Computer Architecture – Overview & Core Concepts

- Prepared by: ***Souvik Roy***
- Training Module
- Date: 12/11/2025

What is Computer Architecture?

- Computer Architecture is the **study of how computers are structured, designed, and how they operate internally**.
  It explains how:
- The CPU processes instructions
- Memory stores and retrieves data
- I/O devices communicate with the CPU
- Software interacts with hardware
- Computer architecture broadly includes:

Instruction Set Architecture (ISA) – Defines instructions the processor can execute

Microarchitecture – Internal organization of CPU components

System architecture – Entire hardware system including CPU, memory, buses, I/O devices

Importance in Software Development

Why developers should learn computer architecture:

Performance Optimization

- Knowing how cache, memory, and CPU pipelines work helps write faster programs.

Efficient Memory Usage

- Understanding memory allocation and paging helps reduce memory leaks and improve program stability.

Better Debugging

- When you understand registers, stack, heap, and instructions, debugging becomes easier.

Writing System-Level Code

Essential for:

- OS development
- Embedded systems
- Device drivers
- Compilers and interpreters

Working with Modern Technologies

- AI, cloud computing, and distributed systems require strong knowledge of architectural principles.

Basic Terminology

- **Instruction** – One operation the CPU can execute
- **Clock Cycle** – Time taken for one CPU tick
- **CPI** – Cycles required per instruction
- **Latency** – Time delay before execution starts
- **Throughput** – Amount of work done per unit time
- **Register** – Fastest form of internal storage
- **Cache** – Memory that stores recently used data
- **ALU** – Performs arithmetic and logical operations

Historical Development

- **1. Vacuum Tube Computers –** Huge, slow, power-consuming
- **2. Transistor Computers –** Smaller, faster, reliable
- **3. Integrated Circuits (ICs) –** Complex processors on chips
- **4. Microprocessors –** Complete CPU on a single chip
- **5. Modern Era –** Multi-core CPUs, GPUs, AI accelerators

RISC vs CISC

RISC (Reduced Instruction Set Computer)

- Simple, uniform instructions
- More registers
- Executed in one clock cycle
- Examples: ARM, RISC-V

CISC (Complex Instruction Set Computer)

- Large, complex instructions
- Fewer registers
- Multi-cycle instruction execution
- Examples: Intel x86, AMD

Modern CPUs actually use both techniques internally.

Pipeline & Superscalar Execution

- **Pipeline Execution**

Pipelining breaks instruction execution into stages—**Fetch, Decode, Execute, Memory Access, Write Back**—so multiple instructions can be processed simultaneously, each in a different stage. This increases **instruction throughput** without increasing clock speed.

- **Superscalar Execution**

Superscalar CPUs can issue **multiple instructions per clock cycle** by using multiple execution units (ALUs, FPUs, etc.). This enables parallel execution of independent instructions, further improving performance beyond pipelining.

Computer Arithmetic & ALU

ALU operations include:

- Integer arithmetic: +, -, ×, ÷
- Logic: AND, OR, XOR, NOT
- Shifts & rotates
- Comparisons
- Floating Point operations are handled by **FPU**:
- IEEE 754 standard

Virtual Memory & Paging

- Virtual memory makes a program think it has more RAM than available.

Key components:

- **Virtual Address → Physical Address translation**
- **Page** – 4KB block of memory
- **Page Table** – Maps virtual pages to physical pages
- **TLB** – Fast cache for page translations
- Benefits:
  Multi-programming
  Protection between processes

  Running large apps without enough RAM

Benchmarking & Metrics

- Benchmarking measures system performance using standardized tests.

Metrics:

CPI (Cycles Per Instruction)

- Average number of cycles the CPU takes to complete an instruction.

MIPS (Millions of Instructions Per Second)

- Indicates how many instructions the CPU executes in one second.

FLOPS

- Measures performance in scientific and AI tasks which rely on floating-point operations.

SPEC Benchmarks

- Industry-standard tests that compare the performance of different processors.

CPU Components

- **Control Unit (CU)** – Directs operations like a traffic controller
- **ALU** – Performs arithmetic and logic operations
- **Registers** – Temporary fast storage
- **Cache** – L1, L2, L3 memory
- **Buses** – Data and address pathways

Instruction Execution Steps

- **1. Fetch –** Get instruction from memory
- **2. Decode –** CU interprets it
- **3. Execute –** ALU performs operation
- **4. Memory Access –** Load/store operations
- **5. Write Back –** Store result in register

CPU Performance Factors

- **Clock speed (GHz):** Higher clock speed allows the CPU to execute more cycles per second, improving instruction throughput.
- **Pipeline efficiency:** Better pipelining keeps all stages busy, reducing stalls and increasing execution speed.
- **Cache hit ratio:** A higher hit ratio means data is found in cache more often, reducing slow memory accesses.
- **Branch prediction accuracy:** Accurate prediction prevents pipeline flushes and keeps instructions flowing smoothly.
- **Instruction-level parallelism:** More independent instructions allow the CPU to execute multiple operations simultaneously.
- **Thermal throttling:** When the CPU overheats, it slows down to reduce temperature, lowering performance.

Microarchitecture

- Microarchitecture is **how CPU components are internally designed**.

Includes:

- Pipelines
- Reorder buffers (ROB)
- Reservation stations
- Load/store buffers
- Multiple execution units
- Different CPUs with same ISA (e.g., x86) can have **different microarchitectures**.

ISA

Instruction Set Architecture defines:

- Available instructions : The set of operations a CPU can execute, defined by its instruction set architecture (ISA).
- Register set : Small, fast storage locations inside the CPU used to hold operands and intermediate results.
- Data types : The kinds of values the architecture supports, such as integers, floats, characters, etc.
- Addressing modes : The different ways an instruction can specify the location of data in memory.
- Memory model : The structure and rules defining how memory is organized, accessed, and shared between processes or threads.
- Examples:

x86 → Intel/AMD **ARM** → Smartphones **RISC-V** → Open-source ISA

Branch Prediction & Speculative Execution

Predicts the next instruction to avoid pipeline delays.

Types:

- Static prediction
- Dynamic prediction
- Two-level predictor
- Branch Target Buffer (BTB)

Speculative Execution

- CPU executes instructions **before it knows if they will be needed**.
  Boosts performance but caused vulnerabilities like **Spectre & Meltdown**.

Instruction-Level Parallelism

Out-of-Order Execution

- Allows instructions to run whenever resources are available.

Register Renaming

- Avoids false data conflicts by giving temporary names to registers.

CPU Cooling

Heat Sinks

- Dissipate heat into air through metal plates.

Liquid Cooling

- Uses liquid to transfer heat away from CPU.

Thermal Throttling

- Lowers CPU speed to prevent overheating.

Future CPUs

- **Quantum computing:** Uses qubits and quantum phenomena (superposition, entanglement) to solve certain problems exponentially faster than classical computers.
- **Neuromorphic computing:** Mimics the structure and functioning of the human brain using artificial neurons and synapses for highly efficient parallel processing.
- **Photonic computing:** Uses light instead of electricity for computation, offering extremely high speed and low energy consumption.

Cache Memory & Coherence

L1, L2, L3, MESI protocol

Cache Memory :

L1 Cache

- Fastest and closest to CPU, but very small.

L2 Cache

- Larger and slightly slower, acts as middle layer.

L3 Cache

- Shared among CPU cores for improved multi-core performance.

Cache Coherence

- Ensures all CPU cores see the same data.

MESI (Modified, Exclusive, Shared, Invalid) is a cache-coherency protocol that keeps data consistent across CPU caches.

- **M:** Cache has updated data not in memory.
- **E:** Cache has the only clean copy.
- **S:** Multiple caches have the same clean copy.
- **I:** Cache line is invalid.
  MESI prevents conflicting updates and reduces memory traffic in multiprocessor systems.

Memory Allocation & GC

Static Allocation

- Memory assigned at compile-time before program runs.

Dynamic Allocation

- Memory assigned at runtime using malloc/new.

Garbage Collection

- Automatically frees unused memory in languages like Java.

SSD Technology

NAND Flash Cells

- Store data electronically without moving parts.

NVMe Interface

- Provides ultra-fast storage access through PCIe.

Wear Leveling

- Spreads writing across cells to increase lifespan.

Error Detection & Correction

Parity Bits

- Add one extra bit to detect single-bit errors.

Hamming Code

- Detects and corrects single-bit errors.

ECC RAM

- Used in servers to prevent crashes due to memory corruption.

Memory Optimization

Prefetching fetches data into cache *before* it is needed, reducing stalls and improving CPU performance.
Cache Blocking (Tiling) reorganizes computations to access smaller "blocks" of data that fit in cache, improving locality and reducing cache misses.
LRU (Least Recently Used) is a cache-replacement policy that removes the block that has not been used for the longest time, helping keep frequently used data in cache.

Types of Memory

Registers

- Smallest, fastest memory inside CPU.

RAM

- Temporary storage used by running programs.

HDD & SSD

- Permanent storage for files and applications.

Data Storage

- **Binary:** The fundamental data representation in computers using 0s and 1s. All digital data—text, images, programs—is ultimately stored in binary form.
- **Blocks:** Fixed-size chunks of data used in **storage devices** (HDD/SSD). Reading or writing happens block-by-block, improving efficiency.
- **Pages:** Fixed-size units of **memory** used in virtual memory systems. Pages allow paging, address translation, and memory protection between processes.

Memory Management

- **Paging** divides memory into fixed-size pages and frames, enabling efficient, non-contiguous allocation and preventing external fragmentation.
- **Segmentation** divides memory into variable-sized logical segments (code, data, stack), matching how programs are structured and supporting protection and sharing.
- **Replacement Algorithms** decide which page to remove from memory when a page fault occurs. Common ones include **FIFO**, **LRU**, and **Optimal**, aiming to minimize page faults and improve performance.

I/O Overview

- I/O systems provide the mechanisms and interfaces through which the CPU exchanges data with external devices.
- **Key ideas:**
- **I/O devices differ widely in speed:** Some devices like keyboards are slow, while SSDs or GPUs are fast.
- **I/O operations are slower than CPU operations**, so the system must manage them efficiently.
- **The OS handles I/O scheduling**, deciding which device gets access first.
- **I/O controllers act as intermediaries**, translating CPU instructions into device-specific signals.

Communication Flow

The CPU uses several mechanisms to coordinate data transfer with I/O devices.

Ways communication happens:

- **Programmed I/O (Polling)**

  CPU repeatedly checks whether a device is ready.
  Disadvantage: CPU wastes time waiting.

- **Interrupt-Driven I/O**

  Device signals the CPU only when it needs attention.
  Advantage: CPU is free to do other work.

- **Direct Memory Access (DMA)**

  Data is transferred directly between device and RAM without CPU involvement.

- **Memory-Mapped I/O**

  Devices are assigned memory addresses so CPU

  reads/writes them like normal memory.

- **I/O Ports**

  Dedicated instructions handle I/O using special port numbers.

Buses & Transfer

A **bus** is a communication pathway that transfers data between components like CPU, memory, and devices.

Types of buses:

- **Address Bus**

Specifies memory or device addresses during data transfer.

- **Data Bus**

Carries actual data in or out of the CPU.

- **Control Bus**

Carries signals like read/write, interrupt requests, or memory access.

- **Examples of modern buses:**

USB → General-purpose peripheral connection

PCIe → High-speed connection for GPUs and SSDs

SATA → Connection for HDDs/SSDs

Bus performance depends on:

- Width (number of wires)
- Speed (frequency)
- Protocol efficiency

DMA

Direct Memory Access is a hardware feature that allows devices to transfer data directly to/from RAM **without CPU intervention**.

- **Why DMA is useful:**

Eliminates CPU involvement in repetitive data transfers.

Allows CPU to perform other tasks simultaneously.

Increases overall system performance.

- **How DMA works:**

CPU gives DMA controller the memory address and size.

DMA controller handles the transfer.

DMA sends interrupt to CPU when done.

- **Used in:**

Disk read/write

Audio/video streaming

High-speed USB transfers

I/O Protocols

Protocols define the rules for communication between devices and the system.

Examples:

- **USB (Universal Serial Bus)**

Connects peripherals like mouse, keyboard, pendrive; supports plug-and-play.

- **Thunderbolt**

High-speed data and display interface commonly used in premium laptops.

- **SATA**

Used mainly for disk storage devices.

- **NVMe (Non-Volatile Memory Express)**

Provides extremely fast communication with SSDs using PCIe lanes.

- **Ethernet / Wi-Fi**

Network communication standards for data transfer across systems.

I/O Performance Issues

I/O performance often determines how "fast" a computer feels, especially in tasks like file transfer and app loading.

Common issues:

- **High Latency**

Devices like HDDs have mechanical delays.

- **Bandwidth Limitations**

Slow buses can bottleneck fast devices.

- **I/O Contention**

Multiple devices compete for bandwidth.

- **CPU Overhead**

Polling or inefficient drivers slow performance.

Solutions:

- **Use caching to reduce access time**
- **Use faster buses (PCIe, USB 3, NVMe)**
- **Use DMA to bypass CPU**
- **Queuing & buffering handle bursts of I/O**
- **Parallel I/O for faster disk performance**

I/O Virtualization

Used mainly in cloud servers and virtual machines.

Key concepts:

- **Virtual I/O Devices**

VMs see "virtual" devices that are backed by real hardware.

- **Hypervisors**

Translate VM requests to physical device operations.

- **SR-IOV (Single Root I/O Virtualization)**

Allows one physical device (like a NIC) to act as multiple virtual devices.

- **Benefits:**

Efficient hardware sharing

Better performance for VMs

Improved security and isolation

Future Trends

- **Optical I/O**

Using light instead of electricity for ultra-fast communication.

- **Wireless connectors (WiGig, Wi-USB)**

High-speed data transfer without cables.

- **AI-driven controllers**

Predict workloads and optimize I/O scheduling.

- **Faster universal interfaces (USB 5.0, Thunderbolt 6)**

More speed and power for modern devices.

Parallel Computing

Parallel computing means performing multiple tasks simultaneously to speed up execution.
Modern systems—from smartphones to supercomputers—use parallelism.

- **Introduction to Parallel Computing :**

Parallel computing divides a big problem into smaller tasks that can run at the same time.

- **Why parallel computing is important:**

CPUs cannot keep getting faster due to power limits.

Large applications (AI, simulations, rendering) require massive processing.

Multi-core processors are now standard in every device.

Multi-core & GPUs

Multi-core CPU

- A CPU with multiple "brains" (cores) on one chip.
- Each core can run a separate thread or task.
- Useful for everyday multitasking (apps, browser tabs, OS tasks).

GPU (Graphics Processing Unit)

- Contains thousands of tiny cores designed for parallel work.
- Ideal for tasks with repetitive calculations:
  - Machine learning
  - Video rendering
  - Scientific simulations
  - Cryptography

Difference:

- CPU → Designed for **complex tasks** with decision-making (serial performance).
- GPU → Designed for **large-scale parallel tasks** without much branching.

SIMD vs MIMD

SIMD (Single Instruction, Multiple Data)

- One instruction operates on many data items at once.
- Used in video processing, image filtering, neural networks.
- Example: Applying a brightness filter to every pixel simultaneously.

MIMD (Multiple Instruction, Multiple Data)

- Each processor works independently on different instructions and data.
- Used in multi-core CPUs running independent programs.
- Example:
  - Core 1 → Running Chrome
  - Core 2 → Running a game
  - Core 3 → Background antivirus scan

FPGAs & ASICs

Hardware Acceleration through FPGAs and ASICs

FPGA (Field Programmable Gate Array)

- A reconfigurable chip that can be programmed many times.
- Used in prototyping, high-frequency trading, custom AI pipelines.
- Advantage: Flexible and customizable.

ASIC (Application-Specific Integrated Circuit)

- A chip designed for **one specific task only**.
- Much faster and more efficient than general-purpose CPUs.
- Used in:
  - Bitcoin miners
  - Smartphone processors
  - AI accelerators (e.g., Google TPU)

Challenges

Challenges in Parallel Programming and Synchronization

- **Race Conditions**

Occurs when two threads try to access/modify data at the same time.

- **Deadlocks**

Two threads wait indefinitely for each other's resources.

- **Load Balancing**

Some cores may finish early while others remain overloaded.

- **Communication Overhead**

Threads need to share data, which can slow down execution.

- **Debugging Complexity**

Parallel programs are much harder to debug due to non-deterministic behavior.

- **Memory Consistency Issues**

Different cores may see different versions of data if cache coherence is poor.

Data vs Task Parallelism

Data Parallelism

- Same task is applied to multiple data chunks at the same time.
- Example: Processing pixels in an image or rows in a matrix.
- GPUs excel at this.

Task Parallelism

- Different tasks run simultaneously.
- Example:
  - One thread fetches data
  - Another processes it
  - Another saves results

Hybrid Parallelism

- Modern systems often combine both for maximum performance.

References

Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann.
Stallings, W. (2016). Computer Organization and Architecture: Designing for Performance (10th ed.). Pearson.
Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design: The Hardware/Software Interface (5th ed.). Morgan Kaufmann.
IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019).
Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
SPEC. Standard Performance Evaluation Corporation (SPEC) Benchmark Suites Documentation.