Cloud Infrastructure

First Generation of Computers: Vacuum Tubes

Use of Vacuum Tubes

The **first generation of digital computers (1940s–mid-1950s)** relied on **vacuum tubes** as the fundamental components for **digital logic circuits** and **main memory**. Vacuum tubes functioned as electronic switches and amplifiers, enabling binary representation of data. However, they were bulky, generated excessive heat, consumed large amounts of power, and were prone to frequent failures.

The IAS Computer

One of the most influential machines of this generation was the **IAS computer**, designed under the guidance of **John von Neumann** at the Princeton Institute for Advanced Study.

- **Stored-Program Concept:**
  - Introduced by **John von Neumann** in 1945 (initially described in the **EDVAC report**).
  - The idea was revolutionary: both **instructions** and **data** could be stored in the same memory, allowing the computer to be more flexible and programmable.
  - This principle became the foundation of what is now known as the **von Neumann architecture**.
- **Design and Completion:**
  - Work on the IAS computer began in the **mid-1940s**.
  - The machine was completed in **1952** at Princeton.
  - Its design served as the **prototype for most subsequent general-purpose computers**, influencing early systems such as the IBM 701 and UNIVAC.

Von Neumann Architecture

The IAS computer, designed in the late 1940s and completed in 1952, is the archetype of the **stored-program computer model**. Its architecture became the foundation of modern computers and is often referred to as the **von Neumann architecture**.

Main Components

1. Main Memory (M)

- Stores both **instructions** and **data** in the same memory space.
- Each location has a unique **address** (M(0) … M(4095), meaning 4096 words in total).
- This dual storage of code and data distinguishes the stored-program concept.
- Memory is accessed via **addresses** supplied by the CPU.

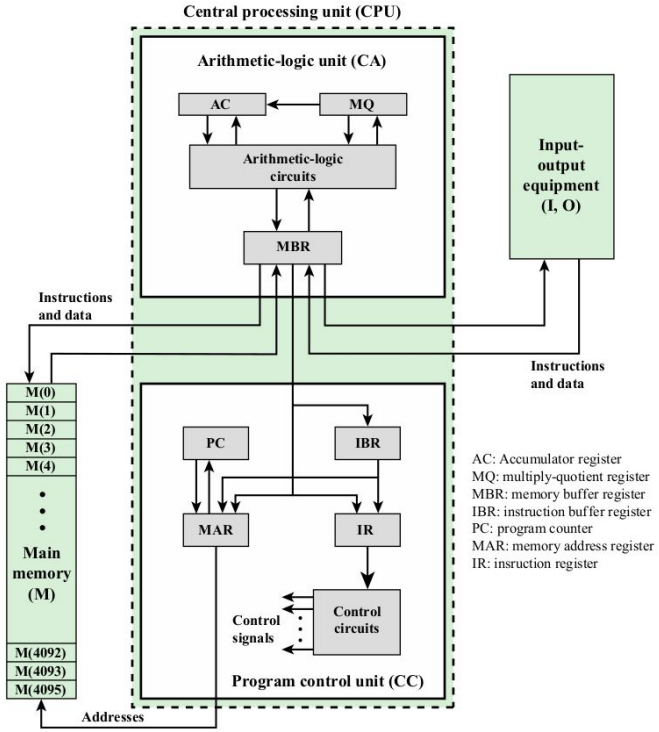


Figure 1.6 IAS Structure

Von Neumann Architecture

2. Central Processing Unit (CPU)

Divided into two main units:

(a) Arithmetic-Logic Unit (CA)

- **Registers inside ALU:**
  - **AC (Accumulator Register):** Holds intermediate results of arithmetic/logic operations.
  - **MQ (Multiplier-Quotient Register):** Used during multiplication and division operations.
- **MBR (Memory Buffer Register):** Temporarily holds data fetched from or written to memory.
- The **arithmetic-logic circuits** perform basic operations (addition, subtraction, logic operations, multiplication, division).

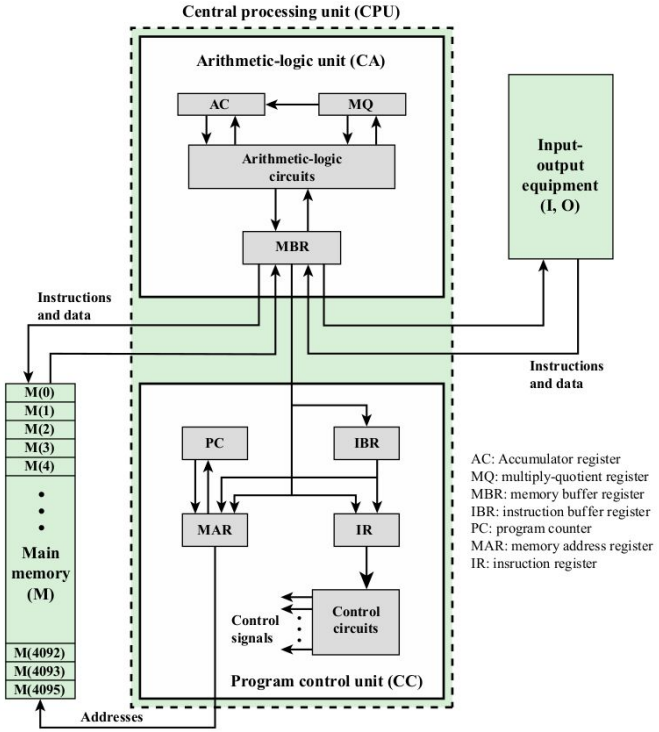


Figure 1.6 IAS Structure

Von Neumann Architecture

(b) Program Control Unit (CC)

- Directs the sequence of operations in the computer.
- Key registers:
  - **PC (Program Counter):** Holds the address of the next instruction to be executed.
  - **MAR (Memory Address Register):** Holds the memory address being accessed.
  - **IR (Instruction Register):** Holds the current instruction being decoded/executed.
  - **IBR (Instruction Buffer Register):** Stores the right-hand instruction of a memory word (since each word could hold two instructions).
- **Control Circuits:** Generate the control signals that coordinate data transfer and execution steps.

3. Input–Output Equipment (I/O)

- Provides a mechanism to transfer data and instructions between the computer and the external world.
- In early IAS design, input was via punched cards or paper tape; output was printed or displayed.
- Instructions and data flow through memory and CPU to/from I/O devices.

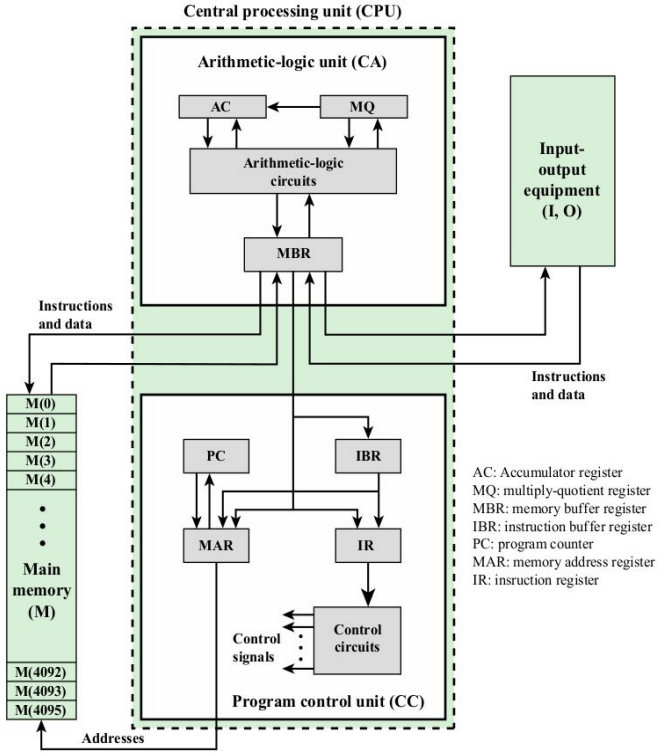


Figure 1.6 IAS Structure

Von Neumann Architecture

Instruction Processing in IAS

1. The **PC** supplies the address of the instruction to the **MAR**.
2. The instruction is fetched into the **MBR** and then placed in **IR** (or IBR, if two instructions are packed).
3. The **control circuits** decode the instruction and issue signals to execute it.
4. Data required is fetched into **MBR**, sent to the ALU (AC/MQ), processed, and results written back to memory or a register.
5. The **PC** is updated to the next instruction.

This cycle is the essence of the **fetch–decode–execute cycle**, still followed in today's processors.

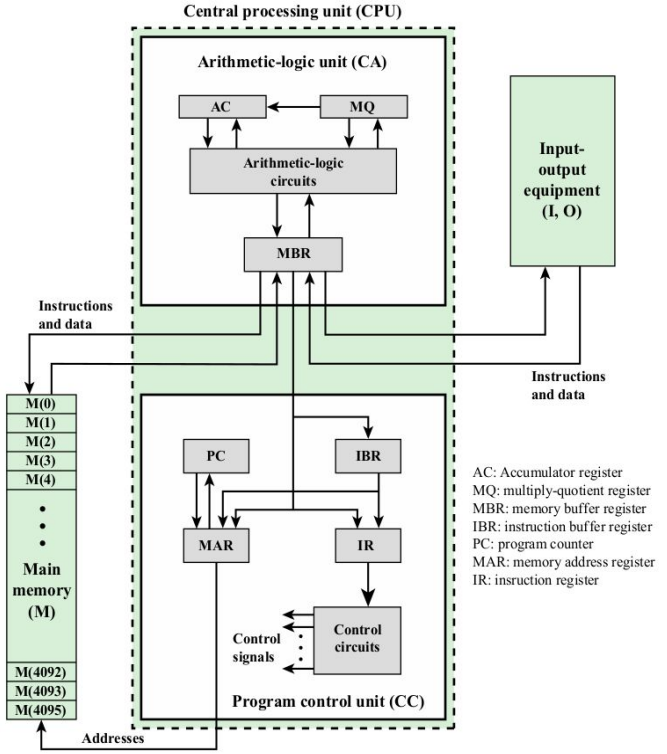


Figure 1.6 IAS Structure

IAS Memory Format

IAS Memory Formats

- **Word length:** 40 bits. Each word stores either a **number** or **two instructions**.

Number Word (Figure 1.7a)

- **Bit 0:** Sign bit (0 = positive, 1 = negative).
- **Bits 1–39:** Magnitude of the number (binary integer).

Instruction Word (Figure 1.7b)

- Each word holds **two 20-bit instructions**:
  - **Left instruction:** bits 0–19
  - **Right instruction:** bits 20–39
- Each instruction = **Opcode (8 bits)** + **Address (12 bits)**.

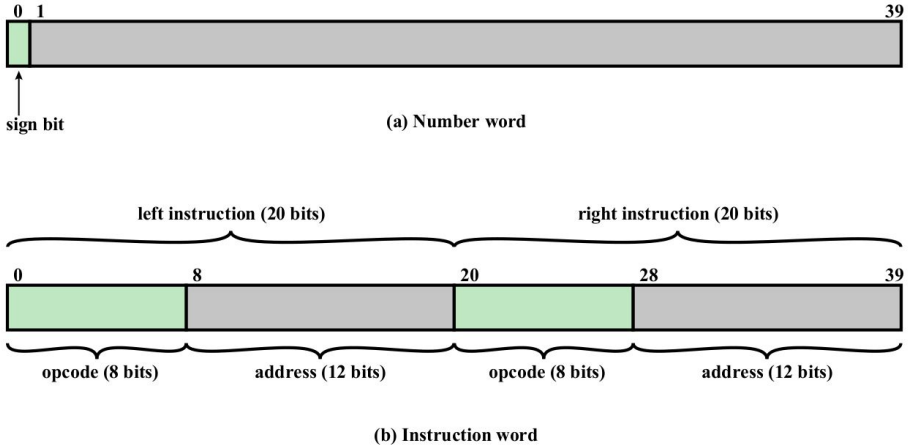


Figure 1.7 IAS Memory Formats

IAS Memory Format

Execution

- **Left instruction** executed first; **Right instruction** held in **IBR**.
- **PC** advances only after both instructions are executed.

Significance

- Compact storage (two instructions per word).
- Early example of **instruction set architecture (ISA)**.
- Foundation for the **fetch–decode–execute cycle** in modern CPUs.

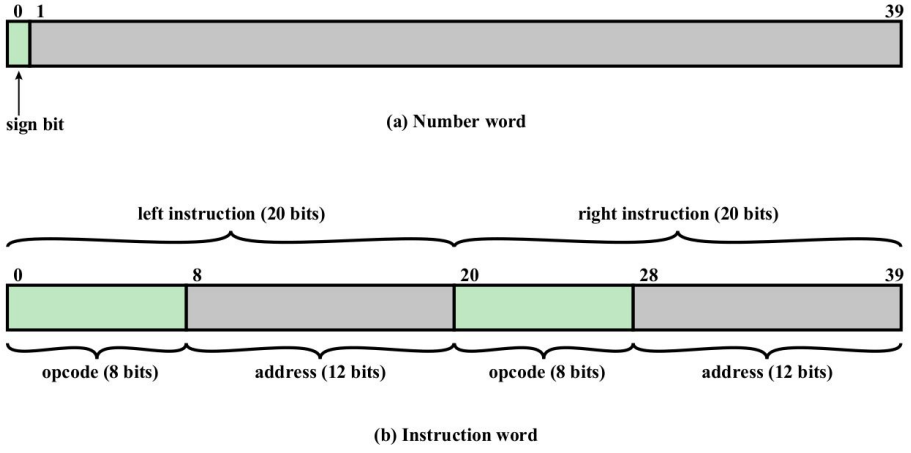


Figure 1.7 IAS Memory Formats

IAS Registers

Memory Buffer Register (MBR):
Holds a word moving between memory and CPU (or I/O).

Memory Address Register (MAR):
Stores the address of the memory word being accessed.

Instruction Register (IR):
Holds the **8-bit opcode** of the instruction currently being executed.

Instruction Buffer Register (IBR):
Temporarily stores the **right-hand instruction** of a 40-bit word until execution.

Program Counter (PC):
Contains the address of the **next instruction pair** to be fetched.

Accumulator (AC) & Multiplier Quotient (MQ):
Work registers in the ALU for storing operands and results of arithmetic/logic operations.

IAS Instruction Set

1. Data Transfer

- **LOAD MQ:** Transfer contents of MQ to AC.
- **LOAD MQ, M(X):** Transfer contents of memory location X into MQ.
- **STOR M(X):** Store contents of AC into memory location X.
- **LOAD M(X):** Load value from memory location X into AC.
- **LOAD –M(X):** Load negative of value at memory location X into AC.
- **LOAD |M(X)|:** Load absolute value of M(X) into AC.
- **LOAD –|M(X)|:** Load negative absolute value of M(X) into AC.

2. Unconditional Branch

- **JUMP M(X, 0:19):** Take next instruction from left half of word at M(X).
- **JUMP M(X, 20:39):** Take next instruction from right half of word at M(X).

3. Conditional Branch

- **JUMP+ M(X, 0:19):** If AC ≥ 0, take next instruction from left half of M(X).
- **JUMP+ M(X, 20:39):** If AC ≥ 0, take next instruction from right half of M(X).

IAS Instruction Set

4. Arithmetic

- **ADD M(X):** Add contents of M(X) to AC.
- **ADD |M(X)|:** Add absolute value of M(X) to AC.
- **SUB M(X):** Subtract M(X) from AC.
- **SUB |M(X)|:** Subtract absolute value of M(X) from AC.
- **MUL M(X):** Multiply M(X) by MQ; result → (high bits in AC, low bits in MQ).
- **DIV M(X):** Divide AC by M(X); quotient in MQ, remainder in AC.
- **LSH:** Shift AC left by 1 bit (multiply by 2).
- **RSH:** Shift AC right by 1 bit (divide by 2).

5. Address Modify

- **STOR M(X, 8:19):** Replace left address field of M(X) with rightmost 12 bits of AC.
- **STOR M(X, 28:39):** Replace right address field of M(X) with rightmost 12 bits of AC.

Second Generation of Computers: Transistors

Transistor Invention: First demonstrated at **Bell Labs in 1947** (Bardeen, Brattain, Shockley).
Solid-State Device: Made from semiconductor materials (silicon or germanium).
Advantages over Vacuum Tubes:

- Much **smaller** in size.
- **Cheaper** to produce.
- More **reliable** (no filaments to burn out).
- **Lower heat dissipation** → reduced cooling needs.
- Faster **switching speed** → better performance.

Commercial Use: By the **late 1950s**, fully transistorised computers became available (e.g., IBM 1401, IBM 7090).

Second Generation Computers (1950s – 1960s)

Hardware advances:

- Used **transistors** → faster, smaller, cheaper, and more reliable than vacuum tubes.
- Supported **more complex arithmetic and logic units** and improved control units.

Software advances:

- Introduction of **high-level programming languages** (e.g., FORTRAN, COBOL).
- Emergence of **system software** that enabled:
  - Loading and managing programs.
  - Data transfer between CPU and peripherals.
  - Use of **libraries** for common computations, reducing repetitive coding.

Impact:

- Increased programmer productivity.
- Laid the foundation for **operating systems** and the idea of software as a separate layer from hardware.

IBM 7094 Computer Configuration (Second Generation)

Main Components

- **CPU:** Executes instructions and manages overall system control.
- **Memory:** Stores both programs and data.
- **Multiplexor:** Acts as a traffic controller, coordinating communication between CPU, memory, and peripheral devices.
- **Data Channels:** Independent I/O pathways that allow data transfer between peripherals and memory/CPU without constant CPU intervention.

Peripheral Devices

- **Magnetic tape units, card punch, line printer, card reader** → batch processing input/output.
- **Drum and disk storage** → secondary storage for programs and data.
- **Hypertapes and teleprocessing equipment** → supported remote processing and larger storage needs.

Significance

- One of the earliest systems to support **multiprogramming** and efficient I/O using **data channels** (precursor to modern **DMA – Direct Memory Access**).
- Demonstrates the increasing separation between **CPU** and **I/O**, freeing the CPU for computation while data transfer occurred in parallel.

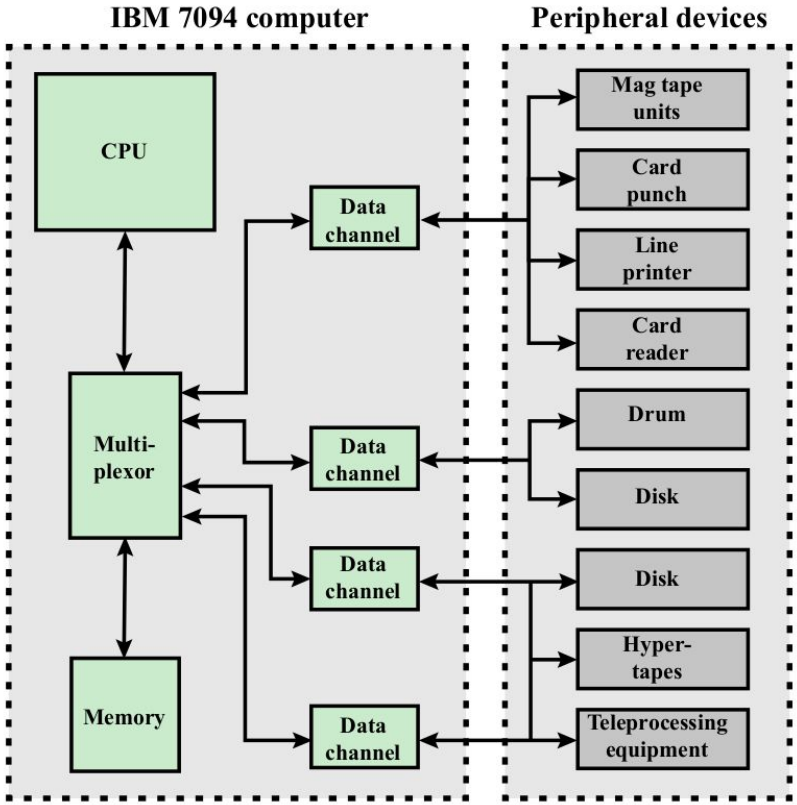


Figure 1.9 An IBM 7094 Configuration

Third Generation of Computers: Integrated Circuits (1960s)

Invention: Integrated Circuit (IC) invented in **1958** (Jack Kilby, Robert Noyce).

Discrete components before ICs:

- Used **individual transistors**, each packaged separately.
- Wired/soldered together on boards → bulky, costly, and error-prone manufacturing.

Integrated Circuit: Combined **multiple transistors, resistors, and capacitors** into a **single silicon chip**, reducing size and cost while improving reliability.

Key Systems:

- **IBM System/360** → introduced the idea of a **family of computers with compatible software**.
- **DEC PDP-8** → one of the first **minicomputers**, making computing affordable for smaller institutions.

Fundamental Computer Elements

At the lowest hardware level, all computers are built from two basic elements:

(a) Gate (Logic Element)

- Implements a **Boolean logic function** (e.g., AND, OR, NOT).
- Accepts one or more binary **inputs** and produces a binary **output**.
- Can be controlled by an **activate signal** (enabling or disabling the function).
- Used in arithmetic operations, comparisons, and control decisions inside the CPU.

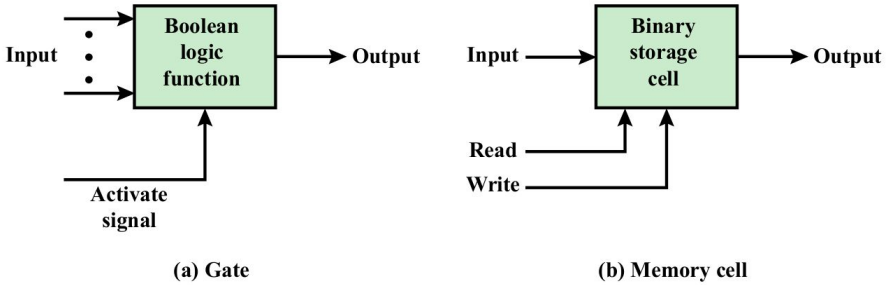


Figure 1.10 Fundamental Computer Elements

(b) Memory Cell (Storage Element)

- Stores a **single binary digit (0 or 1)**.
- Has three basic operations:
  - **Write:** Store a bit value into the cell.
  - **Read:** Retrieve the stored value.
  - **Input/Output:** Flow of binary data in and out.
- Memory cells are combined to form larger **registers, caches, and main memory**.

Integrated Circuits (ICs)

Computer Functions

- **Data storage:** Provided by memory cells.
- **Data processing:** Performed by logic gates.
- **Data movement:** Paths (buses) move data between memory and gates, or memory-to-memory.
- **Control:** Paths also carry control signals to coordinate operations.

Construction of ICs

- Computers are built from **gates + memory cells + interconnections**.
- These are made using **simple digital electronic components** like transistors, resistors, and conductors.
- **Semiconductor fabrication (silicon):**
  - Many transistors can be created simultaneously on a single silicon wafer.
  - Interconnections are made using **processor metallization**, linking transistors into complete circuits.

Importance

- ICs integrate **thousands of transistors** in a compact form.
- Cheaper, faster, and more reliable than discrete components.
- Enabled the development of **powerful third-generation computers**.

Moore’s Law

Origin: Proposed by **Gordon Moore** (Intel co-founder) in **1965**.

Statement: The number of transistors on a chip would **double approximately every two years**, while cost per transistor would **remain nearly constant**.

Implications:

- Computing power increases **exponentially**.
- Performance improves while devices become **smaller, cheaper, and more energy-efficient**.
- Drove rapid progress in electronics, enabling modern **CPUs, memory, and portable devices**.

Industry Impact:

- Became the **guiding principle** for semiconductor design and manufacturing.
- Paved the way for innovations in **supercomputing, personal computers, mobile devices, and cloud infrastructure**.

Consequences of Moore’s Law

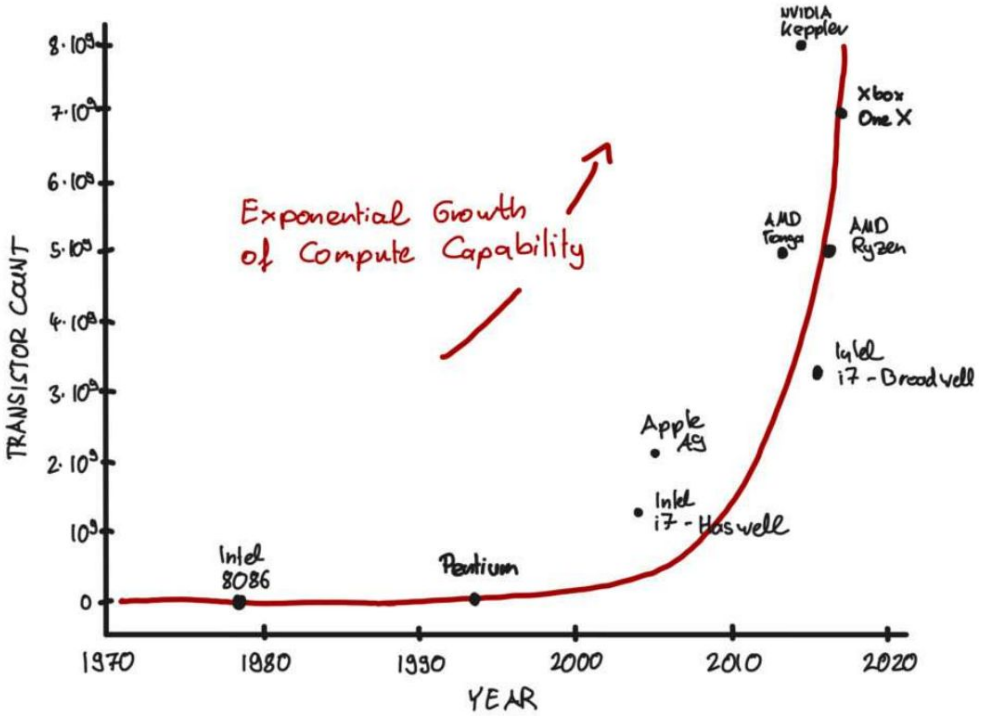
1. **Falling cost:** Computer logic and memory circuitry became dramatically cheaper.
2. **Higher speed:** Shorter electrical paths increased operating speeds.
3. **Miniaturisation:** Computers became smaller, portable, and usable in diverse environments.
4. **Lower power:** Reduced power consumption and cooling requirements.
5. **Integration:** Fewer interchip connections as more components fit onto a single chip.

Moore’s Law – Graph

- **X-axis (Year):** Timeline from 1970 to 2020.
- **Y-axis (Transistor Count):** Number of transistors on a chip (logarithmic scale).

Key Observations

- Shows **exponential growth** in transistor count over time.
- Early chips: **Intel 8086 (1978)**, **Pentium (1993)** had relatively few transistors.
- Later chips: **Intel i7 (Haswell, Broadwell)**, **Apple A9**, **AMD Ryzen**, **NVIDIA Kepler**, **Xbox One X** demonstrate dramatic increases.
- Trend follows Moore’s prediction: **doubling every ~18–24 months**, leading to sharp rises in compute power.



Moore’s Law – Modern Perspective

- **NVIDIA CEO Jensen Huang (2022):** Declared that *“Moore’s Law is dead”*, citing rising costs of semiconductor manufacturing.
- **Reasoning:**
  - **Silicon wafers** and fabrication costs have grown enormously.
  - Shrinking transistors further is reaching **physical and economic limits**.
  - Performance improvements are no longer doubling at the same pace.

Implications

- **End of exponential scaling:** The historical doubling trend (every 18–24 months) is slowing.
- **Higher prices:** Advanced chips (e.g., GPUs) are becoming much more expensive.
- **New directions:** Instead of raw transistor scaling, progress now comes from:
  - **Parallelism** (multi-core, GPUs).
  - **Specialised hardware** (AI accelerators, TPUs).
  - **3D stacking and chiplets** (packaging innovations).
  - **Energy efficiency optimisation.**

Moore’s Law – Future (Projections)

- **Target:** By **2030**, chipmakers (e.g., Intel) aspire to reach **1 trillion transistors per package**.
- **Enabling Technologies:**
  - **RibbonFET:** Next-generation transistor architecture (successor to FinFET).
  - **PowerVia:** Advanced power delivery method for improved efficiency.
  - **High NA EUV (Extreme Ultraviolet Lithography):** Enables finer patterning of chips.
  - **2.5D/3D Packaging:** Stacking and interconnecting chips vertically for higher density.

Outlook

- Traditional transistor scaling is slowing, but **innovation in architecture and packaging** is expected to keep advancing performance.
- The future focus shifts from pure transistor miniaturisation to **new materials, 3D integration, and specialised designs**.

Moore’s Law – Future Directions

Essential Technologies

- **New transistor designs** (e.g., RibbonFET, nanosheet transistors).
- **Lithography breakthroughs** (e.g., EUV, High-NA lithography).
- **Advanced packaging** (2.5D/3D integration, multiple tiles per package).

New Concepts

- **New types of switches** beyond traditional CMOS.
- **Higher performance** through architectural and design improvements.
- **Greater energy efficiency**, enabling sustainable computing.

New Capabilities

- **Efficient power delivery** (technologies like PowerVia).
- **Expanded memory resources** with faster, denser memory.
- **Material breakthroughs** (exploring alternatives to silicon for scaling).

Limitations of Moore's Law

Physical Limits:

- As transistors approach the **atomic scale**, quantum effects (e.g., **electron tunneling**) disrupt reliable operation.

Heat Dissipation:

- Densely packed transistors **generate more heat**, making it harder to cool processors effectively.

Power Consumption:

- Higher transistor density without proportional efficiency improvements → **excessive power use**.

Material Constraints:

- **Silicon technology** is reaching its physical boundaries, requiring exploration of **new materials and architectures** (e.g., graphene, carbon nanotubes, quantum computing).

Diminishing Performance Gains:

- Even with more transistors, the **improvements in speed and energy efficiency are slowing down**.

Manufacturing Complexity:

- Fabricating chips at **nanometer scales** demands **extremely precise and costly** manufacturing technologies (e.g., EUV lithography).

Limitations of Moore's Law

Alternative Paradigms:

- The rise of **specialised computing** (GPUs, TPUs, quantum processors) shifts focus away from general-purpose CPUs.
- This **challenges the relevance** of Moore's Law for traditional CPU scaling.

Economic & Market Shifts:

- Focus is moving from **raw transistor counts** to:
  - **Architectural optimisations** (better CPU/GPU designs).
  - **Parallel processing** (multi-core, GPUs, TPUs).
  - **Alternative computing models** (specialised accelerators).

Outcome:

- Moore's Law is **slowing down** due to physical, economic, and market challenges.

Future Directions Beyond Moore's Law:

- **3D stacking** – vertical integration of chips for higher density.
- **Neuromorphic computing** – brain-inspired architectures.
- **Quantum computing** – exploiting quantum mechanics for massive speedups.

Intel x86 vs ARM

- Two dominant processor families: **Intel x86** and **ARM**.
- **x86**: Based on CISC (Complex Instruction Set Computer), decades of design aimed at supporting complex instructions.
- **ARM**: Based on RISC (Reduced Instruction Set Computer), widely used in embedded systems, known for efficiency and power.

Evolution of Intel Processors

8080 – First general-purpose microprocessor (8-bit), used in early personal computers like the Altair.
8086 – 16-bit machine, introduced x86 architecture, included instruction queue; 8088 variant used in IBM PC.
80286 – Extended 8086, supported up to 16 MB memory.
80386 – First 32-bit Intel processor, introduced multitasking.
80486 – Added pipelining, better cache, built-in math coprocessor.
Pentium Series – Introduced **superscalar execution** (multiple instructions in parallel), MMX technology, floating-point and SIMD (SSE) extensions.
Core Family – First true Intel x86 micro-core.
Core 2 – Extended to 64-bit, multi-core processors (up to 4, later 10 cores), introduced Advanced Vector Extensions (AVX).

Semiconductor Memory

1970 (Fairchild): First capacious semiconductor memory produced.

- About the size of a single core.
- Could hold **256 bits** of memory.
- Non-destructive and much faster than magnetic core memory.

1974: Semiconductor memory became cheaper per bit than core memory.

- Rapid decline in memory cost.
- Increase in memory density.
- Memory and processor advancements changed computers dramatically in less than a decade.

Generations: Since 1970, semiconductor memory has gone through **13 generations**.

- Each generation → 4× storage density of previous one.
- Declining cost per bit and faster access times.

Later Generations:

- **LSI (Large Scale Integration)**
- **VLSI (Very Large Scale Integration)**
- **ULSI (Ultra Large Scale Integration)**

Embedded Systems

- Use of **electronics + software** inside products.
- Billions produced each year; embedded in larger devices.
- Devices powered by electricity often have embedded computing systems.
- **Tightly coupled to environment**, giving rise to **real-time constraints**:
  - Speed, precision, and time duration requirements.
  - Timing of software operations critical.
- Managing multiple simultaneous activities → complex **real-time constraints**.

Examples: Washing machines, calculators, mobile phones, cameras, surveillance systems.

Internet of Things (IoT)

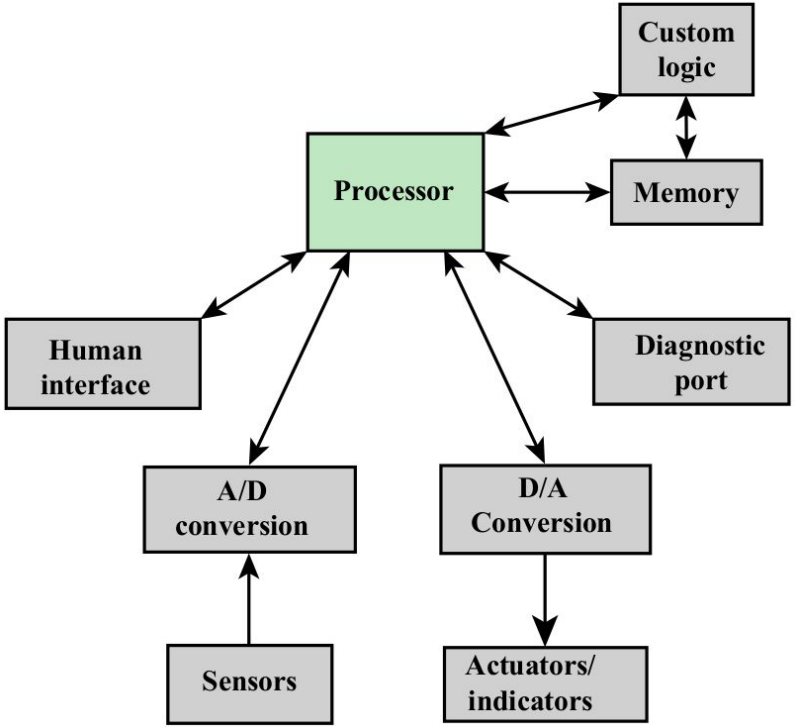
- Refers to the **expanding interconnection of smart devices** (appliances to sensors).
- Driven by deeply embedded devices.

Generations of IoT Deployment:

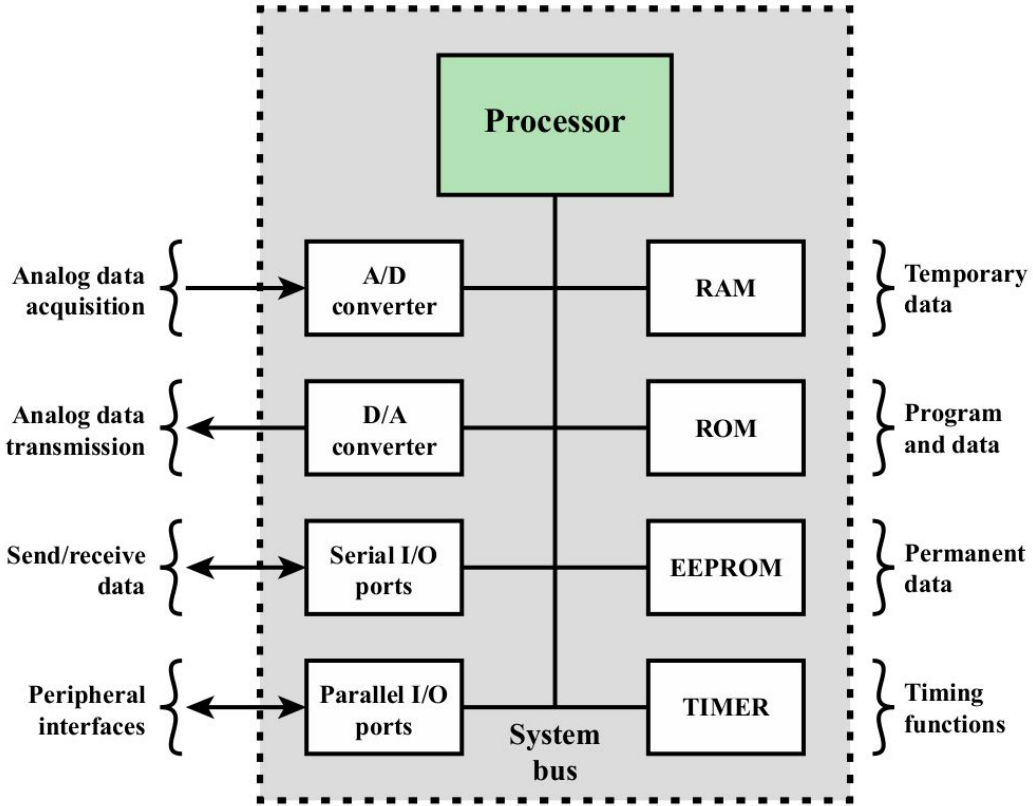
1. **Information Technology (IT):**
   - PCs, servers, routers, firewalls.
   - Bought by enterprise IT.
   - Primarily **wired connectivity**.
2. **Operational Technology (OT):**
   - Machines/appliances with embedded IT (e.g., SCADA, medical machinery, kiosks).
   - Built by non-IT companies.
   - Bought as appliances by enterprise OT.
   - Mostly **wired connectivity**.
3. **Personal Technology:**
   - Smartphones, tablets, eBook readers.
   - Bought by consumers.
   - **Wireless connectivity** (often multiple forms).
4. **Sensor/Actuator Technology:**
   - Single-purpose devices (IoT devices).
   - Exclusively **wireless** connectivity.
   - Often part of larger systems.

Note: IoT = Fourth generation, marked by billions of embedded devices.

Possible Organization of an Embedded System



Typical Microcontroller Chip Elements



ARM Architecture

- **Definition:**
  ARM refers to a processor architecture derived from **RISC (Reduced Instruction Set Computer)** design principles, widely used in embedded systems.
- **Origin:**
  Developed by **ARM Holdings**, based in Cambridge, England.
- **Design Features:**
  - Family of RISC-based **microprocessors** and **microcontrollers**.
  - Known for **high speed**, **small die size**, and **low power consumption**.
- **Adoption:**
  - ARM is the **most widely used embedded processor architecture**.
  - In fact, it is the **most widely used processor architecture in the world** across all domains.
- **Name:**
  ARM originally stood for **Acorn RISC Machine**, later renamed **Advanced RISC Machine**.

ARM Products

1. Cortex-A / Cortex-A50 Series

- Designed for **high-performance applications**.
- Used in **smartphones, tablets, and high-end embedded systems**.
- Supports advanced features like **virtualisation, multimedia, and complex operating systems** (e.g., Android, Linux).

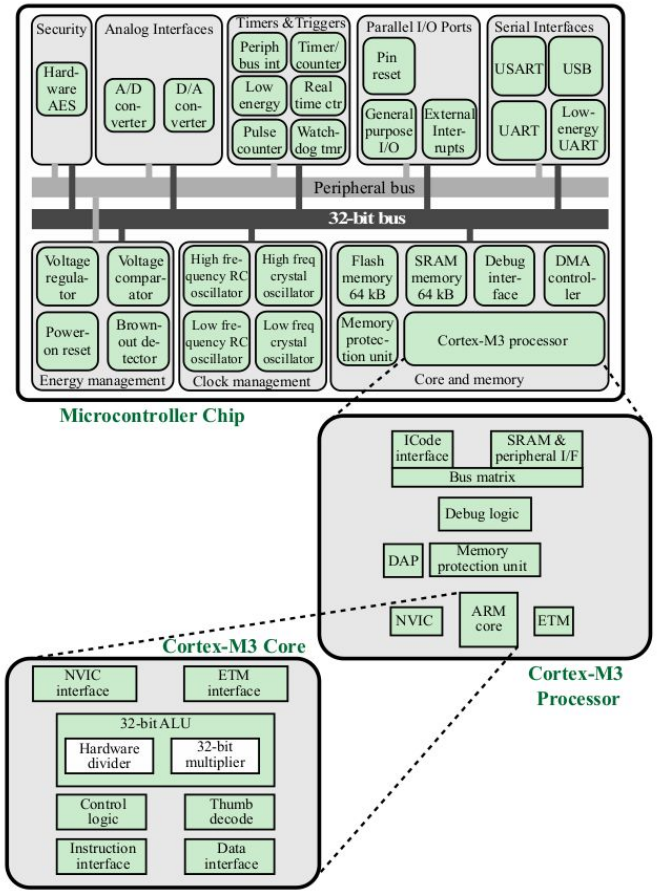
2. Cortex-R Series

- Optimised for **real-time applications**.
- Used in **automotive systems, storage controllers, and industrial automation**.
- Balances performance with **deterministic real-time response**.

3. Cortex-M Series

- Designed for **microcontroller applications**.
- Widely used in **IoT devices, wearables, and low-power embedded systems**.
- Variants:
    - **Cortex-M0 / M0+:** Ultra-low power, basic microcontrollers.
    - **Cortex-M3:** Higher performance, widely used in embedded controllers.
    - **Cortex-M4:** Adds **DSP (Digital Signal Processing)** and floating-point support for signal-intensive tasks.

Typical Microcontroller Chip Based on Cortex-M3



Generations of Computers (Summary)

First Generation (1946–1959)

- Technology: **Vacuum tubes**
- Large, expensive, and generated lots of heat.
- Examples: ENIAC, UNIVAC.

Second Generation (1959–1965)

- Technology: **Transistors**
- Smaller, faster, more reliable, and cheaper than vacuum tubes.
- Widely used in business and scientific applications.

Third Generation (1965–1971)

- Technology: **Integrated Circuits (ICs)**
- Multiple transistors on a single chip.
- Increased speed, efficiency, and reduced size.

Generations of Computers (Summary)

Fourth Generation (1971–1980)

- Technology: **Very Large Scale Integration (VLSI)**
- Thousands of transistors on a single chip.
- Emergence of **microprocessors**.

Fifth Generation (1980–Present)

- Technology: **Ultra Large Scale Integration (ULSI)**
- Millions (now billions) of transistors per chip.
- Basis for **modern processors, AI, IoT, and high-performance computing**.

Sixth Generation of Computers

Timeline: Started around 2000, still evolving.
Debate: Some classify current era as part of the **fifth generation**, since AI is still maturing.
Key Features:

- **Quantum computing** and **nanotechnology** drive advances.
- **Intelligent computers** are a defining characteristic.
- **URLLC (Ultra-Reliable Low-Latency Communication)** is a major feature.
- **Natural Language Processing (NLP)** enables human-like interaction.
- **New interfaces**: voice commands, **AR/VR/MR** integration.

Quantum Computing – Microsoft's Role

Project: Microsoft + Atom Computing, €80 million investment.
Location: Denmark, Nordic region.
Goal: Build the world's most powerful commercial quantum computer, named *Magne*.
Concerns:

- Raises global alarm about **tech power concentration** and **national control**.
- Described as "playing God with physics" due to its disruptive potential.

Significance: Shows how **quantum computing is moving from theory to large-scale commercial projects**, aligning with the **sixth generation of computing**.

Types of Computers

Mainframe

- Multi-user systems.
- Support hundreds of users simultaneously.
- Common in enterprises, banks, and government systems for large-scale transaction processing.

PC (Personal Computer)

- Single-user system.
- Moderately powerful microprocessor.
- Used for general purposes like office work, education, and personal applications.

Workstation

- Also single-user, but with more powerful processors.
- Used in engineering, scientific, or graphics-intensive applications.

Embedded System

- Specialized computing devices integrated into everyday objects.
- Optimized for specific functions (e.g., microcontrollers in washing machines, cars, medical devices).

Types of Computers

Server

- Powerful computers designed to provide resources/services to other devices.
- Handle tasks like data storage, website hosting, and managing network traffic.

HPC Clusters (High-Performance Computing Clusters)

- Groups of **powerful interconnected computers** working collaboratively.
- Designed to solve **complex computational problems** that a single system cannot handle.
- Commonly used in **scientific research, climate modeling, simulations, genomics, and large-scale data analysis**.

Quantum Computers

- Use the **principles of quantum mechanics** (superposition, entanglement) to perform calculations.
- Provide **exponentially faster computation** for certain tasks compared to classical computers.
- Applications include **cryptography, drug discovery, optimization, and material science**.

Computer Architecture

High-level design (abstract)

- Defines **logic**: instruction sets, addressing modes, registers, data types.
- Describes **what the system should do**.
- Blueprint/interface between hardware and software.
- Programmer views it in terms of **functional behaviour** and instruction set.
- Example: deciding whether a CPU supports a multiplication instruction.
- Comes **first** before organisation.

Computer Organisation

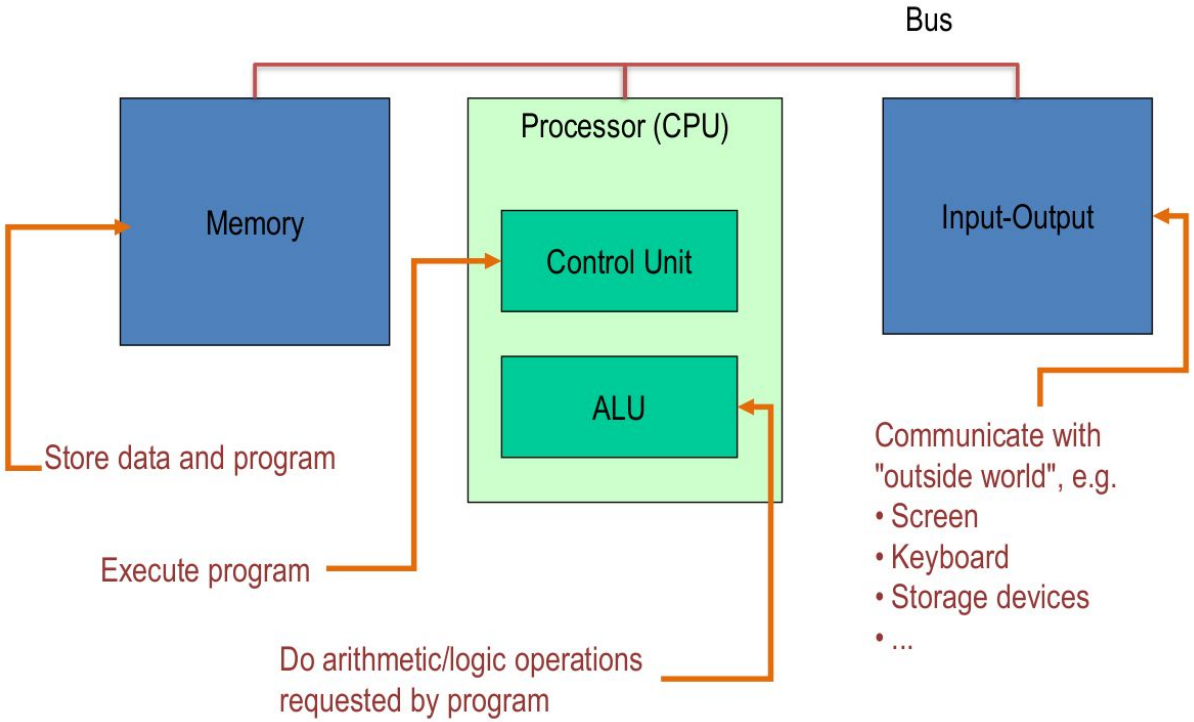
Low-level design (implementation)

- Involves **physical components**: circuits, signals, adders, peripherals.
- Describes **how the system actually works**.
- Provides structural relationships between components.
- Implements the architecture in hardware.
- Programmer sees it as **interfaces, memory technology, and control signals**.
- Example: multiplication could be done with adder + shift register, or a dedicated multiplier circuit.
- Different vendors (Intel, AMD) may implement the same architecture differently.

Relationship

- **Architecture = What to do.**
- **Organisation = How to do it.**
- **One architecture → Several organisations.**
- Example: x86 architecture → implemented by Intel & AMD in multiple ways, across 32-bit & 64-bit processors.
- Variations also depend on clock speeds, generations, and processor types.

The Von Neumann Architecture



Uniprocessor Architecture

- A **uniprocessor system** has a single CPU (Central Processing Unit).
- All instructions are executed sequentially by that one processor.
- It follows the **Von Neumann Architecture**, consisting of:
  - **Memory** – stores data and programs.
  - **Processor (CPU)** – includes:
    - **Control Unit (CU)** – directs operations of the CPU.
    - **ALU (Arithmetic and Logic Unit)** – performs arithmetic and logical operations.
    - **Registers** – small, fast storage inside CPU for immediate data.
  - **Input/Output (I/O)** – communicates with external devices.
  - **System Interconnection (Bus)** – links CPU, memory, and I/O.

Uniprocessor vs Multiprocessor

- **Uniprocessor**: One CPU executes one instruction stream at a time.
- **Multiprocessor**: Two or more CPUs share memory and work together on tasks, providing higher performance and parallelism.

CPU Structure (Uniprocessor)

- **Control Unit** – coordinates instruction execution.
- **ALU** – processes data.
- **Registers** – temporary data storage inside CPU.
- **CPU Interconnection** – pathways linking CU, ALU, and registers.

Multicore Computer Structure

- **CPU**: Fetches and executes instructions.
- **Core**: Each independent processing unit within a CPU (acts like a mini-CPU).
- **Processor**: A chip that may contain multiple cores (called a multicore processor).
- Multicore systems enable **parallel execution** within one processor chip.

Cache Memory

- Cache is a **small, fast memory** located between the CPU and main memory.
- Purpose: **reduces memory access time** by storing frequently used instructions/data.
- Organised in multiple levels:
  a. L1 (fastest, smallest) – closest to the CPU core.
  b. L2 – larger but slower than L1.
  c. L3 (largest, slowest among caches) – shared among cores, closer to main memory.
- Works on the principle of **temporal and spatial locality** (recently/frequently accessed data likely to be reused).
- Improves performance by avoiding repeated slow main memory access.

Processor & Cache Placement

Motherboard: Contains CPU, main memory chips, I/O chips.
Processor chip: Contains multiple cores and shared cache (L3).
Core:

- Instruction logic, ALU, load/store logic.
- Private **L1 caches (instruction + data)**.
- Private or semi-private **L2 cache**.

Example: Server-Class Motherboard

- Shown: **Dual Quad-Core Xeon Processors** with **integrated memory controllers**.
- Up to **48GB DDR3 memory** supported.
- High-speed interconnects: PCI Express, SATA, USB, Ethernet, etc.
- Demonstrates how **multiple processors with multiple cores** interact with memory and I/O efficiently.

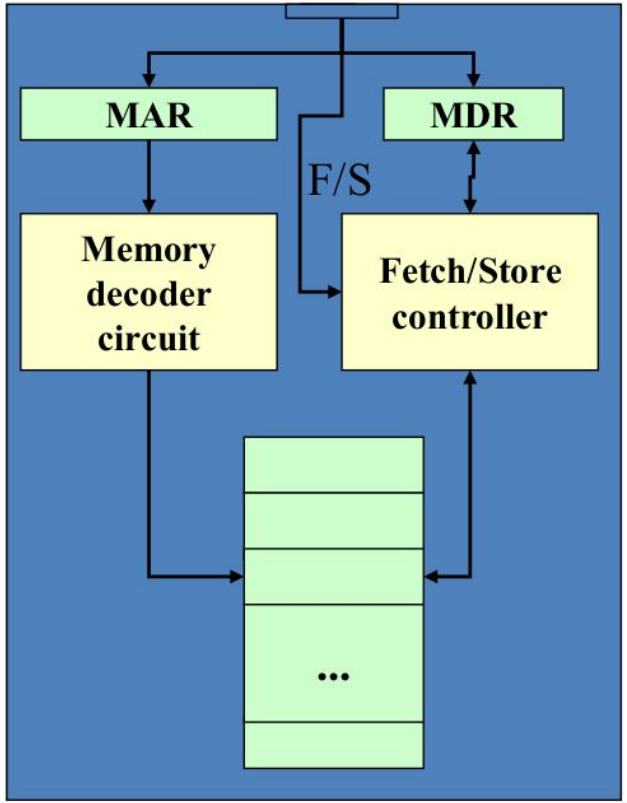
Memory Subsystem

Fetch(address)

1. CPU loads the address into the **Memory Address Register (MAR)**.
2. The **address decoder** inside memory identifies the correct memory cell.
3. The **contents** of that memory cell are copied into the **Memory Data Register (MDR)**.

Store(address, value)

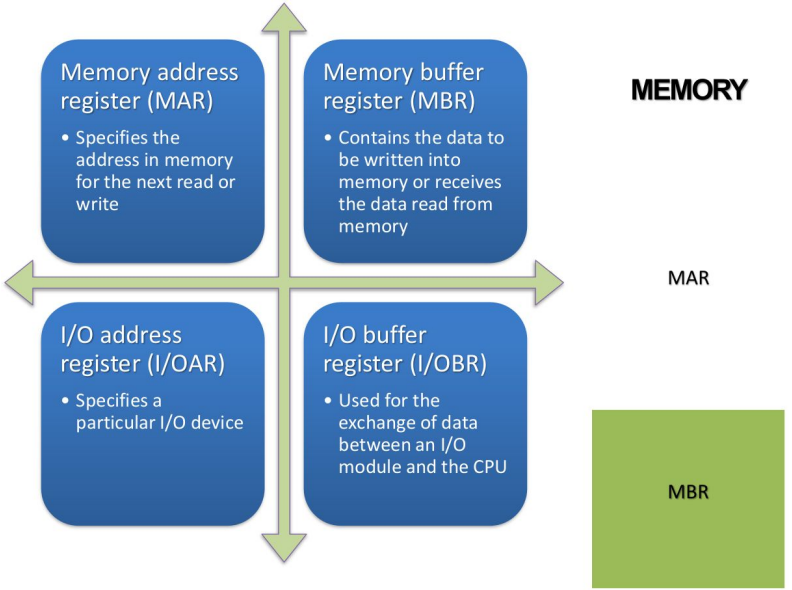
1. CPU loads the target address into the **MAR**.
2. CPU loads the value to be stored into the **MDR**.
3. The memory **address decoder** finds the right memory cell.
4. The **MDR value** is written into that memory location.



Memory Subsystem

Memory Registers

- **MAR (Memory Address Register)**
  Holds the address in memory where the next read or write will take place.
- **MBR (Memory Buffer Register)** (sometimes called MDR – Memory Data Register)
  Temporarily stores the data being transferred.
  - If reading: holds data fetched from memory.
  - If writing: holds data to be written into memory.



I/O Registers

- **I/O Address Register (I/OAR)**
  Identifies the specific I/O device involved in the operation.
- **I/O Buffer Register (IOBR)**
  Stores the data being transferred between CPU and the I/O device.

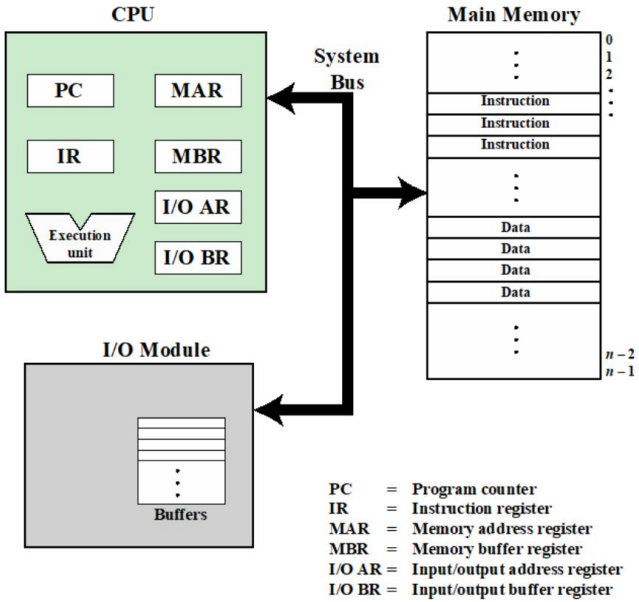
In short:

- **MAR & MBR** handle communication between CPU and **memory**.
- **I/OAR & IOBR** handle communication between CPU and **I/O devices**.

Memory Subsystem

CPU Registers

- **PC (Program Counter):** Holds the address of the next instruction to be executed.
- **IR (Instruction Register):** Holds the currently executing instruction.
- **MAR (Memory Address Register):** Holds the address in memory where the CPU wants to read/write.
- **MBR (Memory Buffer Register):** Temporarily stores data being transferred to/from memory.
- **I/O AR (I/O Address Register):** Holds the address of the I/O device involved.
- **I/O BR (I/O Buffer Register):** Holds the actual data to be transferred between CPU and I/O device.



Main Memory

- Stores both **instructions** (program code) and **data**.
- Instructions are fetched into **IR**, while data moves through **MBR**.

Memory Subsystem

I/O Module

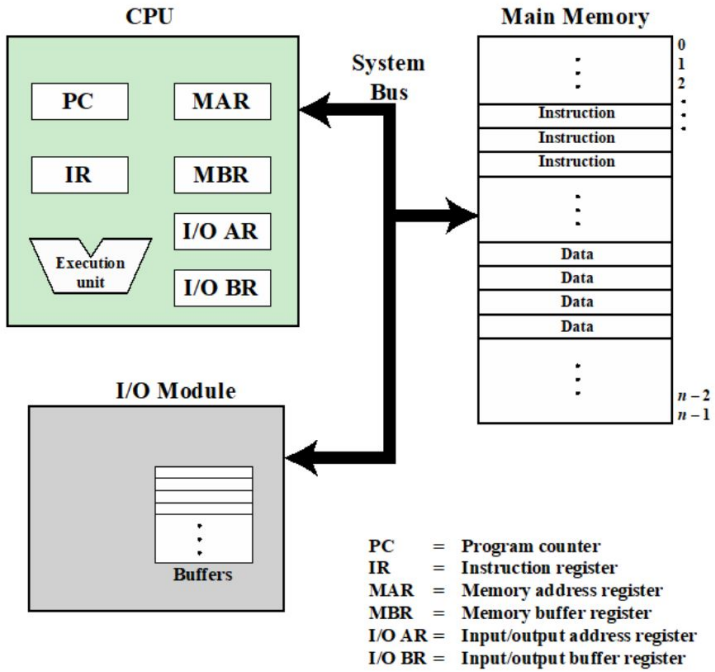
- Manages communication with external devices.
- Has its own **buffers** to temporarily hold data during transfers.

System Bus

- Connects **CPU, Main Memory, and I/O Module**.
- Carries **addresses (via MAR, I/O AR)**, **data (via MBR, I/O BR)**, and **control signals**.

Operation Flow

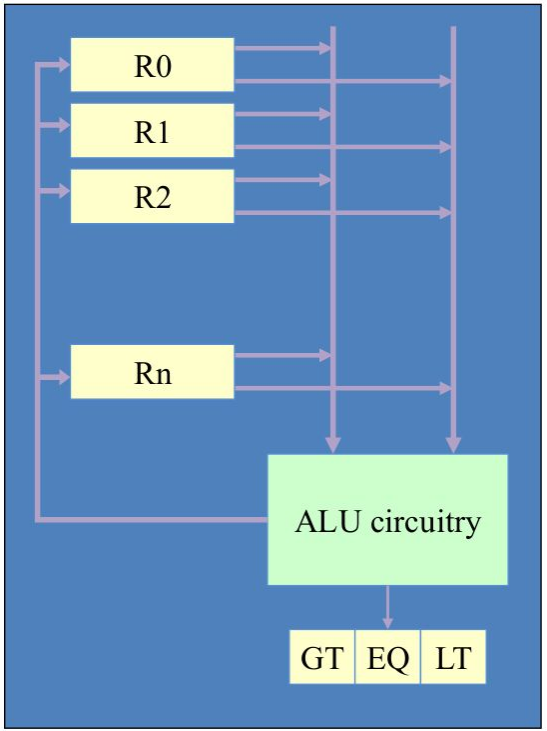
1. **Instruction Fetch:**
   - PC → MAR → Memory → MBR → IR
     (Instruction loaded into CPU for execution).
2. **Data Access:**
   - CPU places address in MAR, fetches or stores data through MBR.
3. **I/O Operation:**
   - CPU specifies device address in I/O AR, exchanges data through I/O BR.



Structure of the ALU

Components of ALU

1. **Registers**
   - Very fast local memory cells inside the CPU.
   - Store operands (inputs to operations) and intermediate results.
   - Includes **CCR (Condition Code Register):**
     - A special-purpose register.
     - Stores results of comparison operations (<, =, >).
     - Helps in decision-making for conditional instructions (like branching).
2. **ALU Circuitry**
   - The actual electronic circuits that perform **arithmetic (addition, subtraction, multiplication, division)** and **logic operations (AND, OR, NOT, XOR, comparisons)**.
3. **Bus**
   - A **data path** that connects registers to the ALU circuitry.
   - Moves operands in, and sends results back to registers or memory.



Registers **R0, R1, R2 … Rn** feed data into the ALU.

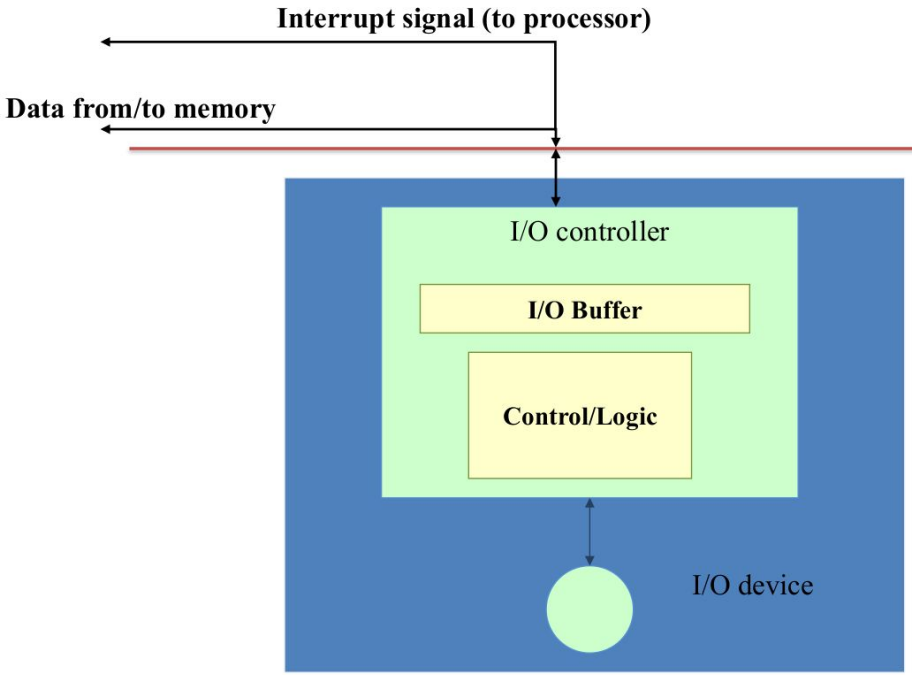
The **ALU circuitry** processes the data.

Results are passed to CCR (storing conditions like GT = greater than, EQ = equal, LT = less than).

Structure of the I/O Subsystem

Main Components

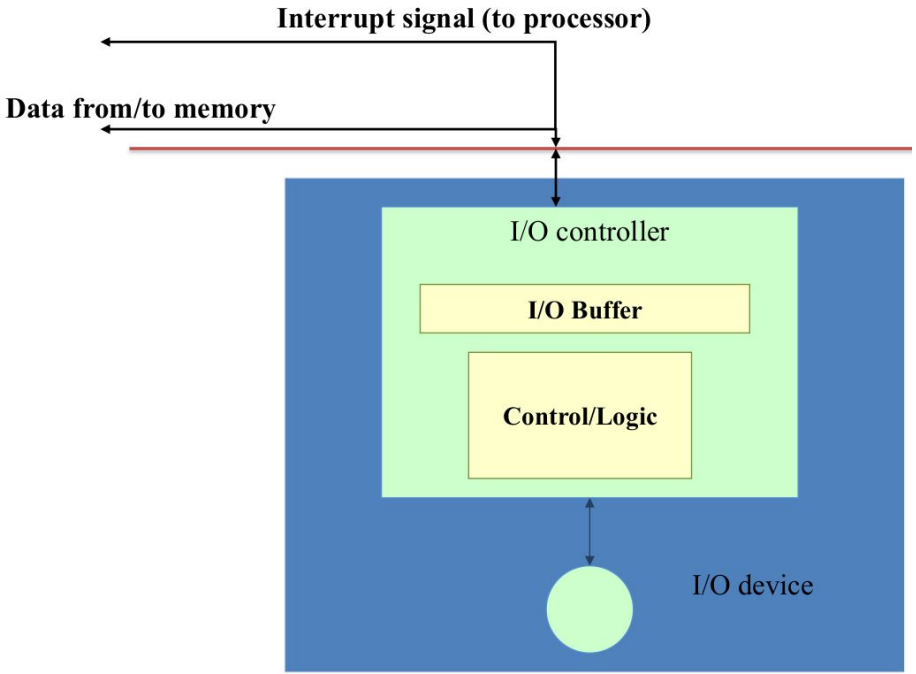
1. **I/O Device**
   - The actual hardware (e.g., keyboard, mouse, disk, printer, network card).
2. **I/O Controller**
   - Acts as a middleman between the CPU and I/O device.
   - Includes:
     - **I/O Buffer** – temporarily stores data being transferred.
     - **Control/Logic** – manages communication, signals, and device control.
3. **Data Path (to/from memory)**
   - Transfers data between memory and the I/O buffer.
4. **Interrupt Signal (to CPU)**
   - Notifies the CPU when the I/O operation is complete or if the device needs attention.
   - Prevents the CPU from continuously polling the device (saves processing time).



Structure of the I/O Subsystem

How it Works

1. CPU issues a command to the I/O controller.
2. The **Control/Logic** unit manages the device operation.
3. Data is placed in the **I/O Buffer** (either from device → memory or memory → device).
4. Once the operation completes, the controller sends an **Interrupt Signal** to the CPU.
5. CPU stops its current task, handles the interrupt, and resumes processing.

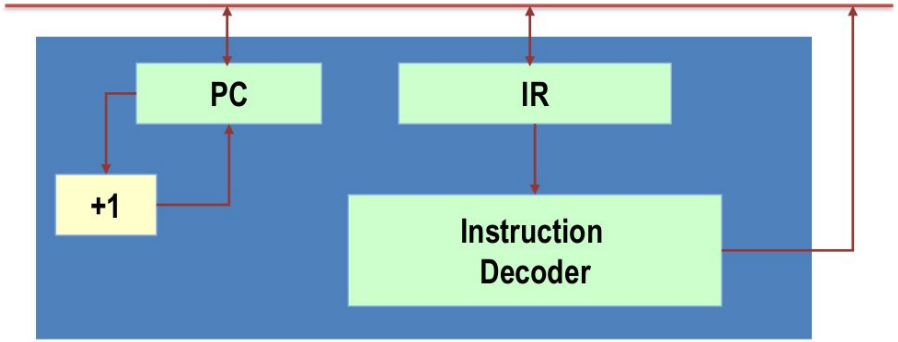


Structure of the Control Unit

The **Control Unit** directs CPU operations:

- **PC (Program Counter):** Holds address of the next instruction.
- **IR (Instruction Register):** Stores the fetched instruction.
- **Instruction Decoder:** Interprets instruction and activates circuits.
- **+1 Incrementer:** Advances PC for the next instruction.

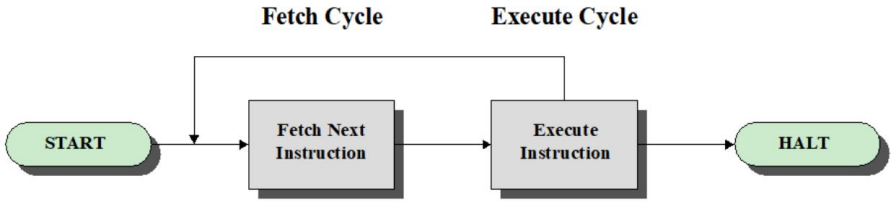
Cycle: Fetch instruction → Store in IR → Decode → Execute → PC +1.



Basic Instruction Cycle

Instruction Cycle

The instruction cycle is the sequence of steps the CPU follows to execute program instructions.



1. Fetch Cycle

- The **Program Counter (PC)** holds the address of the next instruction.
- CPU fetches the instruction from memory using this address.
- The instruction is placed into the **Instruction Register (IR)**.
- The **PC increments** to point to the next instruction.

In short: **PC → Fetch instruction → Load into IR → Increment PC → Decode next**

Basic Instruction Cycle

2. Decode Cycle

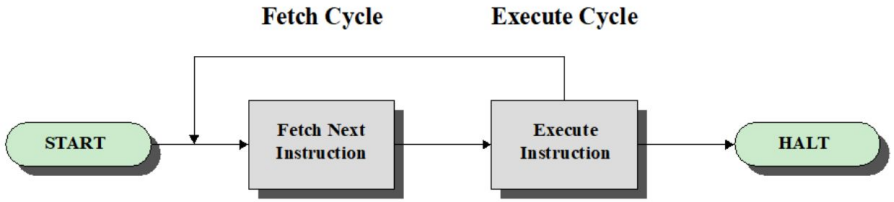
- The **Control Unit** decodes the instruction in IR.
- Determines the operation and identifies required operands.

3. Execute Cycle

- The CPU executes the decoded instruction (ALU operation, memory access, or I/O).
- Results are stored in registers or memory.

4. Repeat / Halt

- If instructions remain, cycle returns to **Fetch**.
- If program ends, CPU enters the **HALT state**.



Action Categories

Processor–Memory

- Transfer data between processor and memory.

Processor–I/O

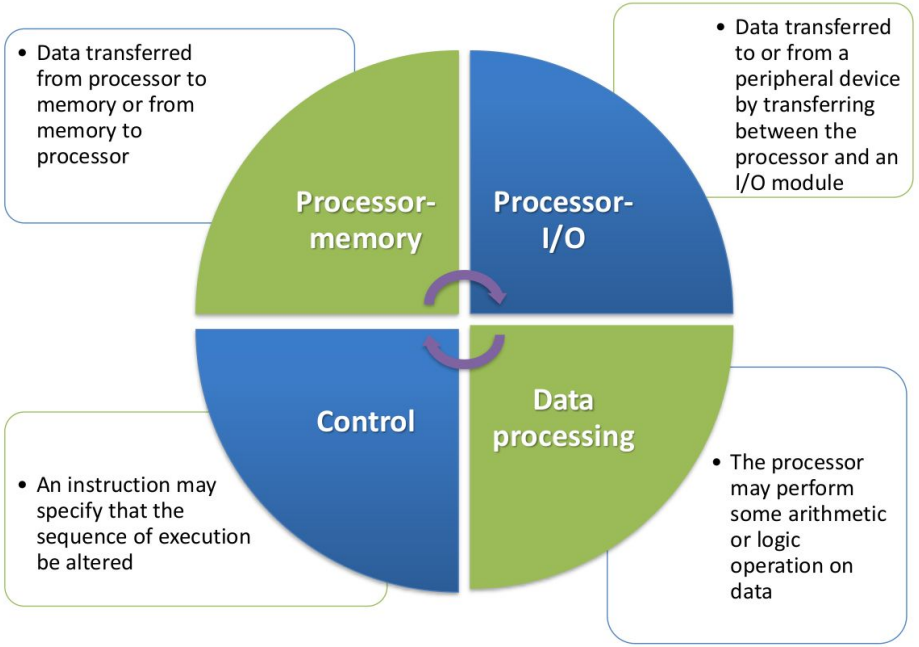
- Transfer data between processor and peripheral devices via I/O modules.

Data Processing

- Perform arithmetic or logical operations on data.

Control

- Alter the sequence of instruction execution (e.g., jumps, branches).



Characteristics of Hypothetical Machine

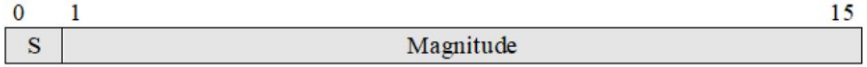
Instruction Format

- 16-bit word.
- First 4 bits: Opcode (operation code).
- Remaining 12 bits: Address field.

(a) Instruction format

Integer Format

- 16-bit word.
- First bit: Sign (S).
- Remaining 15 bits: Magnitude.



(b) Integer format

Key CPU Registers

- **PC (Program Counter):** Holds the address of the next instruction.
- **IR (Instruction Register):** Holds the current instruction.
- **AC (Accumulator):** Temporary storage for arithmetic/logic operations.

Sample Opcodes

- **`0001`** → Load AC from Memory.
- **`0010`** → Store AC to Memory.
- **`0101`** → Add to AC from Memory.

Instruction Cycle State Diagram

Instruction Cycle Overview

The **instruction cycle** is the basic sequence of steps the CPU follows to execute an instruction.
Four key phases:

1. **Fetch** – Get the instruction from memory
2. **Decode** – Interpret what the instruction means
3. **Execute** – Perform the required operation
4. **Store/Write-back** – Save the result

Step 1 – Instruction Fetch

- CPU fetches instruction from **memory** using the **Program Counter (PC)**.
- PC holds the address of the next instruction.
- After fetch, PC is updated to point to the next instruction.

Step 2 – Instruction Decode

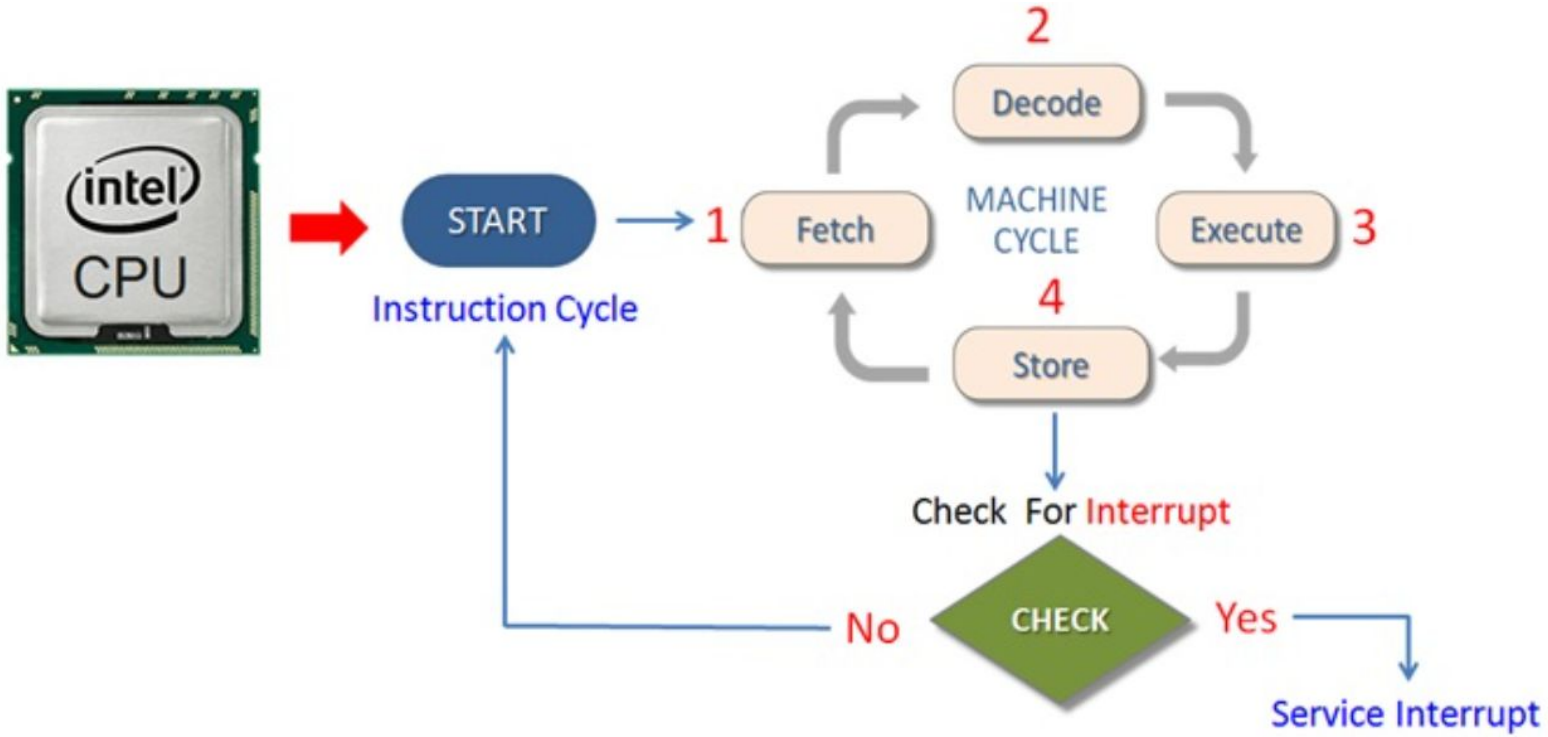
- The fetched instruction is sent to the **Instruction Decoder**.
- CPU identifies:
  - The **operation** (e.g., ADD, LOAD)
  - The **operands** (data or addresses needed).

Step 3 – Operand Handling

- If the instruction requires **operands** (data to work on):
  - CPU calculates operand addresses.
  - Operands are fetched from memory or registers.

Step 5 – Store (Write Back)

- The result of the operation is written:
  - Back to memory, or
  - Into a register.


intel
CPU
START
Instruction Cycle
1
Fetch
2
Decode
MACHINE
CYCLE
3
Execute
4
Store
Check For Interrupt
CHECK
No
Yes
Service Interrupt

Transfer of Control via Interrupts

What Happens When an Interrupt Arrives?

- CPU is busy running a **user program**.
- Suddenly, an **interrupt signal** comes from a device (e.g. disk, network, timer).
- CPU must **pause user work** and **handle the interrupt** immediately.

Saving the Current Context

- CPU automatically:
  - Finishes the current instruction.
  - Pushes critical info (program counter, flags) onto the kernel stack.
- This ensures the CPU can **resume the user program** later exactly where it left off.

Switching into Kernel Mode

- Control jumps to a predefined **interrupt handler address** stored in the Interrupt Descriptor Table (IDT).
- CPU switches from **user mode** → **kernel mode**.
- Kernel mode has full access to hardware and memory.

Running the Interrupt Handler

- The handler is a small piece of code specific to the device.
- Tasks it may perform:
  - Acknowledge the device (tell it we got the signal).
  - Collect data from the device buffer.
  - Schedule further work (e.g. waking up a waiting process).

Processor Clock & Instruction Cycle

Processor Clock

- CPU operations are synchronised by a periodic clock signal.
- Clock source: **oscillator frequency (Fosc)**.
- Clock frequency determines how many ticks occur per second.

Instruction Cycle (Machine Cycle)

- An instruction executes in defined phases:
  1. **Fetch** – retrieve instruction from memory.
  2. **Decode** – interpret opcode and operands.
  3. **Execute** – perform the operation.
  4. **Store (Write-back)** – save result to register or memory.
- Together, these phases form the **instruction cycle**.

Relation Between Clock and Instruction Cycle

- **Fosc** = oscillator frequency.
- **Fcy** = instruction cycle frequency. The rate at which complete instruction cycles (fetch–decode–execute–store) are executed in a simple, sequential CPU.
- **Tcy** = instruction cycle period = 1/Fcy. The time required to complete one full instruction cycle.
- Typically, **Fcy = Fosc ÷ N**, where *N* is an implementation-dependent divider.

Timing Characteristics

- One instruction cycle requires multiple clock ticks (depending on architecture).
- Simple instructions may require 1 cycle.
- Memory access or complex arithmetic may require multiple cycles.
- Performance = Fcy × average instructions per cycle (IPC).

OS Role in Managing Hardware & Services

Core Role of OS

- Acts as a bridge between **user programs** and **hardware**.
- Provides **system calls** for controlled access to resources.
- Ensures safe, efficient, and fair use of hardware.

Major OS Services

- **User Interface**: CLI, GUI, or batch.
- **Program Execution**: Load, run, and terminate programs.
- **I/O Operations**: Manage interaction with devices.
- **File System Management**: Create, delete, read, write, search, and set permissions.
- **Communication**: Enable data exchange between processes (shared memory or message passing).

Major OS Services

- **Error Detection**: Monitor CPU, memory, I/O, and user programs for faults; provide consistent recovery.
- **Resource Allocation**: Share CPU cycles, memory, storage, and devices among processes and users.
- **Accounting**: Track resource usage for analysis.
- **Protection & Security**:
  - *Protection*: Prevent processes from interfering with each other.
  - *Security*: Defend system from unauthorised access or external threats.

Designing for Performance

- The cost of computer hardware keeps dropping, while its performance continues to rise.
- A modern laptop now delivers the computing power that once required a mainframe 10–15 years ago.
- Processors are inexpensive but powerful, making it possible to run demanding applications such as image processing, 3D rendering, speech recognition, video conferencing, multimedia authoring, and simulations.
- Businesses increasingly depend on high-performance servers to handle databases and large client/server workloads, replacing the mainframes of the past.
- Cloud providers operate massive server farms to deliver high-volume, high-transaction services to a wide range of clients.

Microprocessor Speed – Techniques

Pipelining – Break instructions into stages and overlap them, so multiple instructions are in progress at once, increasing throughput.

Branch Prediction – The processor predicts the outcome of branches (like if/else paths) to keep the pipeline busy and avoid delays.

Superscalar Execution – The CPU can issue and execute multiple instructions in the same clock cycle using parallel pipelines.

Microprocessor Speed – Techniques

Data Flow Analysis – The processor examines dependencies between instructions and reorders execution for maximum efficiency.

Modern CPUs have hardware units (like *out-of-order execution engines* and *dependency checkers*) that analyse instruction streams.

They detect which instructions are independent and can be executed in parallel or reordered without breaking program correctness.

Example: If instruction B doesn't depend on the result of instruction A, the CPU may execute them in parallel or even finish B before A.

OpenCL, CUDA, etc. (software frameworks)

- These provide programming models for parallel workloads across CPUs and GPUs.
- They **schedule tasks** and map them to hardware threads, but they don't do low-level instruction dependency analysis — that's the CPU/GPU's job.

Speed Difference (DDR5 vs L3 Cache)

RAM:
– Speed: ~4800–8800 MT/s
– Latency: ~50–100 ns
– Bandwidth: 38–76 GB/s
– Location: DIMM slots (off-chip)
– Purpose: large main memory

L3 Cache:
– Speed: several TB/s
– Latency: ~10–20 ns
– Bandwidth: 100s of GB/s – TB/s
– Location: on CPU die
– Purpose: fast intermediate storage

Performance Balance

Different components of a computer run at very different speeds — processors are fast, while memory and I/O are much slower.

To keep the system efficient, the design must balance these mismatches.

Common strategies include:

- Using **wider DRAM buses** to move more data per transfer.
- Building **multi-level cache hierarchies** to reduce slow memory accesses.
- Adding **on-chip buffering** and smarter DRAM interfaces to improve throughput.
- Employing **high-speed interconnects** (like advanced buses and hierarchical links) to handle large data flows between components.

Improvements in Chip Architecture

Faster processors – Shrinking transistor sizes allows higher clock speeds and shorter signal delays.

Larger caches – Placing bigger caches directly on the chip gives much faster access to frequently used data.

Smarter architectures – Designs that exploit parallelism (e.g. multiple execution units, out-of-order execution) increase the number of instructions completed per cycle.

Problems with Clock Speed & Density

Power and Heat – Packing more logic and pushing higher clock speeds greatly increases power consumption and heat output.

RC Delay – As wires get smaller, their resistance and capacitance rise, which slows down signal propagation across the chip.

Memory Latency – DRAM has not kept pace with CPU speed, so memory access becomes a major bottleneck.

Processor Trends (1970–2010)

- **Transistors** – Grew exponentially for decades, consistent with Moore's Law.
- **Clock Frequency** – Increased rapidly until about 2005, then stalled due to heat and power limits.
- **Power** – Rose steadily and became a critical design constraint.
- **Cores** – Remained mostly single-core until mid-2000s, then shifted to multi-core architectures to keep performance scaling.

Multi-Core

Instead of pushing a single processor to extreme speeds (which causes heat and power issues), chip designers place multiple simpler processors (cores) on the same chip. This boosts performance without raising clock rate. Larger caches are used to feed these cores, and as caches grew, it made sense to add multiple levels (L1, L2, L3) for efficiency.

14 Cores and 20 Logical Processors

Some CPUs use **Hyper-Threading**, where one physical core can handle two instruction streams (threads). But not all cores need it — some may run single-threaded to save power.
Modern CPUs often use **hybrid cores**:

- **Performance (P) Cores** – strong cores that support Hyper-Threading.
- **Efficient (E) Cores** – smaller, power-saving cores without Hyper-Threading.

Example: 6 P-Cores (12 threads) + 8 E-Cores (8 threads) = 20 logical threads.

Base Frequency of the Processor

The **base frequency** is the minimum guaranteed speed of the CPU under normal conditions. But the CPU isn't stuck there — it can **dynamically boost** its speed (e.g., Intel Turbo Boost) when extra performance is needed, depending on power, temperature, and workload.

Turbo Boost Logic

Turbo Boost temporarily increases CPU speed beyond the base frequency. It activates when demanding tasks need more performance, but it balances this with heat and power limits. Essentially: more speed when you need it, efficiency when you don't.

Why Base Frequency Increases

Turbo Boost only kicks in under certain conditions:

- **Workload demand** – heavy apps like gaming or video editing.
- **Thermal headroom** – CPU must be cool enough.
- **Power limits** – must stay within safe energy use.
- **Active cores** – fewer active cores can boost higher; all cores active means lower boost per core.

How Variable Frequency Works

The CPU continuously balances performance and safety:

- **Voltage & Frequency Scaling** – raise clock + voltage when needed.
- **Dynamic Adjustments** – monitor workload, temperature, and power in real time.
- **Thermal Throttling** – if it overheats, reduce speed to protect the chip.

Example of Frequency Variation

- Light tasks (e.g., browsing) → CPU stays at base speed (e.g., 2.3 GHz).
- Heavy tasks (e.g., video editing) → CPU boosts higher (e.g., 4.0 GHz).
- Overheating risk → CPU reduces speed automatically to avoid damage.

Threads in CPU

- A **thread** is the smallest sequence of instructions the CPU can run independently.
- A **process** (application) may have many threads, each handling different tasks at the same time.
- Threads in a process **share memory space** but can run separate parts of the program in parallel.

Why Thread Count Varies

- Threads are **created** when apps or background tasks start.
- Threads are **terminated** when apps close or finish tasks.
- Count changes dynamically depending on system activity.

Example: Web Browser

- **UI rendering thread**
- **Page loading thread**
- **User input thread**
- **Network request thread**

Handles in OS

- A **handle** is a reference to a system resource (file, window, socket, registry key, etc.).
- Processes request handles whenever they interact with system resources.
- OS assigns handles to track and manage resource usage.

Why Handle Count Varies

- **Increases** when new files, network connections, or GUI objects open.
- **Decreases** when resources are closed.
- **Leaked handles** (due to buggy software) can keep counts high even after use.

Many Integrated Core (MIC)

- Built with **hundreds of general-purpose cores** on a single chip.
- The idea was to get a big leap in performance by running many tasks in parallel.
- Homogeneous cores → easier to program than GPUs, but still challenging at large scale.

Graphics Processing Unit (GPU)

GPUs began as **special-purpose chips for graphics**.

- The cores are specialised for **parallel number-crunching**, perfect for rendering pixels.
- Found on graphics cards, GPUs power **3D games, video rendering, and animations**.
- Today, GPUs are used far beyond graphics — they're the engines behind **AI, simulations, and scientific computing**, thanks to massive parallelism.

Amdahl's Law

Programs have two parts:

- A **serial portion** that must run on a single processor.
- A **parallel portion** that can be divided among multiple processors.

Adding processors only helps the parallel portion. The serial portion takes the same time no matter how many processors are available.

Overall speedup is limited by the serial portion. If even a small fraction of the program is serial, it puts a hard cap on the maximum possible improvement.

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

f = fraction of program that can be parallelised
N = number of processors
(1−f) = serial part, unchanged by parallelism

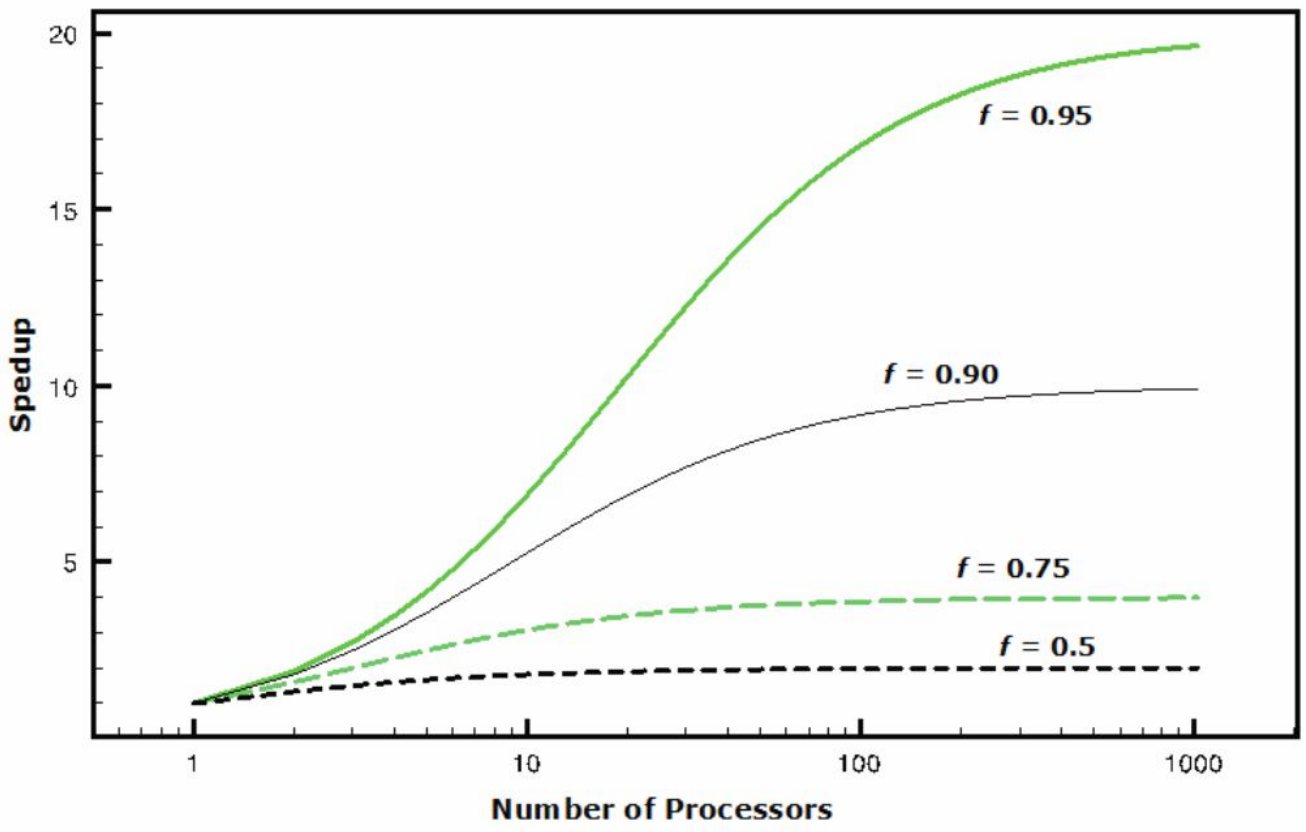
Problem

A program has been updated with 80% parallelizable code and 20% sequential code. The parallelization is done using 6 processors. What is the speedup in percentage?

Parallel fraction (f): 80% = 0.8
Sequential fraction (1 – f): 20% = 0.2
Processors (N): 6
Speedup: ~3.0

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

Amdahl's Law for Multi Processor



Evaluating Processor H/W

Evaluating Processor Hardware

When evaluating processor hardware for new systems, we need to look beyond just raw speed. Performance is the most important parameter, but factors such as cost, physical size, security, reliability, and power consumption also play a critical role. For example, a high-performance processor might be excellent for servers, but if it consumes too much power or costs too much, it won't be practical for mobile devices.

Application Performance

The actual performance of applications depends on multiple factors, not just the processor's clock speed. It is influenced by the instruction set of the processor, the programming language used, and how efficiently the compiler translates code. Even the programmer's skill in structuring the application matters. A powerful processor will underperform if the software running on it is inefficiently designed or compiled.

Review of Processor Operations

Every operation a processor performs — fetching an instruction, decoding it, loading data, and executing arithmetic or logic — is driven by a system clock. The clock generates pulses that act as the rhythm for the processor. Each operation starts with a clock pulse, and the overall speed of the processor is tied to how frequently these pulses occur, measured in Hertz (Hz).

Clock Cycles and Speed

For example, a 1 GHz processor generates one billion pulses per second. Each pulse represents a clock cycle, and the time between pulses is called the cycle time. The higher the clock speed, the more operations the processor can start each second. However, performance isn’t only about clock speed — memory access delays, pipeline efficiency, and instruction parallelism also determine how much work gets done per cycle.

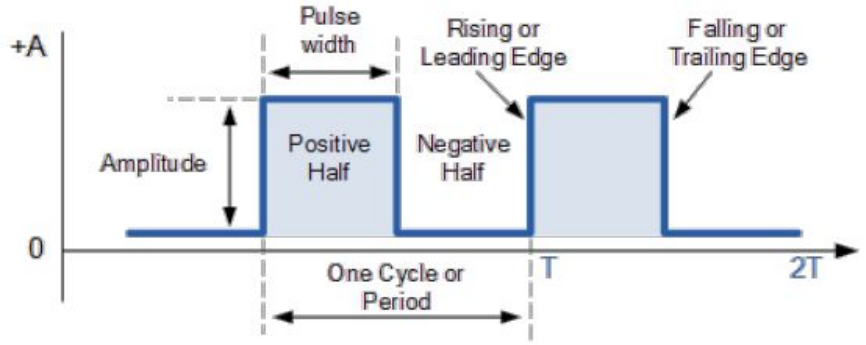
Clock Signal: The Processor's Timing Pulse

Amplitude (+A to 0):
This is the voltage difference of the clock signal. +A is the "high" state, and 0 is the "low" state. The processor recognises these two states as the basis for timing.

Positive Half:
The duration when the clock signal is high (+A). This is often when certain operations (like latching data) are triggered.

Negative Half:
The duration when the clock signal is low (0). Some circuits are designed to react during this phase.

Pulse Width:
The time the signal stays in the high state during one cycle. It is a portion of the total cycle.

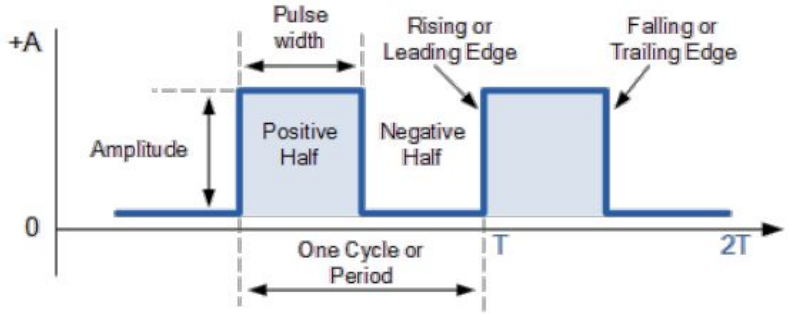


Clock Signal: The Processor's Timing Pulse

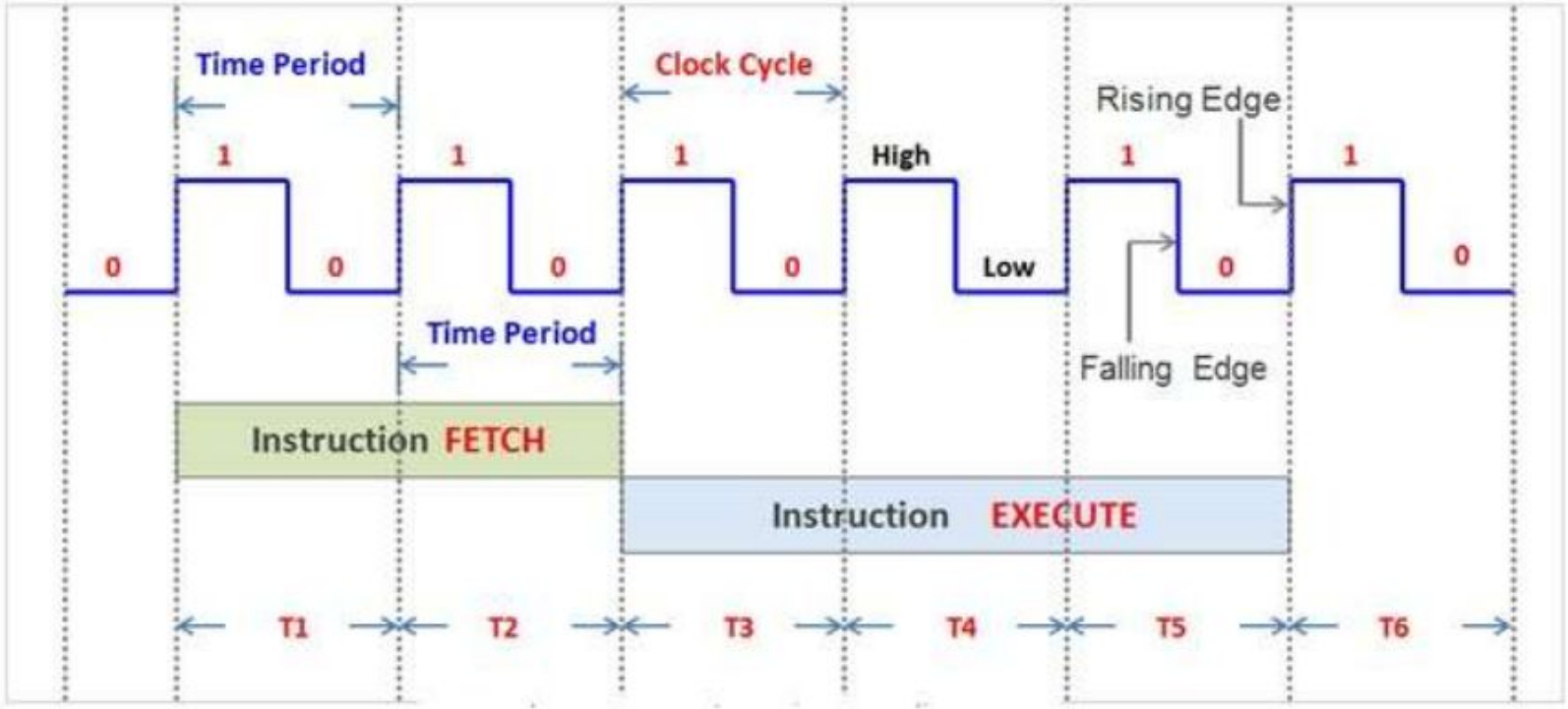
Rising Edge (Leading Edge):
The transition point where the signal goes from low (0) to high (+A). Many digital circuits, including processors, start new operations on this edge.

Falling Edge (Trailing Edge):
The transition where the signal drops from high (+A) back to low (0). Other circuits may use this edge to trigger their operations.

One Cycle (Period, T):
A full cycle is the combination of one positive half and one negative half. It is the repeating unit of the clock signal.



- Measured in **time (seconds)**.
- The **frequency** is the inverse of this period (f = 1/T). For example, if T = 1 nanosecond, f = 1 GHz.


Time Period
Clock Cycle
Rising Edge
1
1
1
High
1
1
0
0
0
0
Low
0
0
Time Period
Falling Edge
Instruction FETCH
Instruction EXECUTE
T1
T2
T3
T4
T5
T6

Clock Cycle and Instruction Execution

Clock cycle and processor operations: Each clock cycle (one high + one low) drives processor activity.

Time slots (T1–T6): The diagram breaks down time into discrete cycles labeled T1, T2, …, T6.

Instruction phases:

- During **T1–T2**, the processor performs **Instruction Fetch** (retrieving the instruction from memory).
- During **T3–T4**, it performs **Instruction Execute** (carrying out the operation).

Synchronization: Both fetch and execute steps are locked to the clock cycle — fetch starts and ends on clock edges, then execute begins on the next set of edges.

Understanding Clock Speed and Cycle Time

A **1-GHz processor** means the chip gets **1 billion clock pulses every second**.

- The number of pulses per second is called the **clock speed**.
- Each single pulse is a **clock cycle**.
- The tiny gap between two pulses is the **cycle time**.

Clock Cycles and Instruction Execution

Instructions take multiple cycles
A single clock tick doesn't always finish an instruction. Most instructions—like add, load, or multiply—need several ticks to go through fetch, decode, execute, and write-back stages.

Not all instructions are equal
Some are simple (like adding two numbers) and finish in just a few cycles. Others, such as memory access or division, may take many cycles.

Pipelining changes the game
With pipelining, multiple instructions overlap in execution. While one instruction is being fetched, another is being decoded, and yet another is being executed. This means the processor can complete more instructions per unit time than clock speed alone would suggest.

Why raw clock speed is misleading
Two processors at the same clock speed might perform very differently depending on how many cycles their instructions need, how well pipelining is used, and how efficient the architecture is.

Cycles Per Instruction (CPI)

Instruction count (I_c) → total number of instructions executed by the program.

CPI_i → cycles per instruction for a specific instruction type *i*.

I_i → number of instructions of type *i*.

CPI (average) → weighted average of cycles per instruction across the program.

Cycle time (τ) → duration of one clock cycle (*1/f*).

Execution time formula:
Execution Time=(Instruction Count)×(Average CPI)×(Cycle Time)

Program runtime is influenced by (1) number of instructions, (2) how many cycles instructions need, and (3) clock speed.

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c}$$

$$T = I_c \times CPI \times \tau$$

MIPS (Millions of Instructions Per Second)

Definition: MIPS measures how many million instructions a processor executes per second.

Variables:

- I_c → number of instructions executed.
- T → total execution time.
- f → processor clock frequency.
- CPI → average cycles per instruction.

$$MIPS = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Formula meaning:

- Higher **clock speed (f)** increases MIPS.
- Lower **CPI** improves MIPS (fewer cycles per instruction).

Case Study

The difference in RAM speed (4400 MT/s vs. 6000 MT/s). Whether changing the RAM from 4400 MT/s to 6000 MT/s will do?

- Simply upgrading RAM from **4400 MT/s to 6000 MT/s** may **not work** unless both the **motherboard** and the **processor** support the higher speed.
- **Motherboard dependency:** The memory controller on the motherboard (chipset + BIOS) must allow 6000 MT/s. If it only supports up to 4400 MT/s, faster RAM will downclock to 4400 MT/s.
- **Processor dependency:** Modern CPUs have integrated memory controllers. If the CPU only supports up to 4800 or 5200 MT/s, installing 6000 MT/s RAM won't run at full speed.
- Changing RAM alone is not enough. You may need to change the **motherboard, processor, or both**, depending on their supported memory speeds.

Case Study

How does the difference in RAM speed (4400 MT/s vs.6000 MT/s) influence the overall system performance in tasks such as gaming, content creation, and AI/ML workloads, given that all other hardware components are identical?

Impact of RAM Speed on System Performance

- **Gaming:**
  - Most modern games are more dependent on GPU and CPU performance.
  - Faster RAM (6000 MT/s vs. 4400 MT/s) can improve *minimum frame rates* and reduce stuttering, but average FPS gains are often modest (~3–10%).
- **Content Creation (e.g., video editing, 3D rendering):**
  - Applications that handle large datasets (like 4K/8K video timelines or huge texture assets) benefit more from higher memory bandwidth.
  - You may see noticeable performance improvements in *export times and real-time previews*.
- **AI/ML Workloads:**
  - These workloads often involve large matrix multiplications and memory-bound operations.
  - Higher RAM speed can give significant boosts in *training throughput* and *data preprocessing speed*.
- **Overall:**
  - Faster RAM gives diminishing returns if the workload is CPU/GPU bound.
  - Workloads that are memory-intensive benefit more.
  - Performance uplift varies from *minor (gaming)* to *moderate/significant (AI/ML, heavy content creation)*.

Case Study

An organization intends to procure a high-performance server to meet its extensive processing and GPU requirements (currently around 500 GB). Also, a separate redundance storage solution is required. The server will be utilized to create virtual machines (VMs) that will be allocated to various associated organizations, with the flexibility to reallocate them based on evolving configuration needs. Additionally, the organization seeks a scalable solution that allows for future infrastructure expansion while maintaining the core system configuration. What would be the most suitable server solution to meet these requirements?

Key Requirements in the Problem

High Processing and GPU Power

- The organization has heavy computational needs (around 500 GB of GPU memory capacity).
- This suggests workloads like **AI/ML, big data processing, or GPU-intensive simulations**.

Separate Redundant Storage

- They don’t just want performance, but also **reliability**.
- Redundant storage means a system like **RAID arrays, SAN (Storage Area Network), or NAS with redundancy** to avoid data loss.

Virtual Machines (VMs)

- The server must support **virtualization**.
- VMs will be allocated to different organizations (multi-tenancy).
- There should be flexibility to **reallocate VMs dynamically** as needs change.

Scalability

- The solution should **scale in the future** without redesigning the entire infrastructure.
- Core configuration (CPU, GPU, memory structure) should remain intact while allowing easy expansion.