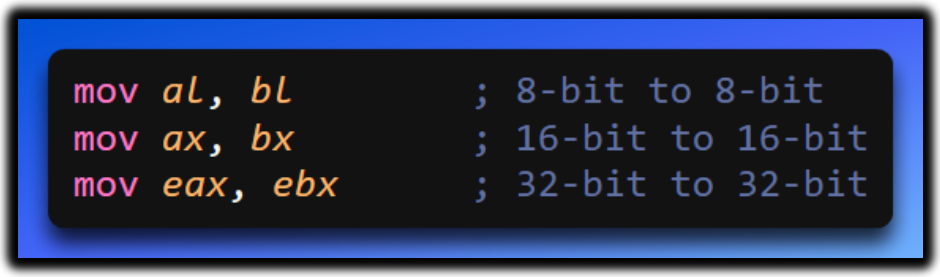
Example:



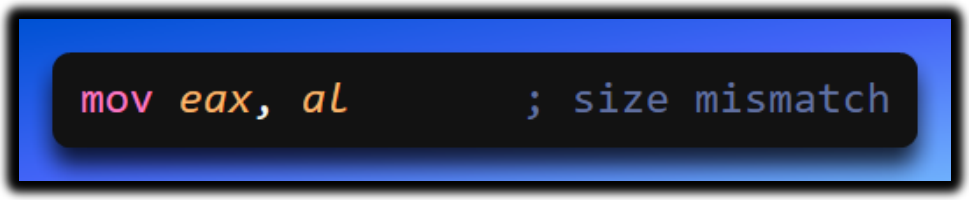
- EAX → destination (changed)
- EBX → source (unchanged)

Operand sizes must match

Both operands must be the **same size**. Valid:



Invalid:



The CPU needs to know *exactly how many bytes* to move.

MOV cannot move memory to memory (and why)

This is one of the most important rules:

❌ **MOV cannot copy data directly from one memory location to another.**

This is not a syntax limitation — it's a **CPU rule**.

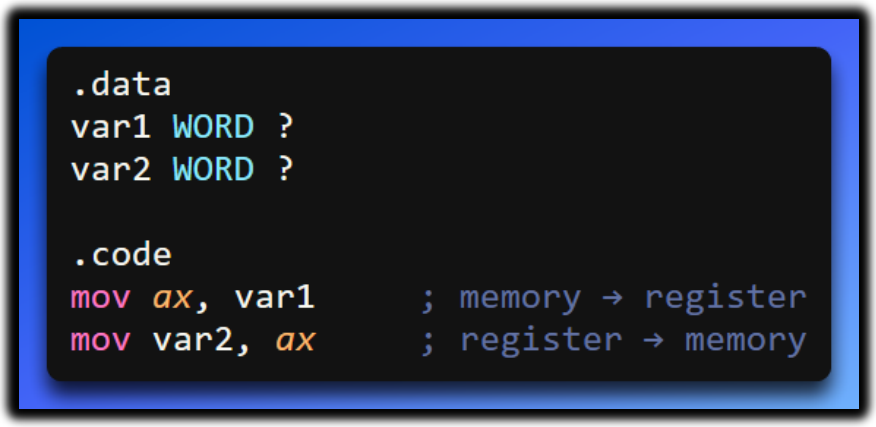
Memory-to-memory moves would require:

- two memory reads
- one memory write
- extra internal buffering

x86 keeps things simple and fast by requiring **at least one operand to be a register**.

How to move data from memory to memory (the correct way)

You use a register as a temporary step.



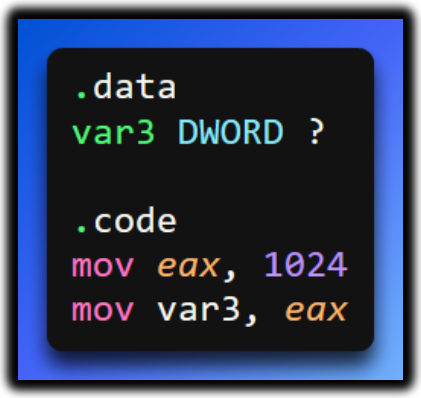
What's happening:

1. The value of var1 is loaded into AX
2. The value in AX is stored into var2

Result:

- var2 now contains the same value as var1

Register → Memory example



Step-by-step:

1. 1024 is loaded into EAX
2. The contents of EAX are copied into memory at var3

Important idea:

- var3 is just a **name for a memory address**
- The assembler reserves **4 bytes** because it is a DWORD

After execution:

- var3 holds the value 1024

Memory → Register example

What happens:

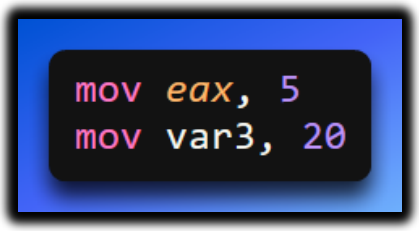
1. The CPU goes to the memory location named var4
2. Reads 4 bytes
3. Copies them into EAX

After this instruction:

- EAX = 1024

Immediate values with MOV

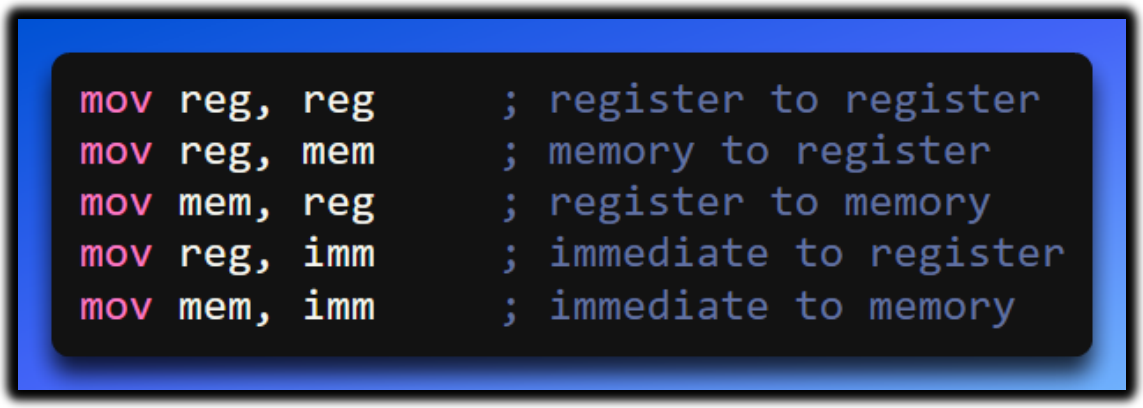
You can also move **immediate (literal) values** into registers or memory.



Rules:

- Immediate → register ✓
- Immediate → memory ✓
- Immediate → immediate ❌ (doesn't make sense)

Summary: All valid MOV forms



And the one that's **not allowed**:

Why MOV is so important

Almost everything in assembly depends on MOV:

- loading values
- storing results
- passing function parameters
- working with memory
- preparing data for arithmetic

If you understand:

- **destination vs source**
- **size matching**
- **register involvement**

then you understand how data flows through the CPU.

Big picture takeaway

MOV is not “move” — it’s **copy**.

It copies:

- values
- addresses
- memory contents

But it always follows strict rules so the CPU knows:

- how much data to copy
- where it’s coming from
- where it’s going

Master MOV, and assembly stops feeling chaotic — it becomes **deliberate and predictable**.

OVERLAPPING REGISTERS AND PARTIAL REGISTER WRITES IN X86

More specifically:

- How **AL, AX, and EAX share the same physical storage**
- How **writing to a smaller part of a register affects (or does not affect) the rest**

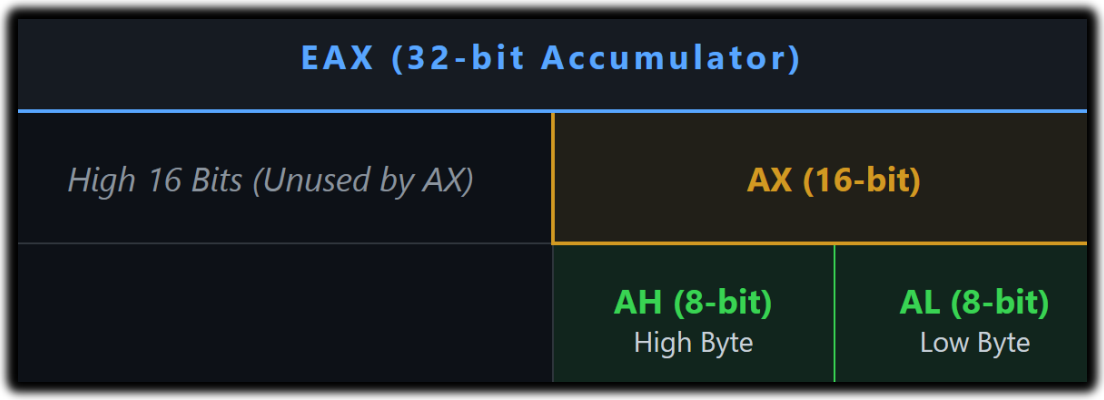
Overlapping Values: How Partial Register Writes Work

In x86 assembly, many registers **overlap**.
This means that smaller registers are **not separate storage** — they are **views into a larger register**.

The best example is EAX.

I. The EAX register layout (mental model)

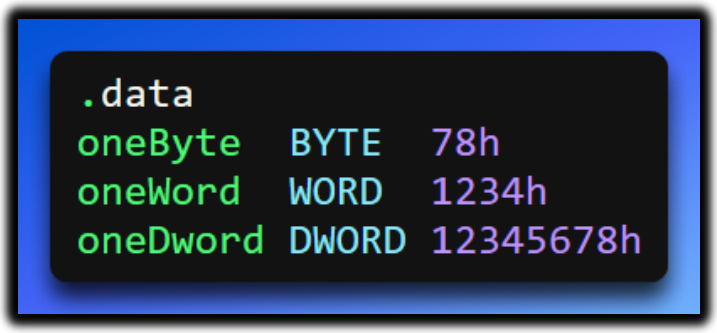
A 32-bit register like EAX is divided like this:



- **AL** → lowest 8 bits
- **AH** → next 8 bits
- **AX** → lowest 16 bits (AH + AL)
- **EAX** → all 32 bits

They all refer to the **same physical register**.

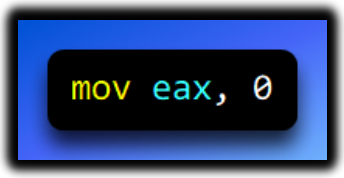
II. Data declarations (what we're loading)



- oneByte → 8 bits → 78h
- oneWord → 16 bits → 1234h
- oneDword → 32 bits → 12345678h

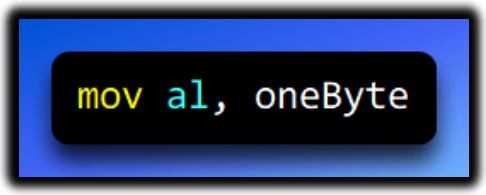
Step-by-step: how EAX changes

I. Clear EAX



All 32 bits are zeroed: **EAX = 00000000h**

II. Move a BYTE into AL



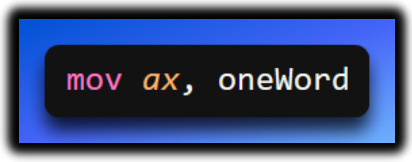
What happens:

- Only **AL** (lowest 8 bits) is overwritten
- The upper 24 bits remain unchanged

EAX = 00000078h

Key idea: Writing to AL affects only 1 byte.

III. Move a WORD into AX



What happens:

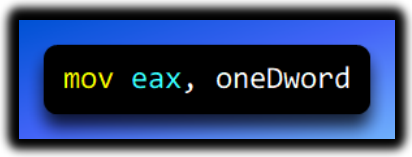
- The lower **16 bits** are overwritten
- The upper **16 bits** remain unchanged

EAX = 00001234h

Notice:

- The previous 78h in AL is gone
- Because AL is part of AX

IV. Move a DWORD into EAX

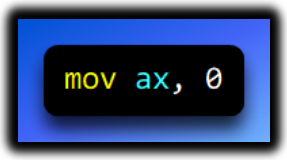


What happens:

All 32 bits are replaced - **EAX = 12345678h**

No overlap concerns here — the entire register is rewritten.

V. Zero AX only



What happens:

- Lower 16 bits → set to zero
- Upper 16 bits → **unchanged**

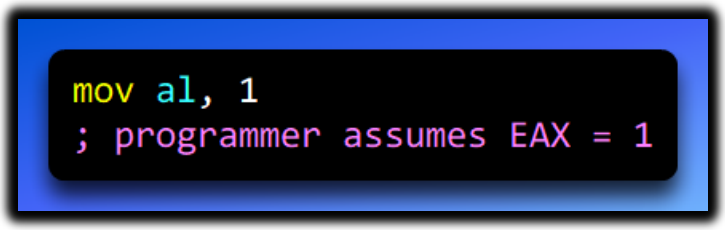
EAX = 12340000h

This is where people get surprised.

VI. Why this matters (the dangerous part)

Because partial writes **do not clear the rest of the register**, you can easily end up with **garbage in the upper bits** if you're not careful.

Example bug:



Reality: **EAX = ??????01h**

Unless EAX was cleared earlier, the upper bits contain whatever was there before.

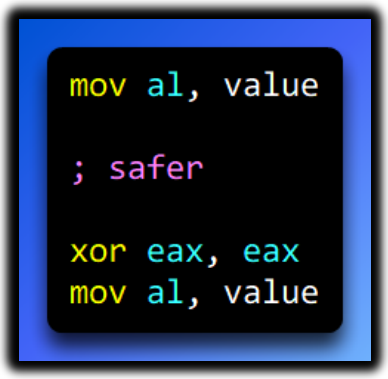
VII. Golden rules for overlapping registers

Rule 1: Smaller writes do not clear larger registers

- mov al, x → changes 8 bits
- mov ax, x → changes 16 bits
- mov eax, x → changes all 32 bits

Rule 2: If you care about the full register, write the full register

Instead of:



Now you *know* the upper bits are zero.

Rule 3: Overlap works both ways

- Writing to AX overwrites AL
- Writing to AL does **not** preserve meaningful values in AX
- Everything overlaps downward

VII. Why x86 works this way

This design exists for:

- backward compatibility (8-bit and 16-bit CPUs)
- performance
- flexibility

It's powerful — but sharp.

VIII. Big picture takeaway

AL, AX, and EAX are not separate registers.
They are different-sized windows into the same storage.

When you move smaller data into a larger register:

- it goes into the **lower portion**
- the **upper bits are untouched**

Understanding this prevents:

- subtle bugs
- incorrect comparisons
- broken arithmetic
- mysterious values

Once this clicks, you start *controlling* the CPU.

SIGNED AND ZERO EXTENSION

Table 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

Zero Extension (moving a smaller unsigned value into a larger register)

Zero extension is used when you move a **smaller unsigned value** (for example, 8-bit or 16-bit) into a **larger register** (like a 32-bit register), and you want the extra space to be filled with **zeros**.

This is important because unsigned values do **not** have a sign. So, when we make them bigger, we must not copy or invent a sign bit. We simply add zeros to the left.

The basic idea is simple:

1. First, clear the larger register (set it to zero).
2. Then, move the smaller value into the lower part of that register.
3. The upper bits stay zero, which is exactly what we want.

Example setup

Let's say we have a variable called count:

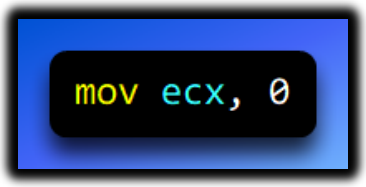
- count is a **16-bit unsigned integer**
- Its value is **1**
- In hexadecimal, that value is **0001h**

Assembly code example (Zero Extension)

```
mov ecx, 0        ; Clear the entire 32-bit ECX register
mov cx, count     ; Move the 16-bit value into the lower 16 bits of ECX
```

What happens step by step

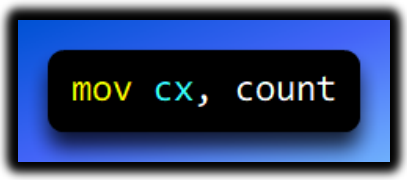
Step 1: Clear ECX



This sets all 32 bits of the ECX register to zero: **ECX = 00000000h**

This step is important because it guarantees that the upper 16 bits are zero before we move anything into ECX.

Step 2: Move the 16-bit value into CX



- CX is the **lower 16 bits** of ECX
- The value of count is 0001h
- This instruction copies 0001h into the lower 16 bits only

After this instruction:

- Upper 16 bits of ECX: 0000h
- Lower 16 bits of ECX: 0001h

So, the full 32-bit value of ECX becomes: **ECX = 00000001h**

We took a **16-bit unsigned value** and placed it into a **32-bit register**.

The extra 16 bits on the left were filled with **zeros**, not sign bits.

That is why this is called **zero extension**.

Final result (important correction)

- Hex value in ECX: 00000001h
- Decimal value in ECX: **1**

We copied the number **1** into ECX and padded it on the left with zeros.
The value stays **1**, not 16.

MOVZX and MOVSX Instructions

(Move with Zero Extension / Move with Sign Extension)

When moving **smaller values** (8-bit or 16-bit) into **larger registers** (16-bit or 32-bit), we often need to **extend** the value.
Intel provides two special instructions that do this automatically:

- **MOVZX** → for **unsigned values**
- **MOVSX** → for **signed values**

These instructions save us from manually clearing registers and make our code **shorter, safer, and easier to read**.

Instruction Forms

I. MOVZX (Zero Extension)

```
MOVZX reg32, reg/mem8
MOVZX reg32, reg/mem16
MOVZX reg16, reg/mem8
```

II. MOVSX (Sign Extension)

```
MOVSX reg32, reg/mem8
MOVSX reg32, reg/mem16
MOVSX reg16, reg/mem8
```

MOVZX — Move with Zero Extension

MOVZX copies a smaller unsigned value into a larger register and fills all the extra bits with zeros.

It does **not care about signs**.
It treats the value as always positive.

I. Example: Zero extension with MOVZX

Data section

```
.data
count WORD 1       ; 16-bit unsigned integer (0001h)
```

Code section

```
.code
movzx ecx, count
```

What happens step by step

- count is **16 bits** and equals 0001h
- ECX is **32 bits**
- MOVZX:
  - Copies count into the lower 16 bits of ECX
  - Fills the upper 16 bits with **zeros**

Final result: **ECX = 00000001h**

Decimal value: **1**

This is **zero extension** — the value is extended by adding zeros on the left.

MOVSX — Move with Sign Extension

MOVSX copies a smaller signed value into a larger register and preserves the sign.

- If the value is **positive**, it fills the upper bits with **0**
- If the value is **negative**, it fills the upper bits with **1**

This keeps the number mathematically correct after the move.

I. Example: Negative signed value

Data section and code section

```
.data
signedVal SWORD -16   ; 16-bit signed integer (FFF0h)

; code section

.code
movsx ecx, signedVal
```

What happens step by step

- signedVal = -16
- Hex value (16-bit): FFF0h
- Sign bit = 1 (negative)

After MOVSX: **ECX = FFFFFFF0h**

The upper bits are filled with **1s**, preserving the negative sign.

Important Question:

Do signed values have to be negative for MOVSX?

No. Absolutely not.

MOVSX works for **both positive and negative signed values**.

Example: Positive signed value

Data section and Code section

```
.data
signedVal SWORD 42     ; 16-bit signed integer (002Ah)

; code section

.code
movsx ecx, signedVal
```

What happens here?

- 42 in hex is 002Ah
- Sign bit = 0 (positive)

After MOVSX: **ECX = 0000002Ah**

The value stays positive, and the upper bits are filled with zeros.

✓ Sign preserved correctly

Key Rule for Sign Extension (Very Important)

When using **MOVSX**, always look at the **most significant bit (sign bit)**:

- **Sign bit = 0** → positive → extend with **zeros**
- **Sign bit = 1** → negative → extend with **ones**

Decimal	Hex Value	Sign Bit	Extended With
42	002Ah	0	Zeros (Positive)
-6	FFFAh	1	Ones (Negative)
-16	FFF0h	1	Ones (Negative)

Why do we need MOVZX if MOVSX exists?

Great question — and this is where many learners get confused.

Short answer: Because MOVSX can break unsigned values.

MOVZX — for unsigned data

- Always fills upper bits with **zeros**
- Safe for values like counters, sizes, indexes

Example:

```
movzx ecx, count
```

MOVSX — for signed data

- Copies the **sign bit**
- Correct for negative numbers
- Dangerous for unsigned values

Example problem:

```
count WORD 0FFFEh    ; unsigned = 65534
movsx ecx, count     ; WRONG!

; Result = ECX = FFFFFFFEh   -   Interpreted as -2
```

🚨 That's why **MOVZX exists**.

Final Summary (Simple and Clear)

- **MOVZX**
  - Used for **unsigned values**
  - Upper bits are filled with **zeros**
  - Prevents accidental negative values
- **MOVSX**
  - Used for **signed values**
  - Upper bits copy the **sign bit**
  - Preserves positive or negative meaning

👉 **Always choose based on the data type, not the instruction size.**

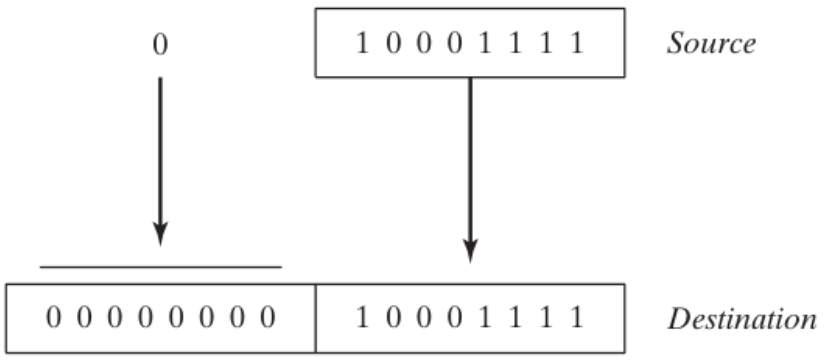
Movzx Repeated

```
.data
    byteVal BYTE 10001111b     ;decimal 143

.code
    movzx ax,byteVal           ; AX = 0000000010001111b
```

What is happening here:

FIGURE 4–1 Using MOVZX to copy a byte into a 16-bit destination.



MOVZX, which stands for Move with Zero Extension, is an instruction that lets us take a value that is stored in a smaller register or memory location and copy it into a bigger register.

When we do this, the extra space in the bigger register (the upper bits) is filled with zeros so that the value stays the same. Let's look at some examples using registers to understand how this works step by step.

1. Example: Register Operands

```
mov bx, 0A69Bh       ; BX = 0A69Bh
movzx eax, bx         ; EAX = 0000A69Bh
movzx edx, bl         ; EDX = 0000009Bh
movzx cx, bl          ; CX = 009Bh
```

In this example, we used the **MOVZX** instruction to copy smaller pieces of data from one place into bigger registers, making sure that any extra space is filled with zeros. Let's take it one step at a time:

1. **mov bx, 0A69Bh** – Here, we put the 16-bit value 0A69Bh into the BX register. Think of BX as a small container that can hold 16 bits (2 bytes).
2. **movzx eax, bx** – Now, we want to copy that 16-bit value from BX into the larger 32-bit register EAX. Because EAX is bigger, MOVZX fills the extra 16 bits at the top with zeros. After this, EAX contains 0000A69Bh.
3. **movzx edx, bl** – BL is the lower 8 bits of BX (just the last byte). We copy this into the 32-bit register EDX using MOVZX. The top 24 bits of EDX are filled with zeros, so the final value is 0000009Bh.
4. **movzx cx, bl** – Finally, we copy that same 8-bit value from BL into the 16-bit register CX. Since CX has space for 16 bits, the upper 8 bits are filled with zeros, giving 009Bh.

2. Example: Memory Operands

```
.data
    byte1 BYTE 9Bh
    word1 WORD 0A69Bh

.code
    movzx eax, word1        ; EAX = 0000A69Bh
    movzx edx, byte1        ; EDX = 0000009Bh
    movzx cx, byte1         ; CX = 009Bh
```

In this example, we used **MOVZX** again, but this time the data comes from memory instead of registers. We have two variables in memory: byte1 (8 bits) and word1 (16 bits). Here's what happens step by step:

1. **movzx eax, word1** – We take the 16-bit value stored in word1 and copy it into the 32-bit register EAX. Since EAX is bigger than word1, MOVZX fills the extra 16 bits at the top with zeros. After this, EAX contains 0000A69Bh.
2. **movzx edx, byte1** – Now we take the 8-bit value in byte1 and copy it into the 32-bit register EDX. The top 24 bits are filled with zeros, so EDX becomes 0000009Bh.
3. **movzx cx, byte1** – Finally, we copy the same 8-bit value from byte1 into the 16-bit register CX. The upper 8 bits of CX are filled with zeros, giving 009Bh.

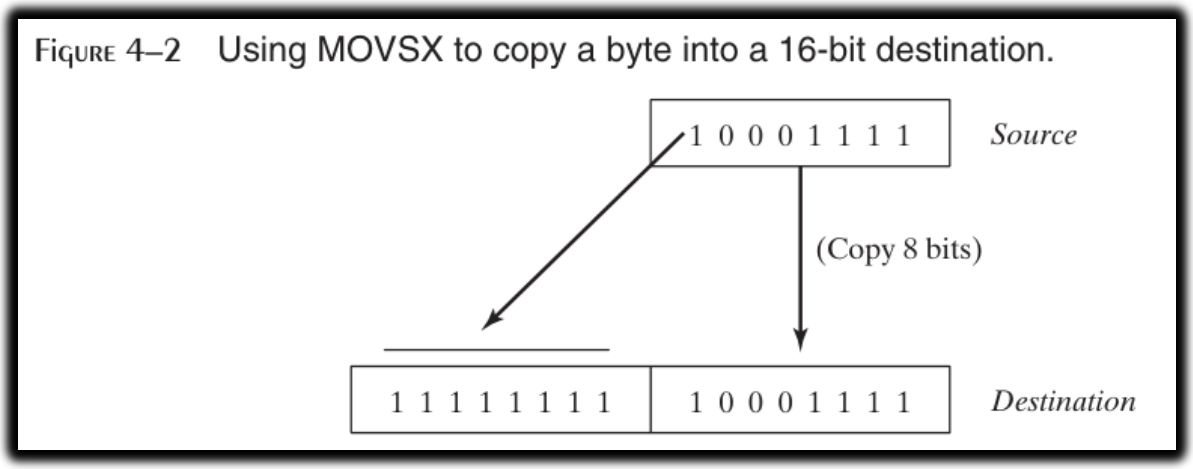
The important idea here is that **MOVZX always fills the extra bits with zeros** when moving smaller values into larger registers. This keeps the value correct and avoids accidentally changing the sign of the number.

MOVSX Repeated

```
;movsx

mov bx, 0A69Bh      ; BX = 0A69Bh
movsx eax, bx        ; EAX = FFFFA69Bh
movsx edx, bl        ; EDX = FFFFFF9Bh
movsx cx, bl         ; CX = FF9Bh
```

What is happening here?



Here, we are looking at **MOVSX**, which copies smaller values into larger registers while **preserving the sign** (positive or negative). We're using the hexadecimal value 0A69Bh as an example. Here's what happens step by step:

1. **mov bx, 0A69Bh** – This puts the 16-bit value 0A69Bh into the BX register. The first digit "A" in hexadecimal means that the highest bit (the **sign bit**) is 1, so this value is considered **negative** in signed representation.
2. **movsx eax, bx** – We copy the 16-bit value in BX into the 32-bit register EAX. Because we are using MOVSX, it **keeps the sign the same**. The upper 16 bits of EAX are filled with ones (F in hex) to preserve the negative sign. After this, EAX = FFFFA69Bh.
3. **movsx edx, bl** – Now we take the lower 8 bits of BX (called BL) and copy them into the 32-bit EDX register. MOVSX again preserves the sign. The upper 24 bits of EDX are filled with ones, so EDX = FFFFFF9Bh.
4. **movsx cx, bl** – Finally, we copy the 8-bit BL into the 16-bit register CX. The upper 8 bits of CX are filled with ones because the sign bit is 1, giving CX = FF9Bh.

Key point: The "A" in 0A69Bh shows that the number is negative in signed form. MOVSX makes sure that when we copy smaller numbers into larger registers, the **sign is preserved** by filling the extra bits with ones for negative numbers.

LAHF AND SAHF INSTRUCTIONS

The **LAHF** (Load AH from Flags) and **SAHF** (Store AH into Flags) instructions are used to **move specific CPU flags to and from the AH register**.

These instructions are useful when you want to **save or restore the status of certain flags** without affecting the rest of the EFLAGS register.

LAHF — Load AH from Flags

I. What it does

- The **LAHF** instruction takes the **low byte of the FLAGS register** (specifically the SF, ZF, AF, PF, and CF flags) and **copies it into the AH register**.
- This allows you to **save the state of these flags** for later use, such as storing them in memory or comparing them without changing the CPU state.

II. Flags included in LAHF

Flag	Meaning	Condition for Set (1)
SF	**Sign Flag**	The Most Significant Bit (MSB) of the result is 1 (Negative).
ZF	**Zero Flag**	All bits of the result are 0.
AF	**Auxiliary Flag**	Carry/Borrow out of bit 3 (Used for BCD arithmetic).
PF	**Parity Flag**	The lowest byte of the result contains an **even** number of 1s.
CF	**Carry Flag**	An unsigned overflow occurred (carry out or borrow in).

Note: LAHF **does not load all EFLAGS**, only these selected flags.

III. Example: Using LAHF

```
; Save the state of SF, ZF, AF, PF, and CF into a variable

lahf                ; Load flags into AH
mov [savedFlags], ah ; Store AH into memory
```

Or

```
.data
    saverflags BYTE ?   ;Define a variable to store the flags

.code
    lahf                ;load flags into AH
    mov saverflags, ah  ;save them in the 'saverflag' variable
```

Step-by-step explanation:

1. lahf copies the **low byte of FLAGS** into the AH register.
2. mov [savedFlags], ah stores the value of AH into a memory variable called savedFlags.
3. Later, you can **restore these flags** using SAHF.

In this example, after executing LAHF, the AH register contains the values of the specified flags, and you can save these values in the saveflags variable for future reference.

SAHF (Store AH into Flags):

The SAHF instruction works in the opposite direction. It copies the value from the AH register into the low byte of the EFLAGS (or RFLAGS) register.

This allows you to restore saved flag values.

Here's an example of how to use SAHF to retrieve saved flag values from a variable:

```
.data
    saveflags BYTE ?  ; Variable containing saved flags

.code
    mov ah, saveflags  ; Load saved flags into AH
    sahf               ; Copy AH into the Flags register
```

EFLAGS / RFLAGS Register

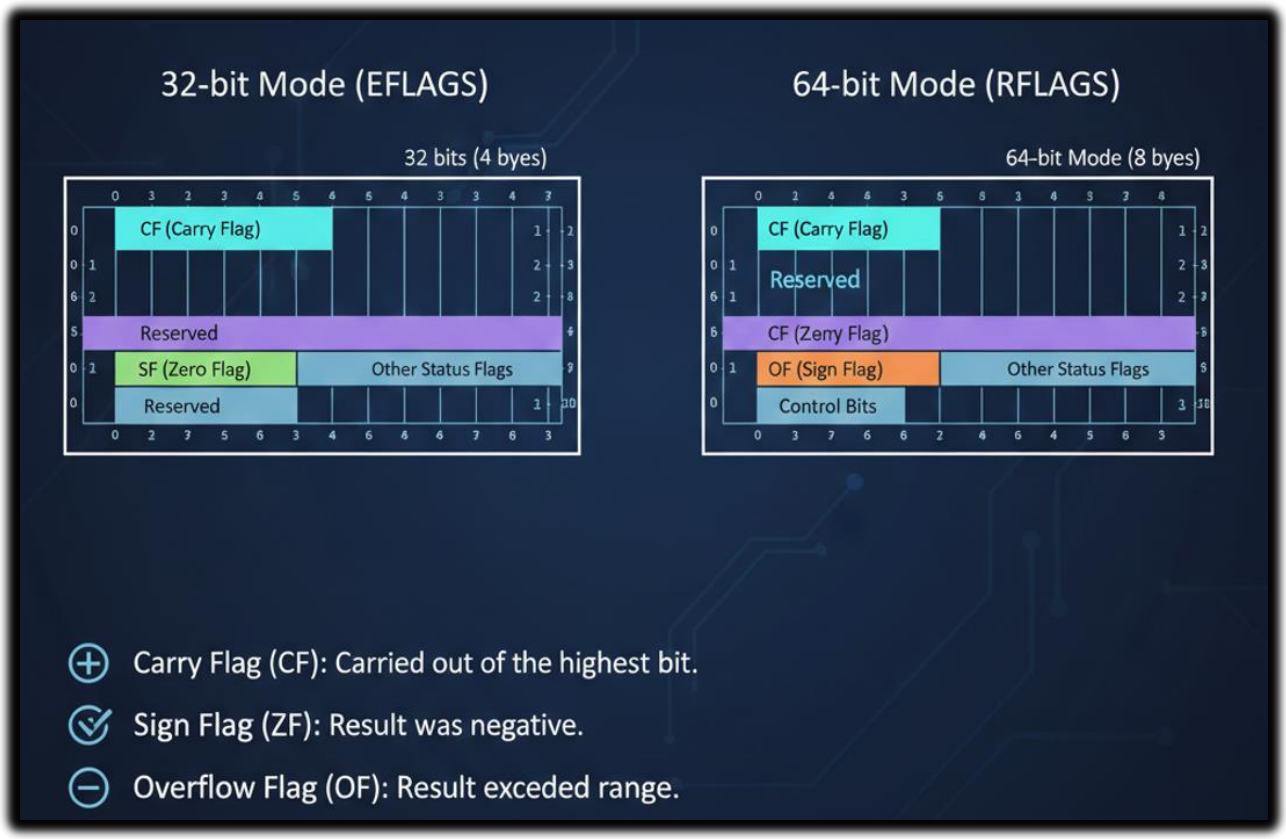
The **EFLAGS** register (called **RFLAGS** in 64-bit mode) is like a **control panel for the CPU**.

- It's **32 bits (4 bytes)** in 32-bit mode, and **64 bits (8 bytes)** in 64-bit mode.
- It **stores status flags** that tell the CPU what happened after each instruction.
- It also has **control bits** that affect how the CPU works.

Some of the important **flags** you'll see in EFLAGS/RFLAGS are:

- **Carry Flag (CF):** Tells if a calculation carried out of the highest bit.
- **Zero Flag (ZF):** Tells if the result of a calculation was zero.
- **Sign Flag (SF):** Tells if the result was negative.
- **Overflow Flag (OF):** Tells if a calculation overflowed the allowed range.
- ...and many others that help the CPU **track what's going on**.

In **64-bit mode**, the **RFLAGS register** works the same way as EFLAGS, just **bigger (64 bits)**.



XCHG (Exchange) Instruction

The XCHG instruction in x86 assembly is used to **swap the contents of two operands**.

Unlike doing a manual swap using a temporary register, XCHG can handle this directly, which makes your code shorter, cleaner, and sometimes faster.

There are **three main variants**:

1. XCHG reg, reg – exchanges contents between two registers.
2. XCHG reg, mem – exchanges contents between a register and a memory location.
3. XCHG mem, reg – exchanges contents between a memory location and a register.

Let's break each one down, with examples and practical notes.

1. XCHG reg, reg

What it does: Swaps the contents of two registers.

- No memory access is required—everything stays inside the CPU registers.
- Useful when you need to reorder values quickly, e.g., swapping two counters.

Example:

```
MOV AX, 5       ; AX = 5
MOV BX, 10      ; BX = 10
XCHG AX, BX     ; Swap AX and BX
```

Result after execution:

- AX = 10
- BX = 5

Key Points:

- Fast, because it only involves registers.
- Atomic on most processors, which means it can safely be used in some synchronization tasks (like implementing locks).

---

2. XCHG reg, mem

What it does: Swaps the value of a register with a value in memory.

- Now the CPU has to read from and write to RAM, which is slower than register-only operations.
- This is very handy when you need to update a variable without using an extra temporary register.

Example:

```
MOV BX, 20          ; BX = 20
XCHG BX, [var1]     ; Swap BX with memory location 'var1'
```

Result:

- BX now contains the previous value of var1.
- var1 now contains 20.

Why it matters:

- Helps in **swapping array elements** directly in memory.
- Can also be used in simple **locking mechanisms**, because XCHG is atomic when working with a register and memory.

---

3. XCHG mem, reg

What it does: Swaps a memory location with a register. This is technically the same as XCHG reg, mem because the CPU treats memory-register swaps identically, but conceptually it's about **memory being the first operand**.

Example (swapping two memory locations using a temporary register):

```
MOV AX, [val1]   ; AX = val1
XCHG AX, [val2]  ; Swap AX with val2
MOV [val1], AX   ; Move AX (original val2) back to val1
```

Result:

- val1 and val2 are swapped.
- AX temporarily holds the value during the swap.

Why it matters:

- Direct memory-to-memory swap isn't supported on x86, so this pattern (using a register as a "bridge") is essential.
- Shows how XCHG can be combined with MOV to manipulate memory efficiently.

Practical Tips & Takeaways

- **Atomicity:** XCHG is inherently atomic when one operand is a memory location. This makes it useful in multithreading scenarios.
- **Performance:** Register-register swaps are fastest; memory involvement slows things down.
- **Common Uses:**
  - Swapping variables (obvious!)
  - Sorting algorithms (like bubble sort)
  - Implementing simple locks or semaphores in low-level concurrent code