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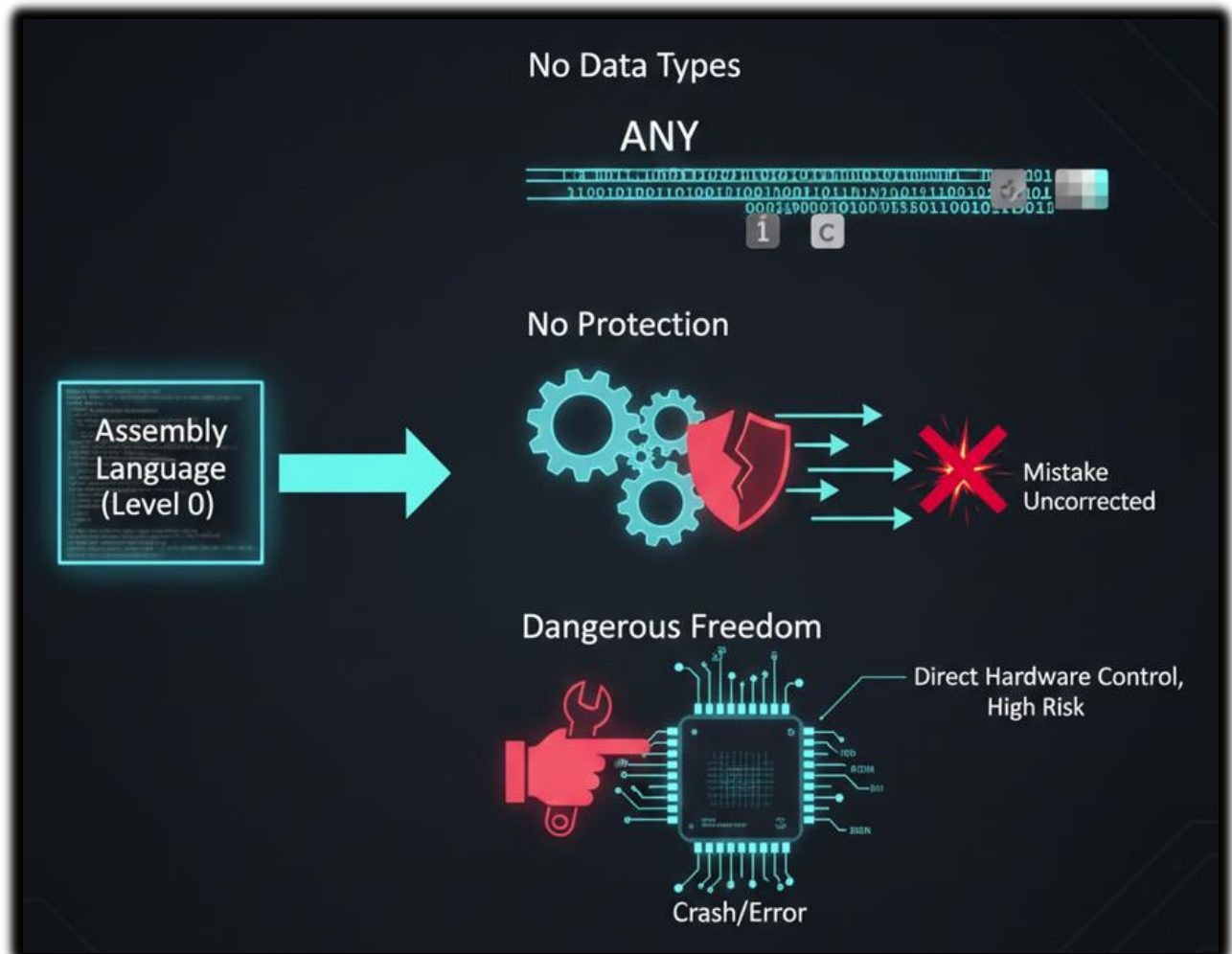
In Assembly, **size is what matters**, not meaning. When you write something like DWORD, you are only telling the computer to reserve **32 bits of memory**. You are not saying what kind of data will be stored there.

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.
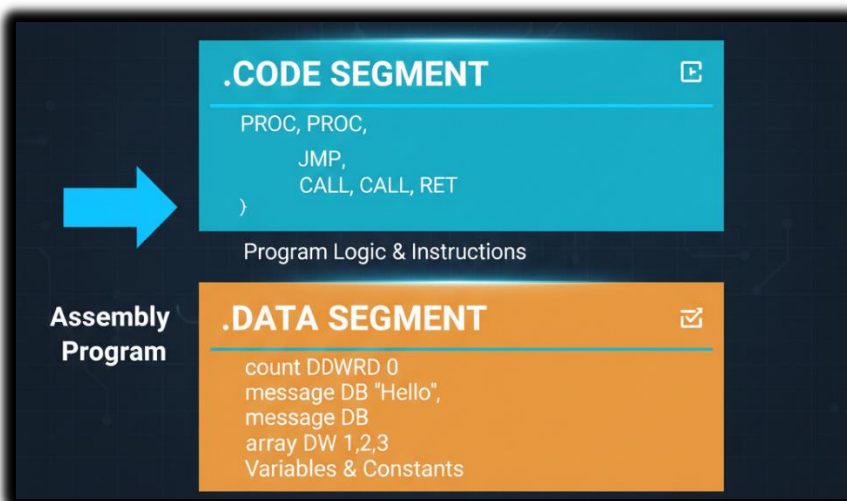
Registers, such as EAX, are the CPU's working space. They temporarily hold data while the processor performs operations.
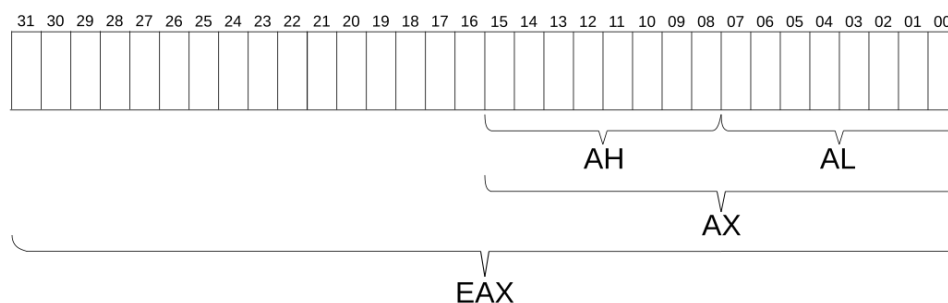
Instructions tell the CPU what to do. MOV copies data, ADD performs math, and INVOKE communicates with the operating system.

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