Contents

INTRODUCTION: ASSEMBLY LANGUAGE X86 TOPICS

Basic Concepts: Applications of assembly language, basic concepts, machine language, and data representation.

x86 Processor Architecture: Basic microcomputer design, instruction execution cycle, x86 processor architecture, Intel64 architecture, x86 memory management, components of a microcomputer, and the input–output system.

Assembly Language Fundamentals: Introduction to assembly language, linking and debugging, and defining constants and variables.

Data Transfers, Addressing, and Arithmetic: Simple data transfer and arithmetic instructions, assemble-link-execute cycle, operators, directives, expressions, JMP and LOOP instructions, and indirect addressing.

Procedures: Linking to an external library, description of the book's link library, stack operations, defining and using procedures, flowcharts, and top-down structured design.

Conditional Processing: Boolean and comparison instructions, conditional jumps and loops, high- level logic structures, and finite-state machines.

Integer Arithmetic: Shift and rotate instructions with useful applications, multiplication and division, extended addition and subtraction, and ASCII and packed decimal arithmetic.

Advanced Procedures: Stack parameters, local variables, advanced PROC and INVOKE directives, and 2/31 recursion.

Strings and Arrays: String primitives, manipulating arrays of characters and integers, two-dimensional arrays, sorting, and searching.

Structures and Macros: Structures, macros, conditional assembly directives, and defining repeat blocks.

MS-Windows Programming: Protected mode memory management concepts, using the Microsoft-Windows API to display text and colors, and dynamic memory allocation.

Floating-Point Processing and Instruction Encoding: Floating-point binary representation and floating-point arithmetic. Learning to program the IA-32 floating-point unit. Under- standing the encoding of IA-32 machine instructions.

High-Level Language Interface: Parameter passing conventions, inline assembly code, and linking assembly language modules to C and C++ programs.

16-Bit MS-DOS Programming: Memory organization, interrupts, function calls, and standard MS- DOS file I/O services.

Disk Fundamentals: Disk storage systems, sectors, clusters, directories, file allocation tables, handling MS-DOS error codes, and drive and directory manipulation.

BIOS-Level Programming: Keyboard input, video text, graphics, and mouse programming.

Expert MS-DOS Programming: Custom-designed segments, runtime program structure, and Interrupt handling. Hardware control using I/O ports.

💻 ASCII Control Characters: Mnemonics & Labels

What they are

ASCII control characters are **non-printable characters**.
They do not show text on the screen.

Why they exist

They are used to **control devices**, not display letters.
Examples: screen, keyboard, printer, serial port.

How they work

- Generated by pressing combinations like Ctrl + C, Ctrl + G
- Sent as **numeric values**, not visible symbols
- The receiving device interprets them as commands

What they control

They are commonly used to:

- Move the cursor
- Start a new line
- Make a beep sound
- Reset positions in terminals or printers

Common examples

- **LF (Line Feed)** – move down one line
- **CR (Carriage Return)** – move cursor to start of line
- **BEL (Bell)** – beep sound
- **ESC (Escape)** – start control sequences

Representation

Each control character has:

- A **mnemonic** (e.g. BEL, CR, ESC)
- A **numeric value** (usually hex, e.g. 07h, 0Dh, 1Bh)
- A **defined behavior**

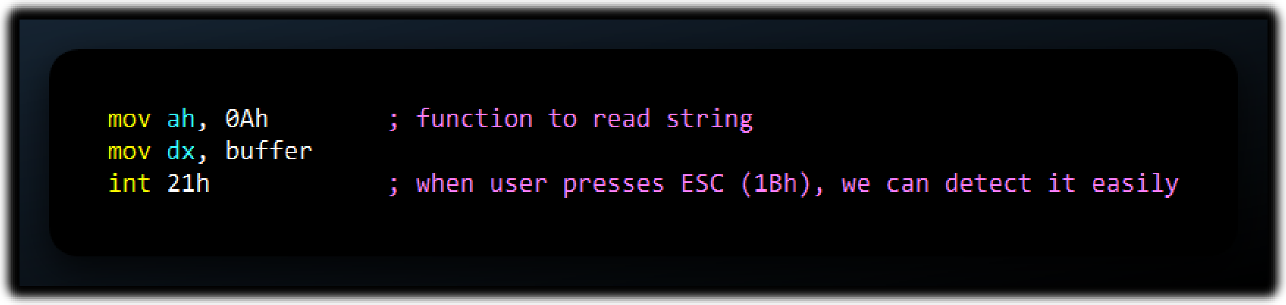
Important note

These are **not magic** and not special to assembly.
They are just agreed-upon signals that software and hardware understand.

Find the ASCII.html OR ASCII.png attached for the full table.

🤖 Why do mnemonics matter?

Why do mnemonics matter? 🤖
Mnemonics are like nicknames.
Instead of remembering what **1Bh** means, you use **ESC**.
This makes code easy to read and easier for your brain to understand.



```
mov ah, 0Ah          ; function to read string
mov dx, buffer
int 21h              ; when user presses ESC (1Bh), we can detect it easily
```

Mnemonics are like **street names**.
Hex codes are like **GPS numbers**.
Both go to the same place, but street names are easier for people.

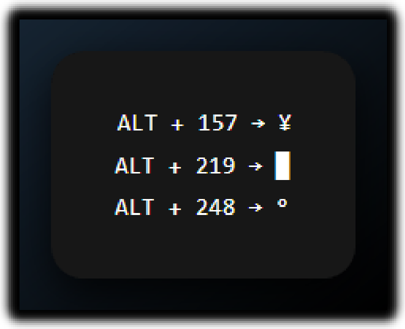
❗Ctrl + Hyphen (–)

Yes, **Ctrl + –** makes ASCII **1Fh**.
It is a real control character.
People don’t use it much, but it exists in the official ASCII list.

🔤 ALT Key Combinations

When you hold **ALT** and press a number or letter, the keyboard makes a special character. These are used to:

- Make keyboard shortcuts
- Type special symbols (©, ■, ü, etc.)
- Work with other languages or old programs



```
ALT + 157 → ¥
ALT + 219 → █
ALT + 248 → °
```

🛠️ Why they mattered (and still matter):

In old DOS and early Windows programs, people used them to:

- Draw text screens (like **ALT + 219** for blocks)
- Read keyboard input with **ALT** codes
- Make custom keyboard shortcuts

Today, modern apps don't use them much,
but they still work in **BIOS**, **command line**, and **low-level programs**.

Find the AltKeyCombinations.html OR AltKeyCombinations.png for the full table.

📦 Keyboard Scan Codes vs ASCII vs ALT Keys

(Know what your keyboard is really sending.)

1. Keyboard Scan Codes – Hardware Level

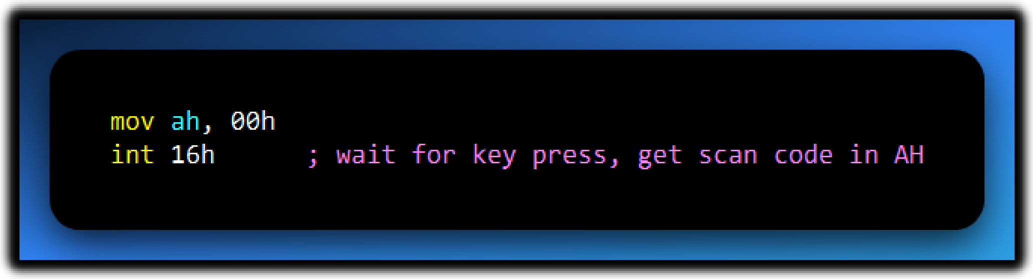
When you press a key (like **A**), the keyboard does not send the letter **A**.
It sends a **scan code**, which is a basic signal.

It is like saying: "Someone pressed this key spot on the keyboard."

These codes are numbers like **1Eh** for the **A** key.
The BIOS or operating system uses them to find out which key was pressed.

Scan codes are about **key position**, not the letter you see on the screen.

🛠 **Example:**



```
mov ah, 00h
int 16h   ; wait for key press, get scan code in AH
```

Find the KeyboardScanCodes.html OR KeyboardScanCodes.png for the full table.

🧩 2. Why this Matters (for ASM or Reverse Engineering)

Keyboard input is not always ASCII

- BIOS keyboard services (int 16h) return **scan codes**
- Scan codes tell you **which key was pressed**, not which character
- This matters for:
    - ➢ Function keys
    - ➢ Arrow keys
    - ➢ Key combinations

ASCII is for characters, not keys

- ASCII is used when working with **text**
- Examples:
  - ➢ Printing characters to the screen
  - ➢ Parsing input strings
  - ➢ Comparing text values
- ASCII does **not** represent physical keys

ALT combinations

- ALT key combinations can generate **specific ASCII values**
- Common in:
  - ➢ Old DOS programs
  - ➢ Terminal-style user interfaces
  - ➢ Custom input handling
- Useful when you want:
  - ➢ Non-standard characters
  - ➢ Direct control over input values

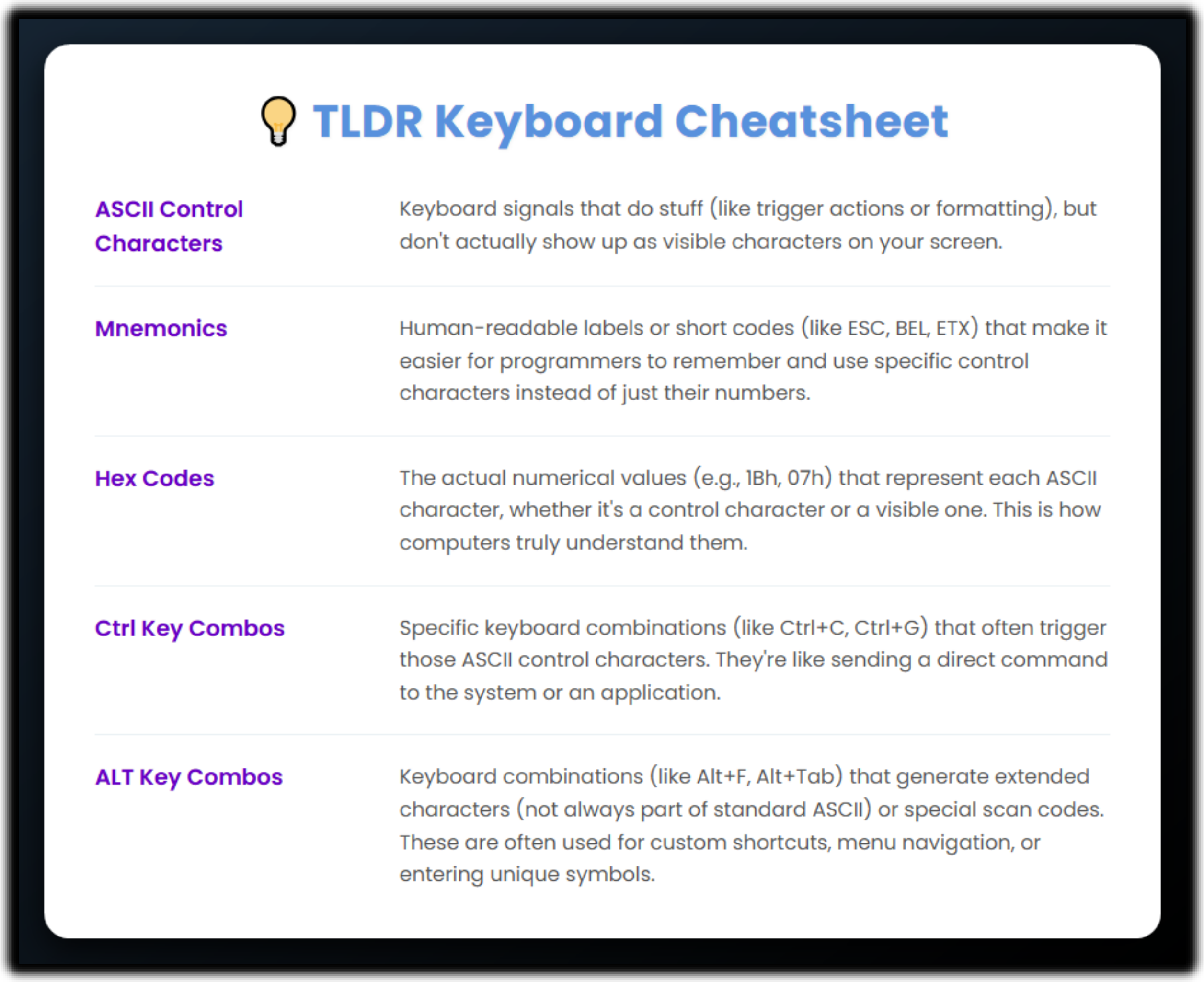
Reverse engineering relevance

- Programs may mix **scan codes** and **ASCII**
- Malware and old software often read raw keyboard input
- Understanding the difference helps you:
  - ➢ Read input-handling code correctly
  - ➢ Avoid misinterpreting key logic

Key Rule to Remember

Scan codes = physical keys
ASCII = characters

If you remember only that, you won't get lost.



💡 TLDR Keyboard Cheatsheet

ASCII Control Characters	Keyboard signals that do stuff (like trigger actions or formatting), but don't actually show up as visible characters on your screen.
Mnemonics	Human-readable labels or short codes (like ESC, BEL, ETX) that make it easier for programmers to remember and use specific control characters instead of just their numbers.
Hex Codes	The actual numerical values (e.g., 1Bh, 07h) that represent each ASCII character, whether it's a control character or a visible one. This is how computers truly understand them.
Ctrl Key Combos	Specific keyboard combinations (like Ctrl+C, Ctrl+G) that often trigger those ASCII control characters. They're like sending a direct command to the system or an application.
ALT Key Combos	Keyboard combinations (like Alt+F, Alt+Tab) that generate extended characters (not always part of standard ASCII) or special scan codes. These are often used for custom shortcuts, menu navigation, or entering unique symbols.

How Programming Languages Understand ASCII

💡 What is ASCII and why is it important?

ASCII is a system that gives letters and symbols a number.
For example:

- **A = 65**
- **! = 33**
- **3 = 51**

Computers only understand numbers (1s and 0s).
ASCII lets computers save and show text in a way people understand.

🧬 Character Encoding = Translation

Computers do not know what a letter is.
They only know numbers.

Character encoding is a translation system:

- You press **A**
- The keyboard sends a scan code
- The system turns it into **ASCII 65**
- The computer stores it as **01000001** (binary)

This lets:

- Programming languages work with text
- Operating systems show letters on the screen
- Programs save readable text in files

ASCII BASICS

ASCII uses **7 bits**, which allows **128 characters** total.

- **0–31** → Control characters
- **32–126** → Printable characters
- **127** → Delete (DEL)

🔧 1. Control Characters

These do not show on the screen. They control how text works.

Examples:

- **LF (10)** → New line
- **CR (13)** → Go to start of line
- **TAB (9)** → Tab space
- **BEL (7)** → Beep sound
- **DEL (127)** → Delete

These were very important in early computers.

🔤 2. Printable Characters

These are the characters you can see:

- Letters (A–Z, a–z)
- Numbers (0–9)
- Punctuation (?, ., ,)
- Symbols (@, #, $, %)

🧮 Binary and Hex

- ASCII numbers are often written in decimal (**65**)
- Inside the computer, they are binary (**01000001**)
- Programmers often use hex (**41h**) because it is shorter

Binary, decimal, and hex all mean the same value —
they are just different ways to write the number.

UNICODE: THE WORLD'S TEXT SYSTEM

ASCII only has **128 characters**.
That works for English, but not for other languages or emojis.

Unicode fixes this.
It supports:

- All languages
- Emojis 🙂
- Math symbols
- Rare and old scripts

Unicode includes ASCII as the **first 128 characters**,
so ASCII is part of Unicode.

🤔 What is Unicode?

Unicode is the modern system computers use to store text.
It works for **every language and symbol** in the world.

Unicode:

- Has **over 144,000 characters** (and growing)
- Supports more than **150 writing systems**
- Includes emojis and special symbols
- Is used in websites, apps, and operating systems

It exists because ASCII was too limited.

📦 Fixed vs Variable Length Encoding

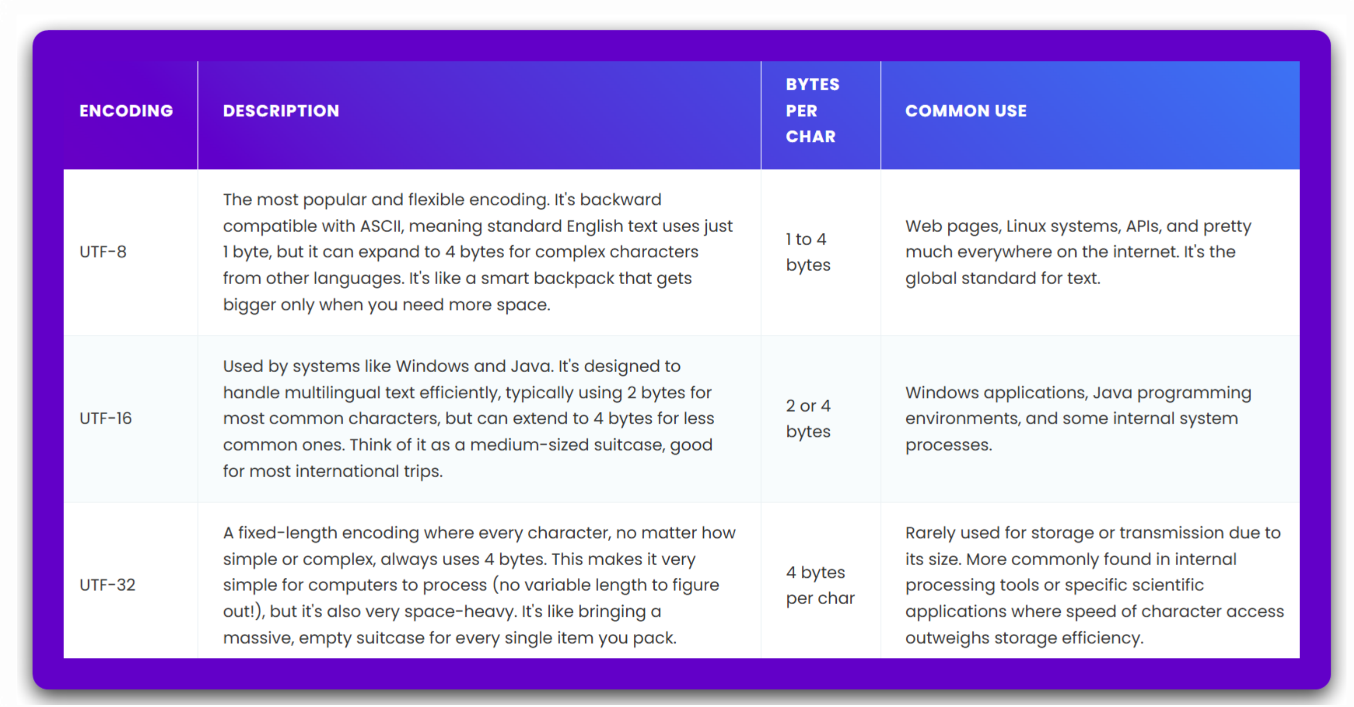
🔷 ASCII (Fixed Size)

- Uses **7 bits** per character
- Takes **1 byte** for each character
- Can only show **128 characters**
- Mostly English only

If you want, I can also simplify the **UTF-8 / UTF-16** part next.

🔷 Unicode: Variable-Length

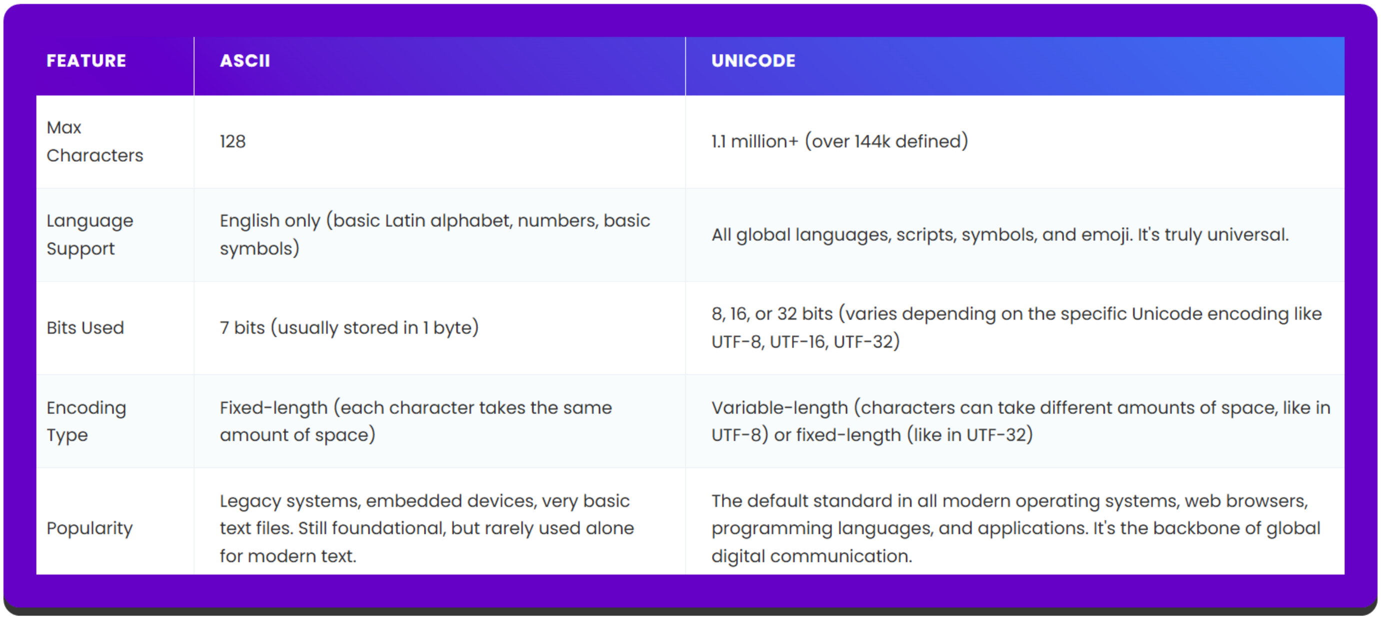
Unicode supports **multiple encodings** depending on the use case (See the UTF.html):



ENCODING	DESCRIPTION	BYTES PER CHAR	COMMON USE
UTF-8	The most popular and flexible encoding. It's backward compatible with ASCII, meaning standard English text uses just 1 byte, but it can expand to 4 bytes for complex characters from other languages. It's like a smart backpack that gets bigger only when you need more space.	1 to 4 bytes	Web pages, Linux systems, APIs, and pretty much everywhere on the internet. It's the global standard for text.
UTF-16	Used by systems like Windows and Java. It's designed to handle multilingual text efficiently, typically using 2 bytes for most common characters, but can extend to 4 bytes for less common ones. Think of it as a medium-sized suitcase, good for most international trips.	2 or 4 bytes	Windows applications, Java programming environments, and some internal system processes.
UTF-32	A fixed-length encoding where every character, no matter how simple or complex, always uses 4 bytes. This makes it very simple for computers to process (no variable length to figure out!), but it's also very space-heavy. It's like bringing a massive, empty suitcase for every single item you pack.	4 bytes per char	Rarely used for storage or transmission due to its size. More commonly found in internal processing tools or specific scientific applications where speed of character access outweighs storage efficiency.

⚠️ **UTF-8** is the king. It's compact, fast, compatible, and everywhere — from HTML files to JSON to APIs.

💬 ASCII vs Unicode: Key Differences



FEATURE	ASCII	UNICODE
Max Characters	128	1.1 million+ (over 144k defined)
Language Support	English only (basic Latin alphabet, numbers, basic symbols)	All global languages, scripts, symbols, and emoji. It's truly universal.
Bits Used	7 bits (usually stored in 1 byte)	8, 16, or 32 bits (varies depending on the specific Unicode encoding like UTF-8, UTF-16, UTF-32)
Encoding Type	Fixed-length (each character takes the same amount of space)	Variable-length (characters can take different amounts of space, like in UTF-8) or fixed-length (like in UTF-32)
Popularity	Legacy systems, embedded devices, very basic text files. Still foundational, but rarely used alone for modern text.	The default standard in all modern operating systems, web browsers, programming languages, and applications. It's the backbone of global digital communication.

🧩 From Text to Machine Code

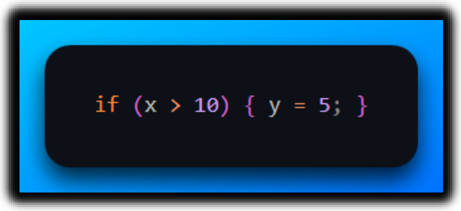
🔠 Step 1: Tokenization (Lexical Analysis)

When you write code, the computer does not understand it right away. First, it **splits the code into small pieces** called **tokens**. This step is called **lexical analysis**.

Think of it like this: A sentence is broken into **words** so it can be read. Code is broken into **tokens** so the computer can understand it. Tokens can be things like:

- Keywords
- Names
- Numbers
- Symbols

Example code:



```
if (x > 10) { y = 5; }
```

This gets broken down into tokens like:

- if → keyword
- (→ punctuation
- x → identifier
- > → operator
- 10 → constant
- { → block opener
- y → identifier
- = → operator
- 5 → constant
- ;, } → punctuation

🔎 Analogy: It’s like chopping a sentence into “The”, “quick”, “brown”, “fox”… so it’s easier to parse.

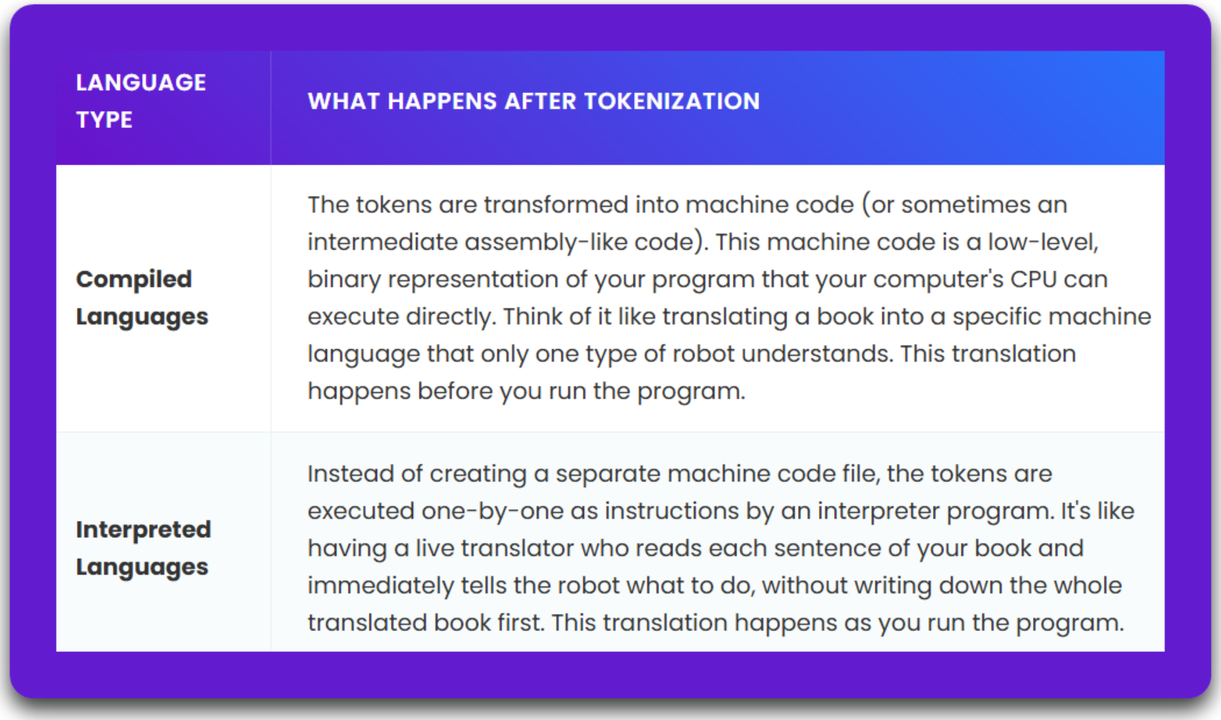
⚙️ Step 2: Compilation or Interpretation

After the code is broken into tokens,
the next step is to turn it into something the computer can run.

What happens next depends on the language:

- **Compiled languages** turn the code into a program first
- **Interpreted languages** run the code step by step

Either way, the computer is getting ready to **execute** your code.



LANGUAGE TYPE	WHAT HAPPENS AFTER TOKENIZATION
Compiled Languages	The tokens are transformed into machine code (or sometimes an intermediate assembly-like code). This machine code is a low-level, binary representation of your program that your computer's CPU can execute directly. Think of it like translating a book into a specific machine language that only one type of robot understands. This translation happens before you run the program.
Interpreted Languages	Instead of creating a separate machine code file, the tokens are executed one-by-one as instructions by an interpreter program. It's like having a live translator who reads each sentence of your book and immediately tells the robot what to do, without writing down the whole translated book first. This translation happens as you run the program.

At the end, all code must become **machine code**.
Machine code is made of **1s and 0s**.
This is the only language the **CPU** understands.

The machine code tells the CPU **exactly what to do**,
one small step at a time.

Where ASCII and Unicode Fit

Your code starts as **text**.
That text is saved using **ASCII or Unicode**.

Before the computer can understand your code:

- It reads the **binary value** of each character

Examples:

- **i** → 01101001 (ASCII 105)
- **f** → 01100110 (ASCII 102)

Every word, symbol, and name in your code starts as **bytes**.
The compiler uses these bytes to find tokens.

Short Version: From Code to CPU

1. You write code in a text editor
2. It is saved using ASCII or Unicode
3. The compiler reads the text as bytes
4. The code is split into tokens
5. Tokens become machine instructions
6. The CPU runs those instructions

That's the full trip — from letters → to logic → to machine.

Machine Code Structure — Opcodes and Operands

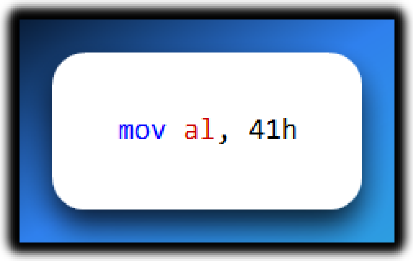
🔧 **What is machine code?**
Machine code is the **binary instructions** the CPU runs directly.
All assembly and C code turns into machine code in the end.

Each machine instruction has two main parts:

- **Opcode** → what to do (like move, add, jump)
- **Operands** → what to use (data, registers, or memory)

The opcode tells the CPU **the action**.
The operands tell the CPU **where and on what** to do it.

🧱 Example Breakdown:



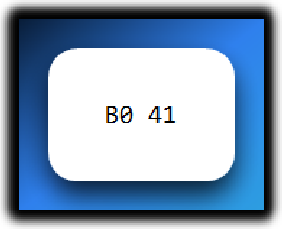
```
mov al, 41h
```

This means: **“Move the value 41h (which is ASCII 'A') into the AL register.”**

Let’s break that into real machine code (x86):

- Opcode for MOV AL, imm8 = B0
- Operand = 41

Final Machine Code:



```
B0 41
```

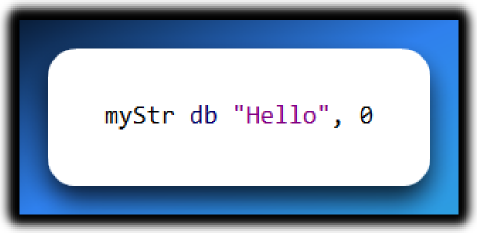
The CPU reads this as:

“Instruction B0 means: move the next byte into AL.
The next byte is 41h — which is ‘A’. Okay, done.”

🔤 2. Assembly and String Handling

I mean, In Assembly, a **string** is just a row of bytes in memory.
There's nothing automatic — you work with each **byte** yourself.
You handle them manually, byte by byte.

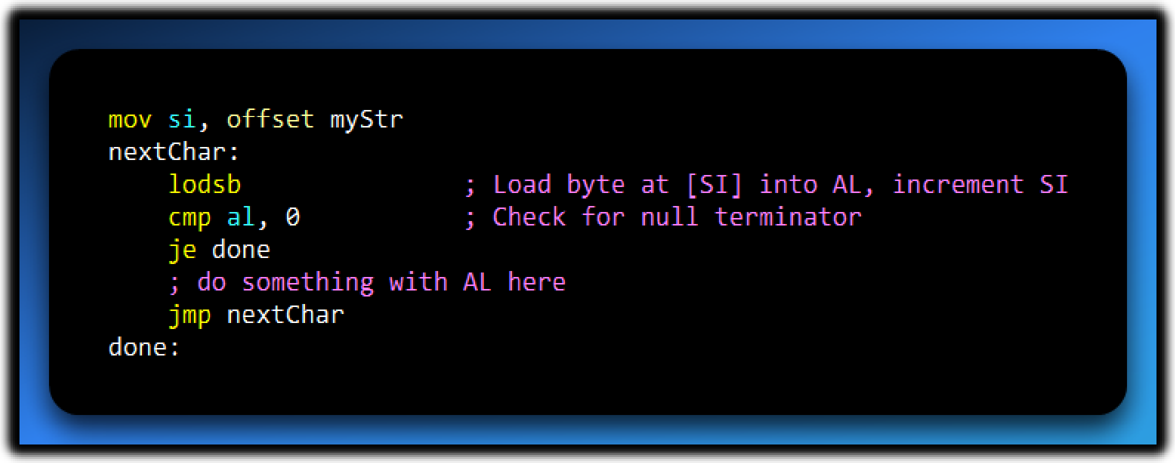
💾 How to make a string:



```
myStr db "Hello", 0
```

- db means "define byte"
- The 0 at the end is the null terminator (like in C)

🧪 Reading character by character:



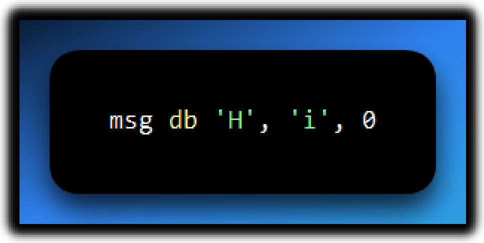
```
mov si, offset myStr
nextChar:
    lodsb               ; Load byte at [SI] into AL, increment SI
    cmp al, 0           ; Check for null terminator
    je done
    ; do something with AL here
    jmp nextChar
done:
```

This is like your bare-metal for (char c : str) loop — just much closer to the metal.

💌 3. Memory and ASCII/Unicode Storage

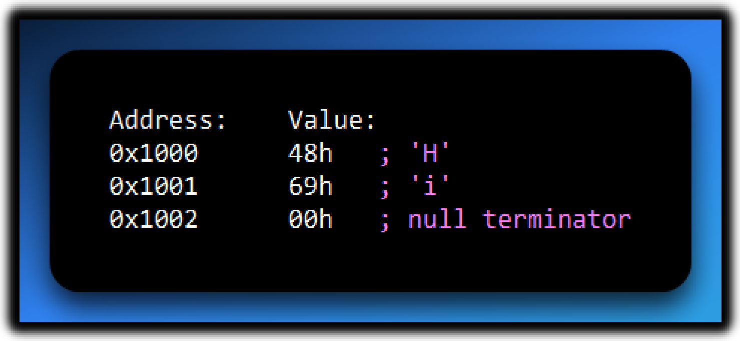
Let’s say you store the word "Hi" in memory.

ASCII:



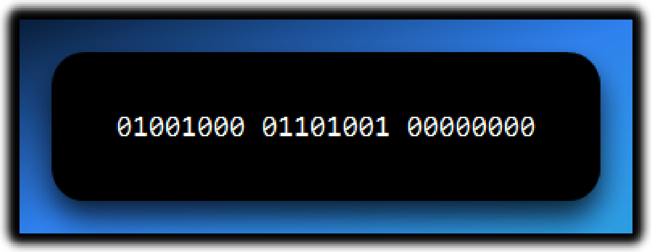
```
msg db 'H', 'i', 0
```

Memory layout (byte by byte):



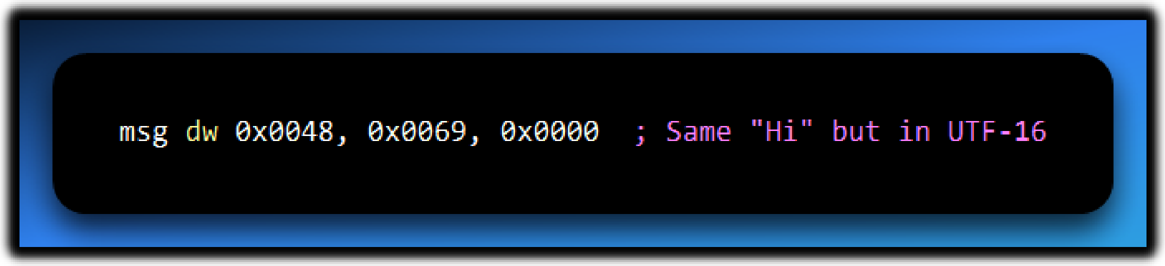
```
Address:    Value:
0x1000      48h   ; 'H'
0x1001      69h   ; 'i'
0x1002      00h   ; null terminator
```

The CPU doesn’t *see* characters — it just sees:



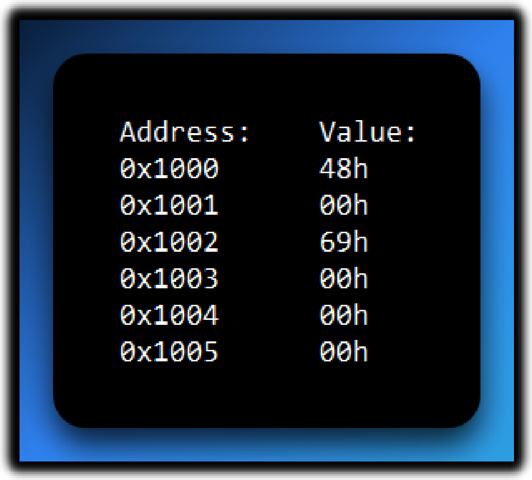
```
01001000 01101001 00000000
```

Unicode (UTF-16):



```
msg dw 0x0048, 0x0069, 0x0000  ; Same "Hi" but in UTF-16
```

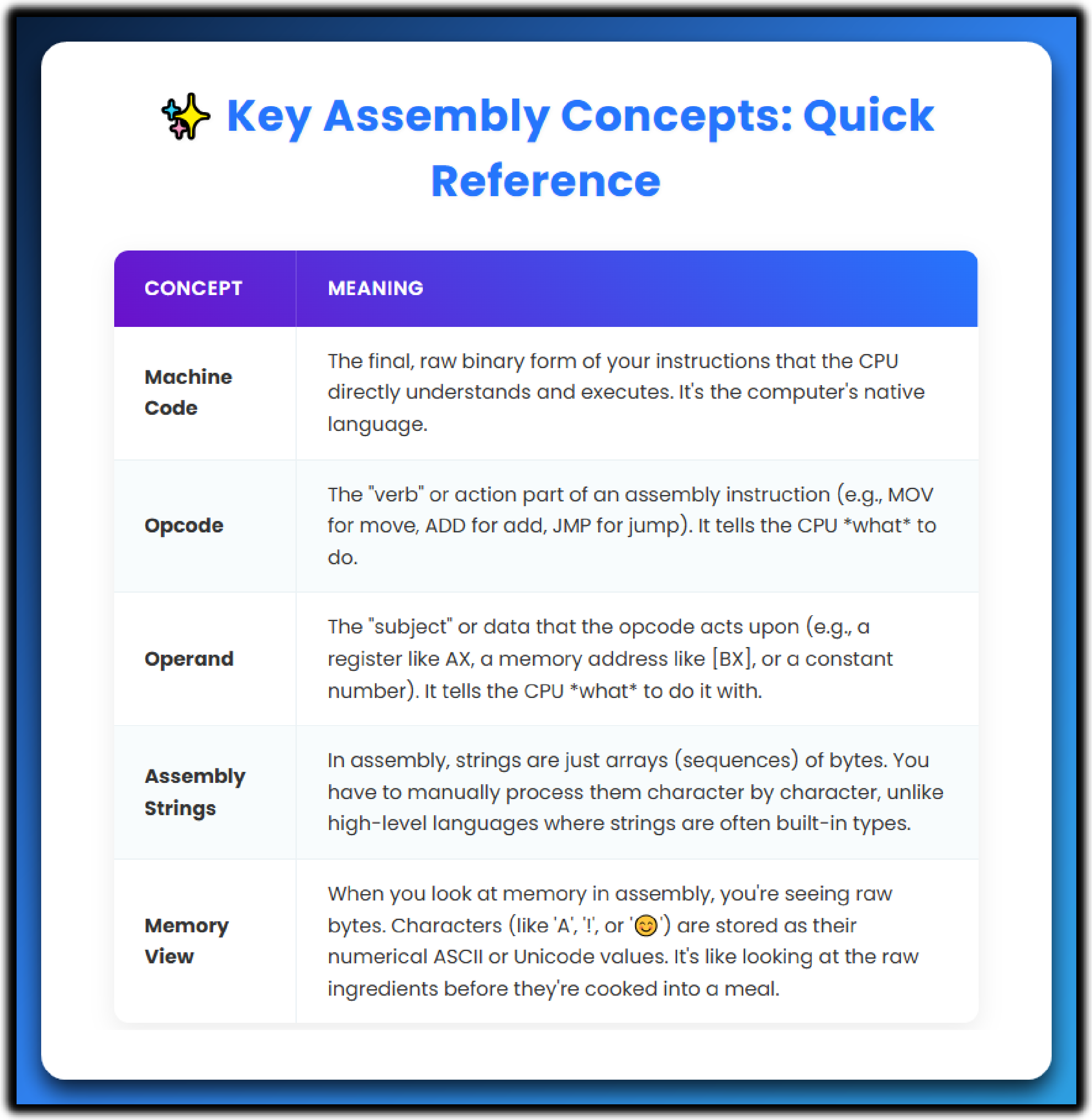
Now every character is 2 bytes:



```
Address:    Value:
0x1000      48h
0x1001      00h
0x1002      69h
0x1003      00h
0x1004      00h
0x1005      00h
```

That’s why **UTF-8 is more compact** — it uses just as many bytes as needed.

Key Takeaways:



✨ Key Assembly Concepts: Quick Reference

CONCEPT	MEANING
Machine Code	The final, raw binary form of your instructions that the CPU directly understands and executes. It's the computer's native language.
Opcode	The "verb" or action part of an assembly instruction (e.g., MOV for move, ADD for add, JMP for jump). It tells the CPU *what* to do.
Operand	The "subject" or data that the opcode acts upon (e.g., a register like AX, a memory address like [BX], or a constant number). It tells the CPU *what* to do it with.
Assembly Strings	In assembly, strings are just arrays (sequences) of bytes. You have to manually process them character by character, unlike high-level languages where strings are often built-in types.
Memory View	When you look at memory in assembly, you're seeing raw bytes. Characters (like 'A', '!', or '😊') are stored as their numerical ASCII or Unicode values. It's like looking at the raw ingredients before they're cooked into a meal.