INTRINSIC DATA TYPES

What does “intrinsic data types” mean?

**Intrinsic data types** are the **built-in data sizes** that the assembler understands.

They answer three simple questions:

1. **How big is the data?** (8 bits, 16 bits, 32 bits, etc.)
2. **Is it signed or unsigned?** (can it be negative?)
3. **Is it an integer or a real (floating-point) number?**

That’s it. No magic.

What the assembler actually cares about

Here’s the key idea:

The assembler mainly cares about **size**.

It needs to know:

* how many bytes to reserve
* how many bytes an instruction will read or write

The assembler **does NOT strongly enforce**:

* signed vs unsigned

That distinction is mostly **for humans**.

Signed vs Unsigned (Important but subtle)

* DWORD → 32-bit **unsigned**
* SDWORD → 32-bit **signed**

Both:

* are **32 bits**
* take up **4 bytes**
* look identical in memory

The only difference is **how *you* interpret the bits**

That’s why programmers often use SDWORD:

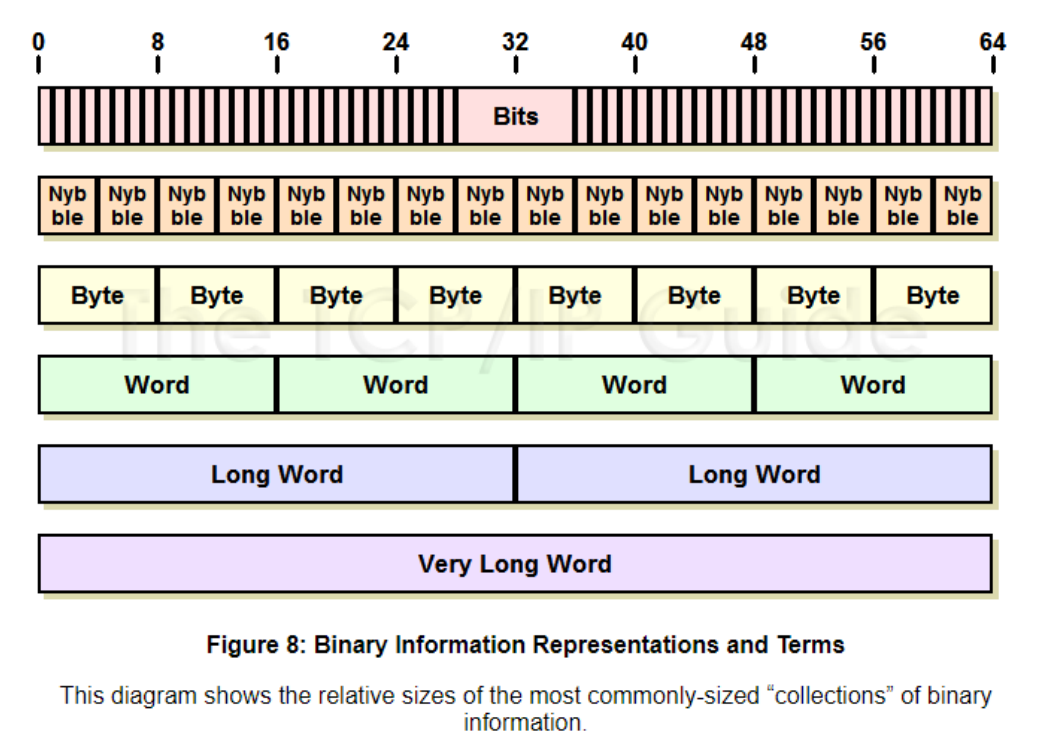
* not because the assembler demands it
* but because it makes intent clear

Why intrinsic data types matter

Intrinsic data types help you:

* choose the correct **operand size**
* avoid reading or writing the wrong number of bytes
* understand how values are stored in memory

If you get the size wrong, the CPU will still execute —   
but your result may be **wrong or corrupted**.



Key Takeaways

Intrinsic data types describe the **size**, **signed/unsigned nature**, and whether the value is an **integer or real number**.

The assembler cares about **operand size**, but does **not enforce signed vs unsigned**.

Programmers often use SDWORD to indicate signedness, but it is **not required**.

Intrinsic data types help explain how data is stored and used in assembly.

About overlapping types (Very important concept)

Some types overlap in functionality.

Example:

* DWORD → 32-bit unsigned
* SDWORD → 32-bit signed

Same size. Same memory.  
Different **meaning**.

The assembler sees “32 bits”.  
The programmer sees “signed” or “unsigned”.

So when I say “intrinsic data types”…

Yes — you mean **the ones in that image**.

These are the **basic building blocks** of all data in a computer.

Let’s walk through them naturally.

Bit-Level Building Blocks (From smallest to bigger)

* **Bit**  
  A single 0 or 1. The smallest possible unit of data.
* **Nibble (4 bits)**  
  Half a byte. One hexadecimal digit fits here.
* **Byte (8 bits)**  
  Stores:
  + a character
  + a small number  
    This is the most common basic unit.
* **Word (16 bits)**  
  Twice a byte. Used for larger numbers.
* **Double Word (32 bits)**  
  Four bytes. Very common in 32-bit programs.
* **Quad Word (64 bits)**  
  Eight bytes. Used for very large numbers.

Everything else is built from these.

Intrinsic Data Types in Assembly

**Integer types**

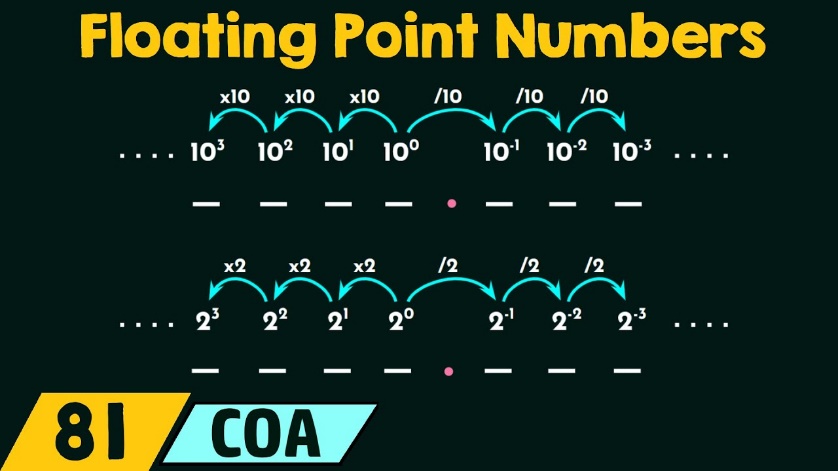
* **BYTE**  
  8-bit **unsigned** integer  
  Range: 0 to 255
* **SBYTE**  
  8-bit **signed** integer  
  Range: –128 to 127
* **WORD**  
  16-bit **unsigned** integer  
  Range: 0 to 65,535
* **SWORD**  
  16-bit **signed** integer  
  Range: –32,768 to 32,767
* **DWORD**  
  32-bit **unsigned** integer  
  Range: 0 to 4,294,967,295
* **SDWORD**  
  32-bit **signed** integer  
  Range: –2,147,483,648 to 2,147,483,647

Larger / special integer types

* **FWORD (48 bits)**  
  Used mainly for **far pointers** (old protected-mode stuff)
* **QWORD (64 bits)**  
  Very large integers
* **TBYTE (80 bits)**  
  Rarely used  
  Mostly related to the floating-point unit

Floating-point (real numbers)

* **REAL4**  
  32-bit floating-point  
  Common for basic decimal values
* **REAL8**  
  64-bit floating-point  
  Higher precision
* **REAL10**  
  80-bit floating-point  
  Very high precision, rarely used



Final idea

The assembler cares about **how many bytes**.  
The programmer cares about **what those bytes mean**.

That’s why intrinsic data types exist.

DATA DEFINITIONS (ASSEMBLY VARIABLES)

A **data definition** in assembly is how you create a variable.

It answers two questions:

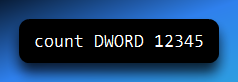
1. **How much memory do I need?**
2. **What value should it start with?**

General syntax



* **label** → the variable name (optional, but almost always used)
* **directive** → the data type / size
* **value** → the initial value

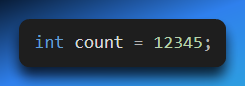
Example



This means:

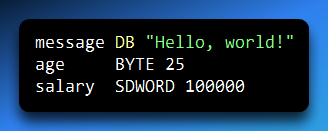
* create a variable named count
* reserve 4 bytes (32 bits)
* store the value 12345 in it

Equivalent C code:



Same idea, different language.

More examples



What’s happening here:

* message
  + DB reserves **1 byte per character**
  + "Hello, world!" takes **13 bytes**
* age
  + BYTE reserves **1 byte**
  + stores the value 25
* salary
  + SDWORD reserves **4 bytes**
  + stores a signed integer value

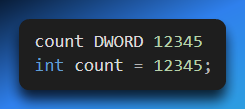
Why the data type matters

The assembler **must know the size** of the variable:

* how many bytes to reserve
* how many bytes instructions should read or write

If you don’t specify the type, the assembler has no idea what to do.

**Assembly vs C (Same concept)**



Both:

* reserve memory
* assign an initial value
* give the memory a name

Assembly just makes the size explicit.

Short forms (Just aliases)

These are **short names**, not new types:

* BYTE → DB
* WORD → DW
* DWORD → DD
* QWORD → DQ
* TBYTE → DT

They all do the same job: **reserve memory**.

Legacy Data Directives (Still used in 2026?)

Yes — **absolutely still used**.

Directives like DB, DW, DD, DQ, and DT are:

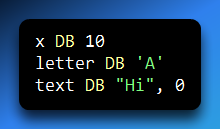
* still supported
* still common
* still the standard way to define data in MASM

They are called “legacy” only because they’ve been around forever —  
not because they’re obsolete.

The Core Data Directives (Explained Clearly)

1. DB — Declare Byte (8 bits)

Reserves **1 byte** per value.

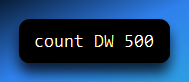


Common uses:

* characters
* small numbers
* strings (byte-by-byte)

2. DW — Declare Word (16 bits)

Reserves **2 bytes**.

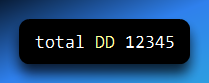


Used for:

* 16-bit values
* older or compact data

3. DD — Declare Doubleword (32 bits)

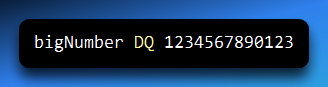
Reserves **4 bytes**.



This is one of the **most common** directives in 32-bit programs.

4. DQ — Declare Quadword (64 bits)

Reserves **8 bytes**.

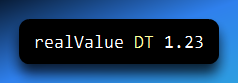


Used for:

* large integers
* 64-bit values

5. DT — Declare Ten Bytes (80 bits)

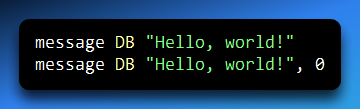
Reserves **10 bytes**.



Used for:

* extended precision floating-point (FPU)
* rare, but valid

About strings and null terminators



Both are valid.

The second one:

* adds a **null terminator**
* is better when interacting with C-style functions

MASM does **not** automatically add 0 for you.

Big Idea to Remember

Data definition directives:

* reserve memory
* define size
* optionally initialize values

The assembler:

* assigns addresses
* tracks them in the symbol table
* replaces variable names with real memory locations

You write **names**.  
The assembler handles **addresses**.

*Data definitions are how assembly creates variables — by explicitly stating how many bytes to reserve and what value to store in them.*

Defining Data Types (Part 1 – Beginner Explanation)

Big Picture: What This Section Is About

This section explains:

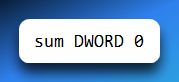
* How variables are **defined** in assembly
* How variables are **initialized**
* What happens if variables are **not initialized**
* How different **byte-sized data types** work

Main Rules for Data Definitions

1. At Least One Initializer Is Required

When you define a variable, the assembler expects **a value**.

Example:

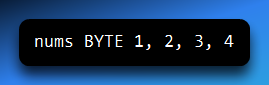


* DWORD → data type (4 bytes)
* 0 → initializer

Even zero counts as a valid initializer.

2. Multiple Initializers Use Commas

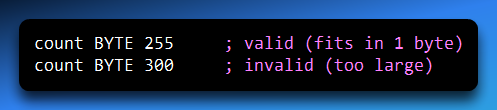
You can define **multiple values** at once by separating them with commas. Example:



This creates **four bytes** in memory.

3. Integer Initializers Must Match the Data Size

For integer data types, the value must **fit in the size** of the variable. Example:



4. Leaving a Variable Uninitialized (?)

If you want to reserve memory **without giving it a value**, use ?.

Example:



This means:

* Memory is reserved
* The value is unknown (garbage) at program start

⚠️ **Important:** Uninitialized variables **must not be used** before assigning a value.

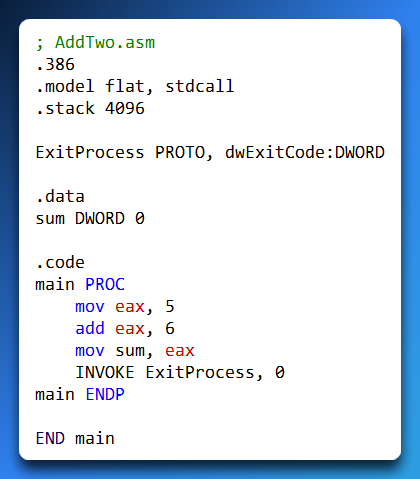
5. Everything Becomes Binary

No matter how you write an initializer:

* Decimal
* Hex
* Character literal

👉 The assembler converts it into **binary** before storing it in memory.

6. Worked Example: Adding Two Numbers



Defines a variable: **sum DWORD 0**

sum is a 4-byte integer initialized to 0; the program loads 5 into eax, adds 6 to it so eax becomes 11, and then stores the result: **mov sum, eax**

The program exits and final value is 11.

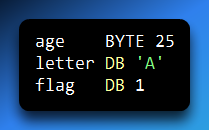
7. Debugging Tip   
To observe the variable, set a breakpoint after mov sum, eax, step through the instructions, and watch sum in the debugger to see the memory value change in real time.

BYTE-SIZED DATA TYPES (Very Important)

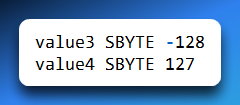
BYTE / DB (Unsigned, 8 bits)

* Size: **1 byte (8 bits)**
* Range: **0 to 255**
* Used for: small numbers, characters, raw data

**Examples:**

**SBYTE** is a signed 8-bit data type that occupies 1 byte of memory, can store values from −128 to +127, and is commonly used for small numbers that may be negative (for example: temp SBYTE -10 or change SBYTE 5).



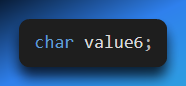
Signed vs Unsigned

* **Unsigned** → only positive values (and zero)
* **Signed** → positive **and** negative values

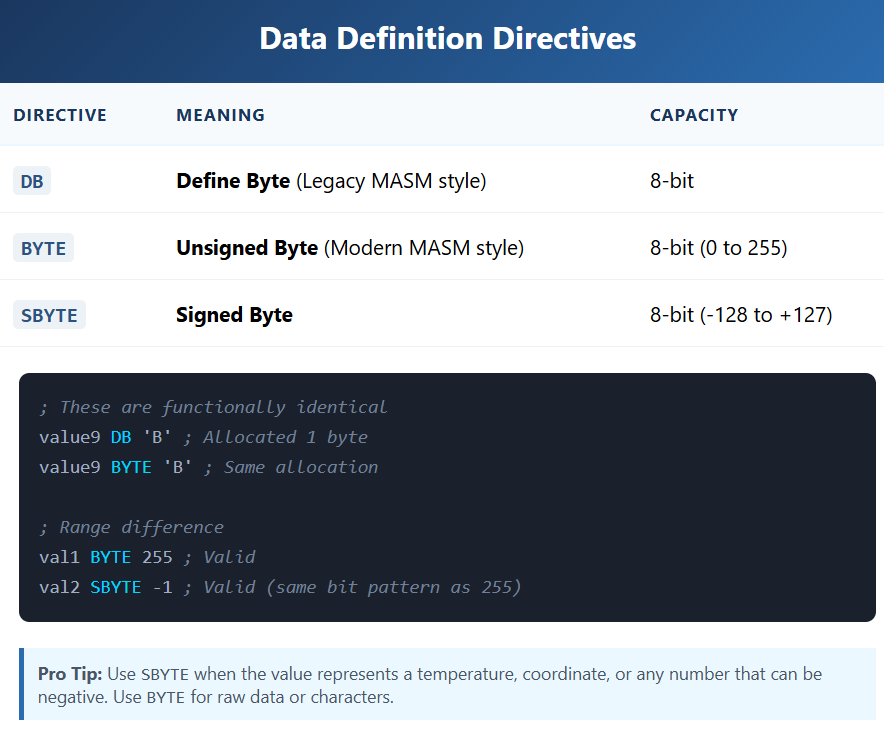
Uninitialized Variables (Important Warning)



Reserves 1 byte of memory but does not initialize it, so the value stored is random garbage just like



…in C language, which is why you must always initialize variables before using them.

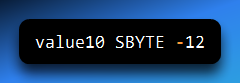


Character Initialization Example



* 'B' is a character
* ASCII value of 'B' = **66**
* Stored as **one byte**

Signed Byte Example



* Stores -12
* Uses signed representation
* Can hold negative values

Key Takeaways (Exam-Ready)

* Variables must have an initializer (or ?)
* ? means uninitialized (garbage value)
* BYTE / DB = unsigned 8-bit
* SBYTE = signed 8-bit
* Character literals are stored as ASCII values
* All data becomes binary in memory

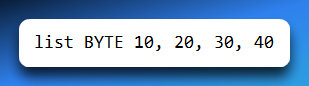
💡 **Defining a variable means reserving memory and deciding how the bits should be interpreted.**

DATA DEFINITION PART 2: ARRAYS & SIZES

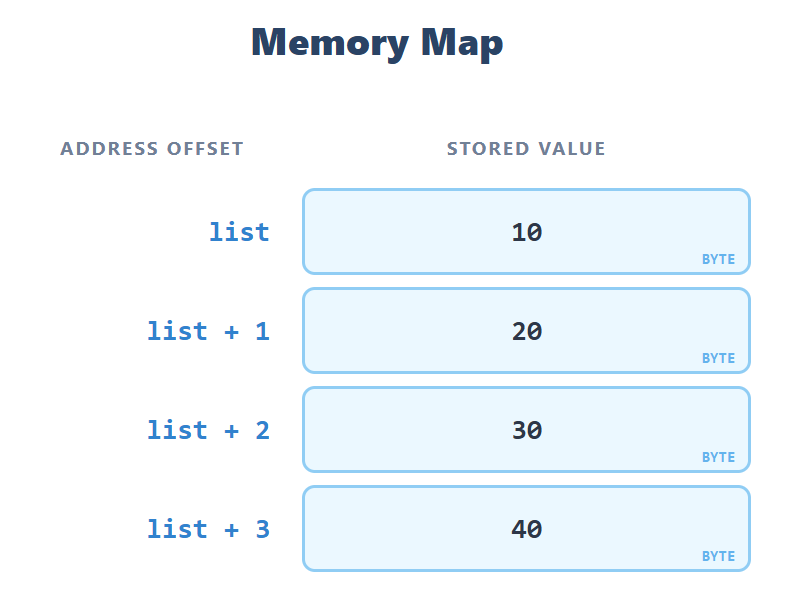
In high-level languages like C++ or Python, you create an array with brackets []. In Assembly, you just list values one after another.

Creating Arrays (The "Label" Trick)

When you define multiple values under one name, you are creating an array.



You are creating **4 bytes in memory**:



* The **label list only points to the first value**, which is 10.
* The assembler doesn’t automatically give names to the other values (20, 30, 40).
* To access them, you have to calculate their position relative to list.

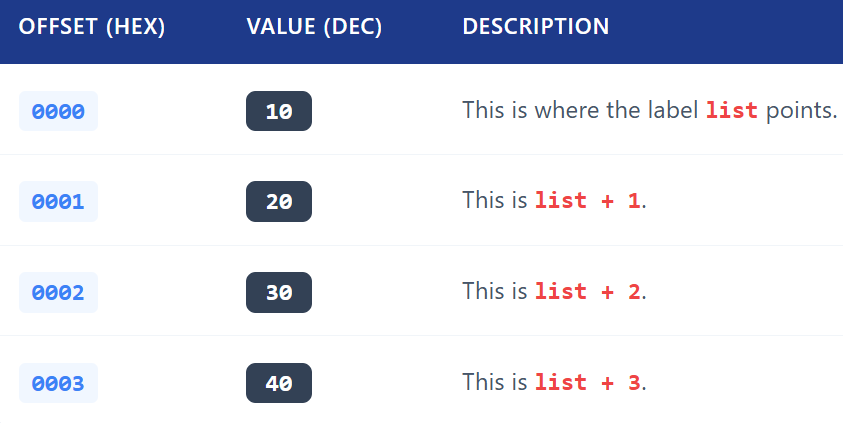
For example:

* list → gives you 10
* list + 1 → gives you 20
* list + 2 → gives you 30
* list + 3 → gives you 40

So, the **label is like the “starting address” of your array**, and the other elements are reached by adding an offset in bytes.

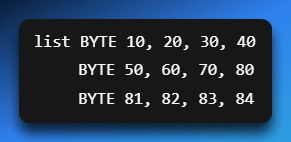
The Memory Map:

If list starts at memory offset **0000**:



Contiguous Memory

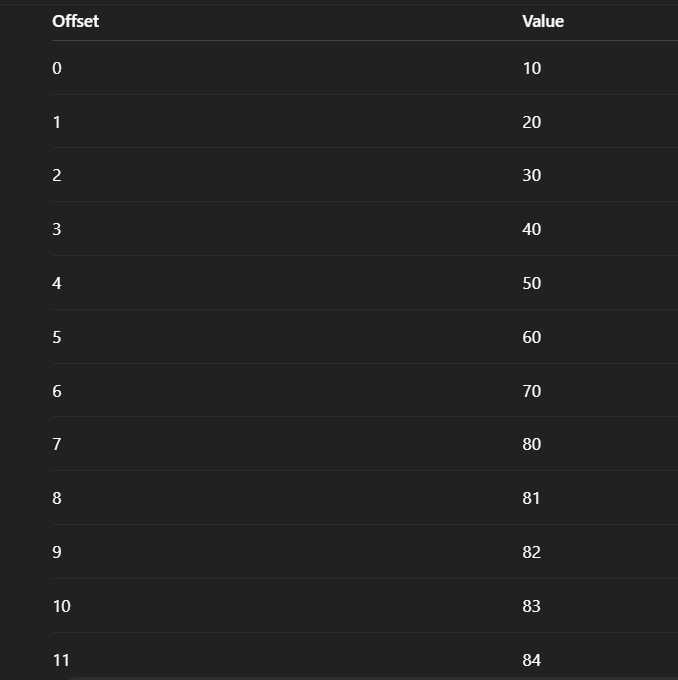
When you write:



here’s what’s happening:

* The assembler **doesn’t care about line breaks**.
* As long as you **don’t give a new label**, it just keeps placing the numbers **right after the previous ones in memory**.
* So all 12 numbers are stored **one after another** in memory.

Memory layout looks like this:



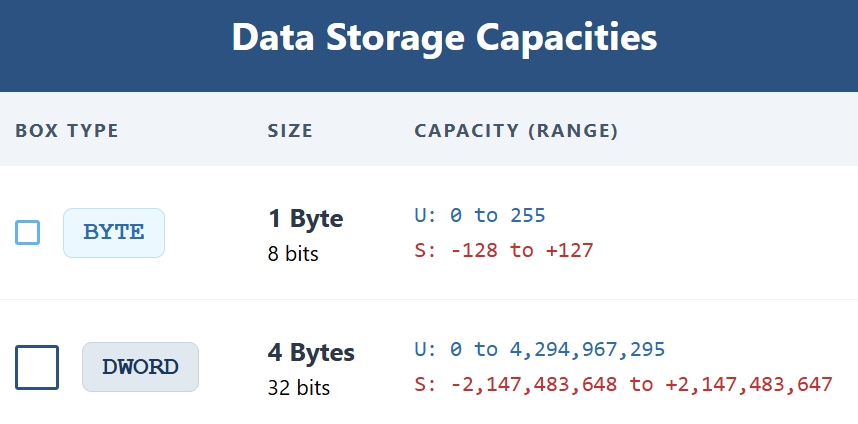
* The **label list points only to the first number (10 at offset 0)**.
* To access the others, you use **offsets**: list + 1 → 20, list + 4 → 50, etc.
* To the computer, this is **just one long strip of memory**, like a long row of boxes.

BYTE vs INTEGER Confusion **🤯**

Many beginners get confused because:

* In **C++/Java**, int is always **4 bytes (32 bits)**.
* In **Assembly**, numbers don’t have a fixed size by default. They are stored in a **container (data type) you choose**.

Think of it like **boxes**:



* **Number 10** fits easily in a BYTE (8-bit box).
* You **don’t need a DWORD (4-byte box)** for such a small number.
* U is unsigned, S is signed.

Why use BYTE instead of DWORD?

1. **Memory efficiency**:
   * 1,000 small numbers (like ages 0–100) → 1,000 bytes with BYTE, but 4,000 bytes with DWORD.
   * Saving 75% of memory!
2. **Compatibility**:
   * Some old hardware or file formats expect data to be **in bytes**.

**⚠️** The Catch

* If you try to put a number bigger than 255 into a BYTE:
  + The assembler will **give an error**, or
  + It might **silently chop off the extra bits**, giving you the **wrong value**.

✅ In short:

* You can spread your data across multiple lines; the assembler just packs them in a row.
* BYTE is just a small container—use it when the number is small.
* Integers in assembly are **as big as you declare** (BYTE, WORD, DWORD, etc.), unlike high-level languages.

MIXING RADIXES (THE "SALAD BOWL")

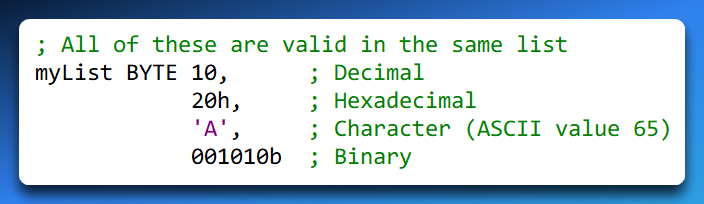
Assembly doesn't care how you write the number.



You can mix Hex, Decimal, Binary, and Character literals in the same list.

They all get converted to binary in the end.





Big Idea to Remember

* **Labels point to the start:** list is just the address of the first item. To get the rest, you add to the address (Offset).
* **Contiguous Memory:** Data defined sequentially sits sequentially in RAM.
* **Size matters, not type:** You can store an "integer" in a BYTE as long as it fits (0-255). You don't always need a DWORD.

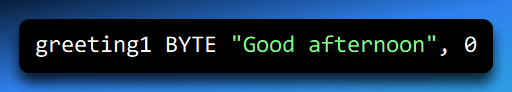
STRINGS

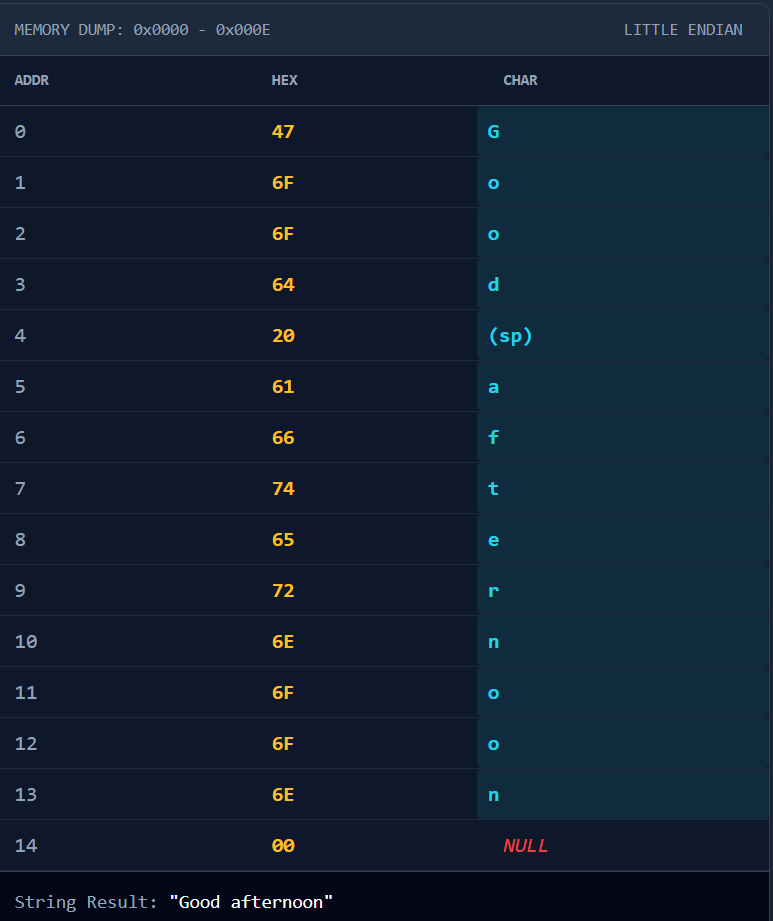
Strings Are Just Arrays of Bytes

In assembly, there is **no “string type”** like in high-level languages (C, Python, etc.).

* A **string is just a sequence of bytes**.
* Each **character** in the string is stored in **one byte**.
* The byte holds the **ASCII value** of the character.

Example:

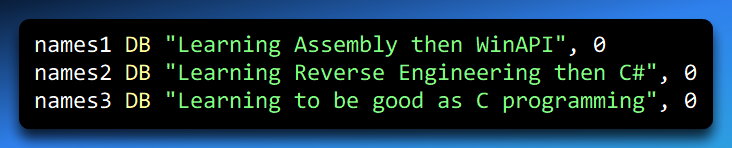




✅ Notice that:

* Each **character takes 1 byte**.
* The **null terminator (0)** is also a **single byte** marking the end of the string.

Labels Are Just Starting Addresses

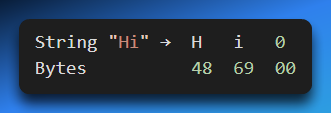


* names1, names2, names3 are **labels**.
* A label is **just a pointer to the first byte** of the string in memory.
* The computer uses the label as a **starting reference**, but it doesn’t know the length of the string unless you tell it.
* Everything after the first byte is **contiguous memory** (like we discussed with list BYTE 10,20…).

Why We Use BYTE

* We write BYTE because each **character fits in 1 byte**.
* Strings are really **arrays of bytes**, not a special datatype.

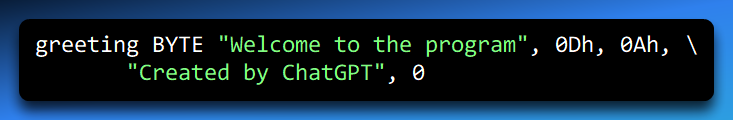
Think of it like this:



* Each character is **stored in one box** (byte).
* The **null byte** (0) is the **stop signal** for string functions, like printf in C or WinAPI string routines.

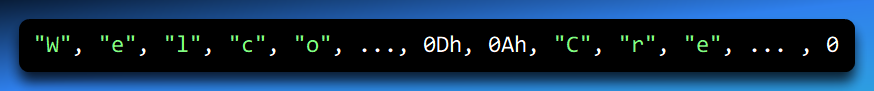
Multi-line Strings & Special Characters

You can split strings across multiple lines or add special characters:



* 0Dh = **carriage return** (CR) → moves cursor to start of line
* 0Ah = **line feed** (LF) → moves cursor down a line
* \ → line continuation character (lets you break one string across multiple lines)

Memory layout is still **just a sequence of bytes**, now including CR/LF:

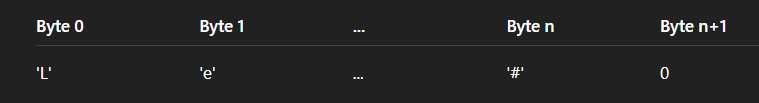


Everything remains a **byte**.

Putting It All Together

1. Each string is a **contiguous sequence of bytes** in memory.
2. The **label points to the first byte**.
3. Each **character = 1 byte (ASCII code)**.
4. **Null terminator (0) = 1 byte** marks the end.
5. Multi-line strings or special characters like CR/LF are just **additional bytes** in the same array.

So even the biggest sentence like "Learning Reverse Engineering then C#" is just **a row of bytes**:



**✅** Key Insight

Strings in assembly are **not magical objects**.

* They are **arrays of bytes**.
* The **label is the pointer**.
* The **assembler only cares about memory**.
* Null terminators allow functions to **know where the string ends**.

DUP Operator (Duplicate Made Easy)

The **DUP operator** in assembly is all about **making copies**—it lets you allocate multiple pieces of memory and optionally initialize them with the same value.

Think of it as a “memory copy machine” for variables, arrays, strings, or even structures.

How it works:

* **Count:** How many times you want to repeat something.
* **Value:** What you want to repeat (it can be a number, a string, or even an uninitialized placeholder).

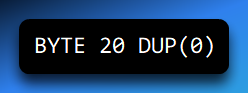
The syntax looks like this:



* <data type> could be BYTE, WORD, DWORD, etc.
* <count> is how many times you want to repeat.
* <value> is what you want to fill each slot with. If you leave it as ?, the memory is just reserved but contains random “garbage” values until you set it.

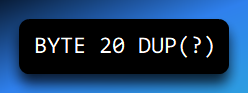
Examples:

Allocate 20 bytes, all zero:



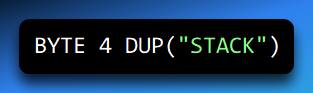
This creates a block of **20 bytes**, each containing 0.

Allocate 20 bytes, uninitialized:



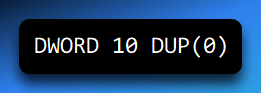
Memory is reserved for 20 bytes, but the values are **undefined**. Think of it like an empty box—you can fill it later.

Create a repeated string:



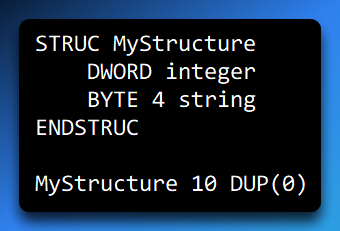
This repeats the sequence "STACK" four times in memory, effectively making "STACKSTACKSTACKSTACK".

Allocate an array of 10 integers, initialized to zero:



Here, you get **10 integers**, each 4 bytes, all set to 0.

Allocate an array of structures:



This reserves space for **10 structures**, each containing a 4-byte integer and a 4-byte string.

Key idea:

Yes, **DUP literally means “duplicate”**. It’s your way to **repeat a value or pattern** efficiently in memory without writing it out multiple times.

Whether you’re filling arrays, initializing strings, or creating structures, **DUP saves time, space, and effort**.

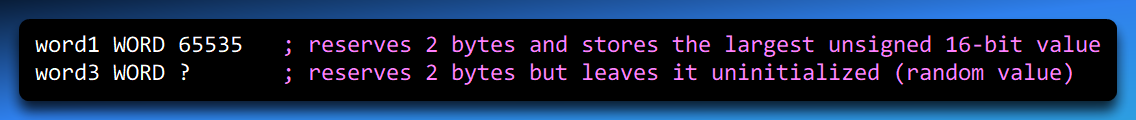
Think of it like telling the assembler: *“Hey, make 10 of this, or 20 of that, all lined up in memory, and set them to this value—or leave them blank for now.”*

WORD and SWORD

In assembly language, **WORD** and **SWORD** are used to work with **16-bit numbers**. Each 16-bit number takes **2 bytes** of memory.

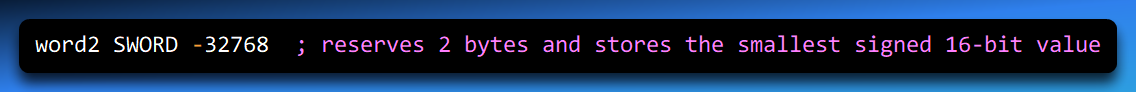
WORD (Unsigned 16-bit Integer)

* **WORD** is for **unsigned numbers**, meaning only positive numbers from 0 to 65535.
* Each WORD reserves **2 bytes** in memory.



SWORD (Signed 16-bit Integer)

* **SWORD** is for **signed numbers**, meaning it can store **negative and positive numbers** from -32768 to 32767.
* Each SWORD also takes **2 bytes** in memory.
* Example:



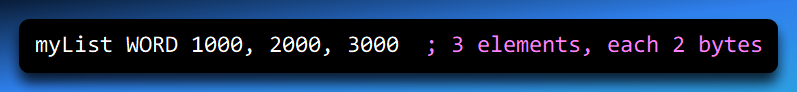
Key Idea:

Think of **WORD as a box that only holds positive numbers**, and **SWORD as a box that can hold negative numbers too**. Both boxes are **16 bits** (2 bytes) wide, so the memory size is the same, only the interpretation changes.

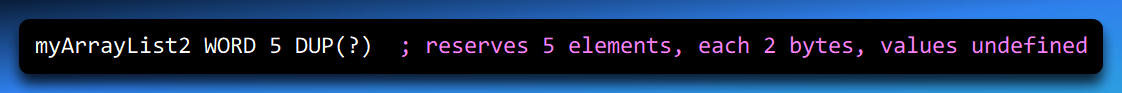
WORD Arrays

You can create **arrays of 16-bit numbers** in assembly, just like arrays in C, using either **explicit listing** or the **DUP operator**.

* **Memory layout:** Each 16-bit element occupies **2 bytes**. So if your array starts at memory offset 0000, the next element is at 0002, then 0004, and so on.
* **Example with explicit listing:**

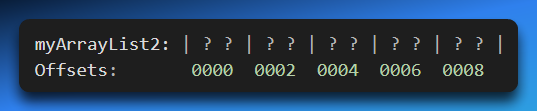


Example with DUP (uninitialized array):



Here, ? means the elements are **uninitialized**. They have random “garbage” values until your code sets them.

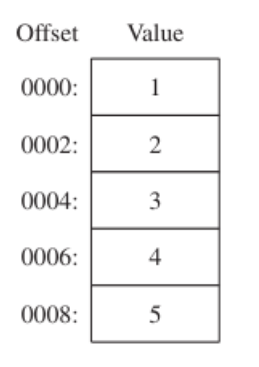
Visualizing Memory (Conceptual):



Each element takes **2 bytes**, so to access the next element, you **increment the offset by 2**.

Summary:

* **WORD:** Unsigned 16-bit number (0 to 65535)
* **SWORD:** Signed 16-bit number (-32768 to 32767)
* Arrays: Use **listing or DUP** to store multiple words, remembering each takes 2 bytes in memory.

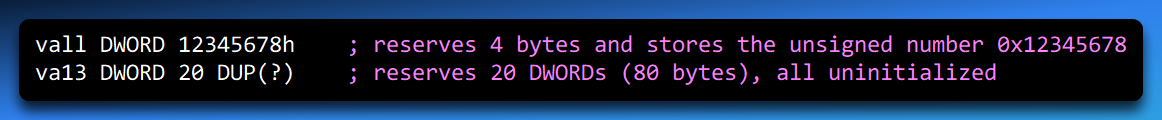


DWORD and SDWORD

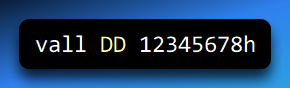
In assembly language, **DWORD** and **SDWORD** are used to work with **32-bit integers**. Each 32-bit number takes **4 bytes** of memory.

I. DWORD (Unsigned 32-bit Integer)

* **DWORD** is for **unsigned numbers**, meaning only positive numbers from 0 to 4,294,967,295.
* Each DWORD reserves **4 bytes** in memory.

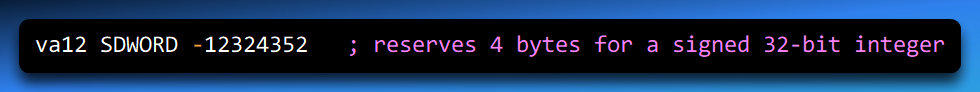


**Usage Tip:** You can also use **DD** (Define Doubleword) as a legacy directive. It works the same as DWORD:



II. SDWORD (Signed 32-bit Integer)

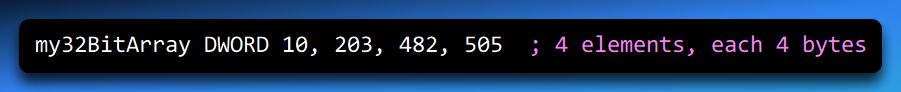
* **SDWORD** is for **signed numbers**, meaning it can store **negative and positive numbers** from -2,147,483,648 to 2,147,483,647.
* Each SDWORD also takes **4 bytes** in memory.



III. Arrays of 32-bit Numbers

You can create arrays of DWORDs or SDWORDs either by listing values explicitly or using the **DUP operator**:

**Explicit initialization:**



**Uninitialized array using DUP:**

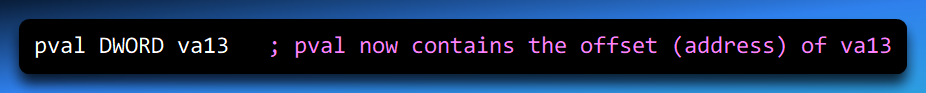


**Memory layout concept:**

* Each element occupies **4 bytes**, so if the first element is at offset 0000, the next is at 0004, then 0008, and so on.
* Arrays let you easily store multiple 32-bit numbers in **contiguous memory**.

IV. Extra Tip: DWORD for Offsets

You can also use **DWORD** to store the **32-bit memory offset of another variable**:



This is useful for pointers or referencing other variables in memory.

V. Summary:

* **DWORD:** Unsigned 32-bit integer, 4 bytes, 0 → 4,294,967,295
* **SDWORD:** Signed 32-bit integer, 4 bytes, -2,147,483,648 → 2,147,483,647
* Arrays: Use listing or **DUP** to store multiple DWORDs
* Legacy DD directive works the same as DWORD

QWORD (Quadword)

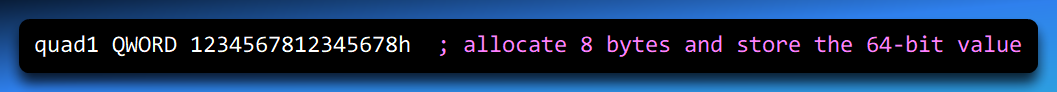
The **QWORD directive** in assembly language is used to allocate storage for **64-bit values**, meaning each QWORD takes **8 bytes** of memory.

Think of it as a really big box that can hold very large numbers.

1. Syntax and Usage

You can define QWORD values in two ways:

**Standard directive:**

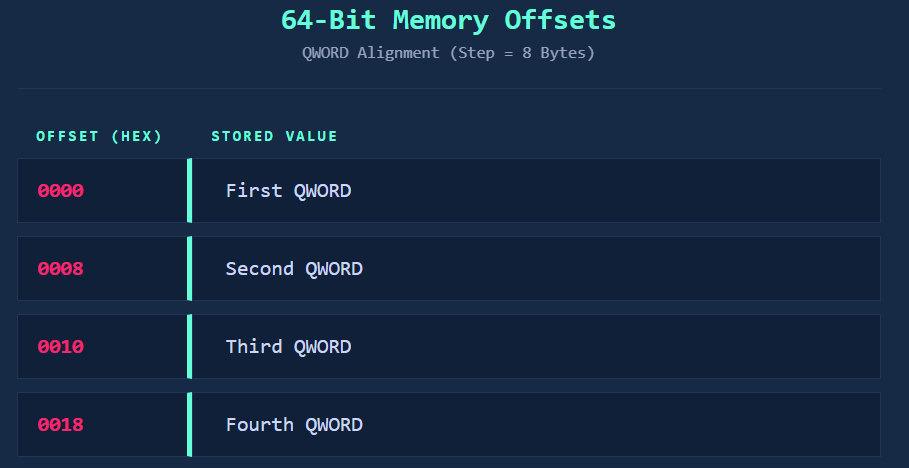
**Short form (DQ – Define Quadword):**



**Tip:** The value must fit in 64 bits, otherwise the assembler will throw an error.

2. Memory Organization

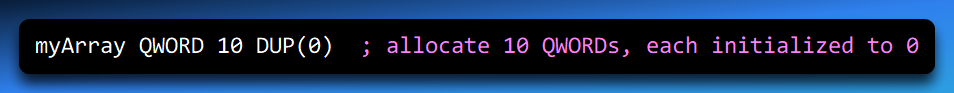
Each QWORD takes **8 bytes**, so if you define multiple QWORDs in an array, memory offsets increase by 8 each time:



This is just like how **DWORD arrays** worked, but each element is double the size.

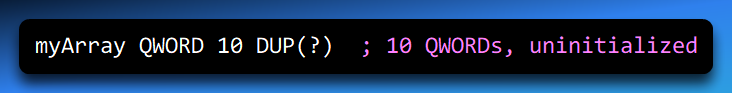
3. Arrays of QWORDs

Just like with BYTE or DWORD, you can use the **DUP operator** to define multiple QWORDs at once:



Each element is **8 bytes**, so this reserves **80 bytes total** (10 × 8).

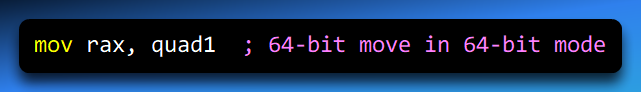
Using ? instead of 0 leaves them uninitialized:



4. QWORD and Registers

* In **32-bit mode**, your registers like **EAX** are 32 bits, so storing a 64-bit QWORD might need **two 32-bit registers** or special memory instructions.
* In **64-bit mode**, the **RAX** register can hold a full QWORD directly.

Example:



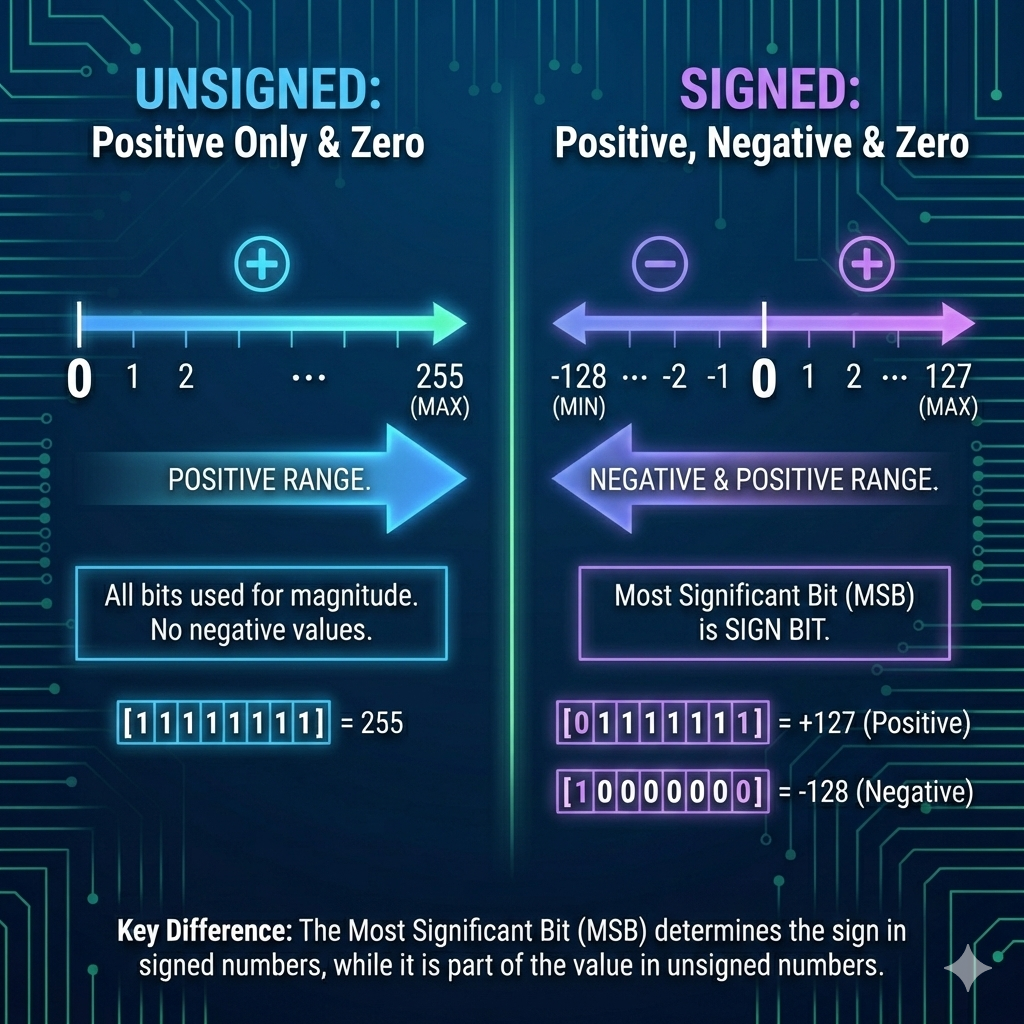
* This is important if you start working with large numbers, addresses, or high-precision calculations.

5. Summary Notes

* QWORD = 64 bits = 8 bytes
* QWORD can be initialized directly or with DUP
* Short form DQ is equivalent to QWORD
* Arrays increment in memory by 8 bytes per element
* 32-bit registers can’t hold QWORDs directly; use 64-bit registers or split into two 32-bit halves

💡 **Memory efficiency tip:**   
*Use QWORD only when you need numbers bigger than 32 bits, otherwise DWORD is enough and takes half the memory.*

Never forget this concept in Assembly:



Let’s continue….

PACKED BCD AND TBYTE

Packed BCD (Binary Coded Decimal) is a **special way to represent decimal numbers in binary**, designed for **efficiency and precision**, especially in financial or scientific applications.

1. What is Packed BCD?

**Packed BCD** stores decimal digits in **pairs**, with **two decimal digits per byte**.

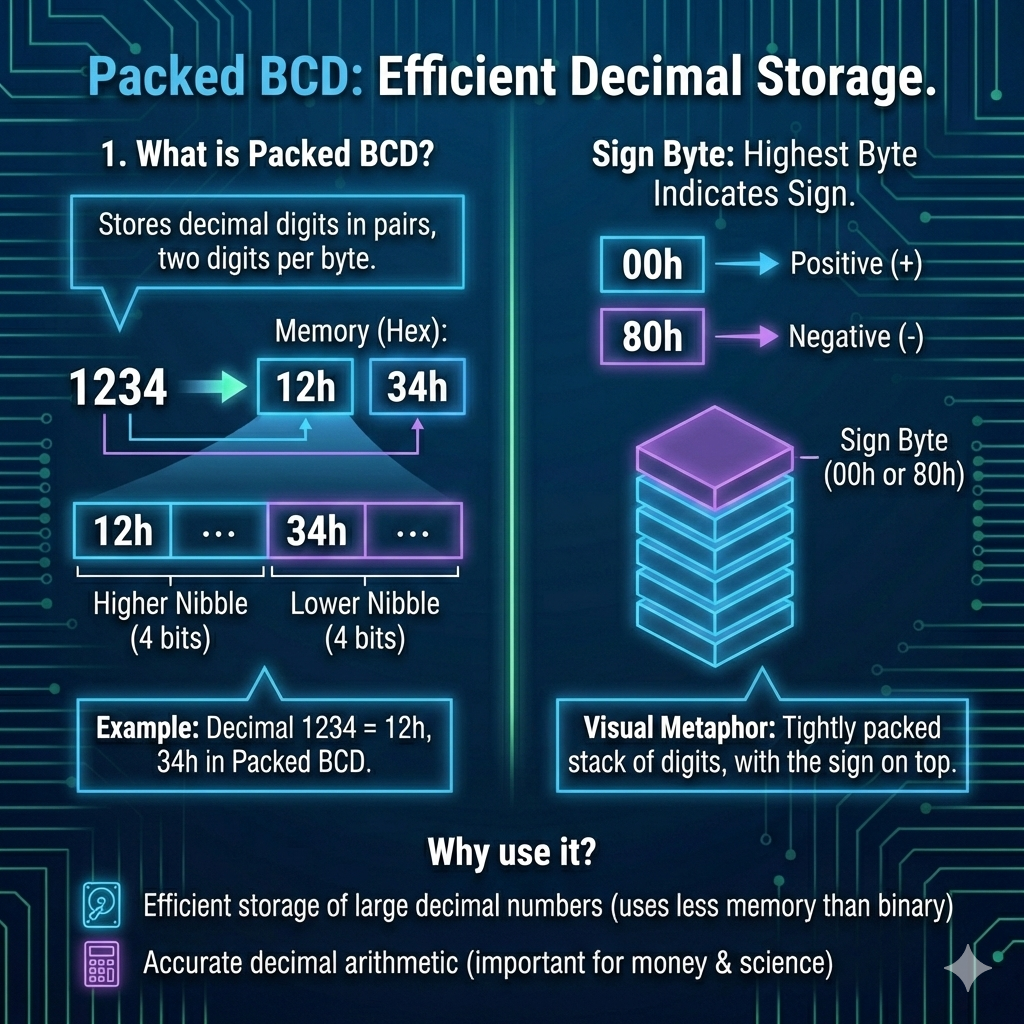
Example: The decimal number 1234 in packed BCD is stored as **34 12** (hex representation in memory).

* The **lower nibble** of a byte stores one digit.
* The **higher nibble** stores the next digit.

**Sign byte:** The highest byte of a packed BCD variable indicates the sign.

* 00h → Positive
* 80h → Negative

Think of it like a **tightly packed stack of digits**, with the sign sitting on top.



Why use it?

* Efficient storage of large decimal numbers (takes less memory than converting to binary integers).
* Accurate decimal arithmetic — important for **money calculations**, **scientific data**, and some **embedded systems**.

2. The TBYTE Directive

In MASM, **TBYTE** is used to declare variables that can store **packed BCD data**.

Even though TBYTE is **80 bits (10 bytes)**, it isn’t just “10 bytes of storage” — it can also hold **floating-point numbers** or other data formats.

**Memory layout for a TBYTE BCD number:**

* **1st byte:** Sign
* **Next 9 bytes:** Decimal digits, 2 digits per byte

Example: Declaring a packed BCD variable