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

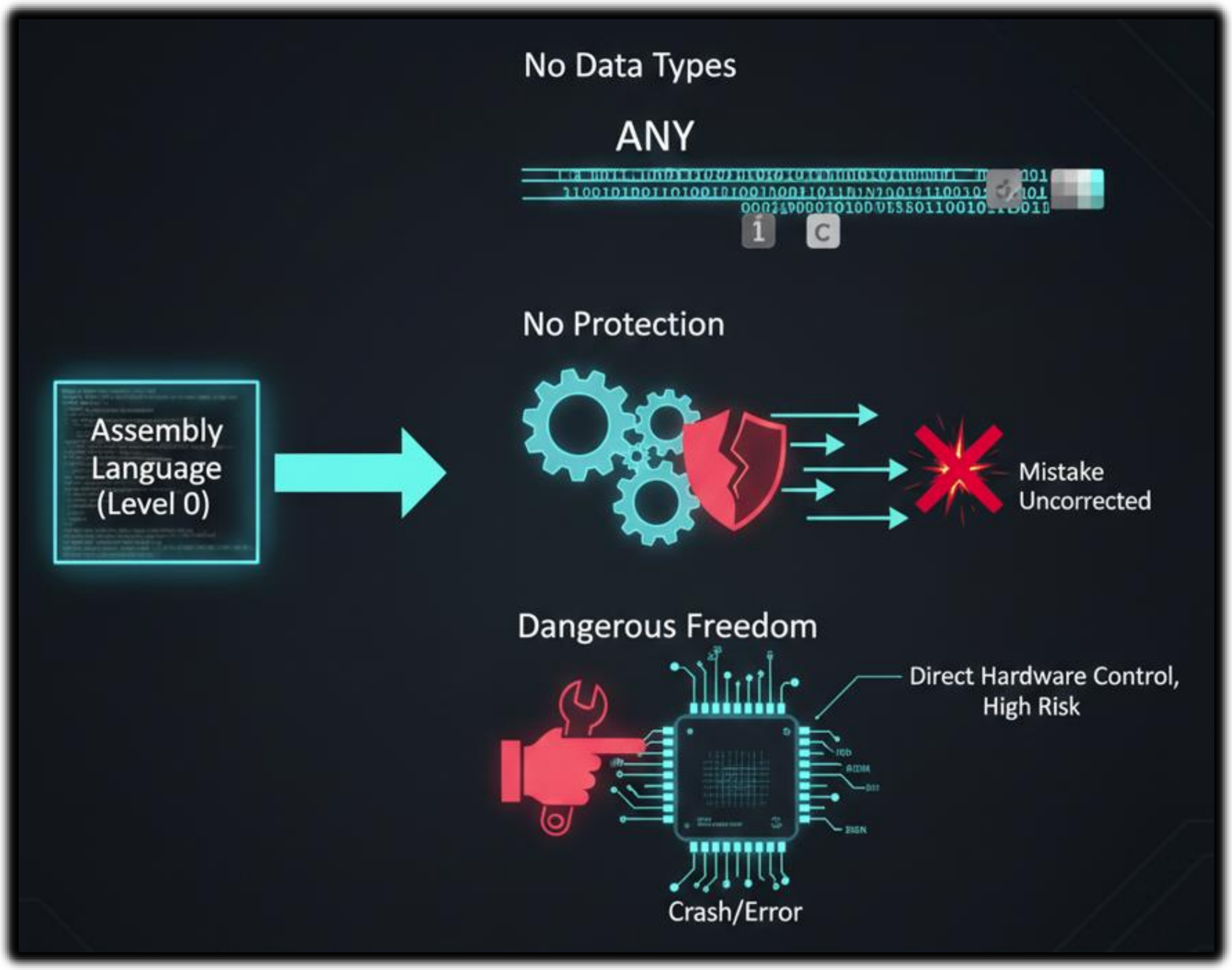
(Mentioned briefly) We will cover this later, but this is a scratchpad area for temporary storage during function calls.

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.



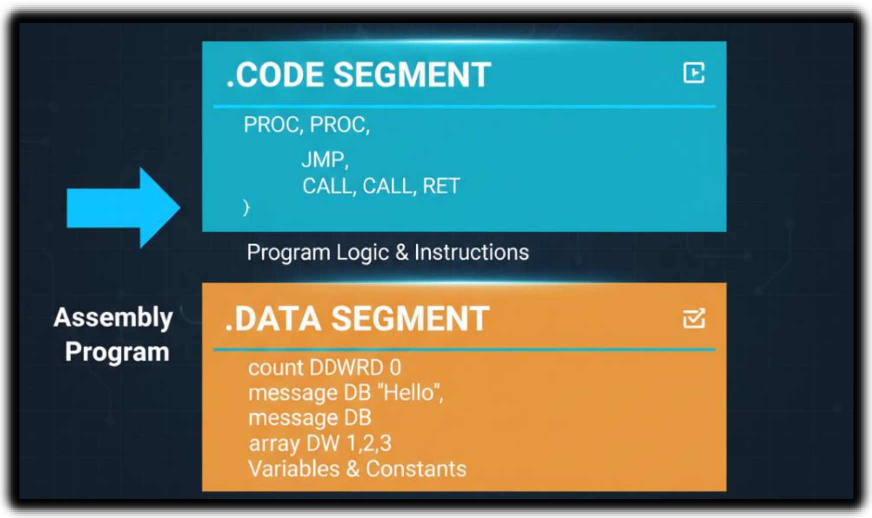
In Assembly, **size is what matters**, not meaning. When you write something like DWORD, you are only telling the computer to reserve **32 bits of memory**. You are not saying what kind of data will be stored there.

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.

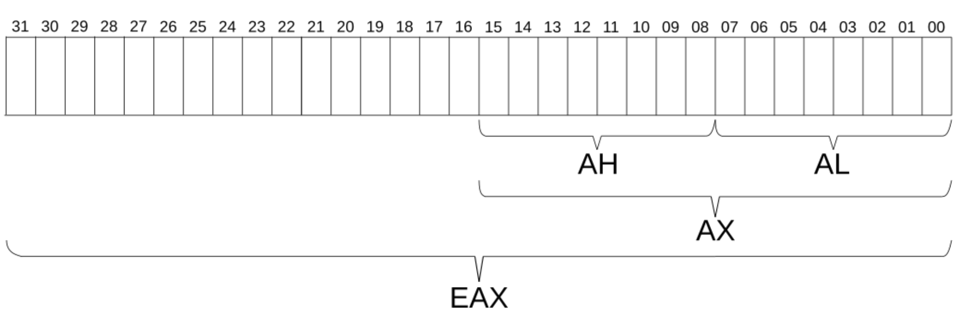
This gives you **total control**, but also total responsibility. You can treat a number like a character or an address if you want, and Assembly will allow it. If you make a mistake, the program will crash or behave incorrectly. There are no safety rails.

Big Idea to Remember

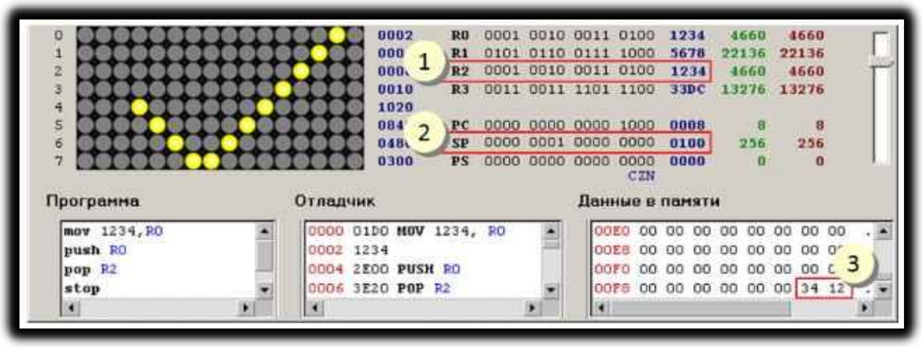
Memory is organized into **segments**. The .code segment holds the program logic, while the .data segment holds variables.



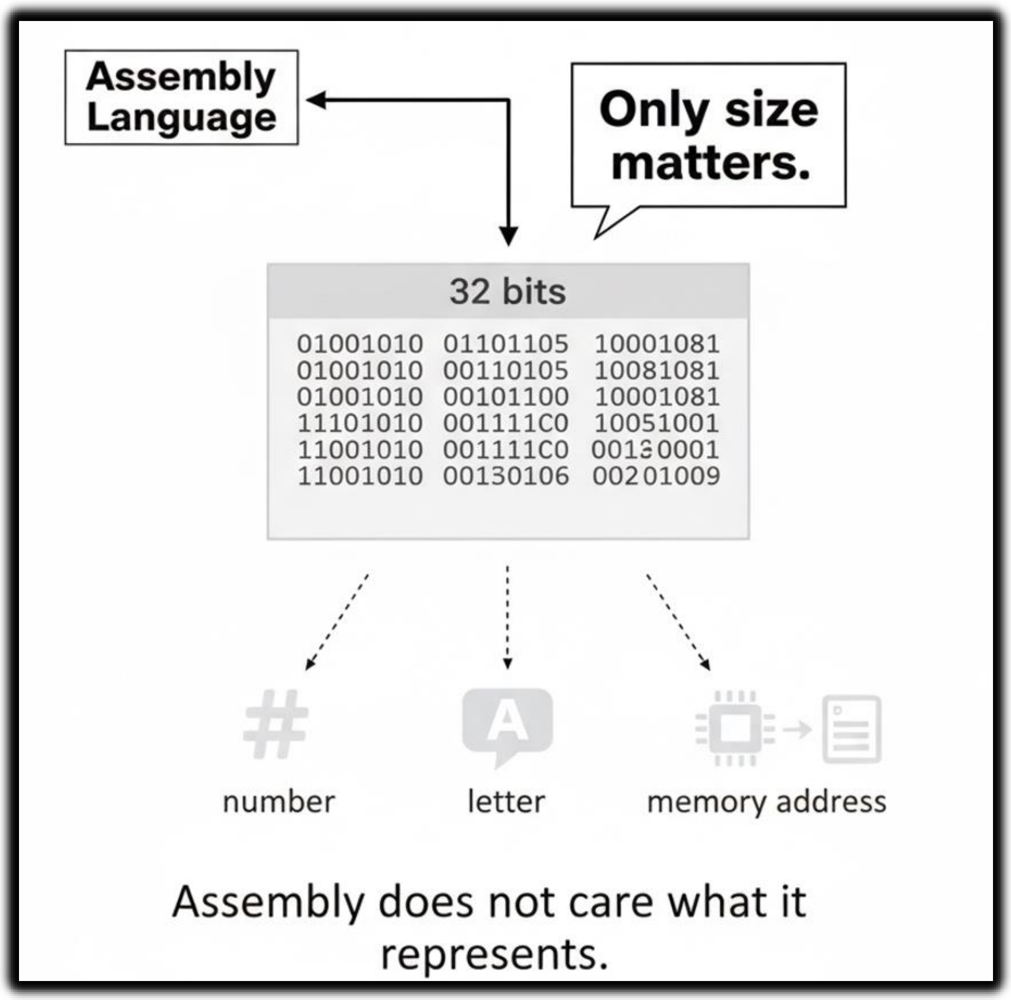
Registers, such as EAX, are the CPU's working space. They temporarily hold data while the processor performs operations.



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

```
mov ax, A123h
```

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)

```
mov ax, 0A123h
```

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:

```
mov eax, -2 * 3
```

What happens:

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

Operator Precedence Examples

```
4 + 5 * 2
```

Multiply first, then add. Result: 14

```
12 - 1 mod 5
```

1 mod 5 first → 1

Then subtraction. Result: 11

```
-5 + 2
```

Unary minus first → -5

Then add. Result: -3

```
(4 + 2) * 6
```

- 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:

```
7 mod 3 = 1
```

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:

```
[sign] integer . [integer] [exponent]
```

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
```

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:

```
number E [+ or -] integer
```

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:

```
A E B    means    A × 10^B
```

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 **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

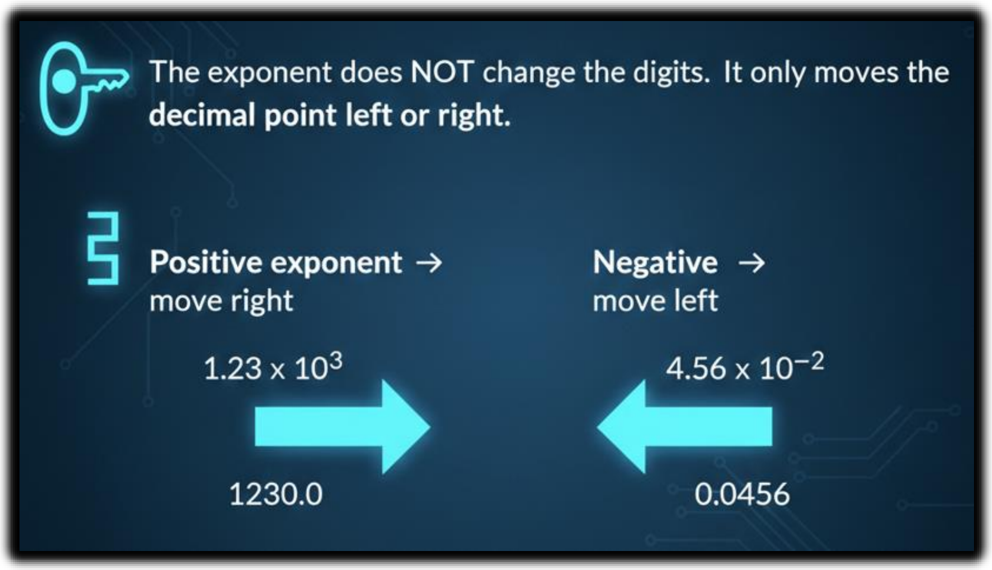
Both are **correct scientific notation**.

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:

```
00111111100000000000000000000000
```

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

```
0011 1111 1000 0000 0000 0000 0000 0000
  3    F    8    0    0    0    0    0
```

That gives: **3F800000**

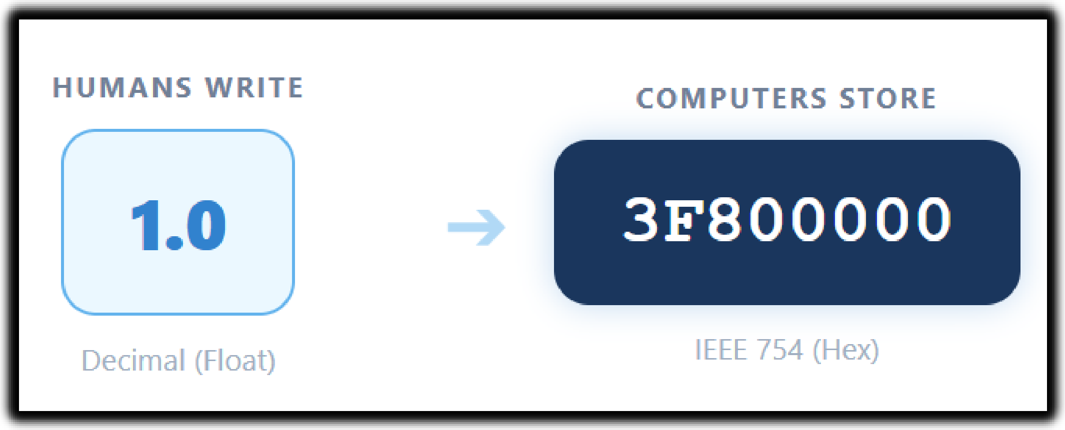
Important Idea (Very Important)

⚠️ **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

```
0011 1111 1000 0000 0000 0000 0000 0000
```

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

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

```
0011 → 3
1111 → F
1000 → 8
0000 → 0
```

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

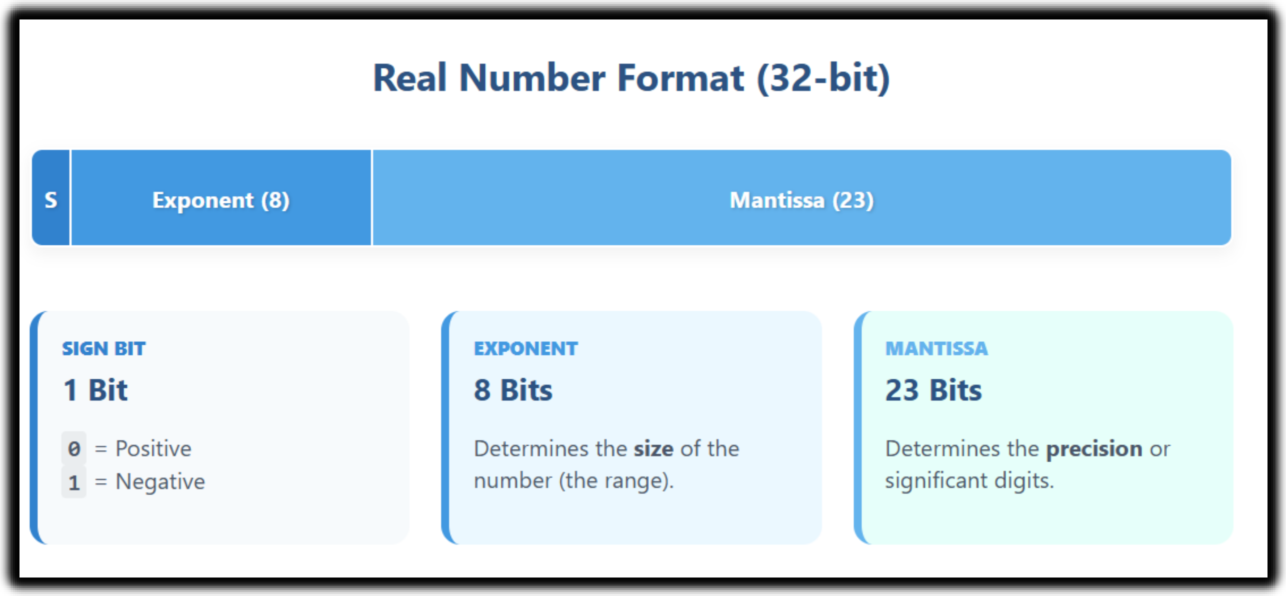
An **encoded real** is how a **computer stores a real number**

- 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