FIRST ASSEMBLY PROGRAM 👶💻

We are done with theory. Let's write code.

We will look at a simple program that takes two numbers, adds them together, and saves the result in a **Register** (a tiny, super-fast storage slot inside the CPU).

The Basic Structure

```
.code           ; Tell the assembler this is the executable code section
main PROC       ; Start of the main procedure (the entry point)

    MOV eax, 5 ; Move the integer 5 into the EAX register
    ADD eax, 6 ; Add 6 to the value inside EAX (5 + 6 = 11)

    INVOKE ExitProcess, 0 ; Call Windows to stop the program neatly
main ENDP       ; End of the main procedure
```

main PROC: This marks the beginning. Think of PROC (Procedure) as the start of a function in Python or C++. It tells the computer, "Start executing here."

MOV eax, 5: This is the assignment operator. We are putting the value 5 into the register named **EAX**.

Note: MOV stands for "Move," but it really means "Copy." The 5 doesn't disappear from where it came from; it just gets copied into EAX.

ADD eax, 6: The math happens here. The CPU takes the value currently in EAX (which is 5), adds 6 to it, and stores the result (11) back into EAX.

INVOKE ExitProcess, 0: This is a call to the Operating System (Level 2!). It tells Windows, "I am done here, shut it down." Without this, the program might crash or hang.

main ENDP: The "End Procedure" marker. It closes the block we opened with main PROC.

Introducing Variables and Segments

Real programs need to store data, not just hard-coded numbers.

To do this, we divide our program into **Segments**.

Think of segments as different rooms in a house, each with a specific purpose.

Here is the upgraded program with variables:

```
.data                ; The DATA segment (Variables live here)
    sum DWORD 0      ; Declare a variable named 'sum', size 32-bits, value 0

.code                ; The CODE segment (Instructions live here)
main PROC
    MOV eax, 5
    ADD eax, 6
    MOV sum, eax     ; Move the result (11) from EAX into the variable 'sum'

    INVOKE ExitProcess, 0
main ENDP
```

I. The .data Segment

This is where you declare variables. It is a specific area in memory reserved just for storage.

sum DWORD 0:

- **Name:** sum
- **Size:** DWORD (Double Word). This means 32 bits.
- **Value:** 0 (The initial value).

II. The .code Segment

This is where your instructions (logic) live. This area is usually "Read-Only" so you don't accidentally overwrite your own program code while it's running.

III. The .stack Segment

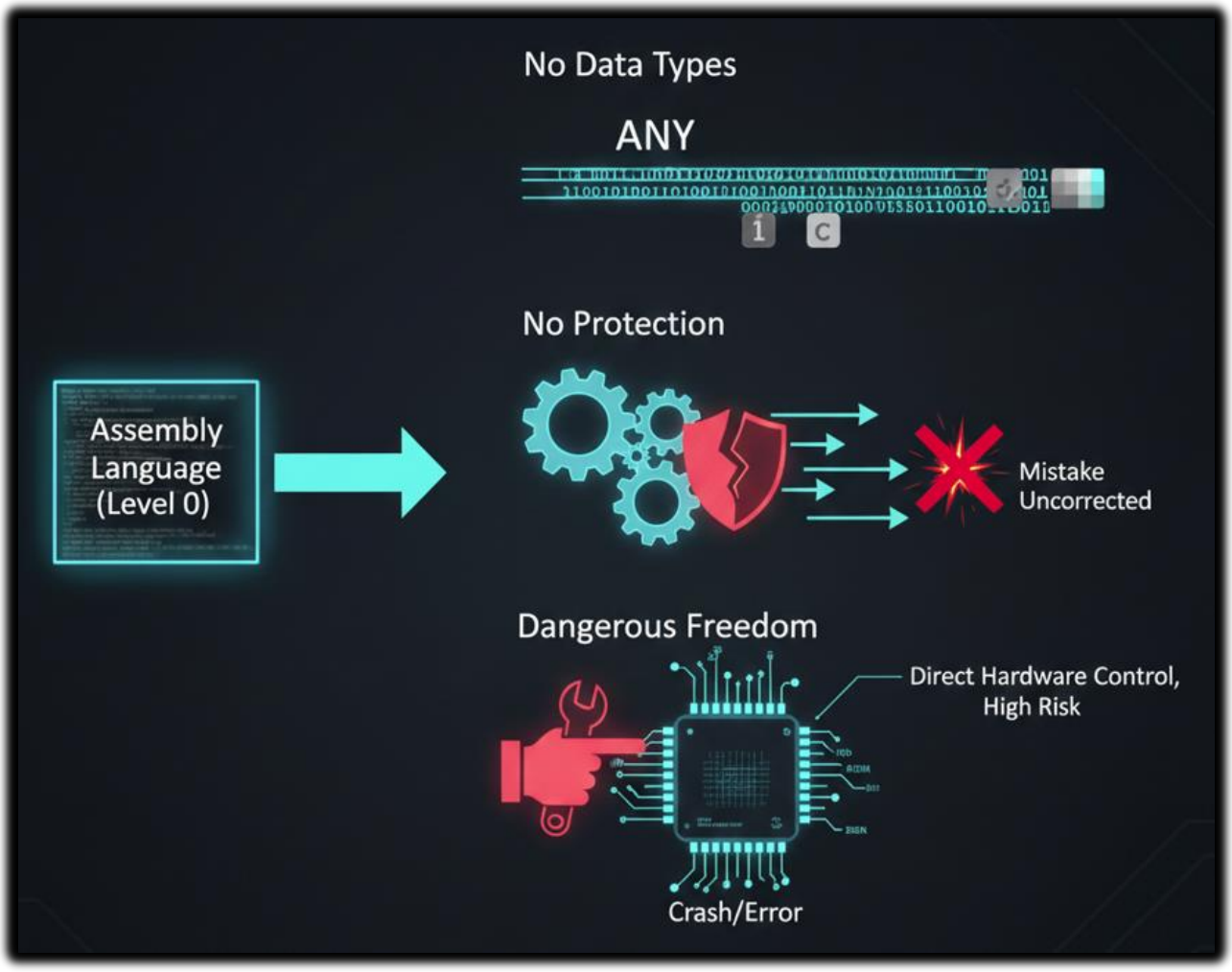
(Mentioned briefly) We will cover this later, but this is a scratchpad area for temporary storage during function calls.

The Wild West of Data Types

In high-level languages like C++ or Java, data types are strict. You must clearly say whether something is an integer, a floating number, or a character.

If you try to store a letter in an integer, the compiler immediately throws an error and stops you.

Assembly language works very differently. Assembly does not enforce data types at all. It does not protect you or correct your mistakes.



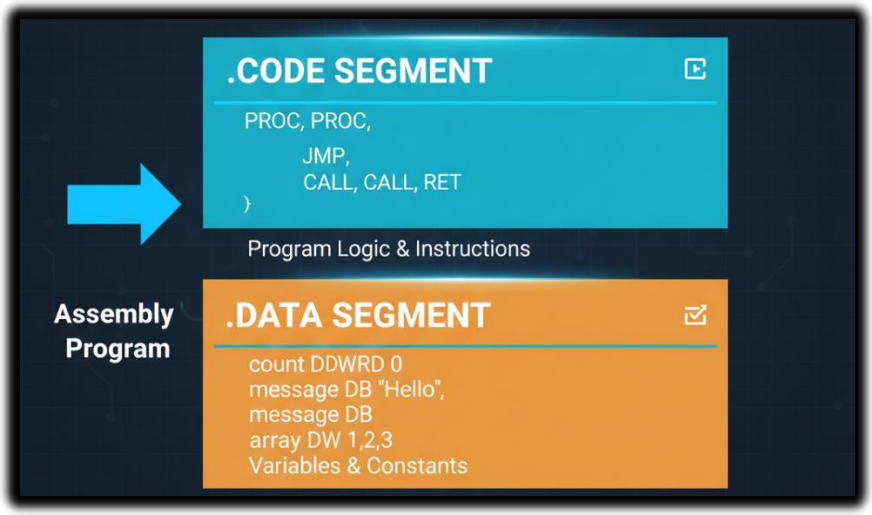
In Assembly, **size is what matters**, not meaning. When you write something like DWORD, you are only telling the computer to reserve **32 bits of memory**. You are not saying what kind of data will be stored there.

There is **no type checking**. The CPU does not know or care whether those 32 bits represent a number, a letter, or a memory address. It will process the data exactly as you tell it to.

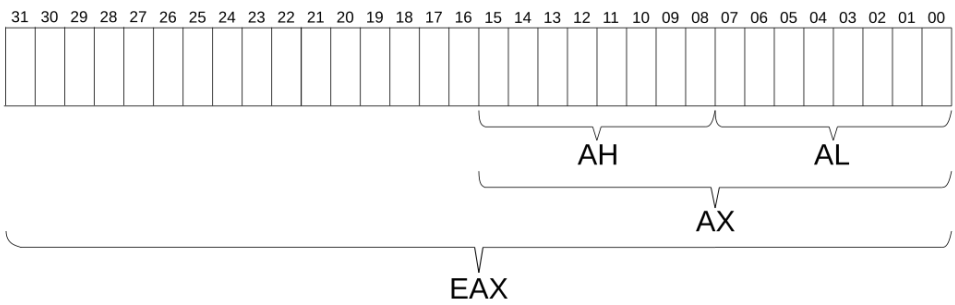
This gives you **total control**, but also total responsibility. You can treat a number like a character or an address if you want, and Assembly will allow it. If you make a mistake, the program will crash or behave incorrectly. There are no safety rails.

Big Idea to Remember

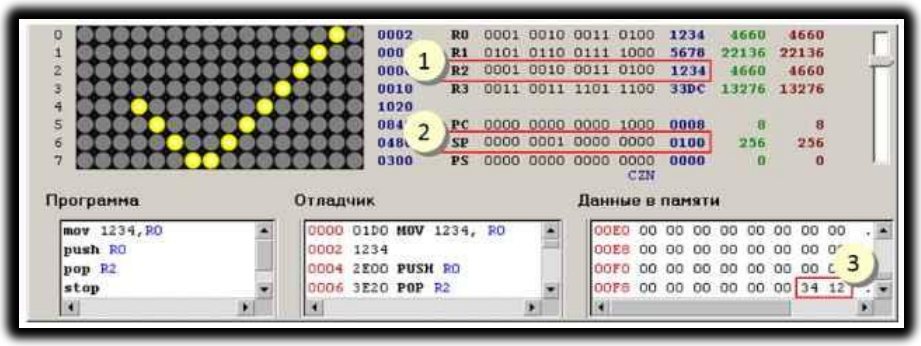
Memory is organized into **segments**. The .code segment holds the program logic, while the .data segment holds variables.



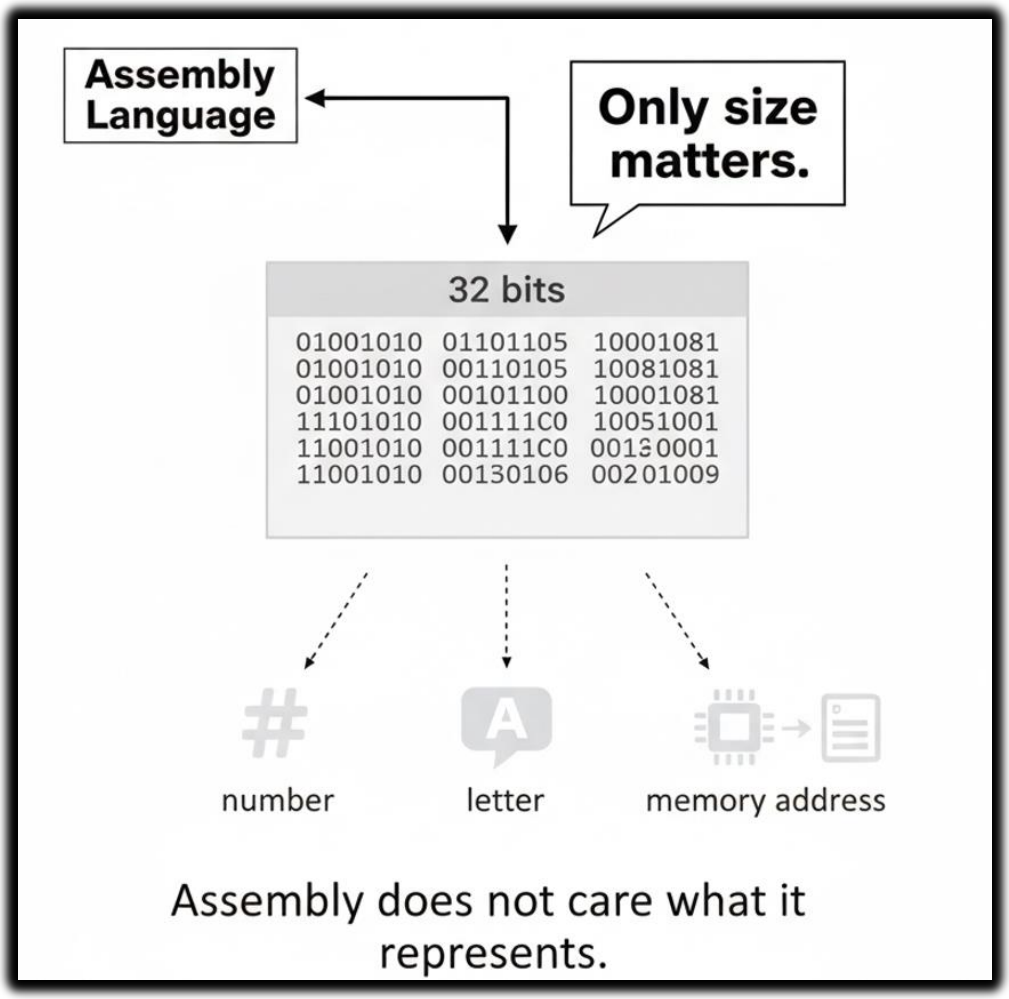
Registers, such as EAX, are the CPU's working space. They temporarily hold data while the processor performs operations.



Instructions tell the CPU what to do. MOV copies data, ADD performs math, and INVOKE communicates with the operating system.



Assembly does not understand data types. It only understands **how many bits** something uses, not what those bits are meant to represent.



INTEGER LITERALS

An **integer literal** (also called an **integer constant**) is a number written directly in a program.

An integer literal can have:

- an **optional sign** (+ or -)
- **one or more digits**
- an **optional radix letter** at the end that tells us what **base** the number is written in

General form:

[{+ | -}] digits [radix]

Examples

- 26
  This is a valid integer literal.
  It has **no radix letter**, so we assume it is **decimal (base 10)**.
- 26h
  This means **26 in hexadecimal (base 16)**.
- 1101
  This is treated as **decimal**, not binary, because there is **no radix letter**.
- 1101b
  The b tells us this number is **binary (base 2)**.

So, **without a radix letter, the number is always assumed to be decimal**.

Radix Table

Here is the table:

NAME	BASE	NOTATION
Decimal	`10`	`d` or no letter
Hexadecimal	`16`	`h`
Octal	`8`	`o` or `q`
Encoded Real	`N/A`	*Special Format*
Binary	`2`	`b`
Binary (Alternate)	`2`	*Implementation-specific*

Important note about Encoded Real

Encoded Real does not have a specific base value.
It is a binary format used to represent floating-point numbers, not normal integers.

Examples of Integer Literals with Radixes

Each line below shows an integer literal, followed by a comment explaining its base:

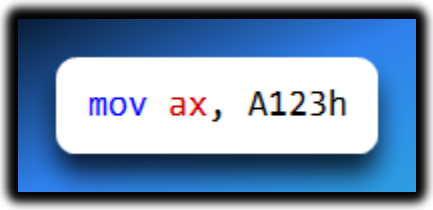
```
26         ; decimal
26d        ; decimal
11011011b  ; binary
42q        ; octal
42o        ; octal
1Ah        ; hexadecimal
0A3h       ; hexadecimal
```

HEXADECIMAL BEGINNING WITH A LETTER

In assembly language, a **hexadecimal number that starts with a letter** must have a **leading zero**.

Why?
Because the assembler might think the value is a **name (identifier)** instead of a number.

Example that causes an error

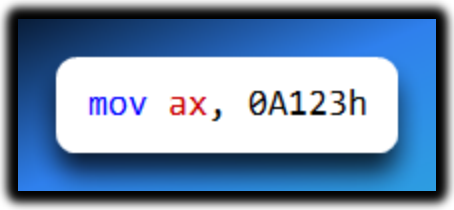


This causes an **undefined symbol error**.

Why this happens:

- The value starts with the letter **A**
- The assembler assumes A123h is the **name of a variable or label**
- Since no such name exists, it throws an error

Correct version (with leading zero)



Now it works correctly.

The **leading zero** tells the assembler:

“This is a hexadecimal number, not an identifier.”

Rule to remember

👉 **Any hexadecimal literal that begins with a letter must start with 0.**

Examples:

- 0A3h ✅
- 0FFh ✅
- A3h ❌

CONSTANT INTEGER EXPRESSIONS

A **constant integer expression** is a math expression made using:

- integer literals
- arithmetic operators

These expressions are **calculated at assembly time**, not while the program is running.

From now on, we'll just call them **integer expressions**.

Important rule

The final result:

- must be an **integer**
- must fit in **32 bits**
- valid range:
  0 to FFFFFFFFh

Arithmetic Operators and Precedence

Operator **precedence** means the order in which operations are done.

Here is the table, from **highest priority** to **lowest priority**:

Operator Precedence

OPERATOR	NAME	PRECEDENCE LEVEL
`()`	Parentheses	1 (Highest)
`+ -`	Unary plus, unary minus	2
`* /`	Multiply, divide	3
`mod`	Modulus	3
`+ -`	Add, subtract	4 (Lowest)

What does unary mean?

Unary means the operator works on **one value only**.

Examples:

- -5 → unary minus (one number)
- +3 → unary plus (one number)

This is different from:

- 5 - 2 → subtraction (two numbers)

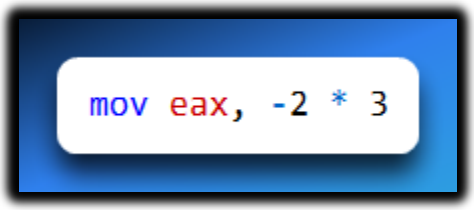
Unary operators explained

- **Unary plus (+)**
  Just returns the value +5 → 5
- **Unary minus (-)**
  Changes the sign -5 → negative five

Why unary has higher precedence

Unary plus and minus are done **before** multiplication and division.

Example:

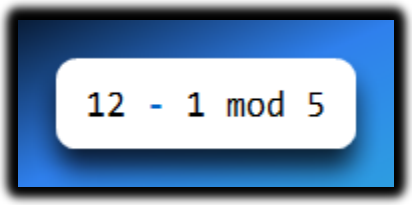


What happens:

1. -2 is evaluated first (unary minus)
2. Then -2 * 3
3. Result is -6

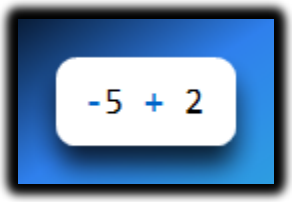
Operator Precedence Examples

Multiply first, then add. Result: 14



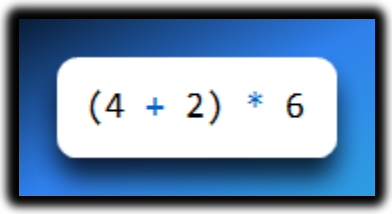
1 mod 5 first → 1

Then subtraction. Result: 11



Unary minus first → -5

Then add. Result: -3



- Parentheses first
- Then multiply
- Result: 36

Using Parentheses (Best Practice)

Even if you know the rules, **use parentheses**.

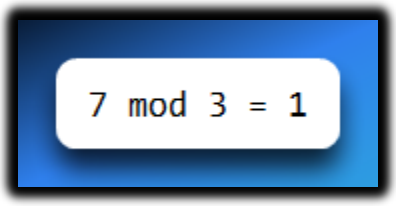
Why?

- Makes expressions easier to read
- Prevents mistakes
- You don’t have to remember precedence rules

Modulus Operator (mod or %)

The **modulus operator** gives the **remainder** of a division.

Example:



That's all it does—no magic.

REAL NUMBER LITERALS

A **real number literal** is just a number that can have:

- a **decimal point**
- or a **fraction**
- or a **very large / very small value**

These are also called **floating-point numbers**.

In assembly, real numbers can be written in **two ways**:

1. **Decimal reals** (the normal way humans write numbers)
2. **Encoded reals** (hexadecimal form, using IEEE format)

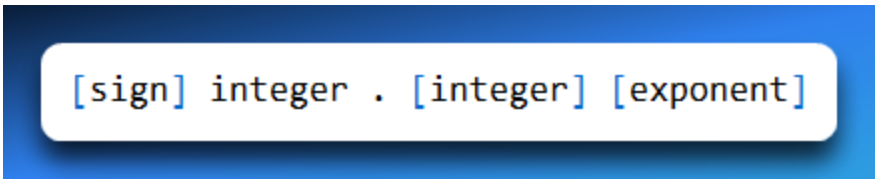
Decimal Real Numbers

A **decimal real** looks like a normal decimal number.

A **decimal real number** is a number written in **base-10 (decimal) notation**, the same format used in everyday arithmetic.

It represents a value on the **real number line** and may include a fractional part and, optionally, an exponent. Examples include 3.14, -0.5, and 6.02×10^{23}.

General form:



Let's break that into plain English.

A decimal real number can be broken into several components. Some parts are **required**, while others are **optional**, depending on how the number is written.

Parts of a decimal real

⭐ The **sign** indicates whether the number is positive or negative.

- Represented by + or -
- If no sign is written, the number is assumed to be positive
- The sign applies to the entire value of the number

Examples:

- +7.25 → positive
- -4.6 → negative
- 9.1 → implicitly positive

⭐ The **integer part** (also called the whole number part) is the sequence of digits **to the left of the decimal point**.

- Represents the whole units of the number
- Can be 0 if the value is less than 1
- Must contain at least one digit if a decimal point is present

Examples:

- 123.45 → integer part is 123
- 0.75 → integer part is 0
- -8.9 → integer part is 8

☆ The **decimal point** separates the **integer part** from the **fractional part**.

- Indicates that digits to the right represent fractions of a whole
- In decimal real numbers, a dot (.) is used (not a comma)
- Without a decimal point, the number is an integer, not a decimal real

Example:

- In 45.67, the dot separates 45 and 67

☆ The **fractional part** consists of digits **to the right of the decimal point**.

- Represents values less than one (tenths, hundredths, thousandths, etc.)
- Each digit has a place value based on powers of 10
- Can be omitted if the number is a whole number

Examples:

- 3.14 → fractional part is 14
- 10.0 → fractional part is 0
- 6. → fractional part omitted (still valid in many contexts)

☆ The **exponent** is used in **scientific notation** to scale the number by a power of 10.

- Written using $\times 10^n$ or e notation (e.g., 1.5e3)
- Allows compact representation of very large or very small numbers
- The exponent indicates how many places the decimal point is shifted

Examples:

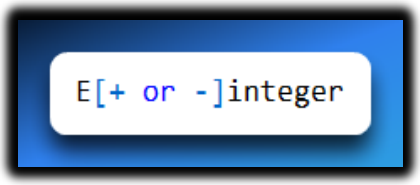
- 6.02×10^{23} → very large number
- 3.1×10^{-4} → very small number
- 7.5e2 → same as 750

⭐ **Why Decimal Reals Are Used**

Decimal real numbers are especially useful because they:

- Accurately represent **fractions and continuous values**
- Are intuitive and easy for humans to read
- Can represent **very large or very small quantities** when combined with exponents
- Are widely used in **science, engineering, finance, and computing**

Exponent format



E [+ or -] integer

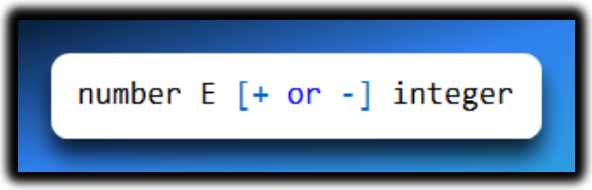
The exponent means:

"Multiply this number by 10 raised to some power."

I. What "Exponent format" means

Exponent format is a **shortcut way of writing big or small decimal numbers**.

It looks like this:



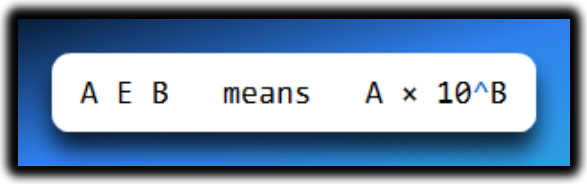
Eg 44.2E5

This **does NOT mean a new kind of number**.
It simply means:

Take the number and multiply it by 10 raised to a power

What the E actually means

The letter **E** stands for **"× 10 to the power of"**. So:



Examples:

- E5 means $\times 10^{5}$
- E-3 means $\times 10^{-3}$

How to Think About Exponents

Golden Rule (memorize this)

👉 **The exponent never changes the digits.**
👉 **It only moves the decimal point.**

That's it. Nothing else.

Step-by-Step Examples (Slow and Clear)

Example 1: **2.**

- Means 2.0
- The decimal point is present
- Any number with a decimal point is a **real number**
- Value = **2**

Example 2: **+3.0**

- + means positive
- Same value as 3.0
- Value = **3**

Example 3: **-44.2E+05** (this looks scary, but it's not)

Step 1: Ignore the sign for now. Start with **44.2**

Step 2: Understand the exponent - **E+05** means **$\times 10^5$**

So, we are doing: **44.2×10^5**

Step 3: Move the decimal point

- Power is **+5**
- Move the decimal **5 places to the right**
- 44.2 → 4,420,000

Step 4: Apply the sign – The original sign was negative

✅ Final answer: **-4,420,000**

Example 4: **26.E5** (this confuses many beginners)

Step 1: Look carefully at the number - **26.**

There are **no digits after the decimal point**.

👉 This is allowed.
👉 It is automatically assumed to be: **26.0**

Step 2: Apply the exponent - **E5** means **$\times 10^5$**

So, 26.0 * 10^5

Step 3: Move the decimal point 5 places to the right **- 26.0 → 2,600,000**

✅ Final answer: **2,600,000**

"But there are no digits after the dot!"

That's okay.

- 26. means 26.0
- Missing fractional digits are assumed to be zero

So: **26.E5 = 26.0 × 10^5**

This is **100% valid**.

Another Example: **44.2E05**

- E05 still means $\mathbf{10^5}$
- Leading zeros in the exponent do not change the value

So: **44.2E05** = $44.2 \times 10^5 = 4{,}420{,}000$

⭐ 26.E5 → valid
⭐ 44.2E05 → valid

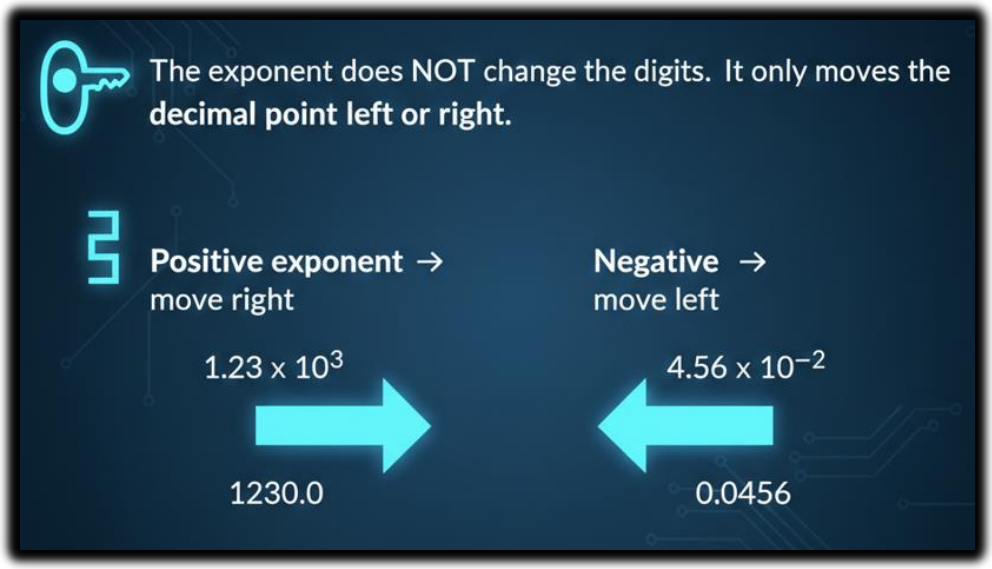
Both are **correct scientific notation**.

The "Aha" Idea (Most Important Part)

🔑 The exponent does NOT change the digits.
🔑 It only moves the decimal point left or right.

- Positive exponent → move right
- Negative exponent → move left

Once this clicks, exponent format becomes easy.



Encoded Real Numbers (Beginner Explanation)

Why this exists?

Humans and computers do not store numbers the same way.

Humans write numbers like: 1.0

- Computers cannot store decimals directly
- Computers store numbers as binary patterns (0s and 1s)

An **encoded real number** is: A real number converted into a binary pattern so the computer can store and process it.

An **encoded real** is a real number that has been:

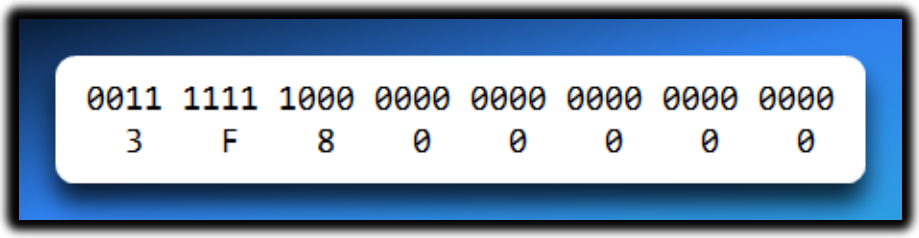
1. Converted into **binary**
2. Stored using a **fixed standard format**
3. Written in **hexadecimal** to make it easier for humans to read

This standard format is called: **IEEE floating-point format.**

Why Hexadecimal Is Used

Binary numbers are very long and hard to read:

So we group the bits into chunks of 4 and write them in **hexadecimal**:



That gives: **3F800000**

Important Idea (Very Important)

⚠ **3F800000 is NOT a normal number**

It does **not** mean "three million something".

It is: A **code** that represents the real number **1.0**

Humans vs Computers (Clear Comparison)

They represent the **same value**, just in different forms.

The r at the End (Assembler Hint)

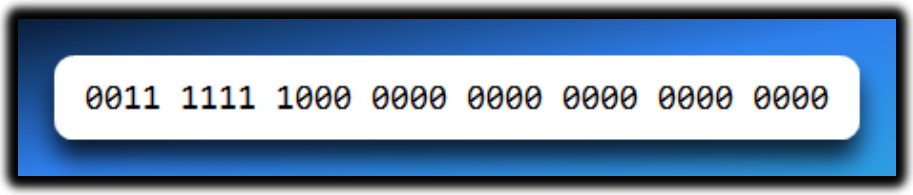
When writing encoded reals in assembly language, you may see: **3F800000r**

The **r** tells the assembler:
"This hexadecimal value is an encoded real number, not an integer."

Without the r, the assembler would treat it as a normal hex integer.

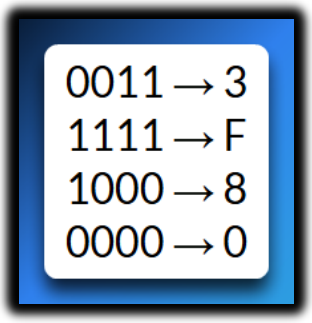
Example 1: Encoded Real for 1.0

Step 1: Binary representation



This binary pattern follows the **IEEE 32-bit floating-point layout**.

Step 2: Convert to hexadecimal - Group bits into 4s.



Final hex: **3F800000**

Step 3: Mark it as a real number

3F800000r

This tells the assembler:

"Store the real number **1.0** using IEEE floating-point encoding."

Summary

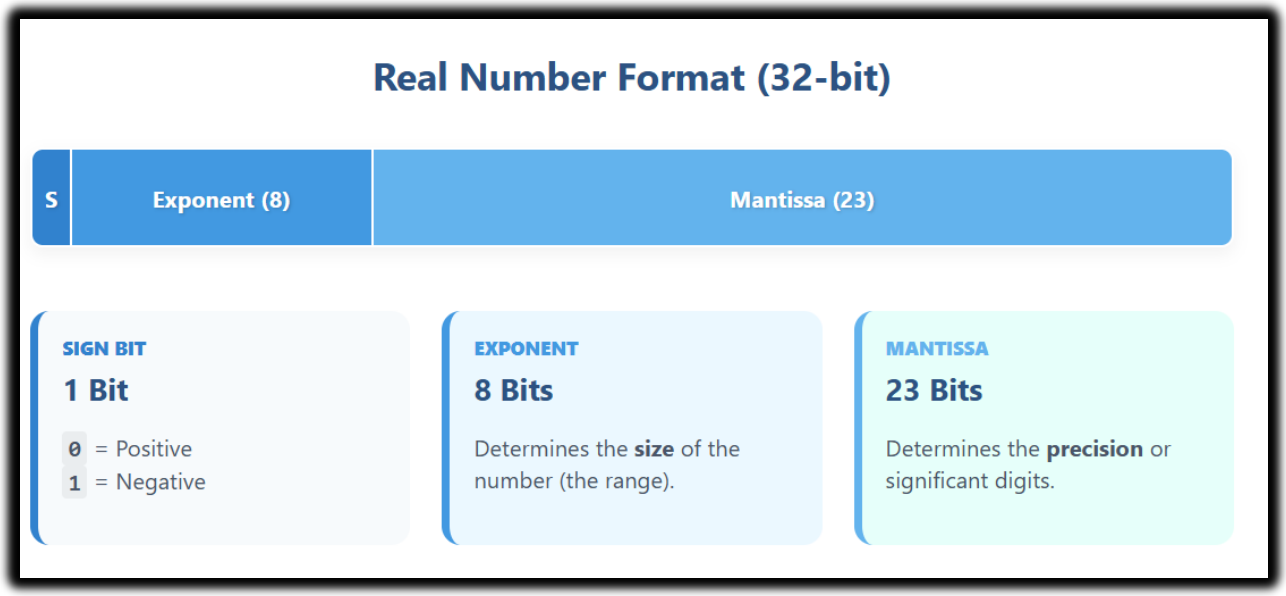
An **encoded real** is how a **computer stores a real number**

- It is written in **hexadecimal**
- It follows the **IEEE floating-point format**
- The hex value is a **bit pattern**, not a normal number
- The suffix **r** tells the assembler it is a real number

💡 **Encoded reals are not numbers — they are instructions for how the computer should interpret bits as a real value.**

IEEE Floating-Point (Short Real)

A **short real** uses **32 bits**, split like this:



Example 2: Decimal +1.0

Binary representation:

0 01111111 00000000000000000000000

Breakdown:

- 0 → positive number
- 01111111 → exponent for 1.0
- 000... → mantissa

Converting to hexadecimal

Group bits into 4s: **0011 1111 1100 0000 0000 0000 0000 0000**

Convert each group to hex: **3FC00000**

So, the encoded real is: **3FC00000**

Important note (and a relief)

We **won't use real-number constants for a while**.

Why?

- Most x86 instructions work with **integers**
- Floating-point math is more advanced

You'll come back to this later (Chapter 12), when it actually makes sense and feels useful.

Big-picture summary (don't skip this)

- **Decimal reals** → for humans
  (3.0, -44.2E5, 26.E5)
- **Encoded reals** → for the computer
  (3F800000r, IEEE format)
- You are **not expected to memorize** the binary layouts right now
- Just understand **what they are**, not how to build them by hand

---

CHARACTER LITERALS

A **character literal** is **one single character** written inside **single quotes** or **double quotes**. Examples: **'a'**, **"d"**

How characters are stored

Even though a character looks like a letter, the computer **stores it as a number**.

This number comes from the **ASCII table**.

Example:

- 'A'
  - ✓ ASCII value (decimal): 65
  - ✓ ASCII value (hex): 41h

So, when you write: 'A'

What actually goes into memory is: 65 or 41h

Important Reminder: Characters Are Just Numbers

The core idea (say this slowly)

👉 **A computer does not understand letters or symbols.**
👉 It only understands **numbers** (stored in binary).

So, when you see a character like A, the computer actually stores **a number** that *stands for* A.

When we say: **"Characters are not magic inside the computer"**

It means:

- The computer does **not** store the shape of the letter
- It does **not** store meaning
- It stores a **number code**

Example: **'A' is stored as the number 65**

The computer treats 65 as just a number.
Humans *interpret* that number as the letter A.

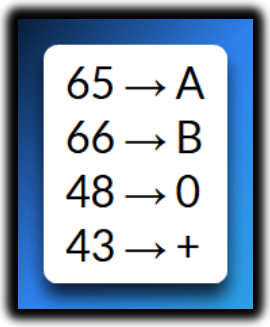
Why We Need a Table (ASCII)

Because characters are just numbers, everyone must agree on:

"Which number represents which symbol?"

That agreement is called **ASCII**.

The **ASCII table** is simply a lookup chart that says:



...and so on.

Nothing more complicated than that.

What the ASCII Table Contains (With Meaning)

1. Letters

Numbers assigned to alphabet characters.

Examples:

- A → 65
- a → 97

Uppercase and lowercase have **different numbers**.

2. Digits

Characters that *look* like numbers, but are still characters.

Examples:

- '0' → 48
- '1' → 49

⚠ Important: **'5' ≠ 5**

- '5' is a character
- 5 is a numeric value

3. Symbols

Punctuation and special characters.

Examples:

- + → 43
- # → 35
- @ → 64

4. Control Characters

Characters that **do not print anything**, but control behavior.

Examples:

- New line
- Tab
- Backspace

They tell the computer **how to format text**, not what to display.

"You're Expected to Recognize Common Ones"

This does **not** mean memorise the whole ASCII table.

It means:

- Know a **few important examples**
- Understand the **idea**, not the entire list

Common ones to recognize:

That's usually enough for exams and understanding code.

💡 **Characters are just numbers.**
💡 **ASCII is the dictionary that maps numbers to symbols.**
💡 **The computer only sees numbers — humans see letters.**

STRING LITERALS

A **string literal** is **more than one character** written inside quotes.

It can include:

- letters
- numbers
- symbols
- spaces

Examples:

Notice:

- Spaces **inside the quotes count**
- '4096' is a string, not a number
- ' 4096 ' includes spaces before and after

How strings are stored in memory

A string is stored as:

a **sequence of bytes**, one byte per character

Each character becomes its **ASCII integer value**.

Example: "ABCD"

Characters: A B C D

ASCII hex values: 41h 42h 43h 44h

So, the string "ABCD" is stored as these **four bytes in memory**.

Why characters and strings are stored as integers

This is the key idea 👇

A computer's memory can **only store numbers**.

- Memory is made of **bits**
- Bits represent **binary numbers**
- So, everything must become a number

Encoding schemes

To make characters possible, we use **encoding schemes** like:

- **ASCII**
- **Unicode**

These schemes:

- assign a **number** to each character
- create a rule everyone agrees on

Example (ASCII):

- 'A' → 65
- 'B' → 66
- 'a' → 97

Strings in memory

A string is stored as:

- many character codes **in order**
- usually followed by a **null terminator**

Null terminator: **0**

It tells the program:

"The string ends here."

Big idea

- Characters look like letters
- Strings look like words
- But in memory:
    - ✓ characters = integers
    - ✓ strings = sequences of integers

At the memory level, everything is numbers.

"CAT" → 67 65 84

Each number is the ASCII code of one character.

Characters and strings are stored as integers because all data in computer memory is represented as numbers, using encoding schemes like ASCII.

👉 There is **no special "text" storage** inside the computer.

What changes is **how the numbers are interpreted**.

ASCII says:

- 65 means A
- 66 means B
- 97 means a

So, the computer stores numbers, and software decides:

"These numbers should be treated as characters."

💡 **Text is not special inside a computer — it is just numbers that we choose to read as letters.**

Characters and strings are stored as integers because computer memory can only represent numbers, and encoding schemes such as ASCII define how numeric values correspond to characters.

RESERVED WORDS

A **reserved word** is a word that the assembler has **already claimed**.

Think of it like this:

The assembler says: "This word already has a job. You can't reuse it for something else."

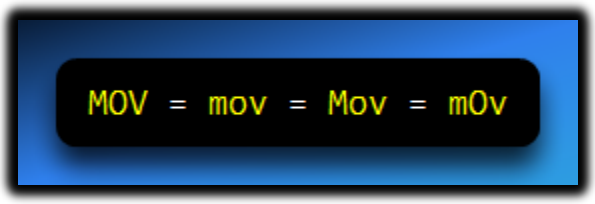
So:

- You **cannot** use reserved words as variable names
- You **must** use them only where they are meant to be used

Case Sensitivity

Reserved words are **not case-sensitive**.

That means:



They all mean the **same instruction**.

Types of Reserved Words (With Meaning)

1. Instruction Mnemonics

These are the actual **commands** the CPU understands.

Examples:

- MOV → move data
- ADD → add values
- MUL → multiply values

You **must not** use these as identifiers.

2. Register Names

Registers are **small storage locations inside the CPU**.

Examples:

- AX, BX, CX
- EAX, EBX

These names are reserved because they refer to real hardware.

3. Directives

Directives tell the **assembler**, not the CPU, what to do.

They control **how the program is built**, not how it runs.

Examples:

- .data → start of data section
- .code → start of code section

4. Attributes

Attributes describe **size or type of data**.

Examples:

- BYTE → 1 byte
- WORD → 2 bytes

They help the assembler know **how much memory to use**.

5. Operators

Operators are symbols or words used in **constant expressions**.

Examples:

- +, -, *
- AND, OR

They are reserved because they perform calculations.

6. Predefined Symbols

These are **special names** that already have values.

Example:

- @data → returns a constant integer at assembly time

You don't define them — the assembler provides them.

7. Summary: Reserved Words

- Reserved words have **special meaning**
- They can **only** be used in their intended context
- They are **not case-sensitive**
- You **cannot** use them as identifiers

IDENTIFIERS

I. What Is an Identifier?

An **identifier** is a name **you choose**.

You use identifiers to name:

- Variables
- Constants
- Procedures
- Labels

Identifiers exist to make code **readable and understandable**.

II. Rules for Forming Identifiers

1. Length

- Must be between **1 and 247 characters**
- Long names are allowed
- Short, meaningful names are recommended

2. Case Sensitivity

Identifiers are **not case-sensitive**. So:

They all refer to the **same identifier**.

3. First Character Rule

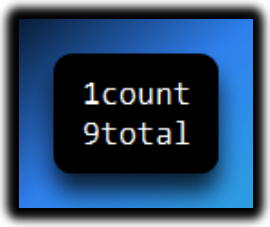
The first character **must be one of these**:

- A letter (A–Z or a–z)
- _ (underscore)
- @
- ?
- $

❌ Cannot start with a digit.

⭐ Valid:

❌ Invalid:

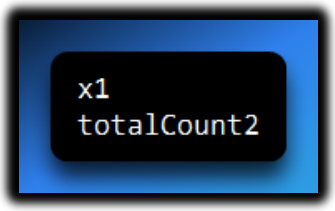


4. Remaining Characters

After the first character, you may also use:

- Letters
- Digits (0–9)

✓ Valid:



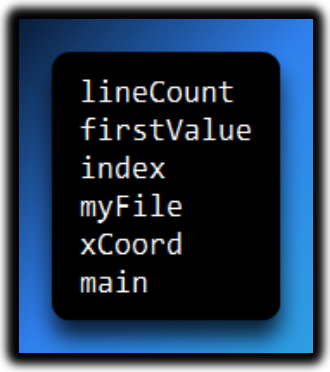
5. Cannot Be a Reserved Word

You **cannot** use:

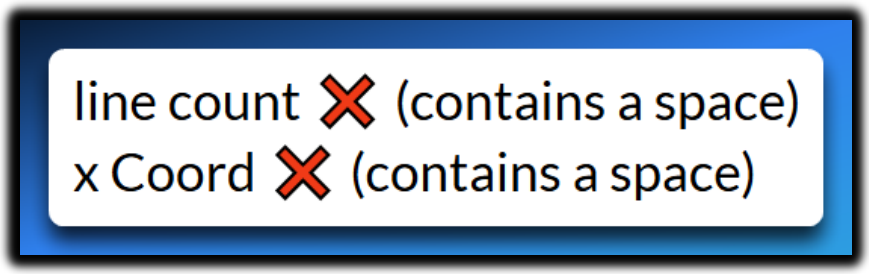
These already belong to the assembler.

Good Identifier Naming (Style Matters)

Even though assembly looks cryptic, **your names don't have to be**.



Invalid identifiers (why they are wrong):



Spaces are **not allowed** in identifiers.

Legal but Not Desirable

These **work**, but are discouraged:

Why?

- _, $, and @ are often used internally by assemblers
- Using them can cause confusion or conflicts

Reserved words have predefined meanings in assembly and cannot be used as identifiers, while identifiers are programmer-defined names that follow specific rules to improve code readability.

💡 **The assembler already owns some words — you choose names for everything else.**

ASSEMBLER DIRECTIVES: THE BLUEPRINTS

If instructions (like MOV and ADD) are the bricks and actions of your program, **Directives** are the blueprints.

Directives are special commands for the *Assembler* (the software building your program), not for the *CPU*.

👉 Directives:

- Are read **only when assembling**
- Do **not run** when you click the .exe file.
- Do **not generate machine code instructions**

Think of directives as **setup instructions**:

"Assembler, here's how to build my program."

They tell the assembler how to set up memory, where to put variables, and how much space to reserve *before* the program ever starts.

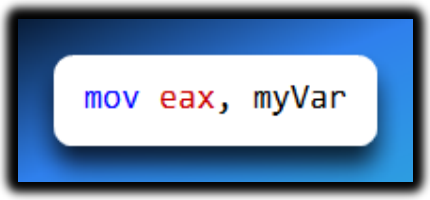
Directives are generally **not** case-sensitive. .data, **.DATA**, and **.Data** are all the same thing.

Directives vs. Instructions

The Directive (DWORD): Tells the assembler,
"Hey, reserve 4 bytes of space right here and call it myVar." (During assembly time).
Talks to the **assembler.**

The Instruction (MOV): Tells the CPU,
"Hey, go grab the data inside myVar and move it to a register." (Happens during run time).
Talks to the **CPU.**

- DWORD is a directive.
- It tells the assembler: "Reserve 4 bytes and store the value 26"
- No CPU action happens here.



- MOV is an instruction.
- It runs at runtime.
- It copies data into a register.

Important Properties of Directives

Directives are not case-sensitive

These all mean the same thing:

Why Directives Exist

Directives are used to:

- Define variables
- Allocate memory
- Organize program sections
- Define constants
- Set up the stack
- Control how code is assembled

PROGRAM SEGMENTS

A program is divided into segments.

Each segment has a specific purpose.

Directives tell the assembler: **“This part of the program is for X.”**

Common Assembly Directives

.data — Initialized Data Segment

The .data directive marks the section where **variables with known initial values** are stored.

“The following lines define data that already has values.”

```
.data
count DWORD 10    ; A 32-bit variable starting at 10
userName BYTE 'A' ; A character variable
```

- Memory is reserved
- Values are stored immediately
- This is where you put constants and variables that have a starting value.
- Defining known values (like a high score starting at 0, or a username).

.bss — Uninitialized Data Segment

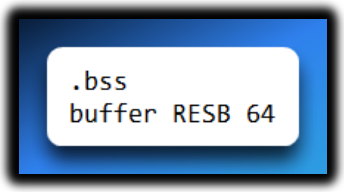
This stands for "Block Started by Symbol" (an old history term), bss is an empty space.

Its used for variables that exist but start with no value. (like a buffer for user input).

“Reserve memory, but don’t store values yet.”

```
.bss
buffer BYTE 100 DUP(?) ; Reserve 100 bytes, contents unknown (?)
```

It saves space in the executable file. You don't need to store 1,000 zeros; you just tell the OS, "I need 1,000 bytes of empty space here."



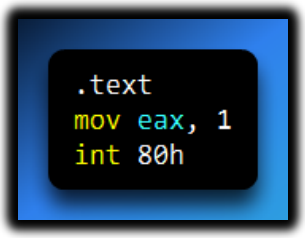
- Space is reserved
- Contents are undefined (garbage)
- Used for arrays and large buffers

.text or .code — Code Segment

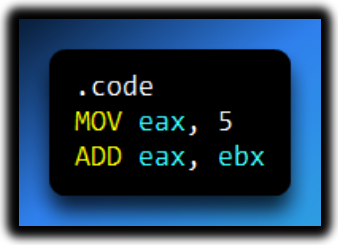
This is the **Read-Only zone** where your actual code instructions live.

The CPU fetches commands from here.

"The CPU will run what comes next."



This is actual program logic.



.equ — Define a Constant Symbol

A symbol is a name that represents:

- A constant value
- A memory location
- An address

Symbols make code **readable and maintainable**.

The .equ directive defines a **constant**.

Once defined, it **cannot change**.

```
MAX_SIZE equ 100 ; Everywhere you write MAX_SIZE its substituted with 100
```

It works like Find and Replace.

It does not use any memory; it just helps you read the code.

```
PI .equ 3          ; Wherever you type PI, the assembler sees 3
MOV eax, PI        ; The assembler reads this as: MOV eax, 3
```

Why .equ is useful

- Avoids magic numbers
- Easy to change values
- Makes code portable

.stack — Define the Runtime Stack

What is the stack?

The stack is a special, dynamic area of memory used for temporary storage.

It manages subroutine calls (keeping track of where to return to) and local variables.

LIFO Structure: It works like a stack of plates. The last plate you put on top (Push) is the first one you take off (Pop).

Growth: Weirdly, the stack usually grows *downwards* in memory (from high addresses to low addresses).

Setting the Size: You must tell the assembler how big this scratchpad should be using the .STACK directive.

The **runtime stack**:

- Stores return addresses
- Stores local variables
- Grows downward in memory
- Uses LIFO (Last In, First Out)

What .STACK does

The .STACK directive:

- Reserves memory for the stack
- Sets its maximum size

- Allocates 100 bytes for the stack
- Prevents stack overflow (if sized correctly)

Why Stack Size Matters

If the stack grows beyond its allocated space:

- Memory gets overwritten
- Program may crash
- Behavior becomes unpredictable

This is called stack overflow.

```
.STACK 100

.data
message BYTE 'Hello, world!', 0

.text
start:
    push message
    call puts
    add esp, 4

    mov eax, 1
    xor ebx, ebx
    int 80h
```

What happens here

- .STACK → sets stack size
- .data → stores text
- .text → contains instructions
- Directives prepare the program
- Instructions run the program

Assemblers share the same **instruction set**, but **directives differ** between assemblers.

Example:

- Microsoft assembler supports REPT
- Other assemblers may not

Directives are assembler commands that control program structure, memory allocation, and symbol definition, and they do not generate executable machine instructions.

💡 **Directives build the program. Instructions run the program.**

- **Directives** (starts with .) = Instructions for the **Assembler** (Setup).
- **Instructions** (like MOV) = Commands for the **CPU** (Action).
- **Segments:**
  - ✓ .data = Variables with values.
  - ✓ .bss = Empty variables.
  - ✓ .code = The actual program logic.
  - ✓ .stack = Temporary scratchpad.

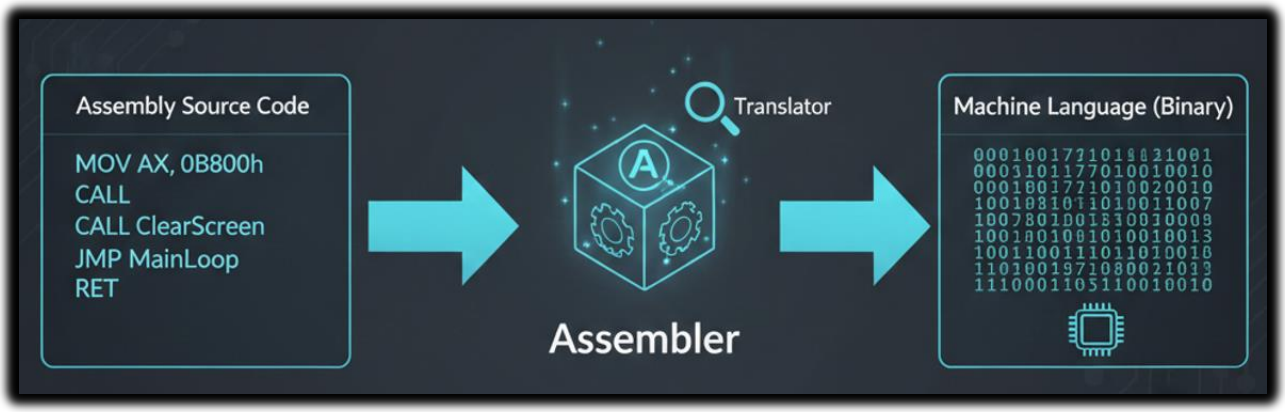
INSTRUCTIONS

Think of an instruction as a single, clear command given to the computer's brain (the CPU).

When you write a line of assembly code, you are basically writing a to-do list for the processor.

However, the CPU doesn't speak English; it only understands bits and bytes.

So, when you assemble your code, a program called an **Assembler** acts as a translator, turning your written instructions into the machine language the computer actually runs.



An instruction isn't just one big lump of text; it's usually broken down into four specific parts.

1. The Label (The "Bookmark")

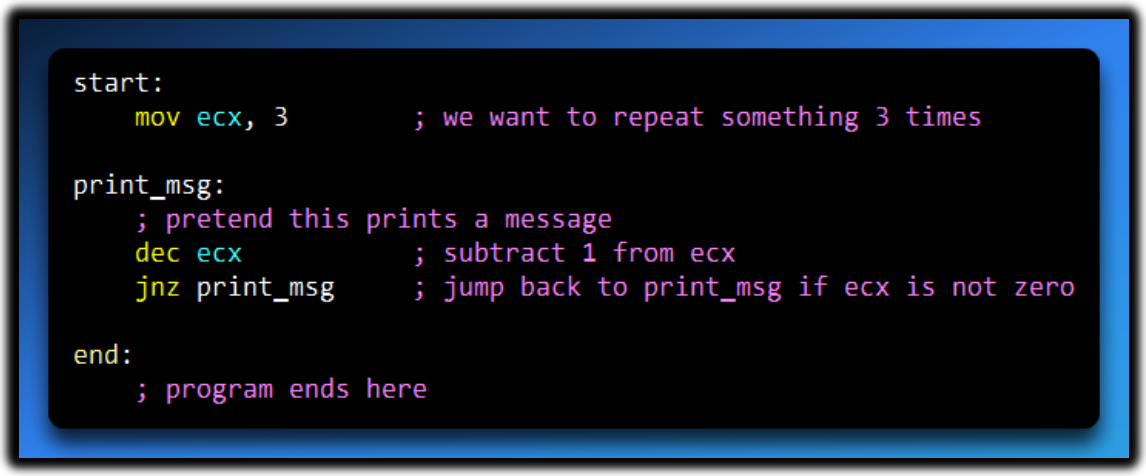
A label is completely optional, but it's incredibly useful.

Think of it like a bookmark or a signpost in your code.

It's just a name you give to a specific spot in the program so you can find it easily later.

If you want the computer to "jump" back to a certain spot or repeat a section of code (like a loop), you give that spot a label.

- **How it looks:** You write a word and follow it with a colon (like loop:).
- **The Golden Rule:** Every label name has to be unique. You can't have two spots named "Step1," or the computer won't know which one you're talking about.



```
start:
    mov ecx, 3          ; we want to repeat something 3 times

print_msg:
    ; pretend this prints a message
    dec ecx             ; subtract 1 from ecx
    jnz print_msg       ; jump back to print_msg if ecx is not zero

end:
    ; program ends here
```

How this explains labels

- **start:**
  This is a **label**. It marks the beginning of the program.
  The program does *not* have to have it, but it's useful.
- **print_msg:**
  This label marks a spot we want to **jump back to**.
  It acts like a **bookmark**.
- **jnz print_msg**
  This tells the computer:
  "If the condition is true, go back to the place named print_msg."
  That's the label being used.
- **end:**
  Another label marking where the program finishes.

2. The Mnemonic

The **mnemonic** is the **actual command** in an assembly instruction.
It is the **only part that must exist**.

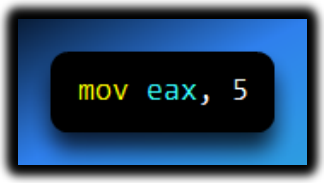
A mnemonic is just a **short, easy-to-remember name** for something the CPU knows how to do.

Before mnemonics existed, programmers had to write **long strings of numbers** to control the computer. That was slow, hard to read, and easy to mess up.
Mnemonics fixed that by giving those numbers **names**.

Think of the mnemonic as the **verb** in a sentence:

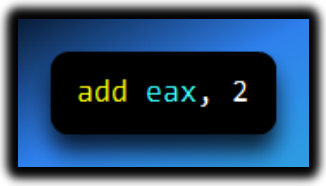
- MOV → move data
- ADD → add numbers
- SUB → subtract
- JMP → jump to another place in the code

If an instruction has no mnemonic, it is **not an instruction**.
The CPU cannot guess what you want—it needs a command.



- mov → mnemonic (the command)
- eax, 5 → operands (what the command works on)

This tells the CPU: **"Move the value 5 into the register EAX."**



- **add** is the mnemonic. It tells the CPU to **perform addition**

Without the mnemonic, this would mean nothing. The mnemonic is the name of the operation being performed. It tells the CPU what action to take and is the only required part of an assembly instruction.

3. The Operands (The "Targets")

If the **mnemonic** is the verb, then the **operands** are the **nouns**.

Operands are the **things the instruction works on**.
Most instructions don't make sense without them. If you tell the CPU to ADD, its next question is:

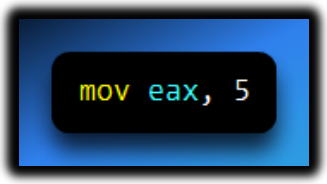
"Add *what* to *what*?"

That's what operands answer.

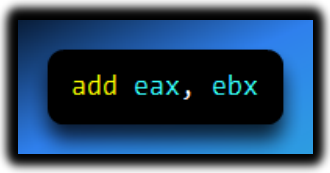
What operands can be

Operands can be different kinds of things, depending on the instruction:

- **Constants**
  A fixed number written directly in the code: **5**
- **Registers**
  Small, fast storage locations inside the CPU: **eax, ebx**
- Memory locations
  A specific place in RAM where data is stored: **[value]**
- **Labels**
  A named location in the program, used mainly with jump instructions: **loop**



- mov → mnemonic
- eax and 5 → operands
- Meaning: move the constant 5 into the register eax



- Operands: eax, ebx
- Meaning: add the value in ebx to the value in eax

```
jmp loop
```

- Operand: loop (a label)
- Meaning: jump to the place in the code named loop

Code label (for jumps/loops):

```
loop_start:
    add eax, 1
    cmp eax, 10
    jnz loop_start  ; jump back to loop_start if eax != 10
```

Data label (for variables):

```
myCount dw 100          ; declare a word variable
count_label dw myCount   ; label pointing to the memory address of myCount

mov eax, [count_label]  ; load value of myCount into eax
```

Array example with offset:

```
array dw 1024, 2048, 3072, 4096
mov eax, [array + 2]     ; load 2048 (second value, 2 bytes offset) into eax
```

Important detail

- Some instructions have **one operand**
- Some have **two**
- A few have **none**

But when operands are present, they always tell the CPU:

where the data comes from and **where the result goes**

Operands are the values, registers, memory locations, or labels that an instruction acts upon.

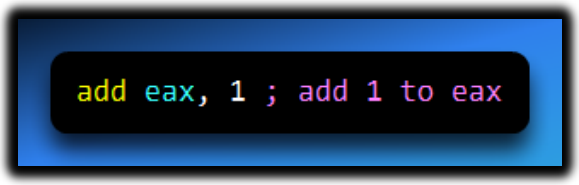
4. The Comments (Notes for Humans)

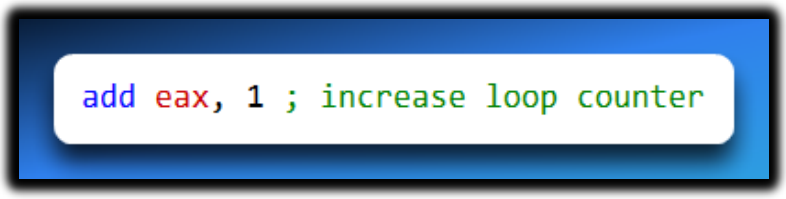
Comments are only for humans.
The CPU and the assembler completely **ignore** them.

- Comments do **not** become machine code
- They do **not** take up memory
- They exist only to help **you** and anyone else reading the code

Assembly can get confusing fast, so comments explain **why** something is done, not just **what** is done.

A comment starts with a **semicolon**:



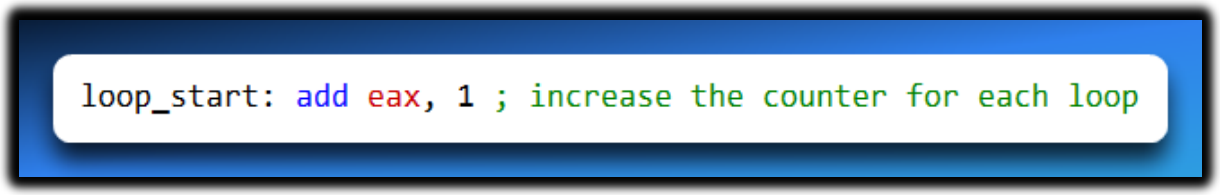


The second one is more useful when you come back later or need to fix a bug.

---

5. Putting it all together

Here is a full instruction showing **all parts working together**:



- loop_start: → **Label** (marks a location in the code)
- add → **Mnemonic** (the action)
- eax, 1 → **Operands** (what the action works on)
- ; increase the counter for each loop → **Comment** (human explanation)

Comments don't affect the program at all, but they make the code understandable and easier to maintain.

NOP (No Operation)

The **NOP** instruction stands for **No Operation**.

- When executed, it **does absolutely nothing**.
- It **takes 1 byte** of memory.
- It's mostly used as a **placeholder** or for **aligning code** in memory.

Why use NOP?

1. **Alignment**: Some processors work faster if instructions start at specific memory addresses (like multiples of 4).
2. **Padding**: To maintain the size of an instruction stream.
3. **Debugging**: You can insert NOPs temporarily to test timing or skip over instructions without changing program behavior.

Example 1: Simple NOP

```
mov eax, 1      ; move 1 into eax
nop             ; do nothing
add eax, 2      ; add 2 to eax
```

- The nop instruction does nothing.
- It's just a placeholder between the two instructions.

Example 2: Alignment Example

```
mov eax, ebx    ; 3 bytes
mov ecx, edx    ; 3 bytes
nop             ; 1 byte to align next instruction to 4-byte boundary
add eax, ecx    ; 3 bytes, now properly aligned
```

- The nop ensures that the **next instruction starts at a multiple-of-4 address**.
- This can **improve performance** because the processor accesses memory more efficiently.

Key Points

- NOP **does nothing** when executed.
- It's used for **padding, alignment, or debugging**.
- Takes **1 byte** of memory.
- Does **not affect registers or memory**.

x86 PROCESSORS AND SPEED

x86 processors work **faster** when code and data start at **even doubleword addresses** (that means addresses that are multiples of **4 bytes**).

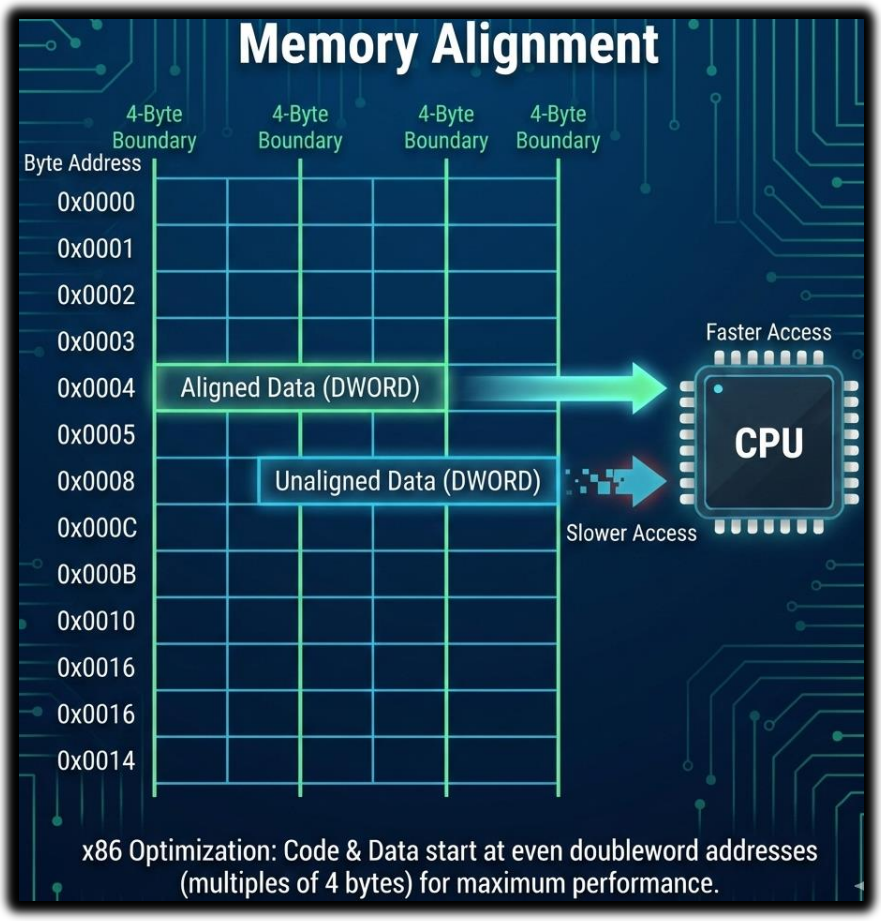
Why this matters

- The x86 CPU moves data in **4-byte chunks**
- If data starts at a **4-byte boundary**, the CPU can fetch it in **one step**
- If it's **not aligned**, the CPU needs **two memory accesses**
- Two accesses = **slower program**

Aligned vs unaligned (idea)

- Aligned address: 0, 4, 8, 12, ...
- Unaligned address: 1, 2, 3, 5, 6, ...

When data is unaligned, performance drops.



How programmers fix this

To avoid slowdown, programmers:

- align code and data to 4-byte boundaries
- use **padding**
- use **NOP instructions** to push code to the correct address

x86 processors load code and data faster from even doubleword (4-byte aligned) addresses because aligned data can be fetched in a single memory access.

ANATOMY OF A 32-BIT ASSEMBLY PROGRAM

Here is the full, working source code for addTwo.asm

```
; addTwo.asm - Adds two 32-bit integers

.386                            ; 1. Target the 80386 processor (allows 32-bit registers)
.model flat, stdcall            ; 2. Use Flat memory model & StdCall convention
.stack 4096                     ; 3. Reserve 4096 bytes for the stack
ExitProcess PROTO, dwExitCode:DWORD  ; 4. Declare the Exit function prototype

.code
main PROC
    mov eax, 5                  ; Move 5 into EAX
    add eax, 6                  ; Add 6 to EAX (Result: 11)

    INVOKE ExitProcess, 0       ; Tell Windows we are done (Return 0)
main ENDP
END main
```

SETUP DIRECTIVES (THE RULES)

Before writing any instructions, we must tell the assembler **what kind of program we are writing**.
These directives do **not** generate machine code. They only set rules.

I. Processor Directive — .386

This tells the assembler:

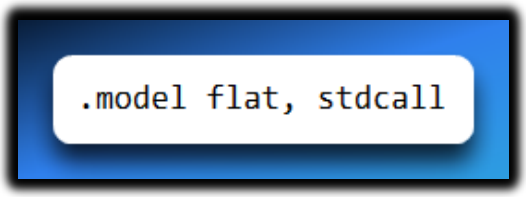
"Generate code for an Intel 80386 processor or newer."

Why this matters:

- The 80386 was the first **32-bit** x86 processor
- This directive enables **32-bit registers** such as EAX, EBX, ECX
- Without it, the assembler assumes **16-bit mode**

Bottom line:
.386 is required for 32-bit assembly programs.

II. Memory Model — .model flat, stdcall



This directive defines **how memory is addressed** and **how functions are called**.

Flat memory model

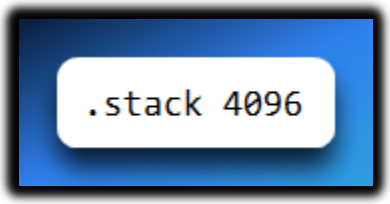
- Memory is treated as **one continuous address space**
- You can access any memory location directly
- This is the standard model used by **32-bit Windows**

Stdcall calling convention

- Used by **Windows API functions**
- Function arguments are passed on the stack
- The **called function** cleans up the stack

Bottom line:
32-bit Windows programs require flat memory and stdcall function calls.

III. Stack Directive — .stack 4096



This reserves **4096 bytes (4 KB)** for the program stack.

Why 4096 bytes:

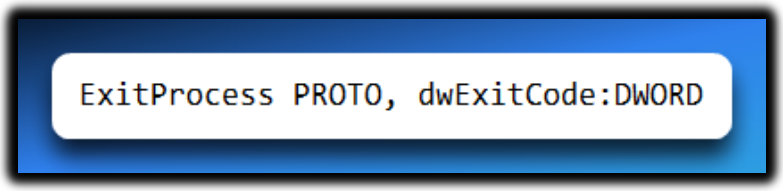
- 4 KB is the size of a standard **memory page**
- Enough space for local variables and function calls in small programs

Bottom line: The stack is required for function calls and parameter passing.

TALKING TO THE OPERATING SYSTEM

A Windows program must tell the OS **when it finishes** and **whether it succeeded**.

I. Function Prototype — PROTO



This tells the assembler:

- There is a function named ExitProcess
- It takes **one parameter**
- The parameter is a **DWORD**

Why this is required:

- The INVOKE instruction needs to know the function's parameters
- Prevents calling functions with the wrong number or type of arguments

Bottom line:
You must declare a function prototype before using INVOKE.

II. Exit Code — dwExitCode

When a program ends, it returns an **exit code** to the operating system.

Common values:

- 0 → Program completed successfully
- Non-zero → Program failed or encountered an error

III. Why the Exit Code Matters

Operating systems and scripts check the program's exit code.

Example:

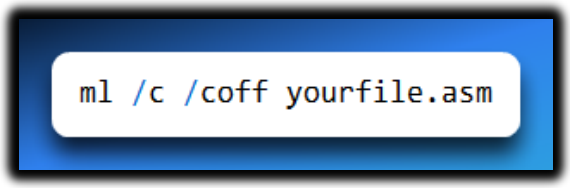
- A batch file runs several programs
- It checks %ERRORLEVEL%
- If the exit code is 0, it continues
- If the exit code is non-zero, it stops or reports an error

Bottom line: Always return 0 if your program finishes correctly.

BUILDING THE PROGRAM (COMPILE & LINK)

Assembly language programs are built in **two steps**.

Step 1: Assembly



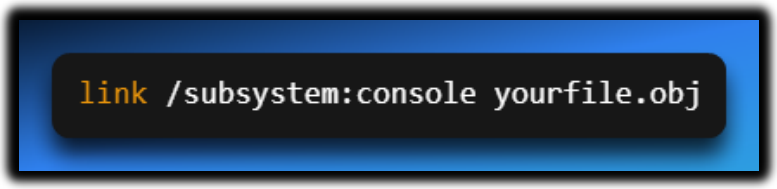
What happens:

- MASM converts .asm source code into machine code
- Output is an **object file** (.obj)

Options:

- /c → assemble only (do not link)
- /coff → use Common Object File Format

Step 2: Linking



What happens:

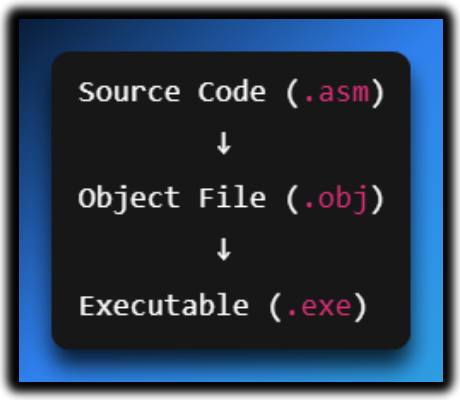
- The linker combines your object file with system libraries
- Produces the final **executable file** (.exe)
- Links against Windows libraries such as kernel32.lib

/subsystem:console:

- Tells Windows this is a **console application**

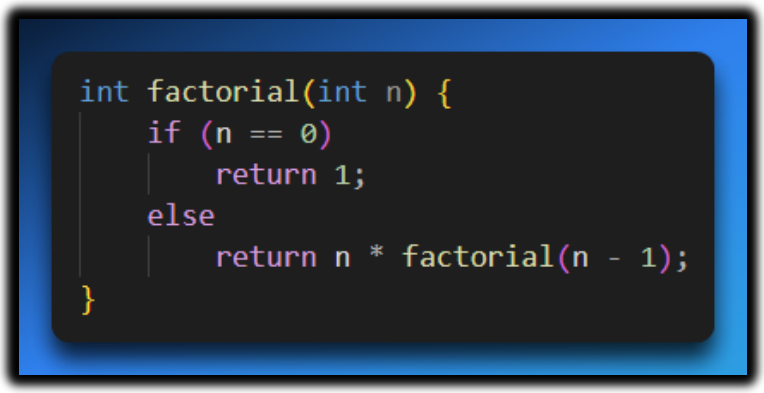
BIG IDEAS TO REMEMBER

- Use .386 to enable **32-bit registers**
- Use .model flat, stdcall for **32-bit Windows programs**
- Declare external functions with PROTO
- Always return 0 to indicate success
- Build process: .asm → .obj → .exe



THE STACK

You already know this idea from C, so we'll use C just to **confirm** what the stack does.



```c
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

What happens on the stack when main() calls factorial(5)

Each function call creates a **stack frame**.

When factorial(5) is called:

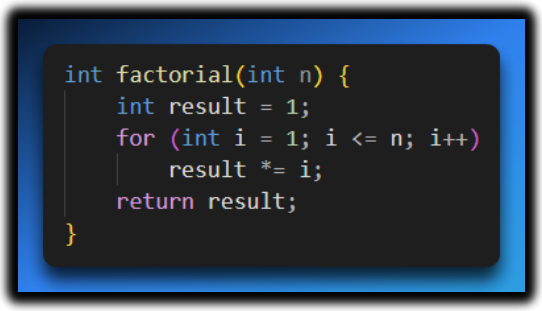
1. The **return address** is pushed onto the stack
   (where to go back after the function finishes)
2. The **parameter (n)** is pushed onto the stack
3. Control jumps to the factorial function
4. The function runs
5. When it returns:
   - ✓ parameters are removed
   - ✓ return address is popped
   - ✓ execution continues where it left off

Key idea (this is all you need)

The stack remembers **where to return** and **stores function data**.

Stack Frames and Local Variables

Local variables live on the **stack**.



```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

- *result* and **i** are **local variables**
- They exist **only while the function runs**
- When the function returns, the stack frame is destroyed

The runtime stack stores return addresses, parameters, and local variables for function calls.

ASSEMBLY PROGRAM STRUCTURE

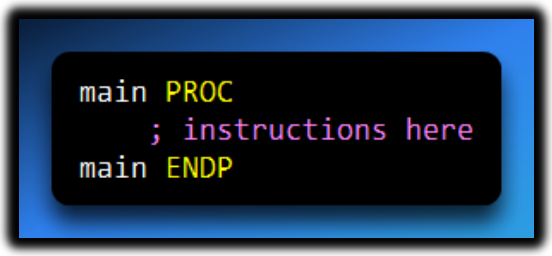
Now let's connect this to **assembly**, very simply.

.CODE Directive

- Marks the **start of executable instructions**
- Everything after this is **code**

Usually followed by the program's **entry point**, commonly main.

Procedures: PROC and ENDP



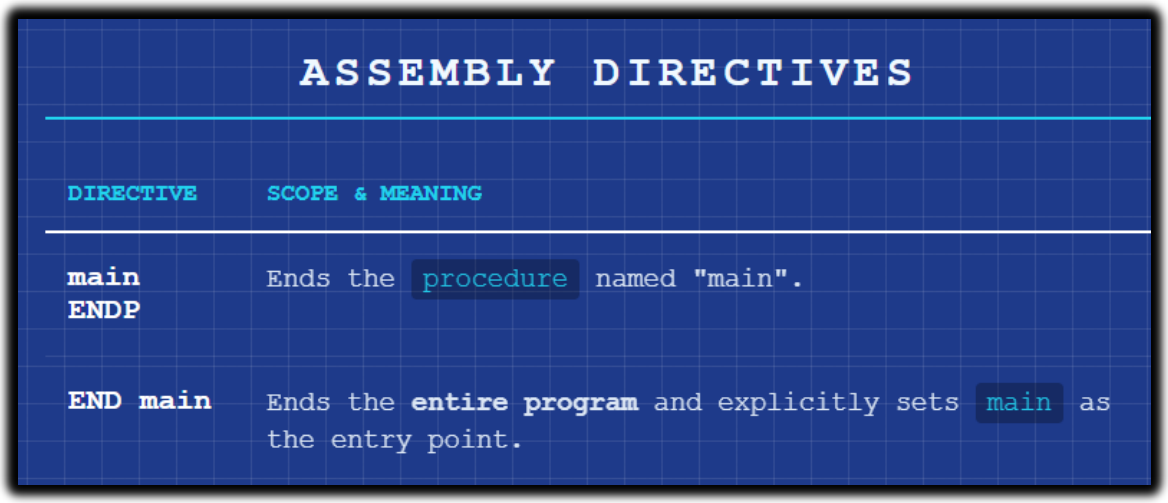
- PROC marks the **start** of a procedure
- ENDP marks the **end**
- The names must **match**

END Directive (Very Important)

What this means:

- Marks the **end of the entire program**
- Tells the assembler where execution **starts**

Difference between ENDP and END



Important note: Any lines written **after END are ignored** by the assembler.

You can put comments there — it won't matter.

RUNNING AND DEBUGGING (SHORT & PRACTICAL)

- Assembly programs run inside a **console window**
- Same window as cmd.exe

Breakpoints (Visual Studio)

- Click in the gray bar next to a line
- A **red dot** appears
- Program pauses *before* executing that line

If you place a breakpoint on a non-executable line:

- VS moves it to the **next executable instruction**

Debug Mode Visual Cues

- **Orange bar** → debugger is running
- **Blue bar** → edit mode

You **cannot edit code while debugging**.

Registers While Debugging

- Registers window shows CPU registers
- Registers that change turn **red**
- EAX = 0000000B → hex for decimal 11

(New VS versions hide some of these by default — that's normal.)

PROGRAM TEMPLATE IDEA

Assembly programs follow a **fixed structure**, so we use templates.

Why?

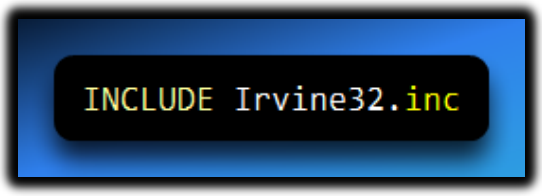
- Avoid rewriting setup code
- Reduce mistakes
- Faster development

Always comment:

- program purpose
- author
- date
- changes

This helps **future you**, not just others.

INCLUDE Directive



- Copies another file into your program
- Often used for:
    - ✓ macros
    - ✓ procedures
    - ✓ library code

ASSEMBLE → LINK → RUN (FINAL FORM)

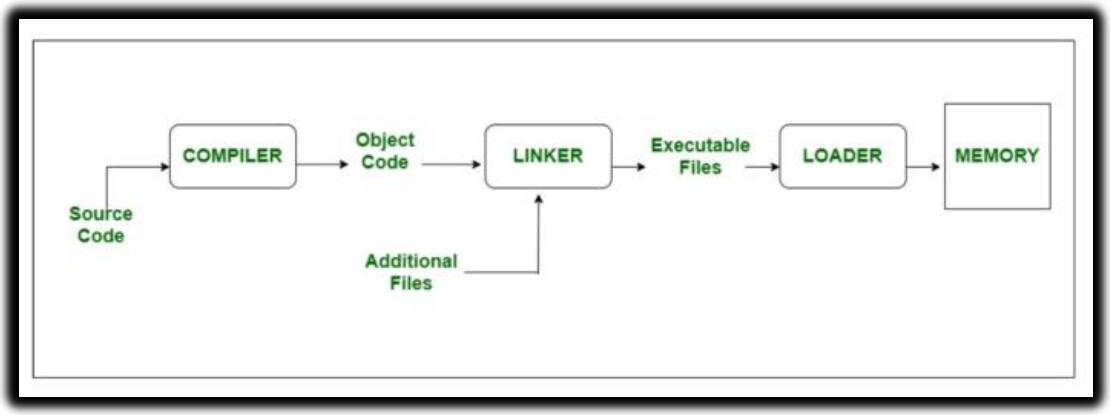
Assembly programs cannot run directly.

Build steps:

1. **Edit**
    - ✓ Write .asm source file
2. **Assemble**
    - ✓ Converts source → .obj
3. **Link**
    - ✓ Combines .obj + libraries → .exe
4. **Run**
    - ✓ OS loads program into memory
    - ✓ CPU jumps to entry point

Final one-line summary

The stack manages function calls, .CODE defines executable instructions, PROC/ENDP define procedures, and END marks the program entry point. Assembly programs are assembled, linked, and then executed.



LISTING FILES & SYMBOL TABLES

What is a Listing File?

A **listing file** is a detailed report created by the assembler.

It shows:

- Your **original source code**
- **Line numbers**
- The **memory address** of each instruction
- The **machine code bytes** (in hex)
- A **symbol table**

Think of it as: “Show me exactly what the assembler generated.”

Who actually needs listing files?

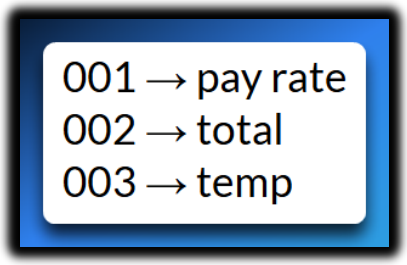
- **Beginners** → to understand how assembly becomes machine code
- **Advanced programmers** → to debug performance or instruction layout

For normal programs, you usually **don't need** it.

```
 1:    ; AddTwo.asm - adds two 32-bit integers.
 2:    ; Chapter 3 example
 3:
 4:    .386
 5:    .model flat,stdcall
 6:    .stack 4096
 7:    ExitProcess PROTO,dwExitCode:DWORD
 8:
 9:     00000000                       .code
10:     00000000                      main PROC
11:     00000000  B8 00000005            mov  eax,5
12:     00000005  83 C0 06               add  eax,6
13:
14:                                      invoke ExitProcess,0
15:     00000008  6A 00                  push   +000000000h
16:     0000000A  E8 00000000 E          call   ExitProcess
17:     0000000F                      main ENDP
18:                                   END main
```

The Symbol Table (The Important Part)

Early programmers had to manually decide memory locations:



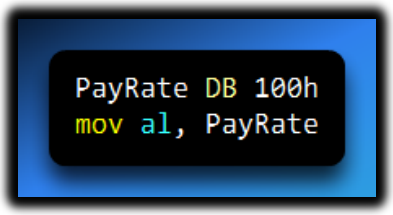
This was:

- hard to remember
- extremely error-prone

The assembler fixes this.

Instead of using raw addresses, we use **symbols**.

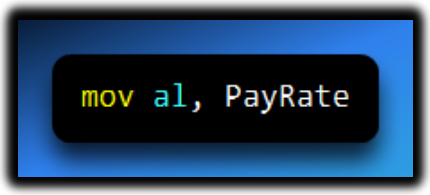
Symbolic Addressing (This is the key idea)



What's happening here?

- PayRate is a **symbolic name**
- DB tells the assembler:
  - ✓ allocate **1 byte of memory**
  - ✓ initialize it to 100h
- The assembler:
  - ✓ assigns a real memory address
  - ✓ stores it in the **symbol table**

When it sees:



It silently replaces PayRate with the correct address.

Why this is powerful

- You don't care *where* the data lives
- You only care that the name stays consistent
- Code becomes:
  - ✓ readable
  - ✓ maintainable
  - ✓ safe

What does the Symbol Table store?

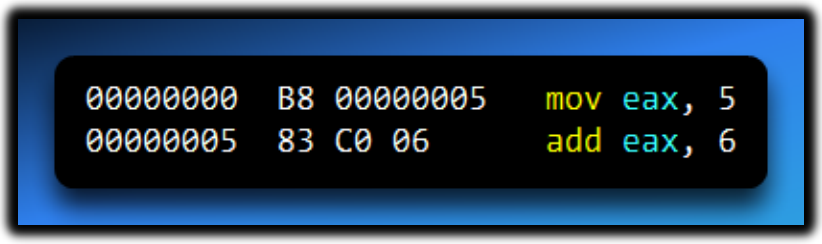
A symbol table keeps track of:

- variables and their addresses
- labels (jump targets)
- procedures
- constants
- segments

In short: Every name in your program gets an address.

Listing File Example (Simple Explanation)

A listing file shows lines like this (conceptually):



What this tells you:

- 00000000 → memory offset
- B8 → opcode for mov eax, imm32
- 00000005 → value being moved
- Instructions are stored as **hex bytes**

INVOKE in the listing file

```
invoke ExitProcess, 0
```

In the listing file, this expands to:

```
push 0
call ExitProcess
```

So, INVOKE is just a **shortcut** — the assembler writes the real instructions.

Why listing files are useful

Listing files help you:

- verify machine code generation
- see instruction sizes
- understand how macros expand
- learn how the CPU really sees your program

They are **learning and debugging tools**, not everyday tools.

80386 Reminder (Very Short)

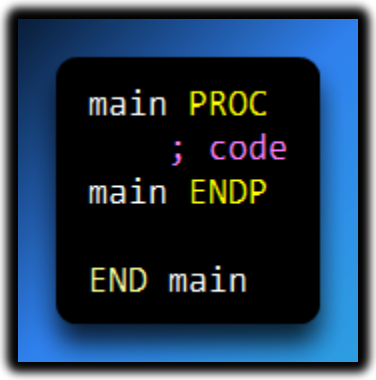
When you see:

```
.386
```

It means target is **80386 or newer**, enables **32-bit registers** and all modern CPUs qualify.

That's it. Nothing more needed.

END vs ENDP



- main ENDP → ends the **procedure**
- END main → ends the **program** and sets entry point

Generating a Listing File (Optional)

Only if you want it:

Visual Studio → Project Properties
→ Microsoft Macro Assembler
→ Listing File
→ Enable listing options

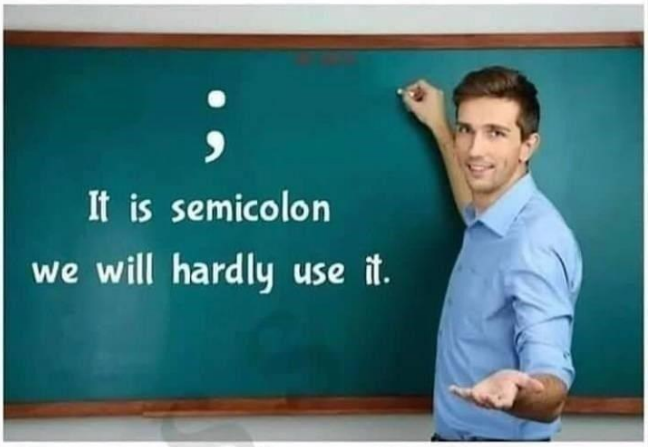
Most of the time, **you don't need this**.

Final Chapter Takeaway

- Listing files show **how source code becomes machine code**
- Symbol tables let you use **names instead of addresses**
- The assembler:
  - ✓ allocates memory
  - ✓ tracks symbols
  - ✓ replaces names with real addresses
- This makes assembly **usable**, **readable**, and **safe**

The assembler does far more than translate instructions—it manages memory, tracks symbols, and bridges human-readable code with machine execution.

English Teacher :



Computer Science Student :