Computer Organization, Microprocessors and Assembly

Dr. Cahit Karakuş

Esenyurt Üniversitesi

İçindekiler Tablosu

Course Title: Computer Organization, Microprocessors and Assembly
Department: Software Engineering – Electric and Electronic Engineering
Credits: 3–4 (depending on your university regulations)
Semester: [Specify Semester]
Prerequisites: Digital Logic Design, Programming Fundamentals, Basic Electronics

Course Objectives
By the end of this course, students will be able to:

1. Understand the fundamental concepts of computer organization and architecture.
2. Differentiate between hardware and software interfaces.
3. Understand microprocessor structure, function, and programming.
4. Write simple assembly language programs for microprocessors.
5. Analyze and design basic CPU operations and memory systems.

Course Outcomes
Students will be able to:

- Explain the Von Neumann model and basic computer components.
- Distinguish between computer organization and architecture.
- Interpret CPU instruction cycles and addressing modes.
- Write, debug, and execute assembly programs on a simulator or real microprocessor.
- Understand input/output interfacing and memory hierarchy.

Week-wise Lecture Plan

Week	Topics	Activities / Lab Exercises
1	**Introduction to Computer Organization** - What is computer organization? - Levels of computer system (hardware, firmware, software)	Overview lecture; discussion of real-world computer components
2	**Computer Architecture vs. Organization** - Differences and relation - Impact on software and performance	Case study: Examples of CPU designs
3	**Von Neumann Architecture** - Components: CPU, memory, I/O - Stored-program concept	Lab: Diagramming Von Neumann vs Harvard architecture
4	**Hardware-Software Interface** - Instruction sets, machine language - Compiler and assembler basics	Lab: Writing simple pseudo-assembly instructions
5	**CPU Basics** - Registers, ALU, control unit - Instruction cycle	Lab: Tracing instruction cycles

Week	Topics	Activities / Lab Exercises
6	**Memory Organization** - RAM, ROM, cache, virtual memory - Memory hierarchy and speed	Lab: Memory mapping exercises
7	**Input/Output Organization** - I/O techniques: polling, interrupts, DMA	Lab: Simulate I/O in microprocessor simulator
8	**Midterm Exam / Review**	Midterm test + Q&A
9	**Microprocessors Introduction** - 8085/8086 architecture overview - Pin diagram, registers, flags	Lab: Exploring 8085/8086 simulator
10	**Instruction Set of Microprocessors** - Data transfer, arithmetic, logical, control instructions - Addressing modes	Lab: Writing simple arithmetic programs
11	**Assembly Language Programming Basics** - Assembler directives, labels, comments - Data storage and constants	Lab: Hello World and basic math operations in assembly
12	**Program Control Instructions** - Jump, loop, subroutine, call/return - Stack operations	Lab: Write loop and subroutine programs
13	**Interfacing and I/O in Assembly** - Ports, interrupts, timers	Lab: Simulate keyboard input and LED output in assembly
14	**Advanced Topics** - Microprocessor interfacing with memory and peripherals - Simple ALU design	Lab: Mini-project: Create an assembly program to solve a problem
15	**Final Exam / Course Review**	Final exam + course wrap-up

Assessment Plan

- **Midterm Exam:** 30%
- **Lab Assignments / Projects:** 20%
- **Quizzes / Participation:** 15%
- **Final Exam:** 35%

Textbooks & References

1. **Primary Textbook:**
    - "Computer Organization and Design" by David A. Patterson & John L. Hennessy
2. **Assembly & Microprocessors:**
    - "The 8086 Microprocessor: Programming & Interfacing" by K. Udaya Kumar, B.S. Umashankar
3. **Supplementary:**
    - Tanenbaum, "Structured Computer Organization"
    - Online resources: emu8086 simulator, Intel Manuals

Lab / Practical Requirements

- 8086 or 8088 simulator (emu8086, MASM)
- Basic electronics lab for interfacing exercises
- IDE or assembler environment for assembly programming

1. Data

Data refers to **raw facts, figures, or information** that can be processed by a computer. It can be anything that represents **values** or **measurements** that a program can manipulate.

- Examples of data include numbers, text, images, audio, and sensor readings.
- Data alone doesn't have meaning until it is **processed** or **interpreted**.

Key points:

- Data can be stored in memory, files, or databases.
- Computers process data to produce **information**, which is meaningful output.

Example:

- 42 → a number (data)
- "Hello" → a string (data)
- true → a boolean value (data)

2. What are Data Types?

Data types define the **kind of data** and the **operations** that can be performed on that data. They are a way for the computer to **classify data** so it knows how to handle it.

- Every variable in programming has a data type that determines the type of values it can hold.

Common Data Types:

1. **Integer (int)**
    - Whole numbers without a fractional part.
    - Example: 10, -5, 0
2. **Floating Point (float, double)**
    - Numbers with a decimal point.
    - Example: 3.14, -0.001
3. **Character (char)**
    - Single characters, often stored in quotes.
    - Example: 'A', 'z', '7'
4. **String (string)**
    - Sequence of characters.
    - Example: "Hello World", "123abc"
5. **Boolean (bool)**
    - True or False values.
    - Example: true, false
6. **Other complex types** (used in advanced programming)
    - **Arrays**: Collection of elements of the same type
      Example: [1, 2, 3, 4]
    - **Structures / Objects**: Collection of variables under one name
    - **Pointers / References**: Memory addresses of other variables

Summary:

- **Data** = Raw facts that computers process.
- **Data type** = The classification of data that tells the computer what **kind of value** it is and what operations can be done with it.

Binary Numbering System

The **binary numbering system** is a **base-2 numeral system**. Unlike the decimal system (base-10), which uses 10 digits (0–9), binary uses only **two digits**: 0 and 1

These two digits are called **bits** (binary digits), the smallest unit of data in computers.

- Each bit can represent one of **two states**, usually:
  - 0 → off / false / low voltage
  - 1 → on / true / high voltage

Binary Signal – Bit:

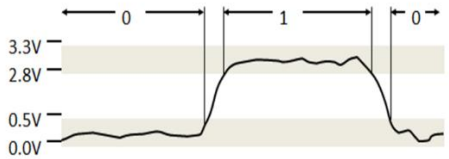
Binary signal: Two-state signal: Data in two states (0/1). off & on. Bit is carried and stored by electrical signals inside a computer. For example: low voltage & high voltage; 0v & 5v

Bit: 0/1. Data represents only as bit inside computer.

We use 10 Powers as all units (gr, m, sec, hz, bit or bit/second)

Bit/second: transfer data or the processor's data processing speed.

Bit is not just a mathematical concept, but also has a counterpart in the physical world. In the binary number system, it has the value 0 or 1 and nothing else. A bit is the smallest unit of data in a computer.



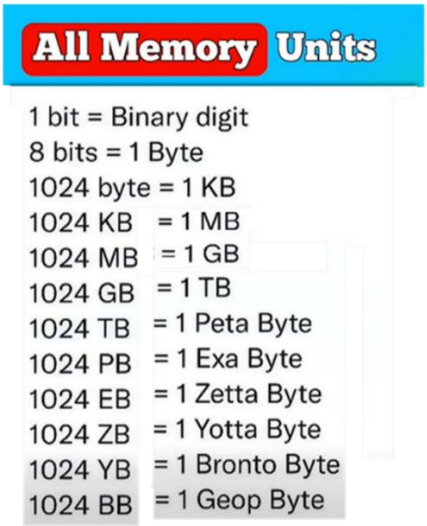
Byte represents as 8 bit. Memory cell sare 8 bits. So totaly memory cells are represented 2^n byte. Here, n: 10, 20,30,40,50,60, ….

Why Computers Use Binary:

1. Computers use **electronic circuits (Transistor)** and electrical signals which have **two states**: on and off (binary)
2. Binary makes it **easy to represent** these two states reliably. (easy to store data, easy to transfer data, easy to manipulate or process data)
3. All data in a computer—numbers, text, images, sound, signals—is ultimately stored, transfered, processed as **binary values**.

Prefix	Symbol	Power of 10	Power of 2	Prefix	Symbol	Power of 10
Kilo	K	1 thousand = 10^3	$2^{10} = 1024$	Milli	m	1 thousandth = 10^{-3}
Mega	M	1 million = 10^6	2^{20}	Micro	μ	1 millionth = 10^{-6}
Giga	G	1 billion = 10^9	2^{30}	Nano	n	1 billionth = 10^{-9}
Tera	T	1 trillion = 10^{12}	2^{40}	Pico	p	1 trillionth = 10^{-12}
Peta	P	1 quadrillion = 10^{15}	2^{50}	Femto	f	1 quadrillionth = 10^{-15}
Exa	E	1 quintillion = 10^{18}	2^{60}	Atto	a	1 quintillionth = 10^{-18}
Zetta	Z	1 sextillion = 10^{21}	2^{70}	Zepto	z	1 sextillionth = 10^{-21}
Yotta	Y	1 septillion = 10^{24}	2^{80}	Yocto	y	1 septillionth = 10^{-24}

- When Referring to Bytes (as in computer memory)
  - Kilobyte (KB) 2^{10} = 1,024 bytes
  - Megabyte (MB) 2^{20} = 1,048,576 bytes
  - Gigabyte (GB) 2^{30} = 1,073,741,824 bytes
  - Terabyte (TB) 2^{40} = 1,099,511,627,776 bytes



- Memory size is always represented by a byte as 2 power.
- In the expression 2^n byte totally capacity of a memory, n = the number of memory address lines. Memory address bus indexing: A0, A1, A2, … , An-1
- 1 byte of a memory cell is 8 bits

Example: 16Gbyte=? Byte.

- 16Gbyte=2^4 * 2^30 byte=2^34byte
- 16Gbytes how much bits? = 2^3 * 2^34 bit =2^37bit

The number of address lines of memory is 34=n

Memory address line indexing: A33, A32, … , A1, A0

$$b^x \cdot b^y = b^{x+y}$$

$$\frac{b^x}{b^y} = b^{x-y}$$

$$\left(b^x\right)^y = b^{xy}$$

$$(ab)^x = a^x b^x$$

$$\left(\frac{a}{b}\right)^x = \frac{a^x}{b^x}$$

Example:
147 Mbyte=(?)byte
2^7<147<2^8, we take 2^8. Because memory is possible.
2^8 * 2^20 =2^28 byte
The number of address lines of memory is 28=n
Memory address line indexing: A27, A26, … , A1, A0

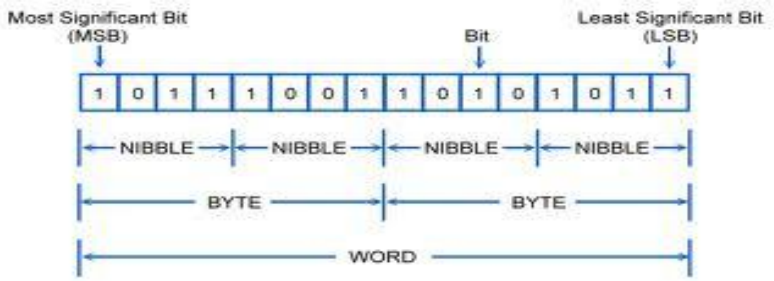
Example: 2^44 bit how much Terabyte?

- 8 bit equals to 1 byte. 2^44/2^3=2^41byte = 2^1 * 2^40 = 2 Terabyte

What is the capacity of a memory (byte)?

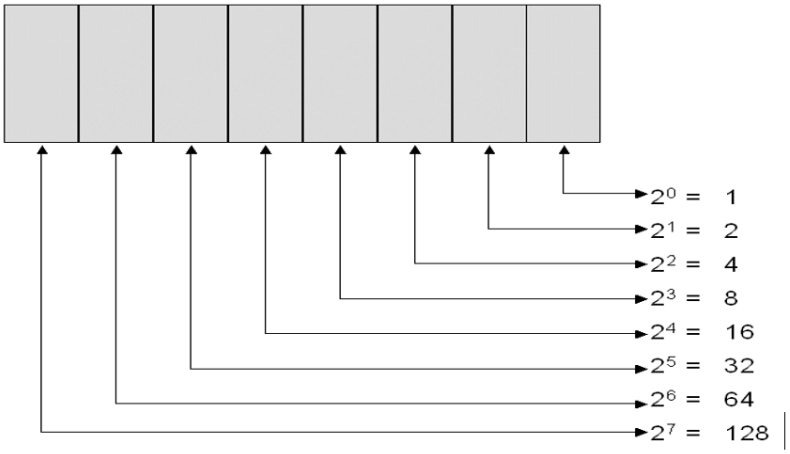
- Answer: 2 Terabyte = 2^41 byte
- In the expression 2^n, n = the number of memory address lines. Here, n=41
- Memory address bus indexing: A0, A1, A2, … , A40

- When Referring to Bits Per Second (as in transmission rates): Bit/sec is used for data processing or transferring. Power of ten.
  - Kilobit per second (Kbps) = 1000 bps (thousand)
  - Megabit per second (Mbps) = 1,000,000 bps (million)
  - Gigabit per second (Gbps) = 1,000,000,000 bps (billion)
  - Terabit per second (Tbps) = 1,000,000,000,000 bps (trillion)



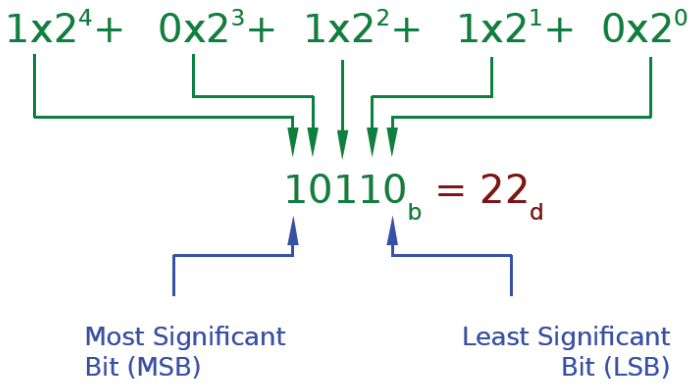
How Binary Numbers Work:

Each position in a binary number represents a **power of 2**, starting from the rightmost bit (least significant bit, LSB).



Binary

- Binary is exactly the same, only instead of ten digits/states (0 to 9) we have just two, so the base becomes 2:



Binary number example: (1011)b

Binary digit	**1**	**0**	**1**	**1**
Index (Power of 2)	**3**	**2**	**1**	**0**

Decimal value: $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

So, binary 1011 equals decimal 8+2+1=11.

Examples of Binary Numbers

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Adding With A Carry:

c_{i-1}	0	0	0	0	1	1	1	1
$+a_i$	0	0	1	1	0	0	1	1
$+b_i$	+0	+1	+0	+1	+0	+1	+0	+1
T	0	1	1	10	1	10	10	11

Negative Numbers as binary

- Computers store numbers in **binary**, but real-world calculations often need **negative numbers**.
- Special methods are used to **represent negative values in binary**, because binary itself only has 0 and 1.

Example: (-5)d=(?)b, (8-bit numbers)

Long of data (binary) is important: 8, 16, 32, 64, 128, 256 bits; always take 4 bit from right side to left side.

a) +5= 0000 0101

b) 1's Complement

- Negative number = **invert all bits** of the positive number. 0 → 1 and 1 → 0.
- +5 : 0000 0101
- 1 -> 0, 0 -> 1: 1111 1010 (invert all bits of 0000 0101)
- +1: 1111 1010 + 1 = 1111 1011=-5d

D. Range of 2's Complement Numbers

- For **n-bit numbers**:
  - Range = **-2^{n-1} to $2^{n-1} - 1$**

Example (8-bit):

- Range = -128 to +127
- +127 → 01111111
- -128 → 10000000

Example: db -4 (db: define byte: 8 bit)

(4)d= (0000 0100)b

(-4)d= Binary (4) -> invert all bits of the positive number +1 :

(-4)d =1111 1011 +1 =1111 1100=FCh

Example: dW -4 (dw: define Word: 16 bit)

(4)d= 0000 0000 0000 0100

(-4)d=Word(4) -> invert all bits of the positive number +1

(-4)d=1111 1111 1111 1011 +1= 1111 1111 1111 1100=FFFC

(-4)d=(0FFFC)h

Example: (-164)d=(?)w

(164)d=(010100100)b

(0000 0000 1010 0100)w

1->0, 0->1; (1111 1111 0101 1011)w

+1; (1111 1111 0101 1100)w

Numbers:

- If no definition is given, it means the decimal number system is being used.
- 11011 decimal
- (11011)b binary
- 64223 decimal
- (-21843)D decimal
- 1,234 illegal, contains a nondigit character
- (1B4D)H hexadecimal number
- (1B4)D illegal hex number, does not end with “H”
- (FFFF)H illegal hex numbe, does not begin with with digit
- (0FFFF)H hexadecimal number

2’s complement is the standard in computers because it simplifies **addition, subtraction, and other arithmetic operations**.

How Hex Numbers Work

Each position in a hexadecimal number represents a **power of 16**, starting from the rightmost digit (least significant digit, LSD).

The **hexadecimal system** is a **base-16 numeral system**.
It uses **16 symbols** to represent values (4 bit):
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F: Here, A–F represent decimal values 10–15:

Example: 2F3 (hex)

(0010 1111 0011)b= 2^9+2^7+2^6+2^5+2^4+2^1+2^0

3. Why Computers Use Hexadecimal

Binary numbers can get **very long**. For example, the binary number 11111111 is **8 bits**.
Hexadecimal **shortens binary** representation by grouping 4 bits per hex digit:

Binary (4 bits)	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A -10
1011	B -11
1100	C-12
1101	D-13
1110	E-14
1111	F -15

Example:
Binary 1101 1110 → group as 1101 1110 → Hex DE.

- Hex is widely used in **programming**, **memory addresses**, and **color codes in graphics** (like #FF5733).

- Hex = **base-16**, uses 0–9 and A–F.
- Each hex digit represents **4 binary bits**.
- Makes binary numbers **shorter and easier to read**.
- Used in **computers, programming, and digital electronics**.

Data Types in the CPU (Central processing Unit=Microprocessor):

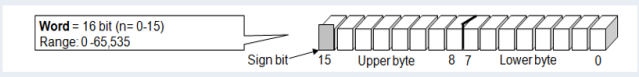
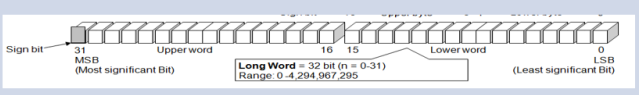
In a CPU, **data types** define the **format of data** that the processor can handle. They tell the CPU **how many bits** to use as 1 second and **what operations are allowed**.
Data types are used in **registers, memory, and instructions**. Choosing the correct data type ensures **efficient processing** and **correct results**.

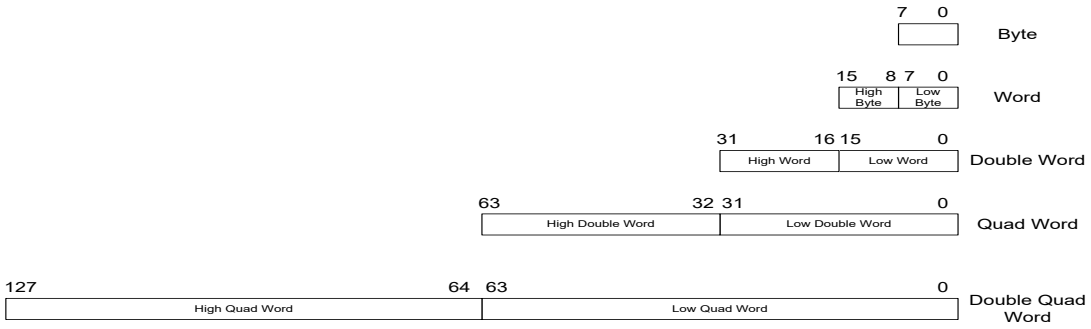
Common Data Types in the CPU

A. Integer Types

- Represent **whole numbers** (no fractions).
- Stored in **binary format**.
- Can be **signed** (positive and negative) or **unsigned** (only positive).

Type	Bits	Range (Signed)	Range (Unsigned)
Byte	8	-128 to 127	0 to 255
Word	16	-32,768 to 32,767	0 to 65,535
Double Word (DWORD)	32	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
Quad Word (QWORD)	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

Nibble	**4 bit**	
Byte	**8 bit**	
Word	**16 bit**	
Long word	**32 bit**	



B. Floating-Point Types

- Represent **real numbers** with fractions (decimals).
- Use **IEEE 754 standard** in most CPUs.
- Stored in three parts: **sign, exponent, mantissa**.

Type	Bits	Approx. Range
Single precision (float)	32	$\pm 3.4 \times 10^{38}$
Double precision (double)	64	$\pm 1.8 \times 10^{308}$

Example: 3.14, -0.001, 2.71828

C. Character Types

- Represent **letters, numbers, or symbols**.
- Usually stored as **ASCII** or **Unicode** codes.

Type	Bits	Example
Char	8	'A', 'z', '7', '#'

In CPU registers, characters are stored as **binary numbers**. For example, ASCII 'A' = 65 = 01000001.

D. Boolean Type

- Represents **true or false** (1 or 0).
- Often stored in **1 bit**, but usually occupies **1 byte** for practical reasons in CPU memory.
- Used in **logical operations, flags, and conditions**.

E. Pointer/Address Types

- Represent **memory addresses** where data is stored.
- Size depends on CPU architecture (e.g., 32-bit or 64-bit).
- Used in accessing **variables, arrays, and functions**.

How CPU Uses Data Types

1. **Registers**: CPU has registers of fixed sizes (8-bit, 16-bit, 32-bit, 64-bit) that match data types.
2. **Instructions**: CPU instructions often depend on data type. For example:
   - ADD BYTE → add 8-bit integers
   - ADD DWORD → add 32-bit integers
3. **Memory Access**: Data type tells CPU **how many bytes** to read/write.
4. **Arithmetic and Logic Operations**: ALU performs operations differently for integers, floating-point, or logical values.

Summary Table

Data Type	Bits	Purpose	Example
Integer	8,16,32,64	Whole numbers	5, -10
Floating-point	32,64	Real numbers	3.14, -0.001
Character	8,16	Text symbols	'A', 'z'
Boolean	1 (or 8)	True/False	1 (true), 0 (false)
Pointer	32,64	Memory address	0x1A3F

Every piece of data in the CPU has a **type**, which determines **how it is stored, processed, and interpreted**. Understanding data types helps in **writing efficient programs** and understanding **CPU architecture**.

Fractional Binary number to Decimal number:

Here are some handy facts:

Power of 2	Base 2	Base 10
2^{-4}	.0001	.0625
2^{-3}	.001	.125
2^{-2}	.01	.25
2^{-1}	.1	.5
2^{0}	1	1
2^{1}	10	2
2^{2}	100	4
2^{3}	1000	8
2^{4}	10000	16
2^{5}	100000	32
2^{6}	1000000	64
2^{7}	10000000	128
2^{8}	100000000	256

N	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Bits as ASCII Codes:

ASCII: Each character is represented by a unique 8-bit code.

256 unique codes for special characters.

Each key on the keyboard has an 8-bit equivalent. Each key represents a letter, number, and special character.

ASCII Character Codes

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Examples:

var1: *db* *0x55* ; define a variable ‘var’ of size byte, initialized by 0x55

var2: *db* *0x55,0x56,0x57*; three bytes in succession

var3: *db* *'a‘* ; character constant 0x61 (ascii code of ‘a’)

var4: *db* *'hello',13,10,'$‘* ; string constant

var5: *dw* *0x1234* ; 0x34 0x12

var6: *dw* *‘A'* ; 0x41 **0x00 – complete to word**

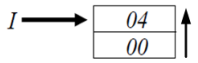
var7: *dw* *‘AB‘* ; 0x41 0x42

var8: *dw* *‘ABC'* ; 0x41 0x42 0x43 **0x00 – complete to word**

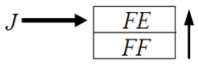
var9: *dd* *0x12345678* ; 0x78 0x56 0x34 0x12

Word Variables

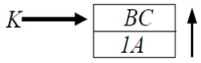
- **Assembler directive format defining a word variable**
  - name DW initial value
- **I DW 4**



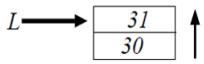
- **J DW -2**



- **K DW 1ABCH**

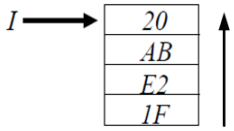


- **L DW "01"**

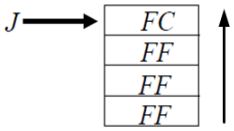


Double Word Variables

- **Assembler directive format defining a word variable**
  - name DD initial value
- **I DD 1FE2AB20H**



- **J DD -4**



2. Signal

A **signal** is any **physical quantity that conveys information**.

- It is a way to **communicate or represent data**.
- Signals can vary with time and can be **measured, transmitted, or processed**.

Examples in daily life:

- Sound waves (voice, music)
- Light signals (traffic lights)
- Electrical signals (voltages in wires)

What is a Signal Inside a Computer?

Inside a computer, a **signal** usually refers to **an electrical voltage or current** used to represent **binary data (0s and 1s) (Transistor)**.

- Computers are digital devices, so signals represent **two states**:
  - 0 → low voltage / off
  - 1 → high voltage / on
- Signals flow through **circuit components** like CPU, memory, and I/O devices.
- They are used to **carry instructions, data, and control information** on the transistors inside the computer.

Example:

- A CPU sends a **control signal** to memory to **read or write data**.
- Data signals carry the **actual binary information** between CPU and memory.

Types of Signals

Signals in general (and inside computers) can be classified as:

A. Analog Signals

- Vary **continuously** over time.
- Can take **any value** within a range.
- Represented as smooth waveforms (like sine waves).
- **Examples:** Sound, temperature, light intensity.

Inside computers: Analog signals are mostly **converted to digital** because computers are digital.

Digital Signals

- Take **discrete values** (usually two states: 0 and 1).
- Represented as **square waves** in electronics.
- **Examples inside a computer:** Binary data, control signals, clock signals.

Why digital signals?

- Easier to **store, process, and transmit** reliably.
- Less affected by noise compared to analog signals.

Classification of computer signals:

1. **Control Signals**:
    - Used to **manage operations** inside the CPU or between components (memory or I/O).
    - Example: Memory Read/Write, CPU Clock, Interrupt signals
2. **Data Signals**:
    - Carry **actual data** (binary 0s and 1s) between components (memory or I/O).
3. **Clock Signals**:
    - Special signals used to **synchronize operations** inside digital circuits.
    - Typically a periodic square wave.

4. Summary

Concept	**Description**
Signal	Physical quantity carrying information
Signal in computer	Electrical voltage representing binary data
Analog signal	Continuous, smooth, e.g., sound wave
Digital signal	Discrete, usually 0 or 1, e.g., CPU data
Control signal	Manages operations (e.g., memory read/write)
Data signal	Carries actual information
Clock signal	Synchronizes operations

3. Computer Functions

A computer performs a variety of **functions** by processing data. These functions can be broadly classified as:

1. **Data transfer functions** – Moving data between CPU and memory or I/O devices.
2. **Control and Interruption functions** – Managing the execution of instructions and timing.
3. Data Storage
4. Logical Circuit of CPU, Memory, and others
5. Data Processing:
   - Arithmetic functions – Operations involving numbers; + and bit shifting or rotating (-, *, /).
   - Logical functions – Operations involving logic and decision-making. OR, AND, NOT, XOR, XNOR, ...
   - Comparison: <, >, ≥, ≤, ==, ≠, ...
   - The microprocessor ALU (Arithmetic Logic Unit) only has addition operations, and multiplication and division operations are performed by shifting.

2. Arithmetic Function

Adding is part of the computer's **arithmetic operations**.

- It is performed by a component called the **Arithmetic Logic Unit (ALU)** inside the CPU.
- The ALU can add, subtract, multiply, divide, and perform other arithmetic operations.

Example of Addition in Binary:

Decimal: 3 + 5 = 8

Binary: 0011 + 0101 = 1000

How computers do it:

1. Binary numbers are input into the ALU.
2. ALU performs bit-wise addition using **logic gates**.
3. Result is stored in **registers** or memory.

All arithmetic operations in a computer are ultimately done using **binary numbers**.

3. Logical Functions

Logical functions allow a computer to **make decisions and comparisons**.

Logical operations are also done in the ALU.

Common Logical Operations:

Operation	Symbol	Description
AND	A · B	True if **both** inputs are 1
OR	A + B	True if **any** input is 1
NOT	¬A	Inverts the input (1→0, 0→1)

Operation	Symbol	Description
XOR	A ⊕ B	True if inputs are **different**

Example:

Binary AND:

A = 1010

B = 1100

A AND B = 1000

Logical functions are used in:

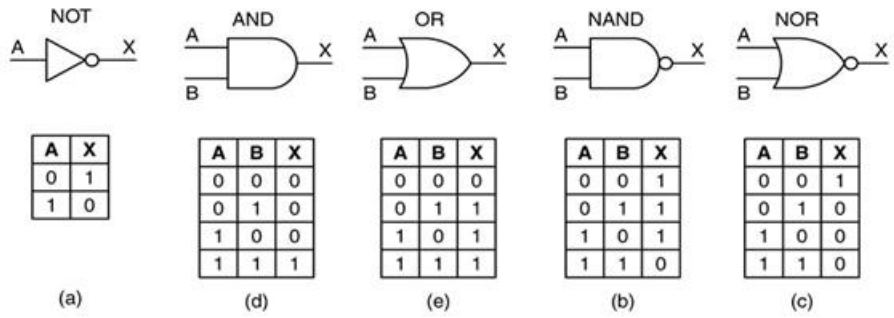
- Comparisons (if A > B)
- Decision-making in programs
- Controlling circuits (like turning devices on/off)

4. Summary Table

Function Type	Performed By	Example
Arithmetic (Adding)	ALU	3 + 5 = 8 (0011 + 0101 = 1000)
Logical	ALU	AND, OR, NOT, XOR on binary numbers
Data Transfer	Control unit	Move data from memory to register
Control	Control unit	CPU decides which instruction to execute next

The **ALU** is the heart of arithmetic and logical operations. All operations inside a computer, whether numbers or logic, are ultimately handled in **binary form**.

Logical Circuits:



A	X
0	1
1	0

(a)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

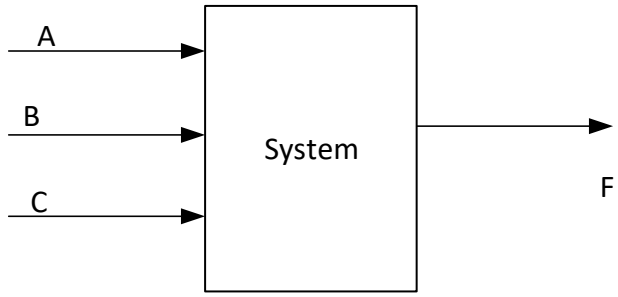
(c)

AND is multiply; OR is plus but not aritmetic with no carry.

Truth Table: 3-input majority function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

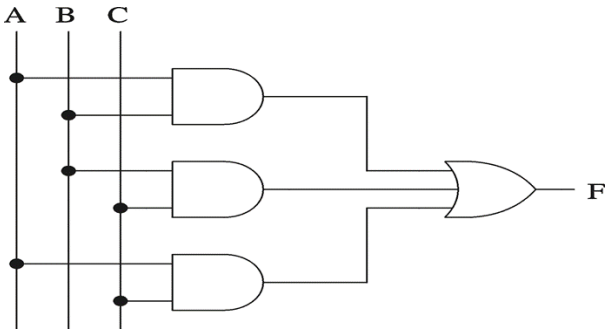
If there are 3 inputs, how many case is possible? 2^3=8 case are possible. A, B, C are inputs; F is output.



The system is logical circuit.

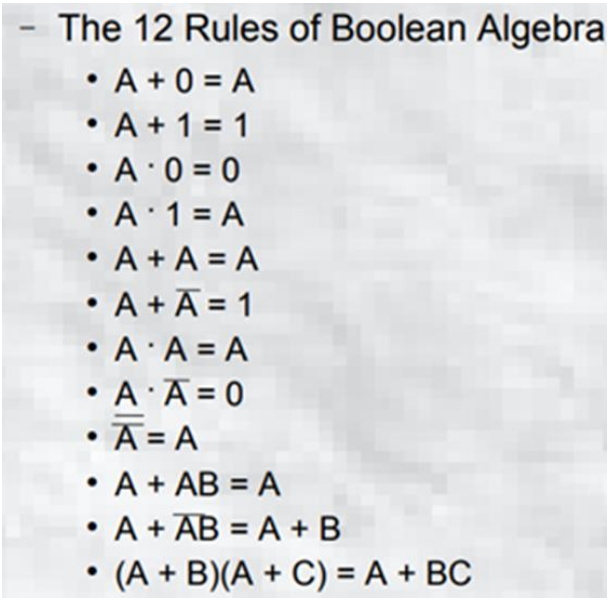
Logical expression form: If we prepare the logical expression, we take only output, F is 1. from the truth table. For example, if A is 1 we take A. If A is 0 we take A'.

F=A'BC+AB'C+ABC'+ABC
F = A B + B C + A C
In the system,



Boolean algebra: All variables are 1 or zero.

1. a) $a+b=b+a$ Değişme Özelliği
   b) $a \cdot b=b \cdot a$
2. a) $a+b+c=a+(b+c)$ Birleşme Özelliği
   b) $a \cdot b \cdot c=a \cdot (b \cdot c)$
3. a) $a+b \cdot c=$ $(a+b) \cdot (a+c)$ Dağılma Özelliği
   b) $a \cdot$ $(b+c)$ = $a \cdot b$ +$a \cdot c$
4. a) $a+a=a$ Değişkende Fazlalık Özelliği
   b) $a*a=a$
5. a) $a+a.b=a(1+\mathrm{b})$; $1+\mathrm{b}=1$ Yutma Özelliği
   b) $a \cdot (a+b)=a*\mathrm{a} + \mathrm{a}*\mathrm{b}=\mathrm{a}+\mathrm{a}*\mathrm{b}=\mathrm{a}(1+\mathrm{b})=\mathrm{a}$
6. a) (a)^n =a işlemde Fazlalık Özelliği
   b) (a x n) =a
7. a) $((\mathrm{a}+\mathrm{b})\overline{\ }=\bar{a} \cdot \bar{b}$ De Morgan Kuralı
   b) $((\mathrm{a} \cdot \mathrm{b})\overline{\ }=(\mathrm{a}\ )\overline{\ } + \bar{b}$
8. a) $a+\bar{a}$ =1 Sabit Özelliği
   b) $a*\bar{a}$ =0
9. a) 0+a=a Etkisizlik Özelliği
   b) 1$\cdot a$=a
10. a) 1+a=1 Yutan Sabit Özelliği
    b) 0$\cdot a$=0
11. a) $(a+b\ ) \cdot b=b$
    b) $a \cdot b$ +b=b

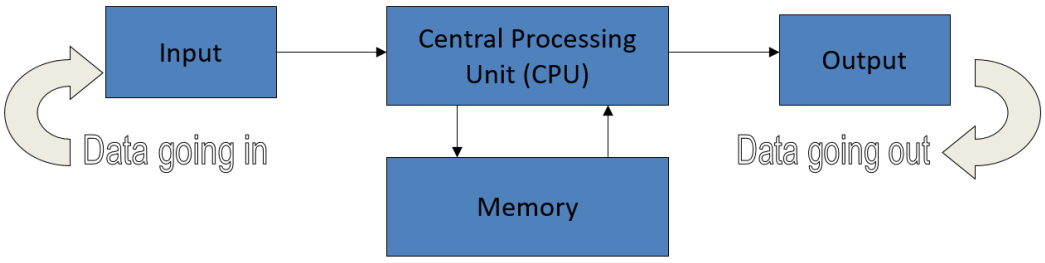


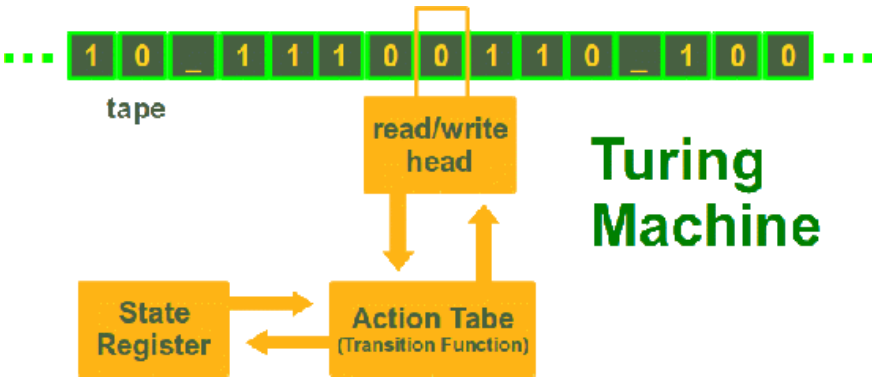
4. Roots of the Computer

The word "computer" has been used in English since 1646. In dictionaries prior to 1940, "computer" was a professional title given to people who performed calculations. Calculator is a machine that performs calculations. The modern definition and use of the term "computer" emerged with the development of the first electronic calculating devices. Almost all devices used today contain computers.

Joseph Marie Jacquard's Loom - weaves fabric using a pattern specified by punched cards (1801). Analytical Engine is conceptual design for a machine consisting of Mill, Store, Printer and Reader. Charles Babbage designed the Difference Engine and Analytical Engine, introduced the idea of a programmable machine, used mechanical gears and punch cards. First concept of CPU, memory, and input/output and foundation of modern computer architecture. Led by Ada Lovelace to define programming concepts such as subprogram.

“Harvard Mark I” used electrical relays and mechanical switches, very large and slow, programs executed sequentially (1930 – 1945). Hardware Characteristics: relay-based logic, limited memory, physical wiring for programming.





John V. Atanasoff and Clifford E. Berry, professors at Iowa State University, developed the Atanasoff-Berry Computer (ABC) between 1937 and 1942. The ABC was the first computer to use vacuum tubes instead of mechanical switches. It was also the first digital computer. Its computational processes were based on the binary number system. During the years Atanasoff was working on the ABC, a German engineer, Konrad Zuse, developed a computer called the Z3. Due to the Nazi regime in Germany and the subsequent Second World War, the design of this computer was kept secret. Information about this design emerged after the war.

In the 1930s, IBM was working on quite different computer architectures. In 1939, IBM supported a project by an engineer named Howard Aiken. 75 IBM Automatic Computing Machines were combined into a single unit. In 1943, a group led by John W. Mauchly and J. Presper Eckert began work on ENIAC (Electronic Numerical Integrator and Calculator). ENIAC was being developed for the United States military's wartime operations. It was completed three months after the end of the war, in November 1945.

ENIAC is one of the first programmable electronic computers (1945). Programmed with routing cables and rotary switches. ENIAC used vacuum tubes for processing, consumed huge amounts of electricity, produced excessive heat, extremely large (room-sized machines). von Neumann Machine stores programs in electronic memory along side the data (1943). It moves and manipulate a program like data. It is enabled high-level programming languages.

Second Generation (1950s–1960s) – Transistors:
Transistors replaced vacuum tubes. Transistors are smaller, faster, more reliable and lower power consumption.

Third Generation (1960s–1970s) – Integrated Circuits: Introduction of Integrated Circuits (ICs), multiple transistors on a single chip and more powerful and reliable systems.

Fourth Generation (1970s–1990s) – Microprocessors: CPU placed on a single chip, personal computers became possible and rapid miniaturization (Moore's Law).
Hardware Revolution: RAM chips, Hard disks, Personal computers

Fifth Generation (2000–Present) – Modern Hardware: NVIDIA GPUs

- Multi-core processors
- GPUs for parallel computing
- Solid State Drives (SSD)
- Cloud data centers
- AI accelerators (TPUs, NPUs)

Hardware Characteristics Today:

- Billions of transistors per chip
- Nanometer-scale fabrication
- Massive parallelism
- Energy-efficient architectures

A **TPU (Tensor Processing Unit)** is a specialized hardware accelerator designed by Google to speed up **machine learning** and **deep learning** workloads. It is an **Application-Specific Integrated Circuit (ASIC)** optimized for tensor operations (large matrix multiplications), which are the core computations in neural networks.

Traditional processors:

- **CPU** → General-purpose, flexible but slower for AI
- **GPU** → Highly parallel, faster for AI but power-hungry
- **TPU** → Designed specifically for AI → faster and more energy-efficient for neural networks

Neural networks mainly perform:
Matrix × Matrix=Matrix
TPUs are optimized for:

- Massive parallel matrix multiplication
- Low-precision arithmetic (e.g., 8-bit, bfloat16)
- High throughput

They use a **systolic array architecture**, which allows:

- Continuous data flow
- High parallelism
- Reduced memory bottlenecks

TPU, CPU and GPU

Feature	CPU	GPU	TPU
Purpose	General computing	Graphics & parallel tasks	AI-specific
Flexibility	Very high	Medium	Low (AI-focused)
Parallelism	Low	High	Extremely high
Energy Efficiency	Moderate	Lower	Very high
Best For	OS, logic	Graphics, ML	Deep learning

TPU is Used in:

- Google Search AI
- Google Translate
- Large language models
- Cloud AI services

Where Are TPUs Used?

- Google data centers
- Cloud platforms (Google Cloud TPU)
- Large-scale AI training clusters
-

They are mainly accessible through:

- TensorFlow
- JAX
- PyTorch (via XLA)

5. Computer Organization

Key Takeaways for Software Engineers

1. Understanding CPU internals helps optimize software performance.
2. Memory hierarchy knowledge (registers → cache → RAM → disk) improves data access strategies.
3. Instruction execution cycle (fetch, decode, execute) affects program design.
4. Hardware-software interface awareness helps in low-level programming, debugging, and compiler optimization.

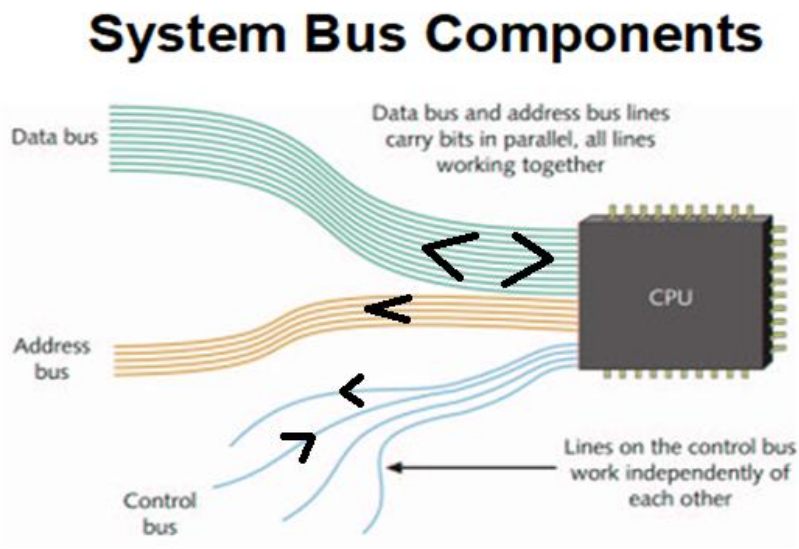
Computer: A programmable machine that takes data as input, stores and processes the data, and provides output data in a useful form.

- Input: Data
- Instructions: Software, Programs
- Output: Information (numbers, words,sounds, images)

Microprocessor is a program-controlled semiconductor device (IC) that transfers program instructions in memory to registers, decodes them, executes them and writes the results to memory. In computers, it is called CPU (Central Processing Unit).

Computer organization is the **operational structure of a computer system**, detailing how the hardware components work together to execute programs. It focuses on the **implementation and internal operations** of a system, as opposed to **computer architecture**, which focuses on the conceptual design and functional behavior.

A computer is a structure capable of processing data. A computer is a general-purpose device that is programmed to autonomously perform a set of arithmetic or logical operations. In the near future, thousands of processors will use quantization computing when working together to process data very quickly. Bits will be represented by electrons or photons. Information System: A system that receives, stores and processes data and provides information as output.

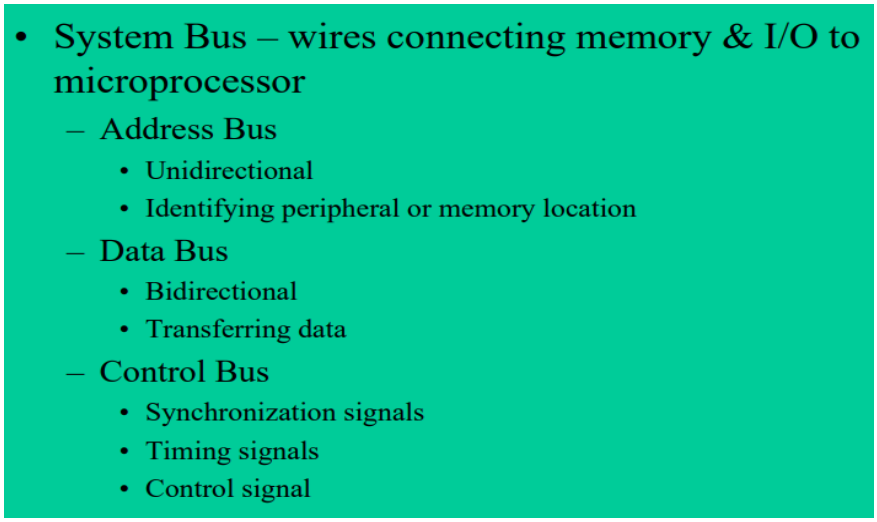


Key Points:

- Deals with **hardware-level operations**.
- Explains **how instructions are executed**.
- Shows how **CPU, memory, and I/O devices** interact.
- Important for **software engineers** to write efficient programs, understand system limitations, and optimize performance.

Example:

Knowing that accessing data from RAM is slower than accessing CPU registers can influence how a programmer structures memory-intensive operations.



Organization and Architecture

Though often used interchangeably, **organization** and **architecture** are different:

Aspect	**Computer Architecture**	**Computer Organization**
Focus	Logical design, instruction set, programmer view	Physical implementation, hardware operations
Scope	What the computer does	How the computer does it
Example	Number of registers, instruction types	Cache memory, control signals, pipelines

Differences Between Computer Architecture and Organization

Feature	**Computer Architecture**	**Computer Organization**
Definition	Defines the **functional behavior** of a computer visible to the programmer.	Defines the **hardware implementation** of the architecture.
Focus	*What* a computer does	*How* a computer does it
Visibility	Programmer-visible	Transparent to the programmer
Components Covered	Instruction set, addressing modes, data types, I/O behavior	CPU design, ALU, pipelines, control signals, memory hierarchy

Feature	Computer Architecture	Computer Organization
Impact on Software	Direct: determines **compatibility** of programs	Indirect: affects **performance and efficiency** of programs
Example	x86 ISA specifies instructions like ADD, MOV	Intel Core vs AMD Ryzen implement the same instructions differently in hardware (pipelines, cache, execution units)

Relationship Between Architecture and Organization

- **Architecture** defines the **blueprint**: the instructions, registers, data types, and I/O operations.
- **Organization** defines the **implementation**: how the CPU executes instructions, manages memory, and handles I/O.
- Two computers can have the **same architecture** but **different organizations**, meaning software behaves the same but performance may differ.
- Organization changes (e.g., faster cache or pipeline design) **do not affect program correctness**, only **execution speed**.
- Organization = engine, transmission, and fuel system (how the car actually moves).

Basic Components of a Computer:

1. **CPU (Central Processing Unit): Processes instructions.**
    - Heart of the computer.
    - Executes instructions.
    - Consists of **ALU, registers, control unit, and cache, system bus, clock and Timing, Flag registers**.
    - ALU (Arithmetic Logic Unit): Performs arithmetic and logical operations.
    - Control Unit: Directs operations and instruction flow.
    -
2. **Memory: Stores instructions and data.**
    - **Primary (RAM):** Fast, volatile storage. Read and Write memory (Data)
    - **Primary (ROM): Read Only memory (Instructions or programme)**
    - **Secondary (Disk/SSD):** Slower, non-volatile storage.
    - **Cache:** Small, ultra-fast memory in the CPU.
3. **I/O Devices: Interface for interacting with the outside world.**
    - Keyboard, mouse, monitor, network interface.
    - Can be **input, output, or both**.
4. **System Bus: Transfers data between CPU, memory, and I/O.**
    - Data highway connecting CPU, memory, and I/O.
    - Types: Data bus, address bus, control bus.
5. **Clock and Timing**

Characteristics:

- Instructions and data share the same memory.
- Program execution is **sequential** (one instruction at a time).
- **Fetch-Decode-Execute Cycle** governs instruction execution.

Programm memory is ROM (ROM is read only memory); Data memry is Ram (RAM is read and wirite memory).

RAM is **volatile main memory** used to temporarily store data and instructions that the CPU is currently using. When the power is turned off, **data does not remains**.

- Volatile memory
- Very fast access speed (nanoseconds)
- Directly connected to CPU
- Read and write capable

What is Stored in RAM?

- Running programs
- Operating system processes
- Active data (variables, buffers)
- Temporary computation results

ROM is **non-volatile memory** that permanently stores critical system instructions.
When the power is turned off, **data remains**.

- Non-volatile
- Mostly read-only
- Slower than RAM
- Stores firmware

What is Stored in ROM?

- BIOS/UEFI firmware
- Bootloader instructions
- Embedded system firmware
-

When you power on a computer:

1. CPU reads instructions from ROM.
2. ROM loads the bootloader.
3. Operating system loads into RAM.

Hardware–Software Interface

Hardware–Software Interface is the **layer where software interacts with hardware**. Understanding this interface helps software engineers optimize performance and troubleshoot low-level issues.

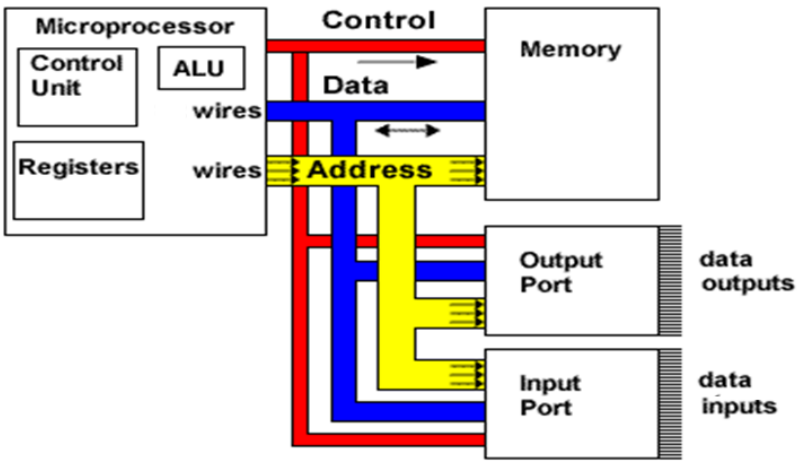
Components:

1. **Instruction Set Architecture (ISA):**
    - Defines the set of machine instructions a processor can execute.
    - Acts as the **boundary between software and hardware**.
    - Example: ADD R1, R2 adds two registers.
2. **Microarchitecture / Organization:**
    - How the CPU implements the ISA.
    - Includes pipelines, caches, control signals.
3. **System Software (Operating System):**
    - Manages memory, I/O, scheduling, and resources.
    - Abstracts hardware complexity for application programs.

Microprocessor: A CPU integrated on a single chip.
Handles **arithmetic, logic, control, and I/O operations**. Data transfer
Examples: Intel Core i7, AMD Ryzen, ARM Cortex.

Why Microprocessors Matter for Software Engineers:

- Determines performance limits.
- Influences optimization strategies for software.
- Multi-core processors allow **parallel processing**.



Common Terms:

- **Clock speed (GHz):** How many instructions write or read at per second.
- **Word size (bits):** How much data bus lines CPU can handle at once.
- **Registers:** Small storage inside CPU for fast access.

Impact on Software

1. **Software Development & Compatibility:** Architecture defines the **instruction set** and data types, which determines what software can run on the system. Example: A program compiled for ARM architecture won't run on x86 architecture without translation.
2. **Performance Optimization:**
   Organization affects **execution speed**:
   - CPU pipelines: deeper pipelines can execute more instructions per cycle.
   - Cache size and memory hierarchy: more cache reduces memory access time.
   - Multi-core or parallel execution units: software can leverage concurrency.
3. **Compiler and OS Design:**
   - Compilers generate machine code based on architecture.
   - OS schedules tasks considering organizational features (cache, cores, memory latency).

Impact on Performance

Organizational Feature	Performance Impact	Example
Pipelining	Increases instruction throughput by overlapping execution	Deeper pipelines in Intel CPUs execute more instructions per clock
Cache Memory	Reduces memory access time, improves speed	Larger L1/L2/L3 caches store frequently used data
Branch Prediction	Minimizes pipeline stalls	Predicts next instruction path to keep CPU busy
Multi-Core Processors	Enables parallel execution of threads	AMD Ryzen has multiple cores for simultaneous program execution
Memory Bandwidth	Determines data transfer speed between CPU and RAM	Faster buses improve large data processing

Even if the architecture (ISA) is the same, **organization differences can make one CPU much faster than another**. Software engineers can optimize code based on **organization knowledge**, e.g., using cache-friendly data structures, minimizing pipeline stalls.

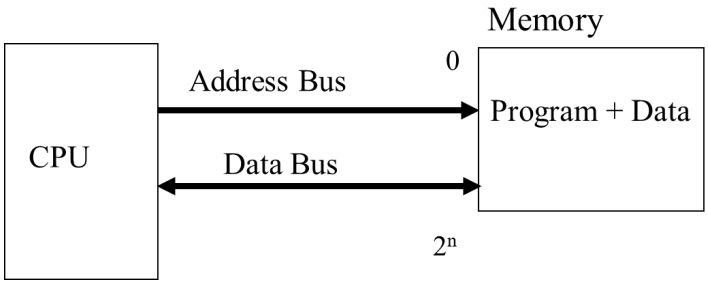
- **Computer Architecture:** defines *what instructions exist* and *how the computer appears to the programmer*.
- **Computer Organization:** defines *how the hardware implements these instructions*.
- **Relation:** Architecture determines software behavior; organization determines performance.
- **Impact on Software:** Architecture ensures **compatibility**, organization affects **efficiency and speed**.

"Architecture tells the computer what to do; organization tells it how fast and efficiently it can do it."

Von Neumann Architecture - Harvard Architecture:

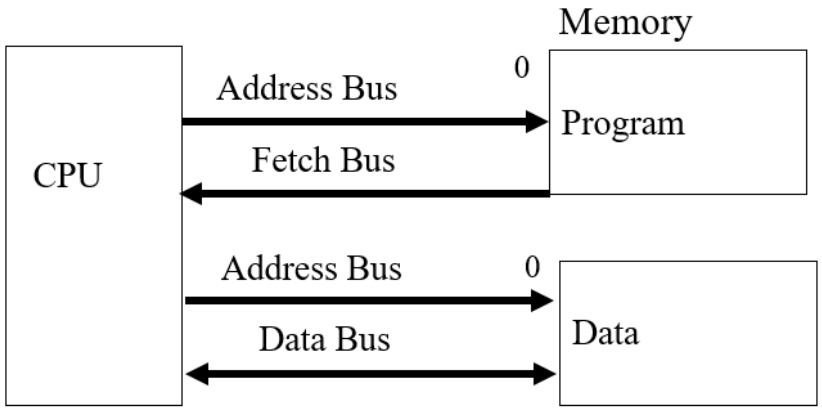
Feature	Von Neumann	Harvard
Memory	Unified (data + instructions)	Separate (data and instructions)
Speed	Potential bottleneck (Von Neumann bottleneck)	Faster due to parallel access
Example	Most PCs	Microcontrollers, DSPs

Most modern computers are based on the **Von Neumann architecture**, which has a **single memory system** storing both data and instructions.



There are two memory program and data; but there is only one address bus and only one data bus. Address bus select memory and memory cells or I/O units. Data bus CPU reads or writes to /from memory or I/O units.

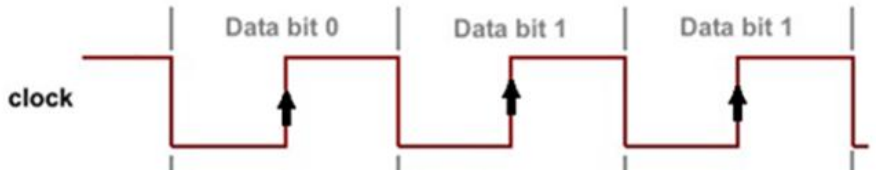
Harvard Architecture:



Clock and Timing (Microprocessor clock)

Also called clock rate, the speed at which a microprocessor executes instructions. Every computer contains an internal clock that regulates the rate at which instructions are executed and synchronizes all the various computer components. The CPU requires a fixed number of clock ticks (or clock cycles) to execute each instruction. The faster the clock, the more instructions the CPU can execute per second. Clock speeds are expressed in megahertz (MHz) or gigahertz ((GHz). Some microprocessors are superscalar, which means that they can execute more than one instruction per clock cycle. Like CPUs, expansion buses also have clock speeds. Ideally, the CPU clock speed and the bus clock speed should be the same so that neither component slows down the other. In practice, the bus clock speed is often slower than the CPU clock speed, which creates a bottleneck. This is why new local buses, such as AGP, have been developed.

In a digital computer system, the **clock period** (also called **clock cycle time**) is the **time it takes for one complete cycle of the system clock signal**. The clock synchronizes all operations inside the CPU. A clock signal consists of square waves that alternates between 0 and 1:



Each full down-and-up wave is **one clock cycle**.
The **clock period (T)** is: T=1/f
Where:

- **T** = clock period (in seconds)
- **f** = clock frequency (in Hertz)

Example: If a processor runs at **2 GHz** (2×10^9 cycles per second)
Then: T=1/(2×10^9)=0.5 * 10^(-9) second, T=0.5 nanoseconds. So one clock cycle takes **0.5 ns**.

If at 1 clock period, a CPU writes 32 bits, at 1 second how much bits writes?

0.5*10^(-9) second a CPU writes 32 bits
1 second ?bits

X=32/(0.5*10^(-9))=64 * 10^9 bits/second = 1 Gigabits/second

Clock Period is important, because the clock period determines:

- How fast instructions can execute
- The maximum speed of combinational logic
- System timing constraints

Shorter clock period → Higher frequency → More cycles per second

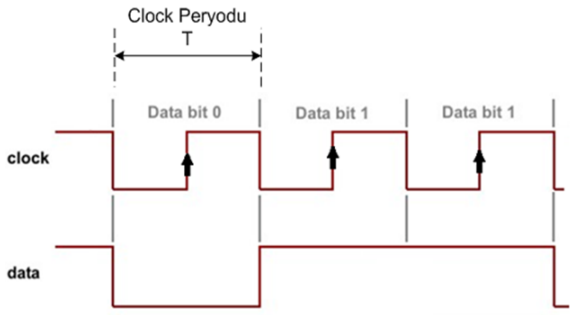
But: Performance ≠ only frequency
Because real performance depends on:

$$\text{Execution Time} = \frac{\text{Instructions} \times \text{Cycles per Instruction}}{\text{Clock Frequency}}$$

Examples: 4 instructions, cycles per instruction ise 20, clock period is 0.5 nansecond. What is the exeuction time?

Execution time= (4*20)/(0.5 * 10^-9)=160*10^9 =160 Gigabit /second

Clock synchronizes all CPU and BUS operations. Machine (clock) cycle measures time of a single operation. Clock is used to trigger datas. Clock determines the instruction processing time.

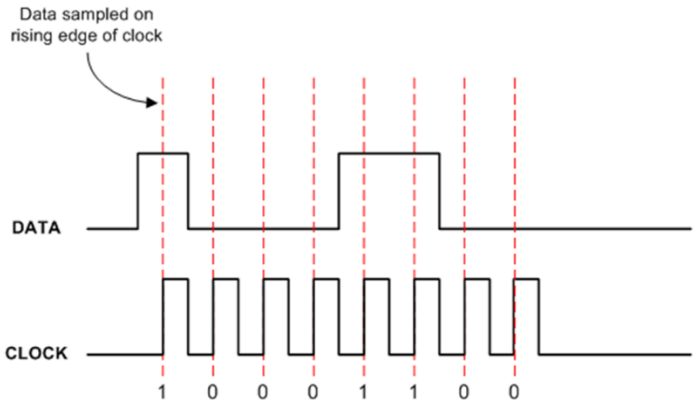


What is the data? 0 1 1 (Look at the trigeer level)

Every computer has a system clock pulse signal. It is continuous, existing in infinity and continuing to infinity. The clock pulse signal is a sequence of pulses (electrical signal) consisting of 1s and 0s.

One clock period is equal to one data length time. It is used to trigger, or process, data defined as bits. Personal computer speeds are usually expressed in gigahertz (GHz).

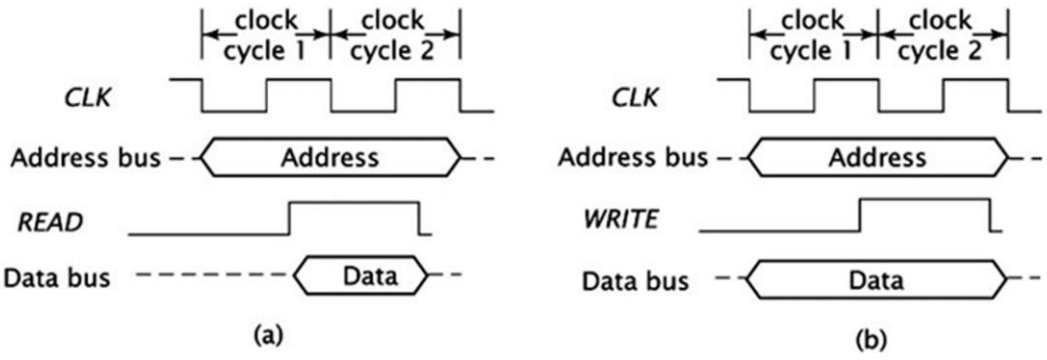
Example: Clock signal: Triggering with rising or falling edge. If f=2GHz and the number of data bus lines=32 bits, determine the amount of bits to be written to memory in 1 second.
Here, f=frequency, which is the operating frequency of the microprocessor.
Its unit is: Hz=1/second
Clock period=clock cycle, T=1/f=1/2GHz=1/(2*10^9Hz)
T=0.5*10^-9 seconds
If 32 bits are written to the memory locations in 0.5*10-9 seconds, how many bits are written in 1 second? X=32/(0.5*10-9)=64*10^9 bits/second=64 Gigabit/second



Reading from Memory:
Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU.
The steps are:

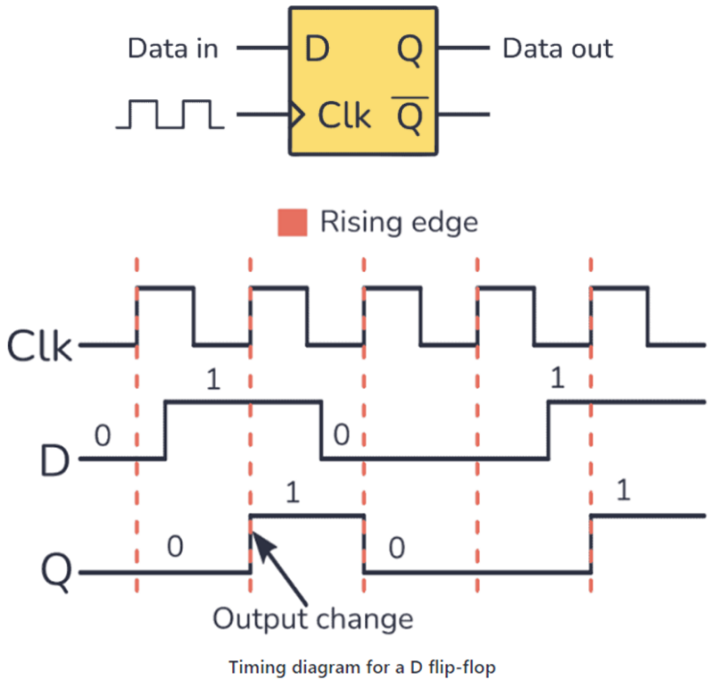
- address placed on address bus (select memory and memory cell or I/O)
- Read Line (RD) set low
- CPU waits one cycle for memory to respond
- Read Line (RD) goes to 1, indicating that the data is on the data bus

Instruction Fetch (Read(a) / Write (b)):



D Flip-Flop:

D flip-flops are synchronous circuits that require a clock signal (Clk). When the clock signal changes from 0 to 1 (rising edge: triggering), the Q output of the D flip-flop will store a new value from the D input. The state of the Q output remains unchanged as long as the clock signal does not change.



Timing diagram for a D flip-flop

6. Computer Architecture

Although the terms **computer architecture** and **computer organization** are often used interchangeably, they refer to **different aspects of a computer system**:
Computer Organization focuses on **how a computer does it. T**he **hardware implementation and operational structure**.
Computer Architecture focuses on **what a computer does. T**he **functional behavior and design principles** visible to a programmer. Computer architecture defines the logical structure, design, and functionality of a computer system. It specifies the instruction set, data types, addressing modes, and I/O operations.
Think of it as **"design - Architecture and implementation - Organization "**.

Computer Architecture Features:

1. **Instruction Set Architecture (ISA):** Defines the **machine-level instructions** that a processor can execute. Example: ADD R1, R2 adds two registers; LOAD R1, 0x1000 loads data from memory.
2. **Data Types and Formats:** Determines what data types are supported: integers, floating-point, characters, etc.
3. **Addressing Modes:** Explains how the CPU accesses operands: direct, indirect, register, immediate, or indexed.
4. **I/O and Memory Access:** Defines how input/output devices and memory interact with the CPU.

CPU Components: Registers, ALU, control unit, cache, pipelines, system buses, timing - clok.
Memory System Design: Cache levels (L1, L2, L3), RAM – ROM access, virtual memory.
Control Signals (Lock period): Hardware signals to execute instructions correctly.
Performance Optimization: Pipelining, parallelism, branch prediction, multi-core design.

Organization is **transparent to the programmer**. A software engineer typically interacts with **architecture**, but understanding organization helps **write faster, more efficient code**.

Key Differences

Feature	Computer Architecture	Computer Organization
Focus	Functionality, behavior, ISA	Implementation, hardware structure
Visibility	Visible to programmer	Invisible to programmer
Concern	What a computer does	How a computer does it
Example	Number of registers, instruction types	ALU design, pipeline depth, cache size

Feature	Computer Architecture	Computer Organization
Software Impact	Direct: programming, compiler design	Indirect: optimization, performance tuning
Flexibility	Changes affect program compatibility	Changes affect performance but not correctness

Examples in Real Life

1. **ISA Perspective (Architecture):** Intel x86 and AMD Ryzen both use the **x86 instruction set**, so software behaves the same on both.
2. **Microarchitecture Perspective (Organization):** Intel and AMD CPUs execute the same instructions differently:
   - Intel may have deeper pipelines, larger caches.
   - AMD may have higher core count or different execution units.

Software Engineering Students

1. **Program Optimization:** Knowing organization allows better use of registers, cache, and memory hierarchy.
2. **System-Level Understanding:** Software interacts with hardware through architecture (ISA) and indirectly through organization (performance).
3. **Debugging and Profiling:** Awareness of pipelines, branching, and memory access patterns helps identify performance bottlenecks.
4. **Cross-Platform Development:** Architecture compatibility ensures code runs on different hardware; organization affects performance tuning.

Summary

- **Computer Architecture = What the computer does.**
  - Instruction set, data types, addressing modes, I/O behavior.
- **Computer Organization = How the computer does it.**
  - CPU design, pipelines, memory hierarchy, control signals.
- **For software engineers:**
  - Architecture ensures **functionality and compatibility**.
  - Organization influences **performance and efficiency**.

Mnemonic Tip:

"Architecture tells you the rules; organization tells you the tools."

Examples of CPU Designs

1. Intel x86 Series (e.g., Intel Core i7)

Architecture: x86-64 (CISC – Complex Instruction Set Computer)

Organization Highlights:

- Deep pipelines (up to 14–19 stages in some generations)
- Large multi-level caches (L1, L2, L3)

- Out-of-order execution to optimize instruction throughput
- Hyper-Threading for simultaneous multithreading

Impact on Software & Performance:

- Compatible with legacy x86 programs due to architecture stability
- High clock speed and large cache improve performance for single-threaded and multi-threaded applications
- Complex instruction set allows compact code but increases decoding complexity
- hows how **organization (pipelines, caches, multithreading)** improves performance without changing the instruction set.

2. AMD Ryzen Series

Architecture: x86-64 (CISC)

Organization Highlights:

- "Zen" microarchitecture with simultaneous multithreading (SMT)
- Multiple cores (4–16 cores per CPU)
- Unified L3 cache shared across cores
- Advanced branch prediction and prefetching

Impact on Software & Performance:

- Same ISA as Intel, so software runs the same (compatibility maintained)
- Organization with multiple cores and large shared cache allows **parallel workloads** to run efficiently
- High core count benefits server workloads and parallel computing applications
- Emphasizes **organization features (multi-core, cache design)** affecting performance in parallel programs

3. ARM Cortex-A Series (e.g., Cortex-A78)

Architecture: ARMv8-A (RISC – Reduced Instruction Set Computer)

Organization Highlights:

- Simpler, fixed-length instructions
- Out-of-order execution pipeline (~13 stages)
- L1/L2 cache per core, optional L3 cache for multi-core clusters
- Low-power design, optimized for mobile devices

Impact on Software & Performance:

- RISC ISA simplifies instruction decoding → lower power consumption
- Pipeline and cache organization improve efficiency without increasing complexity
- Software compiled for ARM can be very efficient on mobile and embedded devices

Key Takeaways:

- Shows **architecture differences (RISC vs CISC)** and how organization optimizes **power and speed** for specific use cases

4. NVIDIA GPUs (e.g., Ampere Architecture)

Architecture: SIMT (Single Instruction Multiple Thread)
Organization Highlights:

- Thousands of small cores (streaming multiprocessors)
- Specialized memory hierarchy (shared memory, L1/L2 caches)
- Massive parallelism for vector/matrix operations

Impact on Software & Performance:

- Ideal for AI, graphics, and scientific computing due to organization supporting parallel workloads
- Software needs to use GPU programming models (CUDA/OpenCL) to exploit hardware organization
- ISA defines instruction types, but performance is mostly dictated by **organization and parallel structure**
- Demonstrates how **organization dominates performance** for specialized computing tasks

5. Comparative Insights

CPU Design	ISA Type	Organization Highlights	Best For
Intel Core i7	x86-64 (CISC)	Deep pipelines, out-of-order, hyper-threading, large caches	General-purpose, high-performance desktop/laptop
AMD Ryzen	x86-64 (CISC)	Multi-core, SMT, shared L3 cache	Multi-threaded workloads, servers
ARM Cortex-A78	ARMv8-A (RISC)	Low power, simple decoding, efficient pipelines	Mobile devices, embedded systems
NVIDIA GPU (Ampere)	SIMT	Thousands of cores, shared memory, parallel execution	AI, graphics, HPC

1. **Architecture ensures compatibility:** x86 programs run across Intel/AMD CPUs.
2. **Organization determines efficiency:** pipeline depth, cache, cores, and execution order affect speed and scalability.
3. **Optimization strategies depend on organization:**
    - CPU: cache-aware programming, minimizing branch misprediction.
    - GPU: maximizing parallel threads and memory coalescing.

Von Neumann Architecture

The **Von Neumann Architecture**, proposed by **John von Neumann in 1945**, is the basis for most modern computers. It describes a design in which **program instructions and data share the same memory space**. A computer stores both **instructions** and **data** in memory and executes them sequentially.

A Von Neumann computer has **five primary components**:

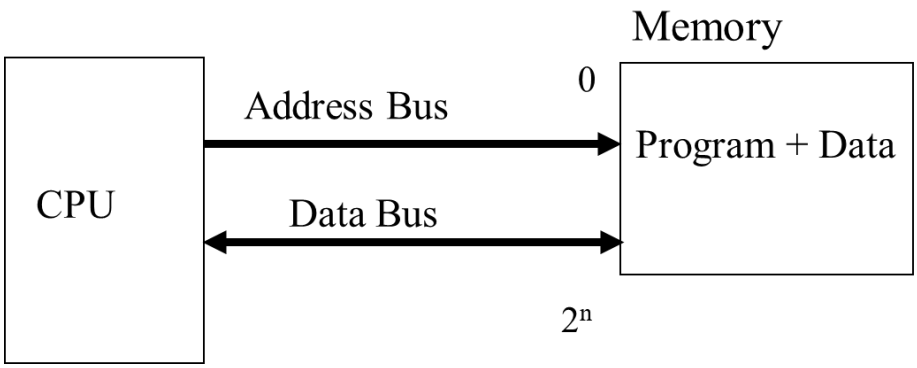
1. **Memory Unit**
    - Stores both **data** and **program instructions**.
    - Organized as a sequence of addressable locations (RAM).
    - Each location holds **binary data** (bits/bytes).
2. **Central Processing Unit (CPU)**
    - Performs all computations and controls the execution of instructions.
    - Contains:
        - **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations.
        - **Registers:** Small, high-speed storage locations inside the CPU.
        - **Control Unit (CU):** Directs the flow of data and instructions, generating control signals.
3. **Input Unit:** Accepts data and instructions from external sources (keyboard, sensors).
4. **Output Unit**: Sends processed results to external devices (monitor, printer).
5. **Bus System**: Data, address, and control lines that **connect CPU, memory, and I/O**.

Instruction Execution: The Fetch-Decode-Execute Cycle

Von Neumann machines operate sequentially using the **Fetch-Decode-Execute cycle**:

1. **Fetch:** CPU retrieves the next instruction from memory using the program counter (PC).
2. **Decode:** The control unit interprets the instruction to determine the operation and operands.
3. **Execute:**
    - ALU performs the operation (arithmetic, logical, or data movement).
    - Results may be stored in registers or written back to memory.
4. **Increment PC:** Program counter moves to the next instruction.

Diagram:
Memory (Instructions + Data)



4. Characteristics

1. **Single Memory for Instructions and Data**
    - Both program and data reside in the same memory space.
2. **Sequential Execution**
    - Instructions are executed one at a time in order.
3. **Program and Data Flexibility**
    - A program can be modified by another program or itself (stored-program concept).
4. **Simplicity**
    - Easy to design and implement compared to other architectures like Harvard.

5. Advantages

- Simple and cost-effective design.
- Flexibility in programming and self-modifying code.
- Widely used in general-purpose computers.

6. Disadvantages (Von Neumann Bottleneck)

- **Memory Bottleneck:** Instructions and data share the same bus → CPU often waits for memory access.
- **Sequential Execution:** Limits performance for modern multi-threaded or parallel tasks.
- **No Parallelism:** Cannot fetch and execute multiple instructions simultaneously.

Mitigations in Modern CPUs:

- Cache memory to reduce memory latency.
- Pipelining to execute multiple instructions at different stages.
- Superscalar architectures and multi-core processors.

7. Comparison with Harvard Architecture

Feature	Von Neumann	Harvard
Memory	Single memory for data & instructions	Separate memory for data & instructions
Speed	Potential bottleneck (shared bus)	Faster (parallel access)
Complexity	Simple	More complex
Usage	Most general-purpose CPUs	Microcontrollers, DSPs

8. Key Takeaways for Software Engineers

1. **Understanding the Fetch-Decode-Execute cycle** helps with debugging and optimization.
2. **Memory hierarchy knowledge** (registers → cache → RAM) is critical for efficient software.
3. **Bottleneck awareness** explains why CPU can be idle despite high-speed processing.
4. Many modern CPUs are **Von Neumann-based but optimized** with caches, pipelines, and multi-core architectures.

Main Components of Von Neumann Architecture

A. Central Processing Unit (CPU)

The CPU is the **brain of the computer**, executing instructions and processing data. It has three main parts:

1. **Arithmetic Logic Unit (ALU)**
    - Performs all **arithmetic** (addition, subtraction, multiplication, division) and **logical** (AND, OR, NOT, comparisons) operations.
2. **Control Unit (CU)**
    - Directs the operation of the computer.
    - Coordinates **fetching, decoding, and executing instructions**.
    - Sends **control signals** to ALU, memory, and I/O devices.
3. **Registers**
    - Small, high-speed storage locations inside the CPU.
    - Store **temporary data, instruction addresses, or results** during computation.

B. Memory Unit

- Stores **both data and program instructions** (shared memory).
- Organized as a series of **addressable locations** (bytes or words).
- Key role: provides **fast access** to instructions and data for the CPU.
- Types in practice:
    - **Primary memory (RAM)**: volatile storage for active programs/data
    - **Cache memory**: small, very fast memory for frequently accessed data

C. Input/Output (I/O) Units

- **Input devices:** Receive data from external sources (keyboard, mouse, sensors).
- **Output devices:** Send processed data to the user or other systems (monitor, printer, actuator).
- CPU communicates with I/O devices via the **bus system**.

D. Bus System (Connecting Component)

- Transfers **data, addresses, and control signals** between CPU, memory, and I/O.
- Types of buses:
    - **Data bus:** moves actual data
    - **Address bus:** specifies memory/I/O addresses
    - **Control bus:** carries signals to manage operations

Stored-Program Concept

Definition:

- Instructions of a program are **stored in memory alongside data**, rather than being hardwired into hardware.
- Introduced by John von Neumann in 1945.

Key Advantages:

1. **Flexibility:** Programs can be loaded, modified, or replaced without changing hardware.
2. **Self-Modifying Code:** Programs can alter their own instructions in memory if needed.
3. **Simpler Hardware Design:** No need for separate instruction storage mechanisms.

Practical Implication:

- Modern software development relies on the stored-program concept: programs are **loaded from storage to memory** and executed by the CPU.

4. Instruction Execution: The Fetch-Decode-Execute Cycle

The stored-program concept works hand-in-hand with this cycle:

1. **Fetch:** Retrieve the instruction from memory using the **program counter (PC)**.
2. **Decode:** Control Unit interprets the instruction to determine the operation and operands.
3. **Execute:** ALU performs the operation; results stored in registers or memory.
4. **Increment PC:** Move to the next instruction.

5. Summary

- **Components:** CPU (ALU, CU, registers), Memory (instructions + data), I/O, Bus.
- **Stored-Program Concept:** Programs stored in memory like data → allows flexible, general-purpose computing.
- **Importance for Software Engineers:** Understanding these concepts helps in optimizing programs, debugging, and designing efficient software systems.

Von Neumann vs Harvard architecture

Absolutely! Let's compare **Von Neumann** and **Harvard architectures** in detail, focusing on their differences, advantages, and impact on software and performance. This is essential for software engineering students to understand how hardware design influences computation.

---

1. Overview

Feature	Von Neumann Architecture	Harvard Architecture
Memory Structure	Single memory stores both instructions and data	Separate memories for instructions and data
Data Bus	Shared between instructions and data	Separate buses for instructions and data
Instruction Execution	Sequential access; fetch instructions and data using the same bus	Instruction and data can be accessed simultaneously
Complexity	Simple design, easier to implement	More complex hardware design
Flexibility	Flexible, easy to modify programs in memory	Less flexible; instruction memory often fixed
Usage	General-purpose computers, PCs, laptops	Microcontrollers, DSPs, embedded systems

2. Von Neumann Architecture

1. Single memory stores both **program instructions and data**.
2. CPU fetches instructions **sequentially** from memory using a shared bus.
3. Simpler and cost-effective design.
4. Limitation: **Von Neumann bottleneck** — CPU may be idle while waiting for memory access because instructions and data share the same bus.

Pros:

- Easier to design and implement
- Flexible and programmable

Cons:

- Limited performance due to shared instruction/data bus
- Sequential execution slows processing

3. Harvard Architecture

Key Features:

1. Separate memory for **instructions** and **data**.
2. Separate buses allow **simultaneous access** to instructions and data.
3. Commonly used in **microcontrollers, embedded systems, DSPs** (Digital Signal Processors).
4. Can have **different word sizes** for instructions and data.

Pros:

- Higher performance due to **parallel access**
- Efficient for real-time and embedded applications
- Reduces instruction/data memory conflicts

Cons:

- More complex hardware
- Less flexible for modifying instructions at runtime

4. Comparison Table

Aspect	**Von Neumann**	**Harvard**
Memory	Single (instructions + data)	Separate (instructions vs data)
Bus	Shared	Separate
Speed	Slower (memory bottleneck)	Faster (parallel access)
Complexity	Simple	More complex
Programmability	Flexible	Less flexible
Common Applications	PCs, general-purpose computers	Microcontrollers, DSPs, embedded devices

5. Practical Implications for Software Engineers

1. **Von Neumann Systems:**
    - Standard PC programming fits well.
    - Optimize code for memory access due to shared instruction/data bus.
    - Example: Cache-aware programming improves performance.
2. **Harvard Systems:**
    - Useful for embedded systems where **speed and predictability** matter.
    - Software must respect separate instruction/data memory layouts.
    - Example: DSP code can fetch data while executing instructions in parallel → real-time processing.

Von Neumann Architecture: Single memory, simpler, flexible, but limited by the memory bottleneck. **Harvard Architecture**: Separate instruction/data memory, faster, ideal for embedded systems, but more complex. **Key Lesson:** Architecture design affects **software efficiency, memory usage, and execution speed**.

Hardware-Software Interface

The **hardware-software interface** is the **boundary where software interacts with hardware**. Understanding this interface is crucial for software engineers to write **efficient programs** and understand **how programs execute at the machine level**.

Key Idea:

- Software tells the hardware **what to do** using a **set of instructions**, and hardware **executes these instructions**.
- This interface is mostly defined by the **Instruction Set Architecture (ISA)**.

2. Instruction Sets and Machine Language

A. Instruction Set Architecture (ISA)

Definition:

ISA is the **set of instructions that a CPU can execute**. It defines the interface between **hardware and software**.

Components of an ISA:

1. **Instructions:** Commands the CPU understands, such as:
    - Arithmetic: ADD, SUB
    - Logical: AND, OR, NOT
    - Data transfer: LOAD, STORE
    - Control: JUMP, CALL, RETURN
2. **Data Types:** Integers, floating-point numbers, characters, etc.
3. **Registers:** Number and type of registers available.
4. **Addressing Modes:** How the CPU accesses operands (immediate, direct, indirect, indexed).
5. **I/O Instructions:** Communicate with input/output devices.

Example:

For x86 architecture:

```
ADD R1, R2  ; Add contents of R2 to R1
MOV R3, #5  ; Load immediate value 5 into register R3
```

B. Machine Language

Definition:

Machine language is the **binary representation of instructions** that the CPU executes directly.

Example:

- Assembly instruction: ADD R1, R2
- Machine code (binary): 0001 0101 0010 0001

Key Points:

- Machine language is **hardware-specific**.
- Programs written in high-level languages must be **translated** into machine language to execute.

3. Compiler and Assembler Basics

A. Assembler

- Converts **assembly language** (human-readable mnemonics) into **machine code**.
- Example:

Assembly: MOV R1, #10

Machine code: 0011 0001 1010

- **Role:** Bridges **low-level programming** and **hardware execution**.
- Useful for **system-level programming**, OS kernels, embedded systems.

B. Compiler

- Converts **high-level language programs** (e.g., C, Java, Python) into **machine code** or **assembly code**.
- Steps:
    1. **Lexical Analysis:** Break source code into tokens.
    2. **Syntax Analysis:** Check grammar rules.
    3. **Semantic Analysis:** Ensure operations make sense (e.g., type checking).
    4. **Optimization:** Improve performance (reduce instructions, memory usage).
    5. **Code Generation:** Produce assembly or machine code.
    6. **Linking:** Combine with libraries to form executable.

Example:

int a = 5 + 3;

- Compiled into:
    - Assembly instructions: MOV R0, #5; ADD R0, #3
    - Machine code: binary equivalent of above instructions

C. Relation Between Hardware and Software

1. **Hardware** executes **machine code**.
2. **Assembly language** is a human-readable representation of machine instructions.
3. **High-level programs** must be compiled/assembled to machine code.
4. **ISA** defines the “contract” between software and hardware.

Diagram (Simplified Flow):

High-Level Language (C, Java)

|

Compiler

↓

Assembly Language (ADD R1, R2)

|

Assembler

↓

Machine Code (Binary)

|

CPU executes

4. Key Takeaways for Software Engineers

1. **ISA is the critical interface**: understanding it helps optimize software.
2. **Machine language is hardware-specific**: software must be translated via compiler/assembler.
3. **Optimizations rely on organization knowledge**: caches, pipelines, and registers affect performance.
4. **Assemblers and compilers bridge the gap** between human-readable code and CPU-executable instructions.

7. Central Processing Unit (CPU)

CPU story all started with the development of the transistor in 1947.

http://www.computerhistory.org/exhibits/microprocessors/index.page

Year	name	Data size	memory size	#instructions	
1971	4004	4	4096 4-bit	45	first microprocessor
1973	8008	8	16K bytes	48	1st 8-bit µP
1973	8080	8	64K bytes		10 times faster than 8008
1973	MC6800	8	64K bytes		1st Motorola µP
1977	8085	8	64K bytes	246	Intel's most successful 8-bit general-purpose µP due to its low cost
	Z80	8			Zilog's most successful microprocessor
1978	8086	8,16	1M bytes	>20,000	1st 16-bit µP
1979	8088	8,16	1M bytes		prefetch instruction using cache
1981	IBM decided to use 8088 in its personal computer				
1983	80286	8,16	16M		
1986	80386	8,16,32	4G		
1989	80486	8,16,32	4G		
1993	Pentium	8,16,32	4G		
1995	Pentium Pro	64	64G		
1997	Pentium II	64	64G		
1999	Pentium III	?	?		
2000	Pentium 4	?	?		

The **Central Processing Unit (CPU)** is the **brain of the computer**. It performs **all computations, decision-making, and instruction execution**.

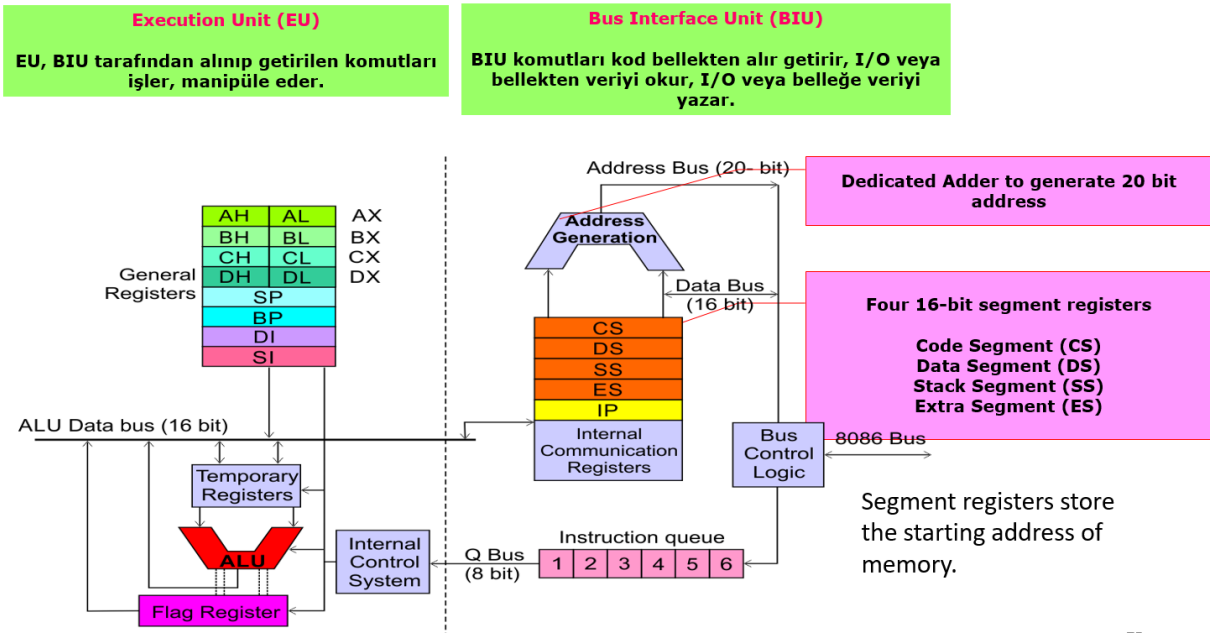
The CPU is divided into three main components:

1. **Registers** – very fast storage locations inside the CPU.
2. **ALU (Arithmetic Logic Unit)** – performs arithmetic and logical operations.
3. **Control Unit (CU)** – orchestrates the execution of instructions and controls data flow.

The CPU executes programs in a repeating sequence known as the **Instruction Cycle**.

CPU Components

Inside CPU (Microprocessor artchitecture):



1. Registers

Registers are **high-speed memory locations inside the CPU** used to store temporary data, instructions, and addresses. Registers = high-speed temporary storage inside the CPU.

Types of Registers:

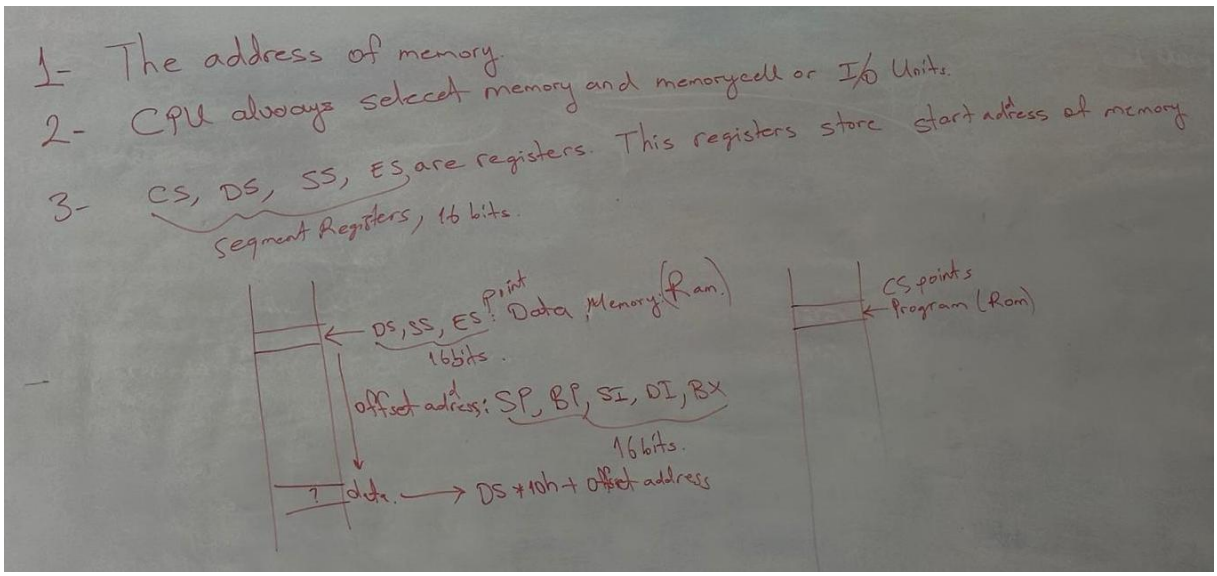
Register	Function
Program Counter (PC)	Holds the address of the next instruction to fetch from memory.
Instruction Register (IR)	Holds the currently executing instruction.
General-Purpose Registers (R0, R1, …)	Temporarily store operands and results for calculations.
Accumulator (AC)	Special register for ALU operations, storing intermediate results.
Memory Address Register (MAR)	Holds memory addresses for reading/writing data.
Memory Data Register (MDR)	Holds data read from or written to memory.
Status Register / Flags	Indicates outcomes of operations (zero, carry, overflow).

- Registers are **faster than cache or RAM**.
- Efficient use of registers **improves program speed**.

Registers are special-purpose, high-speed and temporary storage, Located inside CPU

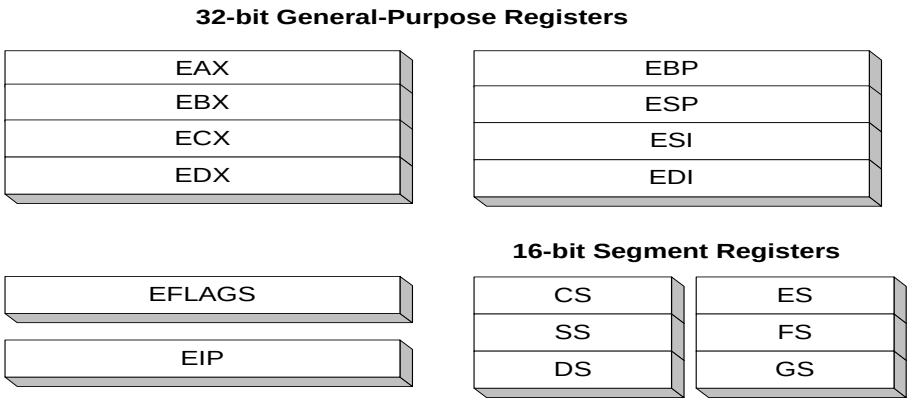
I

	Type	Register width	Name of register
	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
	Pointer register	16 bit	SP, BP
	Index register	16 bit	SI, DI
	Instruction Pointer	16 bit	IP
	Segment register	16 bit	CS, DS, SS, ES
	Flag (PSW)	16 bit	Flag register



Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

Pentinium: 32 Bit General-Purpose Registers:



Arithmetic Logic Unit (ALU)

The **ALU performs all arithmetic (e.g., addition, subtraction) and logical (e.g., AND, OR, NOT, comparisons) operations**. ALU = performs arithmetic and logic operations.

Functions of ALU:

- Arithmetic: ADD, SUB, MUL, DIV
- Logic: AND, OR, XOR, NOT
- Shift operations: SHL, SHR
- Comparisons: ==, <, >

ALU works with registers to fetch operands, perform operations, and store results.

Control Unit (CU)

The **Control Unit directs the operation of the CPU**, telling the ALU, memory, and I/O devices what to do. Control Unit = coordinates the CPU, fetches and decodes instructions, controls execution. **Functions of CU:**

1. **Instruction Fetch:** Get instruction from memory.
2. **Instruction Decode:** Interpret opcode and determine required actions.
3. **Control Signals:** Activate ALU operations, memory read/write, and I/O communication.
4. **Instruction Sequencing:** Maintain the program counter and flow of execution.

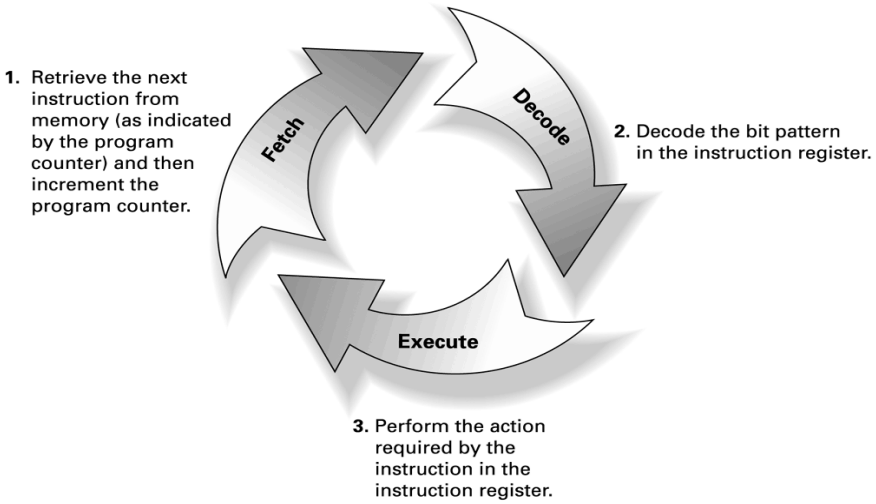
CU is like the **CPU's "traffic controller"**, coordinating data flow and execution.

Instruction Cycle (Fetch-Decode-Execute Cycle)

The **Instruction Cycle** is the repeating process by which a CPU executes a program:

1. **Fetch:** CPU fetches instruction from programme memory at the address in Code segment and **Program Counter (PC)**. Instruction is loaded into **Instruction Pointer Register (IP)**.
2. **Decode:** Control Unit decodes the instruction opcode to determine the operation and operands. Determines which registers or memory locations are involved.
3. **Execute:** ALU performs the operation (arithmetic or logical). Results are stored in registers or memory.
4. **Update Program Counter:** Move to the next instruction (PC incremented).

Instruction Cycle Steps :



1) [Memory] --> fetch
2) fetch --> [IP] --> decode
3) decode --> [CU] --> execute
4) execute --> [ALU/Register]
5) [ALU/Register] --> store result

Modern CPUs often **pipeline** these steps to overlap execution for higher performance. In the pipeline, instruction cyles steps (5 steps) are working at the same time. But different code parts (5 parts) are working at the same time.
Code1-1, Code2-2 Code3-3 Code4-4 Code5-5 these are working at the same time.
Code1-2, Code2-3 Code3-4 Code4-5 Code5-1 these are working at the same time.
Code1-3, Code2-4 Code3-5 Code4-1 Code5-2 these are working at the same time.
….
Instruction Cycle = fetch → decode → execute → update PC. Understanding CPU basics is essential for **software optimization**, **assembly programming**, and **system-level understanding**.

CPU Performance

CPU performance refers to how fast the processor can **execute instructions** and handle **tasks**. Performance depends on several factors:

- Clock speed (GHz)
- Number of cores (CPUs)
- Cache memory size
- Instruction set efficiency
- Memory access speed

Methods to Increase CPU Performance

A. Hardware-Level Methods

1. **Increase Clock Speed**
    - Clock speed is the number of **cycles per second** (GHz).
    - Higher clock → more instructions processed per second.
    - Limitation: Heat generation and power consumption.
2. **Use Multiple Cores (Multi-core CPUs)**
    - Modern CPUs have **2, 4, 8, or more cores**.
    - Each core can process instructions **independently**.
    - Allows **parallel execution** → faster performance for multitasking and multithreaded programs.
3. **Increase Cache Memory**
    - Cache is **fast memory inside CPU**.
    - Stores frequently used instructions/data.
    - Larger cache → fewer memory access delays → faster execution.
4. **Use Pipelining**
    - CPU divides instruction execution into **stages** (fetch, decode, execute).
    - Each stage works **simultaneously** on different instructions.
    - This increases **instruction throughput**.
5. **Hyper-Threading / Simultaneous Multithreading (SMT)**
    - Allows a **single core** to handle **multiple threads** at once.
    - Improves efficiency and utilization of CPU resources.
6. **Superscalar Architecture**
    - CPU can execute **multiple instructions in a single clock cycle**.
    - Uses multiple **execution units** inside the CPU.
7. **Reduce Memory Latency**
    - Use **faster RAM** or **memory closer to CPU** (like L1/L2/L3 cache).
    - Faster memory reduces the **time CPU waits** for data.
8. **Overclocking**
    - Running the CPU **faster than its rated speed**.
    - Improves performance but increases heat and power usage.

B. Software-Level Methods

1. **Efficient Programming**
    - Optimized algorithms use **fewer CPU cycles**.
    - Example: Sorting algorithms, matrix multiplication.
2. **Compiler Optimization**
    - Compilers can generate **optimized machine code** for faster execution.
3. **Load Balancing**
    - Distribute tasks efficiently among CPU cores.
    - Prevents one core from being overworked while others are idle.
4. **Operating System Scheduling**
    - OS can prioritize critical tasks and assign CPU time efficiently.
5. **Reducing Background Processes**
    - Fewer running programs → more CPU resources for main tasks.

---

3. Summary Table

Method	Description	Level
Increase clock speed	Faster cycles per second	Hardware
Multi-core CPUs	Parallel processing	Hardware
Increase cache	Fast memory for frequent data	Hardware
Pipelining	Execute stages simultaneously	Hardware
Hyper-threading	Multiple threads per core	Hardware
Superscalar	Multiple instructions per cycle	Hardware
Faster memory	Reduce data wait time	Hardware
Overclocking	Run CPU above rated speed	Hardware
Efficient algorithms	Fewer CPU cycles	Software
Compiler optimization	Better machine code	Software
Load balancing	Distribute tasks evenly	Software
OS scheduling	Prioritize critical tasks	Software
Reduce background apps	Free CPU resources	Software

Key Insight:

- **CPU performance** depends not just on **hardware**, but also on **how software uses it**.
- Modern performance improvements often combine **multi-core CPUs, pipelining, caching, and efficient software**.

Pipelining

Pipelining is a technique used in CPUs to **increase instruction throughput**.

- The idea is similar to an **assembly line** in a factory:
  - While one worker is **assembling one part**, the next worker **prepares the next part**.
- In a CPU, different **stages of instruction execution** are overlapped, so multiple instructions are **processed at the same time**.

2. How CPU Executes Instructions

A typical CPU instruction goes through **these stages**:

1. **Fetch (IF)** – Get the instruction from memory.
2. **Decode (ID)** – Understand what the instruction wants to do.
3. **Execute (EX)** – Perform the operation (add, subtract, etc.).
4. **Memory Access (MEM)** – Read/write data from/to memory if needed.
5. **Write Back (WB)** – Store the result in a register.

Without pipelining:

- Each instruction completes all stages **one by one**.
- If one instruction takes 5 cycles, and you have 5 instructions, it takes **25 cycles total**.

3. How Pipelining Improves Performance

With pipelining:

- Each stage works on a **different instruction simultaneously**.
- Think of it like a 5-stage assembly line:

Cycle	IF	ID	EX	MEM	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1
6		I5	I4	I3	I2

- After the pipeline is **full**, the CPU can complete **one instruction per cycle**.
- This **increases throughput** without increasing the clock speed.

4. Types of Hazards in Pipelining

Sometimes pipelining can **slow down** due to conflicts:

1. **Data Hazard** – Instruction depends on result of previous instruction.
    - Solution: Forwarding or stalling.
2. **Control Hazard** – Branch instructions (like if) may change the flow.
    - Solution: Branch prediction.
3. **Structural Hazard** – Hardware resource needed by multiple stages.
    - Solution: Duplicate resources or stall.

5. Summary

Feature	Description
Pipelining	Technique to overlap instruction execution stages
Goal	Increase CPU throughput (instructions per cycle)
Stages	Fetch → Decode → Execute → Memory → Write Back
Benefit	More instructions executed simultaneously
Limitation	Hazards (data, control, structural) can reduce efficiency

Key Insight:

- Pipelining **does not make a single instruction faster**, but it allows **more instructions to be completed per unit time**.
- Modern CPUs use **deep pipelines** (10+ stages) for high performance.

8. Memory Organization

Memory is a sequential logic gates with data storage, the data processed or manipulated by the microprocessor. Memories store information such as commands or data consisting of 1 or 0 formats in the binary number system. Memories are used to store data. It is a collection of storage cells with the circuits necessary to transfer commands or data to microprocessors. Transistor is a circuit element produced in semiconductor technology that performs the functions of storaged data, switching or amplifying signals by controlling the flow of electrons (current, voltage) from a different point. Transistors can be produced in atomic structure.

Register is temporary special purpose registers. Register is the most basic storage unit in the microprocessor.

ROM memory is used to store unchanged commands and data. It is a read-only memory.

RAM memory is a memory that is both written and read.

Cache memory is the microprocessor's own cache memory (SRAM) where the data it will process in the next steps is transferred and prepared in advance.

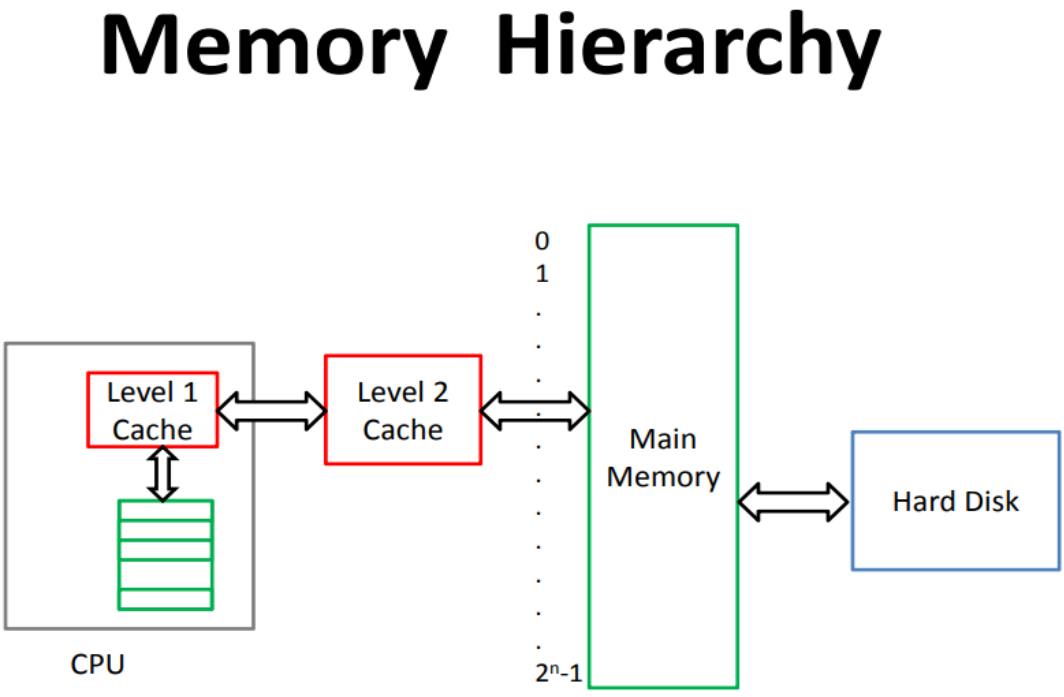
EPROM: erasable programmable read-only memory

Dynamic RAM (DRAM). inexpensive; must be refreshed constantly memory

Static RAM (SRAM): expensive; used for cache memory; no refresh required

Video RAM (VRAM): dual ported; optimized for constant video refresh

CMOS RAM: complimentary metal-oxide semiconductor. system setup information



Memory Organization defines how a computer stores and accesses data and instructions.

- Efficient memory design is crucial because **CPU speed is much higher than main memory**, and programs require **fast access** to instructions and data.
- Memory is organized in a **hierarchy** based on speed, cost, and size.

Types of Memory:

A. RAM (Random Access Memory): Volatile memory used for **temporary storage** while a program is running. Both read and write operations are possible.

Types of RAM:

1. **DRAM (Dynamic RAM):** Needs periodic refreshing; slower but cheaper.
2. **SRAM (Static RAM):** Faster, no refresh required, used in CPU caches.

Characteristics:

- Fast access compared to secondary storage.
- Contents are lost when power is turned off.

Use Case:n Storing currently running programs and data.

B. ROM (Read-Only Memory)

Non-volatile memory that **cannot be modified (or can be modified slowly)**. Stores firmware or permanent instructions needed for booting.

Types of ROM:

1. **PROM:** Programmable once after manufacturing.
2. **EPROM:** Can be erased by UV light and reprogrammed.
3. **EEPROM:** Electrically erasable and reprogrammable (used in BIOS).

Use Case: Bootstrapping the system, firmware storage.

C. Cache Memory

Small, **high-speed memory located close to CPU** to store frequently used instructions and data.

Characteristics:

- Faster than main RAM.
- Reduces CPU waiting time for memory access.
- Usually organized in levels: **L1 (fastest, smallest) → L2 → L3 (larger, slower)**.

Use Case: Storing hot data and instructions for faster CPU access.

High-speed, expensive, static RAM both inside and outside of the CPU.
Level-1 cache: inside the CPU
Level-2 cache: outside the CPU
Cache hit: when data to be read is already in cache memory
Cache miss: when data to be read is not in cache memory.

D. Virtual Memory

Technique that allows a computer to use **disk space as an extension of RAM**, creating the illusion of a large memory space.

Mechanism:

- Operating system divides memory into **pages**.
- Pages not currently in use are stored on disk (page file or swap space).
- When needed, pages are **swapped into RAM**.

Pros:

- Allows execution of programs larger than physical RAM.
- Provides process isolation and protection.

Cons:

- Access is slower than RAM.
- Excessive paging can lead to **thrashing**, reducing performance.

Comparison of Memory Types

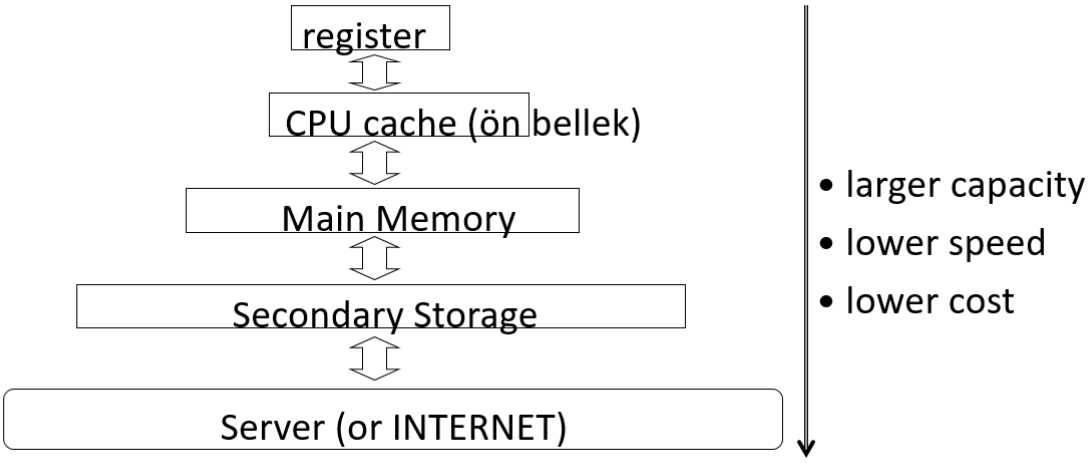
- DRAM
  - very high density → cheap data cache in computers
  - must be periodically refreshed → slower than SRAM
  - volatile; no good for program (long term) storage
- SRAM (basically a Latch)
  - fastest type of memory
  - low density → more expensive
    - generally used in small amounts (L2 cache) or expensive servers
- EEPROM
  - slow/complex to write → not good for fast cache
  - non-volatile; best choice for program memory
- ROM
  - hardware coded data; rarely used except for bootup code
- Register (flip flop)
  - functionally similar to SRAM but less dense (and thus more expensive)
  - reserved for data manipulation applications

3. Memory Hierarchy and Speed

Memory is organized in a **hierarchy** based on **speed, size, and cost**:

Level	Type	Speed	Size	Cost per bit	Location
1	Registers	Fastest	Very small	Very high	Inside CPU
2	Cache (L1, L2, L3)	Very fast	Small	High	Inside/near CPU
3	Main Memory (RAM)	Fast	Moderate	Moderate	Mainboard
4	Secondary Storage (HDD, SSD)	Slow	Large	Low	Disk
5	Tertiary Storage (Tape, Cloud)	Slowest	Very large	Very low	External

Level Type **Speed** **Size** **Cost per bit Location**



9. I/O Techniques

There are three main methods for CPU to communicate with I/O devices:

A. Polling (Programmed I/O)

Definition:

The CPU repeatedly **checks (polls) the status** of an I/O device to see if it is ready for data transfer.

How It Works:

1. CPU continuously reads a **status register** of the device.
2. When the device is ready, CPU reads/writes data.

Pros:

- Simple to implement.
- Works well for **fast devices or low-frequency I/O**.

Cons:

- Wastes CPU cycles while waiting for the device.
- Not efficient for slow devices like disks.

Example:

```
while (device_status != READY) {
   // keep checking
}
data = read_device();
```

B. Interrupt-Driven I/O

The I/O device **sends an interrupt signal** to the CPU when it is ready for data transfer, instead of waiting for the CPU to check.

How It Works:

1. CPU executes other tasks.
2. When I/O device is ready, it sends an **interrupt request (IRQ)**.
3. CPU **pauses current execution**, jumps to an **interrupt service routine (ISR)**, handles the I/O, then resumes.

Pros:

- More efficient than polling.
- CPU is free to perform other tasks while waiting for I/O.

Cons:

- Slightly more complex hardware and software support required.
- Interrupts need proper handling to avoid conflicts or missed requests.

Example Scenario:

- Keyboard: sends an interrupt when a key is pressed → CPU reads the character.

C. Direct Memory Access (DMA)

DMA allows I/O devices to **transfer data directly to/from memory without CPU intervention**.

How It Works:

1. CPU sets up the DMA controller with:
    - Source/Destination address
    - Transfer size
    - Direction (read/write)
2. DMA controller handles the transfer **independently**.
3. CPU is notified via **interrupt** once the transfer completes.

Pros:

- Very efficient for **large data transfers** (e.g., disk to memory).
- CPU can execute other tasks during the transfer.

Cons:

- More complex hardware (DMA controller).
- Care must be taken to avoid memory access conflicts.

Example:

- Reading a file from disk into memory for a program → DMA moves the data while CPU continues executing instructions.

---

3. Comparison of I/O Techniques

Feature	Polling	Interrupt	DMA
CPU Involvement	High (waits actively)	Medium (CPU interrupted when needed)	Low (CPU mostly free)
Complexity	Simple	Medium	High (requires DMA controller)
Efficiency	Low for slow devices	Better	High (best for large transfers)
Use Case	Simple devices, low-speed I/O	Keyboard, mouse, network cards	Disk transfers, graphics buffers, audio streaming

4. Summary

- **Polling:** CPU actively checks device status → wastes cycles.
- **Interrupts:** Device signals CPU when ready → CPU free for other tasks.
- **DMA:** Device transfers data directly to memory → CPU minimally involved → best for large transfers.

Key Takeaway for Software Engineers: Understanding I/O techniques is essential for **writing efficient programs**, **real-time systems**, and **embedded software**.

- Modern operating systems combine interrupts and DMA to maximize CPU efficiency while handling multiple I/O devices.

10. Instruction Set

Definition:

The **Instruction Set** of a microprocessor (or CPU) is the **collection of machine-level instructions** that the CPU can execute. It forms the interface between **software and hardware**.

Purpose:

- Allows software (programs) to communicate with the CPU.
- Determines the **functionality** a processor can perform.

Categories of Instructions:

1. Data Transfer Instructions
2. Arithmetic Instructions
3. Logical Instructions
4. Control Instructions

---

2. Types of Instructions

A. Data Transfer Instructions

Purpose: Move data between **registers, memory, and I/O devices**.

Examples:

Instruction	Function
MOV A, B	Copy data from B to A
LOAD R1, 0x2000	Load value from memory address 0x2000 into register R1
STORE R2, 0x3000	Store data from register R2 to memory address 0x3000
IN R1, PORT1	Read data from I/O port into register
OUT PORT2, R2	Write data from register to I/O port

Notes:

- Do **not perform arithmetic or logic**, just move data.
- Can transfer between memory, registers, and I/O ports.

B. Arithmetic Instructions

Purpose: Perform **mathematical operations** on data.

Examples:

Instruction	Function
ADD R1, R2	Add contents of R2 to R1
SUB R1, R2	Subtract contents of R2 from R1
MUL R1, R2	Multiply R1 by R2
DIV R1, R2	Divide R1 by R2
INC R1	Increment R1 by 1
DEC R2	Decrement R2 by 1

Notes:

- Works on registers, memory locations, or immediate values (depending on addressing mode).

C. Logical Instructions

Purpose: Perform **bitwise and logical operations**.

Examples:

Instruction	Function
AND R1, R2	Bitwise AND of R1 and R2
OR R1, R2	Bitwise OR
XOR R1, R2	Bitwise XOR
NOT R1	Bitwise complement of R1
SHL R1	Shift bits left
SHR R1	Shift bits right

Notes:

- Useful in decision-making, masking, and manipulating individual bits.

D. Control Instructions

Purpose: Control **program execution flow**.

Examples:

Instruction	Function
JMP ADDRESS	Unconditional jump to address
JZ ADDRESS	Jump if zero flag is set
JNZ ADDRESS	Jump if zero flag is not set
CALL ADDRESS	Call subroutine at address
RET	Return from subroutine
NOP	No operation (used for timing or alignment)
HLT	Halt CPU execution

Notes:

- Alters sequence of instruction execution.
- Critical for loops, conditions, subroutines, and program termination.

---

3. Addressing Modes

Definition:

Addressing modes specify **how the operand of an instruction is accessed**.

Common Addressing Modes:

Mode	Description	Example
Immediate	Operand is part of the instruction itself	MOV R1, #5 (load 5 into R1)
Register	Operand is in a CPU register	ADD R1, R2
Direct / Absolute	Operand is in a specific memory address	LOAD R1, 0x2000
Indirect	Operand's address is in a register	LOAD R1, (R2)
Indexed	Operand address = base address + offset	LOAD R1, 0x2000 + R2
Relative	Operand address is relative to PC	JMP +10 (jump 10 instructions ahead)
Implicit	Operand is implied by instruction	CLC (clear carry flag)

Key Points:

- Addressing modes provide **flexibility in programming**.
- Influence **instruction size, execution speed, and memory access**.

---

4. Summary

1. **Instruction Set:** The set of commands a CPU can execute.
2. **Types of Instructions:**
    - **Data Transfer:** Move data (e.g., MOV, LOAD, STORE)
    - **Arithmetic:** Mathematical operations (e.g., ADD, SUB)
    - **Logical:** Bitwise operations (e.g., AND, OR, SHL)
    - **Control:** Change program flow (e.g., JMP, CALL, RET)
3. **Addressing Modes:** Define **how operands are specified**: immediate, register, direct, indirect, indexed, relative, or implicit.

Importance for Software Engineers:

- Understanding instruction sets helps in **assembly programming**, **debugging**, and **performance optimization**.
- Knowledge of addressing modes enables **efficient memory access and code design**.

11. Assembly Language

Assembly Language is a **low-level programming language** that provides a **human-readable representation of machine instructions**.

Key Points:

- Each assembly instruction corresponds to **one machine code instruction**.
- Programs are **closely tied to the CPU's instruction set**.
- Requires **assembler software** to translate assembly code into machine code.

---

2. Assembler Directives

Definition:

Assembler directives (also called pseudo-instructions) are **instructions to the assembler** that **do not generate machine code**, but control **program assembly and memory allocation**.

Common Directives:

Directive	Purpose	Example
ORG	Specify starting memory address	ORG 1000h
DB	Define a byte of data	DB 0x5A
DW	Define a word (2 bytes)	DW 1234h
EQU	Define a constant	PI EQU 3.14
END	Mark end of program	END
INCLUDE	Include another file	INCLUDE macros.asm

Notes:

- Directives help **organize program structure** and **reserve memory for variables and constants**.
- They are **CPU-independent**, interpreted by the assembler.

---

3. Labels

Definition:

Labels are **identifiers used to name memory addresses** or positions in code, usually for **branching or referencing data**.

Syntax:

LabelName: Instruction/Directive

Examples:

```
START: MOV AX, 5      ; Start of program
LOOP:  ADD AX, 1      ; Loop label
    CMP AX, 10
    JNE LOOP       ; Jump back if AX != 10
```

Notes:

- Labels improve **readability** and **program flow control**.
- Commonly used with **jumps, loops, and data storage**.

4. Comments

Definition:

Comments are **notes in the program** ignored by the assembler, used for **documentation and readability**.

Syntax:

- In many assemblers (like MASM/TASM), comments start with ;

```
MOV AX, 5      ; Load 5 into AX register
```

Notes:

- Helps **future programmers** understand the code.
- Essential for **maintaining complex assembly programs**.

---

5. Data Storage and Constants

A. Variables / Data Storage

Assembly programs use **directives** to allocate memory for variables:

Directive	Purpose	Example
DB	Define byte-sized variable	COUNT DB 10
DW	Define word-sized variable (2 bytes)	NUM DW 1000h
DD	Define double word (4 bytes)	VALUE DD 12345678h

Notes:

- Data can be **initialized** at declaration or left uninitialized.
- Memory addresses can be referenced using labels.

B. Constants

Definition:

Constants are **fixed values that do not change** during program execution.

Syntax Examples:

```
PI EQU 3.14      ; Define a constant
MAX EQU 255
```

Notes:

- Use constants for **clarity and maintainability**.
- Unlike variables, constants **do not occupy memory at runtime**, they are replaced during assembly.

6. Example: Simple Assembly Program

```
ORG 1000h          ; Start address

; Constants
TEN EQU 10

; Variables
COUNT DB 0         ; Initialize COUNT to 0

START:
   MOV AL, COUNT   ; Load COUNT into AL register
   ADD AL, TEN     ; Add constant TEN
   MOV COUNT, AL   ; Store back to COUNT

END START          ; End of program
```

Explanation:

1. ORG 1000h → program starts at memory address 1000h
2. TEN EQU 10 → define a constant
3. COUNT DB 0 → allocate a byte for variable COUNT
4. MOV, ADD → instructions to perform addition
5. END START → marks program end

---

7. Key Takeaways for Software Engineers

1. **Assembler Directives:** Control assembly and memory, do not generate machine instructions.
2. **Labels:** Name instructions/data for easy reference and branching.
3. **Comments:** Improve readability and maintainability.
4. **Data Storage & Constants:** Use DB, DW, DD, and EQU to define variables and constants.
5. Understanding these basics is **essential for low-level programming, embedded systems, and performance-critical applications**.

12. Program Control Instructions

Program Control Instructions are used to **alter the normal sequential flow of program execution**.

- They allow **loops, conditional execution, subroutines, and function calls**.
- They are critical for creating **dynamic and reusable programs**.

Key Components:

- **Jump instructions** → unconditional or conditional changes in execution
- **Loop instructions** → repeat code blocks
- **Subroutines / Functions** → modular code execution
- **Stack operations** → manage temporary data, subroutine calls, and returns

---

2. Jump Instructions

Definition:

Jump instructions transfer program control to another memory address.

Types of Jumps:

Instruction	Function	Example
JMP	Unconditional jump	JMP START → always jump to START
JE / JZ	Jump if equal / zero flag set	JE LABEL → jump if last result = 0
JNE / JNZ	Jump if not equal / zero flag not set	JNE LABEL → jump if last result ≠ 0
JG / JNLE	Jump if greater	JG LABEL → jump if last result > 0
JL / JNGE	Jump if less	JL LABEL → jump if last result < 0

Use Case:

- Conditional execution, decision-making, branching.

---

3. Loop Instructions

Definition:

Loop instructions **repeat a block of code** a specified number of times.

Example Using Registers:

```
MOV CX, 5      ; Loop counter = 5
LOOP_START:
   ; Code to repeat
   DEC CX      ; Decrement counter
   JNZ LOOP_START  ; Jump back if CX ≠ 0
```

Key Points:

- Loop counter (like CX in x86) is decremented each iteration.
- JNZ / JNE is used to continue looping.

4. Subroutine / Call and Return

Definition:

A **subroutine** (function or procedure) is a **reusable block of code** that can be executed from multiple locations in a program.

Instructions:

Instruction	Function	Example
CALL	Call subroutine; saves return address	CALL PRINT_MSG
RET	Return from subroutine to saved address	RET

Example:

```
MAIN:
  CALL PRINT_MSG
  JMP END

PRINT_MSG:
  ; Code to print message
  RET

END:
```

Key Points:

- CPU **pushes the return address onto the stack** during CALL.
- RET pops the address and continues execution.
- Enables **modular programming** in assembly.

5. Stack Operations

Definition:

The **stack** is a LIFO (Last-In-First-Out) memory structure used for **temporary storage, subroutine management, and interrupt handling**.

Basic Instructions:

Instruction	Function	Example
PUSH	Push data onto stack	PUSH AX
POP	Pop data from stack	POP AX
CALL	Push return address, jump to subroutine	CALL FUNC
RET	Pop return address to resume execution	RET

Example of Stack Use in Subroutine:

```
MOV AX, 5
PUSH AX      ; Save AX on stack before subroutine
CALL FUNC
POP AX       ; Restore AX after subroutine

FUNC:
  ; Modify AX temporarily
  RET
```

Notes:

- The **stack pointer (SP)** keeps track of the top of the stack.
- Stacks are essential for **nested subroutine calls** and **local variable management**.

6. Summary

1. **Jump Instructions:** Control flow → conditional/unconditional branches.
2. **Loop Instructions:** Repeat code blocks efficiently.
3. **Subroutine Call/Return:** Modular code execution; use CALL and RET.
4. **Stack Operations:** Manage temporary data and return addresses (PUSH/POP).
5. **Software Engineering Insight:** Proper use of control instructions and stacks is critical for **efficient, modular, and maintainable assembly programs**.

13. Interfacing and I/O in Assembly

I/O Interfacing in assembly language refers to how the CPU **communicates with external devices** such as keyboards, displays, sensors, and timers.

Key Concepts:

- **Ports:** Specific addresses for I/O devices.
- **Interrupts:** Signals from devices to CPU to handle asynchronous events.
- **Timers:** Hardware counters used to measure or control time.

Understanding these concepts is crucial for **embedded programming, real-time systems, and low-level device control**.

---

2. I/O Ports

Definition:

- Ports are **special memory or I/O addresses** used to communicate with external devices.
- CPU can **read from** or **write to** these ports to exchange data.

Types of I/O:

1. **Memory-mapped I/O:**
    - I/O devices share the same address space as memory.
    - Accessed using **normal memory instructions** (MOV AX, [PORT_ADDR]).
2. **Port-mapped I/O (Isolated I/O):**
    - Separate address space for I/O devices.
    - Accessed using **special instructions**: IN and OUT.

Example:

```
IN AL, 0x60    ; Read data from keyboard port 0x60 into AL
OUT 0x61, AL   ; Send data from AL to output port 0x61
```

Notes:

- Ports are assigned to specific devices (keyboard, display, sensors).
- IN reads, OUT writes.

3. Interrupts

Definition:

- An **interrupt** is a signal to the CPU that **an event needs immediate attention**.
- Allows CPU to **pause current execution**, service the device, then resume.

Types of Interrupts:

1. **Hardware Interrupts:**
    - Generated by external devices (keyboard, mouse, timers).
    - Example: Pressing a key triggers a keyboard interrupt.
2. **Software Interrupts:**
    - Generated by programs using INT instruction.
    - Example: INT 21h in DOS calls operating system services.

Interrupt Handling in Assembly:

1. CPU **saves current execution state** on stack.
2. CPU jumps to **Interrupt Service Routine (ISR)**.
3. ISR handles the event (reads data, sends signal).
4. CPU executes IRET to **return from interrupt**.

Example:

```
MOV AH, 0      ; Function: Read character from keyboard
INT 16h        ; BIOS keyboard interrupt
MOV AL, AH     ; Character read into AL
```

Notes:

- Interrupts improve **CPU efficiency** by eliminating busy-wait loops (polling).
- Must be used carefully to avoid **nested or lost interrupts**.

4. Timers

Definition:

- Timers are **hardware counters** used for measuring or controlling time intervals.

Uses in Assembly:

1. **Delays:** Wait for a specific time period.
2. **Event scheduling:** Trigger periodic actions.
3. **Generating PWM or clock signals** in embedded systems.

Programming Timers:

- Timers usually work with **interrupts**.
- CPU sets **timer registers**, and when the timer expires, it triggers an **interrupt**.

Example (pseudo-assembly):

```
MOV AX, TIMER_MODE     ; Set timer mode
OUT TIMER_CTRL_PORT, AX
; Wait for timer interrupt to perform task
```

Notes: Timers allow **precise time-dependent operations** without CPU busy-waiting.

5. Summary

Concept	Purpose	Example Instructions
Ports	Communicate with I/O devices	IN, OUT
Interrupts	Handle asynchronous events efficiently	INT, IRET
Timers	Measure or control time intervals	OUT timer port, timer ISR

Key Takeaways for Software Engineers:

1. **Ports** let the CPU read/write to external devices.
2. **Interrupts** improve efficiency by avoiding constant polling.
3. **Timers** are essential for real-time and periodic tasks.
4. Combined, these allow **efficient, precise, and responsive I/O programming** in assembly.

14. Microprocessor Interfacing with Memory and Peripherals

A. Overview

Interfacing is how a microprocessor communicates with **memory units and peripheral devices**.

- The goal is to allow **data and instructions to flow efficiently** between CPU, memory, and I/O devices.
- Interfaces handle **addressing, control signals, and timing synchronization**.

Key Components in Interfacing:

1. **Address bus:** CPU sends memory or I/O addresses.
2. **Data bus:** CPU reads/writes data.
3. **Control lines:** Indicate operation type (Read/Write), enable signals, or chip selection.

B. Memory Interfacing

Objective: Connect memory chips to the CPU so it can fetch instructions and read/write data.

Basic Signals for Memory:

- CS (Chip Select): Activates a specific memory chip.
- RD (Read): Read operation from memory to CPU.
- WR (Write): Write operation from CPU to memory.

Steps for Memory Read:

1. CPU places **memory address** on the address bus.
2. CPU activates **RD** signal.
3. Selected memory sends **data** on the data bus.
4. CPU reads the data into a register.

Steps for Memory Write:

1. CPU places **address** and **data** on respective buses.
2. CPU activates **WR** signal.
3. Memory stores the data at the specified address.

Diagram (Simplified Memory Interface):

CPU <--> Address Bus <--> Memory Address

CPU <--> Data Bus <--> Memory Data

CPU Control Signals --> Memory (CS, RD, WR)

C. Peripheral Interfacing

Objective: Connect I/O devices (keyboard, display, sensors) to the CPU.

I/O Techniques:

1. **Memory-Mapped I/O:**
    - I/O devices share the same address space as memory.
    - Accessed via **regular memory instructions** (MOV).
2. **Port-Mapped I/O (Isolated I/O):**
    - Separate I/O address space.
    - Accessed using IN and OUT instructions.

Peripheral Control Signals:

- CS or OE (Output Enable)
- RD / WR for read/write operations
- Sometimes interrupts for asynchronous handling

Example of Peripheral Read in Assembly:

```
IN AL, 0x60  ; Read data from keyboard port
```

---

2. Simple ALU Design

A. Overview

The **Arithmetic Logic Unit (ALU)** performs **arithmetic and logical operations** in the CPU.

- It is the **core computation unit**.
- Supports addition, subtraction, AND, OR, NOT, and comparisons.

B. Components of a Simple ALU

1. **Inputs:**
    - Two operands (A, B) from registers.
    - Control lines specifying the operation.
2. **Operation Units:**
    - **Arithmetic Unit:** Addition, subtraction, increment, decrement.
    - **Logic Unit:** AND, OR, XOR, NOT.
    - **Shifter:** Shift left/right (optional).
3. **Output:**
    - Result stored in accumulator or output register.
    - Status flags set: Zero, Carry, Overflow, Sign.

C. Example: 4-Bit ALU Design

Input	Operation	Output
A = 0110, B = 0011	ADD	1001 (Sum), Carry = 0
A = 0110, B = 0011	AND	0010
A = 0110, B = 0011	OR	0111

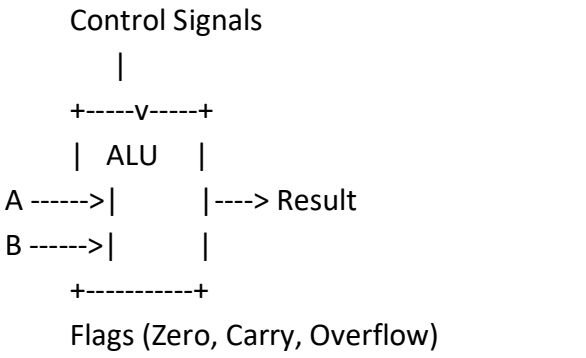
Control Signals:

- 000 → ADD
- 001 → SUB
- 010 → AND
- 011 → OR
- 100 → NOT A

ALU Logic (Simplified):

- Inputs go to **arithmetic unit** or **logic unit** depending on control signals.
- Output goes to **result register**.
- Flags updated according to result.

Diagram (Simplified ALU):



---

3. Key Takeaways

1. **Interfacing:** CPU communicates with memory and peripherals via **address, data, and control signals**.
2. **Memory interface:** CPU uses CS, RD, WR signals for data transfer.
3. **Peripheral interface:** Can be **memory-mapped** or **port-mapped**, optionally using **interrupts**.
4. **ALU design:** Core unit performing arithmetic and logic; controlled by control signals and outputs flags.
5. Understanding these advanced topics is essential for **hardware-software integration**, **embedded systems**, and **CPU design understanding**.

15. GPU and TPU Architectures

1. Introduction

In traditional computer organization courses, the Central Processing Unit (CPU) is introduced as the primary processing element of a computer system. However, modern computing systems increasingly rely on specialized processors such as:

- GPUs (Graphics Processing Units)
- TPUs (Tensor Processing Units)
- AI accelerators

These components have become fundamental in high-performance computing, data science, and artificial intelligence applications.

This lecture focuses on:

- Architectural differences between CPU, GPU, and TPU
- Why GPUs and TPUs are critical for Artificial Intelligence
- Internal organization of these processors
- Their role in modern computing systems

2. Computational Requirements of Artificial Intelligence

Modern Artificial Intelligence (AI), especially Deep Learning, is heavily based on:

- Matrix operations
- Vector computations
- Linear algebra
- Tensor processing

The core computation in most neural networks can be represented as:

Y=W·X+bY = W \cdot X + bY=W·X+b

Where:

- XXX = input vector
- WWW = weight matrix
- bbb = bias vector
- YYY = output vector

Training a neural network involves performing this computation:

- Millions or billions of times
- Over very large datasets
- Using iterative optimization algorithms

This leads to extremely high computational demands.

3. CPU Architecture: Strengths and Limitations

Traditional CPU Design Goals

CPUs are designed to:

- Execute sequential instructions efficiently
- Handle complex control logic
- Perform general-purpose computation

Typical CPU Characteristics

Feature	Description
Number of cores	4 – 16 (typically)
Optimization	Low latency
Execution model	Sequential / limited parallelism
Best for	Control-intensive tasks

Limitation for AI

Although CPUs are powerful, they are not well suited for AI workloads because:

- AI requires massive parallelism
- CPUs have limited number of cores
- Matrix operations are inefficient on CPUs

4. GPU: Graphics Processing Unit

4.1 What is a GPU?

Originally designed for graphics rendering, GPUs have evolved into general-purpose parallel processors.

Serial computing is an approach to solving a problem step-by-step and sequentially on a single processor. In this model, operations are executed sequentially in a specific logical order; the next operation is not started until the previous one is completed. The program flow is linear, and all calculations are performed by a single processing unit. While serial computing, which has been the fundamental mode of operation for traditional computer architectures for many years, is sufficient for small-scale and low-complexity problems, it shows performance limitations in large datasets and computationally intensive applications.

Artificial intelligence is a "computational-intensive" discipline. Today, AI, especially deep learning, requires billions of parameters, trillions of operations (FLOPs - Floating Point Operations), and long-term, parallel computation. Parallel computing is an approach that solves a large and complex problem simultaneously using multiple processors or cores instead of a single processor. In this method, the work is divided into independent or partially dependent sub-parts, and these parts are executed simultaneously on different processing units. This significantly reduces computation time, allows for faster processing of larger datasets, and improves system performance. Thanks to multi-core processors, GPUs,

and distributed systems, parallel computing has become a fundamental necessity in fields such as artificial intelligence, scientific simulations, and big data analysis.

A GPU is optimized to perform the same operation on many data elements simultaneously. This paradigm is called:
SIMT – Single Instruction, Multiple Threads

4.2 GPU vs CPU

Feature	CPU	GPU
Core count	Low	Very high
Parallelism	Limited	Massive
Clock speed	High	Moderate
Throughput	Low	Very high
Best for	Sequential logic	Data-parallel tasks

4.3 GPU Architecture

A modern GPU consists of:

- Thousands of simple processing cores
- Streaming Multiprocessors (SMs)
- High-bandwidth memory
- Specialized units for parallel arithmetic

Simplified GPU Organization

GPU
├── Streaming Multiprocessor 1
├── Streaming Multiprocessor 2
├── ...
├── Global Memory
└── Memory Controllers

4.4 Programming Model

GPU programming is commonly performed using:

- CUDA (NVIDIA)
- OpenCL
- DirectCompute

Execution Model

- A program is divided into thousands of threads
- Threads execute in parallel
- Each thread performs simple arithmetic

4.5 Why GPUs are Perfect for AI

Neural networks rely heavily on:

- Matrix multiplications
- Vector additions
- Convolutions

These operations are:

- Highly parallel
- Repetitive
- Identical across large datasets

GPUs excel at exactly this type of workload.

5. TPU: Tensor Processing Unit

5.1 What is a TPU?

A TPU is a specialized processor designed by Google specifically for machine learning workloads.

Unlike GPUs, which are general-purpose accelerators, TPUs are:

Application-Specific Integrated Circuits (ASICs) optimized for AI.

5.2 Design Philosophy of TPU

TPUs are designed around a simple idea:

Optimize hardware exclusively for tensor operations

Main goals:

- Extremely fast matrix multiplication
- Low power consumption
- High throughput

5.3 Core Component: Matrix Multiply Unit (MXU)

The heart of a TPU is the:

Matrix Multiply Unit

Example configuration:

- 128 × 128 matrix multiplication
- Performed in a single clock cycle

This makes TPUs extremely efficient for deep learning.

5.4 TPU Architecture Overview

Typical TPU components:

- Matrix Multiply Unit (MXU)
- High-bandwidth on-chip memory
- Vector processing units
- Specialized instruction set for ML

6. GPU vs TPU Comparison

Criterion	GPU	TPU
Purpose	General-purpose	AI-specific
Flexibility	High	Limited
Energy efficiency	Moderate	Very high
Best use case	Research and development	Large-scale AI training
Programming	CUDA, OpenCL	TensorFlow / specialized APIs

7. Memory Architecture Considerations

7.1 Importance of Memory Bandwidth

In AI workloads:

- Computation is not the only bottleneck
- Memory access speed is often more critical

Both GPUs and TPUs are designed with:

- Very high memory bandwidth
- Specialized memory hierarchies

7.2 Typical GPU Memory Hierarchy

- Registers
- Shared memory
- L1 / L2 cache
- Global memory (GDDR / HBM)

Efficient AI computation requires careful use of this hierarchy.

8. Role in Modern AI Systems

Large AI models such as:

- ChatGPT
- Gemini
- LLaMA
- Claude

are trained using:

- Thousands of GPUs
- Large TPU clusters

Training such models on CPUs would be practically impossible.

9. Practical Example

Consider training a neural network with:

- 1 billion parameters
- 1 trillion operations per epoch

Approximate training time:

Hardware	Estimated Time
Single CPU	Months
High-end GPU	Days
TPU Cluster	Hours

This illustrates why specialized processors are essential.

10. Conclusion

Key Takeaways

- Modern AI requires massive parallel computation
- CPUs are insufficient for large-scale AI
- GPUs provide general-purpose parallelism
- TPUs provide AI-specific acceleration

From a computer organization perspective:

- GPUs and TPUs represent a new era of specialized microprocessors
- They extend traditional von Neumann architecture
- They demonstrate the importance of domain-specific hardware

Discussion Questions for Students

1. Why are CPUs inefficient for neural network training?
2. What architectural features make GPUs suitable for AI?
3. How does TPU architecture differ fundamentally from GPU architecture?
4. What are the trade-offs between flexibility and performance?
5. How does memory bandwidth affect AI performance?

Suggested Reading

- Hennessy & Patterson – *Computer Architecture: A Quantitative Approach*
- NVIDIA CUDA Programming Guide
- Google TPU Architecture White Papers

2. Artificial Intelligence Data Centers

In artificial intelligence, data centers are of critical importance because they are not merely "infrastructure," but a strategic, data-driven element. This can be clearly explained across several layers.

Computational Power:

Artificial intelligence is a "compute-intensive" discipline. Today, AI—especially in deep learning—requires billions of parameters, trillions of operations (FLOPs – Floating Point Operations), and long-term, parallel computation. Parallel computing is an approach in which a large and complex problem is solved simultaneously using multiple processors or cores instead of a single processor. In this method, the task is divided into independent or partially dependent subcomponents, and these components are executed concurrently on different processing units. As a result, computation time is significantly reduced, larger datasets can be processed more rapidly, and overall system performance is improved. With the availability of multi-core processors, GPUs, and distributed systems, parallel computing has become a fundamental necessity in fields such as artificial intelligence, scientific simulations, and big data analytics.

Data centers enable specialized hardware such as GPUs and TPUs to operate in a highly parallel, uninterrupted, and synchronized manner. Achieving this scale of computation is physically impossible with desktop systems or fragmented infrastructures.

GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units) are specialized hardware developed specifically for applications requiring intensive computation and parallel processing, such as artificial intelligence. Thanks to their architectures consisting of numerous cores, GPUs can perform thousands of operations simultaneously and are therefore widely used in image processing, scientific computing, and deep learning training. TPUs, on the other hand, are hardware accelerators designed by Google specifically to speed up machine learning algorithms and are optimized for matrix and tensor operations. Both architectures provide significantly higher processing power compared to traditional CPUs, enabling the rapid processing of large datasets and forming the core infrastructure of modern AI systems.

Data Proximity:

Moving data is more costly than computing it. Data movement is more expensive than computation—this is one of the fundamental realities that determine performance in AI. Data centers enable data to be processed locally, providing low latency and high bandwidth. This is particularly critical for large language models, real-time inference, and video or sensor data processing.

Transferring data is often more costly than performing computations because, in modern computing systems, processor speeds are far higher than memory and data transmission speeds. While a processor can perform billions of operations per second, transferring large datasets from memory to the processor or across a network to another system causes delays

and increased energy consumption. Bandwidth limitations, cache latencies, and the overhead of communication protocols degrade performance during data movement. Therefore, the primary objective in high-performance systems today is to process data as close as possible to where it resides, minimizing unnecessary data movement and maximizing computational efficiency.

Energy and Cooling:
Artificial intelligence is subject to the laws of physics. AI is fundamentally a thermal problem. High energy consumption generates substantial heat. Heat reduces performance and can cause hardware damage. For this reason, AI systems cannot operate without specialized cooling systems (such as liquid cooling and immersion cooling), energy continuity, and efficient power distribution in data centers.

AI is considered a “thermal problem” because modern AI models—especially deep learning algorithms—require enormous computational power and therefore consume large amounts of energy. During the training of models with billions of parameters, processors and accelerators such as GPUs and TPUs operate intensively, producing significant heat. If this heat cannot be effectively dissipated, hardware performance declines, energy efficiency decreases, and system scalability becomes more difficult. Consequently, in the design of AI systems today, not only algorithmic accuracy but also engineering challenges such as power consumption, cooling infrastructure, and thermal management have become critical factors.

Scalability: Experiment – Model – Product Chain
AI development requires experimentation, rapid iteration, and large-scale trials. Therefore, data centers must support horizontal and vertical scaling, resource pooling, and the concurrent execution of multiple experiments. Without these capabilities, model development slows down and competitive innovation is lost.

AI development processes inherently involve testing numerous hypotheses, continuously retraining models, and rapidly comparing results, which creates the need for large-scale computation and flexible infrastructure. In data centers, horizontal and vertical scaling, resource pooling, and the ability to run multiple experiments simultaneously allow researchers to test different model architectures and parameter configurations in parallel. Without such infrastructure, each experiment takes longer, resources are used inefficiently, and development cycles slow down. As a result, organizations that cannot iterate quickly fall behind in producing better models, lose innovation capacity, and forfeit competitive advantage.

Horizontal scaling in data centers refers to expanding capacity by adding more servers or processing units to handle increased workloads—essentially running many similar machines in parallel. Vertical scaling, on the other hand, involves increasing the hardware resources of

an existing server, such as processor power, memory, or storage capacity. While horizontal scaling generally offers greater flexibility and fault tolerance, vertical scaling provides simpler management but quickly reaches hardware limits. Modern data centers, particularly in AI and big data applications, employ both approaches together to achieve high performance and continuity.

Security and Sovereignty: Data = Power

In AI, training data, model weights, and inference results are strategic assets. Data centers are critical in terms of data sovereignty, regulatory compliance (such as KVKK and GDPR), and military or industrial security. Therefore, possessing AI capability effectively means possessing data centers.

Artificial intelligence has now become a critical infrastructure. Just as ports, power plants, and telecommunications networks are vital today, AI data centers hold the same importance. For this reason, the United States, China, and the European Union are developing their own AI data center strategies. GPU supply has become a geopolitical issue. Energy and AI now form the basis of new industrial policy.

GPU supply has turned into a geopolitical matter because these hardware components, which form the backbone of AI and high-performance computing infrastructure, directly determine the technological competitiveness of nations. Since advanced GPU and chip production is concentrated in a limited number of countries and companies, supply chains have gained strategic importance; export restrictions and technology embargoes have become instruments of international politics. At the same time, because AI systems consume vast amounts of energy, energy infrastructure, data centers, and renewable resources have become integral parts of national security and economic development plans. Consequently, energy and AI are now viewed not merely as technical issues but as strategic domains shaping industrial policies and global power balances.

Data centers are not the “brain” of artificial intelligence; they are the physical body that enables it to function. No matter how advanced the model, algorithm, or data may be, without computation, energy, cooling, and scalability, AI remains only an idea.
Although AI models consist of abstract algorithms, their real-world operation depends on massive physical infrastructures. Processing large datasets, training models, and continuously delivering services require high computational power, extensive storage capacity, uninterrupted energy, and effective cooling systems. Data centers provide these resources in a scalable manner and constitute the fundamental platform that makes the practical implementation of AI possible. Without the necessary hardware and infrastructure, even the best algorithms cannot be realized; AI would remain merely a theoretical design rather than a real technology.

Major Technology Companies (Hyperscalers)

Large technology companies operate massive data centers for their own services while continuously building new AI infrastructure:

1. **Microsoft**
    - Operates hundreds of data centers through the Azure cloud network.
    - Allocates enormous budgets to AI-focused infrastructure investments (at the level of billions of dollars) and plans expansions in new regions.
2. **Amazon Web Services (AWS)**
    - Plans hundreds of data centers and numerous new regions globally.
    - Builds specialized hardware and new facilities to support AI workloads.
3. **Google (Alphabet)**
    - Expands its data center network for Google Cloud.
    - Invests in new AI-oriented infrastructure in Europe, the U.S., and other regions.
4. **Meta Platforms**
    - Operates numerous data centers for Facebook, Instagram, and AI services, and continues investing in new facilities (e.g., the large data center in Texas).

Infrastructure and Operator Companies

These firms directly operate, lease, or build data centers:

5. **CoreWeave**
    - A rapidly growing provider specializing in GPU-intensive data centers.
    - Operates in both the U.S. and Europe with plans for further capacity expansion.
6. **Digital Realty**
    - Operates hundreds of carrier-neutral data centers worldwide.
    - Conducts projects to expand infrastructure capacity for AI workloads.
7. **Equinix**
    - Maintains a global network of over 250 data centers in more than 70 countries.
    - Has plans for new AI-focused facilities and expansions.
8. **Aligned Data Centers**
    - Backed by investors such as BlackRock, Microsoft, Nvidia, and xAI.
    - Operates and plans more than 50 campuses, building large-scale AI infrastructure.

Other Data Center Initiatives and Investments

9. **UPC Volt**
    - Plans an AI-ready data center campus in Telangana, India, supported by large-scale renewable energy.
10. **Starcloud**

- An early-stage initiative developing a "data center in space" concept—an innovative long-term approach.

11. **Major IT and Network Companies**

- Manufacturers such as Wiwynn support data centers by providing optimized servers, networking, and storage solutions.

Key Highlights

- **Mega technology giants:** Microsoft, AWS, Google, Meta
- **Infrastructure providers:** CoreWeave, Digital Realty, Equinix
- **Strategic consortiums:** Aligned Data Centers (BlackRock + Microsoft + Nvidia + xAI)
- **Innovative projects:** UPC Volt, Starcloud

The Stargate Project

Stargate is one of the largest and most up-to-date AI infrastructure and data center initiatives, involving major technology and investment giants. The Stargate Project—established in 2025—aims to build a massive global network of AI-focused data centers and infrastructure over the coming years.

Founding Partners / Stakeholders:

- OpenAI
- SoftBank Group
- Oracle
- MGX
  (Founded under the leadership of Masayoshi Son of SoftBank.)

Investment Target:

Approximately **$500 billion USD** in total investment, aiming for **10 gigawatts** of AI data center capacity.

Purpose:

To build large-scale, high-capacity data center networks to train and operate generative AI models, thereby enhancing global competitiveness in AI research, innovation, and industrial applications.

Active and Planned Facilities:

- **Abilene, Texas (USA):** Main campus and first AI data center—partially operational.
- Additional U.S. locations: Shackelford County (Texas), New Mexico, Midwest, Lordstown (Ohio).
- **United Arab Emirates:** A 1 GW AI infrastructure campus planned for Abu Dhabi (Stargate UAE), targeted for 2026.
- **South Korea:** Planned collaborations with Samsung and SK for new facilities.

Most of these large-scale projects are scheduled to be operational within the next 3–4 years.

Strategic Partners:

- Oracle – cloud infrastructure
- NVIDIA – GPU and AI hardware
- Samsung & SK – memory and semiconductor support
- Cisco, Arm, Microsoft – additional ecosystem partners

Stargate aims not only to build data centers but also to strengthen AI infrastructure leadership, enable high-performance AI model training, support economic growth and employment, and reinforce energy and data sovereignty strategies.

In summary:

Stargate is a $500 billion AI data center and infrastructure initiative led by OpenAI, SoftBank, Oracle, and major technology partners, targeting 10 GW capacity by 2029 and aiming to establish AI data centers globally, with a primary focus on the United States.

Usage Notes:

A lot of slides, pictures and articles are adopted from the presentations and documents published on internet by experts who know the subject very well. I would like to thank who prepared slides and documents. Also, these slides are made publicly available on the web for anyone to use If you choose to use them, I ask that you alert me of any mistakes which were made and allow me the option of incorporating such changes (with an acknowledgment) in my set of slides.

Sincerely,

Dr. Cahit Karakuş
cahitkarakus@esenyurt.edu.tr