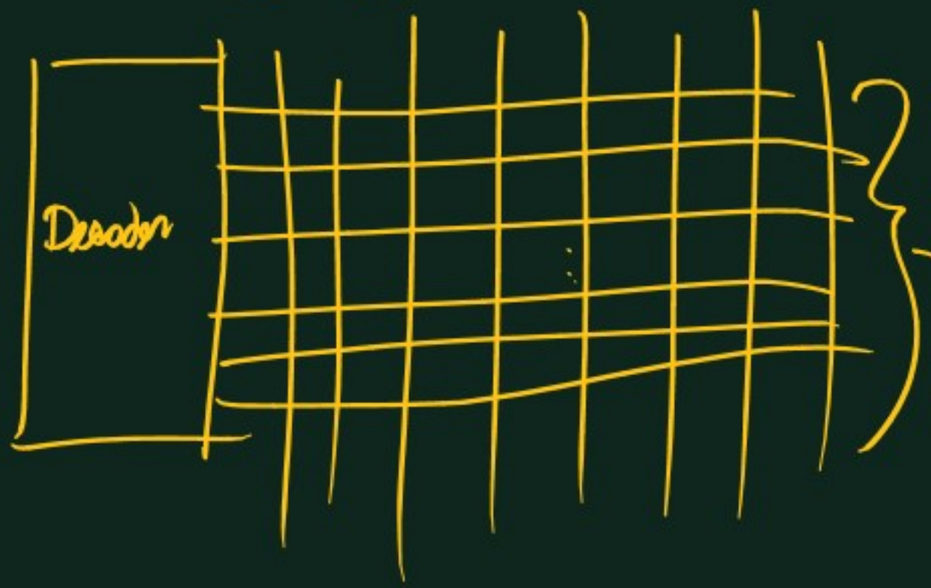
Primary Memory

Read Only Memory → UEFI BIOS chip

Basic I/O System → firmware

RAM

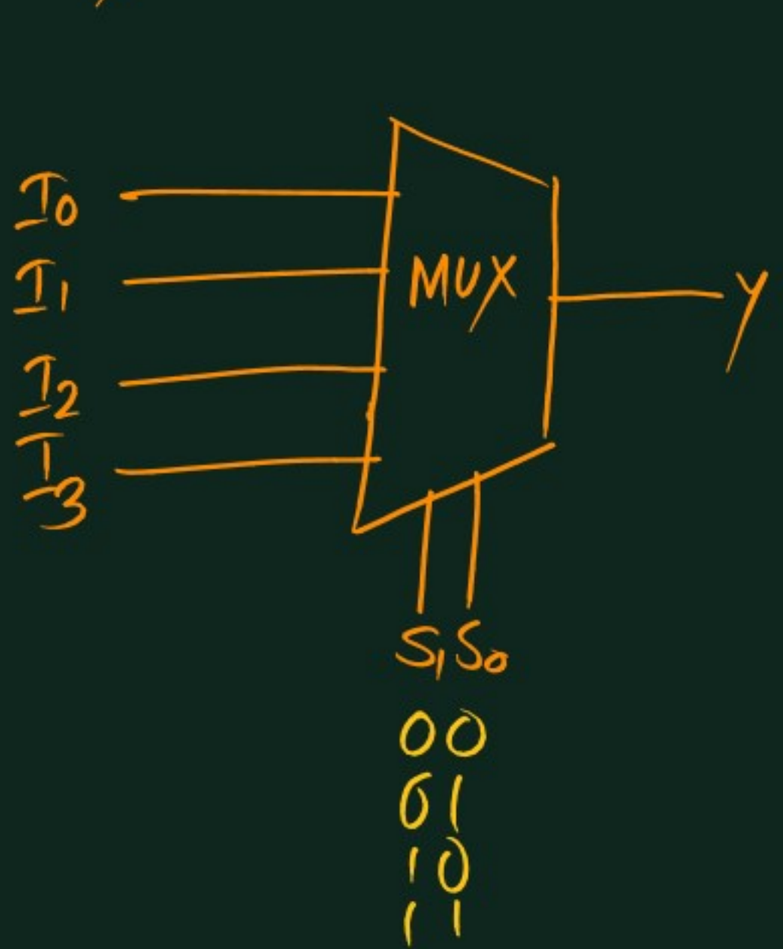
- Volatile

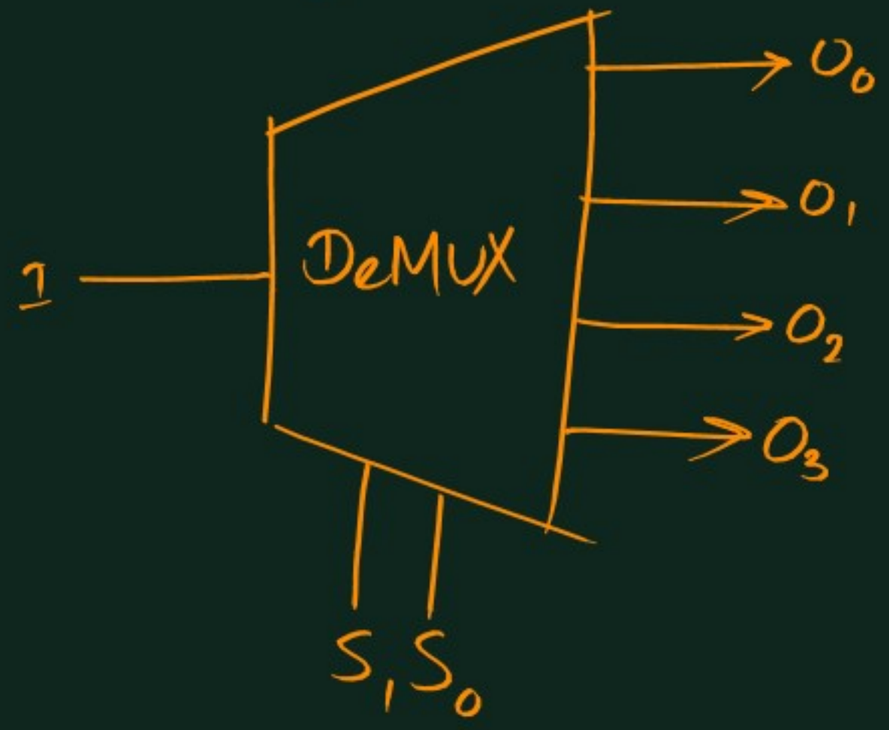


Combinational Circuits

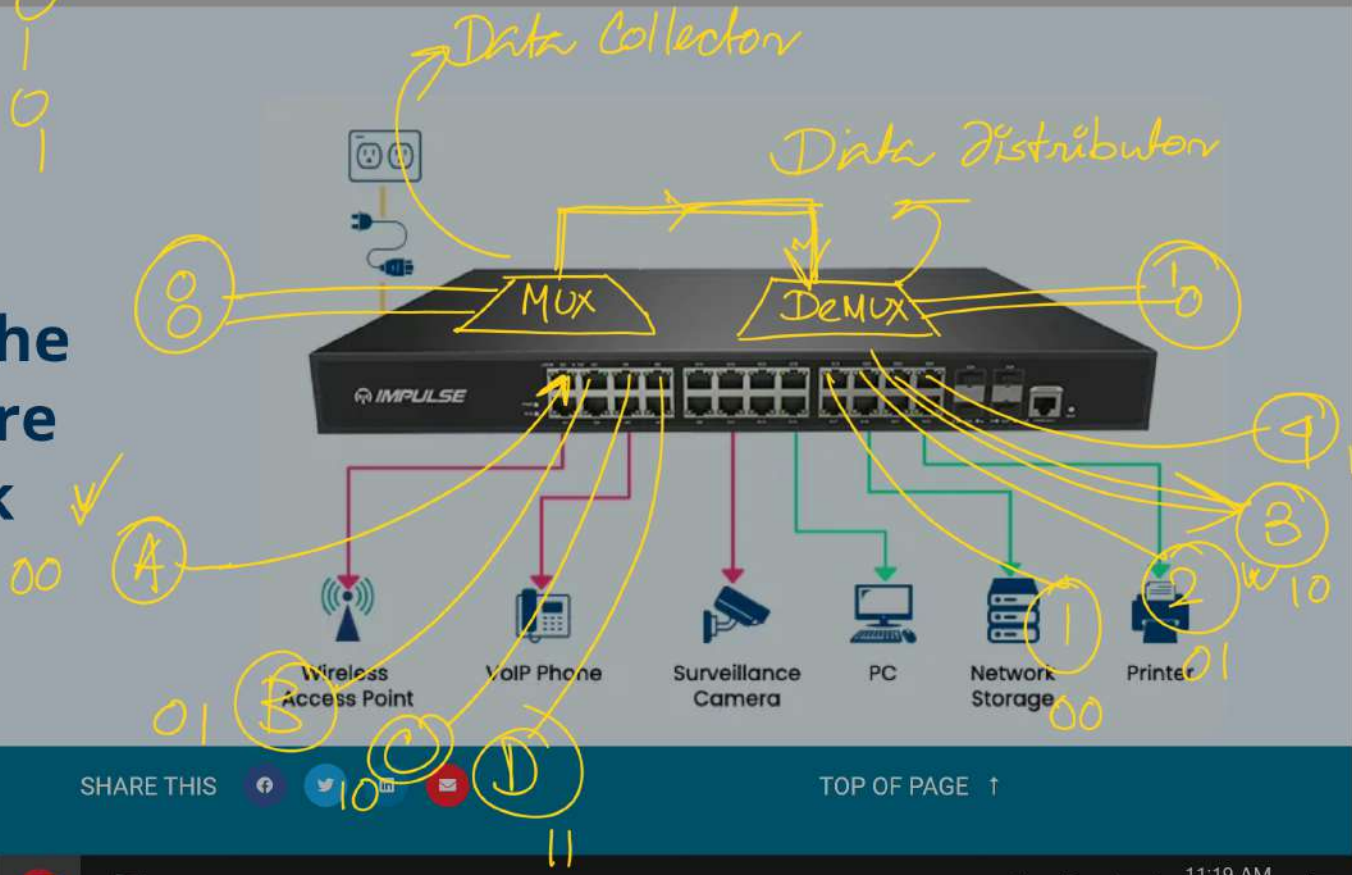
i) Made from combining logic gates
ii) The o/p always depend only on the present i/p.

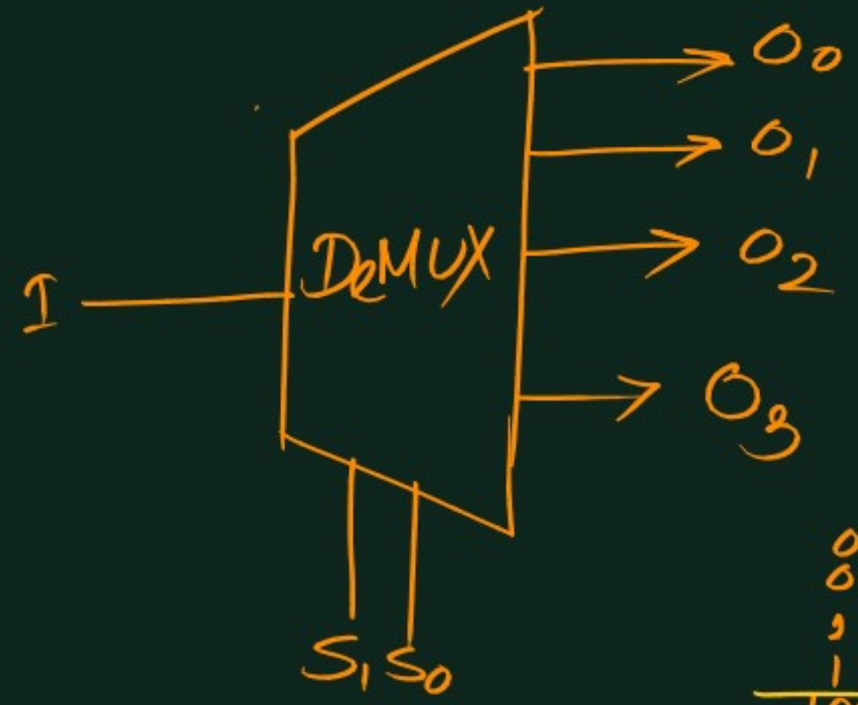
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3





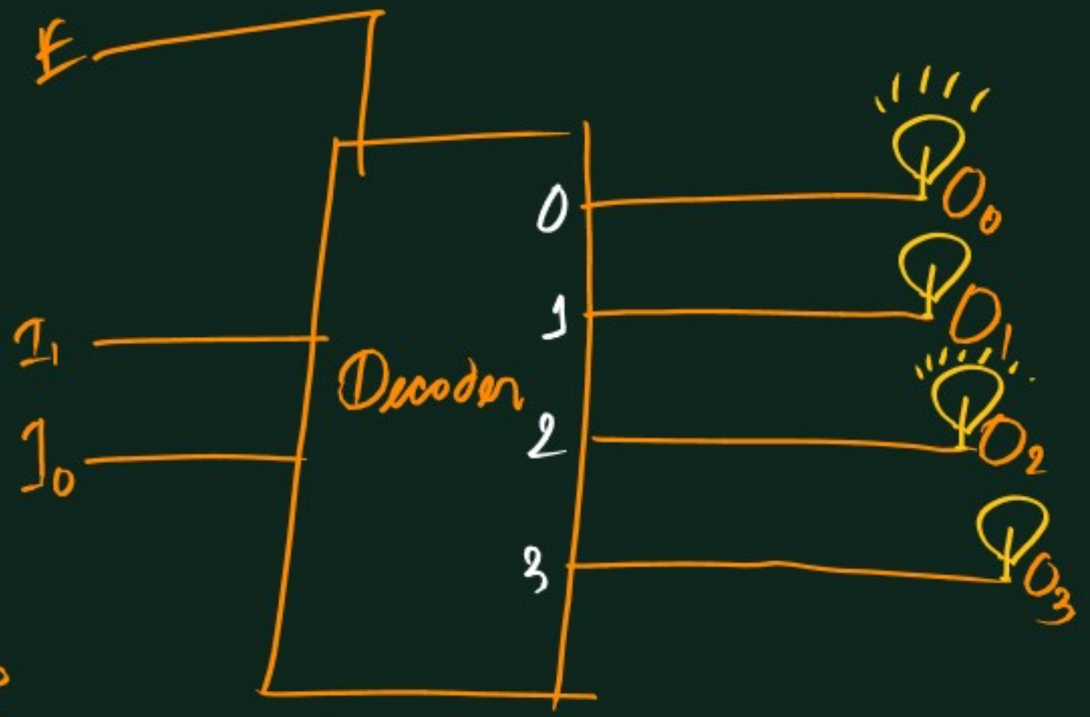
Power Over Ethernet: The Switch That Powers More Than Just Your Network





$00 \rightarrow 0$
$01 \rightarrow 1$
$10 \rightarrow 2$
$11 \rightarrow 3$

$100 \rightarrow 4$
$101 \rightarrow 5$

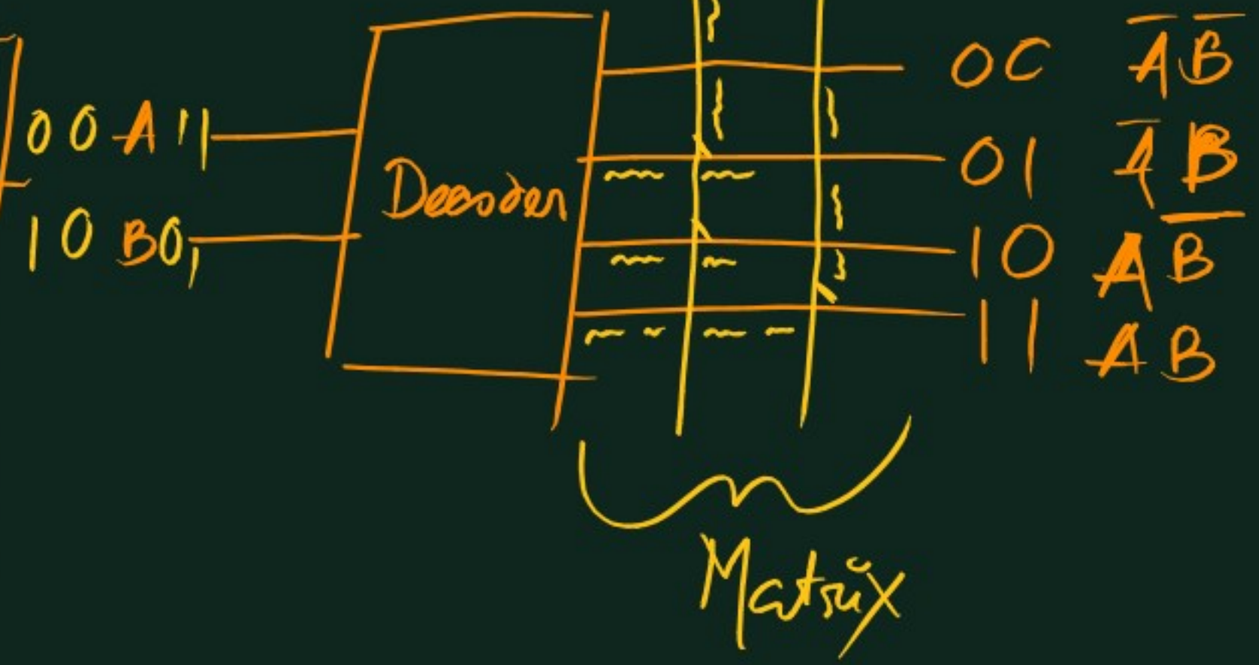


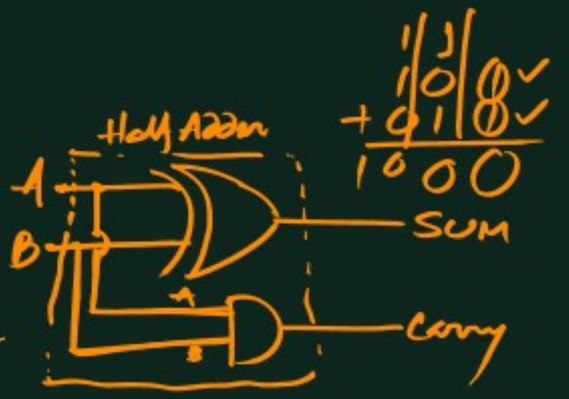
00 10 11 01 10 11 11 00 11 10 01 00

0 2 3 1 2 3 3 0 3 2 1 0

A	0	0	1	1
B	+0	+1	+0	+1
	00	01	01	10
	C S	C S	C S	C S

A	B	SUM	CARRY
0	0	0	0
0	1	1 ✓	0
1	0	1 ✓	0
1	1	0	1 ✓





POST

Power On
Self Test

Class 4: How Computer Programs Execute

Topics:

1. **Bootloader Program**
2. **User Program**
3. **Compiler**
4. **Machine Code**
5. **CPU Execution**

1️⃣ Bootloader Program *(with ROM & Primary-Memory Context)*

Primary Memory and Why ROM Matters

- **Primary Memory** consists of **RAM** (volatile) and **ROM** (non-volatile).
- Because ROM retains data even when power is off, it is ideal for storing **firmware**—the tiny, permanent software that starts the computer.
- This firmware includes the **BIOS** (Basic Input/Output System) or UEFI on modern PCs.

What Happens at Power-On

1. **Power-On Self-Test (POST):**
    - Immediately after you press the power button, the CPU fetches the very first instructions from the **ROM chip**.
    - The BIOS runs diagnostic checks on essential components: CPU, RAM modules, keyboard, and other peripherals.
    - If something is wrong (e.g., a RAM stick is loose) the BIOS issues **beep codes**.
2. **Bootloader Code Execution:**
    - Once POST is successful, the BIOS/UEFI firmware (still executing from ROM) looks for a **secondary bootloader** on storage devices (HDD/SSD, USB, etc.).
    - It then copies this bootloader into **RAM** and transfers control to it.
    - Popular examples include **GRUB** on Linux systems or the Windows Boot Manager.
3. **Operating System Loading:**
    - The bootloader's job is to locate the OS kernel on the disk, load it into RAM, and hand control to it.
    - Only after this step does the OS display the login screen.

Digital-Logic Architecture of ROM (Why the Firmware Is Reliable)

- ROM internally uses an **n-to-2^n address decoder** to activate one word line, and a **fixed ROM matrix** of programmable connections (implemented with fuses/antifuses) to produce the stored output bits.
- These min-terms are selectively "fused" (or left open) to represent the desired logic.
- That fixed pattern ensures the CPU always reads the same boot instructions—crucial for a consistent start-up.

Types of ROM You'll Encounter in Bootloaders

- **PROM:** Programmed once; fuses permanently set.
- **EPROM:** Can be erased with UV light and re-programmed.
- **EEPROM/Flash ROM:** Electrically erasable and re-writable—used in modern motherboards so BIOS/UEFI updates are possible.

Key Idea: The Bootloader stage is possible only because **firmware stored in ROM** (non-volatile, fuse-programmed digital logic) survives power cycles and is instantly available to the CPU at power-on.

Reference Links

- Intel: System Boot Process
- Patterson & Hennessy, *Computer Organization and Design*, Boot process chapters.

2️⃣ User Program

What it is:

- The **source code** written by a programmer in a **high-level language** (C, Java, Python, etc.).
- Human-readable and designed for problem solving, **not directly understood by hardware**.

Example:

```
#include <stdio.h>

int main() {
   printf("Hello, world!\n");
   return 0;
}
```

Reference:

- Kernighan & Ritchie, *The C Programming Language*, Prentice Hall.

3️⃣ Compiler

What it does:

- A **translator** that converts the user program's high-level instructions into **machine code** (or intermediate code, depending on the language).
- Performs lexical analysis, parsing, optimization, and code generation.

Flow:

Source Code → Compiler → Object/Machine Code

- Some languages (like Java) use an extra step: compiling to **bytecode**, which the JVM later interprets or JIT-compiles.

Reference:

- Alfred V. Aho et al., *Compilers: Principles, Techniques, and Tools* (a.k.a. "Dragon Book").

4️⃣ Machine Code

What it is:

- The **binary instruction set** (0s and 1s) that the CPU can execute directly.
- Specific to the processor architecture (x86, ARM, RISC-V, etc.).

Characteristics:

- Consists of **opcodes** (operation codes) and operands.
- Usually represented in hexadecimal for readability (e.g., B8 01 00 for x86).

Reference:

- Patterson & Hennessy, *Computer Organization and Design*.

5️⃣ CPU Execution

How it works:

- The CPU **fetches** machine instructions from RAM, **decodes** them (understands which operation is requested), and **executes** them.
- This cycle repeats billions of times per second (the **fetch–decode–execute** cycle).

Steps:

1. **Fetch:** Program Counter (PC) points to the next instruction.
2. **Decode:** Instruction Decoder determines the required action.
3. **Execute:** ALU or other units perform arithmetic, logic, memory access, or I/O.

Reference:

- David A. Patterson & John L. Hennessy, *Computer Organization and Design*, Chapters 4–5.

🔗 **Putting It All Together (Correct Order)**

1. **Bootloader Program**
    - Initializes hardware and loads the operating system.
2. **User Program**
    - Programmer writes source code inside the OS environment.
3. **Compiler**
    - Translates source code into machine code (binary executable).
4. **Machine Code**
    - Stored in an executable file, loaded into RAM when the program is launched.
5. **CPU Execution**
    - Processor fetches and runs these binary instructions.

Quick Analogy

Think of starting a car:

Computer Step	**Car Analogy**
Bootloader Program	Turning the ignition so the engine starts
User Program	Driver planning the route (writing code)
Compiler	Translating the plan into GPS instructions
Machine Code	The precise GPS directions
CPU Execution	The engine actually moving the car

Summary:
When a computer powers on, the **bootloader** sets the stage. A programmer writes a **user program**, which a **compiler** converts into **machine code**. The **CPU** then **fetches, decodes, and executes** those binary instructions—making the software actually run.

UEFI stands for **Unified Extensible Firmware Interface**.
It is the **modern replacement for the legacy BIOS firmware** that runs when you power on a PC.

Here's what it is and what it does:

---

1. Purpose

UEFI is the low-level software that:

- Initializes and tests the hardware (CPU, memory, devices).
- Locates and loads the operating system's bootloader into RAM.
- Provides a small runtime environment for firmware utilities (e.g., configuration menus, diagnostics).

---

2. Key Characteristics

Feature	UEFI	Legacy BIOS
Interface	Graphical menus, mouse support, larger storage for drivers	Text-only
Storage	Stored in **flash ROM** on the motherboard	Stored in ROM/flash
Boot Method	Uses **GPT (GUID Partition Table)** and loads boot files from an **EFI System Partition**	Uses MBR (Master Boot Record)
Extensibility	Supports drivers, networking, secure boot	Very limited

---

3. How It Fits in the Boot Process

1. **Power On** → CPU executes firmware instructions from flash ROM.
2. **UEFI Firmware Runs** → Performs POST (hardware checks).
3. **Boot Manager** → Reads the EFI System Partition on the disk.
4. **OS Loader** → Loads the operating system kernel (Windows, Linux, etc.) into RAM.

---

4. Why It Matters

- **Secure Boot**: Verifies digital signatures to block unauthorized bootloaders or malware.
- **Faster Startup**: Parallel hardware initialization.
- **Large-Disk Support**: Works with disks larger than 2 TB thanks to GPT**(GUID Partition Table)**.

In short:
UEFI is the **modern firmware interface** between a computer's hardware and its operating system, stored in non-volatile flash memory on the motherboard, and it replaces the older BIOS while adding security, flexibility, and speed.

GUID = Globally Unique IDentifier
Every partition on the disk is assigned a 128-bit unique ID (GUID), ensuring that partitions are uniquely identified across all systems.